Siddhartha Rao

Programowanie w Twoim zasięgu!

Wydanie VII

C++

Dla **każdego**

SAMS

Helion

Tytuł oryginału: Sams Teach Yourself C++ in One Hour a Day (7th Edition)
Tłumaczenie: Robert Górczyński
ISBN: 978-83-246-8080-1

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/cppit7.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cppit7_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Dedykacja

Książkę dedykuję moim kochanym rodzicom i cudownej siostrze za to, że byli przy mnie, kiedy ich najbardziej potrzebowałem.

Podziękowania

Jestem ogromnie zobowiązany moim przyjaciołom, którzy pomagali mi w pracy, kiedy byłem mocno zajęty pisaniem tej książki. Ponadto jestem wdzięczny zespołowi redakcyjnemu za pełne profesjonalizmu zaangażowanie prowadzące do wydania tej książki.

Spis treści

O autorze

Siddhartha Rao jest technologiem w SAP AG, największej na świecie firmie dostarczającej oprogramowanie przemysłowe. Pracuje jako szef SAP Product Security India; do jego podstawowych obowiązków należy zatrudnianie utalentowanych osób zajmujących się zapewnieniem bezpieczeństwa produktom, a także definiowanie najlepszych praktyk programistycznych, dzięki którym oprogramowanie SAP pozostaje globalnie konkurencyjne. Sid ma tytuł Most Valuable Professional for Visual Studio-Visual C++ i jest przekonany, że standard C++11 pomaga w tworzeniu szybszych, prostszych i znacznie efektywniejszych aplikacji C++.

Siddhartha kocha podróże i odkrywanie nowych kultur, gdy tylko ma taką możliwość. Pewna część tej książki powstała nad brzegiem Oceanu Atlantyckiego, w wiosce o nazwie Plogoff w Brytanii (Francja) — a to jeden z czterech krajów, w których powstawała. Sid czeka na uwagi dotyczące tej książki.

Wstęp

Rok 2011 był szczególny dla języka C++. Przyjęcie nowego standardu C++11 daje programistom możliwość tworzenia lepszego kodu przy użyciu nowych słów kluczowych i konstrukcji zwiększających efektywność pracy. Drogi Czytelniku, czytając tę książkę, poznasz standard C++11. Pozycja została starannie podzielona na lekcje prezentujące z praktycznego punktu widzenia podstawy programowania zorientowanego obiektowo. Jeśli posiadane umiejętności na to pozwolą, będziesz mógł opanować standard C++11, poświęcając zaledwie godzinę na lekcję.

Nauka C++ w taki sposób jest najlepszym rozwiązaniem. Umożliwi Ci wypróbowanie licznych przykładowych fragmentów kodu przedstawionych w książce i ułatwi rozwinięcie posiadanych umiejętności w zakresie programowania. Zaprezentowane fragmenty kodu zostały przetestowane w najnowszych dostępnych (w trakcie powstawania książki) wersjach kompilatorów, m.in. Microsoft Visual C++ 2012 i GNU C++ 4.6, które oferują obsługę nowych funkcji wprowadzonych w standardzie C++11.

Kto powinien przeczytać tę książkę?

Na początku książki znajdziesz podstawy dotyczące języka C++. Wcześniejsze doświadczenie w programowaniu na pewno będzie pomocne, ale nie jest wymagane, wystarczy jedynie chęć poznania języka i ciekawość. Z książki możesz również skorzystać, jeśli już znasz C++ i chcesz poznać zmiany wprowadzone w standardzie C++11. Jeżeli jesteś profesjonalnym programistą, w części III, zatytułowanej „Poznajemy standardową bibliotekę wzorców (STL)”, znajdziesz informacje, które pomogą Ci w tworzeniu lepszych i praktyczniejszych aplikacji C++11.

W jaki sposób zorganizowana jest ta książka?

Od Twojej biegłości w posługiwaniu się językiem C++ będzie zależeć wybór części, od której zaczniesz lekturę. Książka została podzielona na pięć części.

- Część I, „Podstawy", pozwoli Ci na rozpoczęcie tworzenia prostych aplikacji C++. W niej zostaną przedstawione słowa kluczowe najczęściej spotykane w kodzie C++, które zapewniają zachowanie bezpieczeństwa typów zmiennych.
- Część II, „Podstawy programowania zorientowanego obiektowo w C++", to prezentacja koncepcji klas. Tu dowiesz się, jak C++ zapewnia obsługę ważnych reguł programowania zorientowanego obiektowo, czyli hermetyzacji, abstrakcji, dziedziczenia i polimorfizmu. W lekcji 9., zatytułowanej „Klasy i obiekty", poznasz nową koncepcję standardu C++11, będzie to konstruktor przenoszący, natomiast w lekcji 12., „Typy operatorów i ich przeciążanie", opisany został przenoszący operator przypisania. Wymienione funkcje wydajności pomagają w uniknięciu niechcianych i niepożądanych kroków kopiowania, co przyczynia się do zwiększenia wydajności działania aplikacji. Z kolei lekcja 14., „Wprowadzenie do makr i wzorców", to wprowadzenie do tworzenia ogólnego kodu C++ o potężnych możliwościach.
- Część III, „Poznajemy standardową bibliotekę wzorców (STL)", to pomoc w tworzeniu efektywnego i praktycznego kodu C++ za pomocą klasy STL `string` oraz kontenerów. Dowiesz się, jak klasa `std::string` może zapewnić bezpieczeństwo prostych operacji łączenia ciągów tekstowych i ułatwić ich przeprowadzanie bez konieczności używania ciągów tekstowych w stylu C (`char*`). W tworzonych aplikacjach będziesz mógł używać tablic dynamicznych i list STL.
- Część IV, „Jeszcze więcej STL", koncentruje się na algorytmach. Przekonasz się, jak za pomocą iteratorów używać funkcji sortowania w kontenerach, takich jak `vector`. W tej części dowiesz się także, jak przy użyciu słowa kluczowego `auto` można znacznie skrócić długość deklaracji iteratorów. W lekcji 22., „Wyrażenia lambda C++11", przedstawiono nową funkcję o użytecznych możliwościach, jej zastosowanie powoduje znaczne zmniejszenie ilości kodu podczas wykorzystywania algorytmów STL.

- Część V, „Zaawansowane koncepcje C++”, zawiera informacje o innych możliwościach języka, takich jak sprytne wskaźniki i obsługa wyjątków, które wprawdzie nie muszą być stosowane w aplikacjach C++, ale ich użycie znacząco poprawia stabilność i jakość kodu. Na końcu tej części znajdują się informacje o najlepszych praktykach wykorzystywanych podczas tworzenia dobrych aplikacji w standardzie C++11.

Konwencje stosowane w książce

W ramach każdej lekcji możesz spotkać pewne elementy dostarczające informacji dodatkowych. Oto one.

Takie ramki zawierają informacje dodatkowe dotyczące przedstawionego materiału.

Uwaga

C++11

Takie ramki zawierają omówienie nowych funkcji wprowadzonych w standardzie C++11. Aby wykorzystać te funkcje, należy użyć najnowszych dostępnych wersji kompilatorów.

Takie ramki mają zwrócić Twoją uwagę na problemy lub efekty uboczne, które mogą wystąpić w pewnych sytuacjach.

Ostrzeżenie

Takie ramki zawierają informacje o najlepszych praktykach stosowanych podczas tworzenia aplikacji w C++.

Wskazówka

TAK	NIE
W ramkach „TAK-NIE” znajdziesz krótkie podsumowanie podstawowych wniosków płynących z lekcji.	Nie przegap użytecznych informacji zawartych w tych ramkach.

W książce zastosowano różne kroje czcionek, pomocne w odróżnieniu kodu C++ od zwykłego tekstu. W poszczególnych lekcjach kod, polecenia oraz wyrażenia powiązane z programowaniem są zapisywane za pomocą `czcionki o stałej szerokości`.

Przykładowe fragmenty kodu dla tej książki

Przykładowe fragmenty kodu wykorzystanego w tej książce można pobrać z witryny internetowej wydawnictwa Helion: *ftp://ftp.helion.pl/przyklady/cppit7.zip*

Część I

Podstawy

Lekcja 1

Zaczynamy

Witamy w *C++. Dla każdego. Wydanie VII*. Po przeczytaniu tego rozdziału będziesz mógł efektywnie rozpocząć programowanie w języku C++.

Z tej lekcji dowiesz się:

- **dlaczego C++ jest standardowym językiem tworzenia oprogramowania,**
- **w jaki sposób wprowadzić, skompilować i zbudować swój pierwszy, działający program w C++,**
- **co nowego oferuje standard C++11.**

Krótka historia języka C++

Celem języka programowania jest ułatwienie konsumpcji zasobów komputera. C++ nie jest nowym językiem, ale jednym z popularniejszych i nieustannie usprawnianych. Najnowsza wersja standardu C++ zatwierdzona w 2011 roku przez Międzynarodową Organizację Normalizacyjną (ISO) nosi nazwę C++11.

Powiązanie z językiem C

C++, początkowo opracowany przez Bjarne'a Stroustroupa w laboratoriach Bell Labs w 1979 roku, powstał jako następca języka C. Wspomniany C to język proceduralny, w którym definiowane są funkcje podejmujące pewne działania. Natomiast C++ został opracowany jako język programowania zorientowanego obiektowo i implementuje koncepcje, takie jak dziedziczenie, abstrakcję, polimorfizm i hermetyzację. C++ oferuje klasy stosowane do przechowywania danych egzemplarza oraz metod operujących na wspomnianych danych. (Metody są bardzo zbliżone do funkcji stosowanych w języku C). W efekcie programista koncentruje się na danych i celach, które chce za ich pomocą osiągnąć. Kompilatory C++ od zawsze kompilują także zwykły kod C. Takie rozwiązanie ma zalety, pod warunkiem, że kod jest zgodny wstecz. Z drugiej strony, to rozwiązanie ma również wady — kompilatory stają się niezwykle skomplikowane, aby zapewnić programistom zgodność wstecz i jednocześnie implementować nowe funkcje wymagane przez ewoluujący język.

Zalety języka C++

C++ jest uznawany za pośredni język programowania, co oznacza, że pozwala na programowanie na zarówno wysokim (aplikacje), jak i niskim (biblioteki ściśle powiązane ze sprzętem) poziomie. Dla wielu programistów C++ stanowi optymalne połączenie języka wysokiego poziomu pozwalającego na tworzenie skomplikowanych aplikacji i jednocześnie oferującego elastyczność umożliwiającą osiągnięcie najlepszej wydajności przez odpowiednią kontrolę dostępności i zużycia zasobów.

Pomimo istnienia nowszych języków programowania, takich jak Java oraz innych opartych na technologii .NET, C++ pozostał ważny i nadal jest rozwijany. Nowsze języki programowania zapewniają pewne funkcje przyciągające niektórych programistów, przykładem może być zarządzanie

pamięcią przez mechanizm usuwania nieużytków zaimplementowany w komponencie środowiska uruchomieniowego. Bardzo wielu z tych programistów wybiera jednak C++, gdy chcą zachować dokładniejszą kontrolę nad wydajnością działania aplikacji. Powszechnie spotykana jest architektura warstwowa, w której serwer WWW został utworzony w C++, natomiast aplikacja graficzna w językach HTML, Java lub .NET.

Ewolucja standardu C++

Lata ewolucji spowodowały, że C++ jest powszechnie akceptowany i stosowany w wielu różnych formach wynikających z użycia różnorodnych kompilatorów charakteryzujących się własnymi cechami. Popularność i różnice w dostępnych wersjach doprowadziły do wielu problemów związanych ze zgodnością i przenośnością, a to spowodowało dążenie do utworzenia standardu.

W roku 1998 pierwszy standard C++ został zatwierdzony przez ISO (ISO/IEC 14882:1998). Następna wersja została zatwierdzona w 2003 roku (ISO/IEC 14882:2003). Obecna wersja standardu C++ została zatwierdzona w sierpniu 2011 roku, oficjalnie nosi nazwę C++11 (ISO/IEC 14882:2011) i zawiera najambitniejsze oraz największe zmiany wprowadzone dotąd w standardzie.

Uwaga

Wiele dokumentacji dostępnej w internecie dotyczy wersji C++ o nazwie C++0x. Oczekiwano, że nowy standard zostanie zatwierdzony w roku 2008 lub 2009 i znaku *x* użyto jako oznaczenia roku. Wreszcie w sierpniu 2011 roku propozycje nowego standardu zostały zatwierdzone i nosi on nazwę C++11.

Innymi słowy, C++11 to nowy C++0x.

Kto używa programów utworzonych w C++?

Niezależnie od tego kim jesteś lub czym się zajmujesz — doświadczonym programistą lub użytkownikiem komputera w ściśle określonych celach — istnieje ogromne prawdopodobieństwo, że nieustannie używasz aplikacji i bibliotek utworzonych w języku C++. Kod C++ znajduje się w systemach operacyjnych, sterownikach urządzeń, aplikacjach biurowych, serwerach WWW, aplikacjach opartych na chmurze, silnikach wyszukiwania, a także w pewnych nowszych językach programowania — C++ to bardzo często język wybierany do budowania wymienionych komponentów.

Tworzenie aplikacji C++

Kiedy w komputerze uruchamiasz program Notatnik lub vi, tak naprawdę nakazujesz procesorowi uruchomienie programu wykonywalnego wskazanej aplikacji. Wspomniany program wykonywalny to ukończony produkt gotowy do uruchomienia, który powinien działać zgodnie z założeniami programisty.

Kroki prowadzące do wygenerowania pliku wykonywalnego

Utworzenie kodu źródłowego w języku C++ to tylko pierwszy krok na drodze do przygotowania pliku wykonywalnego, który można będzie później uruchamiać w systemie operacyjnym. Poniżej przedstawiono podstawowe kroki prowadzące do utworzenia aplikacji w języku C++.

1. Napisanie (utworzenie) kodu źródłowego C++ przy użyciu edytora tekstów.
2. Kompilacja kodu źródłowego za pomocą kompilatora C++, który konwertuje go na wersję w kodzie maszynowym zawartą w „plikach obiektów".
3. Wykorzystanie linkera do utworzenia pliku wykonywalnego (np. *.exe* w systemie Windows) na podstawie danych wyjściowych kompilatora.

Kiedy tworzysz program w C++, przygotowujesz pliki tekstowe zapisywane wraz z rozszerzeniem *.cpp* w nazwie. Jednak mikroprocesor został zaprojektowany do wykonywania instrukcji binarnych, a nie zapisanych w plikach zwykłego tekstu. Kompilacja to krok, w trakcie którego kod C++ znajdujący się w plikach tekstowych z rozszerzeniem *.cpp* jest konwertowany na kod bajtowy zrozumiały dla procesora i możliwy do wykonania przez procesor. Kompilator jednorazowo konwertuje po jednym pliku kodu, generując plik obiektowy z rozszerzeniem *.o* lub *.obj* i ignorując zależności zdefiniowane w danych pliku *.cpp*. Rozwiązanie wspomnianych zależności to zadanie linkera. Oprócz połączenia różnych plików obiektów, linker spełnia inne zależności. Gdy operacja zakończy się powodzeniem, jej efektem jest plik wykonywalny, który programista może uruchamiać i rozpowszechniać.

Analiza błędów i ich usuwanie

Wiele skomplikowanych aplikacji, zwłaszcza tworzonych przez grupy programistów pracujących w zespołach, rzadko kompiluje się i działa bez problemów już za pierwszym razem. Ogromne i skomplikowane aplikacje utworzone w dowolnym języku programowania — także w C++ — bardzo często muszą być wielokrotnie poprawiane w celu przeprowadzenia analizy problemów i wykrycia błędów. Znalezione błędy zostają poprawione, program jest ponownie kompilowany, a proces rozpoczyna się od początku. Dlatego też, oprócz trzech kroków (tworzenie, kompilacja i budowa), prace programistyczne bardzo często obejmują jeszcze czwarty krok, jakim jest usuwanie błędów, w trakcie którego programista analizuje anomalie i błędy w aplikacji przy użyciu odpowiednich narzędzi i funkcji, np. możliwości wykonywania po jednym wierszu kodu źródłowego aplikacji.

Zintegrowane środowiska programistyczne

Wielu programistów preferuje użycie zintegrowanych środowisk programistycznych (ang. *Integrated Development Environment*, IDE), w których kroki tworzenia, kompilacji i budowy aplikacji są zintegrowane w postaci zunifikowanego interfejsu użytkownika. Ponadto środowiska IDE oferują narzędzia służące do usuwania błędów, których używanie ułatwia wykrywanie i rozwiązywanie problemów.

Wskazówka

Istnieje wiele bezpłatnie dostępnych kompilatorów i środowisk IDE dla C++. Popularne rozwiązania to Microsoft Visual C++ Express na platformie Windows i GNU G++ Compiler o nazwie g++ na platformie Linux. Jeżeli zajmujesz się programowaniem na platformie Linux, możesz zainstalować bezpłatnie dostępne środowisko IDE o nazwie Eclipse pozwalające na tworzenie aplikacji C++ za pomocą kompilatora g++.

Ostrzeżenie

Wprawdzie w trakcie pisania tej książki żaden kompilator nie obsługiwał jeszcze w pełni standardu C++11, wiele jego najważniejszych funkcji jest obsługiwanych przez wymienione powyżej kompilatory.

TAK	NIE
Do tworzenia plików kodu źródłowego używaj edytora zwykłego tekstu, np. Notatnika lub gedit bądź też środowiska IDE.	Do tworzenia plików kodu źródłowego nie używaj procesorów tekstów, ponieważ bardzo często dodają one specjalne znaki formatujące.
Zapisuj pliki, nadając im rozszerzenie *.cpp*.	Nie zapisuj plików z rozszerzeniem *.c*, ponieważ wiele kompilatorów potraktuje kod w takich plikach jako kod w języku C, a nie C++.

Tworzenie pierwszej aplikacji w C++

Skoro znasz już narzędzia i kroki wymagane do utworzenia programu, możesz przystąpić do przygotowania pierwszej aplikacji w języku C++. Zgodnie z tradycją, pierwszy program wyświetli na ekranie komunikat „Witaj, świecie!”.

Jeżeli używasz systemu Windows i narzędzia Visual Studio Express 2012 for Windows Desktop, możesz wykonać przedstawione poniżej kroki.

1. Utwórz nowy projekt, wybierając opcje menu *Plik/Nowy Projekt...*.
2. Kliknij *Visual C++*, wybierz szablon *Aplikacja konsoli Win32*. W kolejnym oknie dialogowym usuń zaznaczenie opcji *Nagłówek kompilowany wstępnie*.
3. Tworzonemu projektowi nadaj nazwę *Hello* i zastąp automatycznie wygenerowaną zawartość pliku *Hello.cpp* kodem przedstawionym w listingu 1.1.

   Jeżeli programujesz w systemie Linux, wówczas do utworzenia pliku *hello.cpp* wraz z zawartością przedstawioną w listingu 1.1 użyj zwykłego edytora tekstu (w moim przypadku to gedit w systemie Ubuntu).

Listing 1.1. Plik hello.cpp zawierający kod pierwszego programu w C++

```
1:  #include <iostream>
2:
3:  int main()
4:  {
5:    std::cout << "Witaj, świecie!" << std::endl;
6:    return 0;
7:  }
```

Ta prosta aplikacja jedynie wyświetla wiersz tekstu za pomocą polecenia `std::cout`. Polecenie `std::endl` informuje `cout` o końcu wiersza, a aplikacja kończy działanie, przekazując systemowi operacyjnemu wartość 0.

Uwaga

Aby samodzielnie odczytać program, pomocna może być znajomość wymowy znaków specjalnych i słów kluczowych.

Przykładowo wyrażenie `#include` możesz wymówić jako *hasz-inklad*. Inne spotykane wersje wymowy to *szarp-inklad* lub *paund-inklad*.

Wyrażenie `std::cout` możesz wymówić jako standard-si-aut.

Ostrzeżenie

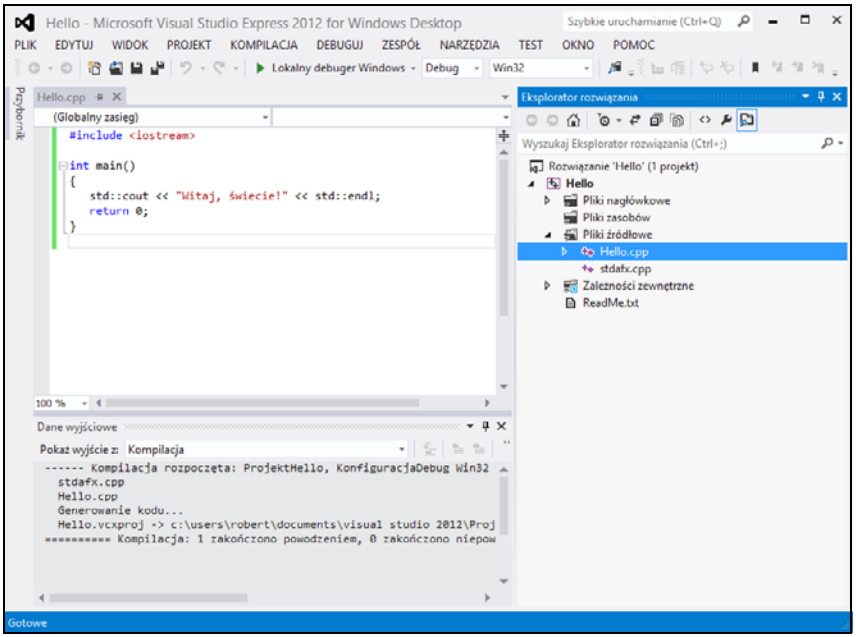
Pamiętaj, że diabeł zawsze tkwi w szczegółach. Oznacza to konieczność wprowadzenia kodu źródłowego w dokładnie takiej samej postaci jak przedstawiona w listingu. Kompilatory są znane z oczekiwania spełnienia ich wymagań. Jeśli przez pomyłkę na końcu polecenia umieścisz dwukropek (`:`) zamiast średnika (`;`), uniemożliwi to prawidłową kompilację kodu.

Kompilacja i uruchomienie pierwszej aplikacji w C++

Jeżeli używasz narzędzia Microsoft Visual C++ Express, naciśnięcie klawiszy *Ctrl+F5* spowoduje uruchomienie aplikacji bezpośrednio przez IDE — środowisko skompiluje, połączy i uruchomi aplikację. Poszczególne kroki możesz również wykonać samodzielnie.

1. Kliknij projekt prawym przyciskiem myszy, a następnie wybierz opcję *Kompiluj* w celu wygenerowania pliku wykonywalnego.
2. Używając wiersza poleceń, przejdź do katalogu zawierającego utworzony plik wykonywalny (najczęściej jest to podkatalog *Debug* w katalogu projektu).
3. Uruchom plik wykonywalny, podając jego nazwę.

Program utworzony w środowisku Microsoft Visual C++ będzie podobny do pokazanego na rysunku 1.1.



RYSUNEK 1.1. Prosty program typu „witaj, świecie" utworzony w Microsoft Visual C++ 2012 Express

Jeżeli używasz systemu Linux, wywołaj kompilator g++ i linker z poziomu wiersza poleceń powłoki:

```
g++ -o hello hello.cpp
```

Wydanie powyższego polecenia spowoduje, że kompilator g++ utworzy plik wykonywalny o nazwie *hello* przez kompilację pliku kodu źródłowego *hello.cpp*. Wydanie polecenia `./hello` w systemie Linux lub `Hello.exe` w Windows spowoduje wygenerowanie poniższych danych wyjściowych:

```
Witaj, świecie!
```

Gratulacje! Jesteś na drodze do poznania jednego z najpopularniejszych i jednocześnie oferującego potężne możliwości języka programowania.

Znaczenie standardu ISO dla C++

Jak możesz się przekonać, zgodność ze standardem powoduje, że kod przedstawiony w listingu 1.1 może być skompilowany i uruchomiony na wielu platformach i systemach operacyjnych — wymaganiem jest dostępność zgodnego ze standardem kompilatora C++. Dlatego też, jeśli musisz utworzyć produkt przeznaczony dla użytkowników systemów np. Windows i Linux, stosowanie się do zasad programowania zgodnego ze standardami (tzn. powstrzymanie się od

używania semantyki charakterystycznej dla danego kompilatora lub platformy) oferuje niedrogi sposób dotarcia do szerszej gamy użytkowników bez konieczności tworzenia oddzielnego programu dla każdego obsługiwanego środowiska. Oczywiście, takie rozwiązanie jest optymalne w przypadku aplikacji, które nie wymagają zbyt dużej interakcji na poziomie systemu operacyjnego.

Błędy kompilacji

Kompilatory są niezwykle skrupulatne i bardzo dokładnie wskazują miejsce, w którym popełniono błąd. Jeżeli napotkasz problem podczas kompilacji aplikacji przedstawionej w listingu 1.1, możesz otrzymać komunikat błędu podobny do przedstawionego poniżej (spowodowany przez pominięcie średnika w wierszu 5. kodu):

```
hello.cpp(6): error C2143: błąd składniowy : brakuje ";" przed "return"
```

Powyższy komunikat błędu został wygenerowany przez kompilator Visual C++ i zawiera dokładne informacje o problemie: podaje nazwę pliku, w którym występuje błąd, numer wiersza (tutaj 6.) oraz opis samego błędu i jego numer (tutaj C2143). Wprawdzie średnik został usunięty w wierszu 5., ale kompilator informuje o błędzie w wierszu 6., ponieważ dopiero w trakcie analizy wiersza 6. istnienie błędu staje się oczywiste dla kompilatora. Możesz więc umieścić średnik na początku wiersza 6., a program zostanie skompilowany bez problemów.

Uwaga

Znak nowego wiersza nie powoduje automatycznego zakończenia polecenia, jak ma to miejsce w niektórych językach, np. VBScript.

W języku C++ pojedyncze polecenie może być zapisane w wielu wierszach.

Co nowego w C++11?

Jeżeli jesteś doświadczonym programistą C++, możesz dostrzec, że prosty program C++ przedstawiony w listingu 1.1 ma taką samą postać jak we wcześniejszych standardach. Standard C++11 zapewnia wsteczną zgodność z poprzednimi wersjami C++, ale włożono naprawdę ogromną ilość pracy, aby język stał się prostszy w użyciu i programowaniu.

Wprowadzone usprawnienia, takie jak nowe słowo kluczowe `auto`, pozwalają na definiowanie zmiennych, których typ będzie automatycznie określony przez kompilator. Dzięki temu następuje skrócenie deklaracji egzemplarza bez wpływu na bezpieczeństwo typów. Z kolei „funkcje lambda” są funkcjami pozbawionymi

nazw. Pozwalają na tworzenie niewielkich obiektów funkcji bez długich definicji klas i znaczne zmniejszenie liczby wierszy kodu. Standard C++11 umożliwia programistom tworzenie przenośnych i wielowątkowych aplikacji C++ zgodnych ze standardem. Tego rodzaju aplikacje po prawidłowym przygotowaniu zapewniają możliwość działania współbieżnego i doskonałego skalowania pod względem wydajności, gdy użytkownik rozbuduje konfigurację sprzętową komputera, np. zwiększając liczbę rdzeni procesora.

Standard C++11 oferuje także wiele innych usprawnień, które będą omówione na kolejnych stronach książki.

Podsumowanie

W czasie tej lekcji nauczyłeś się, jak wprowadzać, kompilować i budować pierwszy program w C++, a także ogólnie poznałeś ewolucję języka C++. Przekonałeś się, że dzięki zgodności ze standardem ten sam program może być skompilowany za pomocą odmiennych kompilatorów w różnych systemach operacyjnych.

Pytania i odpowiedzi

Pytanie: Czy mogę zignorować komunikaty ostrzeżeń generowane przez kompilator?

Odpowiedź: W pewnych sytuacjach kompilatory generują komunikaty ostrzeżeń. Ostrzeżenie różni się od błędu tym, że wskazany wiersz jest syntaktycznie poprawny i możliwy do kompilacji. Jednak istnieje znacznie lepszy sposób jego utworzenia i dobry kompilator wyświetla ostrzeżenie wraz z propozycją poprawy wskazanego wiersza.

Sugerowana poprawka może być znacznie bezpieczniejszym sposobem programowania. Może też umożliwiać aplikacji działanie ze znakami i literami znajdującymi się w innych językach. Zawsze powinieneś brać pod uwagę wygenerowane ostrzeżenia i odpowiednio zmodyfikować program. Nie próbuj maskować komunikatów ostrzeżeń, jeśli nie jesteś absolutnie pewien, że to fałszywy alarm.

Pytanie: Jaka jest różnica pomiędzy językami interpretowanymi a kompilowanymi?

Odpowiedź: Język, taki jak Windows Script, jest interpretowany, co oznacza brak kroku kompilacji. Język interpretowany korzysta z interpretera, który bezpośrednio odczytuje plik skryptu (kodu) i wykonuje wskazane operacje. Dlatego też wykonanie skryptu wymaga zainstalowania w komputerze odpowiedniego interpretera. Wydajność działania programu interpretowanego jest zwykle mniejsza, ponieważ po jego uruchomieniu interpreter działa w charakterze pośrednika pomiędzy mikroprocesorem i utworzonym kodem.

Pytanie: Co to są błędy w trakcie działania programu i czym się różnią od błędów kompilacji?

Odpowiedź: Błędy występujące po uruchomieniu aplikacji są nazywane *błędami w trakcie działania programu*. W starszych wersjach Windows mogłeś się spotkać z niesławnym błędem *Access Violation*, który zalicza się do błędów w trakcie działania programu. Z kolei błędy kompilacji nie dotyczą użytkownika końcowego i wskazują błąd syntaktyczny, który uniemożliwia programiście wygenerowanie pliku wykonywalnego.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Jaka jest różnica pomiędzy interpreterem i kompilatorem?
2. Do jakich celów służy linker?
3. Jakie są kroki w zwykłym cyklu tworzenia programu?
4. Jak standard C++11 zapewnia lepszą obsługę procesorów wielordzeniowych?

Ćwiczenia

1. Spójrz na poniższy program i spróbuj odgadnąć jego sposób działania jeszcze przed uruchomieniem go:

```
1: #include <iostream>
2: int main()
3: {
4:   int x = 8;
5:   int y = 6;
6:   std::cout << std::endl;
7:   std::cout << x - y << "" << x * y << x + y;
8:   std::cout << std::endl;
9:   return 0;
10:}
```

2. Wprowadź program przedstawiony w ćwiczeniu 1., a następnie skompiluj go i zbuduj plik wykonywalny. Jaki jest wynik uruchomienia programu? Czy odgadłeś to wcześniej?

3. Gdzie w poniższym programie znajduje się błąd?

```
1: include <iostream>
2: int main()
3: {
4:     std::cout << "Hello, świecie z błędami \n";
5:     return 0;
6: }
```

4. Popraw błąd w programie przedstawionym w ćwiczeniu 3., a następnie skompiluj program, zbuduj plik wykonywalny i uruchom go. Jaki jest wynik uruchomienia programu?

Lekcja 2

Anatomia programu C++

Programy C++ składają się z obiektów, funkcji, zmiennych i innych elementów. Większość tej książki stanowi obszerny opis tych elementów, jednakże w celu zrozumienia zasad ich współdziałania musisz najpierw poznać cały działający program.

Z tej lekcji dowiesz się:

- **poznasz elementy programu C++,**
- **dowiesz się, jak te elementy współpracują,**
- **dowiesz się, czym jest funkcja i do czego służy,**
- **poznasz podstawowe operacje wejścia-wyjścia.**

Komponenty programu

Pierwszy program w C++, który utworzyliśmy w lekcji 1., był bardzo prosty; jego działanie ograniczało się do wyświetlenia komunikatu *Witaj, świecie!* na ekranie. Nawet ten program, pomimo wspomnianej prostoty, składał się z pewnych najważniejszych i podstawowych komponentów tworzących program w języku C++. Kod przedstawiony w listingu 2.1 wykorzystamy jako punkt wyjścia do analizy komponentów wszystkich programów C++.

Listing 2.1. Plik HelloWorldAnalysis.cpp: analiza programu C++

```
1: // Dyrektywa preprocesora, która powoduje dołączenie pliku nagłówkowego iostream.
2: #include <iostream>
3:
4: // Początek programu: blok funkcji main().
5: int main()
6: {
7:     /* Wyświetlenie komunikatu na ekranie. */
8:     std::cout << "Witaj, świecie" << std::endl;
9:
10:    // Zwrot wartości systemowi operacyjnemu.
11:    return 0;
12: }
```

Powyższy program C++ można z grubsza podzielić na dwie części: pierwsza to dyrektywy preprocesora rozpoczynające się od znaku #, a druga to część główna programu rozpoczynająca się od wiersza `int main()`.

Uwaga

Wiersze 1., 4., 7. i 10. rozpoczynające się znakami // lub /* są komentarzami ignorowanymi przez kompilator. Komentarze są przeznaczone dla ludzi odczytujących kod źródłowy.

Komentarzami dokładniej zajmiemy się w dalszej części lekcji.

Dyrektywa preprocesora #include

Jak sama nazwa wskazuje, *preprocesor* jest narzędziem działającym przed rozpoczęciem faktycznej kompilacji. Dyrektywy preprocesora to polecenia skierowane do preprocesora i zawsze rozpoczynające się od znaku #. Polecenie w wierszu 2. listingu 2.1 (`#include <nazwa-pliku>`) nakazuje preprocesorowi wczytanie zawartości wskazanego pliku (w omawianym przykładzie to *iostream*) i umieszczenie jej w miejscu dyrektywy. Wymieniony *iostream* to standardowy plik nagłówkowy, który jest dołączany, ponieważ zawiera definicję polecenia

`std::cout` użytego w wierszu 8. do wyświetlenia na ekranie komunikatu *Witaj, świecie*. Innymi słowy, kompilator mógł skompilować wiersz 8. kodu zawierający wywołanie `std::cout`, bo w wierszu 2. nakazaliśmy preprocesorowi wczytanie definicji `std::cout`.

Uwaga

W profesjonalnie tworzonych aplikacjach C++ dołączane są nie tylko standardowe pliki nagłówkowe. Kod źródłowy skomplikowanych aplikacji najczęściej znajduje się w wielu plikach, w niektórych z nich są dołączane kolejne. Dlatego też jeśli kod zdefiniowany w *PlikA* musi być użyty w *PlikB*, wtedy *PlikA* trzeba wczytać w *PlikB*. Odbywa się to przez umieszczenie poniższego polecenia `include` w *PlikB*:

```
#include "...względna ścieżka dostępu do PlikA\PlikA"
```

W omawianym przykładzie użyto cudzysłowów zamiast nawiasów ostrych, które najczęściej są używane podczas dołączania standardowych plików nagłówkowych.

Część główna programu — funkcja main()

Po dyrektywach preprocesora znajduje się część główna programu w postaci funkcji `main()`. Uruchomienie programu C++ zawsze rozpoczyna się w tej funkcji. Standardową konwencją jest zadeklarowanie funkcji `main()` poprzedzonej typem `int` — jest to typ zwrotny funkcji `main()`.

Uwaga

W wielu aplikacjach C++ znajdziesz różne odmiany deklaracji funkcji `main()`, np. taką jak poniższa:

```
int main (int argc, char* argv[])
```

To również jest zgodne ze standardem i akceptowalne, ponieważ wartość zwrotna funkcji `main()` jest typu `int`. W nawiasie znajdują się „argumenty" przekazywane programowi. Tego rodzaju program prawdopodobnie pozwala użytkownikowi na jego uruchomienie z poziomu wiersza poleceń wraz z argumentami, np.:

```
program.exe /ZróbCośKonkretnego
```

`/ZróbCośKonkretnego` to argument przekazywany programowi przez system operacyjny, a następnie obsłużony wewnątrz funkcji `main()`.

Przeanalizujmy teraz wiersz 8., który jest sednem omawianego programu.

```
std::cout << "Witaj, świecie" << std::endl;
```

Polecenie `cout` (ang. *console out*, dane wyjściowe konsoli) to polecenie wyświetlające na ekranie komunikat *Witaj, świecie*. Polecenie `cout` to strumień zdefiniowany w standardowej przestrzeni nazw (stąd `std::cout`), a działanie omawianego wiersza polega na umieszczeniu danych `"Witaj, świecie"` w strumieniu przy użyciu operatora <<. Z kolei `std::endl` wskazuje koniec wiersza i jest równoznaczne z umieszczeniem znaku nowego wiersza. Zwróć uwagę, że operator wstawiania do strumienia jest używany za każdym razem, gdy w strumieniu trzeba umieścić kolejne dane.

Cechą strumieni w C++ jest to, że podobna semantyka strumieni używana wraz z innym typem strumienia powoduje wykonanie innej operacji na tym samym tekście, np. jego umieszczenie w pliku zamiast wyświetlenia na ekranie. Dlatego też praca ze strumieniami jest intuicyjna i gdy przywykniesz do jednego strumienia (np. `cout` wyświetlającego tekst na ekranie), bardzo łatwo będziesz mógł pracować z innymi (takimi jak `fstream`, który pomaga w zapisie plików tekstowych na dysku).

Szczegółowe omówienie strumieni znajdziesz w lekcji 27., zatytułowanej „Użycie strumieni w operacjach wejścia-wyjścia”.

Uwaga
Uwaga

Rzeczywisty tekst ujęty w cudzysłów (tutaj `"Witaj, świecie"`) jest nazywany dosłownym ciągiem tekstowym.

Wartość zwrotna

Funkcje w C++ muszą zwracać wartość, o ile nie zostanie wskazane inaczej. Dotyczy to również funkcji `main()`, która zawsze zwraca wartość w postaci liczby całkowitej. Wartość zostaje zwrócona systemowi operacyjnemu i w zależności od natury aplikacji może być bardzo użyteczna, ponieważ większość systemów operacyjnych ma możliwość sprawdzenia wartości zwrotnej aplikacji. W większości przypadków aplikacja jest uruchamiana przez inną aplikację, a aplikacja nadrzędna (uruchamiająca) chce sprawdzić, czy aplikacja potomna (uruchomiona) zakończyła z powodzeniem wykonywanie swoich zadań. Programista może wykorzystać wartość zwrotną funkcji `main()` do wskazania aplikacji nadrzędnej sukcesu bądź niepowodzenia.

Uwaga

Wedle konwencji w przypadku powodzenia zwracana jest wartość 0, natomiast po wystąpieniu błędu wartością zwrotną jest –1. Wartość zwrotna jest liczbą całkowitą i programista dysponuje ogromną elastycznością w zakresie wskazywania wielu różnych stanów sukcesu lub niepowodzenia za pomocą dostępnego zakresu wartości zwrotnych.

Ostrzeżenie

W języku C++ wielkość liter ma znaczenie. Dlatego też użycie zapisu `Int` zamiast `int`, `Void` zamiast `void` lub `Std::Cout` zamiast `std::cout` spowoduje niepowodzenie kompilacji programu.

Koncepcja przestrzeni nazw

Powodem używania w programie wywołania `std::cout` zamiast jedynie `cout` jest fakt, że wywoływany komponent (`cout`) pochodzi ze standardowej (`std`) przestrzeni nazw.

W tym miejscu rodzi się pytanie, czym jest przestrzeń nazw.

Przyjmujemy założenie, że w trakcie wywoływania `cout` nie jest wskazywana przestrzeń nazw, a sam komponent `cout` istnieje w dwóch położeniach znanych kompilatorowi. Którą wersję komponentu powinien wywołać kompilator w takim przypadku? Taka sytuacja prowadzi do konfliktu i — oczywiście — niepowodzenia kompilacji. Wtedy użyteczne staje się zastosowanie przestrzeni nazw. Przestrzenie nazw są nazwami nadawanymi fragmentom kodu w celu zmniejszenia niebezpieczeństwa wystąpienia potencjalnego konfliktu nazw. Używając wywołania `std::cout`, nakazujesz kompilatorowi zastosowanie unikalnego `cout` dostępnego w przestrzeni nazw `std`.

Uwaga

Standardowa przestrzeń nazw `std` jest używana do wywoływania funkcji, strumieni i narzędzi zatwierdzonych przez ISO i tym samym zdefiniowanych w standardzie.

Wielu programistów uznaje za męczące nieustanne podawanie przestrzeni nazw `std` w kodzie podczas używania `cout` oraz innych funkcji w standardowej przestrzeni nazw. Użycie deklaracji `using namespace`, tak jak przedstawiono w listingu 2.2, pozwala na uniknięcie konieczności nieustannego podawania przestrzeni nazw.

Listing 2.2. Deklaracja using namespace

```
 1: // Dyrektywa preprocesora.
 2: #include <iostream>
 3:
 4: // Początek programu.
 5: int main()
 6: {
 7:     // Wskazanie kompilatorowi przestrzeni nazw, która ma być przeszukiwana.
 8:     using namespace std;
 9:
10:     /* Wyświetlenie komunikatu na ekranie za pomocą std::cout. */
11:     cout << "Witaj, świecie" << endl;
12:
13:     // Zwrot wartości systemowi operacyjnemu.
14:     return 0;
15: }
```

Analiza ▼

Zwróć uwagę na wiersz 8. listingu. Dzięki poinformowaniu kompilatora o użyciu przestrzeni nazw `std` nie ma konieczności jej wyraźnego podawania w wierszu 11., np. w postaci wywołań `std::cout` i `std::endl`.

Bardziej restrykcyjny wariant kodu został przedstawiony w listingu 2.3, w którym nie dołączono całej przestrzeni nazw. Wskazane zostały jedynie te komponenty, których chcesz użyć w listingu.

Listing 2.3. Inne użycie słowa kluczowego using

```
 1: // Dyrektywa preprocesora.
 2: #include <iostream>
 3:
 4: // Początek programu.
 5: int main()
 6: {
 7:     using std::cout;
 8:     using std::endl;
 9:
10:     /* Wyświetlenie komunikatu na ekranie za pomocą cout.*/
11:     cout << "Witaj, świecie" << endl;
12:
13:     // Zwrot wartości systemowi operacyjnemu.
14:     return 0;
15: }
```

Analiza ▼

Wiersz 8. listingu 2.2 został zastąpiony w listingu 2.3 wierszami 7. i 8. Różnica pomiędzy poleceniami `using namespace std` i `using std::cout` oraz `using std::endl` polega na tym, że pierwsze pozwala na używanie wszystkich komponentów przestrzeni nazw `std` bez konieczności wyraźnego podawania kwalifikatora przestrzeni nazw (`std::`). Natomiast w drugim poleceniu możliwość pominięcia kwalifikatora przestrzeni nazw dotyczy jedynie `std::cout` i `std::endl`.

Komentarze w kodzie C++

W listingu 2.3 wiersze 1., 4., 10. i 13. zawierają tekst w języku polskim, który jednak nie przeszkadza w prawidłowej kompilacji programu. Wspomniany tekst nie powoduje również zmiany generowanych danych wyjściowych. Tego rodzaju wiersze są nazywane *komentarzami*. Komentarze są ignorowane przez kompilator i chętnie używane przez programistów w celu umieszczania objaśnień kodu — stąd są zapisane w języku czytelnym i zrozumiałym dla ludzi.

Język C++ obsługuje dwa style komentarzy.

- Znaki // wskazują, że dany wiersz jest komentarzem.

  ```
  // To jest komentarz.
  ```

- Tekst ujęty między znakami /* i */ jest traktowany jako komentarz, nawet jeśli obejmuje więcej niż tylko jeden wiersz.

  ```
  /* To jest komentarz
  obejmujący dwa wiersze. */
  ```

Uwaga

Może się wydawać dziwne, że programista musi umieszczać objaśnienia własnego kodu. Jednak w tworzenie dużych programów zaangażowana jest większa liczba programistów pracujących nad poszczególnymi modułami, więc ogromne znaczenie ma tworzenie kodu łatwego do zrozumienia. Bardzo ważne jest wyjaśnienie działania danego fragmentu kodu i powodów jego utworzenia w jasny sposób. Wtedy doskonale sprawdzają się dobrze napisane komentarze.

TAK	NIE
Umieszczaj komentarze objaśniające działanie skomplikowanych algorytmów oraz złożonych fragmentów programu. Komentarze pisz w stylu zrozumiałym dla pozostałych programistów pracujących nad programem.	Nie używaj komentarzy w celu powtórzenia objaśnień lub wyjaśniania oczywistych kwestii. Nie zapominaj, że dodawanie komentarzy nie zwalnia od tworzenia przejrzystego kodu. Nie zapominaj, że uaktualnienie kodu może wymagać także uaktualnienia dotyczących go komentarzy.

Funkcje w C++

Funkcje w językach C i C++ pełnią podobną rolę. Funkcja to po prostu komponent programu i pozwala na podział kodu aplikacji na funkcjonalne jednostki, które będą mogły być wywoływane w wybranej kolejności. Funkcja po wywołaniu najczęściej zwraca pewną wartość. Najbardziej znaną funkcją jest — oczywiście — `main()`. Funkcja ta jest uznawana przez kompilator jako punkt początkowy aplikacji C++, a jej wartością zwrotną jest typ `int` (liczba całkowita).

Jako programista sam decydujesz o potrzebie utworzenia własnych funkcji. W listingu 2.4 przedstawiono kod prostej aplikacji używającej funkcji do wyświetlenia na ekranie komunikatów za pomocą wywołań `std::cout` z różnymi parametrami.

Listing 2.4. Zadeklarowanie, zdefiniowanie i wywołanie funkcji pokazującej pewne możliwości std::cout

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // Deklaracja funkcji.
 5: int DemoConsoleOutput();
 6:
 7: int main()
 8: {
 9:    // Wywołanie funkcji.
10:    DemoConsoleOutput();
11:
12:    return 0;
13: }
14:
15: // Definicja funkcji.
```

```
16: int DemoConsoleOutput()
17: {
18:    cout << "To jest prosty, dosłowny ciąg tekstowy" << endl;
19:    cout << "Wyświetlenie liczby pięć: " << 5 << endl;
20:    cout << "Operacja dzielenia 10 / 5 = " << 10 / 5 << endl;
21:    cout << "Przybliżona wartość Pi 22 / 7 = " << 22 / 7 << endl;
22:    cout << "Dokładniejsza wartość Pi 22 / 7 = " << 22.0 / 7 << endl;
23:
24:    return 0;
25: }
```

Wynik ▼

```
To jest prosty, dosło ny ciąg tekstowy
Wyświetlenie liczby pięć: 5
Operacja dzielenia 10 / 5 = 2
Przybliżona wartość Pi 22 / 7 = 3
Dokładniejsza wartość Pi 22 / 7 = 3.14286
```

Analiza ▼

Najbardziej interesujące są wiersze 5., 10. i od 15. do 25. W wierszu 5. znajduje się tzw. *deklaracja funkcji*, która informuje kompilator, że chcesz utworzyć funkcję o nazwie `DemoConsoleOutput()` zwracającej wartość `int` (liczba całkowita). Ponieważ to jest *deklaracja*, kompilator bez problemów kompiluje ten wiersz kodu, przyjmując założenie, że *definicja* (tzn. właściwa implementacja funkcji) znajduje się dalej, czyli w omawianym listingu w wierszach od 15. do 25.

Funkcja w omawianym programie pokazuje różne możliwości `cout`. Zwróć uwagę, że nie tylko jest wyświetlany tekst, podobnie jak komunikat *Witaj, świecie* w poprzednich przykładach, ale także wyniki prostych operacji arytmetycznych. W wierszach 21. i 22. wyświetlane są wartości Pi (`22 / 7`). Wartość wygenerowana przez wiersz 22. jest dokładniejsza, ponieważ operacja dzielenia `22.0` przez 7 nakazuje kompilatorowi potraktowanie wyniku jako liczby rzeczywistej (`float` w terminologii C++), a nie jako liczby całkowitej.

Zwróć uwagę, że funkcja ma zwracać liczbę całkowitą, a zdefiniowaną wartością zwrotną jest 0. Ponieważ funkcja nie jest używana do podejmowania jakichkolwiek decyzji, nie ma potrzeby zwracania innej wartości. Podobnie wartością zwrotną funkcji `main()` jest 0. Biorąc pod uwagę fakt, że funkcja `main()` przekazała całą aktywność do funkcji `DemoConsoleOutput()`, lepszym rozwiązaniem będzie użycie drugiej z wymienionych funkcji do zwrócenia wartości zwrotnej z `main()`, co przedstawiono w listingu 2.5.

Listing 2.5. Użycie wartości zwrotnej funkcji

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // Deklaracja i definicja funkcji.
 5: int DemoConsoleOutput()
 6: {
 7:    cout << "To jest prosty, dosłowny ciąg tekstowy" << endl;
 8:    cout << "Wyświetlenie liczby pięć: " << 5 << endl;
 9:    cout << "Operacja dzielenia 10 / 5 = " << 10 / 5 << endl;
10:    cout << "Przybliżona wartość Pi 22 / 7 = " << 22 / 7 << endl;
11:    cout << "Dokładniejsza wartość Pi 22 / 7 = " << 22.0 / 7 << endl;
12:
13:    return 0;
14: }
15:
16: int main()
17: {
18:    // Wywołanie funkcji wraz z wartością zwrotną przekazywaną systemowi operacyjnemu.
19:    return DemoConsoleOutput();
20: }
```

Analiza ▼

Dane wyjściowe powyższego programu są dokładnie takie same jak poprzedniego. Drobne zmiany dotyczą jedynie sposobu utworzenia kodu źródłowego. Przede wszystkim, w wierszu 5. zdefiniowano (zaimplementowano) funkcję `DemoConsoleOutput()`, a dopiero później funkcję `main()`. Nie trzeba więc umieszczać dodatkowego wiersza z deklaracją funkcji `DemoConsoleOutput()`. Większość nowoczesnych kompilatorów C++ rozpoznaje taką deklarację i definicję funkcji. Wielkość funkcji `main()` nieco się zmniejszyła. W wierszu 19. następuje wywołanie `DemoConsoleOutput()` i jednocześnie zwrot wartości zwrotnej z aplikacji.

Uwaga

W przypadkach takich jak przedstawiony powyżej, gdy funkcja nie musi podejmować decyzji lub zwracać informacji o powodzeniu bądź niepowodzeniu, można zdefiniować ją jako typ `void`:

```
void DemoConsoleOutput()
```

Tego rodzaju funkcja nie może zwracać wartości, a wykonanie funkcji typu `void` nie może być wykorzystane do podjęcia decyzji.

Funkcje mogą m.in. pobierać parametry, mogą być rekurencyjne, zawierać wiele poleceń zwracających wartość, zostać przeciążone, rozwinięte przez kompilator itd. Szczegółowe omówienie wymienionych koncepcji znajdziesz w lekcji 7., zatytułowanej „Funkcje".

Podstawowe wejście za pomocą std::cin i wyjście za pomocą std::cout

Komputer pozwala na interakcje z uruchomionymi w nim aplikacjami, interakcje te mogą przybierać wiele różnych form. Podstawowym sposobem pracy z programem jest użycie klawiatury i myszy. Informacje wyświetlane na ekranie mogą mieć postać zwykłego tekstu, skomplikowanej grafiki, mogą zostać zapisane do pliku w celu ich późniejszego wykorzystania, a informacje mogą być wydrukowane na papierze przez drukarkę. W tym podrozdziale zostanie przedstawiona najprostsza postać wejścia i wyjścia w programie C++, czyli wykorzystanie konsoli do wyświetlania i pobierania informacji.

Wywołanie `std::cout` (wymowa: *standard si-aut*) jest używane do wyświetlenia danych tekstowych na ekranie, natomiast `std::cin` do pobrania tekstu i liczb (wprowadzonych za pomocą klawiatury) z konsoli. W rzeczywistości wyświetlenie komunikatu „Witaj, świecie" w listingu 2.1 odbyło się za pomocą `cout`:

```
8:    std::cout << "Witaj, świecie" << std::endl;
```

Po poleceniu `cout` znajduje się operator wstawiania `<<` (pomaga we wstawianiu danych do strumienia danych wyjściowych), następnie dosłowny ciąg tekstowy „Witaj, świecie", a dalej znak nowego wiersza w postaci polecenia `std::endl` (wymowa: *standard end-lajn*).

Użycie `cin` również jest bardzo proste. To polecenie służy do pobrania danych wejściowych, a następnie ich umieszczenia we wskazanej zmiennej:

```
std::cin >> Zmienna;
```

Po poleceniu `cin` znajduje się operator wyodrębniania `>>` (wyodrębnia dane ze strumienia danych wejściowych), a następnie nazwa zmiennej przeznaczonej do przechowywania wspomnianych danych. Jeżeli dane wejściowe podawane przez użytkownika muszą być przechowywane w dwóch zmiennych (każda zawiera dane rozdzielone spacją), wtedy można użyć poniższego polecenia:

```
std:cin >> Zmienna1 >> Zmienna2;
```

Zwróć uwagę, że polecenie `cin` może być używane do pobierania od użytkownika danych zarówno tekstowych, jak i liczbowych. Przykład przedstawiono w listingu 2.6.

Listing 2.6. Użycie cin i cout w celu wyświetlenia danych wejściowych (liczba i tekst) podanych przez użytkownika

```
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     // Zadeklarowanie zmiennej przechowującej liczbę całkowitą.
 8:     int InputNumber;
 9:
10:     cout << "Podaj liczbę całkowitą: ";
11:
12:     // Zachowanie liczby całkowitej podanej przez użytkownika.
13:     cin >> InputNumber;
14:
15:     // Zachowanie tekstu podanego przez użytkownika.
16:     cout << "Podaj imię: ";
17:     string InputName;
18:     cin >> InputName;
19:
20:     cout << InputName << " podał liczbę " << InputNumber << endl;
21:
22:     return 0;
23: }
```

Wynik ▼

```
Podaj liczbę całkowitą: 2011
Podaj imię: Robert
Robert podał liczbę 2011
```

Analiza ▼

W wierszu 8. pokazano deklarację zmiennej `InputNumber` przeznaczonej do przechowywania danych typu `int`. W wierszu 10. polecenie `cout` wykorzystano do poproszenia użytkownika o podanie liczby, która następnie w wierszu 13. za pomocą `cin` zostaje umieszczona w zmiennej. Ta sama operacja zostaje powtórzona w celu pobrania imienia użytkownika, ale tym razem dane wejściowe mają postać tekstu i są przechowywane w zmiennej typu `string` (patrz wiersze 17. i 18.). Z tego powodu w wierszu 2. dołączono plik nagłówkowy

<string> zawierający deklarację używanego później w funkcji main() typu string. Wreszcie, w wierszu 20. polecenie cout jest używane do wyświetlenia podanych wcześniej przez użytkownika imienia i liczby, co w omawianym przypadku oznacza wygenerowanie danych wyjściowych Robert podał liczbę 2011.

Jest to bardzo prosty przykład pokazujący działanie standardowego wejścia i wyjścia w programach C++. Nie przejmuj się, jeśli koncepcja zmiennych nie jest zupełnie jasna, ponieważ dokładnie omówiona zostanie w lekcji 3., zatytułowanej „Zmienne i stałe".

Podsumowanie

W tej lekcji przedstawiono podstawowe komponenty prostego programu w C++. Dowiedziałeś się, czym jest funkcja main(), przestrzenie nazw oraz poznałeś podstawy wejścia i wyjścia konsoli. Zaprezentowane w tej lekcji koncepcje będziesz stosował w każdym tworzonym programie.

Pytania i odpowiedzi

Pytanie: Do czego służy dyrektywa #include?

Odpowiedź: To jest dyrektywa dla preprocesora uruchamianego w trakcie działania kompilatora. Ta konkretna dyrektywa powoduje, że zawartość pliku wymienionego w nawiasie <> po słowie kluczowym #include będzie umieszczona w miejscu dyrektywy, dokładnie tak samo jakby została wprowadzona w kodzie źródłowym.

Pytanie: Jaka jest różnica pomiędzy komentarzami w stylu // i /*?

Odpowiedź: Komentarz w postaci podwójnego ukośnika (//) obejmuje dany wiersz. Z kolei komentarz rozpoczęty przez znaki ukośnika i gwiazdki (/*) rozciąga się aż do znacznika końcowego komentarza (*/). Komentarze rozpoczynające się od podwójnego ukośnika są nazywane *komentarzami jednowierszowymi*, natomiast komentarze rozpoczynające się od ukośnika i gwiazdki są nazywane *komentarzami wielowierszowymi*. Musisz pamiętać, że koniec funkcji nie powoduje automatycznego zakończenia komentarza rozpoczętego od znaków ukośnika i gwiazdki. Jeżeli nie umieścisz znacznika końcowego komentarza, możesz spodziewać się błędów podczas kompilacji programu.

Pytanie: Kiedy konieczne jest programowanie argumentów wiersza poleceń?

Odpowiedź: Jeśli chcesz umożliwić użytkownikowi zmianę zachowania programu. Przykładowo polecenie `ls` w systemie Linux i `dir` w systemie Windows pozwala na wyświetlenie zawartości katalogu bieżącego. Aby wyświetlić pliki znajdujące się w innym katalogu, można podać ścieżkę dostępu do wybranego katalogu, używając argumentów wiersza poleceń, np. `ls /` lub `dir \`.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Jaki jest problem w deklaracji `Int main()`?
2. Czy komentarze mogą być dłuższe niż jeden wiersz?

Ćwiczenia

1. **Łowcy błędów:** Wprowadź poniższy program i skompiluj go. Dlaczego wykonanie programu kończy się niepowodzeniem? W jaki sposób można go poprawić?

```
1: #include <iostream>
2: void main()
3: {
4:    std::Cout << Czy tutaj jest błąd?";
5: }
```

2. Popraw błąd w ćwiczeniu 1., a następnie ponownie skompiluj program, zbuduj plik wykonywalny i uruchom go.
3. Zmodyfikuj listing 2.4 w celu pokazania operacji odejmowania liczb (–) i ich mnożenia (*).

Lekcja 3

Zmienne i stałe

Zmienne to narzędzia, które umożliwiają programiście tymczasowe przechowywanie danych. Z kolei *stałe* to narzędzia pozwalające programiście na zdefiniowanie danych, które nie będą mogły ulec zmianie.

Z tej lekcji dowiesz się:

- **jak używać słów kluczowych auto i constexpr wprowadzonych w standardzie C++11,**
- **jak deklarować i definiować zmienne oraz stałe,**
- **jak przypisywać wartości zmiennym oraz jak nimi manipulować,**
- **jak wyświetlać wartość zmiennej na ekranie.**

Czym jest zmienna?

Zanim faktycznie zaczniesz zapoznawać się ze zmiennymi i sposobami ich użycia w języku programowania, warto zrobić krok wstecz i przekonać się, co komputer zawiera i jak działa.

Ogólne omówienie pamięci i adresowania

Wszystkie komputery, smartfony i inne programowalne urządzenia zawierają mikroprocesor oraz pewną ilość pamięci (ang. *Random Access Memory*, RAM) przeznaczoną do tymczasowego przechowywania danych. Ponadto wiele urządzeń pozwala na trwałe przechowywanie danych w pamięci masowej, np. na dysku twardym. Mikroprocesor uruchamiający aplikację wykorzystuje pamięć RAM, umieszczając w niej program przeznaczony do wykonania, a także powiązane z nim dane, m.in. te wyświetlane na ekranie lub wprowadzone przez użytkownika.

Samą pamięć RAM można potraktować jako szereg pojemników ułożonych jeden za drugim. Każdy z nich jest oznaczony numerem, czyli adresem. Aby uzyskać dostęp do adresu, np. 578, procesor musi otrzymać instrukcję pobrania wartości ze wskazanego adresu lub jej zapisania pod wskazanym adresem.

Deklarowanie zmiennych uzyskujących dostęp i używających pamięci

Za pomocą przedstawionych poniżej przykładów zrozumiesz, czym są zmienne. Przyjmujemy założenie, że tworzysz program mnożący dwie liczby podane przez użytkownika. W uruchomionym programie użytkownik jest proszony o podanie mnożnej i mnożnika, a Ty musisz mieć możliwość przechowywania tych liczb w celu ich późniejszego użycia. W zależności od przeznaczenia wyniku operacji mnożenia, również możesz go przechowywać w celu późniejszego użycia w programie. Bezpośrednie operowanie adresami pamięci (takimi jak 578) do przechowywania liczb byłoby wolne i podatne na błędy, ponieważ wymaga zastosowania mechanizmów uniemożliwiających przypadkowe nadpisanie istniejących danych we wskazanym adresie pamięci.

Kiedy programujesz w języku, takim jak C++, wtedy do przechowywania wspomnianych liczb definiujesz zmienne. Operacja definiowania zmiennej jest bardzo prosta i przedstawia się następująco:

```
typ_zmiennej nazwa_zmiennej;
```

lub

```
typ_zmiennej nazwa_zmiennej = wartość_początkowa;
```

Typ zmiennej informuje kompilator o naturze danych przechowywanych przez tę zmienną, co pozwala kompilatorowi na zarezerwowanie dla niej odpowiedniej ilości miejsca. Wybrana przez programistę nazwa zmiennej to przyjazny zamiennik adresu w pamięci, w którym jest przechowywana wartość zmiennej. O ile wartość nie zostanie początkowo przypisana, nie można mieć pewności dotyczącej zawartości danego adresu w pamięci, co może mieć negatywne skutki dla programu. Dlatego też, choć inicjalizacja zmiennej jest opcjonalna, to stanowi dobrą praktykę programistyczną. W listingu 3.1 przedstawiono sposób deklaracji, inicjalizacji i użycia zmiennych w programie, który mnoży dwie liczby podane przez użytkownika.

Listing 3.1. Użycie zmiennych do przechowywania liczb oraz wyniku ich mnożenia

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main ()
 5: {
 6:    cout << "Ten program pomoże w mnożeniu dwóch liczb" << endl;
 7:
 8:    cout << "Podaj pierwszą liczbę: ";
 9:    int FirstNumber = 0;
10:    cin >> FirstNumber;
11:
12:    cout << "Podaj drugą liczbę: ";
13:    int SecondNumber = 0;
14:    cin >> SecondNumber;
15:
16:    // Mnożenie dwóch liczb i umieszczenie wyniku w zmiennej.
17:    int MultiplicationResult = FirstNumber * SecondNumber;
18:
19:    // Wyświetlenie wyniku.
20:    cout << FirstNumber << " x " << SecondNumber;
21:    cout << " = " << MultiplicationResult << endl;
22:
23:    return 0;
24: }
```

Wynik ▼

```
Ten program pomoże w mnożeniu dwóch liczb
Podaj pierwszą liczbę: 51
Podaj drugą liczbę: 24
51 x 24 = 1224
```

Analiza ▼

Przedstawiony program prosi użytkownika o podanie dwóch liczb, które następnie są mnożone, a wynik operacji wyświetlany na ekranie. Aby aplikacja mogła używać liczb wprowadzonych przez użytkownika, musi przechowywać je w pamięci. Zadeklarowane w wierszach 9. i 13. zmienne `FirstNumber` i `SecondNumber` służą do tymczasowego przechowywania liczb całkowitych podanych przez użytkownika. Pobranie wartości od użytkownika odbywa się za pomocą `std::cin` w wierszach 10. i 14., a następnie liczby zostają umieszczone w zmiennych. Użyte w wierszu 21. polecenie `cout` służy do wyświetlenia na ekranie wyniku operacji mnożenia.

Spójrz na wiersz deklaracji zmiennej:

```
9:    int FirstNumber = 0;
```

W powyższym wierszu następuje zadeklarowanie zmiennej typu `int` oznaczającej liczbę całkowitą i nadanie jej nazwy `FirstNumber`. Zmiennej zostaje przypisana wartość początkowa zero.

Dla porównania, w asemblerze musisz wyraźnie poprosić procesor o umieszczenie mnożnika we wskazanym adresie, np. 578. W języku C++ możesz uzyskać dostęp do pamięci oraz pobierać z niej dane, używając do tego przyjaznych koncepcji, takich jak zmienna `FirstNumber`. Zadaniem kompilatora jest mapowanie podanej zmiennej `FirstNumber` na odpowiedni adres w pamięci i zajęcie się wszystkimi wymaganymi operacjami związanymi z obsługą używania zmiennej.

Programista zyskuje więc możliwość pracy z łatwymi dla człowieka nazwami, a kompilator zajmuje się konwersją zmiennej na odpowiedni adres i utworzeniem instrukcji dla mikroprocesora oraz pracą z pamięcią RAM.

Ostrzeżenie

Nadawanie zmiennym odpowiednich nazw jest bardzo ważne w celu tworzenia dobrego, zrozumiałego i łatwego w obsłudze kodu.

Nazwy zmiennych mogą być alfanumeryczne, ale nie mogą rozpoczynać się od cyfry. Ponadto nie mogą zawierać spacji i operatorów arytmetycznych, np. +, – itd. Nazwę zmiennej możesz wydłużyć za pomocą znaków podkreślenia.

Nazwy zmiennych nie mogą być również zarezerwowanymi słowami kluczowymi. Przykładowo zmienna o nazwie `return` uniemożliwi kompilację programu.

Deklarowanie i inicjalizowanie wielu zmiennych tego samego typu

W listingu 3.1 zmienne `FirstNumber`, `SecondNumber` i `MultiplicationResult` są tego samego typu — liczby całkowite — i zostały zadeklarowane w trzech oddzielnych wierszach. Jeżeli chcesz, deklarację trzech wymienionych zmiennych można przeprowadzić w pojedynczym wierszu kodu:

```
int FirstNumber = 0, SecondNumber = 0, MultiplicationResult = 0;
```

Uwaga

Jak widać, C++ pozwala na deklarowanie za pomocą pojedynczego polecenia wielu zmiennych tego samego typu, a nawet na deklarowanie zmiennych na początku funkcji. Zadeklarowanie zmiennej w miejscu, w którym jest potrzebna po raz pierwszy, jest zwykle lepszym rozwiązaniem i pomaga w tworzeniu czytelnego kodu. Typ zmiennej łatwiej dostrzec, gdy jej deklaracja znajduje się blisko pierwszego miejsca użycia tej zmiennej.

Ostrzeżenie

Dane przechowywane w zmiennych znajdują się w pamięci RAM. Dane te zostaną utracone po wyłączeniu komputera lub zakończeniu działania aplikacji, o ile programista wyraźnie nie zapisze ich w pamięci masowej, np. na dysku twardym.

Przechowywanie danych w pliku na dysku zostanie omówione w lekcji 27., zatytułowanej „Użycie strumieni w operacjach wejścia-wyjścia".

Zrozumienie zakresu zmiennej

Zwykłe zmienne mają doskonale zdefiniowany zakres, w którym są dostępne i mogą być używane. Gdy podejmiesz próbę użycia zmiennej poza jej zakresem, nazwa zmiennej nie będzie rozpoznana przez kompilator, a sam program nie zostanie skompilowany. Poza swoim zakresem zmienna pozostaje niezidentyfikowanym komponentem, o którym kompilator nic nie wie.

Aby lepiej zrozumieć zakres zmiennej, zmodyfikujemy program przedstawiony na listingu 3.1. W nowym programie znajdzie się funkcja `MultiplyNumbers()` odpowiedzialna za mnożenie liczb i zwrot wyniku. Zmodyfikowany program przedstawiono w listingu 3.2.

Listing 3.2. Pokazanie zakresu zmiennych

```
#include <iostream>
using namespace std;

void MultiplyNumbers ()
{
   cout << "Podaj pierwszą liczbę: ";
   int FirstNumber = 0;
   cin >> FirstNumber;

   cout << "Podaj drugą liczbę: ";
   int SecondNumber = 0;
   cin >> SecondNumber;

   // Mnożenie dwóch liczb i umieszczenie wyniku w zmiennej.
   int MultiplicationResult = FirstNumber * SecondNumber;

   // Wyświetlenie wyniku.
   cout << FirstNumber << " x " << SecondNumber;
   cout << " = " << MultiplicationResult << endl;
}
int main ()
{
   cout << "Ten program pomoże w mnożeniu dwóch liczb" << endl;

   // Wywołanie funkcji, która wykona całą pracę programu.
   MultiplyNumbers();

   // cout << FirstNumber << " x " << SecondNumber;
   // cout << " = " << MultiplicationResult << endl;

   return 0;
}
```

Wynik ▼

```
Ten program pomoże w mnożeniu dwóch liczb
Podaj pierwszą liczbę: 51
Podaj drugą liczbę: 24
51 x 24 = 1224
```

Analiza ▼

Program przedstawiony w listingu 3.2 działa dokładnie tak samo jak program z listingu 3.1 i generuje te same dane wyjściowe. Jedyna różnica pomiędzy wymienionymi programami polega na umieszczeniu poleceń powiązanych z operacją mnożenia w funkcji o nazwie `MultiplyNumbers()` wywoływanej przez `main()`. Zwróć uwagę, że zmienne `FirstNumber` i `SecondNumber` nie mogą być używane poza funkcją `MultiplyNumbers()`. Jeżeli usuniesz znaki komentarza na początku wierszy 28. i 29. w funkcji `main()`, program nie zostanie skompilowany.

Kompilacja kończy się niepowodzeniem, ponieważ zmienne `FirstNumber` i `SecondNumber` są lokalne, a ich zakres ograniczony do funkcji, w której zostały zadeklarowane, czyli w omawianym przykładzie do funkcji `MultiplyNumbers()`. Zmienna lokalna może być używana w funkcji po jej deklaracji, ale jedynie do końca danej funkcji. Nawias klamrowy } wskazuje koniec funkcji i jednocześnie ogranicza zakres zdefiniowanych w niej zmiennych. Po zakończeniu działania funkcji jej wszystkie zmienne lokalne są usuwane, a zajmowana przez nie pamięć zostaje zwolniona.

Po kompilacji programu zmienne zadeklarowane w funkcji `MultiplyNumbers()` istnieją jedynie w wymienionej funkcji, a próba ich użycia w funkcji `main()` uniemożliwi kompilację, ponieważ zmienne te nie istnieją w `main()`.

Ostrzeżenie

Jeżeli w funkcji `main()` zadeklarujesz inny zestaw zmiennych o takich samych nazwach, nie możesz oczekiwać, że będą miały wartości przypisane w funkcji `MultiplyNumbers()` zmiennym o takich samych nazwach.

Kompilator traktuje zmienne w funkcji `main()` jako zupełnie niezależne od innych, nawet jeśli mają takie same nazwy jak zmienne zadeklarowane w innych funkcjach. Dostępność i działanie zmiennych są po prostu ograniczone do ich zakresu.

Zmienne globalne

Gdyby zmienne użyte w funkcji `MultiplyNumbers()` programu przedstawionego w listingu 3.2 byłyby zadeklarowane poza zakresem funkcji `MultiplyNumbers()` zamiast w niej, wtedy można ich używać w funkcjach `main()` i `MultiplyNumbers()`. W listingu 3.3 pokazano przykład użycia zmiennych globalnych, które są w programach zmiennymi o największym zakresie.

Listing 3.3. Użycie zmiennych globalnych

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // Trzy globalne liczby całkowite.
 5: int FirstNumber = 0;
 6: int SecondNumber = 0;
 7: int MultiplicationResult = 0;
 8:
 9: void MultiplyNumbers ()
10: {
11:    cout << "Podaj pierwszą liczbę: ";
12:    cin >> FirstNumber;
13:
14:    cout << "Podaj drugą liczbę: ";
15:    cin >> SecondNumber;
16:
17:    // Mnożenie dwóch liczb i umieszczenie wyniku w zmiennej.
18:    MultiplicationResult = FirstNumber * SecondNumber;
19:
20:    // Wyświetlenie wyniku.
21:    cout << "Dane wyświetlone przez funkcję MultiplyNumbers(): ";
22:    cout << FirstNumber << " x " << SecondNumber;
23:    cout << " = " << MultiplicationResult << endl;
24: }
25: int main ()
26: {
27:    cout << "Ten program pomoże w mnożeniu dwóch liczb" << endl;
28:
29:    // Wywołanie funkcji, która wykona całą pracę programu.
30:    MultiplyNumbers();
31:
32:    cout << "Dane wyświetlone przez funkcję main(): ";
33:
34:    // Ten wiersz zostanie skompilowany i działa!
35:    cout << FirstNumber << " x " << SecondNumber;
36:    cout << " = " << MultiplicationResult << endl;
37:
38:    return 0;
39: }
```

Wynik ▼

```
Ten program pomoże w mnożeniu dwóch liczb
Podaj pierwszą liczbę: 51
Podaj drugą liczbę: 19
Dane wyświetlone przez funkcję MultiplyNumbers(): 51 x 19 = 969
Dane wyświetlone przez funkcję main(): 51 x 19 = 969
```

Analiza ▼

Dane wyjściowe programu przedstawionego w listingu 3.3 zawierają wynik operacji mnożenia przeprowadzanych w dwóch funkcjach, w których nie zostały zadeklarowane zmienne `FirstNumber`, `SecondNumber` i `MultiplicationResult`. Wymienione zmienne są globalne, zostały zadeklarowane w wierszach od 5. do 7. poza zakresem jakiejkolwiek funkcji. Zwróć uwagę na wiersze 23. i 36., w których wymienionych zmiennych użyto do wyświetlenia ich wartości. Szczególną uwagę zwróć na to, jak w funkcji `MultiplyNumbers()` następuje przypisanie wartości `MultiplicationResult` i ponowne jej użycie w funkcji `main()`.

Ostrzeżenie

Bezkrytyczne używanie zmiennych globalnych jest — ogólnie rzecz biorąc — uznawane za kiepską praktykę programistyczną.

Wartości zmiennym globalnym mogą być przypisywane w dowolnych funkcjach, mogą one znajdować się w nieprzewidywalnym stanie, zwłaszcza jeśli poszczególne moduły funkcji zostały utworzone przez innych programistów zespołu.

W listingu 3.3 eleganckim sposobem otrzymania w funkcji `main()` wyniku mnożenia jest zwrot tej wartości przez funkcję `MultiplyNumbers()`.

Typy zmiennych najczęściej używane w C++

W większości przedstawionych dotąd przykładów definiowaliśmy zmienne typu `int`, czyli liczby całkowite. Jednak programiści C++ mają do dyspozycji znacznie więcej typów danych bezpośrednio obsługiwanych przez kompilator. Wybór odpowiedniego typu danych jest bardzo ważny, podobnie jak wybór właściwego narzędzia do wykonania określonego zadania. Śrubokręt krzyżakowy nie nadaje się do wkręcenia zwykłej śruby, podobnie jak typ liczb całkowitych bez znaku nie może być wykorzystany do przechowywania liczb ujemnych. W tabeli 3.1 wymieniono różne typy zmiennych oraz naturę przechowywanych przez nie danych. Informacje te są bardzo ważne, jeśli chcesz tworzyć efektywne i niezawodne programy w C++.

W kolejnych punktach przedstawiono dokładniejsze omówienie najważniejszych typów.

Tabela 3.1. Typy zmiennych

Typ	Wartości
`bool`	prawda lub fałsz (`true` lub `false`)
`char`	256 różnych znaków
`unsigned short int`	od 0 do 65 535
`short int`	od –32 768 do 32 767
`unsigned long int`	od 0 do 4 294 967 295
`long int`	od –2 147 483 648 do 2 147 483 647
`unsigned long long`	od 0 do 18 446 744 073 709 551 615
`long long`	od –9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
`int (16 bitów)`	od –32 768 do 32 767
`int (32 bity)`	od –2 147 483 648 do 2 147 483 647
`unsigned int (16 bitów)`	od 0 do 65 535
`unsigned int (32 bity)`	od 0 do 4 294 967 295
`float`	od 1.2e-38 do 3.4e38
`double`	od 2.2e-308 do 1.8e308

Użycie typu bool do przechowywania wartości boolowskich

Język C++ oferuje typ `bool` przeznaczony specjalnie do przechowywania wartości boolowskich `true` i `false`, które w C++ są zarezerwowanymi słowami kluczowymi. Ten typ jest szczególnie użyteczny do przechowywania ustawień oraz flag, które mogą oznaczać włączony lub wyłączony, dostępny lub niedostępny itd.

Przykładowa deklaracja zainicjalizowanej zmiennej boolowskiej może mieć poniższą postać:

```
bool AlwaysOnTop = false;
```

Wyrażenie, które przyjmuje typ boolowski, może wyglądać następująco:

```
bool DeleteFile = (UserSelection == "yes");
// Przyjmie wartość true tylko wtedy, gdy wartością UserSelection jest "yes".
// W przeciwnym razie wartością całego wyrażenia będzie false.
```

Użycie typu char do przechowywania znaków

Typ `char` można wykorzystać do przechowywania pojedynczego znaku. Przykładowa deklaracja ma postać:

```
char UserInput = 'Y'; // Inicjalizacja zmiennej typu char wraz z wartością 'Y'.
```

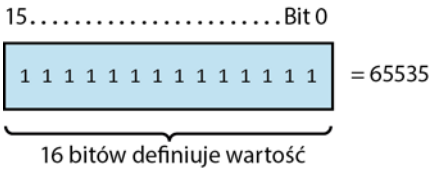
Zwróć uwagę na fakt, że pamięć składa się z bitów i bajtów. Bit może zawierać wartość 0 lub 1, natomiast bajt może zawierać wartości liczbowe zdefiniowane za pomocą wspomnianych bitów. Dlatego też, gdy pracujemy z danymi znakowymi lub przypisujemy dane znakowe, jak to przedstawiono w przykładzie, kompilator konwertuje znaki na wartości liczbowe, które mogą być umieszczone w pamięci. Wartości liczbowe znaków łacińskich od A do Z i od a do z, cyfr od 0 do 9, znaków specjalnych (np. DEL, Backspace) zostały określone w standardzie ASCII (ang. *American Standard Code for Information Interchange*).

Kiedy spojrzysz na przedstawioną w dodatku E tabelę kodów ASCII, przekonasz się, że znak *Y* przypisany zmiennej `UserInput` ma wartość dziesiętną wynoszącą 89. Dlatego też kompilator przechowuje wartość 89 w adresie pamięci zarezerwowanym dla zmiennej `UserInput`.

Koncepcja liczb ze znakiem i bez znaku

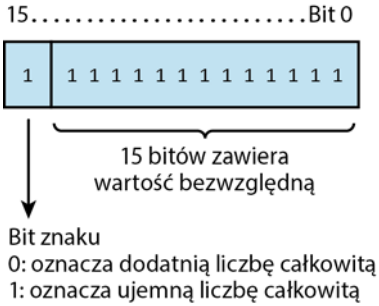
Znak wskazuje na liczbę dodatnią lub ujemną. Wszystkie liczby, z którymi pracujesz w komputerze, są przechowywane w pamięci w postaci bitów i bajtów. Adres w pamięci o wielkości 1 bajta zawiera 8 bitów. Każdy bit może przyjmować wartość 0 lub 1, a tym samym zawierać tylko jedną z dwóch wymienionych wartości. Dlatego też adres w pamięci o wielkości 1 bajta może zawierać maksymalnie 2^8 wartości, czyli 256 unikalnych wartości. Podobnie adres w pamięci o wielkości 16 bajtów może zawierać maksymalnie 2^{16} wartości, czyli 65 536 unikalnych wartości.

Jeżeli wspomniane wartości będą pozbawione znaku — czyli są jedynie dodatnie — wówczas 1 bajt może zawierać liczbę całkowitą o wartości od 0 do 255, natomiast 2 bajty — liczbę całkowitą o wartości od 0 do 65 535. Spójrz na tabelę 3.1 i zwróć uwagę, że typ `unsigned short` obsługuje omówiony zakres i zabiera 16 bitów pamięci. Wartości dodatnie wyrażone w bitach i bajtach można więc przedstawiać bardzo łatwo (patrz rysunek 3.1).



RYSUNEK 3.1. Ułożenie bitów w 16-bitowej liczbie typu unsigned short int

W jaki sposób, używając tej samej przestrzeni, określić liczbę ujemną? Jednym ze sposobów jest „poświęcenie" bitu i wykorzystanie go jako znaku w celu wskazania, czy wartości znajdujące się w pozostałych bitach są dodatnie, czy ujemne (patrz rysunek 3.2). Bit znaku jest określany mianem najbardziej znaczącego bitu (ang. *Most Significant Bit*, MSB), natomiast najmniej znaczący bit (ang. *Least Significant Bit*) służy do określania parzystości liczby. Dlatego też, gdy MSB zawiera informacje o znaku, przyjęło się, że 0 oznacza liczbę dodatnią, 1 liczbę ujemną, natomiast pozostałe bajty zawierają wartość bezwzględną.



RYSUNEK 3.2. Ułożenie bitów w 16-bitowej liczbie typu signed short int

Liczba ze znakiem zajmująca 8 bitów może zawierać wartości od –128 do 127, natomiast zajmująca 16 bitów może zawierać wartości od –32 768 do 32 767. Kiedy ponownie spojrzysz na tabelę 3.1, zauważysz, że typ `(signed) short` obsługuje dodatnie i ujemne liczby całkowite w przestrzeni 16-bitowej.

Liczby całkowite ze znakiem, czyli typy short, int, long i long long

Typy te mają różne wielkości, a tym samym mogą przechowywać odmienne zakresy wartości. Typ `int` jest najczęściej stosowanym typem, w większości kompilatorów zabiera 32 bity. Zawsze używaj odpowiedniego typu w projekcie, w zależności od maksymalnej wartości, jaka ma być przechowywana w danej zmiennej.

Deklaracja zmiennej typu ze znakiem jest bardzo prosta:

```
short int SmallNumber = -100;
int LargerNumber = -70000;
long PossiblyLargerThanInt = -70000; // Na niektórych platformach long jest typu int.
long long LargerThanInt = -70000000000;
```

Liczby całkowite bez znaku, czyli typy unsigned short, unsigned int, unsigned long i unsigned long long

W przeciwieństwie do odpowiadających im typów ze znakiem, typy liczb całkowitych bez znaku nie zawierają żadnych informacji o znaku, a tym samym mogą zapewnić obsługę dwukrotnie większych wartości dodatnich.

Deklaracja zmiennej typu bez znaku również jest bardzo prosta:

```
unsigned short int SmallNumber = 255;
unsigned int LargerNumber = 70000;
// Na niektórych platformach long jest typu int.
unsigned long PossiblyLargerThanInt = 70000;
unsigned long long LargerThanInt = 70000000000;
```

Uwaga

Zmiennych w postaci liczb całkowitych bez znaku używa się tylko wtedy, gdy spodziewane są jedynie wartości dodatnie. Dlatego też, jeśli potrzebujesz zmiennej do przechowywania np. liczby jabłek, nie używaj `int`, ale `unsigned int`. Druga z wymienionych zmiennych pozwala na przechowywanie dwukrotnie większej liczby wartości dodatnich.

Ostrzeżenie

Typ liczby całkowitej bez znaku może być nieodpowiedni w aplikacji finansowej, np. dla zmiennej przechowującej wartość dostępnych środków.

Typy zmiennoprzecinkowe float i double

Liczby zmiennoprzecinkowe to takie, które w szkole poznałeś jako liczby rzeczywiste. Mogą one być dodatnie lub ujemne, a ponadto zawierają wartości po przecinku dziesiętnym. Jeżeli więc w zmiennej C++ chcesz przechowywać wartość Pi (22 / 7, czyli 3.14), powinieneś użyć typu zmiennoprzecinkowego.

Deklaracja liczby typu zmiennoprzecinkowego odbywa się dokładnie tak samo jak typu `int`, np. w listingu 3.1. Typ `float` pozwala na przechowywanie wartości dziesiętnych i jest deklarowany w przedstawiony poniżej sposób:

```
float Pi = 3.14;
```

Podwójnej precyzji zmienna `float` (nazywana po prostu `double`) jest zdefiniowana jako:

```
double MorePrecisePi = 22 / 7;
```

Uwaga

> Wymienione w tabeli 3.1 typy danych są często określane mianem POD (ang. *Plain Old Data*). Kategoria POD obejmuje wymienione typy oraz agregacje (struktury, typy wyliczeniowe, unie i klasy).

Określanie wielkości zmiennej za pomocą sizeof

Wielkość zmiennej to ilość pamięci rezerwowanej przez kompilator, gdy programista zadeklaruje zmienną do przechowywania określonych danych. Wielkość zmiennej zależy od jej typu, a język C++ oferuje bardzo wygodny operator o nazwie `sizeof`, który podaje wyrażoną w bajtach wielkość zmiennej lub typu.

Sposób użycia operatora `sizeof` jest bardzo prosty. W celu ustalenia wielkości liczby całkowitej należy wywołać operator `sizeof` wraz z parametrem `int` (typ), jak to przedstawiono w listingu 3.4.

```
cout << "Rozmiar zmiennej typu int: " << sizeof (int);
```

Listing 3.4. Sprawdzenie rozmiarów standardowych typów zmiennych C++

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:    using namespace std;
 6:    cout << "Określenie rozmiarów pewnych wbudowanych typów zmiennych
       ↪C++" << endl;
 7:
 8:    cout << "Rozmiar zmiennej typu bool: " << sizeof(bool) << endl;
 9:    cout << "Rozmiar zmiennej typu char: " << sizeof(char) << endl;
10:    cout << "Rozmiar zmiennej typu unsigned short int: " <<
       ↪sizeof(unsigned short) << endl;
11:    cout << "Rozmiar zmiennej typu short int: " << sizeof(short) << endl;
12:    cout << "Rozmiar zmiennej typu unsigned long int: " <<
       ↪sizeof(unsigned long) << endl;
```

```
13:    cout << "Rozmiar zmiennej typu long: " << sizeof(long) << endl;
14:    cout << "Rozmiar zmiennej typu int: " << sizeof(int) << endl;
15:    cout << "Rozmiar zmiennej typu unsigned long long: "<<
       ↪sizeof(unsigned long long)<< endl;
16:    cout << "Rozmiar zmiennej typu long long: " << sizeof(long long) <<
       ↪endl;
17:    cout << "Rozmiar zmiennej typu unsigned int: " <<
       ↪sizeof(unsigned int) << endl;
18:    cout << "Rozmiar zmiennej typu float: " << sizeof(float) << endl;
19:    cout << "Rozmiar zmiennej typu double: " << sizeof(double) << endl;
20:
21:    cout << "Otrzymane dane wyjściowe zależą od kompilatora, sprzętu
       ↪i systemu operacyjnego" << endl;
22:
23:    return 0;
24: }
```

Wynik ▼

```
Określenie rozmiarów pewnych wbudowanych typów zmiennych C++
Rozmiar zmiennej typu bool: 1
Rozmiar zmiennej typu char: 1
Rozmiar zmiennej typu unsigned short int: 2
Rozmiar zmiennej typu short int: 2
Rozmiar zmiennej typu unsigned long int: 4
Rozmiar zmiennej typu long: 4
Rozmiar zmiennej typu int: 4
Rozmiar zmiennej typu unsigned long long: 8
Rozmiar zmiennej typu long long: 8
Rozmiar zmiennej typu unsigned int: 4
Rozmiar zmiennej typu float: 4
Rozmiar zmiennej typu double: 8
Otrzymane dane wyjściowe zależą od kompilatora, sprzętu i systemu
operacyjnego
```

Analiza ▼

Dane wyjściowe wygenerowane przez listing 3.4 pokazują wyrażoną w bajtach wielkość różnych typów na danej platformie: kompilatora, sprzętu i systemu operacyjnego. Przedstawione dane wyjściowe są wynikiem uruchomienia programu w trybie 32-bitowym (program skompilowany przez 32-bitowy kompilator) w 64-bitowym systemie operacyjnym. Warto pamiętać, że kompilator 64-bitowy prawdopodobnie wygenerowałby inne wyniki. Zdecydowałem się na użycie kompilatora 32-bitowego, aby otrzymany program mógł być uruchamiany w systemach zarówno 32-bitowych, jak i 64-bitowych. Otrzymane dane wyjściowe pokazują, że wielkość zmiennej jest taka sama dla typu ze znakiem i bez znaku,

jedyna różnica pomiędzy nimi polega na tym, że w zmiennej typu `unsigned` najbardziej znaczący bit (MSB) zawiera informacje o znaku.

Uwaga

> Wszystkie wielkości podane w danych wyjściowych są wyrażone w bajtach. Wielkość obiektu jest ważnym parametrem podczas alokowania dla niego pamięci, zwłaszcza gdy wspomniana alokacja jest przeprowadzana dynamicznie.

C++11

Użycie słowa kluczowego auto i pozostawienie kompilatorowi wyboru typu

Zdarzają się sytuacje, gdy typ zmiennej jest oczywisty, kiedy patrzymy na przypisaną jej wartość początkową. Jeśli np. zmienna została zainicjalizowana wraz z wartością `true`, typem zmiennej jest `bool`. W standardzie C++11 masz możliwość niepodawania typu zmiennej i zamiast tego użycia słowa kluczowego `auto`:

```
auto Flag = true;
```

Zadanie zdefiniowania dokładnego typu zmiennej `Flag` zostało zlecone kompilatorowi. Kompilator sprawdza naturę wartości inicjalizowanej zmiennej, a następnie wybiera dla niej najlepszy typ. W omawianym przykładzie jest oczywiste, że wartość `true` najlepiej pasuje do zmiennej typu `bool`. Dlatego też kompilator wybierze typ `bool` jako najodpowiedniejszy dla zmiennej `Flag` i wewnętrznie potraktuje wymienioną zmienną jako typ `bool`, co zostało przedstawione w listingu 3.5.

Listing 3.5. Użycie słowa kluczowego auto i opieranie się na możliwościach kompilatora w zakresie określania typu zmiennej

```
#include <iostream>
using namespace std;

int main()
{
    auto Flag = true;
    auto Number = 2500000000000;

    cout << "Flaga = " << Flag;
    cout << " , sizeof(Flag) = " << sizeof(Flag) << endl;
    cout << "Numer = " << Number;
    cout << " , sizeof(Number) = " << sizeof(Number) << endl;

    return 0;
}
```

Wynik ▼

```
Flaga = 1 , sizeof(Flag) = 1
Numer = 2500000000000 , sizeof(Number) = 8
```

Analiza ▼

Spójrz, jak zamiast zdefiniowania zmiennej `Flag` jako typu `bool` lub zmiennej `Number` jako typu `long long` podczas deklaracji wymienionych zmiennych w wierszach 6. i 7. użyte zostało słowo kluczowe `auto`. W ten sposób wybór typu zmiennej jest wykonywany przez inicjalizujący je kompilator. Operator `sizeof` został wykorzystany do sprawdzenia, czy kompilator wybrał spodziewane typy zmiennych. Dane wyjściowe wygenerowane przez listing potwierdzają wybór odpowiednich typów.

Uwaga

> Użycie słowa kluczowego `auto` wymaga inicjalizacji przez kompilator zmiennej wraz z wartością początkową, na podstawie której nastąpi określenie typu zmiennej.
>
> Kiedy nie inicjalizujesz zmiennej typu `auto` wraz z wartością początkową, otrzymasz błąd kompilacji.

Użycie funkcji `auto` ułatwia programowanie, zwłaszcza w sytuacjach, gdy typ zmiennej jest skomplikowany. Spójrz teraz na przykład deklaracji dynamicznej tablicy liczb całkowitych w postaci `std::vector`, nazwanej `MyNumbers`:

```
std::vector<int> MyNumbers;
```

Za pomocą przedstawionego poniżej kodu możesz uzyskać dostęp do tablicy lub przeprowadzić iterację elementów w tablicy, a następnie wyświetlić je:

```
for ( vector<int>::const_iterator Iterator = MyNumbers.begin();
      Iterator < MyNumbers.end();
      ++Iterator )
   cout << *Iterator << " ";
```

Dotychczas nie przedstawiono `std::vector` i pętli `for`, więc nie przejmuj się, jeśli przedstawiony poniżej fragment kodu nie jest w pełni jasny. Działanie kodu jest następujące: dla każdego elementu wektora, począwszy od `begin()` i skończywszy na `end()`, następuje wyświetlenie wartości za pomocą polecenia `cout`. Spójrz, jak złożony jest wiersz pierwszy deklarujący zmienną `Iterator` i przypisujący jej wartość początkową, taką jaka jest zwracana przez `begin()`.

Zmienna Iterator jest typu vector<int>::const_iterator, który jest całkiem skomplikowany dla programisty zarówno do odczytu, jak i zapisu. Zamiast podawać ten typ, programista może oprzeć się na typie zwracanym przez begin() i tym samym uprościć wiersz pierwszy pętli for do następującej postaci:

```
for( auto Iterator = MyNumbers.begin();
      Iterator < MyNumbers.end();
      ++Iterator )
   cout << *Iterator << " ";
```

Zwróć uwagę, jak skrócił się wiersz pierwszy w powyższej pętli. Kompilator sprawdza wartość początkową zmiennej Iterator zwracaną przez begin(), a następnie przypisuje zmiennej ten sam typ. To znacznie ułatwia tworzenie kodu C++, zwłaszcza podczas stosowania wzorców.

Użycie typedef do zastąpienia typu zmiennej

C++ pozwala na zastąpienie typu zmiennej nazwą, która może okazać się wygodniejsza w czasie pracy. Do tego celu służy słowo kluczowe typedef. Poniżej przedstawiono przykład, w którym programista chce nazwać unsigned int jako STRICTLY_POSITIVE_INTEGER:

```
typedef unsigned int STRICTLY_POSITIVE_INTEGER;
STRICTLY_POSITIVE_INTEGER PosNumber = 4532;
```

Podczas kompilacji wiersz pierwszy informuje kompilator, że STRICTLY_POSITIVE_INTEGER oznacza po prostu unsigned int. Na dalszych etapach, gdy kompilator napotka zdefiniowany typ STRICTLY_POSITIVE_INTEGER, zastąpi go unsigned int i będzie kontynuował kompilację.

Uwaga
Uwaga

> Słowo kluczowe typedef służące do zastępowania typu jest szczególnie użyteczne w przypadku skomplikowanych typów o złożonej składni, np. w trakcie stosowania szablonów.

Czym jest stała?

Wyobraź sobie, że tworzysz program przeznaczony do obliczenia pola oraz obwodu okręgu. Stosowane w programie wzory są następujące:

```
Pole = Pi * Promień * Promień
Obwód = 2 * Pi * Promień koła
```

W powyższych wzorach `Pi` jest stałą o wartości 22 / 7. Wartość `Pi` nie może być zmieniana w żadnym miejscu programu. Ponadto nie chcesz przypadkowego przypisania nieprawidłowej wartości `Pi`, np. przez niechcianą operację typu kopiuj-wklej lub znajdź-zastąp. Język C++ pozwala na zdefiniowanie `Pi` jako stałej, której wartości nie można zmienić po deklaracji. Innymi słowy, po zdefiniowaniu wartość stałej nie może być zmieniona. Próba przypisania wartości stałej powoduje wygenerowanie błędów w trakcie kompilacji.

Stałe są więc podobne do zmiennych w C++, ale z tym wyjątkiem, że nie mogą być modyfikowane. Podobnie jak zmienna, także stała zajmuje pewną ilość miejsca w pamięci i ma nazwę identyfikującą adres zarezerwowanej dla niej pamięci. Jak już wcześniej wspomniano, zawartość tej pamięci nie może być zmieniona. W języku C++ stała może być:

- dosłowną stałą,
- stałą zadeklarowaną za pomocą słowa kluczowego `const`,
- wyrażeniem stałej zdefiniowanym za pomocą słowa kluczowego `constexpr` (nowość w C++11),
- stałą typu wyliczeniowego zadeklarowaną za pomocą słowa kluczowego `enum`,
- zdefiniowaną stałą (są one uznawane za przestarzałe i nie zaleca się ich stosowania).

Dosłowne stałe

Jeszcze raz spójrz na listing 3.1 — to kod przedstawiający prosty program przeznaczony do mnożenia dwóch liczb. W listingu w poniższy sposób zadeklarowano liczbę całkowitą `FirstNumber`:

```
9: int FirstNumber = 0;
```

Liczbie całkowitej `FirstNumber` została przypisana wartość początkowa wynosząca zero. W omawianym przypadku zero jest częścią kodu, zostaje skompilowane w aplikacji, pozostaje niezmienne i jest nazywane dosłowną stałą. Dosłowne stałe mogą być różnych typów, takich jak wartość boolowska, liczba całkowita, ciąg tekstowy itd. W pierwszym programie C++ przedstawionym w tej książce (patrz listing 1.1) wyświetlony został komunikat *Witaj, świecie* za pomocą poniższego polecenia:

```
std::cout << "Witaj, świecie" << std::endl;
```

W tym przypadku *Witaj, świecie* jest dosłowną stałą w postaci ciągu tekstowego.

Deklarowanie zmiennych jako stałych przy użyciu const

Z praktycznego oraz z programistycznego punktu widzenia najważniejszy typ stałej w C++ jest deklarowany przez użycie słowa kluczowego `const` przed typem zmiennej. Ogólna deklaracja tego rodzaju stałej przedstawia się więc następująco:

```
const nazwa_typu nazwa_stałej;
```

Spójrz teraz na listing 3.6 przedstawiający prostą aplikację, która wyświetla wartość stałej o nazwie `Pi`.

Listing 3.6. Deklarowanie stałej o nazwie Pi

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:    using namespace std;
 6:
 7:    const double Pi = 22.0 / 7;
 8:    cout << "Wartość stałej Pi wynosi: " << Pi << endl;
 9:
10:    // Usunięcie komentarza z poniższego wiersza uniemożliwi kompilację programu.
11:    // Pi = 345;
12:
13:    return 0;
14: }
```

Wynik ▼

```
Wartość stałej Pi wynosi: 3.14286
```

Analiza ▼

Zwróć uwagę na wiersz 7. zawierający deklarację stałej `Pi`. Słowo kluczowe `const` zostało użyte w celu poinformowania kompilatora, że `Pi` jest stałą typu `double`. Jeśli usuniesz znaki komentarza na początku wiersza 11., w którym program próbuje przypisać wartość zmiennej zdefiniowanej jako stała, kompilacja zakończy się niepowodzeniem i zobaczysz komunikat informujący o braku możliwości przypisania wartości zmiennej będącej stałą. Dlatego też stałe do doskonałe rozwiązanie, gwarantujące, że pewne dane nie zostaną zmodyfikowane.

Uwaga

Dobrą praktyką programistyczną jest definiowanie jako `const` tych zmiennych, których wartość nie ulega zmianie. Użycie słowa kluczowego `const` gwarantuje, że wartość tej zmiennej nie zostanie przypadkowo zmieniona w aplikacji.

To jest szczególnie użyteczne w środowisku, w którym współpracuje ze sobą wielu programistów.

Stałe są użyteczne podczas deklaracji wielkości tablic statycznych w trakcie kompilacji. W listingu 4.2 przedstawionym w lekcji 4., zatytułowanej „Tablice i ciągi tekstowe", znajdziesz przykład demonstrujący użycie `const int` do określenia wielkości tablicy.

C++11

Deklarowanie stałych za pomocą constexpr

Koncepcja `constexpr` istniała w C++ od zawsze, nawet przed wprowadzeniem standardu C++11, choć nie została sformalizowana w postaci słowa kluczowego. Spójrz na listing 3.5, w którym wyrażenie 22.0 / 7 jest wyrażeniem stałej obsługiwanym również przez kompilatory wydane przed 2011 rokiem. Jednak wspomniane kompilatory nie pozwalają na definiowanie funkcji obliczanych w trakcie kompilacji. W standardzie C++11 możesz użyć poniższej definicji:

```
constexpr double GetPi() {return 22.0 / 7;}
```

Funkcja `GetPi()` użyta w połączeniu ze stałą powoduje otrzymanie poprawnego polecenia:

```
constexpr double TwicePi() {return 2 * GetPi();}
```

Na początku różnica pomiędzy `const` i `constexpr` wydaje się niewielka. Jednak z punktu widzenia kompilatora i aplikacji wprowadzone zostały nowe rodzaje optymalizacji. Drugie wyrażenie (bez `constexpr`) nie będzie w kompilatorach wydanych przed 2011 rokiem obliczane w trakcie działania aplikacji. Natomiast w kompilatorach zgodnych ze standardem C++11 zostanie obliczone podczas kompilacji, co spowoduje szybsze działanie aplikacji.

Uwaga

W czasie pisania tej książki słowo kluczowe `constexpr` nie było obsługiwane przez kompilator Microsoft Visual C++ Express, natomiast jest obsługiwane przez kompilator GNU g++.

Stałe typu wyliczeniowego

Zdarzają się sytuacje, gdy dana zmienna może przyjmować jedynie określony zestaw wartości. W takich przypadkach możesz nie chcieć, aby np. w kolorach tęczy znajdował się turkusowy lub w kierunkach kompasu znajdowała się wartość „w lewo". W obu wymienionych sytuacjach potrzebujesz typu zmiennej, której wartości będą ograniczone do wskazanego zestawu wartości. Stałe typu wyliczeniowego są dokładnie tym narzędziem, którego w wymienionych przypadkach potrzebujesz. Utworzenie stałej typu wyliczeniowego odbywa się za pomocą słowa kluczowego `enum`.

Przykładowo przedstawiona poniżej stała typu wyliczeniowego określa kolory tęczy:

```
enum RainbowColors
{
   Fioletowy = 0,
   Indygo,
   Niebieski,
   Zielony,
   Żółty,
   Pomarańczowy,
   Czerwony
};
```

Oto inny przykład zawierający strony świata:

```
enum CardinalDirections
{
   Północ,
   Południe,
   Wschód,
   Zachód
};
```

Zwróć uwagę, że stałe typu wyliczeniowego mogą być używane jako typy zmiennych akceptujące wartości zdefiniowane w danym typie wyliczeniowym. Dlatego też, jeśli definiujesz zmienną zawierającą kolory tęczy, możesz ją zadeklarować w przedstawiony poniżej sposób:

```
RainbowColors MyWorldsColor = Niebieski; // Wartość początkowa.
```

W powyższym wierszu kodu zadeklarowano stałą typu wyliczeniowego o nazwie `MyWorldsColor` i typie `RainbowColors`. Zmienna może zawierać jedynie kolory zdefiniowane w stałej typu wyliczeniowego `RainbowColors`.

Uwaga

Podczas deklarowania stałej typu wyliczeniowego wartości typu wyliczeniowego (np. `Fioletowy`) są przez kompilator konwertowane na liczby całkowite. Każda wartość w typie wyliczeniowym jest o jeden większa od wartości poprzedniego elementu typu wyliczeniowego. Masz możliwość wskazania wartości początkowej, ale jeśli tego nie zrobisz, kompilator przyjmie założenie, że wynosi ona zero. W przypadku omawianego kodu będzie to oznaczało, że elementowi `Północ` zostanie przypisana wartość zero.

Jeśli chcesz, przez inicjalizację możesz wyraźnie podać wartości dla wszystkich elementów stałej typu wyliczeniowego.

W listingu 3.7 pokazano, jak stałe typu wyliczeniowego są używane do przechowywania informacji o czterech stronach świata, a podczas inicjalizacji wartość początkową przypisano pierwszej z nich.

Listing 3.7. Użycie wartości typu wyliczeniowego w celu wskazania kierunku wiatru

```
#include <iostream>
using namespace std;

enum CardinalDirections
{
   North = 25,
   South,
   East,
   West
};

int main()
{
    cout << "Wyświetlenie kierunków i ich wartości symbolicznych" << endl;
    cout << "Północ: " << North << endl;
    cout << "Południe: " << South << endl;
    cout << "Wschód: " << East << endl;
    cout << "Zachód: " << West << endl;

   CardinalDirections WindDirection = South;
   cout << "Zmienna WindDirection = " << WindDirection << endl;

   return 0;
}
```

Wynik ▼

```
Wyświetlenie kierunków i ich wartości symbolicznych
Północ: 25
Południe: 26
Wschód: 27
Zachód: 28
Zmienna WindDirection = 26
```

Analiza ▼

Zwróć uwagę, jak cztery strony świata zdefiniowaliśmy jako stałe typu wyliczeniowego, ale tylko pierwsza (`Północ`) otrzymała wartość początkową równą 25 (patrz wiersz 6.). W ten sposób mamy gwarancję, że wartościami kolejnych stałych będą liczby 26, 27 i 28, co zostało przedstawione w danych wyjściowych. W wierszu 20. została utworzona zmienna typu `CardinalDirections`, której przypisano wartość początkową wynoszącą `Południe`. Podczas jej wyświetlania na ekranie w wierszu 21. kompilator sprawdza wartość przypisaną stałej `Południe`, która wynosi 26.

Wskazówka

Warto spojrzeć na listingi 6.4 i 6.5 znajdujące się w lekcji 6., zatytułowanej „Sterowanie przebiegiem działania programu". W wymienionych listingach użyto słowa kluczowego `enum` w celu wyliczenia dni tygodnia, a także do przeprowadzenia przetwarzania warunkowego do wskazania nazwy dnia znajdującego się po wybranym przez użytkownika.

Definiowanie stałych za pomocą dyrektywy #define

Przede wszystkim nie używaj tej dyrektywy podczas tworzenia programu od początku. Jedynym powodem, dla którego w tej książce przedstawiono analizę definicji stałych za pomocą `#define`, jest umożliwienie zrozumienia pewnych starych programów, gdzie stała `Pi` została zdefiniowana za pomocą poniższej składni:

```
#define Pi 3.14286
```

To jest makro preprocesora i jego działanie jest następujące: wszystkie wystąpienia `Pi` w kodzie źródłowym będą w trakcie kompilacji zastąpione przez wartość `3.14286`. Zwróć uwagę na fakt, że jest to wykonywana przez

preprocesor operacja zastąpienia tekstu (czytaj: zastąpienie nieinteligentne). Kompilator nic nie wie i nie przejmuje się rzeczywistym typem wymienionej zmiennej.

> Definiowanie stałych za pomocą makra preprocesora (`#define`) jest uznawane za przestarzałe i nie należy stosować takiego rozwiązania.

Ostrzeżenie

Nazwy zmiennych i stałych

Istnieje wiele różnych sposobów i wiele odmiennych konwencji nadawania nazw zmiennym. Niektórzy programiści preferują stosowanie kilkuznakowych prefiksów wskazujących typ danej zmiennej. Oto przykład:

```
bool bIsLampOn = false;
```

W powyższym polecenie `b` jest prefiksem nadanym przez programistę w celu wskazania zmiennej jako typu `bool`. Taki zapis jest nazywany notacją węgierską, początkowo został opracowany i był promowany przez Microsoft. Jednak C++ jest językiem stosującym ściśle określone typy, więc kompilator zna typ zmiennej nie dzięki prefiksowi w jej nazwie, ale z powodu wskazania typu (tutaj `bool`). Dlatego też obecnie zaleca się, aby programiści nie stosowali notacji węgierskiej. Konieczne jest, aby nazwa zmiennej pozostała zrozumiała, nawet jeśli oznacza to wydłużenie jej nazwy. Przyjmujemy założenie, że przedstawiona powyżej zmienna boolowska była użyta w programie elektroniki samochodu. Nieco lepszą nazwą dla wspomnianej zmiennej będzie:

```
bool IsHeadLampOn = false;
```

Zwróć uwagę, że oba przedstawione warianty są lepsze i zalecane do stosowania zamiast poniższej formy:

```
bool b = false;
```

Tego rodzaju nieopisowych zmiennych należy unikać za wszelką cenę.

TAK	NIE
Nadawaj zmiennym opisowe nazwy, nawet jeśli oznacza to wydłużenie nazwy zmiennej. Upewnij się, że nazwa zmiennej właściwie odzwierciedla jej przeznaczenie. Spróbuj postawić się w sytuacji osoby, która nie zna Twojego kodu źródłowego, i zastanów się, czy nazwa danej zmiennej będzie miała dla niej sens. Sprawdź, czy zespół, w którym pracujesz, ma zdefiniowane określone konwencje nazw, i stosuj się do nich.	Nie nadawaj zmiennym nazw zbyt krótkich lub składających się z pojedynczego znaku. Nie używaj w nazwach zmiennych egzotycznych akronimów znanych jedynie Tobie. Nie nadawaj zmiennym nazw w postaci zarezerwowanych słów kluczowych C++, ponieważ uniemożliwi to kompilację kodu.

Słowa kluczowe, których nie można używać jako nazw zmiennych lub stałych

Pewne słowa są zarezerwowane w języku C++ i nie można ich używać jako nazw zmiennych. Wspomniane słowa kluczowe mają znaczenie specjalne dla kompilatora C++. Do tych słów kluczowych zaliczamy `if`, `while`, `for` i `main`. Listę słów kluczowych zdefiniowanych w C++ przedstawiono w tabeli 3.2 oraz w dodatku B, zatytułowanym „Słowa kluczowe C++”. Używany przez Ciebie kompilator może mieć dodatkowe słowa zarezerwowane, warto zatem sprawdzić to w jego dokumentacji.

Podsumowanie

W tej lekcji dowiedziałeś się nieco o sposobie użycia pamięci w celu tymczasowego przechowywania zmiennych i stałych. Wiesz już, że zmienne mają wielkość określoną przez ich typ, a operator `sizeof` można wykorzystać do sprawdzenia wspomnianej wielkości zmiennej. Powinieneś też już wiedzieć o istnieniu różnych typów zmiennych, np. `bool` i `int`, oraz o ich przeznaczeniu do przechowywania różnego typu danych. Odpowiedni wybór typu zmiennej ma bardzo ważne znaczenie dla efektywnego programowania, a wybór zbyt małej zmiennej może skutkować dziwnymi błędami lub wystąpieniem sytuacji

Tabela 3.2. Słowa kluczowe C++

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	
Ponadto zarezerwowane są poniższe słowa:			
And	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

przepełnienia. Poznałeś także wprowadzone w standardzie C++11 słowo kluczowe `auto`, które pozwala na pozostawienie kompilatorowi podjęcia decyzji o typie danych na podstawie wartości inicjalizacyjnej danej zmiennej.

Poznałeś również różne typy stałych oraz sposoby użycia najważniejszych z nich za pomocą słów kluczowych `const` i `enum`.

Pytania i odpowiedzi

Pytanie: Jaki jest w ogóle sens definiowania stałych, jeśli zamiast nich można używać zwykłych zmiennych?

Odpowiedź: Stałe, zwłaszcza zadeklarowane za pomocą słowa kluczowego `const`, pozwalają na poinformowanie kompilatora, że dana zmienna

nie powinna ulec zmianie. Kompilator zagwarantuje więc, że stałej nigdy nie będzie przypisana inna wartość, nawet jeśli programista się postara i przypadkowo spróbuje nadpisać wartość stałej. Jeśli wiadomo, że wartość danej zmiennej nie powinna ulec zmianie, to zadeklarowanie zamiast niej stałej jest dobrą praktyką programistyczną i przyczynia się do zwiększenia jakości aplikacji.

Pytanie: Dlaczego powinienem inicjalizować wartość zmiennej?

Odpowiedź: Jeżeli nie zainicjalizujesz zmiennej, nie będziesz wiedział, jaką zawiera wartość początkową. Wspomniana wartość początkową to po prostu zawartość danego adresu pamięci zarezerwowanego dla tej zmiennej. Inicjalizacja przeprowadzana następująco

```
int MyFavouriteNumber = 0;
```

powoduje tuż po utworzeniu zmiennej zapisanie wskazanej wartości początkowej (tutaj zero) w adresie pamięci zarezerwowanym dla zmiennej `MyFavouriteNumber`. Istnieją sytuacje, gdy stosowane jest przetwarzanie warunkowe na podstawie wartości zmiennej (często polega na sprawdzeniu dotyczącym wartości niezerowej). Tego rodzaju logika nie będzie działała niezawodnie bez inicjalizacji zmiennej, ponieważ nieprzypisana lub niezainicjalizowana zmienna może zawierać śmieci, często wartości niezerowe lub zupełnie przypadkowe.

Pytanie: Dlaczego C++ daje mi możliwość użycia `short int`, `int` i `long int`? Dlaczego nie mogę zawsze użyć po prostu typu liczby całkowitej pozwalającego na przechowywanie największej liczby?

Odpowiedź: C++ jest językiem programowania używanym do tworzenia wielu różnych aplikacji, z których wiele działa w urządzeniach oferujących niewielką moc obliczeniową lub małą ilość pamięci. Zwykły telefon komórkowy starszego typu (nie smartfon) to jeden z przykładów urządzenia, w którym zasoby dostępnej mocy obliczeniowej i pamięci są ograniczone. W takim przypadku programista może bardzo często zaoszczędzić pamięć lub nieco przyśpieszyć działanie aplikacji (bądź i jedno, i drugie) przez wybór odpowiedniego rodzaju zmiennej, jeśli nie musi korzystać z dużych wartości. Podczas tworzenia aplikacji dla zwykłego komputera lub smartfona wybór odpowiedniego typu zmiennej liczb całkowitych zwykle nie przynosi dużej zmiany w wydajności aplikacji lub oszczędności pamięci, a czasami zupełnie nie powoduje żadnej różnicy.

Pytanie: Dlaczego nie powinienem zbyt często używać zmiennych globalnych? Czy nie jest prawdą, że są one użyteczne w aplikacji i pomagają oszczędzić czas, który w przeciwnym razie stracę na przekazywanie wartości pomiędzy funkcjami?

Odpowiedź: Zmienne globalne mogą być przypisywane i odczytywane globalnie. To drugie jest problemem, ponieważ zmienną można zmienić globalnie. Przyjmujemy założenie, że wraz z kilkoma innymi programistami pracujesz nad projektem. Liczby całkowite i inne zmienne zadeklarowałeś jako globalne. Jeżeli inny programista w zespole przypadkowo zmieni w kodzie wartość zdefiniowanej liczby całkowitej — zmiana może być wprowadzona nawet w innym pliku *.cpp* niż używany przez Ciebie — niezawodność kodu będzie naruszona. Poświęcenie kilku sekund lub minut nie powinno być więc przesłanką do stosowania zmiennych globalnych. Dzięki ich unikaniu możesz zapewnić większą stabilność tworzonego kodu źródłowego.

Pytanie: Język C++ daje mi możliwość deklarowania liczb całkowitych bez znaku, które przechowują jedynie wartości dodatnie i zero. Co się stanie, jeśli zmienna zadeklarowana jako unsigned int będzie miała wartość zero, a ja odejmę od niej inną wartość?

Odpowiedź: Spotkasz się z efektem zawinięcia. Dekrementacja o jeden zmiennej typu `unsigned int` o wartości zero spowoduje zawinięcie wartości zmiennej i przypisanie jej najwyższej możliwej wartości! Sprawdź tabelę 3.1, a przekonasz się, że typ `unsigned int` pozwala na przechowywanie wartości z zakresu od 0 do 65535. Zadeklaruj więc zmienną typu `unsigned int`, a następnie przeprowadź dekrementację:

```
unsigned short MyShortInt = 0; // Wartość początkowa.
MyShortInt = MyShortInt - 1; // Dekrementacja o 1.
std::cout << MyShortInt << std::endl; // Dane wyjściowe: 65535!
```

Zwróć uwagę, że nie jest to problem związany z typem `unsigned short`, ale raczej wynikający ze sposobu jego użycia. Liczba całkowita bez znaku (zarówno krótka, jak i długa) nie jest przeznaczona do użycia z wartościami ujemnymi, co wynika ze specyfikacji. Jeżeli zawartość `MyShortInt` będzie użyta do dynamicznej alokacji liczby bajtów, niewielki błąd pozwalający na dekrementację wartości zero spowoduje alokację 64 kB! Co gorsza, jeśli `MyShortInt` będzie używana jako indeks umożliwiający dostęp do adresów w pamięci, istnieje niebezpieczeństwo, że aplikacja spróbuje uzyskać dostęp do niezarezerwowanej dla niej pamięci i nastąpi awaria programu.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Jaka jest różnica pomiędzy liczbą całkowitą typu `signed` i `unsigned`?
2. Dlaczego nie powinieneś używać dyrektywy `#define` do deklarowania stałej?
3. Dlaczego należy inicjalizować zmienne?
4. Spójrz na poniższy typ `enum`. Jaką wartość ma `DAMA`?

   ```
   enum KARTY {AS, WALET, DAMA, KRÓL};
   ```

5. Co złego jest w poniższej nazwie zmiennej?

   ```
   int Integer = 0;
   ```

Ćwiczenia

1. Zmodyfikuj typ `enum KARTY` przedstawiony w quizie w taki sposób, aby wartość `DAMA` wynosiła 45.
2. Utwórz program pokazujący, że rozmiary liczby całkowitej zwykłej i typu `unsigned` są takie same i mają mniejszy rozmiar niż liczba całkowita typu `long`.
3. Utwórz program obliczający powierzchnię i obwód koła, którego promień jest podawany przez użytkownika.
4. Jeżeli w ćwiczeniu 3. powierzchnia i obszar będą przechowywane w zmiennych typu `int`, jaki to będzie miało wpływ na dane wyjściowe?
5. **Łowcy błędów:** Co jest złego w poniższej inicjalizacji zmiennej:

   ```
   auto Integer;
   ```

Lekcja 4

Tablice i ciągi tekstowe

W poprzednich lekcjach deklarowaliśmy zmienne zawierające pojedyncze wartości typu int, char lub string. Często chcemy jednak deklarować zbiory obiektów, np. 20 wartości typu int lub kilka obiektów typu Cat.

Z tej lekcji dowiesz się:

- **dowiesz się, czym są tablice oraz jak się je deklaruje i jak ich używa,**
- **dowiesz się, czym są ciągi tekstowe i jak je tworzyć za pomocą tablic,**
- **poznasz krótkie wprowadzenie do klasy `std::string`.**

Czym jest tablica?

Słownikowa definicja *tablicy* całkiem dobrze oddaje jej znaczenie. Według słownika Merriam-Webster *tablica* to „grupa elementów tworzących całość, np. zestaw paneli słonecznych".

Poniżej wymieniono pewne cechy charakterystyczne tablicy:

- tablica jest kolekcją elementów,
- wszystkie elementy znajdujące się w tablicy są tego samego typu,
- tablica tworzy kompletną całość.

W języku C++ tablice umożliwiają przechowywanie w pamięci pewnego typu elementów danych, które są uporządkowane.

Kiedy trzeba użyć tablicy?

Wyobraź sobie, że tworzysz program, w którym użytkownik podaje 5 liczb całkowitych wyświetlanych następnie na ekranie. Jednym z rozwiązań możliwych do zastosowania w programie jest zadeklarowanie 5 oddzielnych zmiennych typu liczba całkowita (`int`) i ich wykorzystanie do przechowywania oraz wyświetlania wartości. Wspomniana deklaracja mogłaby przedstawiać się następująco:

```
int FirstNumber = 0;
int SecondNumber = 0;
int ThirdNumber = 0;
int FourthNumber = 0;
int FifthNumber = 0;
```

Jeżeli później będziesz chciał, aby program przechowywał i wyświetlał 500 liczb całkowitych, wtedy przy użyciu powyższego systemu konieczne będzie zadeklarowanie 500 zmiennych przeznaczonych do obsługi liczb całkowitych. Przy odrobinie czasu i cierpliwości wspomniane zadanie da się wykonać. Wyobraź sobie, co będzie, jeśli zleceniodawca programu poprosi o zapewnienie obsługi 500 000 liczb całkowitych zamiast 5. Co wówczas zrobisz?

Od samego początku powinieneś zastosować odpowiednie i inteligentne rozwiązanie polegające na zadeklarowaniu tablicy złożonej z 5 elementów zainicjalizowanych wraz z wartością zero. Oto przykład:

```
int MyNumbers [5] = {0};
```

Następnie, jeśli zostaniesz poproszony o zapewnienie obsługi 500 000 liczb całkowitych, wtedy tablicę możesz bardzo szybko przeskalować, np. tak:

```
int ManyNumbers [500000] = {0};
```

Poniżej przedstawiono przykład zdefiniowania tablicy przechowującej 5 znaków:

```
char MyCharacters [5];
```

Tego rodzaju tablice są nazywane *tablicami statycznymi*, ponieważ liczba przechowywanych przez nie elementów, jak również pamięć zużywana przez tablicę są stałe i ustalone w trakcie kompilacji.

Deklarowanie i inicjalizacja tablic statycznych

W poprzednich wierszach kodu zadeklarowano tablicę o nazwie `MyNumbers` zawierającą 5 elementów typu `int` — czyli liczb całkowitych — zainicjalizowanych z wartością zero. Deklaracja tablicy w C++ jest przeprowadzana za pomocą przedstawionej poniżej prostej składni:

```
typ-elementu nazwa-tablicy [liczba-elementów] = {opcjonalne wartości
początkowe}
```

Istnieje możliwość zadeklarowania tablicy i inicjalizacji jej zawartości przez podanie poszczególnych elementów. Przykładowo przedstawiona poniżej tablica została zadeklarowana z 5 elementami, a każdy z nich jest zainicjalizowany z inną wartością:

```
int MyNumbers [5] = {34, 56, -21, 5002, 365};
```

Pierwsze kilka elementów tablicy można zainicjalizować z różnymi wartościami, a następujące będą automatycznie miały przypisane wartości zero, np.:

```
int MyNumbers [5] = {34, 56}; // Zainicjalizowane elementy: 24, 56, 0, 0, 0.
```

Istnieje również możliwość zainicjalizowania wszystkich elementów tablicy wraz z wartościami zero, np.:

```
int MyNumbers [5] = {0};
```

Ponadto istnieje możliwość częściowej inicjalizacji elementów tablicy. Oto przykład:

```
int MyNumbers [5] = {34, 56}; // Inicjalizacja pierwszych dwóch elementów.
```

Wielkość tablicy (tzn. liczbę znajdujących się w niej elementów) także można zdefiniować w postaci stałej, która następnie będzie użyta w definicji tablicy:

```
const int ARRAY_LENGTH = 5;
int MyNumbers [ARRAY_LENGTH] = {34, 56, -21, 5002, 365};
```

Tego rodzaju rozwiązanie jest szczególnie użyteczne, gdy trzeba uzyskać dostęp i używać wielkości tablicy w wielu miejscach, np. podczas iteracji elementów. Wówczas zamiast podawać wielkość tablicy we wszystkich wymaganych miejscach, wystarczy poprawnie zainicjalizować tę wartość w deklaracji `const int`.

Uwaga

> W częściowo zainicjalizowanych tablicach istnieje możliwość, że pewne kompilatory zainicjalizują zignorowane przez Ciebie elementy i przypiszą im wartość początkową wynoszącą zero.

Jeżeli znane są wartości początkowe elementów tablicy, wtedy można pominąć liczbę wskazującą wielkość tablicy, np.:

```
int MyNumbers [] = {2011, 2052, -525};
```

W powyższym wierszu kodu nastąpiło utworzenie tablicy zawierającej 3 liczby całkowite: 2011, 2052 i –525.

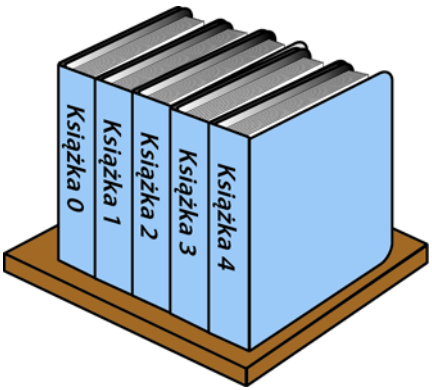
Uwaga

> Zadeklarowane dotąd tablice są *tablicami statycznymi*, ponieważ ich wielkość jest stała i zdefiniowana przez programistę w trakcie kompilacji. Tego rodzaju tablica nie może przechowywać ilości danych większej niż wskazana przez programistę. Ponadto, jeśli nie będzie wykorzystana w pełni, to i tak zajmie całą zarezerwowaną dla niej ilość pamięci.

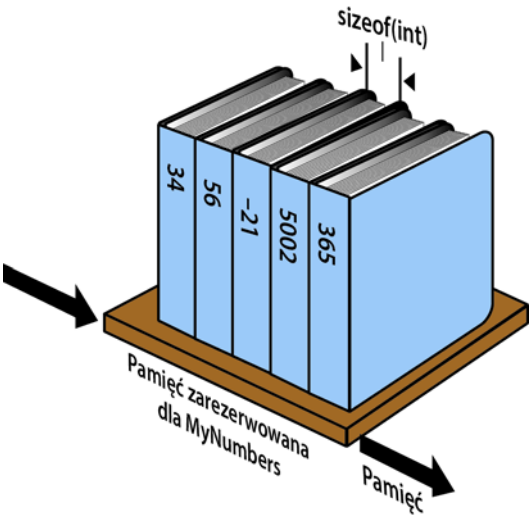
Jak w tablicy przechowywane są dane?

Wyobraź sobie książki ułożone na półce, jedna obok drugiej. To jest przykład jednowymiarowej tablicy, ponieważ jest ona rozbudowywana tylko w jednym kierunku, tzn. zmienia się liczba przechowywanych książek. Każda książka jest elementem tablicy, a półka przypomina pamięć zarezerwowaną do przechowywania danej kolekcji książek (patrz rysunek 4.1).

To nie jest błąd, że książki zostały ponumerowane od zera. Jak się wkrótce przekonasz, w języku C++ wartości indeksów rozpoczynają się od 0, a nie od 1. Podobnie jak pięć książek na półce, używana wcześniej tablica `MyNumbers` zawiera 5 liczb całkowitych i można ją przedstawić tak, jak pokazano na rysunku 4.2.



RYSUNEK 4.1. Książki na półce, przykład tablicy jednowymiarowej



RYSUNEK 4.2. Umieszczona w pamięci struktura tablicy MyNumbers przechowującej 5 liczb całkowitych

Zwróć uwagę, że pamięć zajmowana przez tablicę składa się z pięciu bloków o takiej samej wielkości określanej przez typ danych przechowywanych w tablicy, w omawianym przykładzie to liczby całkowite. Temat wielkości liczb całkowitych został poruszony w lekcji 3., zatytułowanej „Zmienne i stałe”. Wielkość pamięci zarezerwowanej przez kompilator dla tablicy `MyNumbers` wynosi więc `sizeof(int) * 5`. Ogólnie rzecz biorąc, wyrażona w bajtach ilość pamięci rezerwowanej przez kompilator dla tablicy wynosi:

```
bajty używane przez tablicę = sizeof(typ-elementu) * liczba elementów
```

Uzyskanie dostępu do danych przechowywanych w tablicy

Dostęp do elementów tablicy jest możliwy za pomocą indeksu, który rozpoczyna się od zera. Pierwszy element tablicy ma indeks o wartości zero. Dlatego też pierwsza liczba całkowita w tablicy `MyNumbers` jest dostępna jako `MyNumbers[0]`, druga jako `MyNumbers[1]` itd. Piąta liczba jest dostępna jako `MyNumbers[4]`. Innymi słowy, wartość indeksu ostatniego elementu tablicy to zawsze (wielkość tablicy –1).

Kiedy chcesz uzyskać dostęp do elementu o indeksie `N`, kompilator używa adresu pamięci pierwszego elementu (o indeksie zero) jako punktu początkowego, a następnie przeskakuje o *N* elementów. Po dodaniu wartości przesunięcia obliczonej jako `N*sizeof(element)` otrzymuje adres zawierający żądany element. Kompilator C++ nie sprawdza, czy indeks prowadzi do elementu znajdującego się w tablicy. Możesz więc spróbować uzyskać dostęp do elementu o indeksie 1001 w tablicy zawierającej jedynie 10 elementów, ale to spowoduje zmniejszenie bezpieczeństwa i stabilności programu. Obowiązek upewnienia się, że kod nie spróbuje uzyskać dostępu do elementów poza tablicą, należy wyłącznie do programisty.

Ostrzeżenie

> Próba uzyskania dostępu do nieistniejących elementów tablicy skutkuje trudnym do przewidzenia zachowaniem programu. W wielu przypadkach program po prostu ulegnie awarii. Należy za wszelką cenę unikać prób uzyskania dostępu do elementów poza tablicą.

W listingu 4.1 przedstawiono przykład zadeklarowania tablicy liczb całkowitych, zainicjalizowania elementów wraz z wartościami początkowymi, a następnie uzyskania dostępu do elementów w celu ich wyświetlenia na ekranie.

Listing 4.1. Deklarowanie tablicy liczb całkowitych i uzyskanie dostępu do jej elementów

```
0:  #include <iostream>
1:
2:  using namespace std;
3:
4:  int main ()
5:  {
6:     int MyNumbers [5] = {34, 56, -21, 5002, 365};
7:
```

```
 8:     cout << "Pierwszy element o indeksie 0: " << MyNumbers [0] << endl;
 9:     cout << "Drugi element o indeksie 1: " << MyNumbers [1] << endl;
10:     cout << "Trzeci element o indeksie 2: " << MyNumbers [2] << endl;
11:     cout << "Czwarty element o indeksie 3: " << MyNumbers [3] << endl;
12:     cout << "Piąty element o indeksie 4: " << MyNumbers [4] << endl;
13:
14:     return 0;
15: }
```

Wynik ▼

```
Pierwszy element o indeksie 0: 34
Drugi element o indeksie 1: 56
Trzeci element o indeksie 2: -21
Czwarty element o indeksie 3: 5002
Piąty element o indeksie 4: 365
```

Analiza ▼

W wierszu 6. została zadeklarowana tablica 5 liczb całkowitych wraz z podanymi wartościami. W kolejnych wierszach wspomniane liczby całkowite są wyświetlane na ekranie za pomocą polecenia `cout` i odwołania do indeksu danego elementu tablicy `MyNumbers`.

Uwaga

Aby pomóc w przyswojeniu koncepcji indeksu rozpoczynającego się od zera podczas uzyskiwania dostępu do elementów tablicy, wiersze kodu, począwszy od listingu 4.1, są numerowane od zera.

Modyfikacja danych przechowywanych w tablicy

W poprzednim listingu dane zdefiniowane przez użytkownika nie zostały umieszczone w tablicy. Składnia pozwalająca na przypisanie liczby całkowitej elementowi używanej tablicy jest całkiem podobna do składni przypisania liczby całkowitej zmiennej typu `int`.

Przykładowo przypisanie wartości 2011 zmiennej typu `int` odbywa się następująco:

```
int AnIntegerValue;
AnIntegerValue = 2011;
```

Z kolei przypisanie wartości 2011 czwartemu elementowi tablicy odbywa się następująco:

```
MyNumbers [3] = 2011; // Przypisanie wartości 2011 czwartemu elementowi tablicy.
```

W listingu 4.2 zademonstrowano użycie stałych podczas deklarowania wielkości tablicy. Pokazano również, jak można uzyskać dostęp do poszczególnych elementów tablicy w trakcie wykonywania programu.

Listing 4.2. Przypisanie wartości elementom tablicy

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int ARRAY_LENGTH = 5;
 6:
 7:    // Tablica pięciu liczb całkowitych zainicjalizowanych z wartością zero.
 8:    int MyNumbers [ARRAY_LENGTH] = {0};
 9:
10:    cout << "Podaj indeks elementu, który ma zostać zmieniony: ";
11:    int nElementIndex = 0;
12:    cin >> nElementIndex;
13:
14:    cout << "Podaj nową wartość: ";
15:    cin >> MyNumbers [nElementIndex];
16:
17:    cout << "Pierwszy element o indeksie 0: " << MyNumbers [0] << endl;
18:    cout << "Drugi element o indeksie 1: " << MyNumbers [1] << endl;
19:    cout << "Trzeci element o indeksie 2: " << MyNumbers [2] << endl;
20:    cout << "Czwarty element o indeksie 3: " << MyNumbers [3] << endl;
21:    cout << "Piąty element o indeksie 4: " << MyNumbers [4] << endl;
22:
23:    return 0;
24: }
```

Wynik ▼

```
Podaj indeks elementu, który ma zostać zmieniony: 2
Podaj nową wartość: 2011
Pierwszy element o indeksie 0: 0
Drugi element o indeksie 1: 0
Trzeci element o indeksie 2: 2011
Czwarty element o indeksie 3: 0
Piąty element o indeksie 4: 0
```

Analiza ▼

Znajdująca się w wierszu 8. deklaracja tablicy używa `const integer ARRAY_LENGTH` do zainicjalizowania pięciu elementów. Podobnie jak w przypadku tablicy statycznej, wielkość omawianej tablicy została na stałe zdefiniowana w trakcie kompilacji. Kompilator zastępuje `ARRAY_LENGTH` wartością 5 i uznaje `MyArray` za tablicę liczb całkowitych składającą się z pięciu elementów. W wierszach od 10. do 12. użytkownik jest proszony o podanie elementu tablicy, którego wartość chce ustawić. Wartość rozpoczynającego się od zera indeksu jest przechowywana w zmiennej `ElementIndex`. Wspomniana liczba całkowita jest następnie używana w wierszu 14. do zmodyfikowania zawartości tablicy. Dane wyjściowe pokazują, że modyfikacja elementu o indeksie 2 powoduje zmianę trzeciego elementu tablicy, ponieważ indeks rozpoczyna się od zera. Powinieneś do tego przywyknąć.

Uwaga

Wielu początkujących programistów C++ w pięcioelementowej tablicy przypisuje wartość piątemu elementowi, używając indeksu 5. To powoduje wykroczenie poza tablicę, ponieważ skompilowany kod próbuje uzyskać dostęp do szóstego (nieistniejącego) elementu tablicy.

Tego rodzaju błąd jest nazywany *błędem słupka w płocie*. Nazwa błędu wzięła się stąd, że liczba słupków potrzebnych do zbudowania płotu zawsze jest o jeden większa od liczby sekcji tego płotu.

Ostrzeżenie

W listingu 4.2 mamy poważny mankament: brakuje mechanizmu sprawdzającego, czy indeks podany przez użytkownika faktycznie wskazuje element tablicy. Program w listingu 4.2 powinien sprawdzić, czy wartość `ElementIndex` mieści się w zakresie od 0 do 4 i odrzucać inne wartości. Brak wspomnianego mechanizmu sprawdzenia pozwala użytkownikowi na potencjalne przypisanie wartości indeksowi wykraczającemu poza tablicę. Takie działanie może doprowadzić do awarii aplikacji, w najgorszym przypadku również do awarii systemu.

Wykonywanie wspomnianych operacji sprawdzania zostało objaśnione w lekcji 6., zatytułowanej „Sterowanie przebiegiem działania programu".

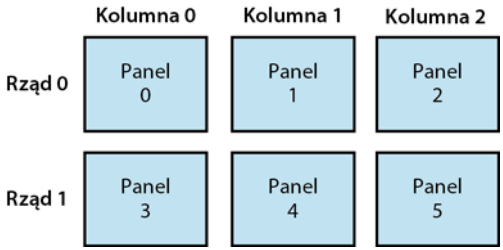
Używaj pętli w celu uzyskania dostępu do elementów tablicy

Podczas pracy z tablicami i ich kolejno ułożonymi elementami dostęp do nich (innymi słowy, iterację) powinieneś uzyskać za pomocą pętli. Zapoznaj się z lekcją 6., a szczególnie z listingiem 6.10, aby szybko dowiedzieć się, jak można efektywnie wstawiać elementy tablicy lub uzyskiwać do nich dostęp.

TAK	NIE
Zawsze przeprowadzaj inicjalizację tablic, w przeciwnym razie będą zawierały przypadkowe wartości.	W tablicy składającej się z N elementów nigdy nie próbuj uzyskać dostępu do N-tego elementu za pomocą indeksu N.
Zawsze upewniaj się, że tablice są używane w ramach ich zdefiniowanych granic.	Nie zapominaj, że pierwszy element tablicy ma indeks zero.

Tablice wielowymiarowe

Omawiane dotąd tablice przypominały książki ułożone na półce. Na dłuższej półce może zmieścić się więcej, natomiast na krótszej mniej. W takim przypadku długość półki to jedyny wymiar definiujący jej pojemność, stąd półka jest jednowymiarowa. Co zrobić w sytuacji, jeśli chcemy tablicę wykorzystać do przedstawienia modelu paneli słonecznych pokazanych na rysunku 4.3? W przeciwieństwie do półek na książki kolejne panele słoneczne mogą być umieszczane w dwóch kierunkach, takich jak długość i szerokość.



RYSUNEK 4.3. Zestaw paneli słonecznych umieszczonych na dachu

Jak możesz zobaczyć na rysunku 4.3, sześć paneli słonecznych zostało umieszczonych w ułożeniu dwuwymiarowym: dwa rzędy i trzy kolumny. Patrząc na to z pewnej perspektywy, można uznać, że mamy tablicę dwóch elementów, z których każdy zawiera trzy kolejne elementy, innymi słowy, tablicę tablic. W języku C++ istnieje możliwość tworzenia tablic dwuwymiarowych, ale nie jesteś ograniczony do jedynie dwóch wymiarów. W zależności od potrzeb i natury aplikacji istnieje możliwość tworzenia w pamięci tablic wielowymiarowych.

Deklarowanie i inicjalizowanie tablic wielowymiarowych

C++ pozwala na deklarację tablic wielowymiarowych przez wskazanie liczby elementów zarezerwowanych dla każdego wymiaru. Dlatego też dwuwymiarową tablicę liczb całkowitych przedstawiających panele słoneczne pokazane na rysunku 4.3 można zdefiniować w następujący sposób:

```
int SolarPanelIDs [2][3];
```

Zwróć uwagę, że na rysunku 4.3 każdemu panelowi został przypisany identyfikator o wartości od 0 do 5 (6 paneli w tablicy). Jeżeli chcesz zainicjalizować tablicę liczb całkowitych i zachować pokazaną na rysunku 4.3 kolejność paneli, wtedy kod powinien przedstawiać się następująco:

```
int SolarPanelIDs [2][3] = {{0, 1, 2}, {3, 4, 5}};
```

Jak możesz się przekonać, użyta tutaj składnia inicjalizacyjna jest w rzeczywistości bardzo podobna do stosowanej podczas inicjalizacji dwóch tablic jednowymiarowych. Zwróć uwagę, że tutaj nie mamy dwóch tablic jednowymiarowych, ale jedną dwuwymiarową składającą się z dwóch rzędów. Jeżeli tablica składałaby się z trzech rzędów i trzech kolumn, wówczas tworzący ją kod byłby następujący:

```
int ThreeRowsThreeColumns [3][3] = {{-501, 206, 2011}, {989, 101, 206}, {303,
456, 596}};
```

Uwaga

Wprawdzie C++ pozwala na tworzenie tablic wielowymiarowych, ale pamięć, w której są one przechowywane, pozostaje jednowymiarowa. Dlatego też kompilator mapuje tablice wielowymiarowe na adresy w pamięci, które rozszerzają się tylko w jednym kierunku.

Jeżeli chcesz, możesz zainicjalizować tablicę o nazwie `SolarPanelIDs` w przedstawiony poniżej sposób, a efekt pozostanie ten sam:

```
int SolarPanelIDs [2][3] = {0, 1, 2, 3, 4, 5};
```

Jednak przedstawione wcześniej rozwiązanie jest znacznie lepsze, ponieważ łatwiej wyobrazić sobie i zrozumieć tablicę wielowymiarową jako tablicę tablic.

Uzyskanie dostępu do elementów tablicy wielowymiarowej

Tablicę wielowymiarową traktuj jako tablicę tablic. Dlatego też tablicę dwuwymiarową składającą się z trzech rzędów i trzech kolumn liczb

całkowitych możesz uznać za tablicę składającą się z trzech elementów, z których każdy jest tablicą trzech liczb całkowitych.

Kiedy trzeba uzyskać dostęp do liczby całkowitej w tablicy, najpierw należy podać adres tablicy zawierającej żądaną liczbę całkowitą, a następnie położenie tej liczby w tablicy. Spójrz na poniższą tablicę:

```
int ThreeRowsThreeColumns [3][3] = {{-501, 206, 2011}, {989, 101, 206}, {303,
456, 596}};
```

Powyższa tablica została zainicjalizowana w sposób, który można przedstawić jako trzy tablice zawierające po trzy liczby całkowite. W omawianym przypadku wartość 206 znajduje się w położeniu [0][1], natomiast wartość 456 jest w położeniu [2][1]. W listingu 4.3 pokazano, jak można uzyskać dostęp do elementów tablicy wielowymiarowej.

Listing 4.3. Uzyskanie dostępu do elementów tablicy wielowymiarowej

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int ThreeRowsThreeColumns [3][3] = \
 6:    {{-501, 206, 2011}, {989, 101, 206}, {303, 456, 596}};
 7:
 8:    cout << "Rząd 0: " << ThreeRowsThreeColumns [0][0] << " " \
 9:                       << ThreeRowsThreeColumns [0][1] << " " \
10:                       << ThreeRowsThreeColumns [0][2] << endl;
11:
12:
13:    cout << "Rząd 1: " << ThreeRowsThreeColumns [1][0] << " " \
14:                       << ThreeRowsThreeColumns [1][1] << " " \
15:                       << ThreeRowsThreeColumns [1][2] << endl;
16:
17:    cout << "Rząd 2: " << ThreeRowsThreeColumns [2][0] << " "\
18:                       << ThreeRowsThreeColumns [2][1] << " " \
19:                       << ThreeRowsThreeColumns [2][2] << endl;
20:
21:    return 0;
22: }
```

Wynik ▼

```
Rząd 0: -501 206 2011
Rząd 1: 989 101 206
Rząd 2: 303 456 596
```

Analiza ▼

Zwróć uwagę na sposób uzyskania dostępu do elementów tablicy poprzez rzędy, od tablicy będącej w rzędzie 0 (pierwszy rząd o indeksie 0) aż do tablicy będącej rzędem 2 (trzeci rząd o indeksie 2). W wierszu 10. pokazano składnię adresowania trzeciego elementu w pierwszym rzędzie.

Uwaga

W listingu 4.3 ilość kodu ulegnie drastycznemu zwiększeniu wraz ze wzrostem liczby elementów tablicy lub wymiaru tablicy. Przedstawiony kod jest niezalecany do stosowania w profesjonalnym środowisku programistycznym.

W lekcji 6., a dokładnie w listingu 6.14, możesz zobaczyć znacznie efektywniejszy sposób uzyskania dostępu do elementów tablicy wielowymiarowej. We wspomnianym rozwiązaniu zastosowano zagnieżdżoną pętlę `for` w celu uzyskania dostępu do wszystkich elementów tablicy. Użycie pętli `for` skraca znacznie kod, który jednocześnie staje się mniej podatny na błędy. Ponadto zmiana liczby elementów tablicy nie spowoduje zmiany wielkości programu.

Tablice dynamiczne

Rozważ aplikację przechowującą informacje metodyczne w szpitalach. Programista w żaden sposób nie może przewidzieć, jaka jest maksymalna liczba rekordów, które będą musiały być obsłużone przez aplikację. Przykładowo programista może przyjąć pewne założenie, które później okaże się znacznie większe niż rozsądne wymagania aplikacji używanej w małym szpitalu. W takim przypadku następuje bezcelowa rezerwacja ogromnej ilości pamięci i zmniejszenie wydajności działania systemu.

Kluczem jest rezygnacja z użycia tablic statycznych, z którymi pracowaliśmy dotąd w rozdziale, i wybór tablic dynamicznych. Zapewnia to możliwość optymalizacji zużycia pamięci oraz skalowalność, w zależności od wymagań dotyczących zasobów i pamięci w trakcie działania programu. Język C++ oferuje wygodne i łatwe w użyciu tablice dynamiczne w postaci klasy `std::vector`, co przedstawiono w listingu 4.4.

Listing 4.4. Tworzenie tablicy dynamicznej liczb całkowitych i dynamiczne dodawanie wartości

```
0: #include <iostream>
1: #include <vector>
2:
3: using namespace std;
4:
```

```
 5: int main()
 6: {
 7:    vector<int> DynArrNums (3);
 8:
 9:    DynArrNums[0] = 365;
10:    DynArrNums[1] = -421;
11:    DynArrNums[2]= 789;
12:
13:    cout << "Ilość elementów tablicy: " << DynArrNums.size() << endl;
14:
15:    cout << "Podaj inną liczbę do umieszczenia w tablicy" << endl;
16:    int AnotherNum = 0;
17:    cin >> AnotherNum;
18:    DynArrNums.push_back(AnotherNum);
19:
20:    cout << "Ilość elementów tablicy: " << DynArrNums.size() << endl;
21:    cout << "Ostatni element w tablicy: ";
22:    cout << DynArrNums[DynArrNums.size() - 1] << endl;
23:
24:    return 0;
25: }
```

Wynik ▼

```
Ilość elementów tablicy: 3
Podaj inną liczbę do umieszczenia w tablicy
2011
Ilość elementów tablicy: 4
Ostatni element w tablicy: 2011
```

Analiza ▼

Nie przejmuj się zastosowaną w listingu 4.4 składnią, ponieważ dotąd nie poruszaliśmy jeszcze tematów wektorów i wzorców. Przeanalizuj dane wyjściowe programu i spróbuj powiązać je z kodem źródłowym. Wedle danych wyjściowych początkowa wielkość tablicy wynosi 3, co odpowiada deklaracji wektora w wierszu 7. Wiedząc o tym, prosimy użytkownika o podanie czwartej liczby (w wierszu 15.), a następnie w wierszu 18. umieszczamy ją w wektorze za pomocą funkcji `push_back()`. Wspomniany obiekt `vector` automatycznie zmienia swoją wielkość w celu przechowywania większej ilości danych. Zwróć uwagę na użycie znanej składni tablic statycznych do uzyskania danych przechowywanych przez wektor. W wierszu 22. następuje uzyskanie dostępu do ostatniego elementu (jego pozycja jest obliczana w trakcie działania programu) za pomocą indeksu rozpoczynającego się od zera. Ostatni element ma indeks

„wielkość –1”. Z kolei wartość zwrotna funkcji `size()` podaje całkowitą ilość elementów (liczb całkowitych) znajdujących się w wektorze.

Uwaga

Aby użyć klasy tablic dynamicznych `std::vector`, konieczne jest dołączenie odpowiedniego pliku nagłówkowego, co w listingu 4.4 widzimy w wierszu 1.

```
#include <vector>
```

Szczegółowe wyjaśnienie wektorów znajduje się w lekcji 17., zatytułowanej „Dynamiczne klasy tablic w STL”.

Ciągi tekstowe w stylu C

Ciąg tekstowy w stylu C to specjalny przypadek tablicy znaków. Wcześniej spotkałeś się już z przykładami ciągów tekstowych w stylu C, które miały postać dosłownych ciągów tekstowych stosowanych w kodzie źródłowym:

```
std::cout << "Witaj, świecie";
```

Powyższe polecenie jest odpowiednikiem przedstawionej poniżej deklaracji tablicy:

```
char SayHello[] = {'W', 'i', 't', 'a', 'j', ',', ' ', 'ś', 'w', 'i', 'e',
↪'c', 'i', 'e', '\0'};
std::cout << SayHello << std::endl;
```

Zwróć uwagę na ostatni znak w tablicy, jakim jest `null` — `'\0'`. Jest on również nazywany znakiem końca ciągu tekstowego, ponieważ informuje kompilator o zakończeniu ciągu tekstowego. Tego rodzaju ciągi tekstowe w stylu C są specjalnymi przypadkami tablic znaków, bo ostatnim znakiem zawsze jest `'\0'`. Po osadzeniu dosłownego ciągu tekstowego w kodzie kompilator sam dodaje na jego końcu znak `'\0'`.

Uwaga

Może się wydawać, że `'\0'` to dwa znaki, ponieważ faktycznie na klawiaturze trzeba nacisnąć dwa znaki. Tutaj ukośnik ma znaczenie specjalne obsługiwane przez kompilator i `\0` oznacza `null` — kompilator wstawia w tym miejscu znak `null`.

Nie możesz napisać po prostu `'0'`, ponieważ zostanie to zinterpretowane jako znak 0, dla którego kod ASCII wynosi 48.

Inne wartości ASCII znajdziesz w tabeli przedstawionej w dodatku E, zatytułowanym „Kody ASCII”.

Jeżeli znak '\0' umieścisz w środku tablicy, nie powoduje on zmiany wielkości tablicy, a jedynie oznacza, że przetwarzanie ciągu tekstowego za pomocą danej tablicy jako danych wejściowych zakończy się w miejscu umieszczenia znaku '\0'. Zostało to zademonstrowane w listingu 4.5.

Listing 4.5. Analiza znaku null w ciągu tekstowym w stylu C

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    char SayHello[] = {'W','i','t','a','j',',','
       ↪','ś','w','i','e','c','i','e','\0'};
 6:    cout << SayHello << endl;
 7:    cout << "Wielkość tablicy: " << sizeof(SayHello) << endl;
 8:
 9:    cout << "Zastąpienie spacji znakiem null" << endl;
10:    SayHello[6] = '\0';
11:    cout << SayHello << endl;
12:    cout << "Wielkość tablicy: " << sizeof(SayHello) << endl;
13:
14:    return 0;
15: }
```

Wynik ▼

```
Witaj, świecie
Wielkość tablicy: 15
Zastąpienie spacji znakiem null
Witaj,
Wielkość tablicy: 15
```

Analiza ▼

W wierszu 10. następuje zastąpienie spacji w „Witaj, świecie” znakiem `null`. Zauważ, że teraz w tablicy znajdują się dwa znaki `null`, ale pierwszy napotkany spełnia swoją funkcję. Po zastąpieniu spacji znakiem `null` wynikiem jest wyświetlenie ciągu tekstowego skróconego do postaci „Witaj,”. Wywołanie funkcji `sizeof()` w wierszach 7. i 12. pokazuje, że wielkość tablicy nie uległa zmianie, nawet mimo zmiany wyświetlanych danych wyjściowych.

Ostrzeżenie

Jeżeli zapomnisz dodać '\0' podczas deklaracji i inicjalizacji tablicy znaków w wierszu 5. listingu 4.5, możesz się spodziewać, że dane wyjściowe po komunikacie „Witaj, świecie" będą zawierały jeszcze inne przypadkowe znaki. Wyjaśnienie jest proste: `std::cout` nie przestanie wyświetlać zawartości tablicy, aż do napotkania znaku `null`, nawet jeśli oznacza to wykroczenie poza tablicę.

Tego rodzaju błąd może doprowadzić do awarii aplikacji, w pewnych przypadkach również do zmniejszenia stabilności systemu.

Ciągi tekstowe w stylu C są niebezpieczne. W listingu 4.6 zademonstrowano ryzyko związane z ich używaniem.

Listing 4.6. Ryzykowna aplikacja używająca ciągów tekstowych w stylu C oraz danych wejściowych użytkownika

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Podaj słowo składające się z maksymalnie 20 znaków:" << endl;
 6:
 7:    char UserInput [21] = {'\0'};
 8:    cin >> UserInput;
 9:
10:    cout << "Wielkość danych wejściowych: " << strlen (UserInput) << endl;
11:
12:    return 0;
13: }
```

Wynik ▼

```
Podaj słowo składające się z maksymalnie 20 znaków:
NieUżywajTegoProgramu
Wielkość danych wejściowych: 22
```

Analiza ▼

Niebezpieczeństwo jest widoczne w danych wyjściowych. Program prosi użytkownika o wprowadzenie danych o maksymalnej długości 20 znaków. Powód jest prosty: zadeklarowany w wierszu 7. bufor do przechowywania danych wejściowych ma zdefiniowaną stałą — czyli statyczną — wielkość wynoszącą 21 znaków. Ponieważ ostatnim znakiem musi być `null` ('\0'), maksymalna wielkość tekstu przechowywanego w buforze wynosi 20 znaków. Zwróć uwagę na użycie funkcji `strlen()` w wierszu 10. w celu sprawdzenia

długości ciągu tekstowego. Funkcja `strlen()` przechodzi przez bufor i zlicza znaki, aż do napotkania znaku `null` oznaczającego koniec ciągu tekstowego. Wspomniany znak `null` został wstawiony przez polecenie `cin` na końcu danych wejściowych podanych przez użytkownika. Takie zachowanie funkcji `strlen()` jest niebezpieczne, ponieważ może ona wykroczyć poza tablicę znaków, jeśli użytkownik podał ciąg tekstowy dłuższy niż 20 znaków. W listingu 6.2 w lekcji 6. pokazano, jak zaimplementować mechanizm sprawdzający, który zagwarantuje, że w tablicy nie zostanie zapisanych więcej danych, niż pozwala jej pojemność (tzn. nie nastąpi wykroczenie poza tablicę).

Ciągi tekstowe C++ — użycie klasy std::string

Standardowy ciąg tekstowy C++ to najefektywniejszy sposób pracy z danymi wejściowymi w postaci tekstu oraz przeprowadzania operacji na ciągach tekstowych, np. ich łączenia.

Ostrzeżenie

> W aplikacjach utworzonych w języku C (lub w C++ przez programistów mających ogromne doświadczenie w programowaniu w C) bardzo często używane są funkcje `strcpy` (kopiowanie ciągu tekstowego), `strcat` (łączenie ciągów tekstowych) oraz `strlen` (określenie długości ciągu tekstowego) i inne tego rodzaju funkcje.
>
> Wymienione funkcje pobierają jako dane wejściowe ciągi tekstowe w stylu C, ich działanie jest niebezpieczne, ponieważ szukają w ciągach tekstowych znaku `null` i mogą wykroczyć poza tablicę znaków, jeśli programista nie zadbał o umieszczenie w ciągu tekstowym wspomnianego znaku `null`.

Język C++ oferuje potężne i bezpieczne możliwości operowania ciągami tekstowymi za pomocą klasy `std::string`, co przedstawiono w listingu 4.7. Obiekt klasy `std::string` nie ma statycznej wielkości, jak tablica `char` będąca implementacją ciągu tekstowego w stylu C, i może być skalowany, gdy trzeba przechowywać większą ilość danych.

Listing 4.7. Użycie obiektu klasy std::string w celu inicjalizacji, przechowywania danych wejściowych użytkownika, kopiowania, łączenia i określenia długości ciągu tekstowego

```
0: #include <iostream>
1: #include <string>
2:
```

```
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    string Greetings ("Witaj, std::string!");
 8:    cout << Greetings << endl;
 9:
10:    cout << "Podaj wiersz tekstu: " << endl;
11:    string FirstLine;
12:    getline(cin, FirstLine);
13:
14:    cout << "Podaj inny tekst: " << endl;
15:    string SecLine;
16:    getline(cin, SecLine);
17:
18:    cout << "Wynik operacji łączenia: " << endl;
19:    string Concat = FirstLine + " " + SecLine;
20:    cout << Concat << endl;
21:
22:    cout << "Kopia utworzonego ciągu tekstowego: " << endl;
23:    string Copy;
24:    Copy = Concat;
25:    cout << Copy << endl;
26:
27:    cout << "Długość utworzonego ciągu tekstowego: " << Concat.length() << endl;
28:
29:    return 0;
30: }
```

Wynik ▼

```
Witaj, std::string!
Podaj wiersz tekstu:
Uwielbiam
Podaj inny tekst:
ciągi tekstowe C++
Wynik operacji łączenia:
Uwielbiam ciągi tekstowe C++
Kopia utworzonego ciągu tekstowego:
Uwielbiam ciągi tekstowe C++
Długość utworzonego ciągu tekstowego: 28
```

Analiza ▼

Spróbuj zrozumieć dane wyjściowe i powiązać je z różnymi elementami w kodzie źródłowym. Nie pozwól, aby nowa składnia przeszkadzała na tym etapie. Program rozpoczyna działanie od wyświetlenia ciągu tekstowego

zainicjalizowanego w wierszu 7. (*Witaj, std::string*). Następnie prosi użytkownika o podanie dwóch wierszy tekstu, które są przechowywane w zmiennych `FirstLine` i `SecLine` (patrz wiersze 12. i 16.). Rzeczywista operacja połączenia ciągów tekstowych jest całkiem prosta i przypomina arytmetyczne dodawanie (patrz wiersz 19.), gdzie nawet spacja została dodana do pierwszego wiersza. Przeprowadzona w wierszu 24. operacja kopiowania polega po prostu na przypisaniu ciągu tekstowego. W celu określenia długości ciągu tekstowego w wierszu 27. następuje wywołanie względem sprawdzanego ciągu funkcji `length()`.

Uwaga

Aby można było używać ciągów tekstowych C++, konieczne jest dołączenie odpowiedniego pliku nagłówkowego:

```
#include <string>
```

W listingu 4.7 powyższe polecenie znajduje się w wierszu 1.

Szczegółowe omówienie różnych funkcji klasy `std::string` znajduje się w lekcji 16., zatytułowanej „Klasa string w STL”. Ponieważ nie omówiliśmy jeszcze klas i wzorców, zignoruj nieznane fragmenty w wymienionej lekcji i skoncentruj się na zrozumieniu przedstawionych w niej przykładów.

Podsumowanie

W czasie tej lekcji poznałeś podstawy dotyczące tablic, dowiedziałeś się, czym są tablice oraz jak można ich używać. Nauczyłeś się deklarować tablice, inicjalizować je, uzyskiwać dostęp do elementów tablicy oraz zmieniać ich wartości. Wiesz już, jak ważne jest, aby nie wykraczać poza granice tablic.

Wspomniane wykroczenie skutkuje błędem typu *przepełnienie bufora*, a sprawdzenie danych wejściowych przed użyciem indeksów elementów pomaga w zapewnieniu, że granice tablic nie będą przekroczone.

Tablice dynamiczne to takie, w których programista nie musi przejmować się ustawieniem w trakcie kompilacji wielkości tablicy. W tablicach dynamicznych pamięć jest efektywniej używana, zwłaszcza w przypadku, gdy tablica zawiera mniej elementów, niż przewidywano.

Dowiedziałeś się również, że ciągi tekstowe w stylu C to specjalny przypadek tablic znaków, w których koniec ciągu tekstowego jest oznaczony znakiem `null` (`'\0'`). Co ważniejsze, wiesz także, że język C++ oferuje znacznie lepsze

rozwiązanie do obsługi ciągów tekstowych, czyli obiekt klasy `std::string`, który zapewnia wygodne funkcje pozwalające m.in. na określenie długości ciągu tekstowego, łączenie ciągów tekstowych i wykonywanie innych podobnych operacji.

Pytania i odpowiedzi

Pytanie: Dlaczego mam zadawać sobie trud i inicjalizować elementy tablic statycznych?

Odpowiedź: Jeśli tablica, w przeciwieństwie do zmiennej innego typu, nie zostanie zainicjalizowana, zawiera śmieci i inne przypadkowe wartości znajdujące się w pamięci jako pozostałość po ostatniej operacji. Inicjalizacja tablicy gwarantuje, że tablica będzie zawierała przewidywalne informacje początkowe.

Pytanie: Czy w tablicy dynamicznej elementy również trzeba inicjalizować z tego samego powodu podanego w pierwszym pytaniu?

Odpowiedź: Nie. Tablica dynamiczna jest tablicą inteligentną. Jej elementów nie trzeba inicjalizować z wartościami domyślnymi, o ile nie występuje konkretny powód związany z aplikacją, która np. wymaga określonych wartości początkowych w tablicy.

Pytanie: Czy, mając wybór, powinienem decydować się na użycie ciągów tekstowych w stylu C wymagających znaku `null` na końcu?

Odpowiedź: Tak, ale tylko wtedy, gdy masz pistolet przystawiony do głowy. Dostarczana przez C++ klasa `std::string` to znacznie bezpieczniejsze i oferujące więcej funkcji rozwiązanie. Dobry programista C++ powinien trzymać się z dala od ciągów tekstowych w stylu C.

Pytanie: Czy długość ciągu tekstowego obejmuje również znak `null` znajdujący się na jego końcu?

Odpowiedź: Nie. Długość ciągu tekstowego „Witaj, świecie” wynosi 14 znaków i obejmuje spację oraz nie obejmuje znaku `null` na jego końcu.

Pytanie: Chciałbym użyć ciągów tekstowych w stylu C w zdefiniowanych tablicach typu `char`. Jaka powinna być wielkość tablicy?

Odpowiedź: W tym momencie stykasz się z komplikacjami wynikającymi z użycia ciągów tekstowych w stylu C. Wielkość tablicy powinna być o 1 większa od najdłuższego ciągu tekstowego, który kiedykolwiek zostanie umieszczony w danej tablicy. To bardzo ważne, ponieważ uwzględnia znak `null` na końcu najdłuższego ciągu tekstowego. Jeżeli „Witaj, świecie" to najdłuższy ciąg tekstowy przechowywany w tablicy typu `char`, wtedy wielkość tablicy powinna wynosić 14+1 = 15 znaków.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Sprawdź tablicę `MyNumbers` w listingu 4.1. Jakie są indeksy pierwszego i ostatniego elementu wymienionej tablicy?
2. Jeżeli musisz pozwolić użytkownikowi na podanie ciągów tekstowych, czy użyjesz ciągów tekstowych w stylu C?
3. Ile kompilator widzi znaków w wyrażeniu `'\0'`?
4. Zapomniałeś zakończyć znakiem `null` ciąg tekstowy w stylu C. Co się stanie, jeśli spróbujesz go użyć?
5. Zobacz deklarację wektora w listingu 4.4 i spróbuj utworzyć tablicę dynamiczną zawierającą elementy typu `char`.

Ćwiczenia

1. Zadeklaruj tablicę przedstawiającą pola na szachownicy; typem tablicy powinien być `enum` definiujący naturę elementów na planszy.

2. **Łowcy błędów:** Co jest złego w poniższym fragmencie kodu?

```
int MyNumbers[5] = {0};
MyNumbers[5] = 450; // Przypisanie piątemu elementowi wartości 450.
```

3. **Łowcy błędów:** Co jest złego w poniższym fragmencie kodu?

```
int MyNumbers[5];
cout << MyNumbers[3];
```

Lekcja 5

Wyrażenia, instrukcje i operatory

Serce programu stanowi zestaw kolejno wykonywanych poleceń. Wspomniane polecenia zostały umieszczone w wyrażeniach i instrukcjach, w nich także używa się operatorów do wykonywania określonych obliczeń lub akcji.

Z tej lekcji dowiesz się:

- **czym są polecenia,**
- **czym są bloki, czyli polecenia złożone,**
- **czym są operatory,**
- **jak przeprowadzać proste operacje arytmetyczne i logiczne.**

Polecenia

Języki zarówno mówiony, jak i programowania składają się z kolejno wykonywanych poleceń. Przeanalizujmy pierwsze ważne polecenie, które poznałeś w tej książce:

```
cout << "Witaj, świecie" << endl;
```

Polecenie wykorzystujące `cout` powoduje wyświetlenie tekstu na ekranie. Wszystkie polecenia w C++ kończą się średnikiem oznaczającym granicę polecenia. Przypomina on kropkę używaną na końcu zadania w języku polskim. Następne polecenie może znaleźć się tuż za średnikiem, ale w celu wygody i przejrzystości kodu źródłowego poszczególne polecenia są najczęściej umieszczane w oddzielnych wierszach. W przedstawionym poniżej wierszu znajdują się dwa polecenia:

```
cout << "Witaj, świecie" << endl; cout << "Inne witaj" << endl;
// Jeden wiersz, dwa polecenia.
```

Uwaga

> Znaki odstępu (m.in. spacje, tabulatory, wysuw wiersza, znak nowego wiersza itd.) zwykle pozostają niewidoczne dla kompilatora. Natomiast wymienione znaki umieszczone w dosłownych ciągach tekstowych mają wpływ na generowane dane wyjściowe.

Przedstawione poniżej polecenie jest nieprawidłowe:

```
cout << "Witaj,
   świecie" << endl; // W dosłownym ciągu tekstowym znak nowego wiersza jest niedozwolony.
```

Powyższy kod zwykle prowadzi do wygenerowania błędu informującego, że w wierszu pierwszym kompilator nie znajduje zamykającego znaku cudzysłów (") oraz średnika kończącego polecenie. Jeżeli z jakiegokolwiek powodu musisz umieścić polecenie w dwóch wierszach, możesz to zrobić, wstawiając ukośnik (\) na końcu pierwszego wiersza:

```
cout << "Witaj, \
   świecie" << endl; // Wiersz można podzielić na dwa.
```

Inny sposób zapisania w dwóch wierszach przedstawionego wcześniej polecenia polega na użyciu dwóch dosłownych ciągów tekstowych zamiast tylko jednego, np. tak:

```
cout << "Witaj, "
   "świecie" << endl; // Dozwolone jest użycie dwóch dosłownych ciągów tekstowych.
```

W powyższym poleceniu kompilator znajduje dwa umieszczone kolejno dosłowne ciągi tekstowe, które następnie łączy.

Uwaga

Podział polecenia na wiele wierszy może być całkiem użyteczny, w przypadku gdy zawiera ono dłuższe elementy tekstowe lub skomplikowane wyrażenia składające się z wielu zmiennych — skutkiem jest powstanie polecenia znacznie dłuższego niż szerokość większości dostępnych ekranów.

Polecenia złożone, czyli bloki

Zgrupowanie poszczególnych poleceń i umieszczenie ich w nawiasie klamrowym {...} powoduje powstanie tzw. polecenia złożonego, czyli bloku:

```
{
   int Number = 365;
   cout << "Ten blok zawiera liczbę całkowitą i polecenie cout" << endl;
}
```

Blok najczęściej zawiera wiele poleceń wskazujących na ich wzajemne powiązanie. Szczególną użyteczność bloki ujawniają w konstrukcjach warunkowych, takich jak `if`, oraz pętlach, które zostaną omówione w lekcji 6., zatytułowanej „Sterowanie przebiegiem działania programu".

Użycie operatorów

Operatory to dostarczane przez C++ narzędzie pozwalające na pracę z danymi, ich transformację, przetwarzanie oraz podejmowanie decyzji na podstawie tych danych.

Operator przypisania (=)

Z operatorem przypisania spotkaliśmy się już w tej książce i używaliśmy go intuicyjnie:

```
int MyInteger = 101;
```

Powyższe polecenie wykorzystuje operator przypisania w celu inicjalizacji liczby całkowitej z wartością 101. Wartość operandu znajdującego się po lewej stronie (nazywana l-wartością) zostaje przez operator przypisania zastąpiona wartością podaną po prawej stronie operatora (nazywaną r-wartością).

Zrozumienie l-wartości i r-wartości

Wspomniana wcześniej l-wartość bardzo często odwołuje się do miejsc w pamięci. Zmienna, taka jak użyta w poprzednim przykładzie `MyInteger`, jest w rzeczywistości uchwytem do miejsca w pamięci i nosi nazwę l-wartości. Z kolei r-wartość może być zawartością znajdującą się we wspomnianym miejscu w pamięci.

Tak więc wszystkie l-wartości mogą być r-wartościami, ale nie na odwrót. Aby to lepiej zrozumieć, spójrz na poniższe polecenie, które nie ma najmniejszego sensu i uniemożliwi kompilację kodu:

```
101 = MyInteger;
```

Operatory dodawania (+), odejmowania (–), mnożenia (*), dzielenia (/) i reszty z dzielenia (%)

Pomiędzy dwoma operandami można przy użyciu operatorów dodawania (+), odejmowania (–), mnożenia (*), dzielenia (/) i reszty z dzielenia (%) przeprowadzać operacje arytmetyczne:

```
int Num1 = 22;
int Num2 = 5;
int addition = Num1 + Num2; // 27.
int subtraction = Num1 – Num2; // 17.
int multiplication = Num1 * Num2; // 110.
int division = Num1 / Num2; // 4.
int modulo = Num1 % Num2; // 2.
```

Zwróć uwagę, że wartością zwrotną operatora dzielenia (/) jest wynik dzielenia dwóch operandów. W przypadku liczb całkowitych wynik nie zawiera części dziesiętnej, ponieważ z natury liczby całkowite nie zawierają takich informacji. Jak sama nazwa wskazuje, operator reszty z dzielenia (%, nazywany modulo) zwraca resztę z dzielenia i można go stosować jedynie dla wartości w postaci liczb całkowitych. W listingu 5.1 przedstawiono prosty program demonstrujący użycie operacji arytmetycznych na dwóch liczbach podanych przez użytkownika.

Listing 5.1. Przykład użycia operatorów arytmetycznych na liczbach całkowitych podanych przez użytkownika

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
```

```
 4: {
 5:    cout << "Podaj dwie liczby całkowite:" << endl;
 6:    int Num1 = 0, Num2 = 0;
 7:    cin >> Num1;
 8:    cin >> Num2;
 9:
10:   cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
11:   cout << Num1 << " - " << Num2 << " = " << Num1 - Num2 << endl;
12:   cout << Num1 << " * " << Num2 << " = " << Num1 * Num2 << endl;
13:   cout << Num1 << " / " << Num2 << " = " << Num1 / Num2 << endl;
14:   cout << Num1 << " % " << Num2 << " = " << Num1 % Num2 << endl;
15:
16:   return 0;
17: }
```

Wynik ▼

```
Podaj dwie liczby całkowite:
365
25
365 + 25 = 390
365 - 25 = 340
365 * 25 = 9125
365 / 25 = 14
365 % 25 = 15
```

Analiza ▼

Większość kodu źródłowego programu powinna być jasna. Prawdopodobnie najbardziej interesującym wierszem jest ten, który zawiera operator modulo. Działanie tego operatora polega na zwróceniu reszty pozostałej w wyniku dzielenia liczby `Num1` (tutaj 365) przez `Num2` (tutaj 25).

Operatory inkrementacji (++) i dekrementacji (--)

Czasami trzeba przeprowadzić inkrementację wartości. Najczęściej dotyczy to zmiennych kontrolujących działanie pętli, a wartość zmiennej musi być inkrementowana lub dekrementowana podczas każdego wykonania pętli.

Do wykonania tego rodzaju zadania język C++ udostępnia operatory inkrementacji (`++`) i dekrementacji (`--`).

Składnia użycia wymienionych operatorów jest następująca:

```
int Num1 = 101;
int Num2 = Num1++; // Postfiksowy operator inkrementacji.
int Num2 = ++Num1; // Prefiksowy operator inkrementacji.
int Num2 = Num1--; // Postfiksowy operator dekrementacji.
int Num2 = --Num1; // Prefiksowy operator dekrementacji.
```

Jak przedstawiono w powyższym fragmencie kodu, mamy dwa sposoby użycia operatorów inkrementacji i dekrementacji: przed operandem i po operandzie. Operator umieszczony przed operandem jest nazywany prefiksowym operatorem inkrementacji lub dekrementacji, natomiast znajdujący się po operandzie nosi nazwę postfiksowego operatora inkrementacji lub dekrementacji.

Operator postfiksowy czy prefiksowy?

Przede wszystkim bardzo ważne jest zrozumienie różnicy pomiędzy operatorami prefiksowym i postfiksowym, a następnie używanie odpowiedniego dla danego zadania. Wynikiem działania operatora postfiksowego jest przypisanie l-wartości do r-wartości, a następnie inkrementacja (lub dekrementacja) r-wartości. Oznacza to, że we wszystkich przypadkach używania operatora postfiksowego wartością `Num2` będzie stara wartość `Num1` (sprzed operacji inkrementacji lub dekrementacji).

Operatory prefiksowe działają w dokładnie odwrotny sposób. Najpierw następuje inkrementacja r-wartości, a następnie jej przypisanie do l-wartości. W takim przypadku `Num1` i `Num2` będą miały taką samą wartość. W listingu 5.2 przedstawiono efekt użycia prefiksowych i postfiksowych operatorów inkrementacji i dekrementacji na przykładowej liczbie całkowitej.

Listing 5.2. Różnica pomiędzy operatorem postfiksowym i prefiksowym

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    int MyInt = 101;
6:    cout << "Początkowa liczba całkowita: " << MyInt << endl;
7:
8:    int PostFixInc = MyInt++;
9:    cout << "Wynik inkrementacji postfiksowej = " << PostFixInc << endl;
```

```
10:    cout << "Po inkrementacji postfiksowej, MyInt = " << MyInt << endl;
11:
12:    MyInt = 101; // Zerowanie.
13:    int PreFixInc = ++MyInt;
14:    cout << "Wynik inkrementacji prefiksowej = " << PreFixInc << endl;
15:    cout << "Po inkrementacji prefiksowej, MyInt = " << MyInt << endl;
16:
17:    MyInt = 101;
18:    int PostFixDec = MyInt--;
19:    cout << "Wynik dekrementacji postfiksowej = " << PostFixDec << endl;
20:    cout << "Po dekerementacji postfiksowej, MyInt = " << MyInt << endl;
21:
22:    MyInt = 101;
23:    int PreFixDec = --MyInt;
24:    cout << "Wynik dekrementacji prefiksowej = " << PreFixDec << endl;
25:    cout << "Po dekrementacji prefiksowej, MyInt = " << MyInt << endl;
26:
27:    return 0;
28: }
```

Wynik ▼

```
Początkowa liczba całkowita: 101
Wynik inkrementacji postfiksowej = 101
Po inkrementacji postfiksowej, MyInt = 102
Wynik inkrementacji prefiksowej = 102
Po inkrementacji prefiksowej, MyInt = 102
Wynik dekrementacji postfiksowej = 101
Po dekerementacji postfiksowej, MyInt = 100
Wynik dekrementacji prefiksowej = 100
Po dekrementacji prefiksowej, MyInt = 100
```

Analiza ▼

Wygenerowane wyniki pokazują, że operatory postfiksowe działają odmiennie od prefiksowych. Przypisane w wierszach 8. i 18. l-wartości mają początkową wartość liczby całkowitej sprzed rzeczywistej operacji inkrementacji lub dekrementacji. Z kolei wynikiem przeprowadzonych w wierszach 13. i 23. operacji jest przypisanie l-wartości po inkrementacji lub dekrementacji. To bardzo ważna różnica, o której należy pamiętać podczas wyboru typu operatora.

Warto zauważyć, że w przedstawionym poniżej fragmencie kodu użycie operatora prefiksowego lub postfiksowego nie ma wpływu na otrzymane dane wyjściowe:

```
MyInt++; // Taki sam efekt jak…
++MyInt;
```

Wynika to z braku przypisania wartości początkowej; w obu przypadkach wynikiem będzie po prostu liczba całkowita po inkrementacji.

Uwaga

Bardzo często można spotkać się z opinią, że prefiksowy operator inkrementacji lub dekrementacji zapewnia lepszą wydajność. Dlatego też użycie `++MyInt` jest bardziej preferowane niż `MyInt++`.

To prawda, przynajmniej teoretycznie, ponieważ po użyciu operatora postfiksowego kompilator musi tymczasowo przechowywać wartość początkową, gdyby musiała być przypisywana. Podczas pracy z liczbami całkowitymi użycie operatora prefiksowego będzie miało niewielki wpływ na wydajność, jednak w trakcie stosowania niektórych klas efekt może być już znaczący.

Unikaj przepełnień i rozsądnie wybieraj odpowiedni typ danych

Typy danych, takie jak `short`, `int`, `long`, `unsigned int`, `unsigned long` itd., mają określoną pojemność pozwalającą na przechowywanie liczb. Kiedy w trakcie operacji arytmetycznych przekroczy się granicę pojemności dla wybranego typu, wtedy mamy do czynienia z tzw. *przepełnieniem*.

Weźmy jako przykład liczbę typu `unsigned short`. Typ danych `short` zajmuje 16 bitów, a więc może przechowywać wartości od 0 do 65 535. Kiedy przeprowadzisz operację 1+65535 przy użyciu typu `short`, nastąpi przepełnienie i wynikiem będzie 0. Przypomina to licznik w samochodzie składający się z jedynie pięciu cyfr, po przejechaniu 99 999 km licznik „przeskoczy" ponownie na początek i wskaże 00 000 km.

W omawianym przykładzie do obsługi wspomnianego licznika typ unsigned short nigdy nie jest dobrym rozwiązaniem. Programista postąpi znacznie lepiej, jeśli użyje typu `unsigned int` i tym samym zapewni obsługę wartości większych niż 65 535.

W przypadku liczby całkowitej typu `signed short` o zakresie od –32 768 do 32 767 dodanie 1 do 32 767 bardzo często powoduje, że liczba przyjmuje największą wartość ujemną — zachowanie zależy jednak od konkretnego kompilatora.

W programie przedstawionym w listingu 5.3 zademonstrowano błędy przepełnienia, które można przypadkowo wprowadzić do programu przy użyciu operacji arytmetycznych.

Listing 5.3. Demonstracja efektów ubocznych przepełnienia liczby całkowitej ze znakiem i bez znaku

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    unsigned short UShortValue = 65535;
 6:    cout << "Inkrementacja liczby typu unsigned short " << UShortValue <<
       ↪" daje: ";
 7:    cout << ++UShortValue << endl;
 8:
 9:    short SignedShort = 32767;
10:    cout << "Inkrementacja liczby typu signed short " << SignedShort<<
       ↪" daje: ";
11:    cout << ++SignedShort << endl;
12:
13:    return 0;
14: }
```

Wynik ▼

```
Inkrementacja liczby typu unsigned short 65535 daje: 0
Inkrementacja liczby typu signed short 32767 daje: -32768
```

Analiza ▼

Jak możesz zobaczyć, dane wyjściowe wskazują na wystąpienie przypadkowego przepełnienia, co może doprowadzić do niechcianego i nieprzewidywalnego zachowania aplikacji. Jeżeli używasz liczb całkowitych do alokacji pamięci, wtedy dla typu `unsigned short` może wystąpić sytuacja żądania zera bajtów, gdy naprawdę konieczne jest zarezerwowanie 65 536 bajtów.

Operatory równości (==) i nierówności (!=)

Bardzo często zachodzi konieczność sprawdzenia spełnienia lub niespełnienia określonego warunku, zanim będzie można kontynuować działanie. Operatory `==` (operandy są równe) i `!=` (operandy są nierówne) pomogą w realizacji wspomnianego zadania.

Wynikiem działania sprawdzenia równości jest typ `bool`, tzn. wartość `true` (prawda) lub `false` (fałsz).

```
int MyNum = 20;
bool CheckEquality = (MyNum == 20); // Prawda.
bool CheckInequality = (MyNum != 100); // Prawda.
bool CheckEqualityAgain = (MyNum == 200); // Fałsz.
bool CheckInequalityAgain = (MyNum != 20); // Fałsz.
```

Operatory relacji

Poza weryfikowaniem równości, można również sprawdzić nierówność pewnej zmiennej względem określonej wartości. Do tego celu język C++ oferuje zestaw operatorów relacji, które wymieniono w tabeli 5.1.

Tabela 5.1. Operatory relacji

Nazwa operatora	Opis
Mniejszy niż (<)	Przyjmuje wartość `true`, jeśli jeden operand jest mniejszy od drugiego (`Op1 < Op2`), w przeciwnym razie przyjmuje wartość `false`.
Większy niż (>)	Przyjmuje wartość `true`, jeśli jeden operand jest większy od drugiego (`Op1 > Op2`), w przeciwnym razie przyjmuje wartość `false`.
Mniejszy niż lub równy (<=)	Przyjmuje wartość `true`, jeśli jeden operand jest mniejszy od drugiego lub mu równy, w przeciwnym razie przyjmuje wartość `false`.
Większy niż lub równy (>=)	Przyjmuje wartość `true`, jeśli jeden operand jest większy od drugiego lub mu równy, w przeciwnym razie przyjmuje wartość `false`.

Jak przedstawiono w tabeli 5.1, wynikiem operacji porównania zawsze jest wartość `true` lub `false`, czyli typ `bool`. W poniższym fragmencie kodu możesz zobaczyć, jak używać operatorów relacji wymienionych w tabeli 5.1.

```
int MyNum = 20; // Przykładowa liczba całkowita.
bool CheckLessThan = (MyNum < 100);  // Prawda.
bool CheckGreaterThan = (MyNum > 100); // Fałsz.
bool CheckLessThanEqualTo = (MyNum <= 20); // Prawda.
bool CheckGreaterThanEqualTo = (MyNum >= 20); // Prawda.
bool CheckGreaterThanEqualToAgain = (MyNum >= 100); // Fałsz.
```

W programie, którego kod źródłowy znajduje się w listingu 5.4, pokazano efekt użycia operatorów relacji do wyświetlenia wyniku na ekranie.

Listing 5.4. Demonstracja użycia operatorów równości oraz relacji

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Podaj dwie liczby całkowite:" << endl;
 6:    int Num1 = 0, Num2 = 0;
 7:    cin >> Num1;
 8:    cin >> Num2;
 9:
10:    bool Equality = (Num1 == Num2);
11:    cout << "Wynik testu równości wynosi: " << Equality << endl;
12:
13:    bool Inequality = (Num1 != Num2);
14:    cout << "Wynik testu nierówności wynosi: " << Inequality << endl;
15:
16:    bool GreaterThan = (Num1 > Num2);
17:    cout << "Wynik testu " << Num1 << " > " << Num2;
18:    cout << " wynosi: " << GreaterThan << endl;
19:
20:    bool LessThan = (Num1 < Num2);
21:    cout << "Wynik testu " << Num1 << " < " << Num2 << " wynosi: " <<
       ↪LessThan << endl;
22:
23:    bool GreaterThanEquals = (Num1 >= Num2);
24:    cout << "Wynik testu " << Num1 << " >= " << Num2;
25:    cout << " wynosi: " << GreaterThanEquals << endl;
26:
27:    bool LessThanEquals = (Num1 <= Num2);
28:    cout << "Wynik testu " << Num1 << " <= " << Num2;
29:    cout << " wynosi: " << LessThanEquals << endl;
30:
31:    return 0;
32: }
```

Wynik ▼

```
Podaj dwie liczby całkowite:
365
-24
Wynik testu równości wynosi: 0
Wynik testu nierówności wynosi: 1
Wynik testu 365 > -24 wynosi: 1
Wynik testu 365 < -24 wynosi: 0
Wynik testu 365 >= -24 wynosi: 1
Wynik testu 365 <= -24 wynosi: 0
```

Kolejne uruchomienie programu:

```
Podaj dwie liczby całkowite:
101
101
Wynik testu równości wynosi: 1
Wynik testu nierówności wynosi: 0
Wynik testu 101 > 101 wynosi: 0
Wynik testu 101 < 101 wynosi: 0
Wynik testu 101 >= 101 wynosi: 1
Wynik testu 101 <= 101 wynosi: 1
```

Analiza ▼

Program wyświetla binarny wynik różnych operacji. Warto zwrócić uwagę, że kiedy podane zostaną dwie jednakowe liczby całkowite (jak zademonstrowano w drugim uruchomieniu programu), wtedy operatory ==, >= i <= powodują wygenerowanie takich samych danych i zwrot wartości 1.

Ponieważ dane wyjściowe operatorów równości i relacyjnych są binarne, doskonale nadają się do zastosowania w poleceniach pomagających w podejmowaniu decyzji oraz w wyrażeniach pętli. W drugim przypadku gwarantują wykonanie pętli tylko wtedy, gdy wartością warunku pętli będzie `true`. Więcej informacji na temat wykonywania warunkowego oraz pętli znajdziesz w lekcji 6.

Operatory logiczne NOT, AND, OR i XOR

Logiczna operacja NOT jest obsługiwana przez operator `!` i działa z pojedynczym operandem. W tabeli 5.2 przedstawiono wynik logicznej operacji NOT, który — zgodnie z oczekiwaniami — jest po prostu odwróconą wartością boolowską operandu.

Tabela 5.2. Wynik logicznej operacji NOT

Operand	Wynik NOT (operand)
Fałsz	Prawda
Prawda	Fałsz

Pozostałe operatory logiczne, czyli AND, OR i XOR, potrzebują do działania dwóch operandów. Wynikiem logicznej operacji AND będzie prawda (`true`)

tylko wtedy, gdy każdy operand przyjmie wartość `true`. W tabeli 5.3 przedstawiono funkcjonowanie logicznej operacji AND.

Tabela 5.3. Wynik logicznej operacji AND

Operand 1	Operand 2	Wynik Operand1 AND Operand2
Fałsz	Fałsz	Fałsz
Prawda	Fałsz	Fałsz
Fałsz	Prawda	Fałsz
Prawda	Prawda	Prawda

Logiczna operacja AND jest obsługiwana przez operator &&.

Przechodzimy teraz do logicznej operacji OR. Jej wynikiem będzie wartość prawda (`true`), gdy przynajmniej jeden z operandów będzie miał wartość `true`, co zostało przedstawione w tabeli 5.4.

Tabela 5.4. Wynik logicznej operacji OR

Operand 1	Operand 2	Wynik Operand1 OR Operand2
Fałsz	Fałsz	Fałsz
Prawda	Fałsz	Prawda
Fałsz	Prawda	Prawda
Prawda	Prawda	Prawda

Logiczna operacja OR jest obsługiwana przez operator ||.

Następnie przechodzimy do logicznej operacji XOR, która działa nieco odmiennie od OR. Wynikiem logicznej operacji XOR będzie wartość prawda (`true`), gdy tylko jeden dowolny operand będzie miał wartość `true`, co zostało przedstawione w tabeli 5.5.

Tabela 5.5. Wynik logicznej operacji XOR

Operand 1	Operand 2	Wynik Operand1 XOR Operand2
Fałsz	Fałsz	Fałsz
Prawda	Fałsz	Prawda
Fałsz	Prawda	Prawda
Prawda	Prawda	Fałsz

Język C++ zapewnia obsługę operacji XOR za pomocą operatora ^. Wymieniony operator pomaga w obliczeniu wyniku wygenerowanego przez operację XOR na bitach operandów.

Użycie w C++ operatorów logicznych NOT (!), AND (&&) i OR (||)

Spójrz na poniższe zdania:

- Jeżeli będzie padało *I* nie będzie jeździł autobus, wtedy nie mogę dotrzeć do pracy.
- Jeżeli otrzymam duży rabat *LUB* ogromną premię, wtedy mogę kupić ten samochód.

Tego rodzaju konstrukcje logiczne w programowaniu są potrzebne, jeśli wynik dwóch operacji jest używany w kontekście logicznym do podjęcia decyzji o dalszym sposobie działania programu. Język C++ oferuje logiczne operatory AND i OR, które można wykorzystać w poleceniach warunkowych i tym samym wpływać na dalsze działanie programu.

W programie, którego kod źródłowy można zobaczyć w listingu 5.5, pokazano użycie operatorów logicznych AND i OR.

Listing 5.5. Analiza operatorów logicznych && i || w C++

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Podaj wartość true(1) lub false(0) dla dwóch operandów:" <<
       ↪endl;
 6:    bool Op1 = false, Op2 = false;
 7:    cin >> Op1;
 8:    cin >> Op2;
 9:
10:    cout << Op1 << " AND " << Op2 << " = " << (Op1 && Op2) << endl;
11:    cout << Op1 << " OR " << Op2 << " = " << (Op1 || Op2) << endl;
12:
13:    return 0;
14: }
```

Wynik ▼

```
Podaj wartość true(1) lub false(0) dla dwóch operandów:
1
0
1 AND 0 = 0
1 OR 0 = 1
```

Następne uruchomienie programu:

```
Podaj wartość true(1) lub false(0) dla dwóch operandów:
1
1
1 AND 1 = 1
1 OR 1 = 1
```

Analiza ▼

Przedstawiony powyżej program w rzeczywistości pokazuje, jak operatory dostarczają programiście logiczne funkcje AND i OR. Natomiast program nie demonstruje, jak używać wspomnianych funkcji do podejmowania decyzji.

Przedstawiony w listingu 5.6 program wykonuje różne wiersze kodu, w zależności od wartości zmiennych. W tym programie wykorzystano przetwarzanie poleceń warunkowych i operatory logiczne.

Listing 5.6. Użycie operatorów logicznych NOT (!) i AND (&&) w poleceniach if w celu zapewnienia przetwarzania warunkowego

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Użyj wartości bool (0 / 1) jako odpowiedzi na pytania" <<
       ↪endl;
 6:    cout << "Czy pada deszcz? ";
 7:    bool Raining = false;
 8:    cin >> Raining;
 9:
10:    cout << "Czy po Twojej ulicy jeździ autobus? ";
11:    bool Buses = false;
12:    cin >> Buses;
13:
14:    // Polecenie warunkowe używa logicznych operatorów AND i NOT.
15:    if (Raining && !Buses)
16:      cout << "Nie możesz dojechać do pracy" << endl;
17:    else
18:      cout << "Możesz dojechać do pracy" << endl;
```

```
19:
20:    if (Raining && Buses)
21:      cout << "Zabierz parasol ze sobą" << endl;
22:
23:    if ((!Raining) && Buses)
24:       cout << "Ciesz się pogodą i miłego dnia" << endl;
25:
26:    return 0;
27: }
```

Wynik ▼

```
Użyj wartości bool (0 / 1) jako odpowiedzi na pytania
Czy pada deszcz? 1
Czy po Twojej ulicy jeździ autobus? 1
Możesz dojechać do pracy
Zabierz parasol ze sobą
```

Następne uruchomienie programu:

```
Użyj wartości bool (0 / 1) jako odpowiedzi na pytania
Czy pada deszcz? 1
Czy po Twojej ulicy jeździ autobus? 0
Nie możesz dojechać do pracy
```

Ostatnie uruchomienie programu:

```
Użyj wartości bool (0 / 1) jako odpowiedzi na pytania
Czy pada deszcz? 0
Czy po Twojej ulicy jeździ autobus? 1
Możesz dojechać do pracy
Ciesz się pogodą i miłego dnia
```

Analiza ▼

W programie przedstawionym w listingu 5.6 użyto poleceń warunkowych w postaci konstrukcji `if`, której jeszcze nie omawialiśmy w książce. Mimo to, spróbuj zrozumieć działanie wymienionej konstrukcji, porównując kod źródłowy programu z otrzymanymi danymi wyjściowymi. W wierszu 15. znajduje się wyrażenie logiczne (`Raining && !Buses`), które można odczytać następująco: „Pada deszcz I NIE ma autobusów”. Wymienione wyrażenie używa operatora logicznego AND do powiązania braku autobusów (na co wskazuje logiczne NOT) z wystąpieniem deszczu.

Uwaga

> Jeśli chcesz dowiedzieć się nieco więcej na temat konstrukcji `if`, możesz zajrzeć do lekcji 6.

W listingu 5.7 użyto logicznych operatorów NOT (!) i OR (||) w celu pokazania przetwarzania warunkowego.

Listing 5.7. Użycie operatorów logicznych NOT i OR w celu podjęcia decyzji o możliwości zakupu wymarzonego samochodu

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Na pytania udzielaj odpowiedzi 0 lub 1" << endl;
 6:    cout << "Czy na wymarzony samochód jest oferowany duży rabat? ";
 7:    bool Discount = false;
 8:    cin >> Discount;
 9:
10:    cout << "Czy otrzymałeś ostatnio dużą premię? ";
11:    bool FantasticBonus = false;
12:    cin >> FantasticBonus;
13:
14:    if (Discount || FantasticBonus)
15:        cout << "Gratulacje, możesz kupić ten samochód!" << endl;
16:    else
17:        cout << "Przykro mi, musisz jeszcze trochę poczekać" << endl;
18:
19:    return 0;
20: }
```

Wynik ▼

```
Na pytania udzielaj odpowiedzi 0 lub 1
Czy na wymarzony samochód jest oferowany duży rabat? 0
Czy otrzymałeś ostatnio dużą premię? 1
Gratulacje, możesz kupić ten samochód!
```

Następne uruchomienie programu:

```
Na pytania udzielaj odpowiedzi 0 lub 1
Czy na wymarzony samochód jest oferowany duży rabat? 0
Czy otrzymałeś ostatnio dużą premię? 0
Przykro mi, musisz jeszcze trochę poczekać
```

Ostatnie uruchomienie programu:

```
Na pytania udzielaj odpowiedzi 0 lub 1
Czy na wymarzony samochód jest oferowany duży rabat? 1
Gratulacje, możesz kupić ten samochód!
```

Analiza ▼

W wierszu 14. użyto konstrukcji `if`, po której w wierszu 16. mamy polecenie `else`. Konstrukcja `if` decyduje o wykonaniu polecenia w wierszu 15., o ile warunek (`Discount || FantasticBonus`) przyjmie wartość `true`. Ten warunek zawiera logiczny operator OR i przyjmuje wartość `true`, jeśli dostaniesz rabat na wymarzony samochód. W przypadku gdy wyrażenie przyjmie wartość `false`, wykonane będzie polecenie po `else`, czyli w wierszu 17.

Bitowe operatory NOT (~), AND (&), OR (|) i XOR (^)

Różnica pomiędzy operatorami logicznymi i bitowymi polega na tym, że wartość zwrotna operatorów bitowych nie jest typu `bool`. Zamiast tego operator bitowy generuje wynik, w którym stan poszczególnych bitów jest określany przez wykonanie operatora na bitach operandów. Język C++ pozwala na przeprowadzenie operacji, takich jak NOT, OR, AND i XOR, w trybie bitowym, gdzie można operować na poszczególnych bitach przy użyciu operatorów negacji (~), OR (|), AND (&) i XOR (^). Ostatnie trzy wymienione operacje są przeprowadzane względem wybranej liczby (zwykle maski bitowej).

Pewne operacje bitowe są użyteczne w sytuacjach, gdy bity znajdujące się w liczbach całkowitych określają stan poszczególnych flag. Dlatego też liczba całkowita o wielkości 32 bitów może być wykorzystana do ustalenia stanu 32 flag boolowskich. Użycie operatorów bitowych zademonstrowano w listingu 5.8.

Listing 5.8. Użycie operatorów bitowych do przeprowadzenia operacji NOT, AND, OR i XOR na poszczególnych bitach liczby całkowitej

```
 0: #include <iostream>
 1: #include <bitset>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Podaj liczbę (0 - 255): ";
 7:    unsigned short InputNum = 0;
 8:    cin >> InputNum;
 9:
10:    bitset<8> InputBits (InputNum);
11:    cout << InputNum << " to binarnie " << InputBits << endl;
```

```
12:
13:    bitset<8> BitwiseNOT = (~InputNum);
14:    cout << "Logiczne NOT |" << endl;
15:    cout << "~" << InputBits  << " = " << BitwiseNOT << endl;
16:
17:    cout << "Logiczne AND w połączeniu z 00001111" << endl;
18:    bitset<8> BitwiseAND = (0x0F & InputNum);// 0x0F is hex for 0001111
19:    cout << "0001111 & " << InputBits  << " = " << BitwiseAND << endl;
20:
21:    cout << "Logiczne OR w połączeniu z 00001111" << endl;
22:    bitset<8> BitwiseOR = (0x0F | InputNum);
23:    cout << "00001111 | " << InputBits  << " = " << BitwiseOR << endl;
24:
25:    cout << "Logiczne XOR w połączeniu z 00001111" << endl;
26:    bitset<8> BitwiseXOR = (0x0F ^ InputNum);
27:    cout << "00001111 ^ " << InputBits  << " = " << BitwiseXOR << endl;
28:
29:    return 0;
30: }
```

Wynik ▼

```
Podaj liczbę (0 - 255): 181
181 to binarnie 10110101
Logiczne NOT
~10110101 = 01001010
Logiczne AND w połączeniu z 00001111
0001111 & 10110101 = 00000101
Logiczne OR w połączeniu z 00001111
00001111 | 10110101 = 10111111
Logiczne XOR w połączeniu z 00001111
00001111 ^ 10110101 = 10111010
```

Analiza ▼

Przedstawiony program używa zestawu bitów — typu danych, który nie został jeszcze omówiony — w celu łatwiejszego wyświetlenia danych binarnych. Rola obiektu `std::bitset` polega jedynie na ułatwieniu wyświetlania danych i nic poza tym. W wierszach 10., 13., 17. i 22. następuje rzeczywiste przypisanie liczby całkowitej obiektowi `std::bitset`, który jest wykorzystywany do wyświetlenia tych samych danych, ale w trybie binarnym. Wspomniane operacje są przeprowadzane na liczbach całkowitych. Na początek skoncentrujemy się na danych wyjściowych. Drugi wiersz danych wyjściowych pokazuje liczbę całkowitą 181 podaną przez użytkownika, liczba ta została wyświetlona binarnie. Następnie przechodzimy do pokazania efektu użycia różnych

operatorów bitowych (~, &, | i ^) względem wymienionej liczby. Jak możesz się przekonać, bitowe NOT w wierszu 14. powoduje odwrócenie poszczególnych bitów. Program pokazuje również, jak działają operatory &, | i ^ — przeprowadzane są operacje względem wszystkich bitów obu operandów w celu wygenerowania wyniku. Otrzymane dane wyjściowe możesz porównać z przedstawionymi wcześniej w rozdziale tabelami, co powinno pomóc w zrozumieniu działania operatorów bitowych.

Uwaga

> Jeżeli chcesz dowiedzieć się więcej na temat przeprowadzania operacji na flagach bitowych w C++, zajrzyj do lekcji 25., zatytułowanej „Praca z opcjami bitowymi za pomocą STL", gdzie znajdziesz szczegółowe omówienie `std::bitset`.

Operatory bitowego przesunięcia w prawo (>>) oraz w lewo (<<)

Operatory przesunięcia powodują przenoszenie całych sekwencji bitów w lewo lub w prawo, a tym samym — poza innymi zastosowaniami w aplikacji — mogą pomóc np. w operacjach dzielenia lub mnożenia przez dwa.

Przykładowe użycie operatora przesunięcia w celu przeprowadzenia operacji mnożenia przez dwa przedstawiono poniżej:

```
int DoubledValue = Num << 1; // Przesunięcie bitów o jedną pozycję w lewo
                             // w celu podwojenia wartości.
```

Z kolei poniżej pokazano użycie operatora przesunięcia do podzielnia wartości przez dwa:

```
int HalvedValue = Num >> 2; // Przesunięcie bitów o jedną pozycję w prawo
                            // w celu podzielenia wartości przez dwa.
```

W programie przedstawionym w listingu 5.9 pokazano, jak można użyć operatorów przesunięcia w celu efektywnego mnożenia lub dzielenia liczby całkowitej.

Listing 5.9. Użycie operatora bitowego przesunięcia w prawo (>>) w celu obliczenia jednej czwartej i jednej drugiej danej wartości oraz operatora bitowego przesunięcia w lewo (<<) w celu obliczenia dwukrotności i czterokrotności wartości na podstawie danej liczby całkowitej

```
0: #include <iostream>
1: using namespace std;
2:
```

```
 3: int main()
 4: {
 5:    cout << "Podaj liczbę: ";
 6:    int Input = 0;
 7:    cin >> Input;
 8:
 9:    int Half = Input >> 1;
10:    int Quarter = Input >> 2;
11:    int Double = Input << 1;
12:    int Quadruple = Input << 2;
13:
14:    cout << "Jedna czwarta: " << Quarter << endl;
15:    cout << "Połowa: " << Half << endl;
16:    cout << "Dwukrotność: " << Double << endl;
17:    cout << "Czterokrotność: " << Quadruple << endl;
18:
19:    return 0;
20: }
```

Wynik ▼

```
Podaj liczbę: 16
Jedna czwarta: 4
Połowa: 8
Dwukrotność: 32
Czterokrotność: 64
```

Analiza ▼

Podana przez użytkownika liczba całkowita to 16, w postaci binarnej 1000. W wierszu 9. następuje przesunięcie o jeden bit w prawo i zmiana na 0100, czyli 8 dziesiętnie, co oznacza połowę wartości początkowej. W wierszu 10. następuje przesunięcie dwóch bitów w prawo, czyli zmiana 1000 na 00100 (dziesiętnie 4). Podobny efekt działania bitowego operatora przesunięcia w lewo w wierszach 11. i 12. daje dokładnie odwrotny wynik. Przesunięcie o jeden bit w lewo daje 10000, czyli 32 dziesiętnie, natomiast o dwa bity w lewo daje 100000, czyli 64 dziesiętnie, a więc efektywnie (odpowiednio) dwukrotność i czterokrotność wartości początkowej.

Uwaga

Operatory bitowego przesunięcia nie powodują rotacji wartości. Ponadto wynik przesunięcia liczb ze znakiem zależy od konkretnej implementacji. W pewnych kompilatorach najbardziej znaczący bit (MSB) po przesunięciu w lewo nie będzie przypisany najmniej znaczącemu bitowi (LSB), zamiast tego drugi z wymienionych będzie miał wartość zero.

Złożone operatory przypisania

Złożone operatory przypisania to operatory przypisania, w których operandowi po lewej stronie zostaje przypisana wartość przeprowadzonej operacji.

Spójrz na poniższy fragment kodu:

```
int Num1 = 22;
int Num2 = 5;
Num1 += Num2; // Po operacji Num1 ma wartość 27.
```

Powyższa operacja daje taki sam wynik jak przedstawiona poniżej:

```
Num1 = Num1 + Num2;
```

Efektem działania operatora += jest obliczenie sumy dwóch operandów, a następnie przypisanie jej operandowi po lewej stronie (tutaj Num1). W tabeli 5.6 wymieniono wiele złożonych operatorów przypisania i wyjaśniono ich działanie.

Tabela 5.6. Złożone operatory przypisania

Operator	Użycie	Odpowiednik
Dodawanie i przypisanie	`Num1 += Num2`	`Num1 = Num1 + Num2`
Odejmowanie i przypisanie	`Num1 -= Num2`	`Num1 = Num1 - Num2`
Mnożenie i przypisanie	`Num1 *= Num2`	`Num1 = Num1 * Num2`
Dzielenie i przypisanie	`Num1 /= Num2`	`Num1 = Num1 / Num2`
Reszta z dzielenia i przypisanie	`Num1 %= Num2`	`Num1 = Num1 % Num2`
Bitowe przesunięcie w lewo i przypisanie	`Num1 <<= Num2`	`Num1 = Num1 << Num2`
Bitowe przesunięcie w prawo i przypisanie	`Num1 >>= Num2`	`Num1 = Num1 >> Num2`
Bitowe AND i przypisanie	`Num1 &= Num2`	`Num1 = Num1 & Num2`
Bitowe OR i przypisanie	`Num1 |= Num2`	`Num1 = Num1 | Num2`
Bitowe XOR i przypisanie	`Num1 ^= Num2`	`Num1 = Num1 ^ Num2`

Przykład zastosowania złożonych operatorów przypisania pokazano w listingu 5.10.

Listing 5.10. Użycie złożonych operatorów przypisania w celu wykonania operacji dodawania, odejmowania, dzielenia, mnożenia, obliczenia reszty z dzielenia oraz przeprowadzenia bitowych operacji OR, AND i XOR

```
0: #include <iostream>
1: using namespace std;
2:
```

```
 3: int main()
 4: {
 5:   cout << "Podaj liczbę: ";
 6:   int Value = 0;
 7:   cin >> Value;
 8:
 9:   Value += 8;
10:   cout << "Po += 8, wartość = " << Value << endl;
11:   Value -= 2;
12:   cout << "Po -= 2, wartość = " << Value << endl;
13:   Value /= 4;
14:   cout << "Po /= 4, wartość = " << Value << endl;
15:   Value *= 4;
16:   cout << "Po *= 4, wartość = " << Value << endl;
17:   Value %= 1000;
18:   cout << "Po %= 1000, wartość = " << Value << endl;
19:
20:   // Uwaga: przypisane w ramach cout.
21:   cout << "Po <<= 1, wartość = " << (Value <<= 1) << endl;
22:   cout << "Po >>= 2, wartość = " << (Value >>= 2) << endl;
23:
24:   cout << "Po |= 0x55, wartość = " << (Value |= 0x55) << endl;
25:   cout << "Po ^= 0x55, wartość = " << (Value ^= 0x55) << endl;
26:   cout << "Po &= 0x0F, wartość = " << (Value &= 0x0F) << endl;
27:
28:   return 0;
29:}
```

Wynik ▼

```
Podaj liczbę: 440
Po += 8, wartość = 448
Po -= 2, wartość = 446
Po /= 4, wartość = 111
Po *= 4, wartość = 444
Po %= 1000, wartość = 444
Po <<= 1, wartość = 888
Po >>= 2, wartość = 222
Po |= 0x55, wartość = 223
Po ^= 0x55, wartość = 138
Po &= 0x0F, wartość = 10
```

Analiza ▼

Zwróć uwagę, że w programie `wartość` ulega ciągłym zmianom podczas wykonywania kolejnych operacji. Każda operacja jest przeprowadzana z użyciem `wartości`, a jej wynik zostaje ponownie przypisywany wymienionej zmiennej.

Dlatego też w wierszu 9. do podanej przez użytkownika liczby 440 następuje dodanie liczby 8, co daje liczbę 448 przypisaną zmiennej `wartość`. W kolejnej operacji (wiersz 11.) od liczby 448 następuje odjęcie liczby 2, co daje w wyniku 446 przypisane ponownie `wartości` itd.

Użycie operatora sizeof w celu określenia ilości pamięci zajmowanej przez zmienną

Operator `sizeof` informuje o wyrażonej w bajtach ilości pamięci zajmowanej przez określony typ danych lub zmienną. Sposób użycia wymienionego operatora przedstawiono poniżej:

```
sizeof (zmienna);
```

lub

```
sizeof (typ);
```

Uwaga

> Operator `sizeof(...)` może wyglądać jak wywołanie funkcji, ale nie jest funkcją, to operator. Co interesujące, operator ten nie może być zdefiniowany przez użytkownika, a więc niemożliwe jest jego przeciążenie.
>
> Więcej informacji na temat definiowania własnych operatorów znajdziesz w lekcji 12., zatytułowanej „Typy operatorów i ich przeciążanie”.

W listingu 5.11 zademonstrowano użycie operatora `sizeof` w celu określenia ilości pamięci zajmowanej przez tablicę. Ponadto możesz powrócić do listingu 3.4 i przeanalizować użycie `sizeof` w celu określenia ilości pamięci zajmowanej przez najczęściej stosowane typy zmiennych.

Listing 5.11. Użycie operatora sizeof w celu określenia liczby bajtów zajmowanych przez tablicę 100 liczb całkowitych

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    cout << "Użycie operatora sizeof do określenia ilości pamięci
      ↪zajmowanej przez tablicę" << endl;
6:    int MyNumbers [100] = {0};
7:
8:    cout << "Liczba bajtów zajmowana przez int: " << sizeof(int) <<
      ↪endl;
```

```
 9:    cout << "Liczba bajtów zajmowana przez tablicę MyNumbers: " <<
       ↪sizeof(MyNumbers) << endl;
10:    cout << "Liczba bajtów zajmowana przez każdy element: " <<
       ↪sizeof(MyNumbers[0]) << endl;
11:
12:    return 0;
13: }
```

Wynik ▼

```
Użycie operatora sizeof do określenia ilości pamięci zajmowanej przez tablicę
Liczba bajtów zajmowana przez int: 4
Liczba bajtów zajmowana przez tablicę MyNumbers: 400
Liczba bajtów zajmowana przez każdy element: 4
```

Analiza ▼

W powyższym programie pokazano, że przy użyciu operatora `sizeof` można określić ilość pamięci zajmowanej przez tablicę składającą się ze 100 liczb całkowitych (400 bajtów). Dane wyjściowe programu wskazują także, że wielkość poszczególnych elementów wynosi 4 bajty.

Operator `sizeof` może być szczególnie użyteczny, gdy trzeba dynamicznie alokować pamięć dla *N* obiektów, zwłaszcza o typie utworzonym przez programistę. Wynik działania operatora `sizeof` można wykorzystać do ustalenia ilości pamięci zajmowanej przez każdy obiekt, a następnie dynamicznie zaalokować pamięć za pomocą słowa kluczowego `new`.

Temat dynamicznej alokacji pamięci zostanie szczegółowo omówiony w lekcji 8., zatytułowanej „Wskaźniki i referencje”.

Pierwszeństwo operatorów

W szkole poznałeś kolejność operatorów arytmetycznych określającą kolejność obliczania skomplikowanych wyrażeń arytmetycznych.

W języku C++ używasz operatorów i wyrażeń w następujący sposób:

```
int MyNumber = 10 * 30 + 20 – 5 * 5 << 2;
```

Pytanie brzmi: jaka jest wartość zmiennej `MyNumber`? Tutaj nie ma miejsca na zgadywanie, ponieważ kolejność wywoływania poszczególnych operatorów jest ściśle określona przez standard C++. Wspomniana kolejność to właśnie pierwszeństwo operatorów (patrz tabela 5.7).

Tabela 5.7. Pierwszeństwo operatorów

Miejsce	Nazwa	Operator
1	Określenie zakresu	`::`

2	Wybór elementu składowego, indeks, wywołanie funkcji, inkrementacja i dekrementacja postfiksowa	`. ->`
		`()`
		`++ --`

3	Operator `sizeof`, inkrementacja i dekrementacja prefiksowa, uzupełnienia, `AND`, `NOT`, jednoargumentowy minus i plus, adres i odniesienie, `new`, `new[]`, `delete`, `delete[]`, rzutowanie, funkcja `sizeof()`	`++ --`
		`^ !`
		`- +`
		`& *`
		`()`

4	Wybór elementu składowego dla wskaźnika	`.* ->*`
5	Mnożenie, dzielenie, reszta z dzielenia	`* / %`
6	Dodawanie, odejmowanie	`+ -`
7	Przesunięcie (w lewo i w prawo)	`<< >>`
8	Nierówność	`< <= > >=`
9	Równość, nierówność	`== !=`
10	Bitowe `AND`	`&`
11	Bitowe `XOR`	`^`
12	Bitowe `OR`	`|`
13	Logiczne `AND`	`&&`
14	Logiczne `OR`	`||`
15	Warunek	`?:`

16	Operatory przypisania	`= *= /= %=`
		`+= -= <<=`
		`>>==`
		`&= |= ^=`

17	Przecinek	`,`

Spójrz jeszcze na inne skomplikowane wyrażenie, które zostało użyte w jednym z wcześniejszych przykładów:

```
int MyNumber = 10 * 30 + 20 – 5 * 5 << 2;
```

Podczas obliczania wartości powyższego wyrażenia konieczne jest zastosowanie reguł pierwszeństwa przedstawionych w tabeli 5.7, aby zrozumieć, jakie wartości będą przypisywane przez kompilator. Ponieważ mnożenie i dzielenie mają pierwszeństwo przed dodawaniem i odejmowaniem, które z kolei mają pierwszeństwo przez przesunięciem, wyrażenie można uprościć do postaci:

```
int MyNumber = 300 + 20 – 25 << 2;
```

Skoro dodawanie i odejmowanie ma pierwszeństwo przed przesunięciem, całość można uprościć do postaci:

```
int MyNumber = 295 << 2;
```

Teraz można już przeprowadzić operację przesunięcia. Wiedząc, że przesunięcie o jeden bit w lewo powoduje podwojenie wartości, natomiast o dwa bity w lewo daje czterokrotność wartości początkowej, możemy podać wartość końcową wyrażenia: 295 * 4 = 1180.

Ostrzeżenie

Używaj nawiasów, ponieważ ułatwiają odczyt kodu.

Przedstawione wcześniej wyrażenie zostało kiepsko zapisane. Wprawdzie kompilator bez problemów je zinterpretuje, ale powinieneś tworzyć kod, który będzie łatwy do odczytu przez człowieka.

Dlatego też wcześniejsze wyrażenie można zapisać w znacznie lepszej i czytelniejszej postaci:

```
int MyNumber = ((10 * 30) – (5 * 5) + 20) << 2;
// Rozwianie wszelkich wątpliwości.
```

TAK	NIE
Używaj nawiasów, aby zapewnić większą przejrzystość kodu źródłowego i wyrażeń.	Nie twórz skomplikowanych wyrażeń, opierając się na tabeli pierwszeństwa operatorów; pisany kod powinien być czytelny również dla ludzi.
Używaj odpowiednich typów zmiennych i zagwarantuj, że nigdy nie dojdzie do sytuacji przepełnienia.	Nie myl operatorów `++Zmienna` ze `Zmienna++`, myśląc, że mają takie samo działanie, ponieważ one różnią się w operacjach przypisania.
Pamiętaj, że wszystkie l-wartości (np. zmienne) mogą być r-wartościami, natomiast nie wszystkie r-wartości (np. „Witaj, świecie") mogą być l-wartościami.	

Podsumowanie

W tej lekcji dowiedziałeś się, czym w C++ są polecenia, wyrażenia i operatory. Nauczyłeś się przeprowadzać w C++ podstawowe operacje arytmetyczne, takie jak dodawanie, odejmowanie, mnożenie i dzielenie. Poznałeś także ogólnie operatory logiczne NOT, AND, OR i XOR. Dowiedziałeś się, jak w C++ operatory logiczne `!`, `&&` i `||` mogą pomóc w poleceniach warunkowych oraz jak operatory bitowe `~`, `&`, `|` i `^` pomagają w operacjach na danych, jednorazowo po jednym bicie.

Poznałeś także pierwszeństwo operatorów, a także wagę używania nawiasów w celu tworzenia kodu łatwego do odczytu przez programistów. Ogólnie poruszony został również temat przepełnienia liczb całkowitych i dowiedziałeś się, jak ważne jest unikanie takich sytuacji.

Pytania i odpowiedzi

Pytanie: Dlaczego niektórzy programiści używają typu `unsigned int` skoro typ `unsigned short` zabiera mniej pamięci i także umożliwia kompilację programu?

Odpowiedź: Typ `unsigned short` najczęściej oznacza ograniczenie do wartości 65 535, po inkrementacji której nastąpi przepełnienie i „przeskok" do zera. Aby uniknąć takiej sytuacji, w dobrze zaprojektowanych aplikacjach używany jest typ `unsigned int`, jeśli nie ma pewności, czy wartość nie wykroczy poza wymienioną granicę.

Pytanie: Muszę obliczyć dwukrotność danej liczby po jej podzieleniu przez trzy. Czy w poniższym kodzie widzisz jakikolwiek błąd?

```
int result = Number / 3 << 1;
```

Odpowiedź: Tak! Dlaczego po prostu nie użyjesz nawiasów w celu uproszczenia wyrażenia i ułatwienia jego odczytu przez innych programistów? Dodanie komentarza również nie zaszkodzi.

Pytanie: Poniższy kod jest przeznaczony do podzielenia dwóch liczb całkowitych (5 i 2).

```
int Num1 = 5, Num2 =  2;
int result = Num1 / Num2;
```

Po jego wykonaniu wynikiem jest 2. Czy w kodzie jest błąd?

Odpowiedź: Niekoniecznie. Liczby całkowite nie przechowują informacji o części dziesiętnej i dlatego też wynikiem przedstawionej operacji dzielenia jest 2 zamiast 2.5. Jeżeli oczekujesz wyniku 2.5, powinieneś zmienić typ danych na `float` lub `double`. Wymienione typy są przeznaczone do obsługi operacji zmiennoprzecinkowych (z uwzględnieniem części dziesiętnej).

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy omówionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Tworzę program przeznaczony do dzielenia liczb. Który typ danych będzie lepszy: `int` czy `float`?
2. Jaki jest wynik operacji 32 / 7?
3. Jaki jest wynik operacji 32.0 / 7?
4. Czy `sizeof(...)` jest funkcją?
5. Muszę obliczyć podwójną wartość liczby, dodać do niej 5, a następnie podwoić. Czy poniższe polecenie jest prawidłowe?

   ```
   int Result = number << 1 + 5 << 1;
   ```

6. Jaki będzie wynik operacji XOR, jeśli oba operandy XOR będą miały wartość `true`?

Ćwiczenia

1. Popraw kod przedstawiony w pytaniu 5. quizu i użyj nawiasów, aby zwiększyć jego czytelność.
2. Jaka będzie wartość zmiennej `Result` w poniższym wyrażeniu?

   ```
   int Result = number << 1 + 5 << 1;
   ```

3. Utwórz program, który prosi użytkownika o podanie dwóch wartości boolowskich, a następnie wyświetla wyniki różnych operacji bitowych przeprowadzonych na podanych wartościach.

Lekcja 6

Sterowanie przebiegiem działania programu

Większość aplikacji powinna być tak zbudowana, by podejmować odpowiednie działania, w zależności od różnych sytuacji lub danych wejściowych wprowadzanych przez użytkownika. Aby umożliwić aplikacji odpowiednią reakcję, konieczne jest zastosowanie poleceń warunkowych wykonujących różne segmenty kodu w konkretnych sytuacjach.

Z tej lekcji dowiesz się:

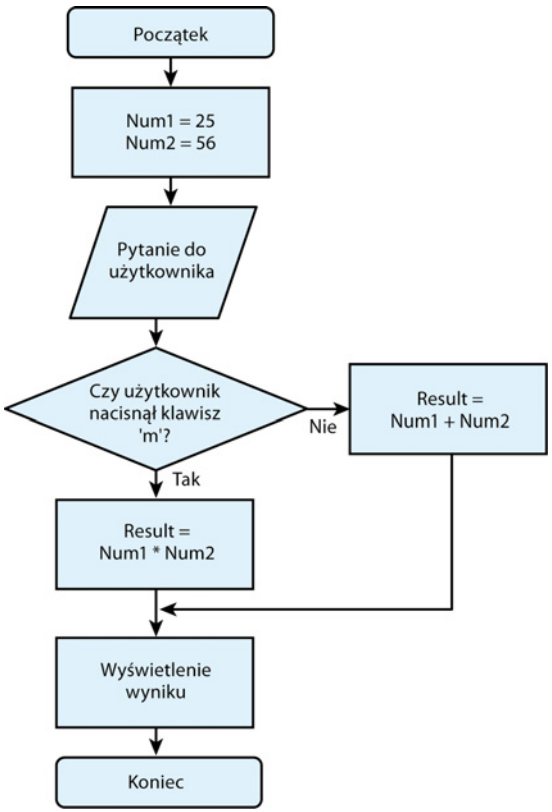
- **jak spowodować, aby program zachowywał się odpowiednio do danej sytuacji,**
- **jak wielokrotnie wykonywać sekcję kodu w pętli,**
- **jak zapewnić lepszą kontrolę sterowaniem przepływu w pętli.**

Wykonanie warunkowe za pomocą if-else

Przedstawione dotąd programy miały zdefiniowaną szeregową kolejność wykonywania poleceń, od początku do końca. Każdy wiersz programu był wykonywany i żaden nie został zignorowany. Jednak w większości aplikacji rzadko się zdarza kolejne wykonywanie wszystkich wierszy kodu.

Wyobraź sobie program mnożący dwie liczby, gdy użytkownik naciśnie klawisz *m*, lub dodający dwie liczby po naciśnięciu innego klawisza.

Jak możesz zobaczyć na rysunku 6.1, nie wszystkie wiersze kodu będą wykonane podczas każdego uruchomienia aplikacji. Jeżeli użytkownik naciśnie klawisz *m*, nastąpi wykonanie kodu odpowiedzialnego za przeprowadzenie operacji mnożenia dwóch liczb. Naciśnięcie klawisza innego niż *m* oznacza wykonanie kodu przeprowadzającego operację dodawania dwóch liczb. Nigdy nie zostaną wykonane obie wymienione gałęzie kodu źródłowego.



RYSUNEK 6.1. Przykład przetwarzania warunkowego odbywającego się na podstawie danych wejściowych użytkownika

Programowanie warunkowe z użyciem if-else

Warunkowe wykonanie kodu jest implementowane w C++ za pomocą konstrukcji `if-else` w przedstawiony poniżej sposób:

```
if (wyrażenie warunkowe)
   Operacja do wykonania, gdy wyrażenie przyjmie wartość true;
else     // Opcjonalnie.
   Inna operacja do wykonania, gdy wyrażenie przyjmie wartość false;
```

Poniżej przedstawiono przykład konstrukcji `if-else` pozwalającej na wykonanie operacji mnożenia, gdy użytkownik naciśnie klawisz *m*, lub operacji dodawania po naciśnięciu innego dowolnego klawisza:

```
if (UserSelection == 'm')
   Result = Num1 * Num2;  // Mnożenie.
else
   Result = Num1 + Num2;  // Dodawanie.
```

Uwaga

Pamiętaj, że jeśli w C++ wyrażenie przyjmie wartość `true`, oznacza to, że wartością wyrażenia nie jest `false` (przy czym `false` ma wartość zero). Dlatego też w poleceniu warunkowym dowolna niezerowa — dodatnia lub ujemna — wartość wyrażenia jest uznawana za `true`.

Przedstawiona wcześniej konstrukcja znajduje się w listingu 6.1 zawierającym kod programu, który po uruchomieniu pozwala użytkownikowi na wybór operacji: mnożenie lub dodawanie. Zatem z wykorzystaniem przetwarzania warunkowego następuje wygenerowanie żądanych danych wyjściowych.

Listing 6.1. Mnożenie lub dodawanie dwóch liczb całkowitych w zależności od danych wejściowych użytkownika

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Podaj dwie liczby całkowite: " << endl;
 6:    int Num1 = 0, Num2 = 0;
 7:    cin >> Num1;
 8:    cin >> Num2;
 9:
10:    cout << "Naciśnij \'m\', aby wykonać mnożenie, inny klawisz oznacza
       ↪dodawanie: ";
11:    char UserSelection = '\0';
12:    cin >> UserSelection;
```

```
13:
14:    int Result = 0;
15:    if (UserSelection == 'm')
16:       Result = Num1 * Num2;
17:    else
18:       Result = Num1 + Num2;
19:
20:    cout << "Wynik operacji: " << Result << endl;
21:
22:    return 0;
23: }
```

Wynik ▼

```
Podaj dwie liczby całkowite:
25
56
Naciśnij 'm', aby wykonać mnożenie, inny klawisz oznacza dodawanie: m
Wynik operacji: 1400
```

Następne uruchomienie programu:

```
Podaj dwie liczby całkowite:
25
56
Naciśnij 'm', aby wykonać mnożenie, inny klawisz oznacza dodawanie: a
Wynik operacji: 81
```

Analiza ▼

Zwróć uwagę na użycie poleceń `if` (wiersz 15.) i `else` (wiersz 17.). Jeżeli w wierszu 15. wartością wyrażenia (`UserSelection == 'm'`) będzie `true`, zostanie przeprowadzona operacja mnożenia, natomiast wartość `false` oznacza wykonanie operacji dodawania. Wymienione wyrażenie będzie miało przypisaną wartość `true`, jeśli użytkownik naciśnie klawisz *m* (wielkość litery ma znaczenie), w przeciwnym razie to będzie `false`. Ten prosty program pasuje do wykresu pokazanego wcześniej na rysunku 6.1 i pokazuje, że aplikacja może zachowywać się odmiennie w różnych sytuacjach.

Uwaga

Część `else` w konstrukcji `if-else` jest opcjonalna i nie musi być używana, gdy nie ma żadnej operacji do wykonania, jeśli wartością wyrażenia jest `false`.

Ostrzeżenie

Jeżeli w listingu 6.1 wiersz 15. miałby postać:

```
15:    if (UserSelection == 'm');
```

wówczas konstrukcja `if` byłaby zbędna, ponieważ kończy się w tym samym wierszu i jest pustym poleceniem. Zachowaj ostrożność i unikaj tego rodzaju sytuacji — brak dołączonego bloku `else` do konstrukcji `if` nie powoduje wygenerowania błędu w trakcie kompilacji.

W przedstawionej sytuacji niektóre dobre kompilatory wyświetlają ostrzeżenie informujące o „pustym poleceniu sterującym".

Warunkowe wykonanie wielu poleceń

Gdy warunek przyjmie wartość `true` lub `false` i chcesz wykonać wiele poleceń, wtedy polecenia powinieneś umieścić w bloku. Wymieniony blok to polecenia ujęte w nawiasy klamrowe {...} i wykonywane jako całość, np.:

```
if (warunek)
{
   // Blok do wykonania, gdy warunek okaże się prawdziwy.
   Polecenie 1;
   Polecenie 2;
}
else
{
   // Blok do wykonania, gdy warunek okaże się fałszywy.
   Polecenie 3;
   Polecenie 4;
}
```

Tego rodzaju bloki są nazywane również poleceniami złożonymi.

W lekcji 4., zatytułowanej „Tablice i ciągi tekstowe", wyjaśniono zagrożenia płynące z użycia tablic statycznych i przekraczania ich granic. Ten problem bardzo często występuje w przypadku tablic znaków. Podczas zapisu ciągu tekstowego w tablicy znaków lub jego kopiowania do tablicy konieczne jest sprawdzenie, czy tablica docelowa ma na tyle dużą pojemność, aby zmieścić dany ciąg tekstowy. W listingu 6.2 pokazano, jak przeprowadzić tę bardzo ważną operację sprawdzania, chroniącą przed przepełnieniem bufora.

Listing 6.2. Sprawdzenie pojemności przed rozpoczęciem kopiowania ciągu tekstowego do tablicy znaków

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
```

```
 3:
 4: int main()
 5: {
 6:    char Buffer[20] = {'\0'};
 7:
 8:   cout << "Podaj wiersz tekstu: " << endl;
 9:   string LineEntered;
10:   getline (cin, LineEntered);
11:
12:   if (LineEntered.length() < 20)
13:   {
14:       strcpy(Buffer, LineEntered.c_str());
15:       cout << "Zawartość bufora: " << Buffer << endl;
16:   }
17:
18:   return 0;
19: }
```

Wynik ▼

```
Podaj wiersz tekstu:
Krótki tekst!
Zawartość bufora: Zawartość bufora!
```

Analiza ▼

W wierszu 12. następuje sprawdzenie długości ciągu tekstowego i jego porównanie z wielkością bufora przed przeprowadzeniem operacji kopiowania ciągu tekstowego do bufora. O wyjątkowości tej konstrukcji `if` świadczy obecność bloku (nazywanego także poleceniem złożonym) obejmującego wiersze od 13. do 16. i wykonywanego, jeśli wyrażenie przyjmie wartość `true`.

Ostrzeżenie

Zwróć uwagę, że wiersz `if(warunek)` nie posiada na końcu średnika. To celowe i gwarantuje wykonanie znajdującego się dalej polecenia, gdy wyrażenie przyjmie wartość `true`.

Przedstawiony poniżej fragment kodu

```
if(warunek);
   polecenie;
```

zostanie skompilowany, ale nie będzie działał zgodnie z oczekiwaniami, ponieważ na końcu polecenia `if` widnieje średnik. Dlatego też polecenie znajdujące się tuż po konstrukcji `if` będzie wykonywane zawsze, niezależnie od wartości wyrażenia `if`.

Zagnieżdżone polecenia if

Często zdarzają się sytuacje, gdy trzeba przeprowadzić sprawdzenie względem wielu różnych warunków, z których część może zależeć od wartości poprzedniego. Język C++ pozwala na stosowanie zagnieżdżonych poleceń `if` w celu wykonania wymienionego zadania.

Poniżej przedstawiono przykład zagnieżdżonych poleceń `if`:

```
if (wyrażenie1)
{
   WykonajOperację1;
   if(wyrażenie2)
      WykonajOperację2;
   else
      WykonajInnąOperację2;
}
else
   WykonajInnąOperację1;
```

Rozważ aplikację podobną do przedstawionej w listingu 6.1, w której użytkownik, naciskając odpowiedni klawisz (*m* lub *d*), wybiera rodzaj wykonywanej operacji: mnożenie lub dzielenie. Dzielenie powinno być dozwolone tylko wtedy, gdy dzielnik ma wartość inną niż zero. Dlatego też, oprócz sprawdzenia danych wejściowych użytkownika wskazujących rodzaj żądanej operacji, po wyborze dzielenia konieczne jest upewnienie się, że dzielnik ma wartość niezerową. W listingu 6.3 do wykonania tego zadania użyto zagnieżdżonych konstrukcji `if`.

Listing 6.3. Użycie zagnieżdżonych poleceń if podczas mnożenia lub dzielenia liczb

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Podaj dwie liczby: " << endl;
 6:    float Num1 = 0, Num2 = 0;
 7:    cin >> Num1;
 8:    cin >> Num2;
 9:
10:    cout << "Naciśnij 'd', aby wykonać dzielenie, inny klawisz oznacza
       ↪mnożenie: ";
11:    char UserSelection = '\0';
12:    cin >> UserSelection;
```

```
13:
14:    if (UserSelection == 'd')
15:    {
16:       cout << "Wybrałeś dzielenie!" << endl;
17:       if (Num2 != 0)
18:      {
19:          cout << "To nie jest dzielenie przez zero, przechodzę
               ↪do obliczeń" << endl;
20:        cout << Num1 << " / " << Num2 << " = " << Num1 / Num2 << endl;
21:      }
22:       else
23:          cout << "Nie wolno dzielić przez zero" << endl;
24:    }
25:    else
26:    {
27:       cout << "Wybrałeś mnożenie!" << endl;
28:      cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
29:    }
30:
31:    return 0;
32: }
```

Wynik ▼

```
Podaj dwie liczby:
45
9
Naciśnij 'd', aby wykonać dzielenie, inny klawisz oznacza mnożenie: m
Wybrałeś mnożenie!
45 x 9 = 405
```

Następne uruchomienie programu:

```
Podaj dwie liczby:
22
7
Naciśnij 'd', aby wykonać dzielenie, inny klawisz oznacza mnożenie: d
Wybrałeś dzielenie!
To nie jest dzielenie przez zero, przechodzę do obliczeń
22 / 7 = 3.14286
```

Ostatnie uruchomienie programu:

```
Podaj dwie liczby:
365
0
Naciśnij 'd', aby wykonać dzielenie, inny klawisz oznacza mnożenie: d
Wybrałeś dzielenie!
Nie wolno dzielić przez zero
```

Analiza ▼

Dane wyjściowe pokazują wyniki trzykrotnego uruchomienia programu, za każdym razem z innym zestawem danych wejściowych. Jak możesz zobaczyć, program wykonywał różne fragmenty kodu, w zależności od wyborów użytkownika. W porównaniu z programem w listingu 6.1, w przedstawionym powyżej mamy kilka istotnych zmian. Oto one.

- Akceptowane liczby są umieszczane w zmiennych typu liczby zmiennoprzecinkowe, to pozwala na lepszą obsługę części dziesiętnych, co ma znaczenie podczas dzielenia liczb.
- Konstrukcja `if` jest inna niż użyta w listingu 6.1. Nie jest przeprowadzane sprawdzanie, czy użytkownik nacisnął klawisz *m*. Zamiast tego wiersz 14. zawiera wyrażenie `(UserSelection == 'd')`, które przyjmuje wartość `true`, jeśli użytkownik nacisnął klawisz *d*. W takim przypadku będzie wykonana operacja dzielenia.
- Jeżeli użytkownik wybierze operację dzielenia dwóch podanych liczb, bardzo ważne jest sprawdzenie, czy dzielnik ma wartość inną niż zero. Do tego celu służy zagnieżdżone polecenie `if` znajdujące się w wierszu 17.

W omówionym programie pokazano, jak bardzo użyteczne mogą być zagnieżdżone konstrukcje `if` w wykonywaniu różnych zadań zależących od wartości wielu parametrów.

> Zagnieżdżone znaki odstępu zastosowane w kodzie są opcjonalne, ale ich użycie znacząco poprawia czytelność zagnieżdżonych konstrukcji `if`.

Wskazówka

Zwróć uwagę na możliwość zgrupowania konstrukcji `if-else`. W listingu 6.4 przedstawiono kod źródłowy programu, który prosi użytkownika o wskazanie dnia tygodnia. Następnie za pomocą zgrupowanych konstrukcji `if-else` program podaje nazwę, od której pochodzi nazwa wybranego przez użytkownika dnia tygodnia.

Listing 6.4. Program, który podaje nazwę, od której pochodzi nazwa dnia tygodnia

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
```

```
 5:    enum DaysOfWeek
 6:    {
 7:       niedziela = 0,
 8:       poniedziałek,
 9:       wtorek,
10:       środa,
11:       czwartek,
12:       piątek,
13:       sobota
14:    };
15:
16:    cout << "Podanie nazwy, od której pochodzi nazwa dnia tygodnia!" <<
       ↪endl;
17:    cout << "Podaj numer dnia tygodnia (niedziela = 0): ";
18:
19:    int Day = niedziela;    // Inicjalizacja dnia z wartością niedziela.
20:    cin >> Day;
21:
22:    if (Day == niedziela)
23:       cout << "Nazwa niedziela oznacza dzień wolny (nie działać)" <<
          ↪endl;
24:    else if (Day == poniedziałek)
25:      cout << "Nazwa poniedziałek oznacza dzień po niedzieli" << endl;
26:    else if (Day == wtorek)
27:       cout << "Nazwa wtorek oznacza dzień wtóry, czyli drugi
          ↪po niedzieli" << endl;
28:    else if (Day == środa)
29:       cout << "Nazwa środa oznacza środkowy dzień tygodnia" << endl;
30:    else if (Day == czwartek)
31:      cout << "Nazwa czwartek oznacza czwarty dzień po niedzieli" <<
         ↪endl;
32:    else if (Day == piątek)
33:       cout << "Nazwa piątek oznacza piąty dzień po niedzieli" << endl;
34:    else if (Day == sobota)
35:       cout << "Nazwa sobota pochodzi od szabatu" << endl;
36:    else
37:       cout << "Błędne dane wejściowe, spróbuj raz jeszcze" << endl;
38:
39:    return 0;
40: }
```

Wynik ▼

```
Podanie nazwy, od której pochodzi nazwa dnia tygodnia!
Podaj numer dnia tygodnia (niedziela = 0): 5
Nazwa piątek oznacza piąty dzień po niedzieli.
```

Następne uruchomienie programu

```
Podanie nazwy, od której pochodzi nazwa dnia tygodnia!
Podaj numer dnia tygodnia (niedziela = 0): 9
Błędne dane wejściowe, spróbuj raz jeszcze
```

Analiza ▼

W powyższym programie konstrukcja `if-else-if` użyta w wierszach od 22. do 37. sprawdza dane wejściowe podane przez użytkownika i na ich podstawie generuje odpowiednie dane wyjściowe. Otrzymane po drugim uruchomieniu dane wyjściowe potwierdzają, że program wykrywa sytuację, gdy użytkownik poda liczbę spoza zakresu od 0 do 6, która nie odpowiada żadnemu dniowi tygodnia. Zaletą zastosowanej konstrukcji jest doskonałe dopasowanie do sprawdzanych warunków, które się wzajemnie wykluczają. Oznacza to, że poniedziałek nigdy nie będzie wtorkiem, a nieprawidłowe dane wejściowe nie mogą wskazać żadnego dnia tygodnia. Warto zwrócić uwagę na jeszcze jeden interesujący fakt: w programie wykorzystano stałą typu wyliczeniowego o nazwie `DaysOfWeek` zadeklarowaną w wierszu 5. i używaną w poleceniach `if`. Dane wejściowe użytkownika można po prostu porównywać do wartości, takich jak 0 (dla niedzieli). Jednak użycie stałej typu wyliczeniowego `niedziela` powoduje, że kod jest znacznie czytelniejszy.

Przetwarzanie warunkowe za pomocą switch-case

Celem konstrukcji `switch-case` jest umożliwienie sprawdzenia określonego wyrażenia względem wielu możliwych stałych oraz prawdopodobnie wykonanie innej operacji, w zależności od dopasowania do różnych wartości. Nowe słowa kluczowe C++, które często będziesz spotykał w tego typu konstrukcjach, to `switch`, `case`, `default` i `break`.

Poniżej przedstawiono składnię konstrukcji `switch-case`:

```
switch(expression)
{
case LabelA:
   WykonajOperację;
   break;

case LabelB:
   WykonajInnąOperację;
   break;
```

```
// itd.
default:
   ZadanieJeśliWyrażenieNieBędzieWcześniejObsłużone;
   break;
}
```

Przedstawiony powyżej kod oblicza wartość wyrażenia, a następnie porównuje ją z każdą etykietą `case`, szukając dopasowania. Wspomniane etykiety `case` muszą być stałymi. W przypadku znalezienia dopasowania kod znajdujący się w danej etykiecie `case` jest wykonywany. Jeżeli `wyrażenie` nie zostanie dopasowane do `LabelA`, kod sprawdza je względem `LabelB`. Gdy wartością operacji sprawdzenia będzie `true`, nastąpi uruchomienie `WykonajInnąOperację`. Sprawdzenie jest kontynuowane aż do napotkania polecenia `break`. Pierwsze napotkane polecenie `break` powoduje opuszczenie bloku kodu, użycie polecenia `break` nie jest obowiązkowe. Jednak jego brak spowoduje, że kod będzie porównywał wartość wyrażenia względem kolejnych etykiet, a takiej sytuacji chcemy uniknąć w konstrukcji `switch-case`. Etykieta `default` jest opcjonalna i zawiera polecenia, które będą wykonane, jeśli nie zostanie dopasowana żadna etykieta konstrukcji `switch-case`.

Wskazówka
Wskazówka

> Konstrukcja `switch-case` jest doskonale przystosowana do pracy ze stałymi w postaci typu wyliczeniowego. Słowo kluczowe `enum` omówiono w lekcji 3., zatytułowanej „Zmienne i stałe”.

W listingu 6.5 przedstawiono program będący funkcjonalnym odpowiednikiem programu z listingu 6.4. W programie użyto stałych typu wyliczeniowego; po uruchomieniu program podaje nazwę, od której pochodzi nazwa dnia tygodnia wskazanego przez użytkownika.

Listing 6.5. Program podający nazwę, od której pochodzi nazwa dnia tygodnia; użyto w nim konstrukcji switch-case, break i default

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    enum DaysOfWeek
6:    {
7:       niedziela = 0,
8:       poniedziałek,
9:       wtorek,
```

```
10:       środa,
11:       czwartek,
12:       piątek,
13:       sobota
14:    };
15:
16:    cout << "Podanie nazwy, od której pochodzi nazwa dnia tygodnia!" <<
       ↪endl;
17:    cout << "Podaj numer dnia tygodnia (niedziela = 0): ";
18:
19:    int Day = niedziela;    // Inicjalizacja dnia z wartością niedziela.
20:    cin >> Day;
21:
22:    switch(Day)
23:    {
24:    case niedziela:
25:       cout << "Nazwa niedziela oznacza dzień wolny (nie działać)" << endl;
26:       break;
27:
28:    case poniedziałek:
29:       cout << "Nazwa poniedziałek oznacza dzień po niedzieli" << endl;
30:       break;
31:
32:    case wtorek:
33:       cout << "Nazwa wtorek oznacza dzień wtóry, czyli drugi
          ↪po niedzieli" << endl;
34:       break;
35:
36:    case środa:
37:       cout << "Nazwa środa oznacza środkowy dzień tygodnia" << endl;
38:       break;
39:
40:    case czwartek:
41:       cout << "Nazwa czwartek oznacza czwarty dzień po niedzieli" << endl;
42:       break;
43:
44:    case piątek:
45:       cout << "Nazwa piątek oznacza piąty dzień po niedzieli" << endl;
46:       break;
47:
48:    case sobota:
49:       cout << "Nazwa sobota pochodzi od szabatu" << endl;
50:       break;
51:
52:    default:
53:       cout << "Błędne dane wejściowe, spróbuj raz jeszcze" << endl;
54:       break;
55:    }
```

```
56:
57:    return 0;
58: }
```

Wynik ▼

```
Podanie nazwy, od której pochodzi nazwa dnia tygodnia!
Podaj numer dnia tygodnia (niedziela = 0): 5
Nazwa piątek oznacza piąty dzień po niedzieli
```

Następne uruchomienie programu

```
Podanie nazwy, od której pochodzi nazwa dnia tygodnia!
Podaj numer dnia tygodnia (niedziela = 0): 9
Błędne dane wejściowe, spróbuj raz jeszcze
```

Analiza ▼

W wierszach od 22. do 55. znajduje się konstrukcja `switch-case`, która generuje różne dane wyjściowe, w zależności od wartości zmiennej `Day` w postaci podanej przez użytkownika liczby całkowitej. Jeżeli użytkownik wprowadzi liczbę 5, aplikacja sprawdzi wyrażenie `Day` konstrukcji `switch` (będzie miało wartość 5) względem etykiet będących stałymi typu wyliczeniowego. Program pominie pierwsze pięć z nich (od `niedzieli` do `czwartku`), ponieważ ich wartości wynoszą od 0 do 4, a więc żadna z nich nie jest równa 5. Po dotarciu do etykiety `piątek` wartość wyrażenia jest równa wartości stałej, co powoduje wykonanie kodu znajdującego się w tej etykiecie. Po dotarciu do polecenia `break` następuje opuszczenie konstrukcji `switch`. W trakcie drugiego uruchomienia programu po podaniu nieprawidłowych danych wejściowych następuje wykonanie kodu znajdującego się w etykiecie `default` i wyświetlenie użytkownikowi odpowiedniego komunikatu.

Omówiony program, w którym wykorzystano konstrukcję `switch-case`, generuje dokładnie takie same dane wyjściowe jak program przedstawiony w listingu 6.4 i używający konstrukcji `if-else-if`. Wersja oparta na konstrukcji `switch-case` wydaje się nieco bardziej strukturalna i lepiej dostosowana do sytuacji, w której wykonywana operacja jest bardziej skomplikowana niż proste wyświetlenie komunikatu na ekranie. W takich przypadkach nie zapomnij o ujęciu poleceń w nawiasy klamrowe, utworzysz w ten sposób bloki.

Wykonywanie warunkowe przy użyciu operatora ?:

W języku C++ został zaimplementowany interesujący i oferujący potężne możliwości operator warunkowy, który jest podobny do zwięzłej konstrukcji `if-else`.

Opisywany operator warunkowy jest nazywany operatorem trójargumentowym, ponieważ pobiera trzy operandy:

```
(wyrażenie warunkowe jako wartość typu bool) ? wyrażenie1 jeśli true :
↪wyrażenie2 jeśli false
```

Tego rodzaju operator może być stosowany w celu zwięzłego określenia większej liczby z dwóch podanych, jak ma to miejsce w poniższym poleceniu:

```
int Max = (Num1 > Num2)? Num1 : Num2; // Zmienna Max zawiera większą wartość
                                      // spośród Num1 i Num2.
```

W listingu 6.6 przedstawiono przetwarzanie warunkowe w programie przy użyciu operatora `?:`.

Listing 6.6. Użycie operatora warunkowego ?: do odszukania większej liczby spośród dwóch podanych

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Podaj dwie liczby" << endl;
 6:    int Num1 = 0, Num2 = 0;
 7:    cin >> Num1;
 8:    cin >> Num2;
 9:
10:    int Max = (Num1 > Num2)? Num1 : Num2;
11:    cout << "Spośród " << Num1 << " i " \
12:        << Num2 << " większą liczbą jest: " << Max << endl;
13:
14:    return 0;
15: }
```

Wynik ▼

```
Podaj dwie liczby
365
-1
Spośród 365 i -1 większą liczbą jest: 365
```

Analiza ▼

Interesujący jest wiersz 10. kodu, ponieważ zawiera zwięzłe polecenie pozwalające na określenie, która z dwóch podanych liczb jest większa. To zwięzłe polecenie to inny sposób zapisania poniższego kodu opartego na konstrukcji `if-else`:

```
int Max = 0;
if (Num1 > Num2)
   Max = Num1;
else
   Max = Num2;
```

Za pomocą operatora warunkowego kod udało się skrócić o kilka wierszy. Jednak zmniejszanie wierszy kodu nie powinno być priorytetem. Niektórzy programiści preferują stosowanie operatorów warunkowych, natomiast inni ich unikają. Bardzo ważne jest używanie operatorów warunkowych w sposób, który pozostanie łatwy do zrozumienia.

TAK	NIE
W wyrażeniach `switch` używaj stałych i typów wyliczeniowych, aby zapewnić większą czytelność kodu. Pamiętaj o obsłużeniu etykiety default, o ile nie będzie zupełnie niepotrzebna. Upewnij się, że w poszczególnych etykietach przez przypadek nie zapomniałeś o poleceniu `break`.	Dwóm etykietom nie nadawaj tej samej nazwy, to nie ma żadnego sensu i uniemożliwia kompilację kodu. Nie komplikuj kodu konstrukcji switch przez umieszczanie etykiet pozbawionych polecenia break i polegających na sekwencji. Takie podejście może doprowadzić do nieprawidłowego działania programu, jeśli później przeniesiesz etykietę i nie zwrócisz uwagi na polecenia `break`. Nie używaj skomplikowanych warunków lub wyrażeń w operatorach warunkowych `?:`.

Wykonywanie kodu w pętlach

Dowiedziałeś się już, w jaki sposób utworzyć program, aby zachowywał się odmiennie, w zależności od wartości zmiennych. Przykładowo program w listingu 6.2 przeprowadzał operację mnożenia po naciśnięciu klawisza *m*, naciśnięcie innego dowolnego klawisza powodowało wykonanie dodawania. Co można zrobić, jeśli użytkownik nie chce zakończyć działania programu? Co można zrobić, jeśli użytkownik chce przeprowadzić inną operację mnożenia,

dzielenia lub nawet wiele tych operacji? W takim przypadku należy ponownie wykonać już istniejący kod.

W tym celu konieczne jest użycie pętli w programie.

Bardzo prosta pętla wykonywana przy użyciu polecenia goto

Jak sama nazwa wskazuje, polecenie `goto` powoduje przejście do wskazanego miejsca w programie i kontynuowanie stamtąd jego działania. Polecenie to można wykorzystać do cofnięcia się w programie i ponownego wykonania pewnych poleceń.

Składnia polecenia `goto` jest następująca:

```
PewnaFunkcja()
{
PrzejścieDoPunktu: // To jest etykieta.
   PowtarzającySięKod;

   goto PrzejścieDoPunktu;
}
```

W powyższym fragmencie kodu zdefiniowano punkt o nazwie `PrzejścieDoPunktu`, a następnie polecenie `goto` zostało użyte do powtórzenia wykonywania programu od wskazanego punktu (przykład użycia tego rodzaju konstrukcji przedstawiono w listingu 6.7). Jeśli polecenie `goto` nie zostanie wywołane z warunkiem przyjmującym wartość `false` w pewnych sytuacjach lub ponownie wykonywany kod nie będzie zawierał polecenia `return` wykonywanego w określonych sytuacjach, to kod pomiędzy poleceniem `goto` i etykietą będzie wykonywany w nieskończoność i uniemożliwi zakończenie działania programu.

Listing 6.7. Zapytanie użytkownika, czy chce powtórzyć obliczenia

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5: JumpToPoint:
 6:    int Num1 = 0, Num2 = 0;
 7:
 8:    cout << "Podaj dwie liczby całkowite: " << endl;
 9:    cin >> Num1;
```

```
10:    cin >> Num2;
11:
12:    cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
13:    cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
14:
15:    cout << "Czy chcesz raz jeszcze wykonać operację (t/n)?" << endl;
16:    char Repeat = 't';
17:    cin >> Repeat;
18:
19:    if (Repeat == 'y')
20:        goto JumpToPoint;
21:
22:    cout << "Do widzenia!" << endl;
23:
24:    return 0;
25: }
```

Wynik ▼

```
Podaj dwie liczby całkowite:
56
25
56 x 25 = 1400
56 + 25 = 81
Czy chcesz raz jeszcze wykonać operację (t/n)?
t
Podaj dwie liczby całkowite:
95
-47
95 x -47 = -4465
95 + -47 = 48
Czy chcesz raz jeszcze wykonać operację (t/n)?
n
Do widzenia!
```

Analiza ▼

Zwróć uwagę na podstawową różnicę pomiędzy listingami 6.7 i 6.1. Program przedstawiony w listingu 6.1 musi być uruchomiony dwukrotnie (dwa oddzielne wykonania), aby pozwolić użytkownikowi na podanie nowego zestawu liczb oraz poznanie wyniku ich dodawania i mnożenia. Z kolei program w listingu 6.7 podaje wspomniane dane w pojedynczym cyklu wykonania, ponieważ pyta użytkownika, czy chce przeprowadzić kolejną operację. Polecenie pozwalające na ponowne przeprowadzenie operacji znajduje się w wierszu 20., w którym następuje wywołanie `goto`, jeśli użytkownik nacisnął klawisz *t*. Wykonanie

polecenia goto w wierszu 20. skutkuje powrotem do etykiety JumpToPoint zadeklarowanej w wierszu 5., co praktycznie powoduje ponowne uruchomienie programu.

Ostrzeżenie

Polecenie goto nie jest zalecaną formą pętli programistycznej, ponieważ zbyt częste używanie goto może prowadzić do nieprzewidywalnego przepływu sterowaniem programu, gdzie nastąpi przeskakiwanie pomiędzy wierszami kodu bez zachowania określonej kolejności. W pewnych przypadkach może również dojść do pozostawienia zmiennych w nieprzewidywalnym stanie.

Błędne użycie polecenia goto może prowadzić do powstania tzw. *kodu spaghetti*. Możesz unikać poleceń goto, używając pętli while, do...while i for, które zostaną omówione dalej w tej lekcji.

Jedyny powód, dla którego przedstawiono polecenie goto w tej książce, to umożliwienie zrozumienia kodu wykorzystującego wymienione polecenie.

Pętla while

Słowo kluczowe while w C++ pozwala na wykonanie w bardziej wyrafinowany sposób zadania, które w listingu 6.7 było realizowane przez polecenie goto. Składnia pętli while jest następująca:

```
while(wyrażenie)
{
   // Warunek przyjmuje wartość true.
   PolecenieBloku;
}
```

Polecenie bloku jest wykonywane, dopóki wartością wyrażenia będzie true. Dlatego ważne jest utworzenie kodu w taki sposób, aby wartością wyrażenia także było false, w przeciwnym razie pętla while będzie wykonywana w nieskończoność.

Program przedstawiony w listingu 6.8 odpowiada programowi z listingu 6.7, ale zamiast polecenia goto do umożliwienia użytkownikowi powtórzenia obliczeń została wykorzystana pętla while.

Listing 6.8. Użycie pętli while, aby pomóc użytkownikowi ponownie przeprowadzić obliczenia

```
 0: #include <iostream>
 1: using namespace std;
 2:
```

```
 3: int main()
 4: {
 5:    char UserSelection = 'm';    // Wartość początkowa.
 6:
 7:    while (UserSelection != 'x')
 8:    {
 9:       cout << "Podaj dwie liczby całkowite: " << endl;
10:       int Num1 = 0, Num2 = 0;
11:       cin >> Num1;
12:       cin >> Num2;
13:
14:       cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
15:       cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
16:
17:       cout << "Naciśnij x, aby zakończyć, inny klawisz ponawia
          ↪operację" << endl;
18:       cin >> UserSelection;
19:    }
20:
21:    cout << "Do widzenia!" << endl;
22:
23:    return 0;
24: }
```

Wynik ▼

```
Podaj dwie liczby całkowite:
56
25
56 x 25 = 1400
56 + 25 = 81
Naciśnij x, aby zakończyć, inny klawisz ponawia operację
r
Podaj dwie liczby całkowite:
365
-5
365 x -5 = -1825
365 + -5 = 360
Naciśnij x, aby zakończyć, inny klawisz ponawia operację
x
Do widzenia!
```

Analiza ▼

Pętla `while` w wierszach od 7. do 19. zawiera większą część logiki biznesowej aplikacji. Zwróć uwagę, jak pętla `while` sprawdza wyrażenie (`UserSelection != 'x'`) i kontynuuje działanie tylko wtedy, kiedy wartością wyrażenia będzie

true. Aby umożliwić pierwszą iterację pętli, zmienna UserSelection została w wierszu 5. zainicjalizowana wraz z wartością m (można użyć innej dowolnej wartości poza x). W przeciwnym razie w trakcie pierwszej iteracji pętli wyrażenie przyjmie wartość false i aplikacja zakończy działanie, nie pozwalając użytkownikowi na wykonanie jakiejkolwiek konstruktywnej operacji. Pierwsza iteracja pętli jest bardzo prosta: w wierszu 17. użytkownik jest pytany, czy chce przeprowadzić inne obliczenia. Wiersz 18. zawierający odpowiedź użytkownika to ten, w którym następuje modyfikacja wyrażenia dająca programowi możliwość kontynuacji lub zakończenia działania. Po przeprowadzeniu pierwszej iteracji pętli wykonywanie programu powraca do obliczenia wartości wyrażenia pętli while (wiersz 7.) i powtórzenia operacji, jeśli użytkownik nie nacisnął klawisza *x*. Jeżeli użytkownik naciśnie klawisz *x*, w trakcie kolejnego obliczenia wyrażenia w wierszu 7. przyjmie ono wartość false, nastąpi opuszczenie pętli while i zakończenie działania programu po wyświetleniu komunikatu pożegnalnego.

Uwaga

Pętla jest nazywana również iteracją. Polecenia wykorzystujące while, do...while i for są nazywane także poleceniami iteracyjnymi.

Pętla do...while

Zdarzają się sytuacje (np. takie jak przedstawiona w listingu 6.8), gdy trzeba mieć pewność, że fragment kodu zostanie powtórzony w pętli i będzie wykonany przynajmniej jeden raz. W takich przypadkach użyteczna jest pętla do...while.

Składnia pętli do...while przedstawia się następująco:

```
do
{
   PolecenieBloku; // Wykonane przynajmniej jeden raz.
} while(warunek); // Wartość false powoduje zakończenie działania pętli.
```

Zauważ, że wiersz zawierający while(warunek) jest zakończony średnikiem. To jest odmienne od sytuacji, gdzie w poprzedniej pętli while średnik powodował zakończenie pętli w miejscu jej zdefiniowania i skutkował powstaniem pustego polecenia.

W listingu 6.9 pokazano użycie pętli do...while w celu zapewnienia przynajmniej jednokrotnego wykonania określonych poleceń.

Listing 6.9. Użycie pętli do…while w celu ponownego wykonania bloku kodu

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    char UserSelection = 'x';    // Wartość początkowa.
 6:    do
 7:    {
 8:       cout << "Podaj dwie liczby całkowite: " << endl;
 9:       int Num1 = 0, Num2 = 0;
10:       cin >> Num1;
11:       cin >> Num2;
12:
13:       cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
14:       cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
15:
16:       cout << "Naciśnij x, aby zakończyć, inny klawisz ponawia
          ↪operację" << endl;
17:       cin >> UserSelection;
18:    } while (UserSelection != 'x');
19:
20:    cout << "Do widzenia!" << endl;
21:
22:    return 0;
23: }
```

Wynik ▼

```
Podaj dwie liczby całkowite:
654
-25
654 x -25 = -16350
654 + -25 = 629
Naciśnij x, aby zakończyć, inny klawisz ponawia operację
m
Podaj dwie liczby całkowite:
909
101
909 x 101 = 91809
909 + 101 = 1010
Naciśnij x, aby zakończyć, inny klawisz ponawia operację
x
Do widzenia!
```

Analiza ▼

Zachowanie programu i wygenerowane dane wyjściowe są bardzo podobne do poprzedniego. Praktycznie jedyna różnica polega na użyciu słowa kluczowego `do` w wierszu 6. oraz użyciu `while` później w wierszu 18. Kod jest wykonywany szeregowo, wiersz po wierszu, aż do osiągnięcia `while` w wierszu 18. W tym wierszu następuje obliczenie wartości wyrażenia `(UserSelection != 'x')`. Jeżeli przyjmie ono wartość `true` (tzn. użytkownik nie nacisnął klawisza *x* w celu zakończenia programu), następuje ponowne wykonanie pętli. Jeśli natomiast wartością wyrażenia będzie `false` (użytkownik nacisnął klawisz *x*), nastąpi opuszczenie pętli, wyświetlenie komunikatu pożegnalnego i zakończenie działania programu.

Pętla for

Polecenie `for` to bardziej złożona pętla, która pozwala na jednokrotne wykonanie polecenia inicjalizującego (najczęściej w celu inicjalizacji licznika), sprawdzenie warunku opuszczenia pętli (zwykle z użyciem wspomnianego licznika) i wykonanie operacji na końcu każdej iteracji pętli (najczęściej inkrementacji lub modyfikacji licznika).

Składnia pętli `for` przedstawia się następująco:

```
for (początkowe wyrażenie wykonane tylko jednokrotnie;
   warunek zakończenia pętli sprawdzany na początku każdej iteracji;
   wyrażenie wykonywane na końcu każdej iteracji)
{
   WykonajPewneOperacje;
}
```

Pętla `for` to konstrukcja pozwalająca programiście na zdefiniowanie zmiennej licznika wraz z wartością początkową, sprawdzenie na początku każdej iteracji wartości licznika dotyczącej warunku opuszczenia pętli oraz zmianę wartości zmiennej na końcu każdej iteracji pętli.

W listingu 6.10 pokazano efektywny sposób uzyskania dostępu do elementów tablicy za pomocą pętli `for`.

Listing 6.10. Użycie pętli for w celu umieszczenia elementów w tablicy, a następnie wyświetlenia jej zawartości

```
 0: #include <iostream>
 1: using namespace std;
```

```
 2:
 3: int main()
 4: {
 5:    const int ARRAY_LENGTH = 5;
 6:    int MyInts[ARRAY_LENGTH] = {0};
 7:
 8:    cout << "Wypełnienie tablicy " << ARRAY_LENGTH << " liczb
       ↪całkowitych" << endl;
 9:
10:    for (int ArrayIndex = 0; ArrayIndex < ARRAY_LENGTH; ++ArrayIndex)
11:    {
12:       cout << "Podaj liczbę całkowitą dla elementu " << ArrayIndex << ": ";
13:       cin >> MyInts[ArrayIndex];
14:    }
15:
16:    cout << "Wyświetlenie zawartości tablicy: " << endl;
17:
18:    for (int ArrayIndex = 0; ArrayIndex < ARRAY_LENGTH; ++ArrayIndex)
19:       cout << "Element " << ArrayIndex << " = " << MyInts[ArrayIndex] <<
          ↪endl;
20:
21:    return 0;
22: }
```

Wynik ▼

```
Wypełnienie tablicy 5 liczb całkowitych
Podaj liczbę całkowitą dla elementu 0: 365
Podaj liczbę całkowitą dla elementu 1: 31
Podaj liczbę całkowitą dla elementu 2: 24
Podaj liczbę całkowitą dla elementu 3: -59
Podaj liczbę całkowitą dla elementu 4: 65536
Wyświetlenie zawartości tablicy:
Element 0: 365
Element 1: 31
Element 2: 24
Element 3: -59
Element 4: 65536
```

Analiza ▼

W listingu 6.10 znajdują się dwie pętle `for` — w wierszach 10. i 18. Pierwsza pomaga w dostaniu się do elementów tablicy, natomiast druga w ich wyświetleniu. Składnia obu wspomnianych pętli jest identyczna. W obu pętlach została zdefiniowana zmienna indeksu `ArrayIndex` umożliwiająca uzyskanie dostępu do elementów tablicy. Inkrementacja wymienionej zmiennej następuje na

końcu każdej iteracji pętli, co umożliwia w kolejnej iteracji uzyskanie dostępu do następnego elementu tablicy. Środkowe wyrażenie w pętli `for` to warunek opuszczenia pętli. Na końcu każdej iteracji wyrażenie sprawdza, czy wartość `ArrayIndex` po inkrementacji nadal mieści się w granicach tablicy. Sprawdzenie odbywa się przez porównanie wartości `ArrayIndex` i `ARRAY_LENGTH`. W ten sposób mamy również pewność, że pętla `for` nigdy nie wykroczy poza granice tablicy.

Uwaga

Zmienna, taka jak `ArrayIndex` w listingu 6.10, pomagająca w uzyskaniu dostępu do elementów kolekcji, np. tablicy, nosi nazwę *iteratora*.

Zakres iteratora zadeklarowany dla konstrukcji `for` jest ograniczony jedynie do pętli `for`. Dlatego też w drugiej pętli `for` w listingu 6.10 zmienna ta została ponownie zadeklarowana, w efekcie jest nową zmienną.

Jednak wykorzystanie inicjalizacji, wyrażenia warunkowego i wyrażenia, którego wartość jest obliczana na końcu każdej iteracji, pozostaje opcjonalne. Istnieje więc możliwość utworzenia pętli `for` pozbawionej wymienionych komponentów, co zostało przedstawione w listingu 6.11.

Listing 6.11. Użycie pętli for, pominięcie wyrażenia pętli i powtórzenie obliczeń na żądanie użytkownika

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     // Bez wyrażenia pętli (brakuje trzeciego wyrażenia).
 6:     for(char UserSelection = 'm'; (UserSelection != 'x'); )
 7:    {
 8:        cout << "Podaj dwie liczby całkowite: " << endl;
 9:        int Num1 = 0, Num2 = 0;
10:        cin >> Num1;
11:        cin >> Num2;
12:
13:        cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
14:        cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
15:
16:        cout << "Naciśnij x, aby zakończyć, inny klawisz ponawia
           ↪operację" << endl;
17:        cin >> UserSelection;
18:     }
19:
20:     cout << "Do widzenia!" << endl;
21:
```

```
22:    return 0;
23: }
```

Wynik ▼

```
Podaj dwie liczby całkowite:
56
25
56 x 25 = 1400
56 + 25 = 81
Naciśnij x, aby zakończyć, inny klawisz ponawia operację
m
Podaj dwie liczby całkowite:
789
-36
789 x -36 = -28404
789 + -36 = 753
Naciśnij x, aby zakończyć, inny klawisz ponawia operację
x
Do widzenia!
```

Analiza ▼

Kod programu jest identyczny z przedstawionym w listingu 6.8 opartym na pętli `while`, a jedyna różnica polega na użyciu pętli `for` w wierszu 8. Interesujące jest, że pętla `for` zawiera jedynie wyrażenia inicjalizacyjne i warunkowe, a ignoruje zupełnie możliwość zmiany zmiennej na końcu każdej iteracji.

Uwaga

Istnieje możliwość inicjalizacji wielu zmiennych w pętli `for` w ramach pierwszego wyrażenia inicjalizującego, które jest wykonywane jednorazowo. Pętla `for` z listingu 6.11 z wieloma inicjalizowanymi zmiennymi mogłaby przedstawiać się następująco:

```
for (int Index = 0, AnotherInt = 5; Index < ARRAY_LENGTH;
↪++Index, --AnotherInt)
```

Zwróć uwagę na dodanie nowej zmiennej o nazwie `AnotherInt` zainicjalizowanej z wartością 5.

Interesujące jest, że można również dekrementować zmienną w wyrażeniu pętli, jednokrotnie w trakcie każdej iteracji.

Zmiana zachowania pętli za pomocą poleceń continue i break

Istnieje kilka przypadków — zwłaszcza w skomplikowanych pętlach obsługujących wiele parametrów z mnóstwem warunków — kiedy nie ma możliwości efektywnego utworzenia warunku pętli i trzeba zmodyfikować działania programu nawet w ramach pętli. W takich sytuacjach przydatne jest użycie poleceń `continue` i `break`.

Polecenie `continue` pozwala na wznowienie działania pętli od początku. Działanie polecenia polega po prostu na pominięciu kodu znajdującego się po poleceniu. Dlatego też efektem wywołania `continue` w pętlach `while`, `do...while` i `for` jest ponowne obliczenie warunku pętli; jeśli przyjmie on wartość `true`, wtedy nastąpi ponowne wykonanie kodu w bloku.

Uwaga

W przypadku użycia polecenia `continue` w pętli `for` wyrażenie pętli (trzecie w konstrukcji `for`, zwykle stosowane do inkrementacji licznika) jest obliczane jeszcze przed sprawdzeniem warunku pętli.

Polecenie `break` powoduje opuszczenie bloku pętli, czyli — praktycznie rzecz biorąc — jego wywołanie kończy daną pętlę.

Ostrzeżenie

Zwykle programiści oczekują, że spełnienie warunku pętli spowoduje wykonanie całego kodu zdefiniowanego w pętli. Polecenia `continue` i `break` zmieniają to zachowanie, ich stosowanie może doprowadzić do powstania nieintuicyjnego kodu.

Dlatego też poleceń `continue` i `break` powinieneś używać rzadko i jedynie wtedy, kiedy niemożliwe jest utworzenie prawidłowej i efektywnej pętli bez wymienionych poleceń.

W znajdującym się w dalszej części rozdziału listingu 6.12 przedstawiono użycie polecenia `continue` w celu poproszenia użytkownika o ponowne podanie liczb przed rozpoczęciem obliczeń z ich użyciem. Natomiast polecenie `break` wykorzystano do zakończenia działania pętli.

Pętle działające w nieskończoność

Pamiętaj, że pętle `while`, `do...while` i `for` mają wyrażenie warunkowe, którego wartość `false` powoduje zakończenie działania pętli. Jeżeli zdefiniujesz warunek,

który zawsze będzie przyjmował wartość `true`, działanie pętli nigdy nie zostanie przerwane.

Poniżej przedstawiono przykład działającej w nieskończoność pętli `while`:

```
while(true)    // Wyrażenie while ma wartość true.
{
   WykonajWielokrotnieOperację;
}
```

Z kolei poniżej pokazano przykład działającej w nieskończoność pętli `do...while`:

```
do
{
   WykonajWielokrotnieOperację;
} while(true);    // Wyrażenie do ... while nigdy nie będzie miało wartości false.
```

Tu natomiast przedstawiono przykład działającej w nieskończoność pętli `for`:

```
for (;;)    // Brak podanego warunku = pętla for działa w nieskończoność.
{
   WykonajWielokrotnieOperację;
}
```

Być może to wydaje się dziwne, ale pętle działające w nieskończoność mają swoje przeznaczenie. Wyobraź sobie system operacyjny okresowo sprawdzający, czy do portu USB zostało podłączone urządzanie, np. pamięci masowej. Wspomniana aktywność powinna trwać tak długo, jak długo działa system operacyjny. W takich przypadkach należy użyć pętli działającej w nieskończoność.

Kontrolowanie pętli działającej w nieskończoność

Jeżeli chcesz zakończyć pracę pętli działającej w nieskończoność (powiedźmy, że wspomniany w poprzednim przykładzie system operacyjny ma zostać zamknięty), konieczne jest użycie polecenia `break` (najczęściej w bloku `if(warunek)`).

Poniżej przedstawiono sposób użycia polecenia `break` do opuszczenia pętli działającej w nieskończoność:

```
while(true)    // Wyrażenie while ma wartość true.
{
   WykonajWielokrotnieOperację;
   if(wyrażenie)
```

```
      break;  // Zakończenie działania pętli, gdy wartością wyrażenia będzie true.
}
```

Z kolei poniżej przedstawiono użycie polecenia `break` w działającej w nieskończoność pętli `do...while`:

```
do
{
   WykonajWielokrotnieOperację;
   if(wyrażenie)
      break;  // Zakończenie działania pętli, gdy wartością wyrażenia będzie true.
} while(true);    // Wyrażenie do ...while nigdy nie będzie miało wartości false.
```

Natomiast poniżej przedstawiono użycie polecenia `break` w działającej w nieskończoność pętli `for`:

```
for (;;)    // Brak podanego warunku = pętla for działa w nieskończoność.
{
   WykonajWielokrotnieOperację;
   if(wyrażenie)
      break;  // Zakończenie działania pętli, gdy wartością wyrażenia będzie true.
}
```

W listingu 6.12 pokazano użycie pętli działającej w nieskończoność wraz z poleceniami `continue` i `break` kontrolującymi kryteria opuszczenia pętli.

Listing 6.12. Użycie polecenia continue w celu ponownego uruchomienia pętli i polecenia break do zakończenia pętli for działającej w nieskończoność

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    for(;;)    // Pętla działająca w nieskończoność.
 6:    {
 7:       cout << "Podaj dwie liczby całkowite: " << endl;
 8:       int Num1 = 0, Num2 = 0;
 9:       cin >> Num1;
10:       cin >> Num2;
11:
12:       cout << "Czy chcesz poprawić podane liczby? (t/n): ";
13:       char ChangeNumbers = '\0';
14:       cin >> ChangeNumbers;
15:
16:       if (ChangeNumbers == 't')
17:          continue;  // Ponowne uruchomienie pętli!
18:
```

```
19:       cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
20:       cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
21:
22:       cout << "Naciśnij x, aby zakończyć, inny klawisz ponawia
          ↪operację" << endl;
23:       char UserSelection = '\0';
24:       cin >> UserSelection;
25:
26:       if (UserSelection == 'x')
27:          break;    // Zakończenie pętli działającej w nieskończoność.
28:    }
29:
30:    cout << "Goodbye!" << endl;
31:
32:    return 0;
33: }
```

Wynik ▼

```
Podaj dwie liczby całkowite:
560
25
Czy chcesz poprawić podane liczby? (t/n): t
Podaj dwie liczby całkowite:
56
25
Czy chcesz poprawić podane liczby? (t/n): n
56 x 25 = 1400
56 + 25 = 81
Naciśnij x, aby zakończyć, inny klawisz ponawia operację
r
Podaj dwie liczby całkowite:
95
-1
Czy chcesz poprawić podane liczby? (t/n): n
95 x -1 = -95
95 + -1 = 94
Naciśnij x, aby zakończyć, inny klawisz ponawia operację
x
Do widzenia!
```

Analiza ▼

Pętla `for` w wierszu 5. różni się od pętli `for` przedstawionej w listingu 6.11, ponieważ działa w nieskończoność i jest pozbawiona warunku wyrażenia obliczanego w trakcie każdej iteracji. Innymi słowy, bez wywołania polecenia

break ta pętla (a tym samym aplikacja) nigdy nie zakończy działania. Zwróć uwagę na dane wyjściowe odróżniające się od przedstawianych dotychczas — użytkownik ma możliwość poprawienia podanych liczb całkowitych, zanim program przystąpi do mnożenia i dodawania tych liczb. Tego rodzaju logika została zaimplementowana za pomocą polecenia continue (patrz wiersz 17.) wydawanego po spełnieniu określonego warunku (patrz wiersz 16.). Kiedy użytkownik naciśnie klawisz *t* po pytaniu, czy chce poprawić podane liczby, wartością warunku w wierszu 16. będzie true i nastąpi wywołanie polecenia continue znajdującego się w kolejnym wierszu. Po wywołaniu continue działanie programu rozpoczyna się od początku pętli, a użytkownik otrzymuje na ekranie polecenia podania dwóch liczb całkowitych. Podobnie, na końcu pętli użytkownik jest pytany, czy chce zakończyć działanie programu. W wierszu 26. naciśnięty przez niego klawisz jest porównywany z *x*, jeśli użytkownik nacisnął *x*, następuje wywołanie polecenia break i opuszczenie pętli.

Uwaga

W kodzie przedstawionym w listingu 6.12 użyto pustego polecenia for(;;) w celu utworzenia pętli działającej w nieskończoność. Taką pętlę możesz zastąpić przez while(true) lub do...while(true) w celu wygenerowania tych samych danych wyjściowych, ale za pomocą innego typu pętli.

TAK	NIE
Używaj pętli do...while, gdy logika pętli powinna być wykonana przynajmniej jednokrotnie. Używaj pętli while, do...while lub for wraz z doskonale zdefiniowanymi wyrażeniami warunkowymi. Stosuj wcięcia w kodzie poleceń bloku pętli, aby zwiększyć czytelność kodu.	Nie używaj polecenia goto. Nie używaj zbyt często poleceń continue i break. Nie twórz pętli działających w nieskończoność wraz z poleceniami break, o ile nie jest to absolutnie konieczne.

Programowanie zagnieżdżonych pętli

Na początku lekcji zobaczyłeś przykład zagnieżdżonych poleceń, czasami występuje konieczność zagnieżdżenia jednej pętli w innej. Wyobraź sobie dwie tablice liczb całkowitych. Jeżeli w tablicy 2. chcesz znaleźć wielokrotność każdej liczby tabeli 1., wtedy konieczne jest użycie zagnieżdżonych pętli. Pierwsza pętla przeprowadza iterację przez Array1, natomiast druga iterację Array2.

W listingu 6.13 pokazano przykład użycia pętli zagnieżdżonych w pętli dowolnego rodzaju.

Listing 6.13. Użycie zagnieżdżonych pętli w celu pomnożenia wszystkich elementów tablicy przez pozostałe

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int ARRAY1_LEN = 3;
 6:    const int ARRAY2_LEN = 2;
 7:
 8:    int MyInts1[ARRAY1_LEN] = {35, -3, 0};
 9:    int MyInts2[ARRAY2_LEN] = {20, -1};
10:
11:    cout << "Mnożenie każdej liczby w MyInts1 przez każdą w MyInts2:" <<
       ↪endl;
12:
13:    for(int Array1Index = 0; Array1Index < ARRAY1_LEN; ++Array1Index)
14:       for(int Array2Index = 0; Array2Index < ARRAY2_LEN; ++Array2Index)
15:          cout << MyInts1[Array1Index] << " x " << MyInts2[Array2Index] \
16:          << " = " << MyInts1[Array1Index] * MyInts2[Array2Index] <<
             ↪endl;
17:
18:    return 0;
19: }
```

Wynik ▼

```
Mnożenie każdej liczby w MyInts1 przez każdą w MyInts2:
35 x 20 = 700
35 x -1 = -35
-3 x 20 = -60
-3 x -1 = 3
0 x 20 = 0
0 x -1 = 0
```

Analiza ▼

Dwie zagnieżdżone pętle `for` zostały zdefiniowane w wierszach 13. i 14. Pierwsza z nich przeprowadza iterację przez tablicę `MyInts1`, podczas gdy druga wykonuje iterację przez tablicę `MyInts2`. W trakcie każdej iteracji pierwszej pętli `for` następuje wykonanie drugiej pętli. Druga pętla `for`

przeprowadza iterację przez wszystkie elementy MyInts2, w każdej iteracji mnoży dany element przez element wskazany w zmiennej Array1Index, pochodzący z pierwszej pętli. Dla każdego elementu w tablicy MyInts1 druga pętla for wykonuje iterację przez wszystkie elementy tablicy MyInts2, co oznacza, że pierwszy element MyInts1 o indeksie 0 zostaje pomnożony przez wszystkie elementy tablicy MyInts2. Następnie drugi element tablicy MyInts1 zostaje pomnożony przez wszystkie elementu tablicy MyInts2. Na końcu trzeci element MyInts1 jest pomnożony przez wszystkie elementy MyInts2.

Uwaga

W celu wygody i zachowania koncentracji na pętlach zawartość tablic w listingu 6.13 została zainicjalizowana. Oczywiście, możesz wykorzystać kod z wcześniejszych przykładów, np. z listingu 6.10, i zapewnić użytkownikowi możliwość podania liczb całkowitych dla tablic.

Użycie zagnieżdżonych pętli do iteracji tablic wielowymiarowych

W lekcji 4. poznałeś tablice wielowymiarowe. W kodzie przedstawionym w listingu 4.3 pokazano, jak można uzyskać dostęp do elementów tablicy dwuwymiarowej składającej się z trzech wierszy i trzech kolumn. W wymienionym listingu polecenia zapewniały dostęp do poszczególnych elementów tablicy, jedno polecenie w oddzielnym wierszu dla każdego elementu. Kod nie zawierał żadnej automatyzacji, a zwiększenie wymiarów tablicy oznaczało konieczność dodania dużej ilości kodu niezbędnego do obsłużenia nowych elementów tablicy. Na szczęście, użycie pętli może to wszystko zmienić, a odpowiedni kod przestawiono w listingu 6.14.

Listing 6.14. Użycie zagnieżdżonych pętli do przeprowadzenia iteracji elementów w dwuwymiarowej tablicy liczb całkowitych

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int MAX_ROWS = 3;
 6:    const int MAX_COLS = 4;
 7:
 8:    // Dwuwymiarowa tablica liczb całkowitych.
 9:    int MyInts[MAX_ROWS][MAX_COLS] = { {34, -1, 879, 22},
10:                                       {24, 365, -101, -1},
```

```
11:                                          {-20, 40, 90, 97} };
12:
13:     // Iteracja rekordów.
14:     for (int Row = 0; Row < MAX_ROWS; ++Row)
15:     {
16:        // Iteracja liczb całkowitych w poszczególnych rekordach (kolumnach).
17:        for (int Column = 0; Column < MAX_COLS; ++Column)
18:        {
19:           cout << "Liczba[" << Row << "][" << Column \
20:                << "] = " << MyInts[Row][Column] << endl;
21:        }
22:     }
23:
24:     return 0;
25: }
```

Wynik ▼

```
Liczba[0][0] = 34
Liczba[0][1] = -1
Liczba[0][2] = 879
Liczba[0][3] = 22
Liczba[1][0] = 24
Liczba[1][1] = 365
Liczba[1][2] = -101
Liczba[1][3] = -1
Liczba[2][0] = -20
Liczba[2][1] = 40
Liczba[2][2] = 90
Liczba[2][3] = 97
```

Analiza ▼

W wierszach od 14. do 22. znajdują się dwie pętle `for`, które są potrzebne do uzyskania dostępu oraz przeprowadzenia iteracji przez dwuwymiarową tablicę liczb całkowitych. Wspomniana tablica dwuwymiarowa to efekt użycia tablicy tablic. Zwróć uwagę, że pierwsza pętla `for` pozwala na uzyskanie dostępu do wierszy (każdy z nich jest tablicą liczb całkowitych), natomiast druga do poszczególnych elementów danej tablicy, tzn. kolumn.

Uwaga
Uwaga

W kodzie przedstawionym w listingu 6.14 użyto nawiasów klamrowych dla zagnieżdżonej pętli `for`, choć jedynym powodem takiego rozwiązania jest zwiększenie czytelności kodu. Zagnieżdżona pętla będzie bez problemów działała bez tych nawiasów, ponieważ blok pętli to tylko pojedyncze polecenie do wykonania (a nie polecenie złożone wymagające ujęcia w nawiasy klamrowe).

Użycie zagnieżdżonych pętli do obliczenia liczb ciągu Fibonacciego

Sławny ciąg Fibonacciego to zbiór liczb zaczynający się od 0 i 1, w którym każda kolejna liczba jest sumą dwóch poprzednich. Dlatego też ciąg Fibonacciego rozpoczyna się od przedstawionej poniżej sekwencji:

0, 1, 1, 2, 3, 5, 8 ... itd.

W listingu 6.15 pokazano, jak można utworzyć ciąg Fibonacciego składający się z dowolnej liczby elementów (ograniczeniem jest jedynie fizyczna pojemność liczby całkowitej przechowującej ostatni element).

Listing 6.15. Użycie zagnieżdżonych pętli do obliczenia elementów ciągu Fibonacciego

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int NumsToCal = 5;
 6:    cout << "Ten program obliczy jednorazowo " << NumsToCal \
 7:         << " elementów ciągu Fibonacciego" << endl;
 8:
 9:    int Num1 = 0, Num2 = 1;
10:    char WantMore = '\0';
11:    cout << Num1 << " " << Num2 << " ";
12:
13:    do
14:    {
15:       for (int Index = 0; Index < NumsToCal; ++Index)
16:       {
17:          cout << Num1 + Num2 << " ";
18:
19:          int Num2Temp = Num2;
20:          Num2 = Num1 + Num2;
21:          Num1 = Num2Temp;
22:       }
23:
24:       cout << endl << "Czy chcesz więcej elementów (t/n)? ";
25:       cin >> WantMore;
26:    }while (WantMore == 't');
27:
28:    cout << "Do widzenia!" << endl;
29:
30:    return 0;
31: }
```

Wynik ▼

```
Ten program obliczy jednorazowo 5 elementów ciągu Fibonacciego
0 1 1 2 3 5 8
Czy chcesz więcej elementów (t/n)? t
13 21 34 55 89
Czy chcesz więcej elementów (t/n)? t
144 233 377 610 987
Czy chcesz więcej elementów (t/n)? t
1597 2584 4181 6765 10946
Czy chcesz więcej elementów (t/n)? n
Do widzenia!
```

Analiza ▼

Zewnętrzna pętla `do...while` w wierszu 13. to praktycznie pętla zapytania, której działanie jest powtarzane, jeśli użytkownik chce wyświetlenia kolejnych liczb ciągu. Wewnętrzna pętla `for` w wierszu 15. wykonuje faktyczne zadanie obliczenia kolejnej liczby w ciągu Fibonacciego i jednorazowo wyświetla pięć kolejnych liczb ciągu. W wierszu 19. mamy przypisanie `Num2` do zmiennej tymczasowej używanej później w wierszu 21. Zwróć uwagę, że w przypadku braku przypisania wartości tymczasowej zmodyfikowana wartość byłaby przypisywana w wierszu 21., co jest niepożądane. Dzięki trzem omówionym wierszom pętla powtarza działanie wraz z nowymi wartościami `Num1` i `Num2`, jeśli użytkownik naciśnie klawisz *t* oznaczający kontynuację.

Podsumowanie

Czytając tę lekcję, dowiedziałeś się, jak można utworzyć kod, który nie będzie wykonywany jedynie od początku do końca. Wiesz, jak utworzyć polecenia warunkowe tworzące alternatywne ścieżki wykonywania programu oraz jak powtarzać bloki kodu w pętli. Poznałeś konstrukcję `if...else` i użycie poleceń `switch-case` do zapewnienia obsługi różnych sytuacji, w zależności od wartości danej zmiennej.

W trakcie omawiania pętli poznałeś polecenie `goto`, ale jednocześnie zostałeś ostrzeżony przed jego stosowaniem, ponieważ może doprowadzić do utworzenia kodu trudnego do zrozumienia. Dowiedziałeś się, jak w C++ tworzyć pętle za pomocą konstrukcji `while`, `do...while` i `for`. Wiesz już, jak przygotować pętle w nieskończoność przeprowadzające iteracje (pętle działające w nieskończoność) oraz jak wykorzystać polecenia `continue` i `break` do lepszej kontroli wymienionych pętli.

Pytania i odpowiedzi

Pytanie: Co się stanie, jeśli pominę polecenie `break` w konstrukcji `switch-case`?

Odpowiedź: Polecenie `break` pozwala programowi na opuszczenie konstrukcji `switch`. Bez tego polecenia program będzie kontynuował działanie i wykonywał kolejne polecenia `case`.

Pytanie: W jaki sposób mogę zakończyć pętlę działającą w nieskończoność?

Odpowiedź: Użyj polecenia `break` do zakończenia działania pętli. Użycie polecenia `return` zamiast `break` powoduje opuszczenie również modułu funkcji.

Pytanie: Moja pętla `while` ma postać `while(Integer)`. Czy pętla będzie kontynuowała działanie, gdy wartość `Integer` wyniesie –1?

Odpowiedź: W idealnej sytuacji wyrażenie `while` powinno mieć wartość boolowską `true` lub `false`, w przeciwnym razie zostanie przyjęta następująca interpretacja: `false` to zero. Warunek, który nie będzie zerem, zostanie uznany za `true`. Ponieważ –1 nie jest zerem, wartością `while` będzie `true` i działanie pętli będzie kontynuowane. Jeżeli chcesz, aby pętla była wykonywana jedynie w przypadku wartości dodatnich, powinieneś użyć wyrażenia w postaci `while(Integer>0)`. Przedstawiona reguła sprawdza się we wszystkich pętlach i poleceniach warunkowych.

Pytanie: Czy istnieje pusta pętla `while` będąca odpowiednikiem pętli `for(;;)`?

Odpowiedź: Nie, pętli `while` zawsze towarzyszy wyrażenie warunkowe.

Pytanie: Pętlę `do...while(exp);` zastąpiłem pętlą `while(exp);`, wykonując operację kopiuj i wklej. Czy powinienem spodziewać się jakichkolwiek problemów?

Odpowiedź: Tak i to dużych! Pętla `while(exp);` jest prawidłową, choć pustą pętlą `while` ze względu na polecenie `null` (średnik) po słowie kluczowym `while`, nawet mimo umieszczenia po niej bloku poleceń. Wspomniany blok poleceń będzie wykonany jednokrotnie, ale poza pętlą. Powinieneś zachować ostrożność podczas kopiowania i wklejania kodu.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Dlaczego mam się przejmować wcięciami kodu w poleceniach bloków, zagnieżdżonych konstrukcjach `if` i zagnieżdżonych pętlach, skoro kod bez wcięć i tak kompiluje się bez problemów?
2. Masz zaimplementować szybką poprawkę z użyciem `goto`. Dlaczego powinieneś unikać takiego rozwiązania?
3. Czy możliwe jest utworzenie pętli `for`, w której następuje dekrementacja licznika? Jak będzie wyglądał kod tego rodzaju pętli?
4. Jaki jest problem w kodzie poniższej pętli?

```
for (int Counter=0; Counter==10; ++Counter)
   cout << Counter << " ";
```

Ćwiczenia

1. Utwórz pętlę `for` pozwalającą na uzyskanie dostępu do elementów tablicy w odwrotnej kolejności?
2. Utwórz zagnieżdżoną pętlę będącą odpowiednikiem przedstawionej w listingu 6.13, dodającej elementy do dwóch tablic, ale w odwrotnej kolejności.
3. Utwórz podobny do przedstawionego w listingu 6.15 program wyświetlający kolejne elementy ciągu Fibonacciego, ale proszący użytkownika o podanie liczby elementów, które mają zostać wygenerowane.
4. Utwórz konstrukcję `switch-case` wyświetlającą komunikat, czy dany kolor jest zaliczany do kolorów tęczy. Użyj stałej w postaci typu wyliczeniowego.

5. **Łowcy błędów:** Jaki jest problem w poniższym fragmencie kodu?

```
for (int Counter=0; Counter=10; ++Counter)
   cout << Counter << " ";
```

6. **Łowcy błędów:** Jaki jest problem w poniższym fragmencie kodu?

```
int LoopCounter = 0;
while(LoopCounter <5);
{
   cout << LoopCounter << " ";
   LoopCounter++;
}
```

7. **Łowcy błędów:** Jaki jest problem w poniższym fragmencie kodu?

```
cout << "Enter a number between 0 and 4" << endl;
int Input = 0;
cin >> Input;

switch (Input)
{
   case 0:
   case 1:
   case 2:
   case 3:
   case 4:
      cout << "Prawidłowe dane wejściowe" << endl;
   default:
      cout << "Nieprawidłowe dane wejściowe" << endl;
}
```

Lekcja 7

Funkcje

Dotychczas w książce przedstawiono proste programy, których cały kod znajdował się w funkcji main(). Takie rozwiązanie sprawdza się doskonale w naprawdę małych aplikacjach. Wraz ze wzrostem wielkości i poziomu skomplikowania programu zwiększa się również ilość kodu w funkcji main(), o ile nie zdecydujesz się na zmianę struktury programu i wykorzystanie w nim innych funkcji.

Wspomniane funkcje pozwalają na podział programu na mniejsze części oraz umożliwiają organizację logiki wykonywania aplikacji. Korzystając z funkcji, możesz podzielić treść programu na logiczne bloki wywoływane w odpowiedniej kolejności.

Funkcja jest więc podprogramem, który opcjonalnie może pobierać parametry i zwracać wartość oraz musi być wywołany w celu wykonania zadania.

Z tej lekcji dowiesz się:

- **dowiesz się, czym są funkcje i kiedy należy je stosować,**
- **poznasz prototypy i definicje funkcji,**
- **zobaczysz, jak przekazywać parametry funkcjom oraz jak zwracać wartość z funkcji,**
- **dowiesz się, jak przeciążać funkcje,**
- **poznasz funkcje rekurencyjne,**
- **poznasz funkcje lambda w standardzie C++11.**

Kiedy należy stosować funkcje?

Wyobraź sobie aplikację, która prosi użytkownika o podanie promienia okręgu, a następnie oblicza obwód i pole tej figury geometrycznej. Jedno z możliwych rozwiązań polega na umieszczeniu całego kodu w funkcji `main()`. Inny możliwy do zastosowania sposób to podział aplikacji na logiczne bloki; szczególnie interesujące są dwa z nich, odpowiedzialne za obliczenie pola oraz obwodu okręgu. Takie rozwiązanie przedstawiono w listingu 7.1.

Listing 7.1. Dwie funkcje odpowiedzialne w programie za obliczenie pola oraz obwodu okręgu na podstawie podanego promienia

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: const double Pi = 3.14159;
 4:
 5: // Deklaracje funkcji (prototypy).
 6: double Area(double InputRadius);
 7: double Circumference(double InputRadius);
 8:
 9: int main()
10: {
11:    cout << "Podaj promień: ";
12:    double Radius = 0;
13:    cin >> Radius;
14:
15:    // Wywołanie funkcji o nazwie "Area".
16:    cout << "Pole wynosi: " << Area(Radius) << endl;
17:
18:    // Wywołanie funkcji o nazwie "Circumference".
19:    cout << "Obwód wynosi: " << Circumference(Radius) << endl;
20:
21:    return 0;
22:  }
23:
24: // Definicje funkcji (implementacje).
25: double Area(double InputRadius)
26: {
27:    return Pi * InputRadius * InputRadius;
28: }
29:
30: double Circumference(double InputRadius)
31: {
32:    return 2 * Pi * InputRadius;
33: }
```

Wynik ▼

```
Podaj promień: 6.5
Pole wynosi: 132.732
Obwód wynosi: 40.8407
```

Analiza ▼

Na początku wydaje się, że otrzymujemy to samo, ale w innym opakowaniu. Jednak warto docenić podział na oddzielne funkcje obliczeń pola i obwodu, ponieważ dzięki temu można wielokrotnie wywoływać te funkcje, kiedy trzeba. Funkcja `main()` jest całkiem zwięzła i oddała logikę biznesową funkcjom `Area()` i `Circumference()`, które są wywoływane w wierszach (odpowiednio) 16. i 19.

W powyższym programie przedstawiono następujące zagadnienia związane z tworzeniem programów opartych na funkcjach.

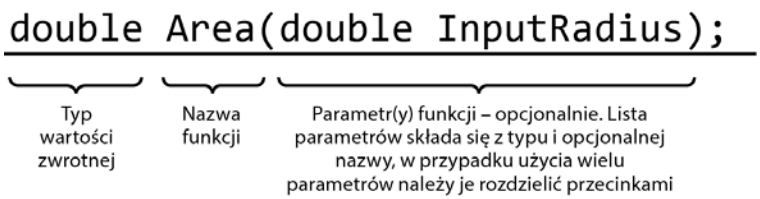
- Prototypy funkcji zostały *zadeklarowane* w wierszach 6. i 7., kompilator więc zna wyrażenia `Area()` i `Circumference()`, które są używane w funkcji `main()`.
- Funkcje `Area()` i `Circumference()` są wywoływane w funkcji `main()` w wierszach 16. i 19.
- Funkcja `Area()` została *zdefiniowana* w wierszach od 25. do 28., natomiast `Circumference()` w wierszach od 30. do 33.

Czym jest prototyp funkcji?

Spójrz ponownie na listing 7.1, a dokładnie na wiersze 6. i 7.:

```
double Area(double InputRadius);
double Circumference(double InputRadius);
```

Na rysunku 7.1 pokazano elementy składające się na prototyp funkcji.



RYSUNEK 7.1. Elementy składowe prototypu funkcji

Prototyp funkcji wskazuje wywoływaną funkcję (jej nazwa, tutaj `Area`), listę parametrów akceptowanych przez funkcję (tutaj jeden parametr o nazwie `InputRadius` i typie `double`) oraz typ wartości zwrotnej funkcji (tutaj `double`).

Bez prototypu funkcji po dotarciu do wierszy 16. i 19. w funkcji `main()` wywołania `Area()` i `Circumference()` byłyby dla kompilatora nieznane. Wymienione funkcje pobierają po jednym parametrze typu `double` i zwracają wartość tego samego typu. Dzięki prototypom kompilator rozpoznaje wymienione wywołania jako poprawne polecenia. Zadanie powiązania wywołania funkcji z jej implementacją oraz zagwarantowanie ich wykonania w działającym programie należy do linkera.

Uwaga

> Funkcja może mieć wiele parametrów rozdzielonych przecinkami, natomiast typ wartości zwrotnej może być tylko jeden.
>
> Kiedy tworzona jest funkcja, która nie musi zwracać żadnej wartości, jej typ zwrotny powinien zostać określony jako `void`.

Czym jest definicja funkcji?

Najbardziej interesujący nas fragment — implementacja funkcji — nosi nazwę *definicji*. Przeanalizujmy przedstawioną poniżej definicję funkcji `Area()`:

```
25: double Area(double InputRadius)
26: {
27:    return Pi * InputRadius * InputRadius;
28: }
```

Definicja funkcji zawsze składa się z blok poleceń. Polecenie `return` nie jest konieczne w przypadku funkcji, której typ zwrotny został zadeklarowany jako `void`. W analizowanej funkcji `Area()` konieczne jest użycie polecenia `return`, ponieważ typ jej wartości zwrotnej nie jest określony jako `void`, ale jako `double`. Blok poleceń zawiera polecenia ujęte w nawias klamrowy `{...}`, które są wykonywane po wywołaniu funkcji. W analizowanej funkcji `Area()` użyto parametru wejściowego `InputRadius` przechowującego podaną przez użytkownika wartość promienia; to jest *argument* wykorzystywany do obliczenia pola okręgu.

Czym jest wywołanie funkcji i argumenty?

Uruchomienie funkcji nosi nazwę *wywołania funkcji*. Kiedy funkcja została zadeklarowana wraz z *parametrami*, w trakcie jej wywołania konieczne jest

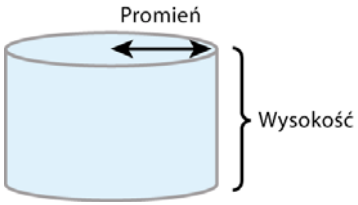
przekazanie *argumentów*, które po prostu są wartościami żądanymi przez funkcję jako parametry. Przeanalizujmy użyte w listingu 7.1 wywołanie funkcji `Area()`:

```
16:    cout << "Pole wynosi: " << Area(Radius) << endl;
```

W powyższym wierszu kodu `Area(Radius)` to wywołanie funkcji, w którym `Radius` jest argumentem przekazywanym funkcji `Area()`. Podczas wywołania program przechodzi do funkcji `Area()` i używa wartości promienia do obliczenia pola okręgu. Gdy działanie funkcji zostanie zakończone, zwraca ona wartość typu `double`. Wartość zwrotna jest następnie wyświetlana na ekranie za pomocą polecenia `cout`.

Tworzenie funkcji z wieloma parametrami

Przyjmujemy założenie, że tworzysz program, którego celem jest obliczenie pola walca pokazanego na rysunku 7.2.



RYSUNEK 7.2. Walec

W programie wykorzystamy następujący wzór:

```
Pole walca = pole górnego okręgu + pole dolnego okręgu + pole boku
           = Pi *  Promień^2 + Pi * Promień ^2 + 2 * Pi * Promień * Wysokość
           = 2 * Pi * Promień^2 + 2 * Pi * Promień * Wysokość
```

W programie w celu obliczenia pola walca konieczne jest użycie dwóch zmiennych przechowujących wartość promienia i wysokość walca. W takim przypadku podczas tworzenia funkcji obliczającej pole walca trzeba podać przynajmniej dwa parametry na liście parametrów w deklaracji funkcji. Poszczególne parametry muszą być rozdzielone przecinkami, tak jak przedstawiono w listingu 7.2.

Listing 7.2. Funkcja akceptująca dwa parametry i przeznaczona do obliczenia pola walca

```
0: #include <iostream>
1: using namespace std;
2:
3: const double Pi = 3.14159;
```

```
 4:
 5: // Deklaracja funkcji zawiera dwa parametry.
 6: double SurfaceArea(double Radius, double Height);
 7:
 8: int main()
 9: {
10:    cout << "Podaj promień walca: ";
11:    double InRadius = 0;
12:    cin >> InRadius;
13:    cout << "Podaj wysokość walca: ";
14:    double InHeight = 0;
15:    cin >> InHeight;
16:
17:    cout << "Pole walca wynosi: " << SurfaceArea(InRadius, InHeight) <<
       ➥endl;
18:
19:    return 0;
20: }
21:
22: double SurfaceArea(double Radius, double Height)
23: {
24:    double Area = 2 * Pi * Radius * Radius + 2 * Pi * Radius * Height;
25:    return Area;
26: }
```

Wynik ▼

```
Podaj promień walca: 3
Podaj wysokość walca: 6.5
Pole walca wynosi: 179.071
```

Analiza ▼

W wierszu 6. znajduje się deklaracja funkcji `SurfaceArea()` wraz z dwoma rozdzielonymi przecinkiem parametrami typu double: `Radius` i `Height`. Wiersze od 22. do 26. zawierają definicję, tzn. implementację funkcji `SurfaceArea()`. Jak możesz zobaczyć, parametry wejściowe `Radius` i `Height` są używane do obliczenia wartości umieszczonej następnie w zmiennej `Area`, która jest zwracana wywołującemu funkcję.

Uwaga

> Parametry funkcji są jak zmienne lokalne. Pozostają ważne jedynie w zakresie funkcji. Dlatego też w listingu 7.2 parametry `Radius` i `Height` funkcji `SurfaceArea()` zachowują ważność i mogą być używane jedynie w wymienionej funkcji, a nie poza nią.

Tworzenie funkcji bez parametrów i bez wartości zwrotnej

Jeżeli jedynym zadaniem funkcji będzie wyświetlenie komunikatu np. *Witaj, świecie*, wtedy taka funkcja nie potrzebuje żadnych parametrów (przecież tylko wyświetla komunikat) i prawdopodobnie nie musi również zwracać żadnej wartości (nie oczekujesz niczego poza wyświetleniem komunikatu). Przykład tego rodzaju funkcji przedstawiono w listingu 7.3.

Listing 7.3. Funkcja bez parametrów i bez wartości zwrotnej

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: void SayHello();
 4:
 5: int main()
 6: {
 7:    SayHello();
 8:    return 0;
 9: }
10:
11: void SayHello()
12: {
13:    cout << "Witaj, świecie" << endl;
14: }
```

Wynik ▼

```
Witaj, świecie
```

Analiza ▼

Zauważ, że w wierszu 3. prototyp funkcji powoduje zadeklarowanie funkcji o nazwie `SayHello()` i typie `void` oznaczającym brak wartości zwrotnej. Dlatego też znajdująca się w wierszach od 11. do 14. definicja funkcji nie zawiera polecenia `return`. Nawet wywołanie funkcji w `main()` w wierszu 7. nie powoduje przypisania wartości funkcji żadnej zmiennej i nie używa jej w żadnym wyrażeniu, ponieważ funkcja `SayHello()` nie zwraca żadnej wartości.

Parametry funkcji wraz z wartościami domyślnymi

W przedstawionych dotąd przykładach przyjęto, że wartość Pi jest stała i użytkownik nigdy nie miał możliwości jej zmiany. Jednak użytkownik może być zainteresowany otrzymaniem mniej lub bardziej dokładnego wyniku obliczeń. W jaki sposób można utworzyć funkcję, która przyjmie pewną wartość Pi, o ile użytkownik jej nie zmieni?

Jednym z rozwiązań jest użycie dodatkowego parametru w funkcji `Area()` dla wartości Pi oraz przypisanie jej wartości domyślnej. Po wprowadzeniu wspomnianej zmiany definicja funkcji `Area()` w listingu 7.1 będzie przedstawiała się następująco:

```
double Area(double InputRadius, double Pi = 3.14);
```

Zwróć uwagę na drugi parametr `Pi` i przypisaną mu wartość domyślną wynoszącą 3.14. Drugi parametr funkcji jest opcjonalny i dlatego też funkcję `Area()` nadal można wywołać w następujący sposób:

```
Area(Radius);
```

W powyższym wywołaniu drugi parametr został zignorowany i tym samym użyta będzie wartość domyślna Pi wynosząca 3.14. Jeśli jednak użytkownik poda inną wartość Pi, wtedy można jej użyć w wywołaniu funkcji `Area()` w następujący sposób:

```
Area(Radius, Pi); // Zdefiniowana przez użytkownika wartość Pi.
```

W listingu 7.4 pokazano, jak można tworzyć funkcje zawierające dla parametrów wartości domyślne, które mogą być nadpisane wartościami podanymi przez użytkownika, o ile będą dostępne.

Listing 7.4. Funkcja obliczająca pole okręgu i używająca Pi jako drugiego parametru wraz z wartością domyślną 3.14

```
0: #include <iostream>
1: using namespace std;
2:
3: // Deklaracja funkcji (prototyp).
4: double Area(double InputRadius, double Pi = 3.14);  // Parametr Pi wraz
   ↪z wartością domyślną.
5:
6: int main()
7: {
8:    cout << "Podaj promień: ";
```

```
 9:    double Radius = 0;
10:    cin >> Radius;
11:
12:    cout << "Pi wynosi 3.14, czy chcesz zmienić tę wartość (t/n)? ";
13:    char ChangePi = 'n';
14:    cin >> ChangePi;
15:
16:    double CircleArea = 0;
17:    if (ChangePi == 't')
18:    {
19:       cout << "Podaj nową wartość Pi: ";
20:       double NewPi = 3.14;
21:       cin >> NewPi;
22:       CircleArea = Area (Radius, NewPi);
23:    }
24:    else
25:       CircleArea = Area(Radius); // Zignorowanie drugiego parametru i użycie
          ↪wartości domyślnej.
26:
27:    // Wywołanie funkcji o nazwie "Area".
28:    cout << "Pole wynosi: " << CircleArea << endl;
29:
30:    return 0;
31: }
32:
33: // W definicji funkcji nie są ponownie podawane wartości domyślne.
34: double Area(double InputRadius, double Pi)
35: {
36:    return Pi * InputRadius * InputRadius;
37: }
```

Wynik ▼

```
Podaj promień: 1
Pi wynosi 3.14, czy chcesz zmienić tę wartość (t/n)? n
Pole wynosi: 3.14
```

Następne uruchomienie programu:

```
Podaj promień: 1
Pi wynosi 3.14, czy chcesz zmienić tę wartość (t/n)? t
Podaj nową wartość Pi: 3.1416
Pole wynosi: 3.1416
```

Analiza ▼

W przedstawionych powyżej danych wyjściowych widać, że w dwóch sesjach pracy z programem podana została taka sama wartość promienia — 1. Jednak w drugim przypadku użytkownik zdecydował się na zmianę precyzji parametru

Pi, stąd otrzymany wynik jest odmienny. Zwróć uwagę, że w obu sesjach następuje wywołanie tej samej funkcji (w wierszach 22. i 25.). W wierszu 25. funkcja `Area()` jest wywoływana bez drugiego parametru, wtedy stosowana jest wartość domyślna Pi wynosząca 3.14 i zdefiniowana w deklaracji funkcji (patrz wiersz 4.).

Uwaga

Istnieje możliwość użycia wielu parametrów wraz z wartościami domyślnymi. Powinny się one znajdować na końcu listy parametrów.

Rekurencja, czyli funkcja wywołująca samą siebie

W pewnych sytuacjach może się zdarzyć, że funkcja będzie wywoływała samą siebie. Tego rodzaju funkcja jest nazywana *funkcją rekurencyjną*. Trzeba koniecznie pamiętać, że funkcja rekurencyjna powinna posiadać czytelnie zdefiniowany warunek zakończenia działania bez ponownego wywoływania samej siebie.

Ostrzeżenie

Kiedy brakuje warunku pozwalającego na zakończenie działania funkcji lub istnieje błąd w programie, działanie programu może zostać zakleszczone w funkcji rekurencyjnej, która będzie nieustannie wywoływała samą siebie. Zatrzymanie działania takiej funkcji nastąpi jedynie po przepełnieniu bufora prowadzącego do awarii aplikacji.

Funkcja rekurencyjna może być użyteczna podczas obliczania liczb ciągu Fibonacciego, co przedstawiono w listingu 7.5. Ciąg rozpoczyna się od dwóch liczb: 0 i 1.

```
F(0) = 0
F(1) = 1
```

Wartość kolejnej liczby w ciągu to suma dwóch poprzednich liczb. Dlatego też *n*-tą wartość (dla `n > 1`) można obliczyć za pomocą poniższego (rekurencyjnego) wzoru:

```
Fibonacci(n) = Fibonacci(n – 1) + Fibonacci(n – 2)
```

Otrzymany w ten sposób ciąg Fibonacciego przedstawia się następująco:

```
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8 itd.
```

Listing 7.5. Użycie funkcji rekurencyjnej w celu obliczenia liczby w ciągu Fibonacciego

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int GetFibNumber(int FibIndex)
 4: {
 5:    if (FibIndex < 2)
 6:       return FibIndex;
 7:    else // Rekurencja, jeśli FibIndex >= 2.
 8:       return GetFibNumber(FibIndex - 1) + GetFibNumber(FibIndex - 2);
 9: }
10:
11: int main()
12: {
13:    cout << "Podaj rozpoczynający się od zera indeks liczby w ciągu
       ↪Fibonacciego: ";
14:    int Index = 0;
15:    cin >> Index;
16:
17:    cout << "Liczba ciągu Fibonacciego to: " << GetFibNumber(Index) <<
       ↪endl;
18:    return 0;
19: }
```

Wynik ▼

```
Podaj rozpoczynający się od zera indeks liczby w ciągu Fibonacciego: 6
Liczba ciągu Fibonacciego to: 8
```

Analiza ▼

Zdefiniowana w wierszach od 3. do 9. funkcja `GetFibNumber()` jest funkcją rekurencyjną, która wywołuje samą siebie w wierszu 8. Zwróć uwagę na warunek wyjścia określony w wierszach 5. i 6.: jeśli wartość indeksu jest mniejsza niż dwa, funkcja nie będzie uruchomiona rekurencyjnie. Ponieważ kolejne wywołania funkcji powodują zmniejszenie wartości `FibIndex`, w pewnym momencie następuje spełnienie warunku zakończenia działania funkcji i rekurencja ustaje.

Funkcje z wieloma poleceniami return

Funkcja może zawierać więcej niż tylko jedno polecenie `return` w swojej definicji. Funkcja może zwracać wartości na różnych etapach działania i wielokrotnie w trakcie jej wykonywania, co przedstawiono w listingu 7.6. W zależności od logiki i wymagań aplikacji może to być dobra lub kiepska praktyka programistyczna.

Listing 7.6. Użycie wielu poleceń return w jednej funkcji

```
 0: #include <iostream>
 1: using namespace std;
 2: const double Pi = 3.14159;
 3:
 4: void QueryAndCalculate()
 5: {
 6:    cout << "Podaj promień: ";
 7:    double Radius = 0;
 8:    cin >> Radius;
 9:
10:    cout << "Pole wynosi: " << Pi * Radius * Radius << endl;
11:
12:    cout << "Czy chcesz obliczyć obwód (t/n)? ";
13:    char CalcCircum = 'n';
14:    cin >> CalcCircum;
15:
16:    if (CalcCircum == 'n')
17:       return;
18:
19:    cout << "Obwód: " << 2 * Pi * Radius << endl;
20:    return;
21: }
22:
23: int main()
24: {
25:    QueryAndCalculate ();
26:
27:    return 0;
28: }
```

Wynik ▼

```
Podaj promień: 1
Pole wynosi: 3.14159
Czy chcesz obliczyć obwód (t/n)? t
Obwód: 6.28319
```

Następne uruchomienie programu:

```
Podaj promień: 1
Pole wynosi: 3.14159
Czy chcesz obliczyć obwód (t/n)? n
```

Analiza ▼

Funkcja `QueryAndCalculate()` zawiera dwa polecenia `return` w wierszach 17. i 20. Funkcja ta wyświetla użytkownikowi pytanie, czy chce obliczenia również obwodu okręgu. Jeżeli użytkownik naciśnie klawisz *n* (odpowiedź odmowna), program kończy działanie przez użycie polecenia `return`. W przypadku naciśnięcia innego dowolnego klawisza następuje obliczenie obwodu okręgu i dopiero wówczas zakończenie działania programu.

Ostrzeżenie

Zachowaj ostrożność podczas umieszczania w funkcji wielu poleceń `return`. Znacznie łatwiej zrozumieć działanie funkcji, której kod jest wykonywany od początku do końca i która zwraca wartość na końcu niż funkcji zawierającej polecenia `return` umieszczone w różnych miejscach.

W kodzie przedstawionym w listingu 7.6 można uniknąć użycia więcej niż jednego polecenia `return` przez zmianę warunku konstrukcji `if` na przedstawiony niżej:

```
if (CalcCircum == 't')
   cout << "Obwód: " << 2 * Pi * Radius << endl;
```

Użycie funkcji do pracy z różnymi formami danych

Funkcje nie muszą ograniczać się tylko do otrzymywania pojedynczych wartości: funkcji można również przekazać tablicę wartości. Istnieje możliwość utworzenia dwóch funkcji o takiej samej nazwie i wartości zwrotnej, ale o różnych parametrach. Można przygotować funkcję w taki sposób, że jej parametry nie są tworzone i usuwane w trakcie wywołania funkcji. Zamiast tego można użyć referencji zachowujących ważność nawet po opuszczeniu funkcji, co pozwala na operowanie w wywołaniu funkcji większą ilością danych lub parametrów. W tym podrozdziale dowiesz się, jak przekazywać tablice funkcjom, przeciążać funkcje i przekazywać argumenty funkcji przez referencje.

Przeciążanie funkcji

Funkcje o takich samych nazwach i typach wartości zwrotnych, ale o różnych parametrach, są nazywane funkcjami przeciążonymi. Funkcja przeciążona może być całkiem użyteczna w aplikacjach, gdzie funkcja o danej nazwie i generująca określony typ danych wyjściowych może być wywoływana z różnym zestawem parametrów. Przyjmujemy założenie, że tworzysz aplikację przeznaczoną do obliczenia pola okręgu oraz walca. Funkcja obliczająca pole okręgu potrzebuje parametru w postaci promienia okręgu. Natomiast funkcja obliczająca pole walca, oprócz promienia okręgu podstawy walca, potrzebuje również wysokości walca. Obie funkcje zwracają dane takiego samego typu określające pole. Dlatego też język C++ pozwala na zdefiniowanie dwóch przeciążonych funkcji o nazwie `Area()`, które będą zwracały dane typu `double`. Jedna z wymienionych funkcji będzie pobierała tylko jeden parametr danych wejściowych (promień), z kolei druga będzie pobierała dwa parametry danych wejściowych (promień i wysokość). Kod wspomnianej funkcji znajdziesz w listingu 7.7.

Listing 7.7. Użycie przeciążonej funkcji do obliczenia pola okręgu lub walca

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: const double Pi = 3.14159;
 4:
 5: double Area(double Radius);    // Funkcja obliczająca pole okręgu.
 6: double Area(double Radius, double Height); // Funkcja obliczająca pole walca.
 7:
 8: int main()
 9: {
10:    cout << "Aby obliczyć pole walca, naciśnij z, okręgu, naciśnij c: ";
11:    char Choice = 'z';
12:    cin >> Choice;
13:
14:    cout << "Podaj promień: ";
15:    double Radius = 0;
16:    cin >> Radius;
17:
18:    if (Choice == 'z')
19:    {
20:       cout << "Podaj wysokość walca: ";
21:       double Height = 0;
22:       cin >> Height;
```

```
23:
24:       // Wywołanie przeciążonej funkcji obliczającej pole walca.
25:       cout << "Pole walca wynosi: " << Area (Radius, Height) << endl;
26:    }
27:    else
28:       cout << "Pole okręgu wynosi: " << Area (Radius) << endl;
29:
30:    return 0;
31: }
32:
33: // Obliczenie pola okręgu.
34: double Area(double Radius)
35: {
36:    return Pi * Radius * Radius;
37: }
38:
39: // Przeciążona funkcja dla walca.
40: double Area(double Radius, double Height)
41: {
42:    // Ponowne użycie pola okręgu.
43:    return 2 * Area (Radius) + 2 * Pi * Radius * Height;
44: }
```

Wynik ▼

```
Aby obliczyć pole walca, naciśnij z, okręgu, naciśnij c: z
Podaj promień: 2
Podaj wysokość walca: 5
Pole walca wynosi: 87.9646
```

Następne uruchomienie programu:

```
Aby obliczyć pole walca, naciśnij z, okręgu, naciśnij c: c
Podaj promień: 1
Pole okręgu wynosi: 3.14159
```

Analiza ▼

W wierszach 5. i 6. znajdują się prototypy przeciążonych wariantów funkcji `Area()`, jedna akceptuje pojedynczy parametr w postaci promienia okręgu, natomiast druga akceptuje dwa parametry w postaci promienia i wysokości walca. Obie funkcje mają takie same nazwy, czyli `Area()`, i typ wartości zwrotnej (`double`), ale różnią się zestawem parametrów — stąd ich przeciążenie. Definicje przeciążonych funkcji znajdują się w wierszach od 34. do 44., zadaniem wspomnianych funkcji jest obliczenie odpowiednio pola okręgu na podstawie promienia i pola walca na podstawie jego promienia i wysokości. Na pole

walca składają się pola dwóch okręgów (na dole i na górze walca) oraz pole powierzchni bocznej. Dlatego też, jak przedstawiono w wierszu 43., przeciążona wersja funkcji obliczającej pole walca może ponownie wykorzystać obliczone wcześniej pole okręgu.

Przekazanie funkcji tablicy wartości

Funkcja wyświetlająca liczbę całkowitą może być przedstawiona w następujący sposób:

```
void DisplayInteger(int Number);
```

Z kolei funkcja wyświetlająca tablicę liczb całkowitych ma nieco odmienny prototyp:

```
void DisplayIntegers(int[] Numbers, int Length);
```

Pierwszy parametr to informacja dla funkcji, że dane wejściowe są w postaci tablicy, natomiast drugi parametr wskazuje wielkość tablicy, zatem funkcja będzie mogła wykorzystać tablicę bez przekraczania jej granic. Odpowiedni kod przedstawiono w listingu 7.8.

Listing 7.8. Funkcja, która pobiera parametr w postaci tablicy

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: void DisplayArray(int Numbers[], int Length)
 4: {
 5:    for (int Index = 0; Index < Length; ++Index)
 6:       cout << Numbers[Index] << " ";
 7:
 8:    cout << endl;
 9: }
10:
11: void DisplayArray(char Characters[], int Length)
12: {
13:    for (int Index = 0; Index < Length; ++Index)
14:       cout << Characters[Index] << " ";
15:
16:    cout << endl;
17: }
18:
19: int main()
20: {
21:    int MyNumbers[4] = {24, 58, -1, 245};
22:    DisplayArray(MyNumbers, 4);
```

```
23:
24:    char MyStatement[7] = {'W', 'i', 't', 'a', 'j', '!', '\0'};
25:    DisplayArray(MyStatement, 7);
26:
27:    return 0;
28: }
```

Wynik ▼

```
24 58 -1 245
W i t a j !
```

Analiza ▼

W kodzie znajdują się dwie przeciążone funkcje o nazwie `DisplayArray()`. Jedna powoduje wyświetlenie zawartości tablicy liczb całkowitych, natomiast druga wyświetla zawartość tablicy znaków. W wierszach 22. i 25. funkcje te są wywoływane z danymi wejściowymi w postaci tablicy (odpowiednio) liczb całkowitych i znaków. Zwróć uwagę, że w deklaracji i inicjalizacji tablicy znaków (wiersz 24.) na końcu celowo został umieszczony znak `null` — to najlepsza praktyka i jednocześnie dobry nawyk — choć tablica nie jest używana w aplikacji jako ciąg tekstowy polecenia `cout` lub w sposób taki jak `cout << MyStatement;`.

Przekazywanie argumentów przez referencję

Spójrz raz jeszcze na funkcję z listingu 7.1 odpowiedzialną za obliczenie pola okręgu na podstawie podanego promienia:

```
24: // Definicje funkcji (implementacje).
25: double Area(double InputRadius)
26: {
27:    return Pi * InputRadius * InputRadius;
28: }
```

W powyższym fragmencie kodu parametr `InputRadius` zawiera wartość, która jest mu przypisywana w chwili wywołania funkcji w `main()`:

```
15:    // Wywołanie funkcji o nazwie "Area".
16:    cout << "Pole wynosi: " << Area(Radius) << endl;
```

Oznacza to, że zmienna `Radius` w funkcji `main()` nie jest w żaden sposób modyfikowana przez wywołanie funkcji, ponieważ funkcja `Area()` pracuje na kopii `Radius` wartości przechowywanej w zmiennej `InputRadius`. Zdarzają się sytuacje, gdy potrzebna jest funkcja wykorzystująca zmienną, która modyfikuje

wartość dostępną poza daną funkcją, np. w funkcji wywołującej. W takich przypadkach należy zadeklarować parametr pobierający argument *przez referencję*. Poniżej przedstawiono wersję funkcji `Area()` obliczającej i zwracającej pole jako parametr przekazywany przez referencję:

```
// Parametr danych wyjściowych Result przekazany przez referencję.
void Area(double Radius, double& Result)
{
   Result = Pi * Radius * Radius;
}
```

Zauważ, że funkcja `Area()` w przedstawionej powyżej postaci pobiera dwa parametry. Nie przeocz znaku & znajdującego się obok drugiego parametru (`Result`). Znak ten informuje kompilator, że drugi argument *nie powinien* być kopiowany do funkcji, zamiast tego następuje przekazanie referencji do zmiennej. Typ wartości zwrotnej został zmieniony na `void`, ponieważ funkcja nie zwraca obliczonego pola jako wartości zwrotnej, a umieszcza w parametrze dostępnym przez referencję. Zwrot wartości przez referencję został przedstawiony w listingu 7.9, w którym znajduje się kod odpowiedzialny za obliczenie pola okręgu.

Listing 7.9. Podanie pola okręgu jako parametru dostępnego przez referencję, a nie jako typowej wartości zwrotnej

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: const double Pi = 3.1416;
 4:
 5: // Parametr danych wyjściowych Result przekazany przez referencję.
 6: void Area(double Radius, double& Result)
 7: {
 8:    Result = Pi * Radius * Radius;
 9: }
10:
11: int main()
12: {
13:    cout << "Podaj promień: ";
14:    double Radius = 0;
15:    cin >> Radius;
16:
17:    double AreaFetched = 0;
18:    Area(Radius, AreaFetched);
19:
20:    cout << "Pole wynosi: " << AreaFetched << endl;
21:    return 0;
22: }
```

Wynik ▼

```
Podaj promień: 2
Pole wynosi: 12.5664
```

Analiza ▼

Zwróć uwagę na wiersze 17. i 18., w którym następuje wywołanie funkcji `Area()` wraz z dwoma parametrami, drugi z wymienionych wierszy zawiera wynik. Ponieważ funkcja `Area()` pobiera drugi parametr przez referencję, zmienna `Result` użyta w funkcji `Area()` — patrz wiersz 8. — prowadzi do tego samego miejsca w pamięci, co zmienna `AreaFetched` typu `double` zadeklarowana w funkcji `main()` w wierszu 17. Dlatego też wynik obliczony w wierszu 8. (funkcja `Area()`) jest dostępny w funkcji `main()` i wyświetlony na ekranie (wiersz 20.).

Uwaga

Za pomocą polecenia `return` funkcja może zwrócić tylko jedną wartość. Jeśli funkcja musi wykonać operacje wpływające na wiele wartości wymaganych przez wywołującego, przekazanie argumentów przez referencję to jedyny sposób, aby funkcja zapewniła modyfikacje wielu wartości dostępnych dla modułu, który ją wywołuje.

Jak wywołania funkcji są obsługiwane przez mikroprocesor?

Wprawdzie nie musisz koniecznie wiedzieć, jak wywołania funkcji są zaimplementowane na poziomie mikroprocesora, ale taka wiedza może okazać się interesująca. Pomoże w zrozumieniu, dlaczego język C++ oferuje możliwość używania funkcji typu inline, które będą omówione dalej w tej lekcji.

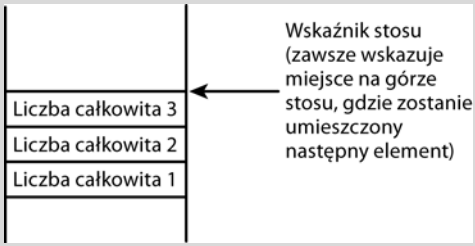
Wywołanie funkcji oznacza, że mikroprocesor przystępuje do wykonania następnej instrukcji należącej do wywoływanej funkcji, ale znajdującej się w innym (niekolejnym) adresie w pamięci. Po zakończeniu wykonywania instrukcji w funkcji mikroprocesor wraca do adresu, w którym znajdował się przed wywołaniem funkcji. Aby zaimplementować tego rodzaju logikę, kompilator konwertuje wywołania funkcji na instrukcje `CALL` dla mikroprocesora i podaje adres, z którego ma być pobrana następna instrukcja —adres ten należy do funkcji. Podczas kompilacji samej funkcji kompilator konwertuje polecenia `return` na instrukcje `RET` dla mikroprocesora.

Kiedy mikroprocesor napotka instrukcję CALL, zapisuje na stosie położenie instrukcji do wykonania po wywołaniu funkcji, a następnie przechodzi do położenia w pamięci wskazywanego przez instrukcję CALL.

Zrozumienie stosu

Stos to struktura typu Last-In-First-Out (ostatni na wejściu, pierwszy na wyjściu) umieszczona w pamięci i przypominająca np. stos talerzy. Gdy bierzesz talerz z góry stosu, jest to ostatni talerz położony na stosie. Umieszczenie danych na stosie nosi nazwę operacji *push*, natomiast pobranie danych ze stosu to operacja *pop*. Kiedy stos pnie się w górę, wskaźnik stosu zawsze ulega inkrementacji wraz z stosem i wskazuje miejsce na górze stosu (patrz rysunek 7.3).

RYSUNEK 7.3. Obrazowe przedstawienie stosu zawierającego trzy liczby całkowite



Natura stosu powoduje, że jest on optymalnym rozwiązaniem do obsługi wywołań funkcji. W trakcie wywołania funkcji wszystkie zmienne lokalne są umieszczane na stosie. Po zakończeniu działania funkcji zmienne są po prostu usuwane ze stosu, a wskaźnik stosu powraca do poprzedniego położenia.

Wspomniane miejsce w pamięci zawiera instrukcje należące do funkcji. Mikroprocesor wykonuje je, aż do chwili napotkania polecenia RET (kod mikroprocesora dla użytego przez programistę polecenia `return`). Polecenie RET powoduje, że mikroprocesor usuwa ze stosu adres umieszczony tam po wywołaniu instrukcji CALL. Adres ten zawiera położenie wywołującej funkcji, od którego ma być kontynuowane działanie. W ten sposób mikroprocesor powraca do miejsca, w którym nastąpiło wywołanie funkcji, i kontynuuje działanie programu od kolejnego polecenia.

Funkcje typu inline

Wywołanie zwykłej funkcji jest przekształcane na instrukcję CALL, co skutkuje przeprowadzeniem operacji na stosie i rozpoczęciem wykonywania funkcji przez mikroprocesor. Wydaje się, że to oznacza wiele pracy do wykonania, ale wszystko dzieje się bardzo szybko — przynajmniej w większości przypadków.

Co jednak dzieje się w przypadku bardzo prostych funkcji, takich jak przedstawiona niżej?

```
double GetPi()
{
   return 3.14159;
}
```

Obciążenie związane z faktycznym wywołaniem funkcji może być całkiem wysokie i zabierać znaczną ilość czasu, w porównaniu do czasu potrzebnego na wykonanie funkcji `GetPi()`. Dlatego też kompilator C++ pozwala programistom na zadeklarowanie tego rodzaju funkcji jako inline. Słowo kluczowe `inline` oznacza zdefiniowane przez programistę żądanie rozwinięcia danej funkcji w momencie jej wywoływania.

```
inline double GetPi()
{
   return 3.14159;
}
```

Podobnie funkcja jedynie mnożąca liczbę przez dwa lub wykonująca inne tego rodzaju proste operacje jest dobrym kandydatem do zdefiniowania jako inline. Przykład użycia funkcji inline w programie przedstawiono w listingu 7.10.

Listing 7.10. Użycie funkcji typu inline, która mnoży przez dwa podaną liczbę całkowitą

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: // Zdefiniowanie funkcji typu inline, która mnoży przez dwa podaną liczbę całkowitą.
 4: inline long DoubleNum (int InputNum)
 5: {
 6:    return InputNum * 2;
 7: }
 8:
 9: int main()
10: {
11:    cout << "Podaj liczbę całkowitą: ";
12:    int InputNum = 0;
13:    cin >> InputNum;
14:
15:    // Wywołanie funkcji typu inline.
16:    cout << "Pomnożona przez dwa: " << DoubleNum(InputNum) << endl;
17:
18:    return 0;
19: }
```

Wynik ▼

```
Podaj liczbę całkowitą: 35
Pomnożona przez dwa: 70
```

Analiza ▼

Słowo kluczowe `inline` zostało użyte w wierszu 4. Kompilator zwykle odczytuje to słowo kluczowe jako żądanie umieszczenia treści funkcji `DoubleNum` bezpośrednio w miejscu jej wywoływania (wiersz 16.), co przyśpiesza wykonanie kodu.

Klasyfikacja funkcji jako inline może skutkować rozdęciem kodu, zwłaszcza wtedy, kiedy funkcja inline przeprowadza skomplikowane operacje przetwarzania. Użycie słowa kluczowego `inline` powinno być ograniczone do minimum i stosowane jedynie dla bardzo małych i prostych funkcji, takich jak przedstawione wcześniej.

Uwaga

> Większość nowoczesnych kompilatorów C++ oferuje różne opcje optymalizacji wydajności działania programu. Niektóre, np. kompilator Microsoft C++, dają możliwość przeprowadzenia optymalizacji wielkości i szybkości działania programu. Optymalizacja pod kątem wielkości programu może być całkiem ważna, gdy tworzone jest oprogramowanie dla urządzeń wyposażonych w niewielką ilość pamięci. Podczas optymalizacji pod kątem wielkości programu kompilator może często odrzucać żądania funkcji typu inline, jeśli uzna, że doprowadzą one do nadmiernego rozdęcia kodu.
>
> Z kolei podczas optymalizacji pod kątem szybkości działania programu kompilator najczęściej dostrzega i wykorzystuje możliwość użycia funkcji typu inline, o ile ma to sens. Kompilator czasami stosuje funkcje typu inline nawet wtedy, gdy nie użyłeś słowa kluczowego `inline` w definicji funkcji.

C++11

Funkcja lambda

W tym punkcie zostanie przedstawione wprowadzenie do koncepcji, która nie jest zbyt łatwa dla początkujących. Przejrzyj przedstawiony tutaj opis i spróbuj ją zrozumieć. Dokładne omówienie funkcji lambda zostanie przedstawione w lekcji 22., zatytułowanej „Wyrażenia lambda w C++11”.

Funkcje lambda są bardzo użyteczne, jeśli często tworzysz kod z użyciem algorytmów biblioteki STL w celu sortowania lub przetwarzania danych,

np. zawartych w kontenerach STL, takich `std::vector` (tablica dynamiczna). Sortowanie najczęściej oznacza konieczność dostarczenia predykatu binarnego implementowanego jako operator w klasie, co oznacza żmudne tworzenie dużej ilości kodu. Kompilatory zgodne ze standardem C++11 pozwalają na tworzenie funkcji lambda i tym samym znaczne zmniejszenie ilości kodu, co przedstawiono w listingu 7.11.

Listing 7.11. Użycie funkcji lambda w celu posortowania i wyświetlenia elementów tablicy

```
 0: #include <iostream>
 1: #include <algorithm>
 2: #include <vector>
 3: using namespace std;
 4:
 5: void DisplayNums(vector<int>& DynArray)
 6: {
 7:    for_each (DynArray.begin(), DynArray.end(), \
 8:            [](int Element) {cout << Element << " ";} ); // Funkcja lambda.
 9:
10:    cout << endl;
11: }
12:
13: int main()
14: {
15:    vector<int> MyNumbers;
16:    MyNumbers.push_back(501);
17:    MyNumbers.push_back(-1);
18:    MyNumbers.push_back(25);
19:    MyNumbers.push_back(-35);
20:
21:    DisplayNums(MyNumbers);
22:
23:    cout << "Sortowanie w kolejności malejącej" << endl;
24:
25:    sort (MyNumbers.begin(), MyNumbers.end(), \
26:         [](int Num1, int Num2) {return (Num2 < Num1); } );
27:
28:    DisplayNums(MyNumbers);
29:
30:    return 0;
31: }
```

Wynik ▼

```
501 -1 25 -35
Sortowanie w kolejności malejącej
501 25 -1 -35
```

Analiza ▼

Program zawiera liczby całkowite umieszczone w tablicy dynamicznej dostarczanej przez bibliotekę standardową C++ w postaci `std::vector` (patrz wiersze od 15. do 19.). Funkcja o nazwie `DisplayNums()` używa algorytmu STL do przeprowadzenia iteracji przez wszystkie elementy tablicy i wyświetlenia ich wartości. W trakcie wspomnianej operacji używana jest funkcja lambda z wiersza 8. zamiast długiej, jednoargumentowej funkcji predykatu. Zastosowany w wierszu 25. obiekt `std::sort` również używa predykatu binarnego (wiersz 26.) w postaci funkcji lambda, która zwraca wartość `true`, jeśli druga liczba jest mniejsza od pierwszej, a tym samym sortuje kolekcję w kolejności malejącej.

Składnia funkcji lambda przedstawia się następująco:

```
[parametry opcjonalne](lista parametrów){ polecenia; }
```

Podsumowanie

W tej lekcji poznałeś podstawy programowania modularnego. Dowiedziałeś się, w jaki sposób funkcje mogą pomóc w utworzeniu lepszej struktury kodu oraz w ponownym wykorzystaniu opracowanych algorytmów. Wiesz, że funkcje mogą pobierać parametry i zwracać wartości. Wspomniane parametry mogą mieć wartości domyślne możliwe do nadpisana przez wywołującego funkcję. Ponadto parametry mogą zawierać argumenty przekazywane przez referencje. Poznałeś sposoby przekazywania tablic funkcjom i tworzenia funkcji przeciążonych, które mają tę samą nazwę i typ wartości zwrotnej, ale inną listę parametrów.

W lekcji pokrótce omówiono także funkcje lambda. Wprowadzone w standardzie C++11 funkcje lambda mogą zmienić sposób tworzenia aplikacji C++, zwłaszcza wtedy, kiedy używana jest biblioteka STL.

Pytania i odpowiedzi

Pytanie: Co się stanie, jeśli utworzę funkcję rekurencyjną, która będzie działać w nieskończoność?

Odpowiedź: Wykonywanie programu nie zostanie zakończone. To wcale nie musi być złe, mamy pętle `while(true)` i `for(;;)`, które dają taki sam efekt. Jednak kolejne wywołania funkcji rekurencyjnej zabierają coraz większą ilość miejsca na stosie, które nie jest nieskończone i wreszcie ulegnie wyczerpaniu. W takim przypadku dojdzie do awarii aplikacji z powodu przepełnienia stosu.

Pytanie: Dlaczego nie powinienem definiować każdej funkcji jako typu inline? Czy funkcja tego typu nie jest wykonywana szybciej?

Odpowiedź: To nie jest takie proste. Zdefiniowanie funkcji jako inline oznacza, że we wszystkich miejscach jej wywołania nastąpi umieszczenie jej kodu, co zwiększa ogólną ilość kodu programu. Większość nowoczesnych kompilatorów znacznie lepiej niż programista ocenia, czy dana funkcja może być zdefiniowana jako inline. Działanie kompilatora zależy od ustawień wydajności.

Pytanie: Czy mogę podać wartości domyślne dla wszystkich parametrów funkcji?

Odpowiedź: Tak, to jest możliwe i zalecane, gdy ma sens.

Pytanie: Mam dwie funkcje o nazwie `Area()`. Jedna z nich pobiera promień, natomiast druga — wysokość. Chcę, aby jedna z nich zwracała wartość `float`, z kolei wartością zwrotną drugiej powinna być wartość `double`. Czy takie rozwiązanie będzie działało prawidłowo?

Odpowiedź: Przeciążenie funkcji wymaga, aby funkcje o takiej samej nazwie miały również ten sam typ wartości zwrotnej. W przedstawionym przypadku kompilator wyświetli komunikat błędu, ponieważ ta sama nazwa została użyta dla dwóch funkcji, które powinny mieć inne nazwy.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Jaki jest zakres zmiennych zadeklarowanych w prototypie funkcji?
2. Jaka jest natura wartości przekazywanej do poniższej funkcji?

   ```
   int Func(int &SomeNumber);
   ```

3. Mam funkcję, która wywołuje samą siebie. Jaka jest nazwa tego rodzaju funkcji?
4. Zadeklarowałem dwie funkcje o takich samych nazwach i typach wartości zwrotnej, ale o różnych listach parametrów. Jaka jest nazwa tego rodzaju funkcji?
5. Czy wskaźnik stosu wskazuje miejsce na górze, w środku, czy na dole stosu?

Ćwiczenia

1. Utwórz przeciążone funkcje obliczające objętość kuli i walca. Poniżej przedstawiono wzory, które wykorzystasz w funkcjach:

   ```
   Objętość kuli = (4 * Pi * promień * promień * promień) / 3
   Objętość walca = Pi * promień * promień * wysokość
   ```

2. Utwórz funkcję akceptującą dane wejściowe w postaci tablicy wartości `double`.
3. **Łowcy błędów:** Co jest nie tak w poniższym fragmencie kodu?

   ```
   #include <iostream>
   using namespace std;

   const double Pi = 3.1416;

   void Area(double Radius, double Result)
   {
      Result = Pi * Radius * Radius;
   }
   ```

```
int main()
{
   cout << "Podaj promień: ";
   double Radius = 0;
   cin >> Radius;

   double AreaFetched = 0;
   Area(Radius, AreaFetched);

   cout << "Pole wynosi: " << AreaFetched << endl;
   return 0;
}
```

4. **Łowcy błędów:** Co jest nie tak z poniższą deklaracją funkcji?

   ```
   double Area(double Pi = 3.14, double Radius);
   ```

5. Utwórz funkcję o typie zwrotnym `void`, która będzie pomagała wywołującemu w obliczeniu pola i obwodu okręgu po podaniu promienia.

Lekcja 8

Wskaźniki i referencje

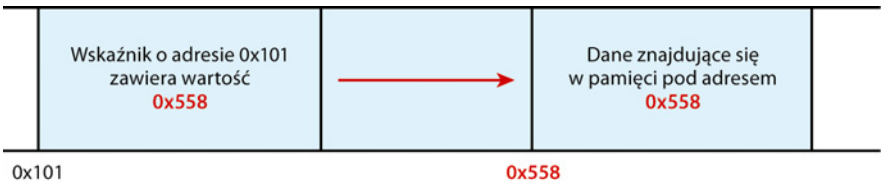
Jedną z największych zalet języka C++ jest możliwość tworzenia aplikacji wysokiego poziomu, które będą działały niezależnie od komputera. Język C++ pozwala na dostrojenie wydajności aplikacji na poziomie bitów i bajtów. Zrozumienie sposobu działania wskaźników i referencji to ważny krok na drodze do zdobycia umiejętności tworzenia programów efektywnie wykorzystujących dostępne zasoby systemu.

Z tej lekcji dowiesz się:

- **czym są wskaźniki,**
- **czym jest pula wolnej pamięci,**
- **jak używać operatorów new i delete w celu alokacji i zwalniania pamięci,**
- **jak tworzyć stabilne aplikacje za pomocą wskaźników i alokacji dynamicznej,**
- **czym są referencje,**
- **na czym polegają różnice pomiędzy wskaźnikami i referencjami,**
- **kiedy używać wskaźników, a kiedy referencji.**

Czym jest wskaźnik?

Ujmując rzecz najprościej, *wskaźnik* to zmienna przechowująca adres w pamięci. Podobnie jak zmienna typu `int` jest przeznaczona do przechowywania wartości w postaci liczby całkowitej, wskaźnik przechowuje adres w pamięci, co pokazano na rysunku 8.1.



RYSUNEK 8.1. Wizualne przedstawienie wskaźnika

Wskaźnik jest więc zmienną i podobnie jak wszystkie zmienne zabiera pewną ilość pamięci (w przypadku pokazanym na rysunku 8.1 będzie to pamięć pod adresem 0x101). Cechą charakterystyczną wskaźnika jest fakt, że przechowywana w nim wartość (tutaj 0x558) jest interpretowana jako adres w pamięci. Dlatego też wskaźnik jest zmienną specjalną *wskazującą* położenie w pamięci.

Deklaracja wskaźnika

Wskaźnik jest zmienną, więc musi być zadeklarowany. Zwykle deklarujesz wskaźnik, aby prowadzić do wartości określonego typu (np. `int`). Oznacza to, że adres przechowywany we wskaźniku prowadzi do miejsca w pamięci, w którym znajduje się liczba całkowita. Istnieje również możliwość określenia wskaźnika w taki sposób, aby wskazywał blok pamięci (to wskaźnik `void`).

Wskaźnik będący zmienną musi być zadeklarowany w sposób podobny do sposobu deklarowania pozostałych zmiennych:

```
TypWskaźnika * NazwaZmiennejWskaźnika;
```

Tak jak w większości zmiennych, jeśli nie zainicjalizujesz wskaźnika, będzie zawierał losowo wybraną wartość. Ponieważ nie chcesz, aby program uzyskał dostęp do losowo wybranego położenia w pamięci, musisz zainicjalizować wskaźnik z wartością `null`. To `null` jest wartością, którą można wykorzystać w operacji sprawdzania i która jednocześnie nie jest adresem w pamięci:

```
TypWskaźnika * NazwaZmiennejWskaźnika = NULL; // Inicjalizacja wartości wskaźnika.
```

Deklaracja wskaźnika typu liczby całkowitej będzie więc miała postać:

```
int *pInteger = NULL;
```

Ostrzeżenie

Wskaźnik, podobnie jak wszystkie poznane dotąd typy danych, zawiera przypadkową wartość dopóty, dopóki nie zostanie zainicjalizowany. Taka przypadkowa wartość jest szczególnie niebezpieczna w przypadku wskaźników, ponieważ wskazuje pewien adres w pamięci. Niezainicjalizowany wskaźnik może spowodować, że program uzyska dostęp do niepoprawnego położenia w pamięci i ulegnie awarii.

Określenie adresu zmiennej przy użyciu operatora referencji (&)

Zmienne to narzędzia dostarczane przez język, aby umożliwić pracę z danymi w pamięci. Koncepcja ta została szczegółowo wyjaśniona w lekcji 3., zatytułowanej „Zmienne i stałe". Wskaźniki również są zmiennymi, ale mają specjalny typ używany jedynie w celu przechowywania adresu w pamięci.

Jeżeli `VarName` to nazwa zmiennej, wtedy `&VarName` podaje adres w pamięci, gdzie przechowywana jest wartość wymienionej zmiennej.

Jeśli zatem zadeklarowałeś liczbę całkowitą, używając przedstawionej poniżej doskonale znanej składni:

```
int Age = 30;
```

wówczas &Age będzie adresem w pamięci przechowującym wartość (30) zmiennej. W listingu 8.1 przedstawiono koncepcję adresu w pamięci dla zmiennej w postaci liczby całkowitej, który jest używany do przechowywania wartości zmiennej.

Listing 8.1. Określenie adresu liczb typu int i double

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    int Age = 30;
6:    const double Pi = 3.1416;
7:
8:    // Użycie operatora & do wyszukania adresu w pamięci.
9:    cout << "Liczba całkowita Age jest pod adresem: 0x" << hex << &Age <<
      ↪endl;
```

```
10:    cout << "Liczba Pi typu double jest pod adresem: 0x" << hex << &Pi <<
       ➥endl;
11:
12:    return 0;
13: }
```

Wynik ▼

```
Liczba całkowita Age jest pod adresem: 0x0045FE00
Liczba Pi typu double jest pod adresem: 0x0045FDF8
```

Analiza ▼

Zwróć uwagę na użycie operatora referencji (&) w wierszach 9. i 10. w celu ustalenia adresów zmiennej Age i stałej Pi. Przedrostek 0x został dodany, aby zachować konwencję stosowaną podczas wyświetlania liczb szesnastkowych.

Uwaga

Wiesz już, że ilość pamięci zużywana przez zmienną zależy od jej typu. W listingu 3.4 przedstawionym w lekcji 3. użyto operatora `sizeof()` w celu pokazania, że wielkość liczby całkowitej to 4 bajty (w systemie autora i używanym przez niego kompilatorze). Opierając się na przedstawionych powyżej danych wyjściowych wskazujących na położenie zmiennej Age pod adresem 0x0045FE08 i znając wielkość liczby całkowitej (4 bajty), wiemy, że cztery bajty w położeniu od 0x0045FE00 do 0x0045FE04 należą do liczby całkowitej Age.

Uwaga

Operator referencji (&) jest nazywany również operatorem adresu.

Użycie wskaźników do przechowywania adresów

Dowiedziałeś się już, jak deklarować wskaźniki oraz jak ustalić adres zmiennej. Wiesz także, że wskaźniki są zmiennymi używanymi do przechowywania adresów w pamięci. Najwyższy czas skonsolidować wiedzę i wykorzystać wskaźniki do przechowywania adresów pobranych przy użyciu operatora referencji (&).

Przyjmujemy założenie, że deklaracja zmiennej jest przeprowadzana w znany sposób:

```
// Deklaracja zmiennej.
Typ NazwaZmiennej = WartośćPoczątkowa;
```

W celu przechowywania adresu zmiennej we wskaźniku konieczne jest zadeklarowanie wskaźnika o takim samym typie jak zmienna (Typ) oraz zainicjalizowanie wskaźnika przy użyciu operatora referencji (&) wraz z adresem w pamięci, gdzie znajduje się wartość zmiennej:

```
// Zadeklarowanie wskaźnika takiego samego typu i zainicjalizowanie go z adresem zmiennej
// w pamięci.
Typ* Wskaźnik = &Zmienna;
```

Dlatego też w przypadku zadeklarowania zmiennej w postaci liczby całkowitej i przy użyciu doskonale znanej składni:

```
int Age = 30;
```

możesz zadeklarować wskaźnik typu `int` zawierający rzeczywisty adres w pamięci, gdzie przechowywana jest wartość zmiennej `Age`:

```
int* pInteger = &Age; // Wskaźnik do liczby całkowitej Age.
```

W listingu 8.2 możesz zobaczyć, jak wskaźnik jest używany do przechowywania adresu uzyskanego za pomocą operatora referencji (&).

Listing 8.2. Przykład deklaracji i inicjalizacji wskaźnika

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Age = 30;
 6:    int* pInteger = &Age;   // Wskaźnik typu int, zainicjalizowany z wartością &Age.
 7:
 8:    // Wyświetlenie wartości wskaźnika.
 9:    cout << "Liczba całkowita Age jest pod adresem: 0x" << hex <<
       ↪pInteger << endl;
10:
11:    return 0;
12: }
```

Wynik ▼

```
Liczba całkowita Age jest pod adresem: 0x0045FE00
```

Analiza ▼

Dane wyjściowe powyższego listingu są w zasadzie takie same jak poprzedniego, w obu przypadkach został wyświetlony ten sam komunikat wraz z adresem

w pamięci, gdzie przechowywana jest wartość zmiennej Age. Różnica w stosunku do poprzedniego programu polega na tym, że adres jest najpierw przypisywany wskaźnikowi (wiersz 6.), a następnie wartość wskaźnika (teraz już adres) zostaje wyświetlona przy użyciu polecenia cout (wiersz 9.).

Uwaga

> Dane wyjściowe, które otrzymasz, mogą się różnić od przedstawionych w książce. Adres zmiennej może być inny po każdym uruchomieniu programu w tym samym komputerze.

Skoro już wiesz, jak przechowywać adres w zmiennej wskaźnika, bardzo łatwo możesz wyobrazić sobie, że tej samej zmiennej wskaźnika można przypisać inny adres w pamięci. W ten sposób wskaźnik będzie prowadził do innej wartości, co przedstawiono w listingu 8.3.

Listing 8.3. Przypisanie wskaźnikowi adresu innej zmiennej

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Age = 30;
 6:
 7:    int* pInteger = &Age;
 8:    cout << "pInteger prowadzi teraz do zmiennej Age" << endl;
 9:
10:    // Wyświetlenie wartości wskaźnika.
11:    cout << "pInteger = 0x" << hex << pInteger << endl;
12:
13:    int DogsAge = 9;
14:    pInteger = &DogsAge;
15:    cout << "pInteger prowadzi teraz do zmiennej DogsAge" << endl;
16:
17:    cout << "pInteger = 0x" << hex << pInteger << endl;
18:
19:    return 0;
20: }
```

Wynik ▼

```
pInteger prowadzi teraz do zmiennej Age
pInteger = 0x002EFB34
pInteger prowadzi teraz do zmiennej DogsAge
pInteger = 0x002EFB1C
```

Analiza ▼

Program pokazuje, że wskaźnik (pInteger) prowadzący do jednej liczby całkowitej może prowadzić także do zupełnie innej liczby całkowitej. W wierszu 7. nastąpiła jego inicjalizacja wraz z wartością &Age, a więc zawierał adres zmiennej Age. W wierszu 14. temu samemu wskaźnikowi przypisano wartość &DogsAge, co oznacza, że prowadzi do innego położenia w pamięci, które zawiera wartość zmiennej DogsAge. Dane wyjściowe pokazują, że wartość wskaźnika (tzn. adres w pamięci) uległa zmianie. Najpierw prowadziła do miejsca w pamięci (0x002EFB34) przechowującego liczbę całkowitą Age, a później do innego miejsca (0x002EFB1C) przechowującego liczbę całkowitą DogsAge.

Uzyskanie dostępu do danych przy użyciu operatora dereferencji (*)

Masz wskaźnik do danych zawierający poprawny adres. W jaki sposób można uzyskać dostęp do wspomnianego położenia, tzn. jak pobrać lub zmienić dane we wskazanym położeniu w pamięci? Odpowiedzią jest użycie operatora dereferencji (*) nazywanego także operatorem wyłuskania. Jeżeli masz prawidłowy wskaźnik pData, wtedy użycie *pData pozwala na uzyskanie dostępu do wartości przechowywanej pod adresem wskazywanym przez wskaźnik. Praktyczne zastosowanie operatora dereferencji przedstawiono w listingu 8.4.

Listing 8.4. Przykład użycia operatora dereferencji (*) w celu uzyskania dostępu do wartości wskazywanej przez wskaźnik

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Age = 30;
 6:    int DogsAge = 9;
 7:
 8:    cout << "Liczba całkowita Age = " << Age << endl;
 9:    cout << "Liczba całkowita DogsAge = " << DogsAge << endl;
10:
11:    int* pInteger = &Age;
12:    cout << "pInteger prowadzi do zmiennej Age" << endl;
13:
14:    // Wyświetlenie wartości wskaźnika.
```

```
15:    cout << "pInteger = 0x" << hex << pInteger << endl;
16:
17:    // Wyświetlenie wartości we wskazanym położeniu.
18:    cout << "*pInteger = " << dec << *pInteger << endl;
19:
20:    pInteger = &DogsAge;
21:    cout << "pInteger prowadzi teraz do zmiennej DogsAge" << endl;
22:
23:    cout << "pInteger = 0x" << hex << pInteger << endl;
24:    cout << "*pInteger = " << dec << *pInteger << endl;
25:
26:    return 0;
27: }
```

Wynik ▼

```
Liczba całkowita Age = 30
Liczba całkowita DogsAge = 9
pInteger prowadzi do zmiennej Age
pInteger = 0x0025F788
*pInteger = 30
pInteger prowadzi teraz do zmiennej DogsAge
pInteger = 0x0025F77C
*pInteger = 9
```

Analiza ▼

Poza zmianą adresu przechowywanego przez wskaźnik, co również miało miejsce w poprzednim przykładzie (listing 8.3), w powyższym listingu użyto operatora dereferencji (*) wraz ze zmienną wskaźnika `pInteger` w celu wyświetlenia różnych wartości przechowywanych w dwóch adresach w pamięci. Zwróć uwagę na wiersze 18. i 24.

W obu tych wierszach dostęp do liczby całkowitej wskazywanej przez wskaźnik `pInteger` następuje za pomocą operatora dereferencji. Po zmianie adresu we wskaźniku `pInteger` (patrz wiersz 20.) ten sam wskaźnik prowadzi teraz do wartości zmiennej `DogsAge`, co powoduje wyświetlenie w danych wyjściowych liczby 9.

Podczas użycia operatora dereferencji adres przechowywany we wskaźniku jest wykorzystywany przez aplikację do pobrania czterech bajtów z pamięci zarezerwowanej dla danej liczby całkowitej (ponieważ to wskaźnik do liczby całkowitej o wielkości 4 bajtów, co potwierdza wynik operacji `sizeof(int)`). Dlatego też zapewnienie poprawności adresu przechowywanego przez wskaźnik

ma znaczenie krytyczne. Przez inicjalizację wskaźnika z wartością &Age (wiersz 11.) gwarantujemy, że wskaźnik zawiera prawidłowy adres w pamięci. Jeżeli nie zainicjalizujesz wskaźnika, może on zawierać dowolną wartość, która istniała w danym położeniu w pamięci w chwili rezerwacji tej pamięci dla wskaźnika. Odwołanie do takiego wskaźnika najczęściej oznacza wystąpienie błędu *Access Violation*, czyli próbę uzyskania przez aplikację dostępu do adresu w pamięci, do którego nie ma uprawnień.

Uwaga

Operator dereferencji jest nazywany również operatorem dostępu pośredniego.

W poprzednim przykładzie wskaźnik był używany w celu odczytu (pobrania) wartości ze wskazanego miejsca w pamięci. W listingu 8.5 pokazano, co się stanie, jeśli wskaźnik `*pInteger` zostanie użyty jako l-wartość, tzn. kod przypisze go zmiennej, zamiast uzyskać do niego dostęp.

Listing 8.5. Operowanie danymi za pomocą wskaźnika i operatora dereferencji

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int DogsAge = 30;
 6:    cout << "Zainicjalizowana zmienna DogsAge = " << DogsAge << endl;
 7:
 8:    int* pAge = &DogsAge;
 9:    cout << "Wskaźnik pAge prowadzi do zmiennej DogsAge" << endl;
10:
11:    cout << "Podaj wiek psa: ";
12:
13:    // Dane wejściowe zostają umieszczone w pamięci wskazywanej przez wskaźnik pAge.
14:    cin >> *pAge;
15:
16:    // Wyświetlenie adresu, gdzie przechowywana jest wartość.
17:    cout << "Dane wejściowe wskazywane przez pAge są pod adresem 0x" <<
       ↪hex << pAge << endl;
18:
19:    cout << "Liczba całkowita DogsAge = " << dec << DogsAge << endl;
20:
21:    return 0;
22: }
```

Wynik ▼

```
Zainicjalizowana zmienna DogsAge = 30
Wskaźnik pAge prowadzi do zmiennej DogsAge
Podaj wiek psa: 10
Dane wejściowe wskazywane przez pAge są pod adresem 0x0025FA18
Liczba całkowita DogsAge = 10
```

Analiza ▼

W omawianym listingu kluczowy krok to wiersz 14., w którym liczba całkowita podana przez użytkownika zostaje zapisana w położeniu w pamięci wskazywanym przez wskaźnik pAge. Zauważ, że pomimo umieszczenia danych wejściowych w położeniu wskazywanym przez pAge, polecenie w wierszu 19. wyświetlające wartość zmiennej DogsAge pokazuje wartość wskazaną przez wskaźnik. Wynika to z faktu, że wskaźnik pAge prowadzi do zmiennej DogsAge, na co wskazuje polecenie inicjalizujące w wierszu 8. Wszelkie zmiany w położeniach w pamięci przechowujących zmienną DogsAge i wskazywanych przez wskaźnik pAge są odzwierciedlane w obu elementach.

Ile pamięci zabiera wskaźnik?

Dowiedziałeś się, że wskaźnik to po prostu rodzaj zmiennej przechowującej adres położenia w pamięci. Dlatego też, niezależnie od wskazywanego typu, zawartością wskaźnika jest adres, czyli liczba. Wielkość adresu to liczba bajtów wymaganych do przechowywania liczby w postaci stałej w danym systemie. Wynik działania operatora `sizeof()` względem wskaźnika będzie zależał od kompilatora i systemu operacyjnego, dla którego program został skompilowany. Natomiast nie zależy od natury danych, do których prowadzi wskaźnik, co potwierdza program przedstawiony w listingu 8.6.

Listing 8.6. Potwierdzenie, że wskaźniki prowadzące do różnych typów danych mają tę samą wielkość

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    int Age = 30;
6:    double Pi = 3.1416;
7:    char SayYes = 'y';
```

```
 8:
 9:     // Inicjalizacja wskaźników wraz z adresami zmiennych.
10:    int* pInt = &Age;
11:    double* pDouble = &Pi;
12:    char* pChar = &SayYes;
13:
14:    cout << "sizeof podstawowych typów -" << endl;
15:    cout << "sizeof(int) = " << sizeof(int) << endl;
16:    cout << "sizeof(double) = " << sizeof(double) << endl;
17:    cout << "sizeof(char) = " << sizeof(char) << endl;
18:
19:    cout << "sizeof wskaźników do podstawowych typów -" << endl;
20:    cout << "sizeof(pInt) = " << sizeof(pInt) << endl;
21:    cout << "sizeof(pDouble) = " << sizeof(pDouble) << endl;
22:    cout << "sizeof(pChar) = " << sizeof(pChar) << endl;
23:
24:    return 0;
25: }
```

Wynik ▼

```
sizeof podstawowych typów -
sizeof(int) = 4
sizeof(double) = 8
sizeof(char) = 1
sizeof wskaźników do podstawowych typów -
sizeof(pInt) = 4
sizeof(pDouble) = 4
sizeof(pChar) = 4
```

Analiza ▼

Dane wyjściowe jasno pokazują, że nawet gdy wartość `sizeof(char)` wynosi 1 bajt, a `sizeof(double)` wynosi 8 bajtów, to wielkość wskaźnika (`sizeof(pointer)`) jest stała i wynosi 4 bajty. Po prostu ilość pamięci wymagana do przechowywania adresów w pamięci jest taka sama, niezależnie od tego, czy wskaźnik prowadzi do 1, czy do 8 bajtów.

Uwaga

Dane wyjściowe listingu 8.6 pokazują, że wielkość wskaźnika wynosi 4 bajty. Jednak w Twoim systemie operacyjnym wielkość ta może być inna. Przedstawione w książce dane wyjściowe zostały wygenerowane przez kod skompilowany przez 32-bitowy kompilator. Jeżeli używasz 64-bitowego kompilatora i uruchomisz program w 64-bitowym systemie operacyjnym, możesz zobaczyć, że wielkość zmiennej wskaźnika wynosi 64 bity, czyli 8 bajtów.

Dynamiczna alokacja pamięci

Kiedy tworzysz program zawierający deklarację tablicy poniższego typu:

```
int Numbers[100]; // Tablica statyczna stu liczb całkowitych.
```

program boryka się z dwoma problemami. Oto one.

1. Ograniczasz pojemność programu, który będzie mógł przechowywać maksymalnie sto liczb całkowitych.
2. Zmniejszasz wydajność systemu, w przypadku gdy tablica zarezerwowana dla stu liczb całkowitych będzie przechowywała tylko jedną.

Wymienione problemy istnieją z powodu alokacji pamięci dla tablicy — dla podanej wcześniej deklaracji tablicy kompilator rezerwuje statyczną ilość pamięci.

Aby utworzyć aplikację optymalnie wykorzystującą zasoby pamięci na podstawie potrzeb i działań użytkownika, należy zastosować dynamiczną alokację pamięci. W ten sposób będzie można zaalokować większą ilość pamięci, gdy będzie potrzebna, oraz zwalniać pamięć, kiedy nie będziemy z niej korzystać. Język C++ oferuje dwa operatory — `new` i `delete` — pomagające w lepszym zarządzaniu pamięcią przez aplikację. Wskaźniki będące zmiennymi używanymi do przechowywania adresów w pamięci odgrywają krytyczną rolę w efektywnym, dynamicznym zarządzaniu pamięcią.

Użycie operatorów new i delete w celu dynamicznej alokacji i zwalniania pamięci

Operator `new` służy do alokacji nowych bloków pamięci. Najczęściej używana forma operatora `new` powoduje zwrot wskaźnika do żądanej pamięci, o ile operacja zakończyła się powodzeniem, w przeciwnym razie następuje zgłoszenie wyjątku. Podczas użycia operatora `new` konieczne jest wskazanie typu danych, dla których następuje alokacja pamięci:

```
Typ* Wskaźnik = new Typ; // Żądanie pamięci dla jednego elementu.
```

Istnieje również możliwość wskazania liczby elementów, dla których ma być zaalokowana pamięć (w przypadku alokacji pamięci dla więcej niż tylko jednego elementu):

```
Typ* Wskaźnik = new Typ[LiczbaElementów]; // Żądanie pamięci dla podanej liczby
                                          // elementów.
```

Dlatego też, jeśli musisz zaalokować pamięć dla liczb całkowitych, możesz skorzystać z poniższych składni:

```
int* pNumber = new int;  // Pobranie wskaźnika do liczby całkowitej.
int* pNumbers = new int[10];  // Pobranie wskaźnika do bloku dziesięciu liczb całkowitych.
```

> **Uwaga** Zauważ, że new oznacza żądanie pamięci. Nie ma gwarancji, że wykonanie polecenia żądania pamięci zakończy się powodzeniem — zależy to od stanu systemu oraz dostępności zasobów pamięci.

Dla każdej przeprowadzonej operacji alokacji pamięci za pomocą operatora new konieczne jest przeprowadzenie operacji zwolnienia tej pamięci przy użyciu operatora delete:

```
Typ* Wskaźnik = new Typ;
delete Wskaźnik;  // Zwolnienie pamięci zaalokowanej wcześniej dla jednego egzemplarza wskazanego typu.
```

Ta sama reguła obowiązuje w przypadku żądania alokacji pamięci dla wielu elementów:

```
Typ* Wskaźnik = new Typ[LiczbaElementów];
delete[] Wskaźnik; // Zwolnienie zaalokowanego powyżej bloku pamięci.
```

> **Uwaga** Zwróć uwagę na użycie delete[] w czasie alokacji bloku pamięci przy użyciu operatora new[...] i delete po alokacji tylko jednego elementu za pomocą operatora new.

Jeżeli nie zwolnisz zaalokowanej wcześniej pamięci, której już nie używasz, pozostanie zarezerwowana dla aplikacji. To z kolei zmniejszy ilość pamięci systemowej dostępnej dla innych programów i prawdopodobnie zmniejszy szybkość działania Twojej aplikacji. Taka sytuacja nosi nazwę *wycieku pamięci*; należy jej unikać za wszelką cenę.

Kod przedstawiony w listingu 8.7 pokazuje dynamiczną alokację i zwalnianie pamięci.

Listing 8.7. Wykorzystanie operatora dereferencji w celu uzyskania dostępu do pamięci zaalokowanej przez operator new oraz jej zwolnienie za pomocą operatora delete

```
 0: #include <iostream>
 1: using namespace std;
 2:
```

```
 3: int main()
 4: {
 5:     // Żądanie alokacji pamięci dla elementu typu int.
 6:     int* pAge = new int;
 7:
 8:     // Użycie zaalokowanej pamięci do przechowywania liczby.
 9:     cout << "Podaj wiek psa: ";
10:     cin >> *pAge;
11:
12:     // Użycie operatora dereferencji * w celu uzyskania dostępu do przechowywanej wartości.
13:     cout << "Wartość zmiennej Age wynosi " << *pAge << " i jest
        ➥przechowywana pod adresem 0x" << hex << pAge << endl;
14:
15:     delete pAge; // Zwolnienie pamięci.
16:
17:     return 0;
18: }
```

Wynik ▼

```
Podaj wiek psa: 9
Wartość zmiennej Age wynosi 9 i jest przechowywana pod adresem 0x00338120
```

Analiza ▼

W wierszu 6. pokazano przykład użycia operatora `new` w celu alokacji pamięci dla liczby całkowitej. W zarezerwowanej pamięci będą przechowywane podane przez użytkownika dane wejściowe, czyli wiek psa. Zauważ, że operator `new` zwraca wskaźnik i to jest powodem przypisania mu wartości. Wiek podany przez użytkownika jest przechowywany w nowo zaalokowanej pamięci, odbywa się to w wierszu 10. za pomocą polecenia `cin` i operatora dereferencji (*). W wierszu 13. kod wyświetla przechowywaną wartość (ponownie posługując się operatorem dereferencji), a także adres w pamięci, pod którym znajduje się przechowywana wartość. Zwróć uwagę, że adres znajdujący się we wskaźniku `pAge` (wiersz 13.) to ten sam adres, który został zwrócony przez operator `new` w wierszu 6. i od tamtej chwili nie uległ zmianie.

Ostrzeżenie
Ostrzeżenie

Operator `delete` nie może być wywoływany względem dowolnego adresu znajdującego się we wskaźniku, to musi być adres zwrócony wcześniej przez operator `new` i tylko ten, który nie był wcześniej zwolniony przez inne wywołanie `delete`.

Mimo iż wskaźniki przedstawione w listingu 8.6 zawierają poprawne adresy, jednak nie powinny być zwalniane za pomocą operatora `delete`, ponieważ wspomniane adresy nie zostały otrzymane na skutek wywołania operatora `new`.

Pamiętaj, że po użyciu `new[...]` w celu alokacji bloku pamięci dla wielu elementów zwolnienie pamięci musi nastąpić przez wywołanie `delete[]`, jak to przedstawiono w listingu 8.8.

Listing 8.8. Alokacja pamięci przy użyciu operatora new[...] i jej zwolnienie za pomocą operatora delete[]

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Podaj imię: ";
 7:    string Name;
 8:    cin >> Name;
 9:
10:    // Dodanie 1 do rezerwowanej pamięci (na znak null).
11:    int CharsToAllocate = Name.length() + 1;
12:
13:    // Żądanie pamięci potrzebnej do przechowywania kopii danych wejściowych.
14:    char* CopyOfName = new char [CharsToAllocate];
15:
16:    // Funkcja strcpy() kopiuje dane z ciągu tekstowego zakończonego znakiem null.
17:    strcpy(CopyOfName, Name.c_str());
18:
19:    // Wyświetlenie skopiowanego ciągu tekstowego.
20:    cout << "Dynamicznie zaalokowany bufor zawiera: " << CopyOfName <<
       ↪endl;
21:
22:    // Po zakończeniu pracy z buforem należy go usunąć.
23:    delete[] CopyOfName;
24:
25:    return 0;
26: }
```

Wynik ▼

```
Podaj imię: Robert
Dynamicznie zaalokowany bufor zawiera: Robert
```

Analiza ▼

Najbardziej interesujące nas wiersze w powyższym programie to te, które zawierają operatory `new` i `delete[]`, czyli (odpowiednio) wiersze 11. i 23. Różnica w stosunku do poprzedniego przykładu (patrz listing 8.7) polega na alokacji bloku pamięci wystarczającego do przechowywania wielu elementów — kod w listingu 8.7 alokował pamięć tylko dla jednego elementu. Alokacji pamięci dla tablicy elementów musi odpowiadać użycie operatora `delete[]` w celu zwolnienia pamięci po zakończeniu pracy z tablicą. Liczba znaków, dla których trzeba zaalokować pamięć, jest obliczana w wierszu 11. Liczba ta jest o jeden większa od liczby znaków wprowadzonych przez użytkownika, ponieważ trzeba uwzględnić znak `null`, bardzo ważny w ciągach tekstowych typu C. Konieczność umieszczenia znaku `null` na końcu ciągu tekstowego została wyjaśniona w lekcji 4., zatytułowanej „Tablice i ciągi tekstowe". Rzeczywista operacja kopiowania odbywa się w wierszu 17., jest przeprowadzana przez funkcję `strcpy()` używającą z kolei funkcji `c_str()` dostarczanej przez klasę `std::string`. Znaki są kopiowane do bufora typu `char` o nazwie `CopyOfName`.

Uwaga

> Operatory `new` i `delete` alokują pamięć pochodzącą z puli wolnej pamięci. Pula wolnej pamięci to rodzaj abstrakcji w postaci puli pamięci, którą aplikacja może zaalokować (tzn. zarezerwować) oraz dealokować (tzn. zwalniać).

Efektywne użycie operatorów inkrementacji (++) i dekrementacji (--) na wskaźnikach

Wskaźnik zawiera adres w pamięci. Przykładowo w przedstawionym wcześniej listingu 8.3 wskaźnik prowadzący do liczby całkowitej zawiera wartość `0x002EFB34`, czyli adres w pamięci, gdzie jest przechowywana wspomniana liczba całkowita. Sama liczba ma wielkość czterech bajtów, a tym samym zabiera cztery miejsca w pamięci od `0x002EFB34` do `0x002EFB37`. Inkrementacja wskaźnika za pomocą operatora ++ *nie spowoduje*, że wskaźnik będzie prowadził do `0x002EFB35`. Takie działanie, które ma na celu wskazanie środka liczby całkowitej, jest dosłownie bezcelowe.

Operacja inkrementacji lub dekrementacji wskaźnika jest interpretowana przez kompilator jako potrzeba wskazania kolejnej wartości w bloku pamięci, przy przyjęciu założenia, że będzie ona takiego samego typu, a *nie następnym bajtem* (o ile wartość nie ma długości jednego bajta, np. `char`).

Dlatego też inkrementacja wskaźnika, takiego jak `pInteger`, użytego w listingu 8.3 powoduje inkrementację o cztery bajty, co odpowiada wynikowi wywołania `sizeof(int)`. Użycie operatora ++ względem wskaźnika oznacza poinformowanie kompilatora, że wskaźnik ma prowadzić do kolejnej liczby całkowitej. Po operacji inkrementacji wskaźnik będzie prowadził do adresu `0x002EFB38`. Podobnie dodanie wartości 2 do wskaźnika spowoduje jego przesunięcie o dwie liczby całkowite do przodu, czyli o osiem bajtów. Dalej w tej lekcji zobaczysz korelację pomiędzy tym zachowaniem wskaźników a indeksami używanymi w tablicach.

Dekrementacja wskaźników przy użyciu operatora `--` daje taki sam efekt, tzn. wartość adresu zawartego we wskaźniku zostaje zmniejszona o wynik operacji `sizeof` przeprowadzonej na typie danych wskazywanych przez wskaźnik.

Co się stanie po inkrementacji lub dekrementacji wskaźnika?

Adres znajdujący się we wskaźniku jest inkrementowany lub dekrementowany o wartość zwrotną działania operatora `sizeof` względem typu danych wskazywanego przez wskaźnik (to niekoniecznie będzie bajt). Dzięki temu kompilator gwarantuje, że wskaźnik nigdy nie będzie prowadzić do środka lub na koniec danych umieszczonych pod danym adresem w pamięci — wskaźnik zawsze prowadzi na początek danych.

Jeżeli wskaźnik zostałby zadeklarowany w postaci:

```
Typ* pTyp = Adres;
```

wówczas `++pTyp` oznacza, że `pTyp` zawiera (a więc wskazuje) wartość `Adres + sizeof(Typ)`.

W kodzie przedstawionym w listingu 8.9 znajdziesz wyjaśnienie efektu inkrementacji wskaźnika i dodawania do niego wartości przesunięcia.

Listing 8.9. Alokacja dynamiczna w zależności od potrzeb, inkrementacja wskaźników za pomocą wartości przesunięcia i operatora ++

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:   cout << "Ile liczb całkowitych chcesz podać? ";
6:   int InputNums = 0;
7:   cin >> InputNums;
8:
9:   int* pNumbers = new int [InputNums]; // Alokacja żądanych liczb całkowitych.
```

```
10:   int* pCopy = pNumbers;
11:
12:   cout<<"Udało się zaalokować pamięć dla "<<InputNums<< " liczb
      ↪całkowitych"<<endl;
13:   for(int Index = 0; Index < InputNums; ++Index)
14:   {
15:      cout << "Podaj liczbę "<< Index << ": ";
16:      cin >> *(pNumbers + Index);
17:   }
18:
19:   cout << "Wyświetlenie wszystkich podanych liczb: " << endl;
20:   for(int Index = 0; Index < InputNums; ++Index)
21:      cout << *(pCopy++) << " ";
22:
23:    cout << endl;
24:
25:    // Po zakończeniu pracy ze wskaźnikiem trzeba zwolnić pamięć.
26:    delete[] pNumbers;
27:
28:    return 0;
29: }
```

Wynik ▼

```
Ile liczb całkowitych chcesz podać? 2
Udało się zaalokować pamięć dla 2 liczb całkowitych
Podaj liczbę 0: 789
Podaj liczbę 1: 575
Wyświetlenie wszystkich podanych liczb:
789 575
```

Następne uruchomienie programu:

```
Ile liczb całkowitych chcesz podać? 5
Udało się zaalokować pamięć dla 5 liczb całkowitych
Podaj liczbę 0: 789
Podaj liczbę 1: 12
Podaj liczbę 2: -65
Podaj liczbę 3: 285
Podaj liczbę 4: -101
Wyświetlenie wszystkich podanych liczb:
789 12 -65 285 -101
```

Analiza ▼

Przed alokacją pamięci w wierszu 9. program prosi użytkownika o podanie ilości liczb całkowitych, które mają być przechowywane w systemie. Zwróć uwagę na wykonanie kopii adresu w wierszu 10. do późniejszego użycia

podczas zwalniania bloku pamięci w wierszu 26. przy użyciu operatora `delete`. W tym programie pokazano zalety użycia wskaźników i dynamicznej alokacji pamięci w porównaniu do zastosowania tablicy statycznej. Omawiana aplikacja zużywa mniejszą ilość pamięci, gdy użytkownik przechowuje małą ilość liczb całkowitych, oraz większą, kiedy trzeba przechować dużą ilość liczb całkowitych. W żadnym z wymienionych przypadków nie są marnowane zasoby systemowe. Z powodu zastosowania dynamicznej alokacji pamięci nie ma żadnej sztucznej górnej granicy ilości liczb całkowitych, które mogą być przechowywane — jedynym ograniczeniem jest dostępność zasobów systemowych. W wierszach od 13. do 17. znajduje się pętla `for` pozwalająca użytkownikowi na wprowadzenie liczb, które następnie są przechowywane w kolejnych miejscach w pamięci przy użyciu wyrażenia zdefiniowanego w wierszu 16. Do wskaźnika została dodana liczona od zera wartość przesunięcia (`Index`). Kompilator tworzy więc program wstawiający wartość podaną przez użytkownika w odpowiednim miejscu w pamięci bez nadpisywania poprzedniej. Innymi słowy, `(pNumber + Index)` to wyrażenie, które zwraca wskaźnik prowadzący do liczby całkowitej w rozpoczynającym się od zera indeksie położenia w pamięci. W takim przypadku wartość 1 indeksu wskazuje drugą liczbę całkowitą. Z kolei operator dereferencji w `*(pNumber + Index)` to wyrażenie używane przez polecenie `cin` w celu uzyskania dostępu do wartości we wspomnianym indeksie rozpoczynającym się od zera. Pętla `for` w wierszach 21. i 22. jest używana do wyświetlania wartości przechowywanych w poprzedniej pętli. Polecenie `for` wykorzystuje wiele inicjalizatorów, tworzy kopię w `pCopy` i inkrementuje tę kopię w wierszu 21., aby wyświetlić wartość.

Powodem utworzenia kopii w wierszu 10. jest fakt, że pętla z wykorzystaniem operatora inkrementacji modyfikuje (++) używany wskaźnik. Początkowy wskaźnik zwrócony przez operator `new` musi być zachowany nietknięty, aby w wierszu 26. można było wywołać odpowiednie polecenie `delete[]` wraz z dokładną wartością podaną przez operator `new`, a nie inną, przypadkową wartością.

Użycie słowa kluczowego const ze wskaźnikami

W lekcji 3. dowiedziałeś się, że deklaracja zmiennej jako `const` powoduje przypisanie tego rodzaju zmiennej wartości na stałe, tzn. wartości, która nie może być zmieniana w trakcie cyklu życiowego zmiennej. Tak więc wartość takiej zmiennej nie może być zmodyfikowana lub użyta jako l-wartość.

Wskaźniki również są zmiennymi, a tym samym słowo kluczowe `const` ma dla nich znaczenie. Jednak wskaźnik to specjalny rodzaj zmiennej zawierającej adres w pamięci i używanej do modyfikacji bloku danych w pamięci. Dlatego też dla wskaźników i stałych mamy do dyspozycji następujące rozwiązania.

- Dane wskazywane przez wskaźnik są stałą i nie mogą być zmodyfikowane, adres zawarty we wskaźniku może ulec zmianie, tzn. wskaźnik może prowadzić do zupełnie innego miejsca w pamięci.

```
int HoursInDay = 24;
const int* pInteger = &HoursInDay; // Nie można użyć pInteger do zmiany.
HoursInDay
int MonthsInYear = 12;
pInteger = &MonthsInYear; // OK!
*pInteger = 13; // Błąd kompilacji: nie można zmienić danych.
int* pAnotherPointerToInt = pInteger; // Błąd kompilacji: stałej nie można
                                      // przypisać niestałej.
```

- Adres zawarty we wskaźniku jest stałą i nie może być zmodyfikowany, ale zmienione mogą być dane, do których prowadzi wskaźnik.

```
int DaysInMonth = 30;
// Wskaźnik pInteger nie może prowadzić do niczego innego.
int* const pDaysInMonth = &DaysInMonth;
*pDaysInMonth = 31; // OK! Wartość może być zmieniona.
int DaysInLunarMonth = 28;
pDaysInMonth = &DaysInLunarMonth; // Błąd kompilacji: nie można zmienić
                                  // adresu!
```

- Zarówno adres znajdujący się we wskaźniku, jak i wartość, do której on prowadzi, są stałymi i nie mogą być zmodyfikowane (to najbardziej restrykcyjny wariant).

```
int HoursInDay = 24;
// Wskaźnik może prowadzić jedynie do HoursInDay.
const int* const pHoursInDay = &HoursInDay;
*pHoursInDay = 25; // Błąd kompilacji: nie można zmienić wartości wskazywanej
                   // przez wskaźnik.
int DaysInMonth = 30;
pHoursInDay = &DaysInMonth; // Błąd kompilacji: nie można zmienić wartości
                            // wskaźnika.
```

Wymienione różne formy `const` są szczególnie użyteczne podczas przekazywania wskaźników funkcjom. Parametry funkcji muszą być zadeklarowane w sposób zapewniający obsługę maksymalnie najwyższego (restrykcyjnego) poziomu zachowania stałości, aby zagwarantować, że funkcja (jeśli nie ma takiej potrzeby) nie będzie modyfikowała wartości,

do której prowadzi wskaźnik. Takie rozwiązanie pomaga w zachowaniu przejrzystości funkcji, zwłaszcza gdy zmiany w funkcji są wprowadzane na przestrzeni dłuższego czasu lub zmieniają się pracujący nad nią programiści.

Przekazywanie wskaźników funkcjom

Wskaźniki to efektywny sposób przekazania funkcji adresu w pamięci, zawierającego wartość, lub zwrócenia wyniku wygenerowanego przez funkcję. Podczas używania wskaźników z funkcjami bardzo ważne staje się zapewnienie, że funkcja wywołująca może modyfikować jedynie te parametry, które chcesz zmienić, a nie inne. Przykładowo funkcja obliczająca pole okręgu na podstawie danego promienia przekazywanego przez wskaźnik nie powinna mieć możliwości modyfikacji wspomnianego promienia. W takiej sytuacji użycie słowa kluczowego `const` we wskaźniku pozwala na efektywną kontrolę tego, co funkcja może modyfikować i czego nie może. Przykład rozwiązania przedstawiono w listingu 8.10.

Listing 8.10. Użycie słowa kluczowego new podczas obliczania pola okręgu, gdy promień i Pi są dostarczane w postaci stałych

```
#include <iostream>
using namespace std;

void CalcArea(const double* const pPi, // Stała wskaźnika do stałej danych.
              const double* const pRadius, // Nic nie można zmienić.
              double* const pArea) // Zmiana wskazywanej wartości, a nie adresu.
{
   // Przed użyciem wskaźników należy je sprawdzić!
   if (pPi && pRadius && pArea)
      *pArea = (*pPi) * (*pRadius) * (*pRadius);
}

int main()
{
   const double Pi = 3.1416;

   cout << "Podaj promień okręgu: ";
   double Radius = 0;
   cin >> Radius;

   double Area = 0;
   CalcArea (&Pi, &Radius, &Area);

   cout << "Pole wynosi = " << Area << endl;
```

```
24:
25:    return 0;
26: }
```

Wynik ▼

```
Podaj promień okręgu: 10.5
Pole wynosi = 346.361
```

Analiza ▼

W wierszach od 3. do 5. pokazano dwie formy `const`, gdzie zarówno wskaźnik `pRadius`, jak i `pPi` zostały dostarczone w postaci „stałego wskaźnika do stałych danych". Tak więc nie ma możliwości modyfikacji adresu znajdującego się we wskaźniku oraz danych, do których ten wskaźnik prowadzi. Wskaźnik `pArea` to parametr przeznaczony do przechowywania danych wyjściowych, wartość wskaźnika (adres) nie może być modyfikowana, ale dane, do których prowadzi, mogą być zmienione. W wierszu 7. widać, jak parametry funkcji w postaci wskaźników są sprawdzane pod kątem poprawności, zanim zostaną użyte. Nie chcesz przecież, aby funkcja obliczyła pole, jeśli wywołujący przypadkowo przekaże wskaźnik `null` jako dowolny z trzech wymienionych parametrów, ponieważ to doprowadzi do awarii aplikacji.

Podobieństwa pomiędzy tablicami i wskaźnikami

Czy nie uważasz, że przykład przedstawiony w listingu 8.9, gdzie wskaźnik był inkrementowany za pomocą rozpoczynającego się od zera indeksu w celu uzyskania dostępu do kolejnej liczby całkowitej w pamięci, ma wiele wspólnego ze sposobem indeksowania tablic? Kiedy deklarujesz tablicę liczb całkowitych przy użyciu poniższego polecenia:

```
int MyNumbers[5];
```

kompilator alokuje ściśle określoną i stałą ilość pamięci do przechowywania pięciu liczb całkowitych i daje wskaźnik prowadzący do pierwszego elementu tablicy. Wskaźnik jest identyfikowany przez nazwę przypisaną zmiennej tablicy. Innymi słowy, `MyNumber` to wskaźnik prowadzący do pierwszego elementu — `MyNumber[0]`. Przedstawioną korelację zaprezentowano w listingu 8.11.

Listing 8.11. Pokazanie, że zmienna w postaci tablicy jest wskaźnikiem do pierwszego elementu

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     // Tablica statyczna pięciu liczb całkowitych.
 6:     int MyNumbers[5];
 7:
 8:     // Tablica zostaje przypisana wskaźnikowi typu int.
 9:     int* pNumbers = MyNumbers;
10:
11:     // Wyświetlenie adresu przechowywanego przez wskaźnik.
12:     cout << "pNumbers = 0x" << hex << pNumbers << endl;
13:
14:     //  Adres pierwszego elementu tablicy.
15:     cout << "&MyNumbers[0] = 0x" << hex << &MyNumbers[0] << endl;
16:
17:     return 0;
18: }
```

Wynik ▼

```
pNumbers = 0x004BFE8C
&MyNumbers[0] = 0x004BFE8C
```

Analiza ▼

W powyższym prostym programie pokazano, że zmienna tablicy może być przypisana wskaźnikowi tego samego typu (wiersz 9.). W zasadzie otrzymujemy potwierdzenie, że tablica jest zbliżona do wskaźnika. W wierszach od 12. do 15. pokazano, że adres przechowywany we wskaźniku jest taki sam jak adres w pamięci dla umieszczenia pierwszego elementu tablicy (czyli o wartości indeksu 0). Ten program jasno pokazuje, że tablica jest wskaźnikiem do pierwszego elementu znajdującego się w tablicy.

Dostęp do drugiego elementu tablicy uzyskasz za pomocą wyrażenia `MyNumbers[1]`, ale możesz również wykorzystać wskaźnik `pNumbers` i składnię `*(pNumbers + 1)`. Trzeci element tablicy statycznej jest dostępny przy użyciu wyrażenia `MyNumbers[2]`, natomiast w tablicy dynamicznej można w tym celu użyć składni `*(pNumbers + 2)`.

Ponieważ zmienne tablic to w zasadzie wskaźniki, powinno być możliwe wykorzystanie operatora dereferencji (*) używanego we wskaźnikach podczas pracy z tablicami. Podobnie powinno być możliwe użycie operatora tablicy ([]) do pracy ze wskaźnikami, co przedstawiono w listingu 8.12.

Listing 8.12. Uzyskanie dostępu do elementów tablicy za pomocą operatora dereferencji (*) i użycie operatora tablicy ([]) wraz ze wskaźnikiem

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int ARRAY_LEN = 5;
 6:
 7:    // Zainicjalizowana tablica statyczna pięciu liczb całkowitych.
 8:    int MyNumbers[ARRAY_LEN] = {24, -1, 365, -999, 2011};
 9:
10:    // Wskaźnik zainicjalizowany z pierwszym elementem tablicy.
11:    int* pNumbers = MyNumbers;
12:
13:    cout << "Wyświetlenie tablicy za pomocą składni wskaźnika,
       ↪operatora*" << endl;
14:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
15:       cout << "Element " << Index << " = " << *(MyNumbers + Index) <<
          ↪endl;
16:
17:    cout << "Wyświetlenie tablicy za pomocą wskaźnika wraz ze składnią
       ↪tablicy, operator[]" << endl;
18:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
19:       cout << "Element " << Index << " = " << pNumbers[Index] << endl;
20:
21:    return 0;
22: }
```

Wynik ▼

```
Wyświetlenie tablicy za pomocą składni wskaźnika, operatora*
Element 0 = 24
Element 1 = -1
Element 2 = 365
Element 3 = -999
Element 4 = 2011
Wyświetlenie tablicy za pomocą wskaźnika wraz ze składnią tablicy, operator[]
Element 0 = 24
Element 1 = -1
```

```
Element 2 = 365
Element 3 = -999
Element 4 = 2011
```

Analiza ▼

Aplikacja deklaruje tablicę statyczną pięciu liczb całkowitych, która jest zainicjalizowana wraz z pięcioma wartościami (patrz wiersz 8.). Program wyświetla na ekranie zawartość tablicy, korzysta przy tym z dwóch alternatywnych rozwiązań. Pierwsze polega na użyciu zmiennej tablicy wraz z operatorem dereferencji (patrz wiersz 15.), natomiast drugie wykorzystuje zmienną wskaźnika wraz z operatorem tablicy (patrz wiersz 19.).

Działanie programu pokazuje więc, że zarówno tablica `MyNumbers`, jak i wskaźnik `pNumber` w rzeczywistości działają jak wskaźnik. Innymi słowy, deklaracja tablicy jest podobna do wskaźnika, który będzie utworzony w celu przeprowadzania operacji w ramach na stałe zdefiniowanego obszaru pamięci. Zwróć uwagę na możliwość przypisania tablicy wskaźnikowi (wiersz 11.), ale też na brak możliwości przypisania wskaźnika do tablicy — wynika to z natury tablicy statycznej, która nie może być użyta jako l-wartość.

Ostrzeżenie
Ostrzeżenie

Musisz koniecznie zapamiętać, że wskaźniki zaalokowane dynamicznie za pomocą operatora `new` muszą być zwalniane przy użyciu operatora `delete` nawet wtedy, kiedy skorzystałeś ze składni podobnej do składni tablicy statycznej.

Jeżeli zapomnisz o tym, w aplikacji wystąpi wyciek pamięci, a tego należy unikać za wszelką cenę.

Najczęstsze błędy programistyczne popełniane podczas używania wskaźników

Język C++ pozwala na dynamiczną alokację pamięci, co sprawia, że zużycie pamięci przez aplikację jest optymalne. W przeciwieństwie do nowszych języków programowania, takich jak C# i Java, opartych na środowisku uruchomieniowym, C++ nie oferuje mechanizmu automatycznego usuwania nieużytków odpowiedzialnego za zwalnianie pamięci zaalokowanej przez aplikację, ale nieużywanej. Ponieważ temat wskaźników jest trudny, programista ma — niestety — wiele możliwości popełnienia błędów.

Wycieki pamięci

Jest tp prawdopodobnie jeden z najczęściej spotykanych problemów z aplikacjami utworzonymi w C++: im dłużej program działa, tym większą ilość pamięci zużywa, a sam system działa coraz wolniej. Tego rodzaju sytuacja występuje zwykle wtedy, kiedy programista nie zadbał o to, by w aplikacji pamięć dynamicznie zaalokowana przy użyciu operatora `new` była zwalniania przez odpowiednie wywołanie `delete`, gdy dany blok pamięci nie jest dłużej potrzebny.

Obowiązkiem programisty jest zagwarantowanie, że cała zaalokowana przez aplikację pamięć zostanie przez nią zwolniona. Poniżej przedstawiono sytuację, która nigdy nie powinna mieć miejsca.

```
int* pNumbers = new int [5]; // Początkowa alokacja.
// Użycie wskaźnika pNumbers.
...
// Zapomnienie o konieczności użycia polecenia delete[] pNumbers; w celu zwolnienia pamięci.
...
// Przeprowadzenie innej alokacji i nadpisanie wskaźnika.
pNumbers = new int[10]; // Wyciek poprzednio zaalokowanej pamięci.
```

Kiedy wskaźnik nie prowadzi do poprawnego adresu w pamięci?

Kiedy odwołujesz się do wskaźnika za pomocą operatora dereferencji w celu uzyskania dostępu do wartości wskazywanej przez ten wskaźnik, musisz mieć 100% pewności, że wskaźnik zawiera prawidłowy adres w pamięci. W przeciwnym razie program może ulec awarii lub zachowywać się niezgodnie z oczekiwaniami. Wydaje się logiczne, że nieprawidłowe wskaźniki to również częsta przyczyna awarii aplikacji. Wskaźnik może być nieprawidłowy z wielu różnych powodów, ale wszystkie są powiązane z kiepskim zarządzaniem pamięcią. Typową sytuacją, w której wskaźnik może stać się nieprawidłowy, przedstawiono w listingu 8.13.

Listing 8.13. Przykład złego programowania z użyciem niepoprawnych wskaźników

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Niezainicjalizowany wskaźnik (błąd).
6:     int* pTemperature;
7:
```

```
 8:    cout << "Czy jest pogodnie (t/n)?" << endl;
 9:    char UserInput = 't';
10:    cin >> UserInput;
11:
12:    if (UserInput == 't')
13:    {
14:       pTemperature = new int;
15:       *pTemperature = 30;
16:    }
17:
18:    // Wskaźnik pTemperature zawiera nieprawidłową wartość, jeśli użytkownik nacisnął
       ↪klawisz 'n'.
19:    cout << "Temperatura wynosi: " << *pTemperature;
20:
21:    // Wywołanie operatora delete nawet wtedy, gdy nie został użyty operator new.
22:    delete pTemperature;
23:
24:    return 0;
25: }
```

Wynik ▼

```
Czy jest pogodnie (t/n)? t
Temperatura wynosi: 30
```

Następne uruchomienie programu:

```
Czy jest pogodnie (t/n)? n
<AWARIA PROGRAMU!>
```

Analiza ▼

W powyższym programie występuje wiele problemów, niektóre z nich zostały już wskazane w kodzie. Zwróć uwagę na alokację pamięci i przypisanie jej wskaźnikowi (wiersz 14.), co następuje warunkowo po naciśnięciu klawisza *t* przez użytkownika. W przypadku naciśnięcia innego dowolnego klawisza blok `if` nie zostanie wykonany i wskaźnik `pTemperature` pozostanie nieprawidłowy. Dlatego też naciśnięcie klawisza *n* przez użytkownika podczas drugiego uruchomienia programu prowadzi do awarii aplikacji. Powód awarii to nieprawidłowy adres w pamięci przechowywany przez wskaźnik `pTemperature`, do którego odwołanie następuje w wierszu 19.

Równie niebezpieczne jest przeprowadzone w wierszu 22. wywołanie operatora `delete` względem wskaźnika, dla którego nie zaalokowano pamięci przy użyciu operatora `new`. Jeżeli masz kopię wskaźnika, wywołanie `delete`

wystarczy przeprowadzić tylko na jednej kopii (staraj się również unikać posiadania zbyt wielu kopii wskaźnika).

Lepsza (tzn. bezpieczniejsza i stabilniejsza) wersja programu przedstawionego w listingu 8.13 będzie miała zainicjalizowane wskaźniki używane po wcześniejszym sprawdzeniu poprawności oraz zwalniane jednokrotnie i tylko wtedy, gdy są poprawne.

Zawieszone wskaźniki (nazywane również zabłąkanymi)

Zauważ, że każdy prawidłowy wskaźnik staje się nieprawidłowy po jego zwolnieniu za pomocą operatora `delete`. Jeżeli wskaźnik `pTemperature` miałby być użyty po wierszu 22., nawet w sytuacji, gdy użytkownik nacisnął klawisz *t* i wskaźnik zachował ważność do tej chwili, po wywołaniu `delete` staje się nieprawidłowy i nie powinien być używany.

Aby uniknąć tego rodzaju problemu, wielu programistów przypisuje wskaźnikom wartość `null` podczas inicjalizacji lub po ich zwolnieniu, a przed użyciem wskaźnika za pomocą operatora dereferencji sprawdza jego poprawność.

Najlepsze praktyki podczas pracy ze wskaźnikami

Poniżej wymieniono pewne podstawowe reguły, których zastosowanie podczas używania wskaźników w aplikacji powinno ułatwić pracę.

TAK	NIE
Zawsze przeprowadzaj inicjalizację zmiennych wskaźników, ponieważ w przeciwnym razie będą zawierały zupełnie przypadkowe wartości. Wspomniane wartości są interpretowane jako adresy w pamięci, ale aplikacja nie ma uprawnień do uzyskania do nich dostępu. Jeżeli nie możesz zainicjalizować wskaźnika z poprawnym adresem zwróconym przez operator `new` lub inną prawidłową zmienną, zainicjalizuj go z wartością `null`.	Nie próbuj uzyskać dostępu do bloku pamięci, używając wskaźnika po jego zwolnieniu za pomocą operatora delete. Nie wywołuj więcej niż tylko jednokrotnie operatora `delete` względem adresu w pamięci.

TAK	NIE
Przed użyciem wskaźnika sprawdź, czy nie ma przypisanej wartości `null`. Tego rodzaju sprawdzenie gwarantuje, że wskaźniki nieposiadające przypisanych poprawnych adresów w poleceniach znajdujących się po ich deklaracji (gdzie zostały zainicjalizowane wraz z wartościami null) nie będą mogły być użyte (przykład takiego rozwiązania pokazano w listingu 8.13). Upewnij się o zaprogramowaniu wskaźników w taki sposób, że będą używane jedynie po potwierdzeniu ich poprawności. W przeciwnym przypadku może dojść do awarii aplikacji. Pamiętaj o użyciu operatora `delete` w celu zwolnienia pamięci zaalokowanej przez operator `new`. W przeciwnym razie aplikacja będzie miała wyciek pamięci, który może doprowadzić do zmniejszenia wydajności działania systemu.	Nie twórz wycieku pamięci w aplikacji na skutek zapomnienia o konieczności użycia operatora `delete` po zakończeniu pracy z zaalokowanym blokiem pamięci.

Po zapoznaniu się z najlepszymi praktykami podczas pracy ze wskaźnikami nadeszła doskonała chwila na poprawienie wyjątkowo nieudanego kodu przedstawionego w listingu 8.13. Poprawiona wersja kodu znajduje się w listingu 8.14.

Listing 8.14. Bezpieczniejsze programowanie wskaźników, poprawiona wersja listingu 8.13

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Czy jest pogodnie (t/n)? ";
 6:    char UserInput = 't';
 7:    cin >> UserInput;
 8:
 9:    if (UserInput == 't')
10:    {
11:       // Zainicjalizowany wskaźnik (to dobrze).
12:       int* pTemperature = new int;
13:       *pTemperature = 30;
14:
15:       cout << "Temperatura wynosi: " << *pTemperature << endl;
```

```
16:
17:         // Użycie operatora delete po zakończeniu pracy ze wskaźnikiem.
18:         delete pTemperature;
19:     }
20:
21:     return 0;
22: }
```

Wynik ▼

```
Czy jest pogodnie (t/n)? t
Temperatura wynosi: 30
```

Następne uruchomienie programu:

```
Czy jest pogodnie (t/n)? n
```

(Zakończenie działania bez awarii).

Analiza ▼

Podstawowa różnica polega na tym, że wskaźnik jest tworzony wtedy, gdy naprawdę jest potrzebny, czyli po naciśnięciu klawisza *t* przez użytkownika, i zainicjalizowany w chwili tworzenia (patrz wiersz 12.). Zwolnienie pamięci następuje w tym samym bloku, a tym samym nie występuje sytuacja, w której wskaźnik został użyty (w odwołaniu lub w wywołaniu `delete`), kiedy nie miał przypisanego prawidłowego adresu w pamięci.

Sprawdzenie, czy żądanie alokacji zakończyło się powodzeniem

Działanie operatora `new` kończy się powodzeniem, o ile użytkownik nie zażąda alokacji ogromnej ilości pamięci lub system nie znajduje się w stanie krytycznym i ma niewielką ilość dostępnych zasobów. Istnieją aplikacje żądające dużych ilości pamięci (np. bazy danych) i dlatego ważne jest, aby nie przyjmować założenia, że alokacja pamięci na pewno zakończy się niepowodzeniem. Język C++ oferuje dwie metody zagwarantowania poprawności wskaźnika. Metoda domyślna używa wyjątków, zakończona niepowodzeniem operacja alokacji spowoduje zgłoszenie wyjątku w postaci obiektu `std::bad_alloc`. Wynikiem zgłoszenia tego wyjątku jest przerwanie działania aplikacji, tak więc jeśli nie przygotowałeś *procedury obsługi wyjątków*, aplikacja zakończy działanie wraz z komunikatem błędu informującym o nieobsłużonym wyjątku.

W lekcji 28., zatytułowanej „Obsługa wyjątków”, temat związany z obsługą wyjątków zostanie dokładnie omówiony. W listingu 8.15 znajdziesz przykład rozwiązania, w którym procedura obsługi wyjątków jest używana do sprawdzenia, czy alokacja pamięci zakończyła się niepowodzeniem.

Listing 8.15. Obsługa wyjątków, eleganckie zakończenie pracy, gdy wywołanie operatora new zakończy się niepowodzeniem

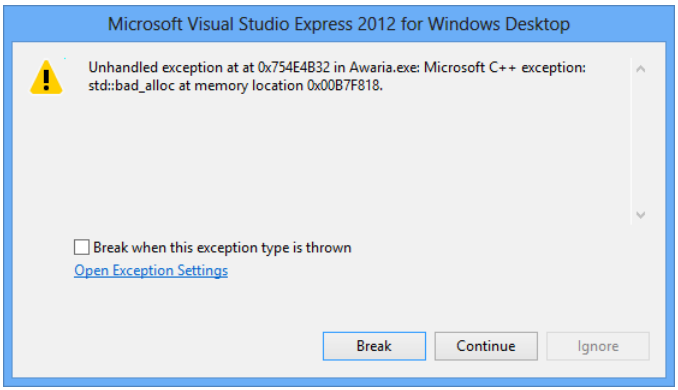
```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    try
6:    {
7:       // Żądanie alokacji dużej ilości pamięci.
8:       int* pAge = new int [536870911];
9:
10:       // Użycie zaalokowanej pamięci.
11:
12:       delete[] pAge;
13:    }
14:    catch (bad_alloc)
15:    {
16:       cout << "Alokacja pamięci zakończona niepowodzeniem. Zamknięcie
          ↪programu" << endl;
17:    }
18:    return 0;
19: }
```

Wynik ▼

```
Alokacja pamięci zakończona niepowodzeniem. Zamknięcie programu
```

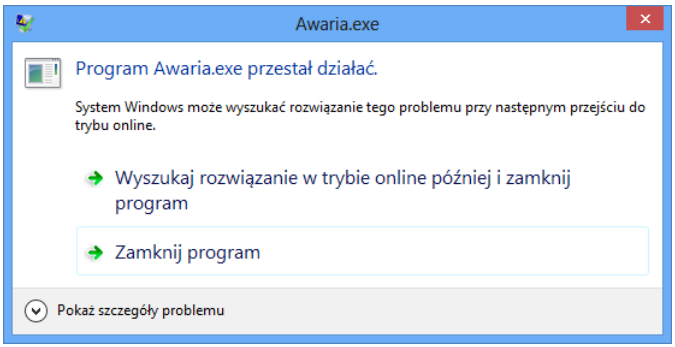
Analiza ▼

Program ten może być wykonany na różne sposoby w konkretnych komputerach. Moje środowisko pracy nie pozwala na spełnienie żądania alokacji pamięci dla 536870911 liczb całkowitych. Gdyby nie przygotowana procedura obsługi wyjątków (blok catch w wierszach od 14. do 17.), program zakończyłby niespodziewanie działanie. Uruchomienie programu w trybie debugowania w Visual Studio powoduje wygenerowanie danych wyjściowych pokazanych na rysunku 8.2.



RYSUNEK 8.2. Awaria programu na skutek braku w listingu 8.15 procedury obsługi wyjątku (program uruchomiony w trybie debugowania w kompilatorze MSVC)

Tryb debugowania powoduje wstawienie do środowiska programistycznego procedur obsługi wyjątków, a skutkiem ich działania jest komunikat pokazany na rysunku 8.2. Z kolei po uruchomieniu programu w trybie Release system operacyjny (tutaj Windows) powoduje zakończenie działania aplikacji w sposób pokazany na rysunku 8.3.



RYSUNEK 8.3. Awaria programu na skutek braku w listingu 8.15 procedury obsługi wyjątku (program uruchomiony w trybie Release)

Kiedy aplikacja ulega awarii w sposób pokazany na rysunku 8.3, jej działanie zostaje zakończone przez system operacyjny. W takim przypadku brak zaimplementowanej procedury obsługi wyjątków uniemożliwia nawet wyświetlenie jakiegoś sensownego komunikatu użytkownikowi.

Implementacja w kodzie programu procedury obsługi wyjątków daje możliwość eleganckiego zakończenia pracy aplikacji i poinformowania użytkownika o wystąpieniu problemu, zamiast pozwolenia systemowi operacyjnemu na wyświetlenie komunikatu o awarii programu.

Istnieje wariant operatora `new` o nazwie `new(nothrow)`, który w przypadku nieudanej alokacji, zamiast zgłosić wyjątek, powoduje zwrócenie wskaźnikowi wartości `null`. Przed użyciem wskaźnika można sprawdzić jego poprawność. Wersja programu wraz z operatorem `new(nothrow)` została przedstawiona w listingu 8.16.

Listing 8.16. Użycie operatora new(nothrow), który zwraca wartość null, gdy alokacja zakończy się niepowodzeniem

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    // Żądanie alokacji dużej ilości pamięci, wersja, która nie zgłasza wyjątku.
 6:    int* pAge = new(nothrow) int [0x1fffffff];
 7:
 8:    if (pAge) // Sprawdzenie pAge != NULL.
 9:    {
10:       // Użycie zaalokowanej pamięci.
11:       delete[] pAge;
12:    }
13:    else
14:       cout << "Alokacja pamięci zakończona niepowodzeniem. Zamknięcie
          ↪programu" << endl;
15:
16:    return 0;
17: }
```

Wynik ▼

```
Alokacja pamięci zakończona niepowodzeniem. Zamknięcie programu
```

Analiza ▼

Program jest taki sam jak w poprzednim przykładzie (patrz listing 8.15), ale używa operatora `new(nothrow)`, który w przypadku nieudanej alokacji pamięci zwraca wartość `null`, zamiast zgłosić wyjątek `std::bad_alloc`. Obie przedstawione formy są dobre, wybór odpowiedniej należy do Ciebie.

Czym jest referencja?

Referencja jest aliasem dla zmiennej. Po zadeklarowaniu referencji konieczne jest jej zainicjalizowanie ze zmienną. Dlatego też zmienna referencji to po prostu inny sposób uzyskania dostępu do danych we wskazanej zmiennej. Referencję deklaruje się przy użyciu operatora referencji (&), jak to przedstawiono w poniższym fragmencie kodu:

```
TypZmiennej NazwaZmiennej = Wartość;
TypZmiennej& ZmiennaReferencyjna = NazwaZmiennej;
```

Aby dokładnie zrozumieć, jak zadeklarować referencje i używać ich w kodzie, zapoznaj się z listingiem 8.17.

Listing 8.17. Pokazanie, że referencje są aliasami dla przypisanych wartości

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Original = 30;
 6:    cout << "Zmienna Original = " << Original << endl;
 7:    cout << "Zmienna Original jest pod adresem: " << hex << &Original <<
       ↪endl;
 8:
 9:    int& Ref = Original;
10:    cout << "Ref jest pod adresem: " << hex << &Ref << endl;
11:
12:    int& Ref2 = Ref;
13:    cout << "Ref2 jest pod adresem: " << hex << &Ref2 << endl;
14:    cout << "Ref2 pobiera wartość, Ref2 = " << dec << Ref2 << endl;
15:
16:    return 0;
17: }
```

Wynik ▼

```
Zmienna Original = 30
Zmienna Original jest pod adresem: 0044FB5C
Ref jest pod adresem: 0044FB5C
Ref2 jest pod adresem: 0044FB5C
Ref2 pobiera wartość, Ref2 = 30
```

Analiza ▼

Dane wyjściowe programu pokazują, że referencja, niezależnie od tego, czy została zainicjalizowana z oryginalną zmienną (patrz wiersz 9.), czy do innej referencji (patrz wiersz 12.), zawsze odwołuje się do tego samego położenia w pamięci, w którym początkowo się znajdowała. Dlatego też referencje są prawdziwymi aliasami, po prostu innymi nazwami zmiennej `Original`. Wyświetlona w wierszu 14. wartość `Ref2` jest taka sama jak zmiennej `Original` (wyświetlona w wierszu 6.), ponieważ `Ref2` jest aliasem do `Original` i prowadzi do tego samego miejsca w pamięci.

Dlaczego referencje są użyteczne?

Referencje pozwalają na pracę z miejscem w pamięci, z którym zostały zainicjalizowane. Z tego powodu referencje są szczególnie użyteczne podczas programowania funkcji. Jak się dowiedziałeś w lekcji 7., zatytułowanej „Funkcje", typowa funkcja jest deklarowana w przedstawiony poniżej sposób:

```
TypZwrotny DowolnaOperacja(Typ Parametr);
```

Funkcja `DowolnaOperacja()` jest wywoływana następująco:

```
TypZwrotny Wynik = DowolnaOperacja(argument); // Wywołanie funkcji.
```

Powyższe polecenie powoduje skopiowanie `argumentu` do `Parametru`, który następnie będzie użyty przez funkcję `DowolnaOperacja()`. Krok kopiowania może być całkiem sporym obciążeniem, jeśli dany `argument` zużywa dużą ilość pamięci. Podobnie gdy funkcja `DowolnaOperacja()` zwraca wartość, jest ona kopiowana z powrotem do zmiennej `Wynik`. Byłoby idealnie, gdyby można było pominąć krok kopiowania i umożliwić funkcji bezpośrednią pracę z danymi na stosie programu wywołującego. Dzięki referencjom takie rozwiązanie jest możliwe.

Wersja funkcji bez kroku kopiowania przedstawia się następująco:

```
TypZwrotny DowolnaOperacja(Typ& Parametr); // Zwróć uwagę na referencję &.
```

Funkcję tę można wywołać w poniższy sposób:

```
TypZwrotny Wynik = DowolnaOperacja(argument);
```

Ponieważ argument jest przekazywany przez referencję, `Parametr` nie jest kopią `argumentu`, a raczej aliasem do niego, podobnie jak w przypadku `Ref` w listingu 8.17. Ponadto funkcja akceptująca parametr jako referencję może opcjonalnie zwracać wartość, używając parametru referencji. Zapoznaj się

z listingiem 8.18, aby przekonać się, jak funkcja może używać referencji zamiast wartości zwrotnej.

Listing 8.18. Funkcja obliczająca kwadrat zwrócony później w parametrze przez referencję

```
0: #include <iostream>
1: using namespace std;
2:
3: void ReturnSquare(int& Number)
4: {
5:    Number *= Number;
6: }
7:
8:
9: int main()
10: {
11:    cout << "Podaj liczbę, którą chcesz podnieść do kwadratu: ";
12:    int Number = 0;
13:    cin >> Number;
14:
15:    ReturnSquare(Number);
16:    cout << "Kwadrat podanej liczby wynosi: " << Number << endl;
17:
18:    return 0;
19: }
```

Wynik ▼

```
Podaj liczbę, którą chcesz podnieść do kwadratu: 5
Kwadrat podanej liczby wynosi: 25
```

Analiza ▼

Funkcja obliczająca kwadrat podanej liczby znajduje się w wierszach od 3. do 6. Zwróć uwagę, że dane wejściowe akceptuje w postaci parametru przekazywanego przez referencję, a wynik zwraca w ten sam sposób. Jeśli zapomnisz oznaczyć parametr `Number` jako referencję (`&`), wynik nie zostanie przekazany wywołującej funkcji `main()`, ponieważ funkcja `ReturnSquare()` będzie przeprowadzała operację na lokalnej kopii `Number` niszczonej w chwili zakończenia działania funkcji. Dzięki użyciu referencji masz pewność, że funkcja `ReturnSquare()` działa względem tego samego adresu w pamięci, który zaalokowano dla `Number` w funkcji `main()`. Wynik operacji jest więc dostępny w funkcji `main()` nawet po zakończeniu działania funkcji `ReturnSquare()`.

W omówionym programie został zmodyfikowany parametr danych wejściowych zawierający liczbę podaną przez użytkownika. Jeżeli potrzebujesz obu wartości, czyli liczby początkowej i podniesionej do kwadratu, konieczne jest przygotowanie funkcji akceptującej dwa parametry: z liczbą początkową oraz z liczbą podniesioną do kwadratu.

Użycie słowa kluczowego const w referencjach

Być może wystąpi konieczność posiadania referencji, jaka nie będzie mogła zmienić wartości początkowej zmiennej, której jest aliasem. Użycie słowa kluczowego `const` podczas deklarowania tego rodzaju referencji to jedno z możliwych rozwiązań:

```
int Original = 30;
const int& ConstRef = Original;
ConstRef = 40; // Niedozwolone: ConstRef nie może zmienić wartości Original.
int& Ref2 = ConstRef; // Niedozwolone: Ref2 nie jest stałą.
const int& ConstRef2 = ConstRef; // OK.
```

Przekazywanie funkcji argumentów przez referencję

Jedną z największych zalet referencji jest to, że umożliwia wywoływanej funkcji pracę z parametrami, które nie muszą być kopiowane z funkcji wywołującej, co znacznie poprawia wydajność działania wywoływanej funkcji. Jednak wywoływana funkcja działa, używając parametrów bezpośrednio znajdujących się na stosie funkcji wywołującej. Bardzo często ważne jest zagwarantowanie, że wywołana funkcja nie będzie mogła zmienić wartości zmiennej w funkcji wywołującej. Referencje zdefiniowane z użyciem słowa kluczowego `const` doskonale się przydają w takich sytuacjach, co zostało zademonstrowane w listingu 8.19. Parametr referencji `const` nie może być używany jako l-wartość, więc próba przypisania mu wartości spowoduje wygenerowanie błędu w trakcie kompilacji.

Listing 8.19. Użycie referencji const w celu zagwarantowania, że wywołująca funkcja nie będzie mogła zmodyfikować wartości podanej przez referencję

```
0: #include <iostream>
1: using namespace std;
2:
```

```
 3: void CalculateSquare(const int& Number, int& Result) // Zwróć uwagę na "const".
 4: {
 5:    Result = Number*Number;
 6: }
 7:
 8: int main()
 9: {
10:    cout << "Podaj liczbę, którą chcesz podnieść do kwadratu: ";
11:    int Number = 0;
12:    cin >> Number;
13:
14:    int Square = 0;
15:    CalculateSquare(Number, Square);
16:    cout << Number << "^2 = " << Square << endl;
17:
18:    return 0;
19: }
```

Wynik ▼

```
Podaj liczbę, którą chcesz podnieść do kwadratu: 27
27^2 = 729
```

Analiza ▼

W przeciwieństwie do poprzedniego programu, w którym zmienna zawierająca liczbę początkową przechowywała także wynik działania funkcji, powyższy program używa dwóch zmiennych: jednej przeznaczonej dla liczby początkowej podanej przez użytkownika i drugiej dla wyniku przeprowadzonej operacji. Aby zagwarantować, że liczba podana przez użytkownika nie będzie mogła być zmieniona, użyto referencji `const` zdefiniowanej za pomocą słowa kluczowego `const` (patrz wiersz 3.). W ten sposób parametr `Number` jest automatycznie parametrem danych wejściowych, a jego wartość nie może być zmodyfikowana.

Jako eksperyment możesz wprowadzić modyfikację w wierszu 5., aby zwrócić wynik z użyciem logiki takiej samej jak w listingu 8.18:

```
Number *= Number;
```

Bez wątpienia w trakcie kompilacji nastąpi wygenerowanie błędu informującego o braku możliwości zmiany wartości `const`. Dlatego też referencja `const` to użyteczne narzędzie udostępniane przez język C++ wskazujące, że dany parametr jest parametrem danych wejściowych. Jednocześnie masz gwarancję, że wartość przekazywana przez referencję nie będzie mogła być modyfikowana przez

wywołaną funkcję. To może wydawać się oczywiste, ale w środowisku pracy obejmującym wielu programistów, gdzie jedna osoba tworzy pierwszą wersję funkcji, a inna poprawia ją i usprawnia, użycie referencji `const` wprowadza dużą różnicę w jakości tworzonego kodu.

Podsumowanie

W tej lekcji poznałeś wskaźniki i referencje. Dowiedziałeś się, jak wskaźniki mogą być wykorzystywane do uzyskania dostępu i operowania pamięcią oraz jakie to przydatne narzędzie pomagające w trakcie dynamicznej alokacji pamięci. Poznałeś operatory `new` i `delete` używane do alokacji pamięci dla elementu. Ponadto poznałeś ich odpowiedniki `new[...]` i `delete[]` pomagające w alokacji pamięci dla tablicy danych. W lekcji przedstawiono również pułapki czyhające na programistów w trakcie pracy ze wskaźnikami i dynamiczną alokacją pamięci. Przekonałeś się też, jak ważne jest zwalnianie dynamicznie alokowanej pamięci, aby uniknąć tzw. wycieków pamięci. Referencje są aliasami i jednocześnie oferującą potężne możliwości alternatywą dla używania wskaźników podczas przekazywania argumentów funkcjom — referencje gwarantują zachowanie ich poprawności. Dowiedziałeś się, jak we właściwy sposób stosować `const` podczas pracy ze wskaźnikami i referencjami oraz jak deklarować funkcje na najwyższym możliwym poziomie zachowania stałości parametrów.

Pytania i odpowiedzi

Pytanie: Dlaczego stosować dynamiczną alokację pamięci, skoro mogę zdefiniować tablicę statyczną i nie przejmować się zwalnianiem pamięci?

Odpowiedź: Tablice statyczne mają stałą wielkość i nigdy nie zapewniają możliwości skalowania w górę, jeśli aplikacja będzie wymagała większej ilości pamięci, ani nie optymalizują użycia pamięci, gdy aplikacja korzysta z mniejszej ilości danych. W takich sytuacjach stosowanie dynamicznej alokacji pamięci powoduje istotną różnicę.

Pytanie: Mam dwa przedstawione poniżej wskaźniki:

```
int* pNumber = new int;
int* pCopy = pNumber;
```

Czy dobrym rozwiązaniem będzie wywołanie operatora `delete` względem obu, aby zagwarantować całkowite zwolnienie pamięci?

Odpowiedź: To jest błędne rozwiązanie. Można tylko jednokrotnie wywołać operator `delete` względem danego adresu w pamięci zwróconego przez operator `new`. Ponadto warto unikać sytuacji, w której dwa wskaźniki prowadzą do tego samego adresu w pamięci, ponieważ wywołanie operatora `delete` spowoduje unieważnienie obu wskaźników. Nie powinieneś także tworzyć programu w taki sposób, że nie masz pewności dotyczącej poprawności używanych wskaźników.

Pytanie: Kiedy powinienem używać operatora `new(nothrow)`?

Odpowiedź: Jeśli nie chcesz zajmować się obsługą wyjątku `std::bad_alloc`, wtedy możesz użyć wersji `new(nothrow)` operatora `new`, ponieważ w przypadku nieudanej operacji alokacji pamięci zwraca on wartość `null`.

Pytanie: Potrzebuję wywołać funkcję obliczającą pole. Do dyspozycji mam dwie przedstawione poniżej metody:

```
void CalculateArea (const double* const pRadius, double* const pArea);
void CalculateArea (const double& radius, double& area);
```

Który z powyższych wariantów powinienem wybrać?

Odpowiedź: Drugi z wymienionych, użycie referencji jako referencji nie jest błędem, podczas gdy w przypadku wskaźników już tak. Ponadto druga metoda jest prostsza.

Pytanie: Mam dwa wskaźniki:

```
int Number = 30;
const int* pNumber = &Number;
```

Rozumiemwzględu na deklarację `const`. Czy mogę przypisać `pNumber` do wskaźnika typu innego niż `const`, a następnie użyć go do modyfikacji wartości znajdującej się w `Number`?

Odpowiedź: Nie, nie ma możliwości zmiany wskaźnika `const`:

```
int* pAnother = pNumber; // Wskaźnika const nie można przypisać do nieconst.
```

Pytanie: Dlaczego mam trudzić się przekazywaniem funkcji wartości przez referencję?

Odpowiedź: Nie musisz tego robić, o ile takie rozwiązanie nie ma dużego wpływu na kod. Jeśli jednak parametry funkcji akceptują obiekty, które mogą być całkiem duże (wielkość wyrażona w bajtach), ich przekazanie przez wartość stanie się naprawdę kosztowną operacją. Wywołanie funkcji może być znacznie efektywniejsze po użyciu referencji. Pamiętaj o częstym użyciu `const`, z wyjątkiem funkcji, które muszą w zmiennej przechowywać wynik działania.

Pytanie: Jaka jest różnica pomiędzy dwoma poniższymi deklaracjami?

```
int MyNumbers[100];
int* MyArrays[100];
```

Odpowiedź: `MyNumbers` to tablica liczb całkowitych, tzn. `MyNumbers` jest wskaźnikiem do miejsca w pamięci przechowującego sto liczb całkowitych i wskazuje pierwszą z nich (indeks 0). To jest statyczna alternatywa dla poniższego fragmentu kodu:

```
int* MyNumbers = new int [100]; // Tablica zaalokowana dynamicznie.
// Użycie MyNumbers.
delete MyNumbers;
```

Z drugiej strony, `MyArrays` to tablica stu liczb całkowitych, każdy wskaźnik może prowadzić do liczby całkowitej lub tablicy liczb całkowitych.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Dlaczego nie można przypisać referencji `const` do referencji innej niż `const`?
2. Czy `new` i `delete` to funkcje?
3. Jaka jest natura wartości przechowywanej w zmiennej wskaźnika?
4. Jakiego operatora użyjesz w celu uzyskania dostępu do danych wskazywanych przez wskaźnik?

Ćwiczenia

1. Co zostanie wyświetlone po wykonaniu przedstawionych poniżej poleceń?

```
0: int Number = 3;
1: int* pNum1 = &Number;
2: *pNum1 = 20;
3: int* pNum2 = pNum1;
4: Number *= 2;
5: cout << *pNum2;
```

2. Jakie są podobieństwa i różnice pomiędzy przedstawionymi poniżej trzema przeciążonymi funkcjami?

```
int DoSomething(int Num1, int Num2);
int DoSomething(int& Num1, int& Num2);
int DoSomething(int* pNum1, int* pNum2);
```

3. Jak zmienisz deklarację pNum1 w ćwiczeniu 1. (wiersz 1.), aby przypisanie w wierszu 3. stało się nieprawidłowe? (Podpowiedź: musisz upewnić się, że pNum1 nie może zmienić wskazywanych danych).

4. **Łowcy błędów:** Co jest nie tak z poniższym fragmentem kodu?

```
#include <iostream>
using namespace std;
int main()
{
   int *pNumber = new int;
   pNumber = 9;
   cout << "Wartość wskaźnika pNumber: " << *pNumber;
   delete pNumber;
   return 0;
}
```

5. **Łowcy błędów:** Co jest nie tak z poniższym fragmentem kodu?

```
#include <iostream>
using namespace std;
int main()
{
   int pNumber = new int;
   int* pNumberCopy = pNumber;
   *pNumberCopy = 30;
   cout << *pNumber;
   delete pNumberCopy;
   delete pNumber;
   return 0;
}
```

6. Jakie będą dane wyjściowe powyższego programu po jego poprawieniu?

Część II

Podstawy programowania zorientowanego obiektowo w C++

Lekcja 9

Klasy i obiekty

Poznałeś już strukturę prostego programu, którego wykonanie rozpoczyna się w funkcji main(). Tego rodzaju program pozwala na deklarowanie zmiennych lokalnych i globalnych, stałych, a także na tworzenie gałęzi logiki wykonywania w postaci modułów funkcji, które mogą pobierać parametry i zwracać wartości. Wszystkie wymienione cechy powodują, że program jest podobny do utworzonego w proceduralnym języku C, który jest pozbawiony funkcji zorientowanych obiektowo. Innymi słowy, musisz poznać temat zarządzania danymi i łączenia tych danych z metodami.

Z tej lekcji dowiesz się:

- **czym są klasy,**
- **jak klasy mogą pomóc w konsolidacji danych wraz z metodami (zbliżonymi do funkcji), które działają ze wspomnianym danymi,**
- **czym są konstruktory, konstruktory kopiujące i destruktory,**
- **jak standard C++11 pomaga w poprawieniu wydajności przy użyciu konstruktora przenoszącego,**
- **jak przedstawia się zorientowana obiektowo koncepcja hermetyzacji i abstrakcji,**
- **czym jest wskaźnik this,**
- **co to jest struktura i czym różni się od klasy.**

Koncepcja klas i obiektów

Wyobraź sobie, że tworzysz program modelujący istotę ludzką. Istota ta ma tożsamość, na którą składają się imię, data i miejsce urodzenia, płeć itd. Istota ludzka może wykonywać pewne funkcje, np. komunikować się i przedstawiać innym. Wymienione wcześniej informacje to dane dotyczące istoty ludzkiej, natomiast kolejne to informacje o jej funkcjach (patrz rysunek 9.1).

RYSUNEK 9.1. Informacje dotyczące obiektu Human (człowiek)

Aby utworzyć model istoty ludzkiej, potrzebna jest konstrukcja pozwalająca na zgrupowanie atrybutów definiujących człowieka (dane) *i* czynności, które może wykonywać (metody — są podobne do funkcji), przy użyciu dostępnych atrybutów. Taką konstrukcją jest *klasa*.

Deklarowanie klasy

Deklaracja klasy zawiera słowo kluczowe `class`, po którym znajduje się nazwa klasy, a dalej blok poleceń `{...}` obejmujący zestaw atrybutów elementów składowych i metod. Po nawiasie klamrowym, w którym ujęty jest blok poleceń klasy, znajduje się średnik.

Deklaracja klasy przypomina deklarację funkcji i dostarcza kompilatorowi informacje na temat danej klasy oraz jej właściwości. Sama deklaracja klasy nie ma żadnego wpływu na działanie programu, ponieważ klasa musi być używana w dokładnie taki sam sposób jak wywoływana jest funkcja.

Poniżej przedstawiono przykładową klasę modelującą istotę ludzką (na razie zignoruj syntaktyczne skróty):

```
class Human
{
   // Atrybuty danych:
   string Name;
   string DateOfBirth;
   string PlaceOfBirth;
   string Gender;

   // Metody:
   void Talk(string TextToTalk);
   void IntroduceSelf();
   ...
};
```

Trzeba w tym miejscu dodać, że w metodzie `IntroduceSelf()` używa się metody `Talk()` oraz pewnych atrybutów danych zgrupowanych w klasie `Human`. Dlatego też udostępniane przez język C++ słowo kluczowe `class` to oferujące potężne możliwości rozwiązanie pozwalające na tworzenie własnych typów danych *hermetyzujących* atrybuty i funkcje używające wspomnianych atrybutów. Wszystkie atrybuty klasy (w omawianym przykładzie to `Name`, `DateOfBirth`, `PlaceOfBirth` i `Gender`) i wszystkie zadeklarowane w niej funkcje (tutaj `Talk()` i `IntroduceSelf()`) są nazywane elementami składowymi klasy `Human`.

Hermetyzacja to możliwość logicznego grupowania danych i metod wykorzystujących wspomniane dane; to również bardzo ważna koncepcja programowania zorientowanego obiektowo.

Metody to w zasadzie funkcje będące elementami składowymi klasy.	Uwaga

Tworzenie obiektu klasy

Klasa przypomina matrycę, a sama deklaracja klasy nie ma żadnego wpływu na przebieg działania programu. Rzeczywistą postacią klasy w trakcie działania programu jest obiekt. W celu użycia funkcji klasy zwykle należy utworzyć obiekt danej klasy, a następnie użyć go, aby uzyskać dostęp do atrybutów i metod elementów składowych.

Utworzenie obiektu klasy `Human` jest podobne do utworzenia egzemplarza innego typu, np. `double`:

```
double Pi = 3.1415;  // Zmienna typu double zadeklarowana jako zmienna lokalna (na stosie).
Human Tomek;  // Obiekt klasy Human zadeklarowany jako zmienna lokalna.
```

Alternatywnie można dynamicznie zaalokować pamięć dla egzemplarza klasy `Human` przy użyciu słowa kluczowego `new`, podobnie jak np. liczby całkowitej typu `int`:

```
int* pNumber = new int;  // Liczba całkowita zaalokowana dynamicznie w wolnej pamięci.
delete pNumber;  // Zwolnienie pamięci.
Human* pAnotherHuman = new Human(); // Dynamicznie zaalokowany obiekt Human.
delete pAnotherHuman;  // Zwolnienie pamięci zaalokowanej dla obiektu Human.
```

Uzyskanie dostępu do elementów składowych przy użyciu operatora kropki

Przykładem istoty ludzkiej może być Tomek, mężczyzna urodzony w roku 1970 w Łodzi. Egzemplarz `Tomek` jest obiektem klasy `Human`, to postać klasy istniejąca w trakcie działania programu:

```
Human Tomek;  // Egzemplarz obiektu Human.
```

Jak widać w deklaracji klasy, obiekt `Tomek` ma atrybuty, takie jak `DateOfBirth`, do których dostęp można uzyskać za pomocą operatora kropki (`.`):

```
Tomek.DateOfBirth = "1970";
```

To jest możliwe, ponieważ `DateOfBirth` należy do klasy `Human`, czyli jest częścią wspomnianej wcześniej matrycy, co widać w deklaracji klasy. Atrybut `DateOfBirth` istnieje w rzeczywistości — tzn. w trakcie działania programu — tylko po utworzeniu obiektu. Operator kropki pozwala na uzyskanie dostępu do atrybutów obiektu.

To samo dotyczy metod, takich jak `IntroduceSelf()`:

```
Tomek.IntroduceSelf();
```

Jeżeli masz wskaźnik `pTomek` wskazujący egzemplarz klasy `Human`, w celu uzyskania dostępu do elementów składowych klasy możesz użyć operatora wskaźnika (`->`), co zostanie przedstawione w kolejnym punkcie, lub operatora dereferencji (`*`), odwołując się do obiektu za pomocą operatora kropki:

```
Human* pTomek = new Human();
(*pTomek).IntroduceSelf();
```

Uzyskanie dostępu do elementów składowych przy użyciu operatora wskaźnika

Jeżeli obiekt został utworzony w puli wolnej pamięci przy użyciu słowa kluczowego `new` lub jeśli masz wskaźnik prowadzący do obiektu, wtedy dostęp do atrybutów i funkcji elementów składowych może odbywać się z wykorzystaniem operatora wskaźnika (`->`):

```
Human* pToekm = new Human();
pTomek->DateOfBirth = "1970";
pTomek->IntroduceSelf();
delete pTomek;

// Alternatywne rozwiązanie, gdy masz wskaźnik.
Human Tomek;
Human* pTomek = &Tomek; // Przypisanie adresu za pomocą operatora referencji&.
pTomek->DateOfBirth = "1970"; // To odpowiednik polecenia Tomek.DateOfBirth = "1970";.
pTomek->IntroduceSelf(); // To odpowiednik polecenia Tomek.IntroduceSelf();.
```

Warta przeprowadzenia kompilacji klasa `Human` wykorzystująca słowa kluczowe, takie jak `private` i `public`, została przedstawiona w listingu 9.1.

Listing 9.1. Warta kompilacji klasa Human

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:    string Name;
 8:    int Age;
 9:
10: public:
11:    void SetName (string HumansName)
12:    {
13:       Name = HumansName;
14:    }
15:
16:    void SetAge(int HumansAge)
17:    {
18:       Age = HumansAge;
19:    }
20:
21:    void IntroduceSelf()
22:    {
```

```
23:       cout << "Mam na imię " + Name << " i mam ";
24:       cout << Age << " lat" << endl;
25:    }
26: };
27:
28: int main()
29: {
30:    // Utworzenie obiektu klasy Human wraz z atrybutem Name o wartości "Adam".
31:    Human FirstMan;
32:    FirstMan.SetName("Adam");
33:    FirstMan.SetAge(30);
34:
35:    // Utworzenie obiektu klasy Human wraz z atrybutem Name o wartości "Ewa".
36:    Human FirstWoman;
37:    FirstWoman.SetName("Ewa");
38:    FirstWoman.SetAge (28);
39:
40:    FirstMan.IntroduceSelf();
41:    FirstWoman.IntroduceSelf();
42: }
```

Wynik ▼

```
Mam na imię Adam i mam 30 lat
Mam na imię Ewa i mam 28 lat
```

Analiza ▼

W wierszach od 4. do 26. przedstawiono konstrukcję bardzo prostej klasy w C++. Warto na tę klasę spojrzeć z praktycznego punktu widzenia i zignorować pojęcia oraz koncepcje, których jeszcze nie rozumiesz, ponieważ ich dokładne omówienie znajduje się dalej w tej lekcji. Skoncentruj się na strukturze klasy `Human` i sposobie jej użycia w funkcji `main()`.

Klasa zawiera dwie zmienne prywatne (`private`): zdeklarowaną w wierszu 7. zmienną `Name` typu `string` i zadeklarowaną w wierszu 8. zmienną `Age` typu `int`. Ponadto w klasie mamy kilka funkcji publicznych (nazywanych również metodami): `SetName()`, `SetAge()` i `IntroduceSelf()` zadeklarowanych w wierszach — odpowiednio — 11., 16. i 21. Wymienione metody używają zmiennych prywatnych. Następnie w wierszach 31. i 36. tworzone są w funkcji `main()` dwa obiekty klasy `Human`. Po utworzeniu obiektów `FirstMan` i `FirstWoman` następuje przypisanie im zmiennych elementów składowych przy użyciu metod `SetName()` i `SetAge()` nazywanych metodami akcesora. Zwróć uwagę na

wywołanie metody `IntroduceSelf()` w wierszach 40. i 41. Metoda jest wywoływana względem dwóch różnych obiektów i powoduje wygenerowanie dwóch odmiennych wierszy danych wejściowych wykorzystujących zmienne przypisane nieco wcześniej w kodzie.

Czy zauważyłeś użycie słów kluczowych `private` i `public` w listingu 9.1? Najwyższa pora poznać oferowane przez język C++ funkcje pomagające w chronieniu atrybutów klasy, jakie powinny pozostać ukryte przed tymi, którzy ich używają.

Słowa kluczowe public i private

Każdy z nas charakteryzuje się wieloma informacjami, z których część (np. imię) jest dostępna dla innych ludzi, jacy nas otaczają. Tego rodzaju informacje można nazywać publicznymi. Jednak istnieją również informacje, np. wysokość dochodu, którymi nie chcemy się dzielić z otaczającym nas światem. Tego rodzaju informacje nazywamy prywatnymi i nie ujawniamy ich innym.

Język C++ pozwala na określenie atrybutów i metod klasy jako publicznych (obiekt klasy może je wywoływać) lub prywatnych (tylko klasa bądź jej „przyjaciele" mogą mieć dostęp do prywatnych elementów składowych). Oferowane przez C++ słowa kluczowe `public` i `private` pomagają tworzącemu klasę na wskazanie elementów, które mogą być wywoływane z zewnątrz, np. z funkcji `main()`, oraz elementów niedostępnych na zewnątrz danej klasy.

Jakie zalety dla programisty ma możliwość określenia atrybutu lub metody jako prywatnych (`private`)? Spójrz na przedstawioną poniżej deklarację klasy `Human` i zignoruj wszystko poza elementem składowym w postaci atrybutu `Age`:

```
class Human
{
private:
   // Dane prywatnych elementów składowych:
   int Age;
   string Name;

public:
   int GetAge()
   {
      return Age;
   }

   void SetAge(int InputAge)
```

```
    {
        Age = InputAge;
    }

// Pozostałe elementy składowe i deklaracje.
};
```

Przyjmujemy założenie, że egzemplarz klasy `Human` nosi nazwę `Ewa`:

```
Human Ewa;
```

Kiedy użytkownik tego egzemplarza próbuje uzyskać dostęp do atrybutu `Age` przy użyciu poniższego polecenia:

```
cout << Ewa.Age; // Błąd kompilacji.
```

otrzymuje błąd kompilacji o treści podobnej do: „Błąd: `Human::Age` — nie ma możliwości uzyskania dostępu do prywatnego elementu składowego zadeklarowanego w klasie `Human`". Jedyny dozwolony sposób na uzyskanie dostępu do atrybutu `Age` to użycie publicznej metody `GetAge()` oferowanej przez klasę `Human` i zaimplementowanej w sposób, który twórca klasy uznał za odpowiedni w celu udostępniania wartości atrybutu `Age`:

```
cout << Ewa.GetAge(); // OK.
```

Twórca klasy `Human` może przygotować metodę `GetAge()` w taki sposób, że podawany wiek (`Age`) Ewy będzie mniejszy niż w rzeczywistości. Innymi słowy, język C++ pozwala klasie na kontrolę oferowanych atrybutów i sposób ich udostępniania. W przypadku braku publicznej metody `GetAge()` w klasie `Human` taka deklaracja klasy gwarantuje, że użytkownik nie będzie mógł pobrać wartości atrybutu `Age`. Taka możliwość jest użyteczna w sytuacjach, które zostaną przedstawione dalej w tej lekcji.

Podobnie nie ma możliwości bezpośredniego przypisania `Human::Age`:

```
Ewa.Age = 22; // Błąd kompilacji.
```

Jedyny dozwolony sposób przypisania wartości atrybutowi `Age` to użycie metody `SetAge()`:

```
Ewa.SetAge(22); // OK.
```

Przedstawiony mechanizm ma wiele zalet. Bieżąca implementacja metody `SetAge()` pozwala jedynie na bezpośrednie ustawienie wartości elementu składowego `Human::Age`. Jednak metodę `SetAge()` można również wykorzystać do sprawdzenia, czy przypisywana zmiennej `Age` wartość jest niezerowa

i nieujemna, a tym samym przeprowadzić weryfikację zewnętrznych danych wejściowych:

```
class Human
{
private:
   int Age;

public:
   void SetAge(int InputAge)
   {
      if (InputAge > 0)
         Age = InputAge;
   }
};
```

Jak się przekonałeś, język C++ pozwala twórcy klasy na zachowanie kontroli nad sposobem uzyskania dostępu do danych i operowania atrybutami danych.

Abstrakcja danych dzięki słowu kluczowemu private

C++ pozwala na projektowanie klasy jako pojemnika hermetyzującego dane i metody działające z tymi danymi, a także na wskazanie przy użyciu słowa kluczowego `private` informacji niedostępnych dla świata zewnętrznego (tzn. na zewnątrz klasy). Jednocześnie dzięki metodom publicznym (`public`) masz zapewnioną kontrolę nad udostępnianiem informacji zadeklarowanych jako prywatne (`private`). Dlatego też implementacja klasy pozwala na abstrakcję danych, o których nie powinien wiedzieć świat zewnętrzny, tzn. inne klasy i funkcje, takie jak `main()`.

Powracamy do przykładu, w którym `Human::Age` to prywatny element składowy. Jak pewnie wiesz, w świecie rzeczywistym wiele osób woli nie ujawniać swojego prawdziwego wieku. Jeżeli klasa `Human` miałaby podawać wiek o dwa lata mniejszy niż w rzeczywistości, można to łatwo zrealizować z wykorzystaniem publicznej funkcji `GetAge()` używającej parametru `Human::Age`. Wystarczy od wieku odjąć wartość 2, a następnie zwrócić wynik. Gotowe rozwiązanie przedstawiono w listingu 9.2.

Listing 9.2. Model klasy Human, w której prawdziwy wiek przechowywany w atrybucie Age jest ukrywany przed użytkownikiem, któremu podawany jest wiek pomniejszony o dwa

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Human
 4: {
 5: private:
 6:    // Dane prywatnych elementów składowych:
 7:    int Age;
 8:
 9: public:
10:    void SetAge(int InputAge)
11:    {
12:       Age = InputAge;
13:    }
14:
15:    // Klasa Human kłamie, podając wartość atrybutu Age (jeśli jest większa niż 30).
16:    int GetAge()
17:    {
18:       if (Age > 30)
19:          return (Age - 2);
20:       else
21:          return Age;
22:    }
23: };
24:
25: int main()
26: {
27:    Human FirstMan;
28:    FirstMan.SetAge(35);
29:
30:    Human FirstWoman;
31:    FirstWoman.SetAge(22);
32:
33:    cout << "Wiek pierwszego mężczyzny " << FirstMan.GetAge() << endl;
34:    cout << "Wiek pierwszej kobiety " << FirstWoman.GetAge() << endl;
35:
36:    return 0;
37: }
```

Wynik ▼

```
Wiek pierwszego mężczyzny 33
Wiek pierwszej kobiety 22
```

Analiza ▼

Zwróć uwagę na publiczną metodę `Human::GetAge()` w wierszu 16. Ponieważ rzeczywisty wiek jest przechowywany w prywatnym elemencie składowym `Human::Age`, który nie jest bezpośrednio dostępny, użytkownik z zewnątrz może otrzymać wartość atrybutu `Age` jedynie przy użyciu publicznej metody `GetAge()` klasy `Human`. Dlatego też rzeczywisty wiek przechowywany w `Human::Age` pozostaje ukryty przed światem zewnętrznym.

Abstrakcja to bardzo ważna koncepcja w językach zorientowanych obiektowo. Pozwala programiście na wskazanie atrybutów klasy, które pozostaną znane jedynie klasie i jej elementom składowym, a tym samym niedostępne dla nikogo z zewnątrz (z wyjątkiem elementów zadeklarowanych jako „przyjaciele").

Konstruktory

Konstruktor to funkcja (lub metoda) specjalna wywoływana w trakcie tworzenia obiektu. Podobnie jak funkcje, także konstruktory mogą być przeciążane.

Deklarowanie i implementowanie konstruktora

Konstruktor to funkcja specjalna o takiej samej nazwie jak nazwa klasy i pozbawiona wartości zwrotnej. Konstruktor omawianej klasy `Human` jest deklarowany w następujący sposób:

```
class Human
{
public:
   Human(); // Deklaracja konstruktora.
};
```

Powyższy konstruktor może być zaimplementowany w miejscu jego deklaracji lub na zewnątrz deklaracji klasy. Implementacja (tzn. definicja) w klasie przedstawia się następująco:

```
class Human
{
public:
   Human()
   {
```

```
        // Miejsce na kod konstruktora.
    }
};
```

Z kolei wariant definicji konstruktora na zewnątrz deklaracji klasy przedstawia się następująco:

```
class Human
{
public:
   Human(); // Deklaracja konstruktora.
};

// Definicja konstruktora (implementacja).
Human::Human()
{
   // Miejsce na kod konstruktora.
}
```

Uwaga

> Dwa dwukropki (`::`) są nazywane operatorem wyboru zakresu. Przykładowo `Human::DateOfBirth` odnosi się do zmiennej `DateOfBirth` zadeklarowanej w zakresie klasy `Human`. Z drugiej strony, `::DateOfBirth` odnosi się do innej zmiennej o nazwie `DateOfBirth` w zakresie globalnym.

Kiedy i jak używać konstruktorów?

Konstruktor jest wywoływany zawsze podczas tworzenia obiektu. Z tego powodu konstruktor to doskonałe miejsce na inicjalizację zmiennych będących elementami składowymi klasy, np. liczb całkowitych, wskaźników itd., oraz przypisanie im wartości początkowych. Spójrz ponownie na listing 9.2. Zwróć uwagę, że jeśli zapomniałbyś o metodzie `SetAge()`, liczba całkowita `Human::Age` miałaby przypisaną przypadkową wartość, ponieważ wymieniona zmienna nie byłaby zainicjalizowana (przekonaj się sam i umieść znaki komentarzy na początku wierszy 28. i 31.). W kodzie przedstawionym w listingu 9.3 użyto konstruktorów w celu implementacji lepszej wersji klasy `Human`, w której zmienna `Age` jest inicjalizowana.

Listing 9.3. Użycie konstruktora w celu inicjalizacji zmiennych składowych klasy

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
```

```
 4: class Human
 5: {
 6: private:
 7:    // Dane prywatnych elementów składowych:
 8:    string Name;
 9:    int Age;
10:
11: public:
12:    // Konstruktor.
13:    Human()
14:    {
15:       Age = 0; // Inicjalizacja zmiennej, aby nie miała losowej wartości.
16:       cout << "Utworzono egzemplarz klasy Human" << endl;
17:    }
18:
19:    void SetName (string HumansName)
20:    {
21:       Name = HumansName;
22:    }
23:
24:    void SetAge(int HumansAge)
25:    {
26:       Age = HumansAge;
27:    }
28:
29:    void IntroduceSelf()
30:    {
31:       cout << "Mam na imię " + Name << " i mam ";
32:       cout << Age << " lat " << endl;
33:    }
34: };
35:
36: int main()
37: {
38:    Human FirstMan;
39:    FirstMan.SetName("Adam");
40:    FirstMan.SetAge(30);
41:
42:    Human FirstWoman;
43:    FirstWoman.SetName("Ewa");
44:    FirstWoman.SetAge (28);
45:
46:    FirstMan.IntroduceSelf();
47:    FirstWoman.IntroduceSelf();
48: }
```

Wynik ▼

```
Utworzono egzemplarz klasy Human
Utworzono egzemplarz klasy Human
Mam na imię Adam i mam 30 lat
Mam na imię Ewa i mam 28 lat
```

Analiza ▼

W porównaniu do danych wyjściowych listingu 9.1, w powyższych znajdują się dwa dodatkowe wiersze komunikatów. Spójrz na funkcję `main()` zdefiniowaną w wierszach od 36. do 48. Jak możesz się przekonać, wspomniane dwa wiersze komunikatów zostały wygenerowane pośrednio w trakcie tworzenia (konstruowania) dwóch obiektów `FirstMan` (wiersz 38.) i `FirstWoman` (wiersz 42.). Ponieważ te dwa obiekty są typu klasy `Human`, podczas ich tworzenia następuje automatyczne wywołanie konstruktora klasy `Human` zdefiniowanego w wierszach od 13. do 17. W konstruktorze znajduje się polecenie `cout` generujące komunikat umieszczony w danych wyjściowych. Zauważ, że konstruktor inicjalizuje liczbę całkowitą `Age` wraz z wartością zero. Jeżeli zapomnisz o przypisaniu wartości zmiennej `Age` nowo utworzonego obiektu, nie musisz się martwić, ponieważ konstruktor zadbał, aby wymieniona zmienna nie miała przypadkowej wartości. Wprawdzie taka przypadkowa wartość może wyglądać jak prawidłowa, ale konstruktor przypisuje wartość zero oznaczającą niepowodzenie w ustawieniu wartości atrybutu `Human::Age`.

Uwaga

> Konstruktor, który może być wywołany bez argumentu, nosi nazwę konstruktora domyślnego. Utworzenie konstruktora domyślnego jest opcjonalne.
>
> Jeżeli nie przygotujesz żadnego konstruktora, jak miało to miejsce w listingu 9.1, kompilator utworzy go za Ciebie. Taki konstruktor będzie tworzył elementy składowe, ale nie przeprowadzi inicjalizacji typów POD, takich jak `int`.

Przeciążanie konstruktorów

Podobnie jak funkcje, także konstruktory mogą być przeciążane. Istnieje możliwość zbudowania konstruktora wymagającego użycia parametru podczas tworzenia egzemplarza `Human`, np.:

```
class Human
{
public:
   Human()
```

```
    {
        // Miejsce na kod domyślnego konstruktora.
    }

    Human(string HumansName)
    {
        // Miejsce na kod przeciążonego konstruktora.
    }
};
```

Aplikacja wykorzystująca przeciążony konstruktor została przedstawiona w listingu 9.4. Program tworzy obiekt klasy `Human`, w trakcie tego procesu przypisuje mu wartość podaną w parametrze.

Listing 9.4. Klasa Human z wieloma konstruktorami

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:     // Dane prywatnych elementów składowych:
 8:     string Name;
 9:     int Age;
10:
11: public:
12:     // Konstruktor.
13:     Human()
14:     {
15:         Age = 0; // Inicjalizacja zmiennej, aby nie miała losowej wartości.
16:         cout << "Konstruktor domyślny tworzy egzemplarz klasy Human" <<
            ↪endl;
17:     }
18:
19:     // Przeciążony konstruktor pobierający ciąg tekstowy Name.
20:     Human(string HumansName)
21:     {
22:         Name = HumansName;
23:         Age = 0; // Inicjalizacja zmiennej, aby nie miała losowej wartości.
24:         cout << "Konstruktor przeciążony tworzy obiekt " << Name << endl;
25:     }
26:
27:     // Przeciążony konstruktor pobierający ciąg tekstowy Name i liczbę Age.
28:     Human(string HumansName, int HumansAge)
29:     {
30:         Name = HumansName;
31:         Age = HumansAge;
```

```
32:       cout << "Konstruktor przeciążony tworzy obiekt ";
33:       cout << Name << " w wieku " << Age << " lat" << endl;
34:    }
35:
36:    void SetName (string HumansName)
37:    {
38:       Name = HumansName;
39:    }
40:
41:    void SetAge(int HumansAge)
42:    {
43:       Age = HumansAge;
44:    }
45:
46:    void IntroduceSelf()
47:    {
48:       cout << "Mam na imię " + Name << " i mam ";
49:      cout << Age << " lat" << endl;
50:    }
51: };
52:
53: int main()
54: {
55:    Human FirstMan; // Użycie konstruktora domyślnego.
56:    FirstMan.SetName("Adam");
57:    FirstMan.SetAge(30);
58:
59:    Human FirstWoman ("Ewa"); // Użycie konstruktora przeciążonego.
60:    FirstWoman.SetAge (28);
61:
62:    Human FirstChild ("Róża", 5);
63:
64:    FirstMan.IntroduceSelf();
65:    FirstWoman.IntroduceSelf();
66:    FirstChild.IntroduceSelf();
67: }
```

Wynik ▼

```
Konstruktor domyślny tworzy egzemplarz klasy Human
Konstruktor przeciążony tworzy obiekt Ewa
Konstruktor przeciążony tworzy obiekt Róża w wieku 5 lat
Mam na imię Adam i mam 30 lat
Mam na imię Ewa i mam 28 lat
Mam na imię Róża i mam 5 lat
```

Analiza ▼

Obiekt Adam został utworzony za pomocą konstruktora domyślnego. Z kolei obiekt Ewa powstał przy użyciu konstruktora przeciążonego akceptującego zmienną typu string jako parametr przypisywany atrybutowi Human::Name. Natomiast ostatni obiekt — Róża — został zbudowany z wykorzystaniem konstruktora przeciążonego pobierającego zmienne typu string i int jako parametry, przy czym wartość zmiennej int jest przypisywana atrybutowi Human::Age. W omówionym programie pokazano, jak użyteczne mogą być konstruktory przeciążone i jak pomagają w inicjalizacji zmiennych.

> Możesz zdecydować o niezaimplementowaniu konstruktora domyślnego, aby w ten sposób wymusić utworzenie egzemplarza obiektu z określonymi parametrami minimalnymi.

Wskazówka

Klasa bez konstruktora domyślnego

W listingu 9.5 przedstawiono przykład klasy Human bez konstruktora domyślnego, co wymusza podanie wartości parametrów Name i Age w celu utworzenia obiektu.

Listing 9.5. Klasa z przeciążonym konstruktorem i bez konstruktora domyślnego

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:     // Dane prywatnych elementów składowych:
 8:     string Name;
 9:     int Age;
10:
11: public:
12:
13:     // Przeciążony konstruktor (brak konstruktora domyślnego).
14:     Human(string HumansName, int HumansAge)
15:     {
16:        Name = HumansName;
17:        Age = HumansAge;
18:        cout << "Konstruktor przeciążony tworzy obiekt " << Name;
19:        cout << " w wieku " << Age << endl;
```

```
20:    }
21:
22:    void IntroduceSelf()
23:    {
24:       cout << "Mam na imię " + Name << " i mam ";
25:       cout << Age << " lat" << endl;
26:    }
27: };
28:
29: int main()
30: {
31:    // Usuń znak komentarza z początku poprzedniego wiersza, aby spróbować użyć
      ↪konstruktora domyślnego.
32:    // Human FirstMan;
33:
34:    Human FirstMan("Adam", 30);
35:    Human FirstWoman("Ewa", 28);
36:
37:    FirstMan.IntroduceSelf();
38:    FirstWoman.IntroduceSelf();
39: }
```

Wynik ▼

```
Konstruktor przeciążony tworzy obiekt Adam w wieku 30 lat
Konstruktor przeciążony tworzy obiekt Ewa w wieku 28 lat
Mam na imię Adam i mam 30 lat
Mam na imię Ewa i mam 28 lat
```

Analiza ▼

Uwagę należy zwrócić na wiersz 32. w funkcji `main()`. Polecenie jest bardzo podobne do tworzącego obiekt `FirstMan` w listingu 9.3. Jeżeli jednak usuniesz znak komentarza na początku wymienionego wiersza, kompilacja zakończy się niepowodzeniem z komunikatem błędu podobnym do: `Błąd: 'Human' : brak odpowiedniego konstruktora domyślnego`. W omawianym przykładzie klasa `Human` ma tylko jeden konstruktor, który pobiera parametry danych wejściowych (typu `string` i `int`), jak pokazano w wierszu 14. W klasie nie ma konstruktora domyślnego, a ponieważ istnieje konstruktor przeciążony, kompilator C++ nie generuje domyślnego.

W omówionym przykładzie pokazano możliwość przygotowywania klas wymuszających podanie określonych parametrów (tutaj `Name` i `Age`) w celu utworzenia obiektów klasy `Human`. Kod w listingu 9.5 pokazuje również

możliwość utworzenia obiektu klasy Human po podaniu parametru Name w chwili tworzenia i braku możliwości jego zmiany po utworzeniu. Wymieniony atrybut został w klasie Human zdefiniowany jako zmienna prywatna o nazwie Name i typie string, a tym samym nie ma do niej dostępu z poziomu funkcji main() lub innego elementu, który nie jest elementem składowym klasy Human. Innymi słowy, użytkownik klasy Human jest zmuszony przez przeciążony konstruktor do podania wartości atrybutów Name i Age dla każdego tworzonego obiektu i nie ma możliwości późniejszej zmiany wartości atrybutu Name. Takie rozwiązanie nie jest zbyt użyteczne w rzeczywistych aplikacjach. Człowiek imię nadane przez rodziców po urodzeniu zawsze może zmienić, ale (poza nim samym) nikt inny nie może tego zrobić.

Parametry konstruktora wraz z wartościami domyślnymi

Konstruktory, podobnie jak funkcje, mogą mieć parametry z wartościami domyślnymi. W przedstawionym poniżej kodzie znajduje się nieco zmodyfikowana wersja konstruktora z listingu 9.5, ale parametrowi Age (wiersz 14. w tym listingu) została przypisana wartość domyślna 25.

```
class Human
{
private:
   // Dane prywatnych elementów składowych:
   string Name;
   int Age;

public:
   // Przeciążony konstruktor (brak konstruktora domyślnego).
   Human(string HumansName, int HumansAge = 25)
   {
      Name = HumansName;
      Age = HumansAge;
      cout << "Konstruktor przeciążony tworzy obiekt " << Name;
      cout << " w wieku " << Age << endl;
   }

// Inne elementy składowe.
};
```

Egzemplarz tego rodzaju klasy można utworzyć przy użyciu poniższej składni:

```
Human Adam("Adam"); // Przypisanie Adam.Age wartości domyślnej wynoszącej 25.
Human Ewa("Ewa", 18); // Przypisanie Ewa.Age podanej wartości 18.
```

Uwaga

Zwróć uwagę, że konstruktor domyślny jest tworzony bez argumentów i niekoniecznie musi pobierać parametry. Poniższy konstruktor z dwoma parametrami wraz z przypisanymi wartościami domyślnymi jest konstruktorem domyślnym:

```
class Human
{
private:
   // Dane prywatnych elementów składowych:
   string Name;
   int Age;

public:
   // Zwróć uwagę na wartości domyślne dla dwóch parametrów danych wejściowych.
   Human(string HumansName = "Adam", int HumansAge = 25)
   {
      Name = HumansName;
      Age = HumansAge;
      cout << "Konstruktor przeciążony tworzy obiekt " << Name;
      cout << " w wieku " << Age << endl;
   }
};
```

Z tego powodu egzemplarz klasy Human nadal może być utworzony, mimo braku podanych argumentów:

```
Human Adam; // Obiekt Human wraz z wartościami domyślnymi dla Name (Adam)
            // i Age (25).
```

Konstruktory wraz z listami inicjalizacyjnymi

Przekonałeś się, że konstruktory są użyteczne podczas inicjalizacji zmiennych. Innym sposobem inicjalizacji elementów składowych jest użycie tzw. list inicjalizacyjnych. Poniżej przedstawiono wariant pobierającego dwa parametry konstruktora z listingu 9.5, ale przygotowanego do użycia list inicjalizacyjnych:

```
class Human
{
private:
   string Name;
   int Age;

public:
   // Konstruktor pobiera dwa parametry w celi inicjalizacji elementów składowych Age i Name.
   Human(string InputName, int InputAge)
      :Name(InputName), Age(InputAge)
   {
      cout << "Utworzono obiekt Human o imieniu" << Name;
```

```
      cout << ", " << Age << " lat" << endl;
   }
// Inne elementy składowe klasy.
};
```

Na listę inicjalizacyjną wskazuje dwukropek, po którym znajduje się ujęta w nawias deklaracja parametru, a następnie zmienna składowa i przypisywana jej wartość. Ta wartość inicjalizacyjna może być parametrem, takim jak `InputName`, lub nawet na stałe określoną wartością. Listy inicjalizacyjne są użyteczne podczas wywoływania konstruktorów klas bazowych wraz z określonymi argumentami, co zostanie omówione w lekcji 10., zatytułowanej „Dziedziczenie".

Przykład klasy `Human` zawierającej konstruktor domyślny, wraz parametrami z przypisanymi wartościami domyślnymi i używający list inicjalizacyjnych, został przedstawiony w listingu 9.6.

Listing 9.6. Konstruktor domyślny akceptujący parametry z wartościami domyślnymi służącymi do ustawienia elementów składowych za pomocą list inicjalizacyjnych

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:    int Age;
 8:    string Name;
 9:
10: public:
11:    Human(string InputName = "Adam", int InputAge = 25)
12:         :Name(InputName), Age(InputAge)
13:    {
14:       cout << "Utworzono obiekt Human o imieniu " << Name;
15:       cout << ", " << Age << " lat" << endl;
16:    }
17: };
18:
19: int main()
20: {
21:    Human FirstMan;
22:    Human FirstWoman("Ewa", 18);
23:
24:    return 0;
25: }
```

Wynik ▼

```
Utworzono obiekt Human o imieniu Adam, 25 lat
Utworzono obiekt Human o imieniu Ewa, 18 lat
```

Analiza ▼

Konstruktor wraz z listami inicjalizacyjnymi znajduje się w wierszach od 11. do 16., w których możesz również dostrzec parametry mające przypisane wartości domyślne: `Adam` dla `Name` i `25` dla `Age`. Dlatego też po wywołaniu metody `FirstMan()` w wierszu 21. następuje utworzenie egzemplarza klasy `Human`, którego elementom składowym zostają automatycznie przypisane wartości domyślne. Natomiast metoda `FirstWoman()` jest wywoływana wraz z wyraźnie podanymi wartościami parametrów `Name` i `Age` (patrz wiersz 22.).

Destruktor

Destruktory, podobnie jak konstruktory, są funkcjami specjalnymi. W przeciwieństwie do konstruktorów, destruktory są wywoływane automatycznie w chwili usuwania obiektów.

Deklarowanie i implementowanie destruktora

Destruktor wygląda jak funkcja o nazwie takiej samej jak nazwa klasy, ale poprzedzonej tyldą (~). Dlatego też klasa `Human` ma destruktor zadeklarowany w przedstawiony poniżej sposób:

```
class Human
{
   ~Human(); // Deklaracja destruktora.
};
```

Destruktor może być zaimplementowany na miejscu lub na zewnątrz poza deklaracją klasy. Implementacja (czyli definicja) destruktora w klasie przedstawia się następująco:

```
class Human
{
public:
   ~Human()
   {
      // Miejsce na kod destruktora.
   }
};
```

Z kolei wariant definicji destruktora na zewnątrz (poza deklaracją klasy) przedstawia się następująco:

```
class Human
{
public:
   ~Human(); // Deklaracja destruktora.
};

// Definicja destruktora (implementacja).
Human::~Human()
{
   // Miejsce na kod destruktora.
}
```

Jak możesz się przekonać, deklaracja destruktora tylko nieznacznie różni się od konstruktora z powodu znaku tyldy umieszczonego w nazwie. Jednak rola destruktora jest zupełnie inna.

Kiedy i jak używać destruktorów?

Destruktory są wywoływane zawsze wtedy, gdy obiekt klasy wykracza poza zakres lub w chwili niszczenia obiektu przy użyciu operatora `delete`. Z tego powodu destruktor to idealne miejsce na umieszczenie poleceń odpowiedzialnych za zerowanie zmiennych i dynamiczne zwalnianie pamięci oraz innych zasobów.

W książce nieustannie zachęcam do używania obiektu `std::string` zamiast bufora `char` w stylu C, ponieważ zyskujesz możliwość zarządzania alokacją pamięci w odpowiadający Ci sposób. Obiekt `std::string` i inne tego rodzaju narzędzia są obiektami klas, które starają się w pełni wykorzystać zalety konstruktorów i destruktorów (poza operatorami, które poznasz w lekcji 12., zatytułowanej „Typy operatorów i ich przeciążanie"). Przeanalizujmy przykładową klasę `MyString` przedstawioną w listingu 9.7, która w konstruktorze alokuje pamięć dla ciągu tekstowego, natomiast w destruktorze zwalnia zaalokowaną wcześniej pamięć.

Listing 9.7. Prosta klasa, która hermetyzuje bufor w stylu C, aby zapewnić zwolnienie pamięci przy użyciu destruktora

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
```

```
 5: private:
 6:    char* Buffer;
 7:
 8: public:
 9:    // Konstruktor.
10:    MyString(const char* InitialInput)
11:    {
12:       if(InitialInput != NULL)
13:       {
14:          Buffer = new char [strlen(InitialInput) + 1];
15:          strcpy(Buffer, InitialInput);
16:       }
17:       else
18:          Buffer = NULL;
19:    }
20:    // Destruktor: wyczyszczenie bufora zaalokowanego w konstruktorze.
21:    ~MyString()
22:    {
23:       cout << "Wywołanie destruktora i wyczyszczenie bufora" << endl;
24:       if (Buffer != NULL)
25:          delete [] Buffer;
26:    }
27:
28:    int GetLength()
29:    {
30:       return strlen(Buffer);
31:    }
32:
33:    const char* GetString()
34:    {
35:        return Buffer;
36:    }
37: }; // Koniec klasy MyString.
38:
39: int main()
40: {
41:    MyString SayHello("Witaj z klasy String");
42:    cout << "Bufor w klasie MyString zawiera " << SayHello.GetLength();
43:    cout << " znaków" << endl;
44:
45:    cout << "Bufor zawiera: ";
46:    cout << "Bufor zawiera: " << SayHello.GetString() << endl;
47: }
```

Wynik ▼

```
Bufor w klasie MyString zawiera 20 znaków
Bufor zawiera: Witaj z klasy String
Wywołanie destruktora i wyczyszczenie bufora
```

Analiza ▼

Przedstawiona klasa praktycznie hermetyzuje w obiekcie `MyString::Buffer` ciąg tekstowy w stylu C i zwalnia użytkownika klasy z wykonania zadania dotyczącego alokacji pamięci. Zwolnienie pamięci następuje automatycznie za każdym razem, gdy musisz użyć ciągu tekstowego. Najbardziej interesujące fragmenty to kod konstruktora `MyString()` znajdujący się w wierszach od 10. do 19. oraz kod destruktora `~MyString()` w wierszach od 21. do 26. Konstruktor wymusza utworzenie egzemplarza z użyciem danych wejściowych w postaci ciągu tekstowego podawanego przez parametr. Następnie wspomniany ciąg tekstowy jest kopiowany do bufora w stylu C po zaalokowaniu dla niego pamięci w wierszu 14., w którym funkcja `strlen()` określa wielkość podanego ciągu tekstowego. Z kolei w wierszu 15. funkcja `strcpy()` kopiuje dane wejściowe w postaci ciągu tekstowego do nowo zaalokowanej pamięci. Gdy użytkownik klasy przekaże wartość `null` jako `InitialInput`, obiekt `MyString::Buffer` będzie zainicjalizowany również z wartością `null` (aby wskaźnik nie miał przypisanej przypadkowej wartości, co jest niebezpieczne podczas używania adresów w pamięci). Kod destruktora spełnia swoją rolę i gwarantuje, że pamięć zaalokowana przez konstruktor zostanie zwrócona do systemu. Kod sprawdza, czy wartość `MyString::Buffer` jest inna niż `null` i wtedy przeprowadza operację `delete[]` będącą odpowiednikiem użycia operatora `new` w konstruktorze. Zauważ, że w żadnym miejscu w funkcji `main()` nie następuje użycie operatora `new` lub `delete`. Klasa zapewnia więc doskonałą abstrakcję i gwarantuje prawidłowe zarządzanie pamięcią. Destruktor `~MyString` jest automatycznie wywoływany po zakończeniu działania funkcji `main()` i — jak zademonstrowano w danych wyjściowych programu — wykonuje polecenia `cout`.

Klasy używające destruktorów lepiej obsługują ciągi tekstowe. W lekcji 26., zatytułowanej „Sprytne wskaźniki”, przekonasz się, że destruktory odgrywają krytyczną rolę podczas inteligentnego używania wskaźników.

Uwaga

Destruktory nie mogą być przeciążane, a klasa może zawierać tylko jeden destruktor. Jeżeli zapomnisz o implementacji destruktora, kompilator utworzy podstawowy destruktor, czyli pusty — taki destruktor nie zwalnia dynamicznie alokowanej pamięci.

Konstruktor kopiujący

W lekcji 7., zatytułowanej „Funkcje", dowiedziałeś się, że argumenty przekazywane funkcji, takiej jak przedstawiona w listingu 7.1 funkcja `Area()`, są kopiowane:

```
double Area(double InputRadius);
```

Argument przekazywany jako parametr `InputRadius` jest kopiowany w trakcie wywołania funkcji `Area()`. Ta reguła dotyczy również obiektów i egzemplarzy klas.

Kopiowanie płytkie i związane z tym problemy

Klasy, takie jak `MyString` przedstawiona w listingu 9.7, zawierają element składowy w postaci wskaźnika prowadzącego do dynamicznie zaalokowanej pamięci, alokowanej w konstruktorze przy użyciu operatora `new` i zwalnianej w destruktorze z wykorzystaniem operatora `delete[]`. Podczas kopiowania obiektu tego rodzaju klasy kopiowany jest również wspomniany wskaźnik, ale nie wskazywany bufor. Dlatego też oba obiekty prowadzą do tego samego bufora dynamicznie zaalokowanego w pamięci. To nosi nazwę płytkiej kopii i stanowi zagrożenie stabilności działania programu, co przedstawiono w listingu 9.8.

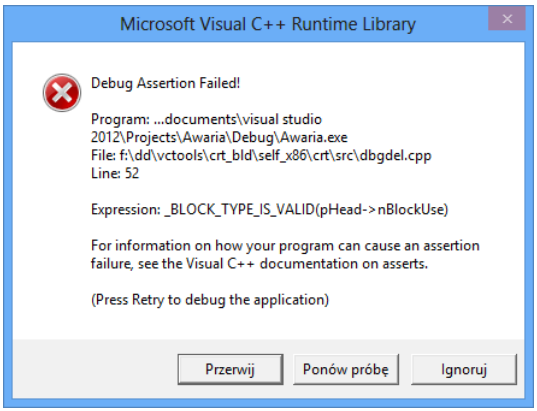
Listing 9.8. Problem związany z przekazywaniem za pomocą wartości obiektów klasy, takiej jak MyString

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:    char* Buffer;
 7:
 8: public:
 9:    // Konstruktor.
10:    MyString(const char* InitialInput)
11:    {
12:       if(InitialInput != NULL)
13:       {
14:          Buffer = new char [strlen(InitialInput) + 1];
15:          strcpy(Buffer, InitialInput);
16:       }
17:       else
```

```
18:          Buffer = NULL;
19:    }
20:
21:    // Destruktor.
22:    ~MyString()
23:    {
24:       cout << "Wywołanie destruktora i wyczyszczenie bufora" << endl;
25:       if (Buffer != NULL)
26:          delete [] Buffer;
27:    }
28:
29:    int GetLength()
30:    {
31:       return strlen(Buffer);
32:    }
33:
34:    const char* GetString()
35:    {
36:       return Buffer;
37:    }
38: };
39:
40: void UseMyString(MyString Input)
41: {
42:    cout << "Bufor w klasie MyString zawiera " << Input.GetLength();
43:    cout << " znaków" << endl;
44:
45:    cout << "Bufor zawiera: " << Input.GetString() << endl;
46:    return;
47: }
48:
49: int main()
50: {
51:    MyString SayHello("Witaj z klasy String");
52:
53:    // Przekazanie obiektu SayHello funkcji jako parametru.
54:    UseMyString(SayHello);
55:
56:    return 0;
57: }
```

Wynik ▼

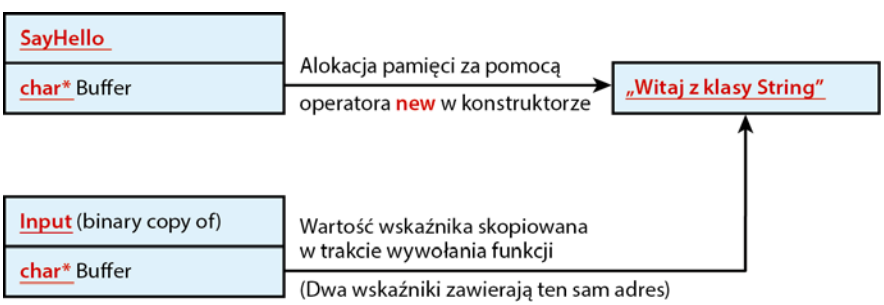
```
Bufor w klasie MyString zawiera 20 znaków
Bufor zawiera: Witaj z klasy String
Wywołanie destruktora i wyczyszczenie bufora
Wywołanie destruktora i wyczyszczenie bufora
<Awaria aplikacji, jak pokazano na rysunku 9.2>
```



RYSUNEK 9.2. Awaria programu spowodowana wykonaniem listingu 9.8 (program został uruchomiony w trybie debugowania w Microsoft Visual Studio)

Analiza ▼

Dlaczego klasa, która doskonale działała w listingu 9.7, powoduje awarię programu w listingu 9.8? Jedyna różnica pomiędzy listingami 9.7 i 9.8 polega na tym, że zadanie metody `SayHello()` klasy `MyString()` zostało delegowane do funkcji `UseMyString()` wywołanej w wierszu 54. Delegacja zadania do wymienionej funkcji spowodowała skopiowanie obiektu `SayHello` do argumentu `Input` używanego w funkcji `UseMyString()`. To jest kopia wygenerowana przez kompilator, ponieważ funkcja została zadeklarowana do pobierania argumentu `Input` jako parametru przez wartość, a nie przez referencję. Kompilator wykonuje więc binarną kopię danych POD. takich jak liczby całkowite, znaki i wskaźniki. Dlatego też wartość zawarta w `SayHello.Buffer` została po prostu skopiowana do atrybutu `Input` — tzn. atrybut `SayHello.Buffer` wskazuje to samo położenie w pamięci, co atrybut `Input.Buffer`. Taka sytuacja została pokazana na rysunku 9.3.



RYSUNEK 9.3. Płytka kopia SayHello w Input po wywołaniu funkcji UseMyString()

Kopia binarna nie powoduje utworzenia głębokiej kopii wskazanego adresu w pamięci i tym samym dwa obiekty klasy `MyString` prowadzą do tego samego adresu w pamięci. Dlatego też po zakończeniu działania funkcji `UseMyString()` zmienna `Input` znajduje się poza zakresem i zostaje usunięta. W tym celu następuje wywołanie destruktora klasy `MyString`, a kod destruktora w wierszu 26. w listingu 9.8 przy użyciu operatora `delete` zwalnia pamięć zaalokowaną dla `Buffer`. Zwróć uwagę, że wywołanie `delete` powoduje unieważnienie w pamięci adresu, do którego prowadzi obiekt `SayHello` w `main()`. Po zakończeniu działania funkcji `main()` obiekt `SayHello` znajdzie się poza zakresem i będzie usunięty. Jednak tym razem wiersz 26. listingu ponowi wywołanie `delete` względem już nieprawidłowego adresu w pamięci (ta pamięć została zwolniona w trakcie usuwania `Input`). Dwukrotne wywołanie `delete` względem tego samego adresu w pamięci prowadzi do awarii aplikacji. Zauważ, że komunikat asercji pokazany na rysunku 9.2 wskazuje na wiersz 52. (czyli 51. w książce, ponieważ tutaj wiersze są numerowane od zera) i głosi, że obiekt `SayHello` nie został prawidłowo zniszczony.

Uwaga

W omawianym przykładzie kompilator nie może zapewnić utworzenia głębokiej kopii, ponieważ w trakcie kompilacji nie zna liczby bajtów, do których prowadzi element składowy `MyString::Buffer` oraz nie zna natury alokacji.

Wykonanie głębokiej kopii przy użyciu konstruktora kopiującego

Konstruktor kopiujący jest specjalnym przeciążonym konstruktorem, który powinien być przygotowany przez twórcę klasy. Kompilator gwarantuje wywołanie konstruktora kopiującego w trakcie każdej operacji kopiowania obiektu klasy, to obejmuje również przekazanie obiektu funkcji przy użyciu wartości.

Składnia deklaracji konstruktora kopiującego dla klasy `MyString` jest następująca:

```
class MyString
{
   MyString(const MyString& CopySource); // Konstruktor kopiujący.
};

MyString::MyString(const MyString& CopySource)
{
   // Kod implementujący konstruktora kopiującego.
}
```

Jako parametr konstruktor kopiujący pobiera przez referencję obiekt ten samej klasy. Wspomniany parametr jest aliasem obiektu źródłowego i uchwytem podczas tworzenia własnego kodu kopiującego (w którym musisz zagwarantować utworzenie głębokiej kopii wszystkich buforów w źródle). Przykład takiego rozwiązania zademonstrowano w listingu 9.9.

Listing 9.9. Zdefiniowanie konstruktora kopiującego w celu zapewnienia utworzenia pełnej kopii dynamicznie alokowanych buforów

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:    char* Buffer;
 7:
 8: public:
 9:    // Konstruktor.
10:    MyString(const char* InitialInput)
11:    {
12:       cout << "Konstruktor: tworzenie nowego egzemplarza MyString" <<
          ↪endl;
13:       if(InitialInput != NULL)
14:       {
15:          Buffer = new char [strlen(InitialInput) + 1];
16:          strcpy(Buffer, InitialInput);
17:
18:          // Wyświetlenie adresu w pamięci wskazywanego przez bufor lokalny.
19:          cout << "Bufor wskazuje adres: 0x" << hex;
20:          cout << (unsigned int*)Buffer << endl;
21:       }
22:       else
23:          Buffer = NULL;
24:    }
25:
26:    // Konstruktor kopiujący.
27:    MyString(const MyString& CopySource)
28:    {
29:       cout << "Konstruktor kopiujący: kopiowanie z egzemplarza MyString" <<
          ↪endl;
30:
31:       if(CopySource.Buffer != NULL)
32:       {
33:          // Zapewnienie utworzenia pełnej kopii, w pierwszej kolejności następuje
             ↪alokacja własnego bufora.
34:          Buffer = new char [strlen(CopySource.Buffer) + 1];
35:
```

```
36:           // Kopiowanie ze źródła do bufora lokalnego.
37:           strcpy(Buffer, CopySource.Buffer);
38:
39:           // Wyświetlenie adresu w pamięci wskazywanego przez bufor lokalny.
40:           cout << "Bufor wskazuje adres: 0x" << hex;
41:           cout << (unsigned int*)Buffer << endl;
42:        }
43:        else
44:           Buffer = NULL;
45:     }
46:
47:     // Destruktor.
48:     ~MyString()
49:     {
50:        cout << "Wywołanie destruktora i wyczyszczenie bufora" << endl;
51:        if (Buffer != NULL)
52:           delete [] Buffer;
53:     }
54:
55:     int GetLength()
56:     {
57:        return strlen(Buffer);
58:     }
59:
60:     const char* GetString()
61:     {
62:        return Buffer;
63:     }
64: };
65:
66: void UseMyString(MyString Input)
67: {
68:    cout << "Bufor w klasie MyString zawiera " << Input.GetLength();
69:    cout << " znaków" << endl;
70:
71:    cout << "Bufor zawiera: " << Input.GetString() << endl;
72:    return;
73: }
74:
75: int main()
76: {
77:    MyString SayHello("Witaj z klasy String");
78:
79:    // Przekazanie obiektu SayHello przez wartość (zostanie utworzona kopia).
80:    UseMyString(SayHello);
81:
82:    return 0;
83: }
```

Wynik ▼

```
Konstruktor: tworzenie nowego egzemplarza MyString
Bufor wskazuje adres: 0x0040DA68
Konstruktor kopiujący: kopiowanie z egzemplarza MyString
Bufor wskazuje adres: 0x0040DAF8
Bufor w klasie MyString zawiera 20 znaków
Bufor zawiera: Witaj z klasy String
Wywołanie destruktora i wyczyszczenie bufora
Wywołanie destruktora i wyczyszczenie bufora
```

Analiza ▼

Większość kodu jest podobna do znajdującego się w listingu 9.8, z wyjątkiem kilku poleceń `cout` dodanych do konstruktora oraz nowego konstruktora kopiującego, którego kod znajduje się w wierszach od 27. do 45. Na początek skoncentrujemy się na funkcji `main()`, która — podobnie jak wcześniej — tworzy w wierszu 77. obiekt `SayHello`. Utworzenie obiektu `SayHello` skutkuje wygenerowaniem pierwszego wiersza danych wyjściowych przez konstruktor `MyString` w wierszu 12. Aby było wygodniej, konstruktor wyświetla także adres w pamięci, do którego prowadzi atrybut `Buffer`. W wierszu 80. funkcja `main()` przekazuje funkcji `UseMyString()` obiekt `SayHello` przez wartość, co powoduje automatyczne wywołanie konstruktora kopiującego, o czym można się przekonać, patrząc na dane wyjściowe programu. Kod w konstruktorze kopiującym jest bardzo podobny do znajdującego się w konstruktorze. Podstawowa zasada działania pozostaje taka sama: sprawdzenie wielkości ciągu tekstowego w stylu C znajdującego się w atrybucie `Buffer` w kopii źródła (wiersz 34.), alokacja pamięci na własny egzemplarz `Buffer`, a następnie użycie funkcji `strcpy()` do skopiowania źródła do celu (wiersz 37.). To nie jest płytka kopia wartości wskaźnika. To jest głęboka kopia, gdzie wartość, do której prowadzi wskaźnik, zostaje skopiowana do nowo zaalokowanego bufora należącego do obiektu (patrz rysunek 9.4).

Dane wyjściowe listingu 9.9 pokazują, że adres w pamięci, do którego prowadzi `Buffer`, jest inny niż w kopii — tzn. dwa obiekty nie prowadzą do tego samego dynamicznie zaalokowanego adresu w pamięci. Dlatego też, gdy funkcja `UseMyString()` zakończy działanie i nastąpi zniszczenie parametru `Input`, kod destruktora wykona operację `delete[]` względem adresu pamięci zaalokowanego w konstruktorze kopiującym i należącym do kopii obiektu.

RYSUNEK 9.4. Ilustracja głębokiej kopii argumentu SayHello do parametru Input podczas wywołania funkcji UseMyString()

W ten sposób pamięć, do której prowadzi obiekt `SayHello` w `main()`, pozostanie nietknięta. Gdy obie funkcje zakończą działanie, wtedy nastąpi zakończone powodzeniem usunięcie ich obiektów bez awarii aplikacji.

Uwaga

Konstruktor kopiujący gwarantuje wykonanie głębokiej kopii w przypadku wywołań funkcji, takich jak poniższe:

```
MyString SayHello("Witaj z klasy String");
UseMyString(SayHello);
```

Zupełnie inaczej wygląda sytuacja, jeśli spróbujesz utworzyć kopię przy użyciu operacji przypisania:

```
MyString overwrite("Kto się tym przejmuje?");
overwrite = SayHello;
```

W takim przypadku nadal tworzona będzie płytka kopia, ze względu na użycie dostarczanego przez kompilator domyślnego operatora przypisania, ponieważ w kodzie nie umieściłeś żadnego. Aby uniknąć tworzenia płytkiej kopii w trakcie operacji przypisania, konieczne jest zaimplementowanie kopiującego operatora przypisania.

Szczegółowe omówienie kopiującego operatora przypisania znajduje się w lekcji 12. W listingu 12.9 znajdziesz poprawioną wersję `MyString` implementującą wspomniany operator:

```
MyString::operator= (const MyString& CopySource)
{
    // Kopiowanie kodu operatora przypisania.
}
```

Ostrzeżenie

Użycie `const` w deklaracji konstruktora kopiującego gwarantuje, że konstruktor kopiujący nie zmodyfikuje obiektu źródłowego, do którego prowadzi.

Ponadto parametr w konstruktorze kopiującym jest z konieczności przekazywany przez referencję. Jeżeli nie byłby przekazany przez referencję, konstruktor sam wywoła kopię przez wartość, a tam samym utworzy płytką kopię danych źródłowych, czyli zrobi dokładnie to, czego chcemy uniknąć.

TAK	NIE
Jeżeli klasa zawiera zwykłe elementy składowe w postaci wskaźników (char* i podobne), zawsze umieszczaj w niej konstruktor kopiujący i kopiujący operator przypisania. Konstruktor kopiowania zawsze twórz z użyciem parametru const, który przez referencję odwołuje się do źródła. Jako elementów składowych używaj klas, takich jak `std::string`, i klas sprytnych wskaźników zamiast zwykłych wskaźników, ponieważ wymienione klasy implementują konstruktory kopiujące, więc unikniesz konieczności ich tworzenia.	Nie używaj zwykłych wskaźników jako elementów składowych klas, o ile nie będzie to absolutnie konieczne.

Uwaga

Klasa `MyString` wraz z elementem składowym w postaci zwykłego wskaźnika `char* Buffer` została użyta jako przykład pokazujący, że trzeba stosować konstruktory kopiujące.

Jeżeli tworzysz klasę, która musi zawierać dane w postaci ciągu tekstowego do przechowywania np. imion, nazw itd., wtedy zamiast `char*` używaj `std::string`. W takim przypadku może nawet nie istnieć potrzeba stosowania konstruktora kopiującego, co wynika z braku zwykłych wskaźników. Po prostu domyślny konstruktor kopiujący wstawiony przez kompilator zagwarantuje wywołanie wszystkich dostępnych konstruktorów kopiujących obiektów elementów składowych, takich jak `std::string`.

C++11

Konstruktory przenoszące pomagają w poprawieniu wydajności

Zdarzają się sytuacje, gdy obiekty są kopiowane automatycznie ze względu na naturę języka i jego potrzeby. Spójrz na przedstawiony poniżej fragment kodu:

```
class MyString
{
   // Umieść tutaj implementację z listingu 9.9.
};
MyString Copy(MyString& Source)
{
   MyString CopyForReturn(Source.GetString());  // Utworzenie kopii.
   return CopyForReturn;  // Zwrot przez wartość wywołuje konstruktor kopiujący.
```

```
}
int main()
{
   MyString sayHello ("Witaj, świecie C++");
   MyString sayHelloAgain(Copy(sayHello)); // Dwukrotne wywołanie konstruktora
                                           // kopiującego.
   return 0;
}
```

Zgodnie z informacjami zawartymi w komentarzach, podczas tworzenia egzemplarza `SayHelloAgain` konstruktor kopiujący będzie wywołany dwukrotnie, a tym samym głęboka kopia zostanie wykonana dwukrotnie z powodu wywołań do funkcji `Copy(sayHello)`, która przez wartość zwraca `MyString`. Jednak ta zwrócona wartość jest tylko tymczasowa i niedostępna poza wymienionym wyrażeniem. Tak więc choć konstruktor kopiujący jest przez kompilator C++ wywołany w dobrej wierze, w rzeczywistości prowadzi do zmniejszenia wydajności, co może być szczególnie widoczne podczas pracy z dynamicznymi tablicami obiektów o znacznej wielkości.

Aby uniknąć spadku wydajności, należy oprócz konstruktora kopiującego umieszczać w klasie także konstruktor przenoszący. Składnia konstruktora przenoszącego została przedstawiona poniżej:

```
// Konstruktor przenoszący.
MyString(MyString&&, MoveSource)
{
   if(MoveSource.Buffer != NULL)
   {
      Buffer = MoveSource.Buffer; // Przejęcie własności.
      MoveSource.Buffer = NULL;   // Ustawienie źródła przeniesienia jako NULL.
   }
}
```

Kiedy istnieje możliwość, kompilator C++11 automatycznie wybiera konstruktor przenoszący podczas „przenoszenia” tymczasowego zasobu, a tym samym unika kroku tworzenia głębokiej kopii. Po zaimplementowaniu konstruktora przenoszącego komentarz w kodzie powinien być zmieniony na poniższy:

```
MyString sayHelloAgain(Copy(sayHello)); // Wywołanie konstruktorów kopiującego
                                        // i przeniesienia.
```

Konstruktor przenoszący jest zwykle implementowany za pomocą przenoszącego operatora przypisania, który będzie szczegółowo omówiony w lekcji 12.
W listingu 12.1 przedstawiono znacznie lepszą wersję klasy `MyString`, która implementuje konstruktor przenoszący oraz przenoszący operator przypisania.

Różne sposoby użycia konstruktorów i destruktora

W tej lekcji poznałeś kilka podstawowych i ważnych koncepcji, m.in. koncepcje konstruktora, destruktora, a także użycie słów kluczowych, takich jak `public` i `private`, do abstrakcji danych oraz metod. Wymienione koncepcje pozwalają na tworzenie klas i zachowanie kontroli nad sposobem ich budowania, kopiowania, niszczenia i udostępniania danych.

Zapoznamy się teraz z kilkoma interesującymi wzorcami, które pomogą w rozwiązaniu wielu ważnych problemów związanych z projektem klasy.

Klasa, której nie można kopiować

Zostałeś poproszony o zbudowanie modelu konstytucji Twojego kraju. Ponieważ krajem rządzi tylko jeden prezydent, utworzenie klasy `President` w przedstawiony poniżej sposób jest ryzykowne.

```
President OurPresident;
DoSomething(OurPresident); // Duplikat utworzony w wyniku przekazania przez wartość.
President clone;
clone = OurPresident; // Duplikat utworzony w wyniku przypisania.
```

Bez wątpliwości tego rodzaju sytuacji należy unikać. Poza tworzeniem modelu określonej konstytucji, być może pracujesz nad systemem operacyjnym i musisz przygotować modele sieci lokalnej, procesora itd. Chcesz uniknąć sytuacji, w której zasoby będą kopiowane. Jeżeli nie zadeklarujesz konstruktora kopiującego, kompilator C++ umieści w klasie domyślny, publiczny konstruktor kopiujący. W ten sposób projekt klasy będzie zniszczony, a implementacja zagrożona. Na szczęście, język C++ zapewnia możliwość rozwiązania tego problemu.

Można zagwarantować, że klasa nie będzie kopiowana przez zadeklarowany prywatny konstruktor kopiujący. W takim przypadku wywołanie funkcji `DoSomething(OurPresident)` spowoduje wygenerowanie błędu w trakcie kompilacji. Aby uniknąć przypisania, należy zadeklarować prywatny operator przypisania:

```
class President
{
private:
   President(const President&); // Prywatna kopia konstruktora.
```

```
    President& operator= (const President&); // Prywatna kopia operatora przypisania.
    // Inne atrybuty.
};
```

Nie ma konieczności implementacji prywatnego konstruktora kopiującego lub operatora przypisania. Po prostu zadeklarowanie ich jako prywatnych jest wystarczające do osiągnięcia celu, czyli uniemożliwienia kopiowania obiektów klasy `President`.

Klasa typu Singleton, która pozwala na istnienie tylko jednego egzemplarza

Omawiana wcześniej klasa `President` jest dobra, ale ma pewną wadę: nie pozwala na tworzenie wielu egzemplarzy prezydentów:

```
President One, Two, Three;
```

Poszczególne obiekty nie pozwalają na kopiowanie, co jest skutkiem zdefiniowania prywatnego konstruktora kopiującego, ale najlepszym rozwiązaniem będzie klasa `President` pozwalająca na utworzenie tylko i wyłącznie jednego obiektu i uniemożliwiająca budowanie kolejnych egzemplarzy. Czas na koncepcję klasy typu Singleton, używającej prywatnych konstruktorów, prywatnego operatora przypisania oraz statycznego (`static`) egzemplarza zmiennej składowej do utworzenia tego obiektu o potężnych możliwościach.

Wskazówka

Użycie słowa kluczowego `static` względem elementu składowego danych klasy gwarantuje, że dany element będzie współdzielony przez wszystkie egzemplarze klasy.

Kiedy słowo kluczowe `static` jest używane w zmiennej lokalnej zadeklarowanej w zakresie funkcji, gwarantuje, że dana zmienna zachowa swoją wartość pomiędzy kolejnymi wywołaniami funkcji.

Z kolei użycie słowa kluczowego `static` względem elementu składowego funkcji — metody — powoduje, że dana metoda będzie współdzielona przez wszystkie egzemplarze klasy.

Słowo kluczowe `static` to kluczowy składnik podczas tworzenia klasy typu Singleton, co zostało przedstawione w listingu 9.10.

Listing 9.10. Klasa President typu Singleton, która uniemożliwia kopiowanie, przypisywanie i tworzenie wielu egzemplarzy

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class President
 5: {
 6: private:
 7:    // Prywatny konstruktor domyślny (jego wywołanie z zewnątrz jest zabronione).
 8:    President() {};
 9:
10:    //  Prywatny konstruktor kopiujący (uniemożliwia tworzenie kopii).
11:    President(const President&);
12:
13:    // Prywatny operator przypisania (uniemożliwia przypisanie).
14:    const President& operator=(const President&);
15:
16:    // Dane elementu składowego: imię i nazwisko prezydenta.
17:    string Name;
18:
19: public:
20:    // Kontrolowane tworzenie egzemplarza.
21:    static President& GetInstance()
22:    {
23:       // Obiekty statyczne są tworzone jednokrotnie.
24:       static President OnlyInstance;
25:
26:       return OnlyInstance;
27:    }
28:
29:    // Metody publiczne.
30:    string GetName()
31:    {
32:       return Name;
33:    }
34:
35:    void SetName(string InputName)
36:    {
37:       Name = InputName;
38:    }
39: };
40:
41: int main()
42: {
43:    President& OnlyPresident = President::GetInstance();
44:    OnlyPresident.SetName("Abraham Lincoln");
45:
```

```
46:     // Usuń znaki komentarza na początku poniższych wierszy, aby przekonać się
        ↪o niepowodzeniu kompilacji.
47:     // President Second; // Nie można uzyskać dostępu do konstruktora.
48:     // President* Third= new President(); // Nie można uzyskać dostępu do konstruktora.
49:     // President Fourth = OnlyPresident; // Nie można uzyskać dostępu do konstruktora
        ↪kopiującego.
50:     // OnlyPresident = President::GetInstance(); // Nie można uzyskać dostępu do operatora =.
51:
52:     cout << "Imię i nazwisko prezydenta: ";
53:     cout << President::GetInstance().GetName() << endl;
54:
55:     return 0;
56: }
```

Wynik ▼

```
Imię i nazwisko prezydenta: Abraham Lincoln
```

Analiza ▼

Jeśli spojrzysz na funkcję `main()`, zobaczysz, że składa się z niewielkiej ilości wierszy kodu, a kilka z nich ma na początku umieszczone znaki komentarza — te wiersze pokazują wszystkie niedziałające kombinacje w tworzeniu nowych egzemplarzy lub kopii klasy `President`. Przeanalizujmy je po kolei:

```
47:     // President Second; // Nie można uzyskać dostępu do konstruktora.
48:     // President* Third= new President(); // Nie można uzyskać dostępu do konstruktora.
```

W wierszach 47. i 48. mamy próbę utworzenia obiektu na (odpowiednio) stosie i stercie za pomocą domyślnego konstruktora, który jednak pozostaje niedostępny z powodu zadeklarowania go w wierszu 8. jako konstruktora prywatnego.

```
49:     // President Fourth = OnlyPresident; // Nie można uzyskać dostępu do konstruktora
        ↪kopiującego.
```

W wierszu 49. następuje próba utworzenia kopii istniejącego obiektu przy użyciu konstruktora kopiującego (przypisanie w trakcie tworzenia powoduje wywołanie konstruktora kopiującego). Wymieniony konstruktor jest niedostępny w funkcji `main()`, ponieważ w wierszu 11. został zadeklarowany jako prywatny.

```
50:     // OnlyPresident = President::GetInstance(); // Nie można uzyskać dostępu do operatora =.
```

Wiersz 50. pokazuje próbę utworzenia kopii przez przypisanie, co się nie udaje, ponieważ operator przypisania został w wierszu 14. zadeklarowany jako

prywatny. Dlatego też funkcja `main()` nigdy nie będzie mogła utworzyć egzemplarza klasy `President` i jedyną pozostałą opcję przedstawiono w wierszu 43. — użycie statycznej funkcji `GetInstance()` w celu pobrania egzemplarza klasy `President`. Ponieważ `GetInstance()` to statyczny element składowy klasy, przypomina funkcję globalną, którą można wywoływać bez posiadania obiektu jako uchwytu. Implementacja funkcji `GetInstance()` znajduje się w wierszach od 21. do 27., użyto statycznej zmiennej `OnlyInstance` w celu zagwarantowania istnienia tylko i wyłącznie jednego egzemplarza klasy `President`. Aby to lepiej zrozumieć, wyobraź sobie, że wiersz 24. jest wykonywany tylko jednokrotnie (inicjalizacja statyczna), a tym samym funkcja `GetInstance()` zwraca tylko jeden dostępny egzemplarz klasy `President`, niezależnie od tego, jak ten egzemplarz jest używany (patrz wiersze 43. i 53. w funkcji `main()`).

Ostrzeżenie
Ostrzeżenie

> Mając na uwadze przyszłą rozbudowę aplikacji i jej funkcji, wzorca Singleton używaj tylko wtedy, gdy jest to absolutnie potrzebne. Pamiętaj, że każda funkcja ograniczająca tworzenie wielu egzemplarzy może stać się architekturalnym wąskim gardłem, gdy będzie trzeba utworzyć wiele egzemplarzy klasy.
>
> Jeśli np. Twój projekt zostanie rozbudowany z modelu narodu do ONZ zrzeszającego obecnie ponad 190 krajów, z których każdy ma swojego przywódcę (takiego jak prezydent), klasa `President` w obecnej postaci niewątpliwie będzie przeszkodą architekturalną w aplikacji.

Klasy, których egzemplarze nie mogą być tworzone na stosie

Przestrzeń dostępna na stosie najczęściej jest ograniczona. Jeżeli budujesz bazę danych, która w wewnętrznych strukturach może zawierać gigabajty danych, wtedy musisz się upewnić, że klient tego rodzaju klasy nie będzie mógł tworzyć jej egzemplarzy na stosie. Zamiast tego należy wymusić tworzenie egzemplarzy jedynie na stercie. Kluczowym zadaniem jest zadeklarowanie prywatnego destruktora:

```
class MonsterDB
{
private:
    // Prywatny destruktor.
    ~MonsterDB();
    // Kolekcje rezerwujące ogromne ilości danych.
};
```

Podczas próby użycia klasy MonsterDB nie będzie można jej utworzyć w przedstawiony poniżej sposób:

```
int main()
{
   MonsterDB myDatabase; // Błąd kompilacji.
   // Jeszcze więcej kodu.
   return 0;
}
```

Jeżeli utworzenie egzemplarza w powyższy sposób zakończy się powodzeniem, będzie on umieszczony na stosie. Ponieważ kompilator wie, że egzemplarz myDatabase musi być zniszczony po wykroczeniu poza zakres, automatycznie próbuje wywołać destruktor na końcu działania funkcji main(). Jednak wspomniany destruktor został zadeklarowany jako prywatny, pozostaje więc niedostępny, a kompilacja kończy się niepowodzeniem.

Jednak prywatny destruktor nie jest przeszkodą w utworzeniu egzemplarza na stercie:

```
int main()
{
   MonsterDB* myDatabase = new MonsterDB(); // Brak błędu.
   // Jeszcze więcej kodu.
   return 0;
}
```

Jeżeli sądzisz, że w powyższym fragmencie kodu znajduje się wyciek pamięci, masz rację. Ponieważ destruktor nie jest dostępny z poziomu funkcji main(), nie można zwolnić pamięci. Klasa MonsterDB potrzebuje statycznego, publicznego elementu składowego w postaci funkcji odpowiedzialnej za usunięcie egzemplarza — ponieważ funkcja będzie elementem składowym, ma dostęp do destruktora. Rozwiązanie przedstawiono w listingu 9.11.

Listing 9.11. Klasa bazy danych o nazwie MonsterDB, która pozwala na tworzenie obiektów jedynie w wolnej pamięci (przy użyciu operatora new)

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MonsterDB
 4: {
 5: private:
 6:    ~MonsterDB() {}; // Prywatny destruktor.
 7:
```

```
 8: public:
 9:    static void DestroyInstance(MonsterDB* pInstance)
10:    {
11:       // Statyczny element składowy może uzyskać dostęp do prywatnego destruktora.
12:       delete pInstance;
13:    }
14:
15:    // Kilka innych metod.
16: };
17:
18: int main()
19: {
20:    MonsterDB* pMyDatabase = new MonsterDB();
21:
22:    // pMyDatabase -> metody elementu składowego (....);
23:
24:    // Usuń znaki komentarza na początku poniższego wiersza, aby przekonać się
       ↪o niepowodzeniu kompilacji.
25:    // delete pMyDatabase; // Nie można wywołać prywatnego destruktora.
26:
27:    // Użycie statycznego elementu składowego w celu zwolnienia pamięci.
28:    MonsterDB::DestroyInstance(pMyDatabase);
29:
30:    return 0;
31: }
```

Program przedstawiony w listingu 9.11 nie generuje danych wyjściowych.

Analiza ▼

Celem przedstawionego powyżej kodu jest pokazanie, jak klasa niemożliwa do utworzenia na stosie może być przygotowana za pomocą destruktora prywatnego (patrz wiersz 6.) i statycznej funkcji `DestroyInstance()`, której kod znajduje się w wierszach od 9. do 13. Wymieniona funkcja jest używana w `main()`, co widać w wierszu 28.

Wskaźnik this

Ważna koncepcja w języku C++, czyli `this`, to zarezerwowane słowo kluczowe dostępne w zakresie klasy zawierającej adres obiektu. Innymi słowy, wartością `this` jest `&obiekt`. W metodzie składowej klasy podczas wywoływania innej metody składowej kompilator używa wskaźnika `this` jako niejawnego i niewidocznego parametru w wywołaniu funkcji:

```
class Human
{
private:
// Deklaracje prywatnych elementów składowych.
   void Talk (string Statement)
   {
      cout << Statement;
   }
public:
   void IntroduceSelf()
   {
      Talk("Bla bla");
   }
};
```

W powyższym fragmencie kodu metoda `IntroduceSelf()` używa prywatnego elementu składowego `Talk()` do wyświetlenia komunikatu na ekranie. W rzeczywistości kompilator osadza w kodzie wskaźnik `this` w wywołaniu `Talk()`, które przyjmuje postać `Talk(this, "Bla bla")`.

Z perspektywy programistycznej `this` nie ma zbyt wielu zastosowań, z wyjątkiem tych, w których zwykle jest opcjonalne. Przykładowo kod uzyskujący dostęp do atrybutu `Age` przy użyciu `SetAge()`, co przedstawiono w listingu 9.1, można utworzyć także w postaci:

```
void SetAge(int HumansAge)
{
   this->Age = HumansAge;     // Odpowiednik Age = HumansAge.
}
```

Uwaga

Zwróć uwagę, że wskaźnik `this` nie jest przekazywany metodom klasy zadeklarowanym jako statyczne, ponieważ funkcje statyczne nie są połączone z egzemplarzem klasy. Zamiast tego są współdzielone przez wszystkie egzemplarze.

Jeżeli w funkcji statycznej chcesz używać zmiennej egzemplarza, musisz wyraźnie zadeklarować parametr, który użyje do przekazania wskaźnika `this` jako argumentu.

Operator sizeof() dla klasy

Poznałeś podstawy definiowania własnego typu przy użyciu słowa kluczowego `class`, co pozwala na hermetyzację atrybutów danych oraz metod działających ze wspomnianymi danymi. Omówiony w lekcji 3., zatytułowanej „Zmienne i stałe”, operator `sizeof()` pozwala na określenie wyrażonej w bajtach ilości

pamięci wymaganej przez dany typ. Operator ten można zastosować także dla klasy, podaje wówczas sumę bajtów zajmowanych przez wszystkie atrybuty danych znajdujące się w deklaracji klasy. W zależności od używanego kompilatora, operator `sizeof()` może, choć nie musi, uwzględniać dopełnienia granic słowa dla określonych argumentów. Zauważ, że funkcje składowe i ich zmienne lokalne nie odgrywają roli w określaniu danych wyjściowych operatora `sizeof()` dla klasy. Zapoznaj się z kodem przedstawionym w listingu 9.12.

Listing 9.12. Wynik użycia sizeof względem klas i ich egzemplarzy

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:    char* Buffer;
 7:
 8: public:
 9:    // Konstruktor.
10:    MyString(const char* InitialInput)
11:    {
12:       if(InitialInput != NULL)
13:       {
14:          Buffer = new char [strlen(InitialInput) + 1];
15:          strcpy(Buffer, InitialInput);
16:       }
17:       else
18:          Buffer = NULL;
19:    }
20:
21:    // Konstruktor kopiujący.
22:    MyString(const MyString& CopySource)
23:    {
24:       if(CopySource.Buffer != NULL)
25:       {
26:           Buffer = new char [strlen(CopySource.Buffer) + 1];
27:           strcpy(Buffer, CopySource.Buffer);
28:       }
29:       else
30:          Buffer = NULL;
31:    }
32:
33:    ~MyString()
34:    {
35:       if (Buffer != NULL)
36:          delete [] Buffer;
37:    }
```

```
38:
39:    int GetLength()
40:    {
41:       return strlen(Buffer);
42:    }
43:
44:    const char* GetString()
45:    {
46:       return Buffer;
47:    }
48: };
49:
50: class Human
51: {
52: private:
53:    int Age;
54:    bool Gender;
55:    MyString Name;
56:
57: public:
58:    Human(const MyString& InputName, int InputAge, bool InputGender)
59:       : Name(InputName), Age (InputAge), Gender(InputGender) {}
60:
61:    int GetAge ()
62:    {
63:       return Age;
64:    }
65: };
66:
67: int main()
68: {
69:    MyString FirstMan("Adam");
70:    MyString FirstWoman("Eve");
71:
72:    cout << "sizeof(MyString) = " << sizeof(MyString) << endl;
73:    cout << "sizeof(FirstMan) = " << sizeof(FirstMan) << endl;
74:    cout << "sizeof(FirstWoman) = " << sizeof(FirstWoman) << endl;
75:
76:    Human FirstMaleHuman(FirstMan, 25, true);
77:    Human FirstFemaleHuman(FirstWoman, 18, false);
78:
79:    cout << "sizeof(Human) = " << sizeof(Human) << endl;
80:    cout << "sizeof(FirstMaleHuman) = " << sizeof(FirstMaleHuman) <<
      ↪endl;
81:    cout << "sizeof(FirstFemaleHuman) = " << sizeof(FirstFemaleHuman) <<
      ↪endl;
82:
83:    return 0;
84: }
```

Wynik ▼

```
sizeof(MyString) = 4
sizeof(FirstMan) = 4
sizeof(FirstWoman) = 4
sizeof(Human) = 12
sizeof(FirstMaleHuman) = 12
sizeof(FirstFemaleHuman) = 12
```

Analiza ▼

Powyższy przykład jest obszerny, ponieważ zawiera klasę `MyString` (bez większości poleceń wyświetlających komunikaty), której kod został wcześniej przedstawiony w listingu 9.9, oraz wariant klasy `Human` używającej typu `MyString` do przechowywania zmiennej `Name`. Ponadto dodany został parametr `bool` dla zmiennej `Gender`.

Rozpocznijmy od analizy danych wyjściowych. Z danych wynika, że wynik wywołania `sizeof()` względem klasy jest taki sam jak względem obiektu klasy. Dlatego też wielkość `sizeof(MyString)` jest taka sama jak wielkość `sizeof(FirstMan)`, ponieważ liczba bajtów wymaganych przez tę klasę została ustalona na stałe w trakcie kompilacji i jest znana podczas projektowania klasy. Nie powinieneś być zaskoczony, że obiekty `FirstMan` i `FirstWoman` mają taką samą wielkość w bajtach, choć jeden zawiera dane Adama, a drugi Ewy. Jednak wspomniane dane są przechowywane przez `MyString::Buffer` będący wskaźnikiem typu `char*`, którego wielkość pozostaje niezmienna, wynosi cztery bajty (w systemie 32-bitowym) i jest niezależna od wielkości wskazywanych danych.

Próba ręcznego obliczenia wartości `sizeof()` względem klasy `Human` daje wynik 12. W wierszach 52., 53. i 54. widać, że klasa `Human` składa się ze zmiennych typu `int`, `bool` i `MyString`. Po odwołaniu się do listingu 3.4 pokazującego ilość pamięci wymaganą przez wbudowane typy danych wiadomo, że `int` zabiera 4 bajty, `bool` – 1, a `MyString` — 4 w systemie używanym w przykładzie. To w żaden sposób nie sumuje się do 12 bajtów przedstawionych w danych wyjściowych. Większa wartość wynika z dopełnienia słowa oraz innych czynników wpływających na wynik działania operatora `sizeof()`.

Jaka jest różnica pomiędzy strukturą i klasą?

Słowo `struct` jest słowem kluczowym pochodzącym z języka C; z powodów praktycznych jest traktowane przez kompilator C++ bardzo podobnie jak `class`. Wyjątki polegają na akceptacji innych domyślnych specyfikatorów dostępu (`public` i `private`), gdy programista żadnego nie wskaże. Tak więc, jeśli nie zostanie podane inaczej, elementy składowe struktury domyślnie są publiczne (prywatne w klasie), a sama struktura stosuje publiczne dziedziczenie (prywatne w klasie) po klasie bazowej. Szczegółowe omówienie dziedziczenia znajdziesz w lekcji 10.

Wariant struktury dla klasy `Human` z listingu 9.12 będzie przedstawiał się następująco:

```
struct Human
{
   // Konstruktor, domyślnie jest publiczny (ponieważ nie podano żadnych specyfikatorów dostępu).
   Human(const MyString& InputName, int InputAge, bool InputGender)
      : Name(InputName), Age (InputAge), Gender(InputGender) {}

   int GetAge ()
   {
      return Age;
   }

private:
   int Age;
   bool Gender;
   MyString Name;
};
```

Jak możesz się przekonać, struktura `Human` jest bardzo podobna do klasy `Human`, a tworzenie egzemplarza obiektu typu `struct` także przypomina tworzenie obiektu klasy:

```
Human FirstMan("Adam", 25, true); // To jest egzemplarz struktury Human.
```

Deklaracja „przyjaciela” klasy

Klasa nie pozwala na uzyskanie dostępu z zewnątrz do zadeklarowanych jako prywatne elementów składowych, np. danych i metod. Odstępstwo od tej reguły dotyczy klas i funkcji, które przy użyciu słowa kluczowego `friend`

zostały określone jako przyjacielskie. Przykład takiego rozwiązania przedstawiono w listingu 9.13.

Listing 9.13. Użycie słowa kluczowego friend w celu umożliwienia zewnętrznej funkcji DisplayAge() uzyskania dostępu do danych prywatnych elementów składowych

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:    string Name;
 8:    int Age;
 9:
10:    friend void DisplayAge(const Human& Person);
11:
12: public:
13:    Human(string InputName, int InputAge)
14:    {
15:       Name = InputName;
16:       Age = InputAge;
17:    }
18: };
19:
20: void DisplayAge(const Human& Person)
21: {
22:    cout << Person.Age << endl;
23: }
24:
25: int main()
26: {
27:    Human FirstMan("Adam", 25);
28:    cout << "Uzyskanie dostępu do prywatnego elementu składowego Age
       ↪przy użyciu słowa kluczowego friend: ";
29:    DisplayAge(FirstMan);
30:
31:    return 0;
32: }
```

Wynik ▼

```
Uzyskanie dostępu do prywatnego elementu składowego Age przy użyciu słowa
↪kluczowego friend: 25
```

Analiza ▼

Deklaracja znajdująca się w wierszu 10. informuje kompilator, że funkcja `DisplayAge()` w zakresie globalnym ma specjalne uprawnienia dotyczące dostępu do prywatnych elementów składowych klasy `Human`. Jeżeli na początku wiersza 10. umieścisz znak komentarza, przekonasz się, że wiersz 22. listingu uniemożliwi jego kompilację.

Podobnie jak funkcje, także zewnętrzne klasy mogą być określone jako zaufani przyjaciele, co zademonstrowano w listingu 9.14.

Listing 9.14. Użycie słowa kluczowego friend w celu umożliwienia zewnętrznej klasie Utility uzyskania dostępu do danych prywatnych elementów składowych

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:    string Name;
 8:    int Age;
 9:
10:    friend class Utility;
11:
12: public:
13:    Human(string InputName, int InputAge)
14:    {
15:       Name = InputName;
16:       Age = InputAge;
17:    }
18: };
19:
20: class Utility
21: {
22: public:
23:    static void DisplayAge(const Human& Person)
24:    {
25:       cout << Person.Age << endl;
26:    }
27: };
28:
29: int main()
30: {
31:    Human FirstMan("Adam", 25);
```

```
32:    cout << " Uzyskanie dostępu do prywatnego elementu składowego Age
       ↪przy użyciu friend class: ";
33:    Utility::DisplayAge(FirstMan);
34:
35:    return 0;
36: }
```

Wynik ▼

```
Uzyskanie dostępu do prywatnego elementu składowego Age przy użyciu friend
↪class: 25
```

Analiza ▼

Wiersz 10. wskazuje, że klasa `Utility` jest przyjacielem klasy `Human`. W ten sposób wszystkie metody klasy `Utility` mają dostęp do prywatnych elementów składowych i metod klasy `Human`.

Podsumowanie

W tej lekcji przedstawiono jedno z najważniejszych słów kluczowych i jedną z ważniejszych koncepcji w C++, czyli klasę. Dowiedziałeś się, że klasa hermetyzuje elementy składowe danych i funkcje operujące na wspomnianych danych. Przekonałeś się, jak specyfikatory dostępu, m.in. `public` i `private`, pomagają w abstrakcji danych oraz funkcjonalności, która powinna pozostać niewidoczna dla elementów zewnętrznych klasy.

Poznałeś również koncepcję konstruktorów kopiujących w C++ i wiesz, jak standard C++11 pozwala na użycie konstruktorów przeniesienia do wprowadzenia optymalizacji w postaci pozbycia się niechcianego kroku kopiowania. Ponadto zobaczyłeś kilka przypadków specjalnych, w których wszystkie wymienione elementy razem pozwalają na implementację wzorców projektowych, np. Singleton.

Pytania i odpowiedzi

Pytanie: Jaka jest różnica pomiędzy egzemplarzem klasy i obiektem danej klasy?

Odpowiedź: W zasadzie żadna. Podczas tworzenia klasy otrzymujesz egzemplarz, który można nazywać także obiektem.

Pytanie: Który sposób jest lepszy w celu uzyskania dostępu do elementów składowych: operator kropki (`.`) czy operator wskaźnika (`->`)?

Odpowiedź: Jeżeli masz wskaźnik do obiektu, użycie operatora wskaźnika będzie lepszym rozwiązaniem. W przypadku utworzenia obiektu w postaci zmiennej lokalnej na stosie lepszym rozwiązaniem będzie użycie operatora kropki.

Pytanie: Czy zawsze powinienem umieszczać w klasie konstruktor kopiujący?

Odpowiedź: Jeżeli elementy składowe w postaci danych klasy są doskonale przygotowanymi sprytnymi wskaźnikami, klasami ciągu tekstowego lub kontenerami STL, takimi jak `std::vector`, wtedy domyślny konstruktor kopiujący wstawiany przez kompilator będzie wystarczający. Jeśli jednak klasa zawiera elementy składowe w postaci zwykłych wskaźników (takie jak np. tablica dynamiczna tworzona przez `int*` zamiast przez `std::vector<int>`), konieczne jest umieszczenie konstruktora kopiującego, który zapewni utworzenie głębokiej kopii tablicy podczas wywołań funkcji, gdy obiekt jest przekazywany przez wartość.

Pytanie: Moja klasa ma tylko jeden konstruktor, który został zdefiniowany z parametrem wraz z przypisaną mu wartością domyślną. Czy to nadal będzie konstruktor domyślny?

Odpowiedź: Tak. Jeżeli egzemplarz klasy może być utworzony bez argumentów, wtedy mówimy, że klasa zawiera konstruktor domyślny. W klasie może znajdować się tylko jeden konstruktor domyślny.

Pytanie: Dlaczego w niektórych przykładach przedstawionych w lekcji funkcja, taka jak `SetAge()`, jest używana do ustawienia liczby całkowitej `Human::Age`? Dlaczego zmienna `Age` nie może być publiczna, co pozwoli na przypisywanie jej wartości wedle potrzeb?

Odpowiedź: Z technicznego punktu widzenia zdefiniowanie `Human::Age` jako publiczny element składowy będzie sprawdzało się doskonale. Jednak, z punktu widzenia projektu klasy, zapewnienie prywatności danym składowym jest dobrym rozwiązaniem. Funkcje akcesora, takie jak `GetAge()` i `SetAge()`, to dostosowany do własnych potrzeb i skalowalny sposób na uzyskanie dostępu do prywatnych danych. Ponadto pozwalają na przeprowadzenie operacji sprawdzania danych egzemplarza przed ustawieniem bądź wyzerowaniem wartości `Human::Age`.

Pytanie: Dlaczego parametr konstruktora kopiującego pobiera przez referencję kopię źródła?

Odpowiedź: Kompilator oczekuje, że konstruktor kopiujący będzie działał właśnie w taki sposób. Ponadto, jeśli konstruktor kopiujący akceptowałby przekazanie przez wartość kopii źródła, wtedy wywoływałby samego siebie, co oznacza powstanie pętli działającej w nieskończoność.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Kiedy przy użyciu słowa kluczowego `new` tworzę egzemplarz klasy, a kiedy klasę?
2. Moja klasa ma zwykły wskaźnik `int*` zawierający dynamicznie zaalokowaną tablicę liczb całkowitych. Czy operator `sizeof()` będzie wyświetlał różne wartości, w zależności od ilości liczb całkowitych znajdujących się w dynamicznej tablicy?
3. Wszystkie elementy składowe klasy są prywatne, a sama klasa nie zawiera żadnej zadeklarowanej klasy lub funkcji typu `friend`. Kto będzie mógł uzyskać dostęp do wspomnianych elementów składowych?
4. Czy metoda w jednej klasie może wywołać inną?
5. Do czego służy konstruktor?
6. Do czego służy destruktor?

Ćwiczenia

1. **Łowcy błędów:** Co jest nie tak z poniższą deklaracją klasy?

```
Class Human
{
   int Age;
   string Name;
public:
   Human() {}
}
```

2. W jaki sposób użytkownik klasy przedstawionej w ćwiczeniu 1. może uzyskać dostęp do elementu składowego `Human::Age`?
3. Utwórz lepszą wersję klasy przedstawionej w ćwiczeniu 1. Nowa wersja powinna inicjalizować wszystkie parametry za pomocą listy inicjalizacyjnej w konstruktorze.
4. Utwórz klasę o nazwie `Circle` obliczającą pole i obwód okręgu na podstawie promienia podanego klasie jako parametr w chwili tworzenia jej egzemplarza. Wartość Pi powinna znajdować się w prywatnym elemencie składowym w postaci stałej, wartość ta nie powinna być dostępna na zewnątrz.

Lekcja 10

Dziedziczenie

Programowanie zorientowane obiektowo opiera się na czterech podstawowych aspektach: hermetyzacji, abstrakcji, dziedziczeniu i polimorfizmie. Dziedziczenie to oferujący potężne możliwości sposób ponownego użycia atrybutów; jest ono krokiem milowym w kierunku polimorfizmu.

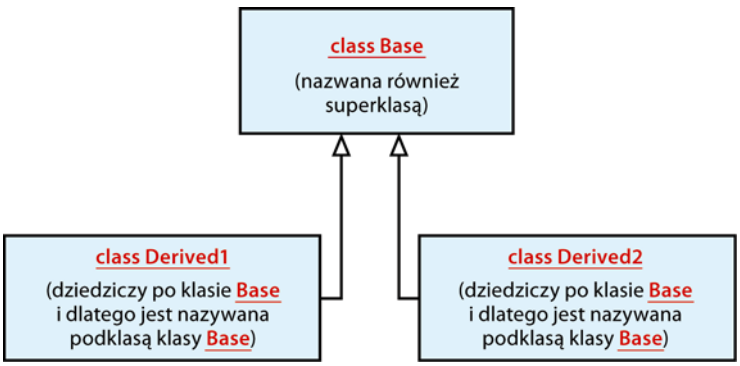
Z tej lekcji dowiesz się:

- **dowiesz się, czym w kontekście programowania jest dziedziczenie,**
- **poznasz składnię dziedziczenia w C++,**
- **poznasz dziedziczenie publiczne, prywatne i chronione,**
- **poznasz dziedziczenie wielokrotne,**
- **poznasz problemy powodowane przez ukrywanie metod klas bazowych.**

Podstawy dziedziczenia

Po swoich przodkach Jan Kowalski dziedziczy przede wszystkim nazwisko, dzięki któremu jest właśnie Kowalskim. Ponadto dziedziczy pewne umiejętności, których nauczyli go rodzice, m.in. rzeźbi w drewnie, czym rodzina Kowalskich zajmuje się od pokoleń. Wymienione atrybuty powodują, że Jan jest potomkiem rodziny Kowalskich.

W żargonie programistycznym bardzo często będziesz spotykał sytuacje, w których komponenty mają podobne atrybuty, różniące się jedynie szczegółami lub zachowaniem. Jednym ze sposobów rozwiązania problemu jest utworzenie klas dla poszczególnych komponentów, przy czym wszystkie klasy implementują wszystkie argumenty, a nawet ponownie implementują te, które są najczęściej używane. Inne rozwiązanie polega na zastosowaniu mechanizmu dziedziczenia pozwalającego na wywodzenie podobnych klas z klasy bazowej implementującej najczęściej używane funkcje. W klasach potomnych nadpisywane będą pewne funkcje w celu implementacji zachowania decydującego o unikalności klasy. Drugie z wymienionych rozwiązań jest preferowane. Witaj w świecie dziedziczenia w programowaniu zorientowanym obiektowo (patrz rysunek 10.1).



RYSUNEK 10.1. Dziedziczenie pomiędzy klasami

Dziedziczenie i pochodzenie

Na rysunku 10.1 pokazano związek zachodzący pomiędzy klasą bazową i jej klasami potomnymi. W tym momencie być może nie potrafisz sobie wyobrazić, czym może być klasa bazowa lub potomna. Spróbuj zrozumieć, że klasa potomna dziedziczy po klasie bazowej, a więc w pewnym sensie jest klasą bazową (podobnie jak Jan jest Kowalski).

Uwaga

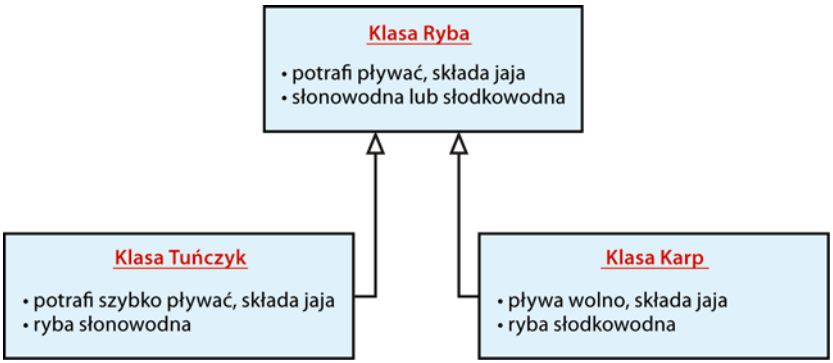
Relacja typu jest-czymś pomiędzy klasą potomną i jej klasą bazową dotyczy jedynie dziedziczenia publicznego. Tę lekcję zaczynam od omówienia dziedziczenia publicznego, aby ułatwić Ci zrozumienie samej koncepcji dziedziczenia i jej najczęściej spotykanej postaci, potem przejdę do dziedziczenia prywatnego lub chronionego.

Aby ułatwić zrozumienie koncepcji, przeanalizujemy klasę `Ptak`. Klasami potomnymi klasy `Ptak` są `Wrona`, `Papuga` lub `Struś`. Klasa `Ptak` będzie definiowała najbardziej podstawowe atrybuty ptaka, czyli pióra, istnienie skrzydeł, składanie jaj i możliwość latania (dla większości ptaków). Klasy potomne, takie jak `Wrona`, `Papuga` lub `Struś`, dziedziczą wymienione atrybuty i dostosowują je do własnych potrzeb (np. klasa `Struś` nie będzie zawierała implementacji metody `Lot()`). W tabeli 10.1 przedstawiono kilka przykładów dziedziczenia publicznego.

Tabela 10.1. Wzięte z życia codziennego przykłady dziedziczenia publicznego

Klasa bazowa	Przykład klasy potomnej
Ryba	Złota rybka, Karp, Tuńczyk (tuńczyk jest rybą)
Ssak	Człowiek, Słoń, Lew, Dziobak (dziobak jest ssakiem)
Ptak	Wrona, Papuga, Struś, Kiwi, Dziobak (dziobak to również ptak)
Figura	Okrąg, Wielokąt (okrąg jest figurą)
Wielokąt	Trójkąt, Ośmiokąt (ośmiokąt jest wielokątem, który jest figurą)

Wymienione w tabeli przykłady pokazują, że jeśli weźmiesz pod uwagę koncepcję programowania zorientowanego obiektowo, to praktycznie wszędzie możesz znaleźć przykłady dziedziczenia. `Ryba` to klasa bazowa dla klasy `Tuńczyk`, ponieważ `Tuńczyk`, podobnie jak i `Karp`, jest `Rybą` i ma wszystkie cechy charakterystyczne ryb, np. zmiennocieplność. Jednak `Tuńczyk` różni się od `Karpia` wyglądem, sposobem pływania oraz tym, że jest rybą słonowodną. Dlatego też `Tuńczyk` i `Karp` dziedziczą pewne wspólne cechy ryb po klasie bazowej `Ryba`, ale, specjalizując pewne atrybuty klasy bazowej, odróżniają się od siebie, co zostało zilustrowane na rysunku 10.2.



RYSUNEK 10.2. Hierarchiczna relacja pomiędzy Tuńczykiem, Karpiem i Rybą

Dziobak potrafi pływać, to zwierzę posiadające cechy ssaków, np. karmi potomstwo mlekiem; ma też cechy ptaków, np. składa jaja, a także ma cechy gadów, np. jest jadowity. Dlatego też klasa Dziobak może zostać przedstawiona jako dziedzicząca po dwóch klasach bazowych: Ssak i Ptak, w ten sposób dziedziczy cechy charakterystyczne ssaków i ptaków. To jest przykład tzw. *dziedziczenia wielokrotnego*, które dokładnie będzie omówione dalej w tej lekcji.

Stosowana w języku C++ składnia pochodzenia

W jaki sposób klasa `Carp` może dziedziczyć po klasie `Fish` lub — bardziej ogólnie — jak klasa `Derived` (potomna) może dziedziczyć po klasie `Base` (bazowej)? Poniżej przedstawiono oferowaną przez język C++ składnię dziedziczenia:

```
// Deklaracja superklasy (klasa bazowa).
class Base
{
    // Elementy składowe klasy bazowej.
};
// Deklaracje klas potomnych.
class Derived: specyfikator-dostępu Base
{
    // Elementy składowe klasy potomnej.
};
```

Użyty specyfikator dostępu może być publiczny (`public`, najczęściej spotykany) dla relacji „klasa potomna jest klasą bazową”, prywatny (`private`) lub chroniony (`protected`) dla relacji „klasa potomna ma klasę bazową”.

Hierarchiczny układ dziedziczenia w przypadku klasy `Carp` będącej klasą potomną klasy `Fish` przedstawia się następująco:

```
class Fish
{
   // Elementy składowe klasy Fish.
};
class Carp:public Fish
{
   // Elementy składowe klasy Carp.
};
```

Kilka słów na temat terminologii

Kiedy będziesz czytał o dziedziczeniu, bardzo często natkniesz się na pojęcia, takie jak *dziedziczyć po* i *wywodzić się z*, które w zasadzie oznaczają dokładnie to samo.

Klasa bazowa jest nazywana również superklasą. Klasa dziedzicząca po klasie bazowej nosi nazwę *klasy potomnej*, choć może być nazywana także *podklasą*.

Przykład aplikacji zawierającej klasy `Carp` i `Tuna` będące klasami potomnymi klasy `Fish` został przedstawiony w listingu 10.1.

Listing 10.1. Prosta hierarchia dziedziczenia pokazana na przykładzie świata ryb

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    bool FreshWaterFish;
 7:
 8:    void Swim()
 9:    {
10:       if (FreshWaterFish)
11:          cout << "ryby pływające w jeziorach" << endl;
12:       else
13:          cout << "ryby pływające w morzach" << endl;
14:    }
15: };
16:
17: class Tuna: public Fish
18: {
19: public:
20:    Tuna()
21:    {
22:       FreshWaterFish = false;
```

```
23:    }
24: };
25:
26: class Carp: public Fish
27: {
28: public:
29:    Carp()
30:    {
31:       FreshWaterFish = true;
32:    }
33: };
34:
35: int main()
36: {
37:    Carp myLunch;
38:    Tuna myDinner;
39:
40:    cout << "Przewidywane pożywienie" << endl;
41:
42:    cout << "Obiad: ";
43:    myLunch.Swim();
44:
45:    cout << "Kolacja: ";
46:    myDinner.Swim();
47:
48:    return 0;
49: }
```

Wynik ▼

```
Przewidywane pożywienie
Obiad: ryby pływające w jeziorach
Kolacja: ryby pływające w morzach
```

Analiza ▼

Zwróć uwagę na wiersze 37. i 38. w metodzie `main()` odpowiedzialne za utworzenie obiektów klas `Carp` i `Tuna` o nazwach (odpowiednio) `myLunch` i `myDinner`. W wierszach 43. i 46. następuje wywołanie metody `Swim()` w utworzonych obiektach. Teraz spójrz na definicję klas `Tuna` (wiersze od 17. do 24.) i `Carp` (wiersze od 26. do 33.). Jak możesz zobaczyć, wymienione klasy są całkiem zwięzłe, a żadna z nich nie zawiera metody `Swim()`, której wywołanie w funkcji `main()` zakończyło się powodzeniem. Bez wątpienia metoda `Swim()` pochodzi ze zdefiniowanej w wierszach od 3. do 15. klasy `Fish`, po której dziedziczą klasy `Tuna` i `Carp`. Ponieważ klasa `Fish` deklaruje publiczną metodę

o nazwie Swim(), klasy Tuna i Carp dziedziczące po klasie Fish (przez dziedziczenie publiczne, jak pokazano w wierszach 17. i 26.) automatycznie mają dostęp do publicznej metody Swim() klasy bazowej. Zobacz, jak konstruktor obiektów klas Carp i Tuna inicjalizuje zadeklarowaną w klasie bazowej flagę FreshWaterFish, która odgrywa ważną rolę w określeniu danych wyjściowych wyświetlanych przez Fish::Swim().

Specyfikator dostępu protected

Klasa Fish przedstawiona w listingu 10.1 ma publiczny atrybut FreshWaterFish, którego wartość jest ustawiana przez klasy potomne Tuna i Carp jako rodzaj dostosowania klasy do własnych potrzeb (to nosi także nazwę *specjalizacji*) w zakresie zachowania obiektu potomnego klasy Fish. Dzięki wymienionemu atrybutowi można zdefiniować rybę jako słonowodną lub słodkowodną. Jednak w kodzie przedstawionym w listingu 10.1 istnieje poważny błąd: jeśli chcesz, nawet funkcja main() może zmienić wartość flagi zadeklarowanej jako publiczna, a tym samym możliwa do modyfikacji na zewnątrz klasy Fish. Modyfikacja flagi jest prosta i sprowadza się do wydania odpowiedniego polecenia, np. takiego jak poniższe:

```
myDinner.FreshWaterFish = true; // Tuńczyk staje się rybą słodkowodną!
```

Tego — oczywiście — należy unikać. Konieczne jest więc zagwarantowanie, że pewne atrybuty w klasie bazowej będą dostępne jedynie dla klas potomnych i niedostępne dla świata zewnętrznego. Oznacza to, że flaga boolowska FreshWaterFish w klasie Fish powinna być dostępna dla klas potomnych Tuna i Carp, ale już nie dla funkcji main(), w której następuje utworzenie obiektów klas Tuna i Carp. Tutaj z pomocą przychodzi słowo kluczowe protected.

Uwaga

Słowo kluczowe protected (chroniony), podobnie jak public (publiczny) i private (prywatny), jest specyfikatorem dostępu. Po zadeklarowaniu, że obiekt jest chroniony, staje się on dostępny jedynie dla klas potomnych i tzw. przyjaciół, a niedostępny dla świata zewnętrznego, w tym także dla metody main().

Słowo kluczowe protected to specyfikator dostępu, którego powinieneś używać, jeśli chcesz, aby określone atrybuty klasy bazowej były dostępne jedynie dla klas potomnych. Tego rodzaju rozwiązanie przedstawiono w listingu 10.2.

Listing 10.2. Lepsza wersja klasy Fish utworzona z użyciem słowa kluczowego protected w celu udostępniania atrybutów elementów składowych jedynie klasom pochodnym

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: protected:
 6:    bool FreshWaterFish; // Atrybut dostępny jedynie dla klas pochodnych.
 7:
 8: public:
 9:   void Swim()
10:    {
11:       if (FreshWaterFish)
12:          cout << "ryby pływające w jeziorach" << endl;
13:       else
14:          cout << "ryby pływające w morzach" << endl;
15:    }
16: };
17:
18: class Tuna: public Fish
19: {
20: public:
21:    Tuna()
22:    {
23:       FreshWaterFish = false; // Zadeklarowanie chronionego elementu
          ↪składowego klasy bazowej.
24:    }
25: };
26:
27: class Carp: public Fish
28: {
29: public:
30:    Carp()
31:    {
32:       FreshWaterFish = false;
33:    }
34: };
35:
36: int main()
37: {
38:    Carp myLunch;
39:    Tuna myDinner;
40:
41:    cout << "Przewidywane pożywienie" << endl;
42:
43:    cout << "obiad: ";
```

```
44:    myLunch.Swim();
45:
46:    cout << "kolacja: ";
47:    myDinner.Swim();
48:
49:    // Usuń znak komentarza na początku poniższego wiersza, aby przekonać się,
50:    // że chronione elementy składowe są niedostępne na zewnątrz hierarchii klasy.
51:    // myLunch.FreshWaterFish = false;
52:
53:    return 0;
54: }
```

Wynik ▼

```
Przewidywane pożywienie
Obiad: ryby pływające w jeziorach
Kolacja: ryby pływające w morzach
```

Analiza ▼

Pomimo faktu, że dane wyjściowe listingów 10.1 i 10.2 są takie same, istnieje wiele poważnych zmian w zdefiniowanej w wierszach od 3. do 19. klasie `Fish` w listingu 10.2. Pierwsza i najbardziej widoczna zmiana polega na tym, że `Fish::FreshWaterFish` jest obecnie chronionym atrybutem, a tym samym niedostępnym dla funkcji `main()`, co przedstawiono w wierszu 51. (po usunięciu znaku komentarza z początku wiersza otrzymasz błąd kompilacji). Wymieniony parametr zdefiniowany wraz ze specyfikatorem dostępu `proteced` jest dostępny dla klas potomnych `Tuna` i `Carp`, o czym możesz się przekonać w wierszach (odpowiednio) 23. i 32. W omawianym programie pokazano użycie specyfikatora dostępu `protected` w celu zagwarantowania, że atrybuty klasy bazowej, które muszą być dziedziczone, są chronione i niedostępne na zewnątrz hierarchii klas.

To jest bardzo ważny aspekt programowania zorientowanego obiektowo, połączenia abstrakcji danych i dziedziczenia oraz zagwarantowania, że klasy potomne mogą bezpiecznie dziedziczyć atrybuty klasy bazowej, które nie będą mogły być modyfikowane przez żaden komponent spoza systemu hierarchicznego.

Inicjalizacja klasy bazowej — przekazywanie parametrów klasie bazowej

Co zrobić w sytuacji, gdy klasa bazowa zawiera przeciążony konstruktor wymagający użycia argumentów w trakcie tworzenia egzemplarza? W jaki sposób tego rodzaju klasa bazowa będzie mogła być utworzona podczas konstrukcji klasy potomnej? Odpowiedź leży w wykorzystaniu list inicjalizacyjnych oraz wywołaniu odpowiedniego konstruktora klasy bazowej z poziomu konstruktora klasy potomnej, co zostało przedstawione w poniższym fragmencie kodu:

```
class Base
{
public:
   Base(int SomeNumber) // Przeciążony konstruktor.
   {
      // Wykonanie dowolnej operacji z SomeNumber.
   }
};
Class Derived: public Base
{
public:
   Derived(): Base(25)   // Utworzenie egzemplarza klasy Base wraz z argumentem 25.
   {
      // Kod konstruktora klasy pochodnej.
   }
};
```

Ten mechanizm może być użyteczny w klasie `Fish`, w której przez dostarczenie boolowskiego parametru danych wejściowych konstruktorowi klasy `Fish` inicjalizującemu atrybut `Fish::FreshWaterFish` każda klasa potomna klasy `Fish` będzie wymuszała określenie, czy dana ryba jest słonowodna, czy słodkowodna. Takie rozwiązanie zastosowano w listingu 10.3.

Listing 10.3. Konstruktor klasy pochodnej wraz z listą inicjalizacyjną

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: protected:
 6:    bool FreshWaterFish; // Atrybut dostępny jedynie dla klas pochodnych.
 7:
 8: public:
 9:    // Konstruktor klasy Fish.
```

```
10:    Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}
11:
12:    void Swim()
13:    {
14:       if (FreshWaterFish)
15:          cout << "ryby pływające w jeziorach" << endl;
16:       else
17:          cout << "ryby pływające w morzach" << endl;
18:    }
19: };
20:
21: class Tuna: public Fish
22: {
23: public:
24:    Tuna(): Fish(false) {}
25: };
26:
27: class Carp: public Fish
28: {
29: public:
30:    Carp(): Fish(true) {}
31: };
32:
33: int main()
34: {
35:    Carp myLunch;
36:    Tuna myDinner;
37:
38:    cout << "Przewidywane pożywienie" << endl;
39:
40:    cout << "Obiad: ";
41:    myLunch.Swim();
42:
43:    cout << "Kolacja: ";
44:    myDinner.Swim();
45:
46:    // Usuń znak komentarza na początku wiersza 48., aby przekonać się,
47:    // że chronione elementy składowe są niedostępne na zewnątrz hierarchii klasy.
48:    // myLunch.FreshWaterFish = false;
49:
50:    return 0;
51: }
```

Wynik ▼

```
Przewidywane pożywienie
Obiad: ryby pływające w jeziorach
Kolacja: ryby pływające w morzach
```

Analiza ▼

Klasa `Fish` zawiera teraz konstruktor pobierający parametr domyślny podczas inicjalizacji atrybutu `Fish::FreshWaterFish`. Dlatego też jedyna możliwość utworzenia obiektu `Fish` to dostarczenie parametru inicjalizującego chroniony atrybut. W ten sposób klasa `Fish` gwarantuje, że chroniony element składowy nie będzie zawierał przypadkowej wartości, zwłaszcza jeśli klasa potomna zapomni o przypisaniu mu wartości. Klasy potomne `Tuna` i `Carp` muszą definiować konstruktor tworzący egzemplarz klasy bazowej `Fish` wraz z odpowiednim parametrem (o wartości `true` lub `false`), co widać w wierszach (odpowiednio) 24. i 30.

Uwaga

W listingu 10.3 możesz zobaczyć, że boolowska zmienna składowa `Fish::FreshWaterFish` nigdy nie jest dostępna bezpośrednio dla klasy potomnej, ponieważ została zadeklarowana jako prywatna i jest ustawiana przy użyciu konstruktora obiektu `Fish`.

Jeśli klasy potomne nie muszą uzyskiwać dostępu do atrybutów klasy bazowej, to aby zapewnić maksymalną ochronę, pamiętaj, by oznaczyć ten atrybut jako prywatny.

Klasy potomne nadpisują metody klasy bazowej

Jeżeli klasa potomna (`Derived`) implementuje te same funkcje z tymi samymi wartościami zwrotnymi i sygnaturami jak w klasie bazowej (`Base`), po której dziedziczy, wtedy praktycznie nadpisuje te metody klasy bazowej, co przedstawiono w poniższym fragmencie kodu:

```
class Base
{
public:
   void DoSomething()
   {
      // Kod implementacji… Wykonuje dowolne operacje.
   }
};
class Derived:public Base
{
public:
   void DoSomething()
   {
```

```
        // Kod implementacji... Wykonuje inne dowolne operacje.
    }
};
```

Dlatego też jeśli metoda `DoSomething()` będzie wywołana z poziomu egzemplarza klasy `Derived`, wówczas nie zaoferuje takiej samej funkcjonalności jak zdefiniowana w funkcji klasy `Base`.

Gdyby klasy `Tuna` i `Carp` implementowały własne wersje metody `Swim()` istniejącej również w klasie bazowej (`Fish::Swim()`), wtedy wywołanie metody `Swim()`, takie jak pokazane w metodzie `main()` w poniższym fragmencie listingu 10.3:

```
36:     Tuna myDinner;
// Inne wiersze kodu.
44:     myDinner.Swim();
```

spowoduje wywołanie lokalnej implementacji `Tuna::Swim()`. Wspomniana lokalna implementacja nadpisuje metodę klasy bazowej (`Fish::Swim()`). Takie rozwiązanie zostało zademonstrowane w listingu 10.4.

Listing 10.4. Klasy pochodne Tuna i Carp nadpisują metodę Swim() klasy bazowej Fish

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: private:
 6:     bool FreshWaterFish;
 7:
 8: public:
 9:     // Konstruktor klasy Fish.
10:     Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}
11:
12:     void Swim()
13:     {
14:        if (FreshWaterFish)
15:           cout << "ryby pływające w jeziorach" << endl;
16:        else
17:           cout << "ryby pływające w morzach" << endl;
18:     }
19: };
20:
21: class Tuna: public Fish
22: {
```

```
23: public:
24:    Tuna(): Fish(false) {}
25:
26:    void Swim()
27:    {
28:       cout << "tuńczyk pływa naprawdę szybko" << endl;
29:    }
30: };
31:
32: class Carp: public Fish
33: {
34: public:
35:    Carp(): Fish(true) {}
36:
37:    void Swim()
38:    {
39:       cout << "karp pływa naprawdę wolno" << endl;
40:    }
41: };
42:
43: int main()
44: {
45:    Carp myLunch;
46:    Tuna myDinner;
47:
48:    cout << "Przewidywane pożywienie" << endl;
49:
50:    cout << "Obiad: ";
51:    myLunch.Swim();
52:
53:    cout << "Kolacja: ";
54:    myDinner.Swim();
55:
56:    return 0;
57: }
```

Wynik ▼

```
Przewidywane pożywienie
Obiad: karp pływa naprawdę wolno
Kolacja: tuńczyk pływa naprawdę szybko
```

Analiza ▼

Powyższe dane wyjściowe pokazują, że polecenie `myLunch.Swim()` w wierszu 51. powoduje wywołanie metody `Carp::Swim()` zdefiniowanej w wierszach od 37. do 40. Podobnie polecenie `myDinner.Swim()` w wierszu 54. powoduje wywołanie

metody Tuna::Swim() zdefiniowanej w wierszach od 26. do 29. Innymi słowy, implementacja metody Fish::Swim() w klasie bazowej Fish (patrz wiersze od 12. do 18.) została nadpisana przez identyczne metody zdefiniowane klasach Tuna i Carp dziedziczących po klasie Fish. Jedyny sposób wywołania metody Fish::Swim() to wyraźne podanie w klasie potomnej nazwy funkcji składowej w klasie bazowej lub użycie w funkcji main() operatora wyboru zakresu, który wyraźnie wywoła Fish::Swim(). To drugie rozwiązanie będzie przedstawione w dalszej części lekcji.

Wywoływanie nadpisanych metod klasy bazowej

W listingu 10.4 widziałeś przykład, w którym klasa potomna Tuna nadpisuje metodę Swim() klasy bazowej Fish przez implementację jej wersji:

```
Tuna myDinner;
myDinner.Swim(); // Polecenie spowoduje wywołanie Tuna::Swim().
```

Jeżeli chcesz w funkcji main() w listingu 10.4 wywołać metodę Fish::Swim(), musisz skorzystać z operatora wyboru zakresu (::) o następującej składni:

```
myDinner.Fish::Swim();  // Polecenie spowoduje wywołanie Fish::Swim(),
                        // mimo że obiektem jest Tuna.
```

W przedstawionym nieco dalej listingu 10.5 zobaczysz, jak wywołać element składowy klasy bazowej z poziomu egzemplarza klasy potomnej.

Wywoływanie metod klasy bazowej z poziomu klas potomnych

Zazwyczaj metoda Fish::Swim() będzie zawierała podstawową implementację dostępnej dla wszystkich ryb umiejętności pływania. Jeżeli w specjalizowanych implementacjach w postaci metod Tuna::Swim() i Carp::Swim() chcesz ponownie wykorzystać ogólną implementację znajdującą się klasie bazowej, musisz skorzystać z operatora wyboru zakresu (::), co przedstawiono w poniższym kodzie:

```
class Carp: public Fish
{
public:
   Carp(): Fish(true) {}
```

```
   void Swim()
   {
      cout << "Karp pływa naprawdę wolno" << endl;
      Fish::Swim();  // Użycie operatora wyboru zakresu (::).
   }
};
```

Powyższe rozwiązanie zastosowano także w listingu 10.5.

Listing 10.5. Użycie operatora wyboru zakresu (::) w celu wywołania metod klasy bazowej z poziomu metod klas pochodnych i funkcji main()

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: private:
 6:    bool FreshWaterFish;
 7:
 8: public:
 9:    // Konstruktor klasy Fish.
10:    Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}
11:
12:    void Swim()
13:    {
14:       if (FreshWaterFish)
15:          cout << "ryby pływające w jeziorach" << endl;
16:       else
17:          cout << "ryby pływające w morzach" << endl;
18:    }
19: };
20:
21: class Tuna: public Fish
22: {
23: public:
24:    Tuna(): Fish(false) {}
25:
26:    void Swim()
27:    {
28:       cout << "tuńczyk pływa naprawdę szybko" << endl;
29:    }
30: };
31:
32: class Carp: public Fish
33: {
34: public:
35:    Carp(): Fish(true) {}
36:
```

```
37:    void Swim()
38:    {
39:       cout << "karp pływa naprawdę wolno" << endl;
40:       Fish::Swim();
41:    }
42: };
43:
44: int main()
45: {
46:    Carp myLunch;
47:    Tuna myDinner;
48:
49:    cout << "Przewidywane pożywienie" << endl;
50:
51:    cout << "Obiad: ";
52:    myLunch.Swim();
53:
54:    cout << "Kolacja: ";
55:    myDinner.Fish::Swim();
56:
57:    return 0;
58: }
```

Wynik ▼

```
Przewidywane pożywienie
Obiad: karp pływa naprawdę wolno
ryby pływające w jeziorach
Kolacja: ryby pływające w morzach
```

Analiza ▼

Zdefiniowana w wierszach od 37. do 41. metoda `Carp::Swim()` pokazuje, jak wywoływać metodę klasy bazowej `Fish::Swim()` przy użyciu operatora wyboru zakresu (`::`). Z kolei w wierszu 55. pokazano, jak ten sam operator wyboru zakresu wykorzystać do wywołania metody klasy bazowej (`Fish::Swim()`) z funkcji `main()` danego egzemplarza klasy `Tuna`.

Klasy potomne ukrywają metody klasy bazowej

Nadpisywanie metod może przyjąć postać ekstremalną, gdy metoda `Tuna::Swim()` potencjalnie ukryje wszystkie dostępne, przeciążone wersje metody `Fish::Swim()`. Taka sytuacja doprowadzi do wygenerowania błędu

w trakcie kompilacji, jeśli w kodzie jest używana (tzn. wywoływana) przeciążona wersja metody. Przykład takiej sytuacji pokazano w listingu 10.6.

Listing 10.6. Pokazanie, że Tuna::Swim() ukrywa przeciążoną metodę Fish::Swim(bool)

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    void Swim()
 7:    {
 8:        cout << "Ryby pływają... !" << endl;
 9:    }
10:
11:    void Swim(bool FreshWaterFish)
12:    {
13:       if (FreshWaterFish)
14:          cout << "ryby pływające w jeziorach" << endl;
15:       else
16:          cout << "ryby pływające w morzach" << endl;
17:    }
18: };
19:
20: class Tuna: public Fish
21: {
22: public:
23:    void Swim()
24:    {
25:       cout << "tuńczyk pływa naprawdę szybko" << endl;
26:    }
27: };
28:
29: int main()
30: {
31:    Tuna myDinner;
32:
33:    cout << "Przewidywane pożywienie" << endl;
34:
35:    // myDinner.Swim(false); // Błąd kompilacji,: Fish::Swim(bool) zostaje ukryta
       ➥przez Tuna::Swim().
36:    myDinner.Swim();
37:
38:    return 0;
39: }
```

Wynik ▼

```
Przewidywane pożywienie
tuńczyk pływa naprawdę szybko
```

Analiza ▼

Powyższa wersja klasy `Fish` jest nieco inna od dotąd prezentowanych. Poza minimalizacją samej klasy w celu dokładnego pokazania problemu, bieżąca wersja zawiera dwie przeciążone metody `Swim()`. Pierwsza z nich (patrz wiersze od 6. do 9.) nie pobiera parametrów, natomiast druga (patrz wiersze od 11. do 17.) pobiera parametr boolowski. Ponieważ klasa `Tuna` stosuje dziedziczenie publiczne po klasie `Fish` (patrz wiersz 20.), można oczekiwać, że obie wersje metody `Fish::Swim()` będą dostępne z poziomu egzemplarza klasy `Tuna`. Jednak, jak pokazano w wierszach od 23. do 26., klasa `Tuna` implementuje własną lokalną wersję metody `Tuna::Swim()`, co powoduje ukrycie `Fish::Swim(bool)` przed kompilatorem. Usunięcie znaku komentarza na początku wiersza 35. spowoduje wygenerowanie błędu w trakcie kompilacji.

Dlatego też, jeśli chcesz wywołać funkcję `Fish::Swim(bool)` z poziomu egzemplarza klasy `Tuna`, do dyspozycji masz trzy przedstawione poniżej rozwiązania.

1. Rozwiązanie 1.: użycie w funkcji `main()` operatora wyboru zakresu:

   ```
   myDinner.Fish::Swim();
   ```

2. Rozwiązanie 2.: użycie słowa kluczowego `using` w klasie `Tuna`; spowoduje to odkrycie metody `Swim()` znajdującej się w klasie `Fish`:

   ```
   class Tuna: public Fish
   {
   public:
      using Fish::Swim;     // Ujawnienie metod Swim w klasie bazowej Fish.

      void Swim()
      {
         cout << "tuńczyk pływa naprawdę szybko" << endl;
      }
   };
   ```

3. Rozwiązanie 3.: nadpisanie wszystkich przeciążonych wariantów metody `Swim()` w klasie `Tuna` (wywołanie metod `Fish::Swim(...)` przez `Tuna::Fish(...)`):

```
class Tuna: public Fish
{
public:
   void Swim(bool FreshWaterFish)
   {
      Fish::Swim(FreshWaterFish);
   }

   void Swim()
   {
      cout << "tuńczyk pływa naprawdę szybko" << endl;
   }
};
```

Kolejność użycia konstruktorów

Kiedy tworzysz obiekt klasy `Tuna` dziedziczący po klasie `Fish`, powstaje pytanie, czy konstruktor klasy `Tuna` będzie wywołany przed konstruktorem klasy `Fish`, czy raczej po nim? Ponadto jaka będzie kolejność atrybutów elementów składowych, takich jak `Fish::FreshWaterFish`, w trakcie tworzenia obiektów w hierarchii klasy? Obiekty klasy bazowej są tworzone przed obiektami klas potomnych. Dlatego też najpierw nastąpi utworzenie obiektu `Fish`, aby jego elementy składowe — zwłaszcza chronione i publiczne znajdujące się w klasie `Fish` — były gotowe do użycia podczas tworzenia egzemplarza klasy `Tuna`. W operacji tworzenia egzemplarzy klas `Fish` i `Tuna` atrybuty składowe (np. `Fish::FreshWaterFish`) są tworzone przed wywołaniem konstruktora `Fish::Fish()`. Dzięki temu wymienione atrybuty składowe są gotowe, zanim konstruktor będzie chciał ich użyć podczas pracy. To samo ma zastosowanie dla `Tuna::Tuna()`.

Kolejność użycia destruktorów

Kiedy egzemplarz klasy `Tuna` wypada z zakresu, kolejność wykonywania destruktorów jest odwrotna, w porównaniu z kolejnością wywoływania konstruktorów. Kod przedstawiony w listingu 10.7 to prosty przykład pokazujący kolejność wywoływania konstruktorów i destruktorów.

Listing 10.7. Kolejność konstrukcji i destrukcji w klasie bazowej, pochodnej oraz ich elementach składowych

```
0: #include <iostream>
1: using namespace std;
```

```
 2:
 3: class FishDummyMember
 4: {
 5: public:
 6:    FishDummyMember()
 7:    {
 8:       cout << "Konstruktor FishDummyMember" << endl;
 9:    }
10:
11:    ~FishDummyMember()
12:    {
13:       cout << "Destruktor FishDummyMember" << endl;
14:    }
15: };
16:
17: class Fish
18: {
19: protected:
20:    FishDummyMember dummy;
21:
22: public:
23:    // Konstruktor klasy Fish.
24:    Fish()
25:    {
26:       cout << "Konstruktor Fish" << endl;
27:    }
28:
29:    ~Fish()
30:    {
31:       cout << "Destruktor Fish" << endl;
32:    }
33: };
34:
35: class TunaDummyMember
36: {
37: public:
38:    TunaDummyMember()
39:    {
40:       cout << "Konstruktor TunaDummyMember" << endl;
41:    }
42:
43:    ~TunaDummyMember()
44:    {
45:       cout << "Destruktor TunaDummyMember" << endl;
46:    }
47: };
48:
49:
50: class Tuna: public Fish
```

```
51: {
52: private:
53:    TunaDummyMember dummy;
54:
55: public:
56:    Tuna()
57:    {
58:       cout << "Konstruktor Tuna" << endl;
59:    }
60:    ~Tuna()
61:    {
62:       cout << "Destruktor Tuna" << endl;
63:    }
64:
65: };
66:
67: int main()
68: {
69:    Tuna myDinner;
70: }
```

Wynik ▼

```
Konstruktor FishDummyMember
Konstruktor Fish
Konstruktor TunaDummyMember
Konstruktor Tuna
Destruktor Tuna
Destruktor TunaDummyMember
Destruktor Fish
Destruktor FishDummyMember
```

Analiza ▼

Zdefiniowana w wierszach od 67. do 70. funkcja `main()` jest zdecydowanie niewielka, gdy weźmiemy pod uwagę ilość generowanych danych wyjściowych. Utworzenie egzemplarza klasy `Tuna` jest wystarczające do wygenerowania przedstawionych danych wyjściowych, ponieważ polecenia `cout` zostały umieszczone w konstruktorach i destruktorach wszystkich wykorzystywanych obiektów. Aby ułatwić zrozumienie kolejności tworzenia i niszczenia zmiennych składowych, zdefiniowano dwie przykładowe klasy o nazwach `FirstDummyMember` i `TunaDummyMember` wraz z poleceniami `cout` w ich konstruktorach i destruktorach. Klasy `Fish` i `Tuna` zawierają elementy składowe wymienionych przykładowych klas (patrz wiersze 20. i 53.). Dane wyjściowe programu pokazują, że utworzenie

obiektu klasy Tuna oznacza budowanie obiektów od góry hierarchii. Dlatego też najpierw następuje utworzenia klasy bazowej Fish dla egzemplarza klasy Tuna — wówczas powstają elementy składowe klasy Fish, tzn. Fish::dummy. Następnie wywoływany jest konstruktor klasy Fish, jego wykonanie prawidłowo następuje po utworzeniu atrybutów składowych, takich jak dummy. Po przygotowaniu klasy bazowej tworzenie egzemplarza Tuna jest kontynuowane — teraz powstaje Tuna::dummy. Dalej wywoływany jest kod konstruktora Tuna::Tuna(). Dane wyjściowe programu pokazują, że kolejność wywoływania destruktorów jest odwrotna do przedstawionej powyżej kolejności wywoływania konstruktorów.

Dziedziczenie prywatne

Dziedziczenie prywatne różni się od omówionego dotąd publicznego użyciem słowa kluczowego private w wierszu, w którym klasa potomna deklaruje dziedziczenie po klasie bazowej:

```
class Base
{
    // Metody i elementy składowe klasy bazowej.
};

class Derived: private Base       // Dziedziczenie prywatne.
{
    // Metody i elementy składowe klasy pochodnej.
};
```

Dziedziczenie prywatne klasy bazowej oznacza, że wszystkie publiczne metody i atrybuty klasy bazowej będą prywatne (tzn. niedostępne) dla każdego egzemplarza klasy potomnej. Innymi słowy, nawet publiczne elementy składowe i metody klasy Base będą mogły być używane przez klasę Derived, ale nie przez żaden inny komponent posiadający egzemplarz klasy Derived.

To jest wyraźne przeciwieństwo w stosunku do przedstawianych, począwszy od listingu 10.1, przykładów z użyciem klas Tuna i Fish. Funkcja main() w listingu 10.1 może wywoływać metodę Fish::Swim() egzemplarza Tuna, ponieważ Fish::Swim() jest metodą publiczną, a klasa Tuna stosuje dziedziczenie publiczne po klasie Fish. Jeżeli w wierszu 17. wymienionego listingu słowo kluczowe public zastąpisz słowem private, kompilacja zakończy się niepowodzeniem.

Dlatego też dla komponentów znajdujących się na zewnątrz hierarchii dziedziczenia dziedziczenie prywatne praktycznie nie oznacza relacji jest-czymś (wyobraź sobie tuńczyka, który nie może pływać). Ponieważ dziedziczenie prywatne oznacza, że atrybuty i metody klasy bazowej mogą być używane jedynie przez podklasy dziedziczące po klasie bazowej, tego rodzaju relacja jest nazywana ma-coś. W życiu codziennym możesz znaleźć wiele przykładów dziedziczenia prywatnego, kilka z nich wymieniono w tabeli 10.2.

Tabela 10.2. Wzięte z życia codziennego przykłady dziedziczenia prywatnego

Klasa bazowa	Przykład klasy potomnej
Silnik	Samochód (samochód ma silnik)
Serce	Ssak (ssak ma serce)
Nabój	Pióro (pióro ma nabój z atramentem)
Księżyc	Niebo (niebo „ma księżyc”)

Przeanalizujmy teraz dziedziczenie prywatne na przykładzie relacji samochodu i jego silnika. Przykład przedstawiono w listingu 10.8.

Listing 10.8. Klasa Car powiązana z klasą Motor za pomocą dziedziczenia prywatnego

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Motor
 4: {
 5: public:
 6:    void SwitchIgnition()
 7:    {
 8:       cout << "Zapłon włączony" << endl;
 9:    }
10:    void PumpFuel()
11:    {
12:       cout << "Paliwo w cylindrach" << endl;
13:    }
14:    void FireCylinders()
15:    {
16:       cout << "Vroooom" << endl;
17:    }
18: };
19:
```

```
20: class Car:private Motor
21: {
22: public:
23:    void Move()
24:    {
25:       SwitchIgnition();
26:       PumpFuel();
27:       FireCylinders();
28:    }
29: };
30:
31: int main()
32: {
33:    Car myDreamCar;
34:    myDreamCar.Move();
35:
36:    return 0;
37: }
```

Wynik ▼

```
Zapłon włączony
Paliwo w cylindrach
Vroooom
```

Analiza ▼

Zdefiniowana w wierszach od 3. do 18. klasa `Motor` jest całkiem prosta i składa się z trzech chronionych funkcji składowych odpowiedzialnych za włączenie zapłonu, dostarczenie paliwa i uruchomienie cylindrów. Jak pokazano w wierszu 20., klasa `Car` dziedziczy po klasie `Motor`; ponieważ użyto słowa kluczowego `private`, mamy do czynienia z dziedziczeniem prywatnym. Publiczna funkcja `Car::Move()` wywołuje elementy składowe klasy bazowej `Motor`. Jeżeli w funkcji `main()` spróbujesz użyć następującego wywołania:

```
myDreamCar.PumpFuel();
```

kompilacja zakończy się niepowodzeniem i wyświetleniem komunikatu błędu, podobnego do: „Błąd C2247: metoda `Motor::PumpFuel` jest niedostępna, ponieważ `Car` stosuje dziedziczenie prywatne po klasie `Motor`”.

Uwaga

> Jeżeli inna klasa `SuperCar` ma dziedziczyć po klasie `Car`, wtedy bez względu na naturę dziedziczenia pomiędzy klasami `SuperCar` i `Car` klasa `SuperCar` nie będzie miała dostępu do żadnych publicznych elementów składowych i metod klasy bazowej `Motor`. Wynika to z faktu, że relacja pomiędzy klasami `Car` i `Motor` jest dziedziczeniem prywatnym. Oznacza to, że dla egzemplarzy innych niż `Car` dostęp jest zdefiniowany jako `private` (czyli brak dostępu), dotyczy to nawet publicznych elementów składowych klasy `Base`.
>
> Innymi słowy, gdy kompilator określa uprawnienia danej klasy względem publicznych i chronionych elementów składowych klasy bazowej, odgrywa dominującą rolę najbardziej restrykcyjny specyfikator dostępu.

Dziedziczenie chronione

Dziedziczenie chronione różni się od publicznego użyciem słowa kluczowego `protected` w wierszu, w którym klasa potomna deklaruje dziedziczenie po klasie bazowej:

```
class Base
{
    // Metody i elementy składowe klasy bazowej.
};

class Derived: protected Base       // Dziedziczenie chronione.
{
    // Metody i elementy składowe klasy potomnej.
};
```

Dziedziczenie chronione jest pod wieloma względami podobne do prywatnego:

- również oznacza relację ma-coś,
- również pozwala klasom potomnym na uzyskanie dostępu do wszystkich publicznych i chronionych elementów składowych klasy `Base`,
- komponenty znajdujące się poza hierarchią dziedziczenia i posiadające egzemplarz klasy `Derived` nie mogą uzyskać dostępu do publicznych elementów składowych klasy `Base`.

Dziedziczenie chronione jest jednak nieco inne, jeśli weźmiemy pod uwagę dziedziczenie po klasie potomnej:

```
class Derived2: protected Derived
{
    // Możliwość uzyskania dostępu do elementów składowych klasy bazowej.
};
```

Hierarchia dziedziczenia prywatnego pozwala podklasom podklas (tutaj `Derived2`) na uzyskanie dostępu do publicznych elementów składowych klasy bazowej (tutaj `Base`), czego przykład przedstawiono w listingu 10.9. Takie rozwiązanie jest niemożliwe w przypadku dziedziczenia prywatnego pomiędzy klasami `Derived` i `Base`.

Listing 10.9. Klasa SuperCar dziedziczy po klasie Car, która z kolei stosuje dziedziczenie chronione po klasie Motor

```
0: #include <iostream>
1: using namespace std;
2:
3: class Motor
4: {
5: public:
6:    void SwitchIgnition()
7:    {
8:       cout << "Zapłon włączony" << endl;
9:    }
10:    void PumpFuel()
11:    {
12:       cout << "Paliwo w cylindrach" << endl;
13:    }
14:    void FireCylinders()
15:    {
16:       cout << "Vroooom" << endl;
17:    }
18: };
19:
20: class Car:protected Motor
21: {
22: public:
23:    void Move()
24:    {
25:       SwitchIgnition();
26:       PumpFuel();
27:       FireCylinders();
28:    }
29: };
30:
31: class SuperCar:protected Car
32: {
33: public:
34:    void Move()
35:    {
36:       SwitchIgnition();   // Ma dostęp do elementów składowych klasy bazowej.
```

```
37:         PumpFuel();          // Ze względu na dziedziczenie "chronione" pomiędzy
            ↪klasami Car i Motor.
38:         FireCylinders();
39:         FireCylinders();
40:         FireCylinders();
41:     }
42: };
43:
44: int main()
45: {
46:     SuperCar myDreamCar;
47:     myDreamCar.Move();
48:
49:     return 0;
50: }
```

Wynik ▼

```
Zapłon włączony
Paliwo w cylindrach
Vroooom
Vroooom
Vroooom
```

Analiza ▼

Jak pokazano w wierszu 20., klasa `Car` stosuje dziedziczenie chronione po klasie `Motor`, natomiast klasa `SuperCar` stosuje dziedziczenie chronione po klasie `Car` (patrz wiersz 31.). Jak możesz się przekonać, implementacja metody `SuperCar::Move()` używa metod publicznych zdefiniowanych w klasie bazowej `Motor`. Ten dostęp do klasy bazowej `Motor` poprzez klasę pośrednią `Car` jest gwarantowany przez relację między klasami `Car` i `Motor`. Jeżeli zamiast dziedziczenia chronionego byłoby zastosowane prywatne, klasa `SuperCar` nie miałaby dostępu do publicznych elementów składowych klasy `Motor`, ponieważ kompilator wybierze najbardziej restrykcyjne specyfikatory dostępu. Zauważ, że natura relacji pomiędzy klasami `Car` i `SuperCar` nie odgrywa żadnej roli w uzyskaniu dostępu do klasy `Base`. Dlatego też, jeśli nawet w wierszu 31. słowo kluczowe `protected` zmienisz na `public` lub `private`, to i tak nic nie zmieni w kompilacji omawianego programu.

Ostrzeżenie

Dziedziczenie prywatne lub chronione stosuj jedynie w ostateczności.

W większości przypadków, gdy używane jest dziedziczenie prywatne, np. w omawianym programie zawierającym klasy `Car` i `Motor`, klasa bazowa będzie atrybutem składowym klasy `Car`, a nie superklasą. Wprowadzając dziedziczenie po klasie `Motor`, w praktyce narzuciłeś ograniczenie, że samochód (klasa `Car`) może mieć tylko jeden silnik (klasa `Motor`). Nie ma więc szczególnych zalet zdefiniowania egzemplarza klasy `Motor` jako prywatnego elementu składowego.

Samochody ewoluują i np. pojazdy hybrydowe są wyposażone w dwa silniki: jeden spalinowy i jeden elektryczny. W takim przypadku przedstawiona tutaj hierarchia dziedziczenia klasy `Car` okazuje się wąskim gardłem.

Uwaga

Posiadanie egzemplarza klasy `Motor` jako prywatnego elementu składowego zamiast dziedziczenia po nim nosi nazwę *kompozycji* lub *agregacji*. Tego rodzaju klasa `Car` będzie przedstawiała się następująco:

```
class Car
{
private:
   Motor heartOfCar;

public:
   void Move()
   {
      heartOfCar.SwitchIgnition();
      heartOfCar.PumpFuel();
      heartOfCar.FireCylinders();
   }
};
```

To może być dobre rozwiązanie, ponieważ pozwala na łatwe dodawanie kolejnych silników jako atrybutów składowych istniejącej klasy `Car` i nie wymaga przy tym zmiany hierarchii dziedziczenia lub projektu klasy względem jej klientów.

Problem segmentowania

Co się stanie, jeśli programista zastosuje przedstawiony poniżej kod?

```
Derived objectDerived;
Base objectBase = objectDerived;
```

Albo użyjesz poniższego fragmentu kodu?

```
void FuncUseBase(Base input);
...
Derived objectDerived;
FuncUseBase(objectDerived);   // W trakcie wywołania funkcji objectDerived nastąpi
                              // segmentowanie podczas kopiowania.
```

W obu pokazanych przypadkach obiekt typu `Derived` zostanie skopiowany do innego obiektu (typu `Base`) wyraźnie przez przypisanie lub na skutek przekazania jako argument. Kompilator po prostu kopiuje jedynie część `Base` obiektu `objectDerived` — tzn. nie jest kopiowany pełny obiekt, tylko część, która pasuje do `Base`. Nie jest to pożądane rozwiązanie, a niechciana redukcja części danych tworzących specjalizację `Derived` klasy `Base` nosi nazwę *segmentowania* (ang. *slicing*).

Ostrzeżenie

> Aby uniknąć problemów związanych z segmentowaniem, nie należy przekazywać parametrów przy użyciu wartości. Zamiast tego przekazuj je jako wskaźniki do klasy bazowej lub jako referencje (opcjonalnie typu `const`).

Dziedziczenie wielokrotne

Wcześniej w tej lekcji wspomniano, że w pewnych sytuacjach może być stosowane wielokrotne dziedziczenie (np. dotyczy ono wymienionego już na początku lekcji dziobaka). Dziobak jest po części ssakiem, ptakiem i gadem. W takich przypadkach język C++ pozwala na utworzenie klasy potomnej dziedziczącej po dwóch lub wielu klasach bazowych:

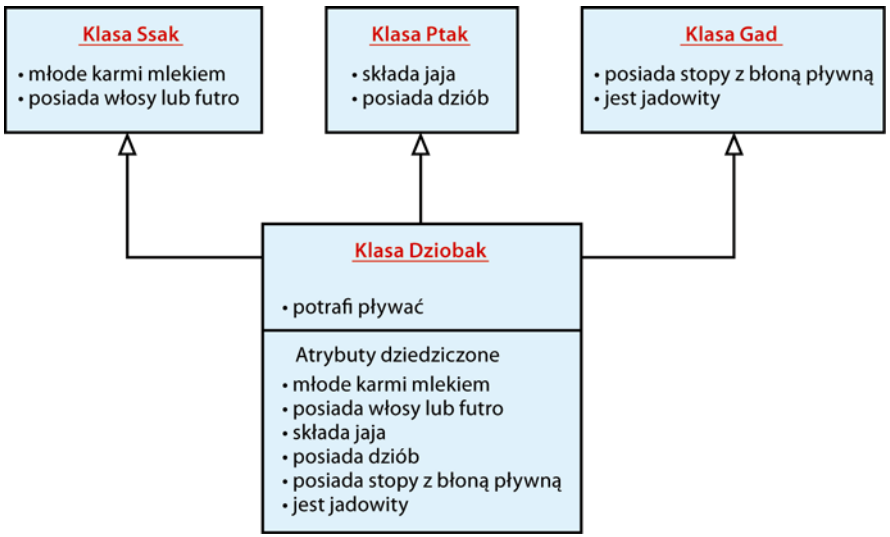
```
class Derived: specyfikator-dostępu Base1, specyfikator-dostępu Base2
{
    // Elementy składowe klasy.
};
```

Pokazana na rysunku 10.3 hierarchia dziedziczenia dla dziobaka wygląda całkiem inaczej niż w przypadku tuńczyka i karpia (patrz rysunek 10.2).

Dlatego też klasę Platypus w C++ możemy przedstawić w następujący sposób:

```
class Platypus: public Mammal, public Reptile, public Bird
{
    // Elementy składowe klasy Platypus.
};
```

Użycie klasy Platypus stosującej dziedziczenie wielokrotne przedstawiono w listingu 10.10.



RYSUNEK 10.3. Relacja klas Dziobak (Platypus) z klasami Ssak, Gad i Ptak

Listing 10.10. Użycie dziedziczenia wielokrotnego w celu utworzenia modelu dziobaka, który jest po części ssakiem, ptakiem i gadem

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Mammal
 4: {
 5: public:
 6:    void FeedBabyMilk()
 7:    {
 8:       cout << "Ssak: młode musi dostać mleko!" << endl;
 9:    }
10: };
11:
12: class Reptile
13: {
14: public:
15:    void SpitVenom()
16:    {
17:       cout << "Gad: przegonić wroga, pluć jadem!" << endl;
18:    }
19: };
20:
21: class Bird
22: {
23: public:
24:    void LayEggs()
25:    {
```

```
26:       cout << "Ptak: muszę złożyć jaja!" << endl;
27:    }
28: };
29:
30: class Platypus: public Mammal, public Bird, public Reptile
31: {
32: public:
33:    void Swim()
34:    {
35:       cout << "Dziobak: uwaga, potrafię pływać!" << endl;
36:    }
37: };
38:
39: int main()
40: {
41:    Platypus realFreak;
42:    realFreak.LayEggs();
43:    realFreak.FeedBabyMilk();
44:    realFreak.SpitVenom();
45:    realFreak.Swim();
46:
47:    return 0;
48: }
```

Wynik ▼

```
Ssak: młode musi dostać mleko!
Gad: przegonić wroga, pluć jadem!
Ptak: muszę złożyć jaja!
Dziobak: uwaga, potrafię pływać!
```

Analiza ▼

Zawarta w wierszach od 30. do 37. definicja klasy `Platypus` jest naprawdę zwięzła. W zasadzie zawiera definicję dziedziczenia po trzech innych klasach: `Mammal`, `Reptile` i `Bird`. Funkcja `main()` zdefiniowana w wierszach od 41. do 44. może wywołać metody trzech wymienionych klas bazowych, używając obiektu potomnego klasy `Platypus` o nazwie `realFreak`. Poza wywołaniem metod odziedziczonych po klasach `Mammal`, `Bird` i `Reptile`, w wierszu 45. funkcji `main()` znajduje się wywołanie `Platypus::Swim()`. Omówiony program prezentuje składnię dziedziczenia wielokrotnego i pokazuje, jak klasa potomna udostępnia wszystkie publiczne atrybuty (w tym przypadku publiczne metody składowe) swoich klas bazowych.

Uwaga

Dziobak potrafi pływać, ale nie jest rybą. Dlatego też w kodzie przedstawionym w listingu 10.10 nie zostało wprowadzone dziedziczenie klasy `Platypus` po klasie `Fish`, co umożliwiłoby wygodne użycie istniejącej metody `Fish::Swim()`. Podczas podejmowania decyzji nie zapominaj, że dziedziczenie publiczne powinno być relacją jest-czymś, natomiast nie powinno być używane do bezkrytycznego osiągania pewnych celów związanych z ponownym użyciem kodu. Cele te można osiągnąć na inne sposoby.

TAK	NIE
Stosuj publiczną hierarchię dziedziczenia w celu utworzenia relacji jest-czymś. Stosuj publiczną hierarchię dziedziczenia w celu utworzenia relacji ma-coś. Pamiętaj: dziedziczenie publiczne oznacza, że klasy potomne klas potomnych mają dostęp do publicznych i chronionych elementów składowych klasy bazowej. Pamiętaj: dziedziczenie prywatne oznacza, że nawet klasy potomne klas potomnych nie mają dostępu do klasy bazowej. Pamiętaj: dziedziczenie chronione oznacza, że klasy potomne klas potomnych mają dostęp do chronionych i publicznych metod klasy bazowej. Pamiętaj: niezależnie od natury relacji dziedziczenia, prywatne elementy składowe w klasie bazowej są niedostępne dla jakiejkolwiek klasy potomnej.	Nie twórz hierarchii dziedziczenia w celu jedynie prostego ponownego użycia kodu funkcji. Nie używaj bezkrytycznie dziedziczenia prywatnego lub publicznego, ponieważ może to doprowadzić do powstania architekturalnych wąskich gardeł utrudniających w przyszłości skalowanie aplikacji. Nie twórz w klasach potomnych metod, które przypadkowo ukrywają metody w klasie bazowej, ponieważ mają takie same nazwy, ale inne parametry wejściowe.

Podsumowanie

W tej lekcji omówiono podstawy dziedziczenia w C++. Dowiedziałeś się, że dziedziczenie publiczne jest rodzajem relacji jest-czymś pomiędzy klasą bazową i potomną, podczas gdy dziedziczenie prywatne i chronione tworzą rodzaj relacji ma-coś. Przekonałeś się, że użyty w aplikacji specyfikator dostępu `protected` powoduje udostępnienie atrybutów klasy bazowej tylko klasom potomnym, natomiast ukrywa je przed klasami znajdującymi się poza hierarchią dziedziczenia. Dowiedziałeś się, że dziedziczenie chronione różni się od

prywatnego — w dziedziczeniu chronionym klasy potomne mogą uzyskać dostęp do publicznych i chronionych elementów składowych klasy bazowej, co jest niemożliwe w dziedziczeniu prywatnym. Przedstawione zostały także podstawy nadpisywania metod oraz ich ukrywania. Przekonałeś się również, jak za pomocą słowa kluczowego `using` uniknąć niechcianego ukrywania metod.

Możesz teraz przystąpić do udzielenia odpowiedzi na kilka pytań, a następnie przejść do kolejnego ważnego filaru programowania zorientowanego obiektowo, czyli polimorfizmu.

Pytania i odpowiedzi

Pytanie: Zostałem poproszony o przygotowanie klasy `Mammal` wraz z kilkoma ssakami, takimi jak `Human`, `Lion` i `Whale`. Czy powinienem użyć hierarchii dziedziczenia? Jeśli tak, to której?

Odpowiedź: Ponieważ `Human` (człowiek), `Lion` (lew) i `Whale` (wieloryb) są ssakami i praktycznie spełniają wymagania relacji jest-czymś, powinieneś użyć dziedziczenia publicznego, w którym klasą bazową jest `Mammal`, a pozostałe klasy, takie jak `Human`, `Lion` i `Whale`, dziedziczą po niej.

Pytanie: Jaka jest różnica pomiędzy pojęciami *klasa potomna* i *podklasa*?

Odpowiedź: W zasadzie żadna. Oba wymienione pojęcia określają klasę pochodną — tzn. specjalistyczną — klasy bazowej.

Pytanie: Klasa potomna używa dziedziczenia publicznego po klasie bazowej. Czy mogę uzyskać dostęp do prywatnych elementów składowych klasy bazowej?

Odpowiedź: Nie. Kompilator zawsze wymusza zastosowanie najbardziej restrykcyjnego specyfikatora dostępu, jakiego można użyć. Bez względu na naturę dziedziczenia, prywatne elementy składowe klasy nigdy nie są ujawniane na zewnątrz danej klasy. Wyjątkiem od tej zasady są klasy i funkcje zadeklarowane przy użyciu słowa kluczowego `friend`.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Chcę, aby pewne elementy składowe klasy bazowej były dostępne dla klasy potomnej, ale nie na zewnątrz hierarchii klas. Którego specyfikatora dostępu powinienem użyć?
2. Co się stanie, jeżeli obiekt klasy potomnej przekażę jako argument funkcji pobierającej za pomocą wartości parametr klasy bazowej?
3. Które rozwiązanie powinienem stosować: dziedziczenie prywatne czy kompozycję?
4. W jaki sposób słowo kluczowe `using` może mi pomóc w hierarchii dziedziczenia?
5. Klasa `Derived` stosuje dziedziczenie prywatne po klasie `Base`. Inna klasa o nazwie `SubDerived` stosuje dziedziczenie publiczne po klasie `Derived`. Czy klasa `SubDerived` może uzyskać dostęp do publicznych elementów składowych klasy `Base`?

Ćwiczenia

1. W jakiej kolejności są wywoływane konstruktory klasy `Platypus` w programie przedstawionym w listingu 10.10?
2. Wyjaśnij, jak są ze sobą powiązane klasy `Polygon`, `Triangle` i `Shape`.
3. Klasa `D2` dziedziczy po `D1`, która z kolei dziedziczy po `Base`. Gdzie i których specyfikatorów dostępu należy użyć, aby uniemożliwić klasie `D2` uzyskanie dostępu do publicznych elementów składowych klasy `Base`?
4. Jaka jest natura dziedziczenia w poniższym fragmencie kodu?

```
class Derived: Base
{
    // Elementy składowe klasy Derived.
};
```

5. **Łowcy błędów:** Co jest nie tak z poniższym fragmentem kodu?

```
class Derived: public Base
{
    // Elementy składowe klasy Derived.
};
void SomeFunc (Base value)
{
    // ...
}
```

Lekcja 11

Polimorfizm

Po poznaniu podstaw dziedziczenia, tworzenia hierarchii dziedziczenia i zrozumieniu, że dziedziczenie publiczne w zasadzie modeluje relację jest-czymś, pora przejść do wykorzystania tej wiedzy, czyli do poznania świętego Graala programowania zorientowanego obiektowo, jakim jest polimorfizm.

Z tej lekcji dowiesz się:

- **czym w rzeczywistości jest polimorfizm,**
- **czym są funkcje wirtualne oraz jak z nich korzystać,**
- **czym są abstrakcyjne klasy bazowe oraz jak je deklarować,**
- **czym jest dziedziczenie wirtualne i kiedy z niego korzystać.**

Podstawy polimorfizmu

W języku greckim „poli” oznacza *wiele*, natomiast „morf” znaczy *postać*. Polimorfizm to funkcja języków zorientowanych obiektowo pozwalających na podobne traktowanie obiektów różnych typów. W tej lekcji koncentrujemy się na zachowaniu polimorficznym (nazywanym również *podtypem polimorfizmu*), które można zaimplementować w C++ przy użyciu hierarchii dziedziczenia.

Potrzeba stosowania polimorfizmu

W lekcji 10., zatytułowanej „Dziedziczenie”, dowiedziałeś się, jak klasy `Tuna` i `Carp` stosują dziedziczenie publiczne po klasie `Fish`, co pozwala im na użycie jej metody `Swim()`, co przedstawiono w listingu 10.1. Istnieje jednak możliwość, aby klasy `Tuna` i `Carp` dostarczały własne metody `Tuna::Swim()` i `Carp::Swim()`, dzięki którym tuńczyk i karp mogą pływać odmiennie. Każdy z wymienionych jest również rybą (klasa `Fish`), więc jeśli użytkownik z egzemplarzem obiektu `Tuna` użyje typu klasy bazowej do wywołania `Fish::Swim()`, nastąpi wywołanie ogólnej metody `Fish::Swim()`, a nie `Tuna::Swim()`, nawet pomimo faktu, że klasa bazowa `Fish` jest częścią obiektu `Tuna`. Tego rodzaju problem zademonstrowano w listingu 11.1.

Uwaga

> Wszystkie przykłady przedstawione w tej lekcji zostały skrócone i zawierają jedynie niezbędne polecenia potrzebne do objaśnienia problemu. Dzięki temu udało się ograniczyć ilość wierszy kodu do minimum i jednocześnie zachować czytelność programów.
>
> W trakcie tworzenia kodu powinieneś budować klasy w odpowiedni sposób, a także stosować hierarchię dziedziczenia, która ma sens, pamiętając jednocześnie o projekcie i przeznaczeniu aplikacji.

Listing 11.1. Wywoływanie metod przy użyciu egzemplarza klasy bazowej Fish, który należy do obiektu Tuna

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: public:
6:    void Swim()
7:    {
8:       cout << "Ryba pływa!" << endl;
```

```
 9:    }
10: };
11:
12: class Tuna:public Fish
13: {
14: public:
15:    // Nadpisanie metody Fish::Swim().
16:    void Swim()
17:    {
18:       cout << "Tuńczyk pływa!" << endl;
19:    }
20: };
21:
22: void MakeFishSwim(Fish& InputFish)
23: {
24:    // Wywołanie Fish::Swim().
25:    InputFish.Swim();
26: }
27:
28: int main()
29: {
30:    Tuna myDinner;
31:
32:    // Wywołanie Tuna::Swim().
33:    myDinner.Swim();
34:
35:    // Wysłanie obiektu Tuna jako Fish.
36:    MakeFishSwim(myDinner);
37:
38:    return 0;
39: }
```

Wynik ▼

```
Tuńczyk pływa!
Ryba pływa!
```

Analiza ▼

Klasa `Tuna` specjalizuje klasę `Fish` przy użyciu publicznego dziedziczenia, co pokazano w wierszu 12. Ponadto nadpisuje metodę `Fish::Swim()`. Jak pokazano w wierszu 33., w funkcji `main()` następuje wykonanie bezpośredniego wywołania `Tuna::Swim()` i przekazanie `myDinner` (typu `Tuna`) jako parametru metodzie `MakeFishSwim()`, który interpretuje parametr jako referencję `Fish&` (patrz wiersz 22.). Innymi słowy, dla wywołania

MakeFishSwim(Fish&) nie ma znaczenia, czy przekazanym obiektem jest Tuna — zostaje obsłużony, jakby to był obiekt Fish i następuje wywołanie metody Fish::Swim(). W drugim wierszu danych wyjściowych widać, że ten sam obiekt Tuna wygenerował dane, które nie wskazują na żadną specjalizację (to równie dobrze mógłby być karp).

W idealnej sytuacji użytkownik oczekuje, że obiekt typu Tuna zachowuje się jak tuńczyk, nawet jeśli wywołaną metodą będzie Fish::Swim(). Innymi słowy, po wywołaniu InputFish.Swim() w wierszu 25., użytkownik oczekuje wywołania metody Tuna::Swim(). Tego rodzaju polimorficzne zachowanie, gdzie obiekt znanego typu klasy Fish może zachowywać się jak rzeczywisty typ (w omawianym przypadku klasa potomna Tuna), można zaimplementować przez zadeklarowanie Fish::Swim() jako funkcji wirtualnej.

Zachowanie polimorficzne implementowane przy użyciu funkcji wirtualnych

Dostęp do obiektu typu Fish masz przy użyciu wskaźnika Fish* lub referencji Fish&. Wymieniony obiekt może zostać utworzony jako jedynie Fish bądź też jako część obiektu Tuna lub Carp dziedziczącego po obiekcie Fish. Tego nie wiesz (to jest nieistotne). Metodę Swim() wywołujesz w przedstawiony poniżej sposób przy użyciu wskaźnika lub referencji:

```
pFish->Swim();
myFish.Swim();
```

Można oczekiwać, że obiekt Fish pływa jak tuńczyk, jeśli jest obiektem typu Tuna, lub jak karp, jeśli jest obiektem typu Carp, lub anonimowo, jeśli obiekt Fish nie został utworzony jako część klasy specjalizowanej Tuna lub Carp. Możesz zagwarantować użycie tego rodzaju rozwiązania przez zadeklarowanie funkcji Swim() w klasie bazowej jako wirtualnej:

```
class Base
{
   virtual TypZwrotny NazwaFunkcji(Lista parametrów);
};
class Derived
{
   TypZwrotny NazwaFunkcji(Lista parametrów);
};
```

Użycie słowa kluczowego `virtual` oznacza, że kompilator gwarantuje wywołanie nadpisanego wariantu żądanej metody klasy bazowej. Dlatego też po zadeklarowaniu `Swim()` jako funkcji wirtualnej wywołanie `myFish.Swim()`, gdzie `myFish` jest typu `Fish&`, oznacza wywołanie metody `Tuna::Swim()`, co zaprezentowano w listingu 11.2.

Listing 11.2. Efekt zadeklarowania Fish::Swim() jako metody wirtualnej

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    virtual void Swim()
 7:    {
 8:       cout << "Ryba pływa!" << endl;
 9:    }
10: };
11:
12: class Tuna:public Fish
13: {
14: public:
15:    // Nadpisanie metody Fish::Swim().
16:    void Swim()
17:    {
18:       cout << "Tuńczyk pływa!" << endl;
19:    }
20: };
21:
22: class Carp:public Fish
23: {
24: public:
25:    // Nadpisanie metody Fish::Swim().
26:    void Swim()
27:    {
28:       cout << "Karp pływa!" << endl;
29:    }
30: };
31:
32: void MakeFishSwim(Fish& InputFish)
33: {
34:    // Wywołanie metody wirtualnej Swim().
35:    InputFish.Swim();
36: }
37:
38: int main()
```

```
39: {
40:     Tuna myDinner;
41:     Carp myLunch;
42:
43:     // Wysłanie obiektu Tuna jako Fish.
44:     MakeFishSwim(myDinner);
45:
46:     // Wysłanie obiektu Carp jako Fish.
47:     MakeFishSwim(myLunch);
48:
49:     return 0;
50: }
```

Wynik ▼

```
Tuńczyk pływa!
Karp pływa!
```

Analiza ▼

Implementacja funkcji `MakeFishSwim(Fish&)` nie uległa żadnej zmianie w stosunku do wersji znajdującej się w listingu 11.1. Jednak otrzymane dane wyjściowe są zupełnie inne. Przede wszystkim w ogóle nie została wywołana metoda `Fish::Swim()`, ponieważ istniejące nadpisane warianty metod `Tuna::Swim()` i `Carp::Swim()` mają pierwszeństwo przed `Fish::Swim()`, gdyż została ona zadeklarowana jako funkcja wirtualna. To jest bardzo ważny krok. Oznacza, że nawet jeśli implementacja `MakeFishSwim()` nie zna dokładnego typu obsługiwanego obiektu `Fish`, wywoła odmienne implementacje `Swim()` zdefiniowane w różnych klasach potomnych, a brać będzie pod uwagę jedynie egzemplarz klasy bazowej.

To jest polimorfizm: traktowanie różnych gatunków ryb jako ogólnie ryb (klasa `Fish`), co gwarantuje wykonanie odpowiedniej implementacji funkcji `Swim()` przez typ potomny.

Konieczność stosowania wirtualnych destruktorów

Istnieje jeszcze jeden negatywny aspekt rozwiązania użytego w listingu 11.1 — przypadkowe wywołanie funkcji klasy bazowej w egzemplarzu klasy potomnej, gdy specjalizacja jest dostępna. Co się stanie, gdy funkcja wywoła

operator `delete` względem wskaźnika typu `Base*`, który faktycznie wskazuje egzemplarz typu `Derived`?

Który destruktor powinien zostać wywołany? Spójrz na kod przedstawiony w listingu 11.3

Listing 11.3. Funkcja wywołująca operator delete względem wskaźnika Base*

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    Fish()
 7:    {
 8:       cout << "Utworzono obiekt Fish" << endl;
 9:    }
10:    ~Fish()
11:    {
12:       cout << "Zniszczono obiekt Fish" << endl;
13:    }
14: };
15:
16: class Tuna:public Fish
17: {
18: public:
19:    Tuna()
20:    {
21:       cout << "Utworzono obiekt Tuna" << endl;
22:    }
23:    ~Tuna()
24:    {
25:       cout << "Zniszczono obiekt Tuna" << endl;
26:    }
27: };
28:
29: void DeleteFishMemory(Fish* pFish)
30: {
31:    delete pFish;
32: }
33:
34: int main()
35: {
36:    cout << "Alokacja pamięci dla obiektu Tuna:" << endl;
37:    Tuna* pTuna = new Tuna;
38:    cout << "Usunięcie obiektu Tuna: " << endl;
39:    DeleteFishMemory(pTuna);
```

```
40:
41:    cout << "Inicjalizacja obiektu Tuna na stosie:" << endl;
42:    Tuna myDinner;
43:    cout << "Automatyczne usunięcie obiektu znajdującego się
       ➥poza zakresem: " << endl;
44:
45:    return 0;
46: }
```

Wynik ▼

```
Alokacja pamięci dla obiektu Tuna:
Utworzono obiekt Fish
Utworzono obiekt Tuna
Usunięcie obiektu Tuna:
Zniszczono obiekt Fish
Inicjalizacja obiektu Tuna na stosie:
Utworzono obiekt Fish
Utworzono obiekt Tuna
Automatyczne usunięcie obiektu znajdującego się poza zakresem:
Zniszczono obiekt Tuna
Zniszczono obiekt Fish
```

Analiza ▼

W funkcji `main()` następuje utworzenie w puli wolnej pamięci egzemplarza obiektu klasy `Tuna` przy użyciu operatora `new` (patrz wiersz 37.), a następnie zwolnienie pamięci natychmiast po zastosowaniu funkcji `DeleteFishMemory()` w wierszu 39. Dla porównania, inny egzemplarz obiektu klasy `Tuna` zostaje utworzony na stosie jako zmienna lokalna `myDinner` (patrz wiersz 42.) i wykracza poza zakres po zakończeniu działania funkcji `main()`. Dane wyjściowe zostały wygenerowane za pomocą poleceń `cout` znajdujących się w konstruktorach i destruktorach klas `Fish` i `Tuna`. Zauważ, że obiekty `Tuna` i `Fish` zostały utworzone w puli wolnej pamięci (użycie operatora `new`), ale destruktor obiektu `Tuna` nie został wywołany przez operator `delete`, nastąpiło jedynie wywołanie destruktora obiektu `Fish`. To jest wyraźna różnica w stosunku do konstrukcji i niszczenia lokalnego elementu składowego `muDinner`, gdzie wywoływane są wszystkie konstruktory i destruktory. W listingu 10.7, przedstawionym w lekcji 10., pokazano poprawną kolejność wywoływania konstruktorów i destruktorów klas w hierarchii dziedziczenia — wywołane były wszystkie destruktory, łącznie z `~Tuna()`. Wygląda na to, że w omawianym listingu nie wszystko jest w porządku.

Wymieniony problem oznacza, że kod w destruktorze klasy potomnej zainicjalizowanej przy użyciu operatora `new` w puli wolnej pamięci nie został wywołany, jeśli wykonywana operacja `delete` stosuje wskaźnik typu `Base*`. W takim przypadku zasoby nie zostaną zwrócone, dojdzie do wycieku pamięci — jest to poważny problem, którego rozwiązanie trzeba znaleźć.

Aby uniknąć tego problemu, można użyć wirtualnych destruktorów, co przedstawiono w listingu 11.4.

Listing 11.4. Użycie wirtualnego destruktora w celu zagwarantowania, że destruktory klas potomnych będą wywoływane podczas usuwania wskaźnika typu Base*

```
#include <iostream>
using namespace std;

class Fish
{
public:
   Fish()
   {
      cout << "Utworzono obiekt Fish" << endl;
   }
   virtual ~Fish()    // Destruktor wirtualny!
   {
      cout << "Zniszczono obiekt Fish" << endl;
   }
};

class Tuna:public Fish
{
public:
   Tuna()
   {
      cout << "Utworzono obiekt Tuna" << endl;
   }
   ~Tuna()
   {
      cout << "Zniszczono obiekt Tuna" << endl;
   }
};

void DeleteFishMemory(Fish* pFish)
{
   delete pFish;
}

```

```
34: int main()
35: {
36:    cout << "Alokacja pamięci dla obiektu Tuna:" << endl;
37:    Tuna* pTuna = new Tuna;
38:    cout << "Usunięcie obiektu Tuna: " << endl;
39:    DeleteFishMemory(pTuna);
40:
41:    cout << "Inicjalizacja obiektu Tuna na stosie:" << endl;
42:    Tuna myDinner;
43:    cout << "Automatyczne usunięcie obiektu znajdującego się poza
       ↪zakresem: " << endl;
44:
45:    return 0;
46: }
```

Wynik ▼

```
Alokacja pamięci dla obiektu Tuna:
Utworzono obiekt Fish
Utworzono obiekt Tuna
Usunięcie obiektu Tuna:
Zniszczono obiekt Tuna
Zniszczono obiekt Fish
Inicjalizacja obiektu Tuna na stosie:
Utworzono obiekt Fish
Utworzono obiekt Tuna
Automatyczne usunięcie obiektu znajdującego się poza zakresem:
Zniszczono obiekt Tuna
Zniszczono obiekt Fish
```

Analiza ▼

Jedyna różnica pomiędzy listingami 11.3 i 11.4 polega na użyciu słowa kluczowego `virtual` w wierszu 10., czyli w deklaracji destruktora klasy bazowej `Fish`. Zauważ, że wprowadzona zmiana powoduje wywołanie przez kompilator metody `Tuna::~Tuna()` obok `Fish::~Fish()`, gdy operator `delete` będzie wywoływany względem wskaźnika `Fish*` prowadzącego do obiektu `Tuna`, jak pokazano w wierszu 31. Dane wyjściowe listingu 11.4 pokazują, że sekwencja wywoływania konstruktorów i destruktorów jest taka sama, niezależnie od tego, czy obiekt `Tuna` jest tworzony przy użyciu operatora `new` w puli wolnej pamięci (patrz wiersz 37.), czy na stosie jako zmienna lokalna (patrz wiersz 42.).

Uwaga

Destruktor klasy bazowej zawsze deklaruj jako wirtualny:

```
class Base
{
public:
   virtual ~Base() {};  // Destruktor wirtualny!
};
```

W ten sposób gwarantujesz, że wskaźnik `Base*` nie będzie mógł wywołać operatora `delete` w sposób uniemożliwiający wywołanie destruktora klasy potomnej.

Jak działa funkcja wirtualna? Zrozumienie tabeli funkcji wirtualnych

Uwaga

Ten punkt jest opcjonalny w trakcie nauki stosowania polimorfizmu. Spokojnie możesz go pominąć lub przeczytać, aby zaspokoić ciekawość.

Funkcja `MakeFishSwim(Fish&)` w listingu 11.2 kończy się wywołaniem metod `Carp::Swim()` lub `Tuna::Swim()`, pomimo zdefiniowania w niej wywołania metody `Fish::Swim()`. W trakcie kompilacji kompilator nic nie wie o naturze obiektów, które napotkają wymienioną funkcję, a więc nie może zagwarantować, że ta sama funkcja będzie wywoływała odmienne metody `Swim()` w różnym czasie. Wybór konkretnej metody `Swim()` do wywołania następuje w trakcie działania aplikacji i odbywa się przy użyciu niewidzialnej logiki implementującej polimorfizm. Logika ta jest dostarczana przez kompilator w chwili kompilacji programu.

Spójrz na klasę `Base`, w której zadeklarowano *N* funkcji wirtualnych:

```
class Base
{
public:
   virtual void Func1()
   {
      // Implementacja Func1.
   }
   virtual void Func2()
   {
      // Implementacja Func2.
   }
   // itd.
   virtual void FuncN()
   {
```

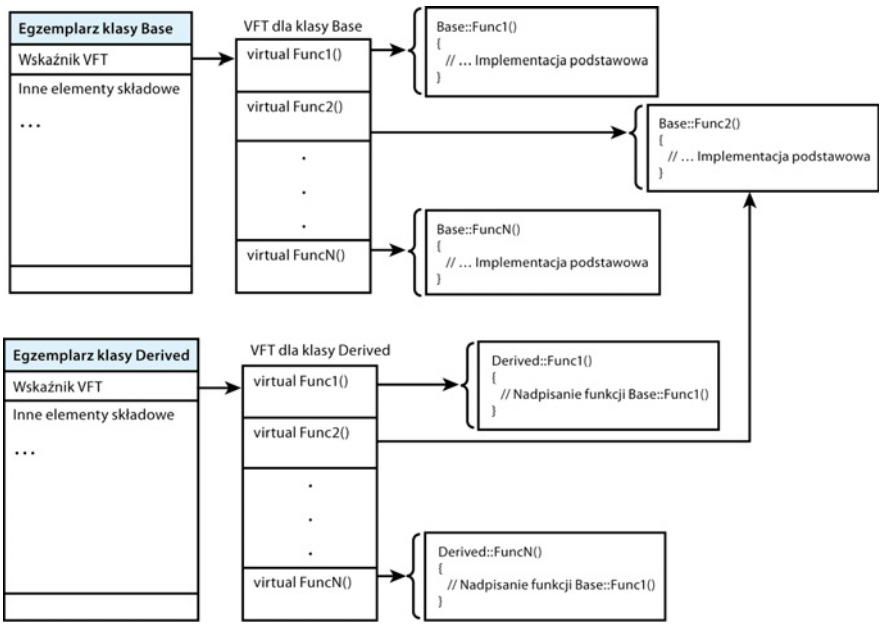
```
        // Implementacja FuncN.
    }
};
```

Klasa `Derived` dziedzicząca po klasie `Base` nadpisuje `Base::Func2()` i udostępnia pozostałe funkcje wirtualne bezpośrednio z klasy bazowej `Base`:

```
class Derived: public Base
{
public:
    virtual void Func1()
    {
        // Func2 nadpisuje Base::Func2().
    }
    // Brak implementacji dla Func2().
    virtual void FuncN()
    {
        // Implementacja FuncN.
    }
};
```

Kompilator znajduje hierarchię dziedziczenia i wie, że klasa bazowa `Base` definiuje określone funkcje wirtualne, które zostały nadpisane w klasie `Derived`. W takiej sytuacji kompilator tworzy tabelę funkcji wirtualnych nazywaną VFT (ang. *Virtual Function Table*). Wspomniana tabela jest tworzona dla każdej klasy implementującej funkcje wirtualne lub dla klasy potomnej, która nadpisuje funkcje wirtualne. Innymi słowy, klasy `Base` i `Derived` otrzymują własne egzemplarze tabeli VFT. Kiedy tworzony jest obiekt wymienionych klas, następuje inicjalizacja ukrytego wskaźnika (nazwijmy go `VFT*`) prowadzącego do odpowiedniej tabeli VFT. Tabelę funkcji wirtualnych można przedstawić w postaci tablicy statycznej zawierającej wskaźniki funkcji, każdy z nich prowadzi do funkcji wirtualnej (lub jej nadpisanej wersji), co pokazano na rysunku 11.1.

Każda tablica składa się ze wskaźników funkcji prowadzących do poszczególnych, dostępnych implementacji funkcji wirtualnych. W przypadku klasy `Derived` wszystkie wskaźniki funkcji, poza jednym, prowadzą do lokalnych implementacji metod wirtualnych w klasie `Derived`. W klasie potomnej `Derived` nie została nadpisana metoda `Base::Func2()` i dlatego odpowiadający jej wskaźnik funkcji prowadzi do implementacji znajdującej się w klasie bazowej `Base`.



RYSUNEK 11.1. Wizualizacja tabeli funkcji wirtualnych dla klas Derived i Base

Oznacza to, że kiedy użytkownik klasy `Derived` używa wywołań:

```
CDerived objDerived;
objDerived.Func2();
```

kompilator sprawdza tabelę VFT klasy `Derived` i upewnia się o wywołaniu implementacji metody `Base::Func2()` w funkcji bazowej. To samo ma zastosowanie względem wywołań nadpisanych metod wirtualnych:

```
void DoSomething(Base& objBase)
{
   objBase.Func1();     // Wywołanie Derived::Func1().
}
int main()
{
   Derived objDerived;
   DoSomething(objDerived);
};
```

W powyższym fragmencie kodu, choć `objDerived` jest interpretowany przez `objBase` jako egzemplarz klasy `Base`, wskaźnik VFT w tym egzemplarzu nadal prowadzi do tej samej tabeli utworzonej dla klasy `Derived`. Dlatego też `Func1()` wywołana przez tabelę VFT to niewątpliwie `Derived::Func1()`.

W ten sposób tabela funkcji wirtualnych pomaga w implementacji (podtypu) polimorfizmu w języku C++.

Dowód na istnienie ukrytego wskaźnika VFT został przedstawiony w listingu 11.5, którego kod przy użyciu operatora `sizeof()` porównuje dwie identyczne klasy — jedna z nich zawiera funkcję wirtualną, natomiast druga nie.

Listing 11.5. Demonstracja istnienia ukrytego wskaźnika VFT w porównaniu dwóch identycznych klas różniących się jedynie deklaracją funkcji wirtualnej

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class SimpleClass
 4: {
 5:    int a, b;
 6:
 7: public:
 8:    void FuncDoSomething() {}
 9: };
10:
11: class Base
12: {
13:    int a, b;
14:
15: public:
16:    virtual void FuncDoSomething() {}
17: };
18:
19: int main()
20: {
21:    cout << "sizeof(SimpleClass) = " << sizeof(SimpleClass) << endl;
22:    cout << "sizeof(Base) = " << sizeof(Base) << endl;
23:
24:    return 0;
25: }
```

Wynik ▼

```
sizeof(SimpleClass) = 8
sizeof(Base) = 12
```

Analiza ▼

Przedstawiony przykład został ograniczony do niezbędnego minimum. W kodzie znajdują się dwie klasy — `SimpleClass` i `Base` — które są identyczne pod względem typów i liczby elementów składowych. W klasie `Base` funkcja `FuncDoSomething()` została zadeklarowana jako wirtualna, podczas gdy w klasie `SimpleClass` to jest zwykła funkcja. Różnica w postaci dodanego słowa kluczowego `virtual` powoduje, że kompilator generuje tabelę funkcji wirtualnych dla klasy `Base` i rezerwuje miejsce dla wskaźnika jako elementu ukrytego. W systemie 32-bitowym wskaźnik zabiera dodatkowe cztery bajty, co potwierdza jego istnienie.

Uwaga

C++ pozwala również na wykonanie zapytania do wskaźnika `Base*` i sprawdzenie, czy jest typu `Derived*`. Do tego używany jest operator rzutowania `dynamic_cast`; następnie przeprowadzane jest uruchomienie warunkowe na podstawie wyniku zapytania.

Taki mechanizm nosi nazwę określania typu w trakcie działania programu (ang. *Run Time Type Identification*, RTTI) i najlepiej go unikać, mimo że jest obsługiwany przez większość kompilatorów C++. Konieczność określenia typu obiektu klasy potomnej „za plecami" wskaźnika klasy bazowej jest najczęściej uznawana za przejaw kiepskiego programowania.

Szczegółowe omówienie RTTI i operatora `dynamic_cast` znajdziesz w lekcji 13., zatytułowanej „Operatory rzutowania".

Abstrakcyjne klasy bazowe i funkcje czysto wirtualne

Klasa bazowa, której egzemplarza nie można utworzyć, nosi nazwę *abstrakcyjnej klasy bazowej*. Tego rodzaju klasy bazowe służą do jednego celu: inne klasy mają dziedziczyć po niej. Język C++ pozwala na tworzenie abstrakcyjnych klas bazowych używających funkcji czysto wirtualnych.

Metoda wirtualna jest określana mianem *czysto wirtualnej*, gdy ma deklarację, taką jak przedstawiono poniżej:

```
class AbstractBase
{
public:
   virtual void DoSomething() = 0;   // Metoda czysto wirtualna.
};
```

Powyższa deklaracja informuje kompilator, że metoda `DoSomething()` musi być zaimplementowana przez klasę dziedziczącą po klasie `AbstractBase`:

```
class Derived: public AbstractBase
{
public:
   void DoSomething()    // Metoda czysto wirtualna.
   {
      cout << "Zaimplementowana funkcja wirtualna" << endl;
   }
};
```

Klasa `AbstractBase` wymusza, aby klasa potomna (tutaj `Derived`) zawierała implementację metody wirtualnej `DoSomething()`. Funkcjonalność polegająca na tym, że klasa bazowa wymusza w klasach dziedziczących po niej zapewnienie obsługi metod o podanej nazwie i sygnaturze nosi nazwę interfejsu. Powróćmy raz jeszcze do klasy `Fish`. Wyobraź sobie, że tuńczyk (obiekt klasy `Tuna`) nie może szybko pływać, ponieważ w klasie `Tuna` nie nadpisano metody `Fish::Swim()`. Taka implementacja jest nieprawidłowa i zawiera błąd. Uczynienie klasy `Fish` abstrakcyjną klasą bazową, która zawiera metodę czysto wirtualną `Swim()`, powoduje, że jeśli klasa `Tuna` dziedziczy po klasie `Fish`, to implementuje metodę `Tuna::Swim()` — tuńczyk pływa jak tuńczyk (obiekt klasy `Tuna`), a nie jak ogólnie ryba (obiekt klasy `Fish`). Rozwiązanie przedstawiono w listingu 11.6.

Listing 11.6. Klasa Fish jako abstrakcyjna klasa bazowa dla klas Tuna i Carp

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    // Definicja funkcji czysto wirtualnej o nazwie Swim().
 7:    virtual void Swim() = 0;
 8: };
 9:
10: class Tuna:public Fish
11: {
12: public:
13:    void Swim()
14:    {
15:       cout << "Tuńczyk pływa szybko w morzu!" << endl;
16:    }
17: };
18:
```

```
19: class Carp:public Fish
20: {
21:    void Swim()
22:    {
23:       cout << "Karp pływa powoli w jeziorze!" << endl;
24:    }
25: };
26:
27: void MakeFishSwim(Fish& inputFish)
28: {
29:    inputFish.Swim();
30: }
31:
32: int main()
33: {
34:    // Fish myFish;   // Błąd, brak możliwości utworzenia abstrakcyjnej klasy bazowej.
35:    Carp myLunch;
36:    Tuna myDinner;
37:
38:    MakeFishSwim(myLunch);
39:    MakeFishSwim(myDinner);
40:
41:    return 0;
42: }
```

Wynik ▼

```
Tuńczyk pływa szybko w morzu!
Karp pływa powoli w jeziorze!
```

Analiza ▼

Umieszczony w komentarzu wiersz 34., czyli pierwszy wiersz funkcji `main()`, jest najważniejszy. Pokazuje, że kompilator nie pozwala na utworzenie egzemplarza klasy `Fish`. Zamiast tego oczekuje konkretów, np. specjalizacji klasy `Fish` w postaci obiektu, takiego jak `Tuna`, co ma sens nawet w świecie rzeczywistym. Dzięki zadeklarowanej w wierszu 7. czysto wirtualnej funkcji `Fish::Swim()` zarówno klasa `Tuna`, jak i `Carp` są zmuszone do implementacji metod (odpowiednio) `Tuna::Swim()` i `Carp::Swim()`. W wierszach od 27. do 30. znajduje się implementacja metody `MakeFishSwim(Fish&)` pokazująca, że choć nie można utworzyć egzemplarza abstrakcyjnej klasy bazowej, to nadal można jej użyć jako referencji lub wskaźnika. Dlatego też abstrakcyjne klasy bazowe to bardzo dobry mechanizm pozwalający na zadeklarowanie funkcji, które powinny być zaimplementowane we wszystkich klasach potomnych. Jeżeli

klasa `Trout` będzie dziedziczyła po `Fish`, ale zabraknie w niej implementacji `Trout::Swim()`, wtedy kompilacja zakończy się niepowodzeniem.

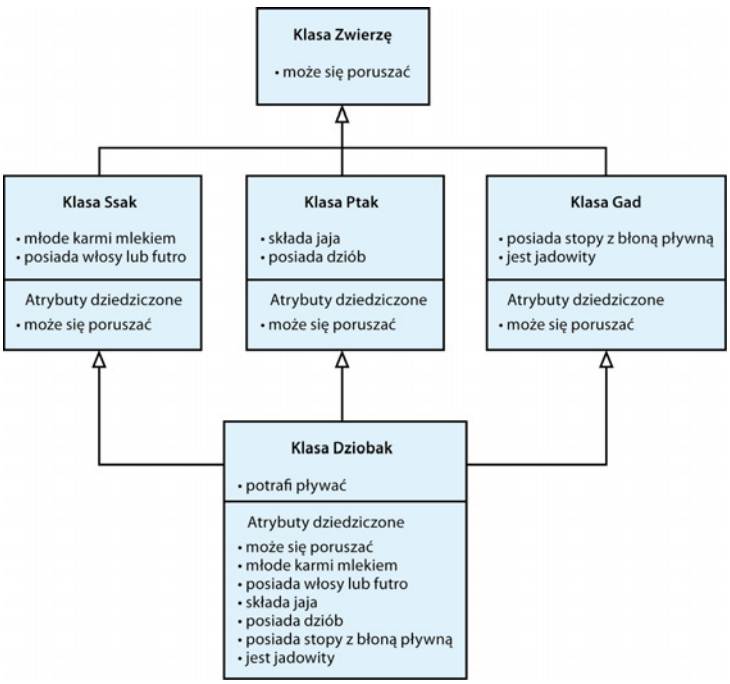
Uwaga

> Abstrakcyjna klasa bazowa jest często określana skrótem ABC (ang. *Abstract Base Class*).

Abstrakcyjne klasy bazowe mogą pomóc w wymuszeniu zastosowania w programie pewnych ograniczeń projektowych.

Użycie dziedziczenia prywatnego do rozwiązania problemu niejednoznaczności semantycznej

W lekcji 10. spotkałeś się z ciekawym przykładem zwierzęcia, jakim jest dziobak, po części ssak, ptak i gad. To był przykład pokazujący, że klasa `Platypus` musi dziedziczyć po klasach `Mammal`, `Bird` i `Reptile`. Jednak każda z trzech wymienionych klas dziedziczy po bardziej ogólnej klasie `Animal`, jak to zostało pokazane na rysunku 11.2.



RYSUNEK 11.2. Wykres pokazujący dziedziczenie wielokrotne w przypadku dziobaka

Co się stanie po utworzeniu egzemplarza klasy `Platypus`? Ile egzemplarzy klasy `Animal` będzie musiało być utworzonych dla jednego egzemplarza klasy `Platypus`? Odpowiedź znajdziesz w programie przedstawionym w listingu 11.7.

Listing 11.7. Określenie liczby egzemplarzy klasy bazowej Animal dla jednego egzemplarza klasy Platypus

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Animal
 4: {
 5: public:
 6:    Animal()
 7:    {
 8:       cout << "Konstruktor obiektu Animal" << endl;
 9:    }
10:
11:    // Przykładowa metoda.
12:    int Age;
13: };
14:
15: class Mammal:public Animal
16: {
17: };
18:
19: class Bird:public Animal
20: {
21: };
22:
23: class Reptile:public Animal
24: {
25: };
26:
27: class Platypus:public Mammal, public Bird, public Reptile
28: {
29: public:
30:    Platypus()
31:    {
32:       cout << "Konstruktor obiektu Platypus" << endl;
33:    }
34: };
35:
36: int main()
37: {
38:    Platypus duckBilledP;
39:
40:    // Usunięcie znaku komentarzu na początku poniższego polecenia uniemożliwia kompilację.
```

```
41:     // Atrybut Age jest niejednoznaczny, ponieważ istnieją trzy egzemplarze klasy
        ↪bazowej Animal.
42:     // duckBilledP.Age = 25;
43:
44:     return 0;
45: }
```

Wynik ▼

```
Konstruktor obiektu Animal
Konstruktor obiektu Animal
Konstruktor obiektu Animal
Konstruktor obiektu Platypus
```

Analiza ▼

Jak możesz zobaczyć w danych wyjściowych, ze względu na wielokrotne dziedziczenie i trzy klasy bazowe, po których dziedziczy klasa `Platypus` dziedzicząca z kolei po klasie `Animal`, program automatycznie tworzy trzy egzemplarze klasy `Animal` dla każdego egzemplarza klasy `Platypus` (patrz wiersz 38.). To śmieszne, ponieważ `Platypus` to pojedyncze zwierzę dziedziczące określone atrybuty po trzech klasach `Mammal`, `Bird` i `Reptile`. Problem związany z liczbą tworzonych egzemplarzy klasy `Animal` nie dotyczy jedynie zużycia pamięci. Klasa `Animal` zawiera publiczny (to ułatwia wyjaśnienie problemu) element składowy w postaci atrybutu typu `int` `Animal::Age`). Kiedy chcesz uzyskać dostęp do atrybutu `Animal::Age` z poziomu egzemplarza `Platypus`, jak pokazano w wierszu 42., otrzymujesz błąd kompilacji, ponieważ kompilator nie wie, którego atrybutu chcesz użyć: `Mammal::Animal::Age`, `Bird::Animal::Age` czy `Reptile::Animal::Age`. To może stać się jeszcze bardziej absurdalne — możesz ustawić wszystkie trzy:

```
duckBilledP.Mammal::Animal::Age = 25;
duckBilledP.Bird::Animal::Age = 25;
duckBilledP.Reptile::Animal::Age = 25;
```

Oczywiste jest, że pojedynczy dziobak powinien mieć tylko jeden atrybut `Age` określający jego wiek. Ponadto nadal chcemy, aby klasa `Platypus` dziedziczyła po trzech klasach bazowych `Mammal`, `Bird` i `Reptile`. Rozwiązaniem jest użycie dziedziczenia wirtualnego. Jeżeli oczekujesz, aby klasa potomna była używana w charakterze klasy bazowej, dobrym pomysłem jest zdefiniowanie jej relacji względem klasy bazowej z zastosowaniem słowa kluczowego `virtual`:

```
class Derived1: public virtual Base
{
   // Elementy składowe i funkcje.
};
class Derived2: public virtual Base
{
   // Elementy składowe i funkcje.
};
```

Lepsza wersja klasy `Platypus` (w rzeczywistości lepsze klasy `Mammal`, `Bird` i `Reptile`) została przedstawiona w listingu 11.8.

Listing 11.8. Demonstracja, jak słowo kluczowe virtual w hierarchii dziedziczenia pomaga w ograniczeniu do jednego liczby egzemplarzy klasy bazowej Animal

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Animal
 4: {
 5: public:
 6:    Animal()
 7:    {
 8:       cout << "Konstruktor obiektu Animal" << endl;
 9:    }
10:
11:    // Przykładowy atrybut.
12:    int Age;
13: };
14:
15: class Mammal:public virtual Animal
16: {
17: };
18:
19: class Bird:public virtual Animal
20: {
21: };
22:
23: class Reptile:public virtual Animal
24: {
25: };
26:
27: class Platypus:public Mammal, public Bird, public Reptile
28: {
29: public:
30:    Platypus()
31:    {
```

```
32:       cout << "Konstruktor obiektu Platypus" << endl;
33:    }
34: };
35:
36: int main()
37: {
38:    Platypus duckBilledP;
39:
40:    // Kompilacja przebiegnie bez problemów, ponieważ jest tylko jeden atrybut Animal::Age.
41:    duckBilledP.Age = 25;
42:
43:    return 0;
44: }
```

Wynik ▼

```
Konstruktor obiektu Animal
Konstruktor obiektu Platypus
```

Analiza ▼

Jeśli otrzymane dane wyjściowe porównasz z danymi wyjściowymi listingu 11.7, przekonasz się, że liczba egzemplarzy klasy `Animal` spadła do jednego, co wreszcie odzwierciedla fakt tworzenia tylko i wyłącznie jednego egzemplarza klasy `Platypus`. Słowo kluczowe `virtual` użyte w relacji pomiędzy klasami `Mammal`, `Bird` i `Reptile` gwarantuje, że po ich zgrupowaniu w ramach obiektu `Platypus` ich wspólny przodek `Animal` będzie istniał tylko w postaci jednego egzemplarza. W ten sposób następuje rozwiązanie wielu problemów — m.in. wiersz 41. w listingu 11.7 zostałby skompilowany bez żadnych problemów.

Uwaga
Uwaga

Problemy powodowane przez hierarchię dziedziczenia zawierającą dwie lub więcej klas bazowych dziedziczących po tej samej klasie bazowej (co oznacza konieczność wskazywania zakresu w przypadku braku dziedziczenia wirtualnego) są określane mianem niejednoznaczności semantyki (ang. *diamond problem*).

Nazwa „diamond" (diament) prawdopodobnie została zainspirowana przez kształt wizualizacji klasy (przedstawiona na rysunku 11.2 wizualizacja jest całkiem prosta, ale jeśli narysujesz linie łączące klasy, zobaczysz kształt diamentu).

Uwaga

Słowo kluczowe `virutal` w języku C++ często jest używane w różnych kontekstach do odmiennych celów. (Uważam, że ktoś chciał zaoszczędzić sobie konieczności wymyślenia innego słowa kluczowego). Oto krótkie podsumowanie użycia wymienionego słowa kluczowego.

Funkcja zadeklarowana jako *wirtualna* oznacza, że w klasie potomnej nastąpi wywołanie nadpisanej wersji funkcji.

Dziedziczenie wirtualne zadeklarowane za pomocą słowa kluczowego `virtual` w deklaracji klas `Derived1` i `Derived2` dziedziczących po klasie `Base` oznacza, że w przypadku istnienia innej klasy potomnej (np. `Derived3`) dziedziczącej po `Derived1` i `Derived2` nadal będzie tworzony tylko jeden egzemplarz klasy `Base` w trakcie tworzenia egzemplarza typu `Derived3`.

Jak widzisz, to samo słowo kluczowe `virtual` jest używane do implementacji dwóch odmiennych koncepcji.

Wirtualne konstruktory kopiujące?

Znak zapytania na końcu tytułu tego podrozdziału jest jak najbardziej uzasadniony. Pod względem technicznym, w kodzie pisanym w języku C++ nie mogą istnieć wirtualne konstruktory kopiujące. Oczywiście, tego rodzaju funkcja pomaga w tworzeniu kolekcji (np. tablicy statycznej) typu `Base*`, gdzie każdy element jest specjalizacją danego typu:

```
// Klasy Tuna, Carp i Trout stosują dziedziczenie publiczne po klasie Fish.
Fish* pFishes[3];
Fishes[0] = new Tuna();
Fishes[1] = new Carp();
Fishes[2] = new Trout();
```

Następnie przypisanie kolekcji innej tablicy tego samego typu — gdzie wirtualny konstruktor kopiujący zapewnia utworzenie głębokiej kopii obiektów klasy potomnej — gwarantuje, że obiekty `Tuna`, `Carp` i `Trout` zostaną skopiowane jako `Tuna`, `Carp` i `Trout`, nawet jeśli konstruktor kopiujący działa na typie `Fish*`.

Cóż, to tylko takie marzenie.

Wykorzystanie wirtualnych konstruktorów kopiujących jest niemożliwe, ponieważ słowo kluczowe `virtual` w kontekście metod klasy bazowej nadpisywanych przez implementacje dostępne w klasie potomnej dotyczy zachowania polimorficznego w trakcie działania programu. Konstruktory z natury nie są polimorficzne i mają możliwość tworzenia jedynie ściśle zdefiniowanego w nich typu i dlatego język C++ nie pozwala na użycie wirtualnych konstruktorów kopiujących.

Mając to wszystko na uwadze, warto dodać, że istnieje pewne obejście w postaci definicji klonu funkcji pozwalającej na działanie w charakterze podobnym do wirtualnego konstruktora kopiującego:

```
class Fish
{
public:
   virtual Fish* Clone() const = 0; // Funkcja czysto wirtualna.
};
class Tuna:public Fish
{
// Inne elementy składowe.
public:
   Tuna * Clone() const  // Klon funkcji wirtualnej.
   {
      return new Tuna(*this);  // Zwrot nowego egzemplarza Tuna, który będzie kopią this.
   }
};
```

W ten sposób funkcja wirtualna `Clone()` jest symulowanym wirtualnym konstruktorem kopiującym, który trzeba wyraźnie wywołać, co przedstawiono w listingu 11.9.

Listing 11.9. Obiekty Tuna i Carp, które obsługują funkcję Clone() jako symulowany wirtualny konstruktor kopiujący

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Fish
 4: {
 5: public:
 6:    virtual Fish* Clone() = 0;
 7:    virtual void Swim() = 0;
 8: };
 9:
10: class Tuna: public Fish
11: {
12: public:
13:    Fish* Clone()
14:    {
15:       return new Tuna (*this);
16:    }
17:
18:    void Swim()
19:    {
20:       cout << "Tuńczyk pływa szybko w morzu" << endl;
21:    }
```

```
22: };
23:
24: class Carp: public Fish
25: {
26:    Fish* Clone()
27:    {
28:       return new Carp(*this);
29:    }
30:    void Swim()
31:    {
32:       cout << "Karp pływa powoli w jeziorze" << endl;
33:    }
34: };
35:
36: int main()
37: {
38:    const int ARRAY_SIZE = 4;
39:
40:    Fish* myFishes[ARRAY_SIZE] = {NULL};
41:    myFishes[0] = new Tuna();
42:    myFishes[1] = new Carp();
43:    myFishes[2] = new Tuna();
44:    myFishes[3] = new Carp();
45:
46:    Fish* myNewFishes[ARRAY_SIZE];
47:    for (int Index = 0; Index < ARRAY_SIZE; ++Index)
48:       myNewFishes[Index] = myFishes[Index]->Clone();
49:
50:    // Wywołanie metody wirtualnej.
51:    for (int Index = 0; Index < ARRAY_SIZE; ++Index)
52:       myNewFishes[Index]->Swim();
53:
54:    // Zwolnienie pamięci.
55:    for (int Index = 0; Index < ARRAY_SIZE; ++Index)
56:    {
57:       delete myFishes[Index];
58:       delete myNewFishes[Index];
59:    }
60:
61:    return 0;
62: }
```

Wynik ▼

```
Tuńczyk pływa szybko w morzu
Karp pływa powoli w jeziorze
Tuńczyk pływa szybko w morzu
Karp pływa powoli w jeziorze
```

Analiza ▼

W wierszach od 40. do 44. funkcji `main()` pokazano deklarację statycznej tablicy wskaźników `Fish*` do klasy bazowej oraz przypisanie poszczególnych elementów do nowo utworzonych obiektów typu (odpowiednio) `Tuna`, `Carp`, `Tuna` i `Carp`. Zwróć uwagę, że wymieniona tablica `myFishes` może przechowywać różne typy, ale powiązane ze sobą wspólną klasą bazową `Fish`. To dobre rozwiązanie, jeśli przypomnisz sobie, że tablice przedstawione we wcześniejszej części książki najczęściej przechowywały elementy pojedynczego typu `int`. Mimo wszystko, jeśli nie uznasz tego za dobre rozwiązanie, tablicę możesz skopiować do nowej tablicy typu `Fish*` o nazwie `myNewFishes` przy użyciu funkcji wirtualnej `Fish::Clone()` w pętli, jak pokazano w wierszu 48. Zauważ, że użyta w programie tablica jest całkiem mała i zawiera tylko cztery elementy. Wprawdzie mogłaby być znacznie większa, ale to nie spowodowałoby większych różnic w logice kopiującej — konieczna byłaby jedynie zmiana parametru warunku kończącego pętlę. W wierszu 52. przeprowadzane jest rzeczywiste wywołanie funkcji wirtualnej `Fish::Swim()` względem każdego elementu przechowywanego w nowej tablicy. Następuje wówczas sprawdzenie, czy funkcja `Clone()` skopiowała obiekt `Tuna` jako `Tuna`, a nie jako `Fish`. Dane wyjściowe listingu potwierdzają, że obiekty `Tuna` i `Carp` zostały skopiowane zgodnie z oczekiwaniami.

TAK	NIE
Pamiętaj o oznaczeniu jako wirtualne tych funkcji klasy bazowej, które mają być nadpisane przez wersje znajdujące się w klasach potomnych. Pamiętaj, że funkcje czysto wirtualne tworzą abstrakcyjną klasę bazową; wspomniane funkcje muszą być zaimplementowane przez klasy potomne. Rozważ użycie dziedziczenia wirtualnego.	Nie zapominaj o umieszczaniu wirtualnych destruktorów w klasach bazowych. Nie zapominaj, że kompilator nie pozwoli na utworzenie samodzielnego egzemplarza abstrakcyjnej klasy bazowej. Nie zapomnij, że dziedziczenie wirtualne to sposób zagwarantowania, iż podstawowa klasa bazowa w hierarchii diamentu będzie miała tylko jeden egzemplarz.

TAK	NIE
	Nie myl funkcji słowa kluczowego `virtual` używanego w tworzeniu hierarchii dziedziczenia z tym samym słowem stosowanym podczas deklaracji funkcji klasy bazowej.

Podsumowanie

W tej lekcji poznałeś użyteczne możliwości, jakie daje w kodzie C++ tworzenie hierarchii dziedziczenia przy użyciu polimorfizmu. Dowiedziałeś się, jak deklarować i definiować funkcje wirtualne — gwarantują one, że implementacja klasy potomnej nadpisze dane funkcje klasy bazowej, nawet jeśli egzemplarz klasy bazowej będzie używany do wywoływania metody wirtualnej. Przekonałeś się, że funkcje czysto wirtualne są specjalnego typu funkcjami wirtualnymi gwarantującymi brak możliwości utworzenia egzemplarza klasy bazowej. Stanowią więc doskonałe miejsce do definiowania interfejsów, które muszą być stosowane przez klasy potomne. Na końcu poznałeś problem niejednoznaczności semantycznej powstający na skutek dziedziczenia wielokrotnego i dowiedziałeś się, jak dziedziczenie wirtualne pomaga w rozwiązaniu tego problemu.

Pytania i odpowiedzi

Pytanie: Dlaczego miałbym używać słowa kluczowego `virtual` w funkcji klasy bazowej, skoro bez niego kod również się kompiluje?

Odpowiedź: Bez słowa kluczowego `virtual` nie będziesz w stanie zagwarantować, że w przypadku wywołania `objBase.Function()` nastąpi przekierowanie do `Derived::Function()`. Ponadto udana kompilacja kodu nie oznacza automatycznie, że jest dobrej jakości.

Pytanie: Dlaczego kompilator tworzy tabelę funkcji wirtualnych?

Odpowiedź: W celu przechowywania wskaźników gwarantujących wywołanie odpowiedniej funkcji wirtualnej.

Pytanie: Czy klasa bazowa zawsze powinna zawierać wirtualny destruktor?

Odpowiedź: W idealnej sytuacji tak. Tylko wtedy można zagwarantować, że jeśli nastąpi wykonanie operacji `delete` względem wskaźnika typu `Base*`, skutkiem będzie wywołanie destruktora `~Derived()`, o ile destruktor `~Base()` został zadeklarowany jako wirtualny.

```
Base* pBase = new Derived();
delete pBase;
```

Pytanie: Czy abstrakcyjna klasa bazowa jest dobrym rozwiązaniem, skoro nawet nie można utworzyć jej samodzielnego egzemplarza?

Odpowiedź: Abstrakcyjna klasa bazowa nie jest przeznaczona do tworzenia w postaci samodzielnego egzemplarza, zamiast tego jest przeznaczona do dziedziczenia po niej. Zawiera funkcje czysto wirtualne definiujące minimalne wersje funkcji, które mają być zaimplementowane w klasach potomnych. Tym samym przejmuje rolę interfejsu.

Pytanie: Biorąc pod uwagę hierarchię dziedziczenia, czy muszę używać słowa kluczowego `virtual` względem wszystkich deklaracji funkcji wirtualnych, czy jedynie w klasie bazowej?

Odpowiedź: Wystarczające jest posiadanie po prostu jednej deklaracji funkcji wirtualnej, która powinna znajdować się w klasie bazowej.

Pytanie: Czy w abstrakcyjnej klasie bazowej mogę definiować funkcje składowe i mieć atrybuty składowe?

Odpowiedź: Oczywiście, że tak. Pamiętaj, że nie można tworzyć egzemplarza abstrakcyjnej klasy bazowej, ponieważ zawiera ona co najmniej jedną funkcję wirtualną, która musi być implementowana przez klasy potomne.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Modelujesz kształty — okrąg i trójkąt — i chcesz, aby każdy kształt obowiązkowo implementował funkcje `Area()` oraz `Print()`. Jakie rozwiązanie zastosujesz?
2. Czy dla wszystkich klas kompilator tworzy tabelę funkcji wirtualnych?
3. Klasa `Fish` zawiera dwie metody publiczne, jedną funkcję czysto wirtualną oraz kilka atrybutów składowych. Czy to nadal jest abstrakcyjna klasa bazowa?

Ćwiczenia

1. Zaprezentuj hierarchię dziedziczenia, która implementuje pytanie 1. w quizie (dla trójkąta i okręgu).
2. **Łowcy błędów:** Co jest nie tak z poniższym fragmentem kodu?

```
class Vehicle
{
public:
   Vehicle() {}
   ~Vehicle(){}
};
class Car: public Vehicle
{
public:
   Car() {}
   ~Car() {}
};
```

3. Jaka jest kolejność wykonywania konstruktorów i destruktorów w (niepoprawionym) kodzie przedstawionym w poprzednim ćwiczeniu, jeśli egzemplarz klasy `Car` zostanie utworzony i usunięty w poniższy sposób:

```
Vehicle* pMyRacer = new Car;
delete pMyRacer;
```

Lekcja 12

Typy operatorów i ich przeciążanie

Słowo kluczowe class pozwala na hermetyzację nie tylko danych i metod, ale również operatorów ułatwiających przeprowadzanie operacji na obiektach danej klasy. Operatorów można używać do wykonywania operacji, takich jak przypisywanie lub dodawanie obiektów klasy, w sposób podobny do sposobu przeprowadzania tego rodzaju operacji na liczbach całkowitych, co pokazano w lekcji 5., zatytułowanej „Wyrażenia, instrukcje i operatory". Ponadto, podobnie jak funkcje, operatory mogą być przeciążane.

Z tej lekcji dowiesz się:

- **jak używać słowa kluczowego `operator`,**
- **czym są operatory jedno- i dwuargumentowe,**
- **jak przeprowadzać konwersję operatorów,**
- **czym jest operator przeniesienia w C++11,**
- **jakie operatory nie mogą być ponownie definiowane.**

Czym są operatory w C++?

Na poziomie syntaktycznym istnieje bardzo mała różnica między operatorem i funkcją — sprowadza się do użycia słowa kluczowego `operator`. Deklaracja operatora przypomina deklarację funkcji:

```
wartość_zwracana operator symbol_operatora (... lista parametrów...);
```

W powyższym przykładzie *symbol_operatora* mógłby być dowolnym z dostępnych typów operatorów, które programista może zdefiniować, takim jak operator + (dodawanie), && (logiczne AND) itd. Operand pomaga kompilatorowi w odróżnieniu jednego operatora od innego. Rodzi się więc pytanie: skoro język obsługuje funkcje, to w jakim celu C++ oferuje również operatory?

Wyobraź sobie, że używasz klasy narzędziowej o nazwie `Date`, która hermetyzuje dzień, miesiąc i rok:

```
Date Holiday (25, 12, 2011); // Inicjalizacja dla 25 grudnia 2011 roku.
```

Jeśli będziesz chciał, aby wskazywaną datą był kolejny dzień — w omawianym przypadku to 26 grudnia 2011 roku — która z poniższych opcji będzie wygodniejsza i bardziej intuicyjna?

```
Opcja 1.:
++Holiday;
```

Opcja 2.:

```
Holiday.Increment(); // 26 grudnia 2011 roku.
```

Operator inkrementacji bez wątpienia jest znacznie lepszym rozwiązaniem niż funkcja `Increment()`. W obu przypadkach otrzymujemy taki sam wynik, jednak mechanizm bazujący na operatorze wykorzystuje intuicyjną i łatwą do zrozumienia (więc także łatwą w obsłudze) implementację. Implementacja operatora < w klasie `Date` pomoże w porównywaniu dwóch dat, np.:

```
if(Date1 < Date2)
{
   // Wykonanie pewnej operacji.
}
else
{
   // Wykonanie innej operacji.
}
```

Operatory znajdują się ponad klasami, takimi jak przeznaczone do zarządzania datami. Wyobraź sobie operator dodawania (+), który pozwala na łatwe łączenie ciągów tekstowych w klasie narzędziowej, takiej jak `MyString` przedstawionej w listingu 9.9:

```
MyString sayHello ("Witaj, ");
MyString sayWorld ("świecie");
MyString sumThem (sayHello + sayWorld);  // Użycie operatora+ (niedostępnego
w listingu 9.9).
```

> Dodatkowy wysiłek związany z implementacją odpowiednich operatorów bardzo szybko zwraca się w postaci łatwości, z jaką można używać danej klasy.

Uwaga

Na bardzo ogólnym poziomie operatory w C++ można podzielić na dwa rodzaje: operatory jednoargumentowe i operatory dwuargumentowe.

Operatory jednoargumentowe

Jak sama nazwa wskazuje, operatory funkcjonujące z pojedynczym operandem są nazywane *operatorami jednoargumentowymi*. Typowa definicja operatora jednoargumentowego zaimplementowana jako funkcja globalna bądź statyczna funkcja składowa jest następująca:

```
wartość_zwracana operator typ_operatora (typ_parametru)
{
   // ... implementacja
}
```

Operator jednoargumentowy będący elementem składowym klasy jest zdefiniowany jako:

```
wartość_zwracana operator typ_parametru()
{
   // ... implementacja
}
```

Typy operatorów jednoargumentowych

Operatory jednoargumentowe, które mogą być przeciążone (lub ponownie zdefiniowane), zostały wymienione w tabeli 12.1.

Tabela 12.1. Operatory jednoargumentowe

Operator	Nazwa
`++`	Inkrementacja
`--`	Dekrementacja
`*`	Dereferencja wskaźnika
`->`	Wybór elementu składowego
`!`	Logiczne NOT
`&`	Adres
`~`	Negacja bitowa
`+`	Jednoargumentowe dodawanie
`-`	Jednoargumentowa negacja
`Operatory konwersji`	Operatory konwersji

Programowanie jednoargumentowego operatora inkrementacji i dekrementacji

Prefiks jednoargumentowego operatora inkrementacji (++) może być programowany przy użyciu przedstawionej poniżej składni w deklaracji klasy:

```
// Jednoargumentowy operator inkrementacji (prefiks).
Date& operator ++ ()
{
   // Kod implementacji operatora.
   return *this;
}
```

Postfiks operatora inkrementacji (++) ma inną wartość zwrotną i parametr danych wejściowych (nie zawsze używany):

```
Date operator ++ (int)
{
   // Zachowanie kopii bieżącego stanu obiektu przed przeprowadzeniem inkrementacji dnia.
   Date Copy (*this);

   // Kod implementacji operatora (odpowiedzialny za inkrementację obiektu this).

   // Przywrócenie stanu sprzed inkrementacji.
   return Copy;
}
```

Prefiksowe i postfiksowe operatory dekrementacji mają składnię podobną do składni operatorów inkrementacji, ale w deklaracji znajdują się dwa minusy (--) zamiast dwóch plusów (++). W listingu 12.1 przedstawiono prostą klasę `Date`, która pozwala na przeprowadzanie inkrementacji dni przy użyciu operatora ++.

Listing 12.1. Klasa kalendarza obsługująca dni, miesiące i lata, która pozwala na przeprowadzanie inkrementacji dni

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5: private:
 6:    int Day;  // Zakres: 1 - 30 (przyjmujemy założenie, że wszystkie miesiące mają
       ↪po 30 dni!
 7:    int Month;
 8:    int Year;
 9:
10: public:
11:    // Konstruktor inicjalizujący obiekt dla dnia, miesiąca i roku.
12:    Date (int InputDay, int InputMonth, int InputYear)
13:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
14:
15:    // Jednoargumentowy operator inkrementacji (prefiks).
16:    Date& operator ++ ()
17:    {
18:       ++Day;
19:       return *this;
20:    }
21:
22:    // Jednoargumentowy operator dekrementacji (prefiks).
23:    Date& operator – ()
23:    Date& operator -- ()
24:    {
25:       --Day;
27:    }
28:
29:    void DisplayDate ()
30:    {
31:       cout << Day << " / " << Month << " / " << Year << endl;
32:    }
33: };
34:
35: int main ()
36: {
37:    // Utworzenie i inicjalizacja obiektu wraz z datą 25 grudnia 2011 roku.
```

```
38:    Date Holiday (25, 12, 2011);
39:
40:    cout << "Obiekt został zainicjalizowany z datą: ";
41:    Holiday.DisplayDate ();
42:
43:    // Zastosowanie prefiksowego operatora inkrementacji.
44:    ++ Holiday;
45:
46:    cout << "Data po zastosowaniu prefiksowego operatora inkrementacji: ";
47:
48:    // Wyświetlenie daty po inkrementacji.
49:    Holiday.DisplayDate ();
50:
51:    -- Holiday;
52:    -- Holiday;
53:
54:    cout << "Data po dwukrotnym zastosowaniu prefiksowego operatora
       ↪dekrementacji: ";
55:    Holiday.DisplayDate ();
56:
57:    return 0;
58: }
```

Wynik ▼

```
Obiekt został zainicjalizowany z datą: 25 / 12 / 2011
Data po zastosowaniu prefiksowego operatora inkrementacji: 26 / 12 / 2011
Data po dwukrotnym zastosowaniu prefiksowego operatora dekrementacji:
↪24 / 12 / 2011
```

Analiza ▼

Interesujące nas operatory znajdują się w wierszach od 16. do 27. i pomagają w inkrementacji obiektów klasy `Date` w zakresie dodawania lub odejmowania dni, jak przedstawiono w wierszach 44., 51. i 52. funkcji `main()`. Prefiksowe operatory inkrementacji to te, które najpierw przeprowadzają operację inkrementacji i zwracają referencję do tego samego obiektu.

Uwaga

> Przedstawiona wersja klasy `Date` zawiera jedynie minimalną implementację, co pozwala na zmniejszenie liczby wierszy kodu i koncentrację na sposobie implementacji prefiksowych operatorów inkrementacji (`++`) i dekrementacji (`--`). W kodzie przyjęto założenie, że miesiące mają po 30 dni.

Aby zapewnić obsługę postfiksowych operatorów inkrementacji i dekrementacji wystarczy po prostu umieścić w klasie Date przedstawiony poniżej fragment kodu:

```
// Operator postfiksowy różni się od prefiksowego typem wartości zwrotnej oraz parametrami.
Date operator ++ (int)
{
   // Zachowanie kopii bieżącego stanu obiektu przed przeprowadzeniem inkrementacji dnia.
   Date Copy (Day, Month, Year);

   ++Day;

   // Przywrócenie stanu sprzed inkrementacji.
   return Copy;
}
// Postfiksowy operator dekrementacji.
Date operator – (int)
{
   Date Copy (Day, Month, Year);

   –Day;

   return Copy;
}
```

Kiedy przygotowana wersja klasy Date obsługuje prefiksowe i postfiksowe operatory inkrementacji i dekrementacji, wtedy masz możliwość użycia obiektów tej klasy za pomocą następującej składni:

```
Date Holiday (25, 12, 2011);  // Utworzenie egzemplarza klasy.
++ Holiday;  // Użycie prefiksowego operatora inkrementacji (++).
Holiday ++;  // Użycie postfiksowego operatora inkrementacji (++).
– Holiday;  // Użycie prefiksowego operatora dekrementacji (--).
Holiday –;  // Użycie postfiksowego operatora dekrementacji (--).
```

Uwaga

Jak pokazano w implementacji operatorów postfiksowych, przed przeprowadzeniem operacji inkrementacji lub dekrementacji wykonywana jest kopia istniejącego stanu obiektu, który później będzie przywrócony.

Innymi słowy, jeśli masz wybór pomiędzy użyciem polecenia `++obiekt;` lub `obiekt++;` do przeprowadzenia tylko inkrementacji, wtedy powinieneś zastosować pierwsze z wymienionych poleceń, aby uniknąć tworzenia kopii tymczasowej, która nie będzie używana.

Programowanie operatorów konwersji

Jeżeli użyjesz listingu 12.1 i do funkcji `main()` dodasz poniższy wiersz kodu:

```
cout << Holiday;  // Błąd w przypadku braku operatora konwersji.
```

wtedy kompilacja listingu zakończy się niepowodzeniem i wygenerowaniem błędu o komunikacie podobnym do: „Błąd: binarne '<<' : brak operatora, który pobiera prawy operand typu 'Date' (lub nie można przeprowadzić konwersji)”. Błąd ten oznacza, że polecenie `cout` nie potrafi zinterpretować egzemplarza klasy `Date`, ponieważ klasa `Date` nie oferuje obsługi odpowiednich operatorów.

Jednak polecenie `cout` doskonale działa wraz z `const char*`:

```
std::cout << "Witaj, świecie"; // Użycie const char* działa!
```

Dlatego też zmuszenie polecenia `cout` do pracy z obiektem typu `Date` jest proste i sprowadza się do dodania operatora zwracającego wartość w postaci `const char*`:

```
operator const char*()
{
    // Implementacja operatora, którego wartością zwrotną jest char*.
}
```

Prostą implementację wymienionego operatora przedstawiono w listingu 12.2.

Listing 12.2. Implementacja operatora konwersji const char* dla klasy Date

```
 0: #include <iostream>
 1: #include <sstream>
 2: #include <string>
 3: using namespace std;
 4:
 5: class Date
 6: {
 7: private:
 8:     int Day;  // Zakres: 1 - 30 (przyjmujemy założenie, że wszystkie miesiące mają po 30 dni!
 9:     int Month;
10:     int Year;
11:
12:     string DateInString;
13:
14: public:
15:
16:     // Konstruktor inicjalizujący obiekt dla dnia, miesiąca i roku.
17:     Date (int InputDay, int InputMonth, int InputYear)
```

```
18:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
19:
20:    operator const char*()
21:    {
22:       ostringstream formattedDate;
23:       formattedDate << Day << " / " << Month << " / " << Year;
24:
25:       DateInString = formattedDate.str();
26:       return DateInString.c_str();
27:    }
28: };
29:
30: int main ()
31: {
32:    // Utworzenie i inicjalizacja obiektu wraz z datą 25 grudnia 2011 roku.
33:    Date Holiday (25, 12, 2011);
34:
35:    cout << "Dzień świąteczny: " << Holiday << endl;
36:
37:    return 0;
38: }
```

Wynik ▼

```
Dzień świąteczny: 25 / 12 / 2011
```

Analiza ▼

Zaleta implementacji w wierszach od 20. do 27. operatora `const char*` jest widoczna w funkcji `main()`, a dokładnie wierszu 35. Egzemplarz klasy `Date` może być bezpośrednio użyty w poleceniu `cout`, bo wykorzystany został fakt, że polecenie `cout` rozpoznaje `const char*`. Kompilator automatycznie używa danych wyjściowych odpowiedniego (w omawianym przypadku jedynego dostępnego) operatora i dostarcza go do polecenia `cout`, które wyświetla datę na ekranie. W przedstawionej implementacji operatora `const char*` użyto `std::ostringstream` w celu konwersji elementów składowych w postaci liczb całkowitych na postać obiektów `std::string`, jak pokazano w wierszach 23. i 25. Nie można bezpośrednio zwrócić `formattedDate.str()`, kopia jest przechowywana w prywatnym elemencie składowym `Date::DateInString` (patrz wiersz 25.), ponieważ `formattedDate` to zmienna lokalna niszczona po zakończeniu działania operatora. Dlatego też wskaźnik otrzymany przez `str()` będzie nieprawidłowy po zakończeniu działania operatora.

Zaprezentowany operator otwiera nowe możliwości w zakresie wykorzystania klasy `Date`. Obiekt `Date` można nawet bezpośrednio przypisać do ciągu tekstowego:

```
string strHoliday (Holiday); // OK! Kompilator wywołuje operator const char*.
strHoliday = Date(11, 11, 2011); // Również OK!
```

Uwaga

Twórz tyle operatorów, ile będzie używanych w klasie. Jeżeli w aplikacji obiekt `Date` ma zostać przedstawiony w postaci liczby całkowitej, możesz utworzyć następujący operator:

```
operator int()
{
    // Miejsce na kod przeprowadzający konwersję.
}
```

W ten sposób egzemplarz klasy `Date` będzie mógł zostać użyty jako liczba całkowita lub skonwertowany na jej postać.

```
FunkcjaPobierającaInt(Date(25, 12, 2011));
```

Programowanie operatora dereferencji (*) i operatora wyboru elementu składowego (->)

Te dwa operatory są prawdopodobnie najczęściej używane w aplikacjach podczas programowania klas sprytnych wskaźników. Sprytne wskaźniki (ang. *smart pointer*) to klasy opakowujące zwykłe wskaźniki, ich celem jest ułatwienie zarządzania pamięcią poprzez zajęcie się kwestiami praw własności i kopiowania. W niektórych przypadkach mogą się nawet przyczynić do poprawy wydajności aplikacji. Sprytne wskaźniki zostaną dokładnie omówione w lekcji 26., „Sprytne wskaźniki". W tej lekcji dowiesz się, jak przeciążanie operatorów pomaga w działaniu sprytnych wskaźników.

Teraz spójrz na sposób użycia klasy `std::unique_ptr` w listingu 12.3 i spróbuj zrozumieć, jak zastosowanie operatorów dereferencji * i wyboru elementu składowego (`->`) powoduje, że klasa sprytnego wskaźnika zachowuje się jak zwykły wskaźnik.

Listing 12.3. Użycie sprytnego wskaźnika unique_ptr w celu zarządzania dynamicznie alokowanym egzemplarzem klasy Date

```
0: #include <iostream>
1: #include <memory>   // To polecenie jest konieczne w celu użycia std::unique_ptr.
```

```
 2: using namespace std;
 3:
 4: class Date
 5: {
 6: private:
 7:    int Day;
 8:    int Month;
 9:    int Year;
10:
11:    string DateInString;
12:
13: public:
14:    // Konstruktor inicjalizujący obiekt dla dnia, miesiąca i roku.
15:    Date (int InputDay, int InputMonth, int InputYear)
16:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
17:
18:    void DisplayDate()
19:    {
20:       cout << Day << " / " << Month << " / " << Year << endl;
21:    }
22: };
23:
24: int main()
25: {
26:    unique_ptr<int> pDynamicAllocInteger(new int);
27:    *pDynamicAllocInteger = 42;
28:
29:    // Użycie sprytnego wskaźnika typu, takiego jak  int*.
30:    cout << "Liczba całkowita to: " << *pDynamicAllocInteger << endl;
31:
32:    unique_ptr<Date> pHoliday (new Date(25, 11, 2011));
33:    cout << "Nowy egzemplarz daty zawiera: ";
34:
35:    // Użycie pHoliday w taki sposób, jakby to był Date*.
36:    pHoliday->DisplayDate();
37:
38:    // Podczas używania unique_ptr poniższe polecenia są zbędne:
39:    // delete pDynamicAllocInteger;
40:    // delete pHoliday;
41:
42:    return 0;
43: }
```

Wynik ▼

```
Liczba całkowita to: 42
Nowy egzemplarz daty zawiera: 25 / 11 / 2011
```

Analiza ▼

W wierszu 26. następuje zadeklarowanie sprytnego wskaźnika typu `int`. W wierszu tym pokazano wzorzec składni inicjalizacji sprytnego wskaźnika klasy `unique_ptr`. Podobnie w wierszu 32. zadeklarowano sprytny wskaźnik do egzemplarza klasy `Date`. W tym momencie skoncentruj się na wzorcu deklaracji i zignoruj dotyczące jej szczegóły.

Uwaga

> Nie przejmuj się, jeśli przedstawiony wzorzec składni wydaje się dziwny. Wzorce będą tematem lekcji 14., zatytułowanej „Wprowadzenie do makr i wzorców".

W powyższym przykładzie nie tylko pokazano, jak sprytny wskaźnik pozwala na użycie zwykłej składni wskaźnika (patrz wiersze 30. i 36.). W wierszu 30. program wyświetla wartość zmiennej `int` przy użyciu wskaźnika `*pDynamicAllocInteger`, podczas gdy w wierszu 36. używane jest polecenie `pHoliday->DisplayDate()`, jakby dwie wymienione zmienne były (odpowiednio) `int*` i `Date*`. Sekret kryje się w klasie sprytnego wskaźnika `std::unique_ptr` implementującej operator (odpowiednio) dereferencji (*) i wyboru elementu składowego (->).

Prosta klasa sprytnego wskaźnika została przedstawiona w listingu 12.4.

Listing 12.4. Implementacja operatora (*) i (->) w prostej klasie sprytnego wskaźnika

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: template <typename T>
 4: class smart_pointer
 5: {
 6: private:
 7:     T* m_pRawPointer;
 8: public:
 9:      smart_pointer (T* pData) : m_pRawPointer (pData) {}      // Konstruktor.
10:      ~smart_pointer () {delete m_pRawPointer ;}          // Destruktor.
11:
12:      T& operator* () const     // Operator dereferencji.
13:      {
14:          return *(m_pRawPointer);
15:      }
16:
17:      T* operator-> () const    // Operator wyboru elementu składowego.
```

```
18:      {
19:          return m_pRawPointer;
20:      }
21:  };
22:
23: class Date
24: {
25: private:
26:    int Day, Month, Year;
27:    string DateInString;
28:
29: public:
30:    // Konstruktor inicjalizujący obiekt dla dnia, miesiąca i roku..
31:    Date (int InputDay, int InputMonth, int InputYear)
32:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
33:
34:    void DisplayDate()
35:    {
36:       cout << Day << " / " << Month << " / " << Year << endl;
37:    }
38: };
39:
40: int main()
41: {
42:    smart_pointer<int> pDynamicInt(new int (42));
43:    cout << "Dynamicznie zaalokowana liczba całkowita = " << *pDynamicInt;
44:
45:    smart_pointer<Date> pDate(new Date(25, 12, 2011));
46:    cout << "Data to = ";
47:    pDate->DisplayDate();
48:
49:    return 0;
50: };
```

Wynik ▼

```
Dynamicznie zaalokowana liczba całkowita = 42
Data to = 25 / 12 / 2011
```

Analiza ▼

Wersja kodu przedstawionego w listingu 12.3 używa własnej klasy `smart_pointer` zdefiniowanej w wierszach od 3. do 24. Wykorzystano składnię deklaracji wzorca, co pozwala na dostosowanie sprytnego wskaźnika do własnych potrzeb; może on prowadzić do dowolnego typu, np. `int` (patrz wiersz 45.) lub `Date` (patrz wiersz 48.). Pokazana w listingu klasa sprytnego wskaźnika zawiera w wierszu 7.

prywatny element składowy typu, do którego prowadzi. W zasadzie celem klasy sprytnego wskaźnika jest automatyzacja zarządzania zasobami wskazywanymi przez wspomniany element składowy, co obejmuje automatyczne zwalnianie pamięci w destruktorze (patrz wiersz 10.). Ten destruktor gwarantuje, że nawet po użyciu operatora `new` nie trzeba wywoływać operatora `delete` i ręcznie zwalniać pamięć, a mimo to wyciek pamięci nie wystąpi. Skoncentruj się na przedstawionej w wierszach od 12. do 15. implementacji operatora * zwracającego wartość `T&` (tzn. referencji do typu, dla którego jest specjalizowany dany wzorzec). Implementacja zwraca referencję do wskazanego egzemplarza, jak widać w wierszu 14. Podobnie operator `->` zdefiniowany w wierszach od 17. do 20. zwraca typ `T*` (tzn. wskaźnik typu, dla którego jest specjalizowany dany wzorzec). W wierszu 19. implementacja operatora `->` zwraca element składowy w postaci wskaźnika. Razem dwa wymienione operatory gwarantują, że klasa `smart_pointer` zajmuje się zarządzaniem pamięcią dla zwykłego wskaźnika i pozwala na użycie funkcji zwykłego wskaźnika; w ten sposób powstaje „sprytny" wskaźnik.

Uwaga

> Klasa sprytnego wskaźnika ma znacznie większe możliwości niż tylko udawanie zwykłych wskaźników i zwalnianie pamięci, kiedy wykroczą one poza zakres. Szczegółowe omówienie tych możliwości zostanie przedstawione w lekcji 26.
>
> Jeżeli zainteresowało Cię przedstawione w listingu 12.3 użycie klasy `unique_ptr`, przyjrzyj się implementacji `unique_ptr` w pliku nagłówkowym `<memory>` dostarczanym przez kompilator lub środowisko IDE. W ten sposób dowiesz się, co musi się dziać w tle, aby obiekty funkcjonowały jak zwykłe wskaźniki.

Operatory dwuargumentowe

Operatory funkcjonujące na dwóch operandach noszą nazwę *operatorów dwuargumentowych*. Definicja operatora dwuargumentowego zaimplementowanego jako funkcja globalna bądź statyczna funkcja składowa jest następująca:

```
wartość_zwracana typ_operatora (parametr1, parametr2)
```

Definicja operatora dwuargumentowego zaimplementowanego jako element składowy klasy jest następująca:

```
wartość_zwracana typ_operatora(parametr)
```

Powodem, dla którego wersja operatora dwuargumentowego zaimplementowanego jako element składowy klasy przyjmuje tylko jeden parametr, jest fakt, że drugi parametr najczęściej wywodzi się z atrybutów samej klasy.

Typy operatorów dwuargumentowych

W tabeli 12.2 podano operatory dwuargumentowe, które mogą być przeciążone (ponownie zdefiniowane) w tworzonych aplikacjach C++.

Tabela 12.2. Operatory dwuargumentowe, które można przeciążyć

Operator	Nazwa
,	przecinek
!=	nierówność
%	reszta z dzielenia (modulo)
%=	reszta z dzielenia (modulo)/przypisanie
&	bitowe AND
&&	logiczne AND
&=	bitowe AND/przypisanie
*	mnożenie
*=	mnożenie/przypisanie
+	dodawanie
+=	dodawanie/przypisanie
-	odejmowanie
-=	odejmowanie/przypisanie
->*	wybór wskaźnik-element składowy
/	dzielenie
/=	dzielenie/przypisanie
<	mniejszy niż
<<	przesunięcie w lewo
<<=	przesunięcie w lewo/przypisanie
<=	mniejszy niż lub równy
=	przypisanie, przypisanie kopiujące, przypisanie przenoszące
==	równy

Tabela 12.2. Operatory dwuargumentowe, które można przeciążyć (cd.)

Operator	Nazwa
>	większy niż
>=	większy niż lub równy
>>	przesunięcie w prawo
>>=	przesunięcie w prawo/przypisanie
^	wyłączające OR
^=	wyłączające OR/przypisanie
|	bitowe OR
|=	bitowe OR/przypisanie
||	logiczne OR
[]	operator indeksowania

Programowanie operatorów dodawania (a+b) i odejmowania (a–b)

Podobnie jak w przypadku operatorów inkrementacji i dekrementacji, dwuargumentowe operatory plus i minus, jeśli zostały zdefiniowane, pozwalają na dodanie lub odjęcie wartości obsługiwanego typu danych obiektu klasy implementującej te operatory. Spójrzmy ponownie na klasę `Date`. Wprawdzie zaimplementowaliśmy już możliwość inkrementacji `Date` powodującej przejście o jeden dzień do przodu, jednak klasa wciąż nie obsługuje możliwości przejścia np. o pięć dni do przodu. Uzyskanie takiej funkcji wymaga implementacji dwuargumentowego operatora dodawania, co przedstawiono w poniższym kodzie, w listingu 12.5.

Listing 12.5. Klasa kalendarza obsługująca dwuargumentowy operator dodawania

```
0: #include <iostream>
1: using namespace std;
2:
3: class Date
4: {
5: private:
6:    int Day, Month, Year;
7:
8: public:
```

```
 9:
10:    // Konstruktor inicjalizujący obiekt dla dnia, miesiąca i roku.
11:    Date (int InputDay, int InputMonth, int InputYear)
12:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:    // Dwuargumentowy operator dodawania.
15:    Date operator + (int DaysToAdd)
16:    {
17:       Date newDate (Day + DaysToAdd, Month, Year);
18:       return newDate;
19:    }
20:
21:    // Dwuargumentowy operator odejmowania.
22:    Date operator - (int DaysToSub)
23:    {
24:       return Date(Day - DaysToSub, Month, Year);
25:    }
26:
27:    void DisplayDate ()
28:    {
29:       cout << Day << " / " << Month << " / " << Year << endl;
30:    }
31: };
32:
33: int main()
34: {
35:    // Utworzenie i inicjalizacja obiektu wraz z datą 25 grudnia 2011 roku.
36:    Date Holiday (25, 12, 2011);
37:
38:    cout << "Dzień świąteczny: ";
39:    Holiday.DisplayDate ();
40:
41:    Date PreviousHoliday (Holiday - 19);
42:    cout << "Poprzedni dzień świąteczny: ";
43:    PreviousHoliday.DisplayDate();
44:
45:    Date NextHoliday(Holiday + 6);
46:    cout << "Następny dzień świąteczny: ";
47:    NextHoliday.DisplayDate ();
48:
49:    return 0;
50: }
```

Wynik ▼

```
Dzień świąteczny: 25 / 12 / 2011
Poprzedni dzień świąteczny: 6 / 12 / 2011
Następny dzień świąteczny: 31 / 12 / 2011
```

Analiza ▼

Wiersze od 14. do 25. zawierają implementacje dwuargumentowych operatorów dodawania (+) i odejmowania (-), które pozwalają na użycie prostej składni dodawania i odejmowania, jak pokazano w funkcji `main()`, a dokładnie w wierszach (odpowiednio) 41. i 45.

Dwuargumentowy operator dodawania będzie także bardzo użyteczny podczas tworzenia klasy ciągu tekstowego. W lekcji 9., zatytułowanej „Klasy i obiekty", przeanalizowaliśmy proste opakowanie ciągu tekstowego przygotowane w postaci klasy `MyString` hermetyzującej zarządzanie pamięcią, kopiowanie ciągu tekstowego w postaci C itd. (patrz listing 9.9). Jednak wymieniona klasa nie obsługiwała łączenia dwóch ciągów tekstowych przy użyciu poniższej składni:

```
MyString Hello("Witaj, ");
MyString World(" świecie");
MyString HelloWorld(Hello + World);   // Błąd: operator + nie został zdefiniowany.
```

Trzeba w tym miejscu powiedzieć, że operator dodawania (+) znacznie ułatwia łączenie ciągów tekstowych przy użyciu klasy `MyString` i dlatego warto włożyć wysiłek w jego przygotowanie:

```
MyString operator+ (const MyString& AddThis)
{
   MyString NewString;

   if (AddThis.Buffer != NULL)
   {
      NewString.Buffer = new char[GetLength() + strlen(AddThis.Buffer) +
1];
      strcpy(NewString.Buffer, Buffer);
      strcat(NewString.Buffer, AddThis.Buffer);
   }

   return NewString;
}
```

Przedstawiony powyżej fragment kodu dodaj do listingu 9.9 z prywatnym domyślnym konstruktorem `MyString()` i pustą implementacją, a będziesz mógł używać składni dodawania. W listingu 12.12, znajdującym się dalej w tej lekcji, przedstawiono wersję klasy `MyString` wraz z zaimplementowanym operatorem (+) i innymi.

Uwaga

Dzięki operatorom zwiększa się użyteczność klasy. Jednak należy je implementować, o ile ma to sens. Warto zwrócić uwagę na implementację operatorów dodawania i odejmowania w klasie `Date`, ale tylko operatora dodawania w klasie `MyString`. Wynika to z faktu, że prawdopodobieństwo przeprowadzania na ciągu tekstowym operacji odejmowania jest praktycznie zerowe i tego rodzaju operator jest po prostu zbędny.

Implementowanie operatorów dodawania/przypisania (+=) i odejmowania/przypisania (-=)

Operator dodawania/przypisania pozwala na użycie składni, takiej jak `"a += b";`, która powoduje zwiększenie wartości obiektu `a` o wartość `b`. Użyteczność operatora dodawania/przypisania podczas takiej operacji polega na tym, że może być przeciążany i akceptować różne typy parametru `b`. W przykładzie przedstawionym w listingu 12.6 liczba całkowita zostaje dodana do obiektu `Date`.

Listing 12.6. Użycie operatora dodawania/przypisania i odejmowania/przypisania w celu dodania lub odjęcia dni w podanej dacie o wartość w postaci liczby całkowitej

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5: private:
 6:    int Day, Month, Year;
 7:
 8: public:
 9:
10:    // Konstruktor inicjalizujący obiekt dla dnia, miesiąca i roku.
11:    Date (int InputDay, int InputMonth, int InputYear)
12:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:    // Dwuargumentowy operator dodawania.
15:    void operator+= (int DaysToAdd)
16:    {
17:       Day += DaysToAdd;
18:    }
19:
20:    // Dwuargumentowy operator odejmowania.
```

```
21:    void operator-= (int DaysToSub)
22:    {
23:       Day -= DaysToSub;
24:    }
25:
26:    void DisplayDate ()
27:    {
28:       cout << Day << " / " << Month << " / " << Year << endl;
29:    }
30: };
31:
32: int main()
33: {
34:    // Utworzenie i inicjalizacja obiektu wraz z datą 25 grudnia 2011 roku.
35:    Date Holiday (25, 12, 2011);
36:
37:    cout << "Dzień świąteczny: ";
38:    Holiday.DisplayDate ();
39:
40:    cout << "Dzień świąteczny -= 19 daje: ";
41:    Holiday -= 19;
42:    Holiday.DisplayDate();
43:
44:    cout << "Dzień świąteczny += 25 daje: ";
45:    Holiday += 25;
46:    Holiday.DisplayDate ();
47:
48:    return 0;
49: };
```

Wynik ▼

```
Dzień świąteczny: 25 / 12 / 2011
Dzień świąteczny -= 19 daje: 6 / 12 / 2011
Dzień świąteczny += 25 daje: 31 / 12 / 2011
```

Analiza ▼

Interesujące nas operatory dodawania/przypisania i odejmowania/przypisania są zdefiniowane w wierszach od 14. do 24. Dzięki nim można dodawać lub odejmować liczbę całkowitą od dni, co pokazano w funkcji `main()`:

```
41: Holiday -= 19;
45: Holiday += 25;
```

Omawiana klasa `Date` pozwala teraz użytkownikom na dodawanie lub odejmowanie ich od daty tak, jakby operacje były przeprowadzane na liczbach

całkowitych przy użyciu operatorów pobierających `int` jako parametr. Istnieje również możliwość przeciążenia operatora += i pobrania egzemplarza fikcyjnej klasy `CDays`:

```
// Operator dodawania/przypisania, który dodaje obiekt CDays do istniejącej daty
void operator += (const CDays& mDaysToAdd)
{
   Day += mDaysToAdd.GetDays();
}
```

Uwaga

Operatory mnożenie/przypisanie *=, dzielenie/przypisanie /=, reszta z dzielenia/przypisanie %=, odejmowanie/przypisanie -=, przesunięcie w lewo/przypisanie <<=, przesunięcie w prawo/przypisanie, bitowe XOR/przypisanie ^=, bitowe OR/przypisanie |= i bitowe AND/przypisanie (&=) mają składnię podobną do przedstawionej w listingu 12.6 składni operatora dodawania/przypisania.

Warto zwrócić uwagę, że ostatecznym celem przeciążania operatorów jest to, aby klasy stała się łatwiejsza i bardziej intuicyjna w użyciu. Istnieje wiele sytuacji, kiedy przeciążanie operatorów nie ma sensu. Przykładowo w naszej klasie kalendarza `Date` w ogóle nie używamy operatora bitowego AND i przypisania (&=). Żaden użytkownik klasy nie powinien kiedykolwiek oczekiwać (lub nawet o tym pomyśleć) otrzymania użytecznych wyników po wykonaniu operacji, takiej jak `greatDate &= 20;`.

Przeciążanie operatorów równości (==) i nierówności (!=)

Czego oczekujemy, kiedy użytkownik klasy `Date` będzie porównywał jedną klasę z inną, np. w następujący sposób:

```
if (Date1 == Date2)
{
   // Podejmij działanie.
}
else
{
   // Podejmij inne działanie.
}
```

Ponieważ (jeszcze) nie zdefiniowaliśmy operatora równości, kompilator po prostu wykona binarne porównanie dwóch obiektów i zwróci wartość `true`, jeśli będą one identyczne. W pewnych sytuacjach to może działać (w tym momencie dotyczy to również klasy `Date`). Jednak wynik prawdopodobnie

nie będzie zgodny z oczekiwaniami, jeśli klasa `Date` będzie miała niestatyczny element składowy zawierający wartość w postaci ciągu tekstowego (`char*`), przykładem może być przedstawiona w listingu 9.9 klasa `MyString`. W takim przypadku binarne porównywanie atrybutów składowych będzie w rzeczywistości porównaniem ciągów tekstowych wskaźników, które nie są identyczne (nawet jeśli ciągi tekstowe wskazują identyczną zawartość). Stąd wynikiem porównania będzie wartość `false`.

Dlatego też dobrą praktyką jest zdefiniowanie operatora porównywania. Ogólne wyrażenie operatora równości przedstawia się następująco:

```
bool operator== (const ClassType& compareTo)
{
    // Miejsce na kod porównujący, wartością zwrotną jest true w przypadku równości i false
    // w pozostałych przypadkach.
}
```

W operatorze nierówności można ponownie wykorzystać operator równości:

```
bool operator!= (const ClassType& compareTo)
{
    // Miejsce na kod porównujący, wartością zwrotną jest true w przypadku nierówności
    // i false w pozostałych przypadkach.
}
```

Operator nierówności może być inwersją (logiczne `NOT`) wyniku działania operatora równości. W listingu 12.7 zademonstrowano operatory porównywania zdefiniowane w naszej klasie `Date`.

Listing 12.7. Prezentacja operatorów równości i nierówności

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5: private:
 6:    int Day, Month, Year;
 7:
 8: public:
 9:
10:    // Konstruktor inicjalizujący obiekt dla dnia, miesiąca i roku.
11:    Date (int InputDay, int InputMonth, int InputYear)
12:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:    bool operator== (const Date& compareTo)
15:    {
```

```
16:       return ((Day == compareTo.Day)
17:             && (Month == compareTo.Month)
18:            && (Year == compareTo.Year));
19:    }
20:
21:    bool operator!= (const Date& compareTo)
22:    {
23:       return !(this->operator==(compareTo));
24:    }
25:
26:    void DisplayDate ()
27:    {
28:       cout << Day << " / " << Month << " / " << Year << endl;
29:    }
30: };
31:
32: int main()
33: {
34:    Date Holiday1 (25, 12, 2011);
35:    Date Holiday2 (31, 12, 2011);
36:
37:    cout << "Dzień świąteczny 1: ";
38:    Holiday1.DisplayDate();
39:    cout << "Dzień świąteczny 2: ";
40:    Holiday2.DisplayDate();
41:
42:    if (Holiday1 == Holiday2)
43:       cout << "Operator równości: to jest ten sam dzień" << endl;
44:    else
45:       cout << "Operator równości: to są dwa różne dni" << endl;
46:
47:    if (Holiday1 != Holiday2)
48:       cout << "Operator nierówności: to są dwa różne dni"<< endl;
49:    else
50:       cout << "Operator nierówności: to jest ten sam dzień" << endl;
51:
52:    return 0;
53: }
```

Wynik ▼

```
Dzień świąteczny 1: 25 / 12 / 2011
Dzień świąteczny 2: 31 / 12 / 2011
Operator równości: to są dwa różne dni
Operator nierówności: to są dwa różne dni
```

Analiza ▼

Operator równości (==) to prosta implementacja zwracająca wartość true, jeśli dzień, miesiąc i rok w podanych datach są takie same (patrz wiersze od 14. do 19.). Operator nierówności (!=) po prostu ponownie wykorzystuje kod operatora równości, co widać w wierszu 23. Istnienie tych dwóch operatorów pomaga w porównywaniu dwóch obiektów Date (Holiday1 i Holiday2) w funkcji main(), co przedstawiono w wierszach 42. i 47.

Przeciążanie operatorów <, >, <= i >=

Dzięki kodowi przedstawionemu w listingu 12.7 klasa Date jest inteligentna na tyle, aby stwierdzić, czy dwa obiekty Date są równe lub nierówne. Jednak co się stanie w sytuacji, gdy użytkownik klasy zażąda wykonania operacji sprawdzania podobnych do przedstawionych poniżej?

```
if (Date1 < Date2) {// Podejmij działanie.}
```

lub

```
if (Date1 <= Date2) {// Podejmij działanie.}
```

lub

```
if (Date1 > Date2) {// Podejmij działanie.}
```

lub

```
if (Date >= Date2) {// Podejmij działanie.}
```

Użytkownik naszej klasy bezsprzecznie uzna ją za bardzo użyteczną, jeśli będzie miał możliwość prostego porównania dwóch dat w celu sprawdzenia, która jest wcześniejsza, a która późniejsza. Programista klasy musi zaimplementować takie porównanie, aby klasa była tak przyjazna dla użytkownika i tak intuicyjna w użyciu, jak to tylko możliwe. Implementacja wspomnianego porównania została przedstawiona w listingu 12.8.

Listing 12.8. Implementacja operatorów <, <=, > oraz >=

```
0: #include <iostream>
1: using namespace std;
2:
3: class Date
4: {
5: private:
6:    int Day, Month, Year;
```

```
 7:
 8: public:
 9:
10:    // Konstruktor inicjalizujący obiekt dla dnia, miesiąca i roku.
11:    Date (int InputDay, int InputMonth, int InputYear)
12:       : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:    bool operator== (const Date& compareTo)
15:    {
16:       return ((Day == compareTo.Day)
17:             && (Month == compareTo.Month)
18:            && (Year == compareTo.Year));
19:    }
20:
21:    bool operator< (const Date& compareTo)
22:    {
23:       if (Year < compareTo.Year)
24:          return true;
25:       else if (Month < compareTo.Month)
26:          return true;
27:       else if (Day < compareTo.Day)
28:          return true;
29:       else
30:          return false;
31:    }
32:
33:    bool operator<= (const Date& compareTo)
34:    {
35:       if (this->operator== (compareTo))
36:          return true;
37:       else
38:          return this->operator< (compareTo);
39:    }
40:
41:    bool operator > (const Date& compareTo)
42:    {
43:       return !(this->operator<= (compareTo));
44:    }
45:
46:    bool operator>= (const Date& compareTo)
47:    {
48:       if(this->operator== (compareTo))
49:          return true;
50:       else
51:          return this->operator> (compareTo);
52:    }
53:
54:    bool operator!= (const Date& compareTo)
55:    {
```

```
56:       return !(this->operator==(compareTo));
57:    }
58:
59:    void DisplayDate ()
60:    {
61:       cout << Day << " / " << Month << " / " << Year << endl;
62:    }
63: };
64:
65: int main()
66: {
67:    Date Holiday1 (25, 12, 2011);
68:    Date Holiday2 (31, 12, 2011);
69:
70:    cout << "Dzień świąteczny 1: ";
71:    Holiday1.DisplayDate();
72:    cout << "Dzień świąteczny 2: ";
73:    Holiday2.DisplayDate();
74:
75:    if (Holiday1 < Holiday2)
76:       cout << "operator <: pierwszy dzień świąteczny będzie wcześniej" <<
       ↪endl;
77:
78:    if (Holiday2 > Holiday1)
79:       cout << "operator >: drugi dzień świąteczny będzie później" << endl;
80:
81:    if (Holiday1 <= Holiday2)
82:       cout << "operator <=: pierwszy dzień świąteczny jest wcześniej
       ↪lub tego samego dnia" << endl;
83:
84:    if (Holiday2 >= Holiday1)
85:       cout << "operator >=: drugi dzień świąteczny jest później
       ↪lub tego samego dnia" << endl;
86:
87:    return 0;
88: }
```

Wynik ▼

```
Dzień świąteczny 1: 25 / 12 / 2011
Dzień świąteczny 2: 31 / 12 / 2011
operator <: pierwszy dzień świąteczny będzie wcześniej
operator >: drugi dzień świąteczny będzie później
operator <=: pierwszy dzień świąteczny jest wcześniej lub tego samego dnia
operator >=: drugi dzień świąteczny jest później lub tego samego dnia
```

Analiza ▼

Interesujące nas operatory zostały zaimplementowane w wierszach od 21. do 52., a ponadto częściowo wykorzystano kod operatora równości (==) z listingu 12.7. Warto zwrócić uwagę na sposób ich implementacji: kod jednego ponownie użyto w drugim.

Przedstawione w wierszach od 75. do 84. zastosowanie wymienionych operatorów wewnątrz funkcji `main()` pokazuje, że ich implementacja znacznie ułatwia używanie klasy `CDate`.

Przeciążanie kopiującego operatora przypisania (=)

Zdarzają się sytuacje, gdy chcesz przypisać zawartość egzemplarza klasy innemu egzemplarzowi, np. w następujący sposób:

```
Date Holiday(25, 12, 2011);
Date AnotherHoliday(1, 1, 2010);
AnotherHoliday  = Holiday;  // Użycie kopiującego operatora przypisania.
```

Powyższy fragment kodu powoduje wywołanie domyślnego kopiującego operatora przypisania, który zostanie zbudowany przez kompilator, jeśli w klasie nie umieścisz własnej jego implementacji. W zależności od natury klasy, domyślny konstruktor kopiujący może być nieodpowiedni, zwłaszcza gdy klasa zarządza zasobami, które nie są kopiowane. Aby zapewnić utworzenie głębokiej kopii przy użyciu konstruktora kopiującego, konieczne jest użycie kopiującego operatora przypisania:

```
ClassType& operator= (const ClassType& CopySource)
{
   if(this != &copySource)   // Ochrona przed kopiowaniem do samego siebie.
   {
      // Implementacja operatora przypisania.
   }
   return *this;
}
```

Głęboka kopia jest ważna, jeśli klasa hermetyzuje zwykły wskaźnik, np. tak jak klasa `MyString` przedstawiona w listingu 9.9. Gdy brakuje operatora przypisania, wtedy dostarczany przez kompilator domyślny kopiujący operator przypisania po prostu kopiuje ze źródła do celu adres znajdujący się w `char*`, bez tworzenia głębokiej kopii wskazywanego adresu w pamięci. Jest to rozwiązanie stosowane

w przypadku braku konstruktora kopiującego. Aby zapewnić utworzenie głębokiej kopii podczas operacji przypisania, trzeba zdefiniować kopiujący operator przypisania, co przedstawiono w listingu 12.9.

Listing 12.9. Wyposażona w kopiujący operator przypisania lepsza wersja klasy MyString z listingu 9.9

```
#include <iostream>
using namespace std;

class MyString
{
private:
   char* Buffer;

public:
   // Konstruktor.
   MyString(const char* InitialInput)
   {
      if(InitialInput != NULL)
      {
         Buffer = new char [strlen(InitialInput) + 1];
         strcpy(Buffer, InitialInput);
     }
      else
         Buffer = NULL;
   }

   // Wstaw konstruktor kopiujący z listingu 9.9.
   MyString(const MyString& CopySource);

   // Kopiujący operator przypisania.
   MyString& operator= (const MyString& CopySource)
   {
      if ((this != &CopySource) && (CopySource.Buffer != NULL))
      {
         if (Buffer != NULL)
          delete[] Buffer;

         // Zapewnienie utworzenia głębokiej kopii przez alokację własnego bufora
         ↪w pierwszej kolejności.
         Buffer = new char [strlen(CopySource.Buffer) + 1];

         // Operacja kopiowania źródła do bufora lokalnego.
         strcpy(Buffer, CopySource.Buffer);
      }
     return *this;
```

```
39:    }
40:
41:    // Destruktor.
42:    ~MyString()
43:    {
44:       if (Buffer != NULL)
45:          delete [] Buffer;
46:    }
47:
48:    int GetLength()
49:    {
50:       return strlen(Buffer);
51:    }
52:
53:    operator const char*()
54:    {
55:       return Buffer;
56:    }
57: };
58:
59: int main()
60: {
61:    MyString String1("Witaj, ");
62:    MyString String2(" świecie");
63:
64:    cout << "Przed przypisaniem: " << endl;
65:    cout << String1 << String2 << endl;
66:    String2 = String1;
67:    cout << "Po przypisaniu String2 = String1: " << endl;
68:    cout << String1 << String2 << endl;
69:
70:    return 0;
71: }
```

Wynik ▼

```
Przed przypisaniem:
Witaj, świecie
Po przypisaniu String2 = String1:
Witaj, Witaj,
```

Analiza ▼

W omawianym listingu celowo pominięto konstruktor kopiujący, aby zmniejszyć liczbę wierszy kodu przykładu. Oczywiście, podczas tworzenia tego rodzaju klasy powinieneś wstawić kod wymienionego konstruktora, znajdziesz go w listingu 9.9.

Kopiujący operator przypisania został zaimplementowany w wierszach od 25. do 39. W działaniu jest bardzo podobny do konstruktora kopiującego i przeprowadza na początku sprawdzenie, czy obiekt nie jest zarówno źródłem, jak i celem. Jeżeli wynikiem operacji sprawdzenia będzie `true`, kopiujący operator przypisania dla `MyString` najpierw zwalnia wewnętrzny bufor, a następnie alokuje pamięć dla tekstu pochodzącego ze źródła i używa funkcji `strcpy()` do wykonania kopii, jak przedstawiono w wierszu 36.

Uwaga

Inna subtelna zmiana w listingu 12.9 w porównaniu z listingiem 9.9 polega na zastąpieniu funkcji `GetString()` przez operator `const char*`, co widać w wierszach od 53. do 56. Dzięki wspomnianemu operatorowi użycie klasy `MyString` stało się jeszcze łatwiejsze, o czym możesz się przekonać w wierszu 68., w którym polecenie `cout` wykorzystano do wyświetlenia dwóch egzemplarzy `MyString`.

Ostrzeżenie

Podczas implementacji klasy zarządzającej dynamicznie alokowanymi zasobami, takimi jak ciąg tekstowy w stylu C (`char*`), tablica dynamiczna i konstrukcje jej podobne zawsze gwarantują, że poza konstruktorem i destruktorem zaimplementowałeś (lub przynajmniej rozważyłeś implementację) konstruktor kopiujący i kopiujący operator przypisania.

Jeśli dokładnie nie przeanalizujesz kwestii własności zasobu w trakcie kopiowania obiektu klasy, tego rodzaju klasa jest niekompletna, a nawet niebezpieczna w użyciu.

Wskazówka

W celu utworzenia klasy, której nie można kopiować, należy zadeklarować prywatny konstruktor kopiujący i kopiujący operator przypisania. Deklaracja (nawet pozbawiona implementacji) jest wystarczająca, aby kompilator zgłaszał błąd w trakcie każdej próby skopiowania danej klasy przez przekazywanie jej funkcji poprzez wartości lub podczas przypisywania jednego egzemplarza drugiemu.

Operatory indeksowania

Operatory, które pozwalają na uzyskanie dostępu do klasy przy użyciu stylu tablicy [], są nazywane *operatorami indeksowania*. Typowa składnia operatora indeksowania jest następująca:

```
typ_zwracany& operator [] (typ_indeksowania& indeks);
```

Stąd, w trakcie zapisu klasy, takiej jak `MyString`, hermetyzującej klasę tablicy dynamicznej znaków w `char* Buffer`, operator indeksowania będzie naprawdę ułatwiał losowe uzyskanie dostępu do poszczególnych znaków w buforze:

```
class MyString
{
    // Inne elementy składowe klasy.
  public:
    /*const*/ char& operator [] (int Index) /*const*/
    {
        // Zwrot liczby całkowitej znajdującej się w pozycji Index.
    }
};
```

Przykładowy fragment kodu umieszczony w poniższym listingu 12.10 pokazuje, jak operator indeksowania pomaga użytkownikowi w iteracji poprzez elementy znajdujące się egzemplarzu klasy `MyString` za pomocą zwykłej semantyki tablic.

Listing 12.10. Implementacja operatora indeksowania w klasie MyString w celu umożliwienia losowego dostępu do znaków znajdujących się w obiekcie MyString::Buffer

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class MyString
 5: {
 6: private:
 7:    char* Buffer;
 8:
 9:    // Domyślny konstruktor prywatny.
10:    MyString() {}
11:
12: public:
13:    // Konstruktor.
14:    MyString(const char* InitialInput)
15:    {
16:       if(InitialInput != NULL)
17:       {
18:          Buffer = new char [strlen(InitialInput) + 1];
19:          strcpy(Buffer, InitialInput);
20:       }
21:       else
22:          Buffer = NULL;
23:    }
24:
```

```
25:    // Konstruktor kopiujący: wstaw tutaj kod z listingu 9.9.
26:    MyString(const MyString& CopySource);
27:
28:    // Kopiujący operator przypisania: wstaw tutaj kod z listingu 12.9.
29:    MyString& operator= (const MyString& CopySource);
30:
31:    const char& operator[] (int Index) const
32:    {
33:       if (Index < GetLength())
34:          return Buffer[Index];
35:    }
36:
37:    // Destruktor.
38:    ~MyString()
39:    {
40:       if (Buffer != NULL)
41:          delete [] Buffer;
42:    }
43:
44:    int GetLength() const
45:    {
46:       return strlen(Buffer);
47:    }
48:
49:    operator const char*()
50:    {
51:       return Buffer;
52:    }
53: };
54:
55: int main()
56: {
57:    cout << "Podaj zdanie: ";
58:    string strInput;
59:    getline(cin, strInput);
60:
61:    MyString youSaid(strInput.c_str());
62:
63:    cout << "Użycie operatora [] do wyświetlenia danych wejściowych: " <<
       ↪endl;
64:    for(int Index = 0; Index < youSaid.GetLength(); ++Index)
65:       cout << youSaid[Index] << " ";
66:    cout << endl;
67:
68:    cout << "Podaj indeks z zakresu 0 - " << youSaid.GetLength() - 1 <<
       ↪": ";
69:    int InIndex = 0;
70:    cin >> InIndex;
71:    cout << "Znak danych wejściowych w położeniu liczonym od zera: " <<
       ↪InIndex;
```

```
72:    cout << " to: "<< youSaid[InIndex] << endl;
73:
74:    return 0;
75: }
```

Wynik ▼

```
Podaj zdanie: Operator indeksowania jest przydatny
Użycie operatora [] do wyświetlenia danych wejściowych:
O p e r a t o r  i n d e k s ow a n i a  j e s t  p r z y d a t n y
Podaj indeks z zakresu 0 - 35: 2
Znak danych wejściowych w położeniu liczonym od zera: 2 to: e
```

Analiza ▼

To jest zabawny program, który pobiera zdanie wprowadzone przez użytkownika, tworzy na jego podstawie obiekt `MyString` (patrz wiersz 61.), a następnie używa pętli `for` w celu wyświetlenia poszczególnych znaków ciągu tekstowego przy użyciu operatora indeksowania (`[]`) i składni przypominającej tablicę, co przedstawiono w wierszach 64. i 65. Sam operator indeksowania został zdefiniowany w wierszach od 31. do 35. i zapewnia bezpośredni dostęp do znaku we wskazanym położeniu po wcześniejszym upewnieniu się, że żądane położenie nie wykracza poza bufor `char* Buffer`.

Ostrzeżenie

Użycie słowa kluczowego const w operatorze

Stosowanie słowa kluczowego `const` jest ważne nawet podczas tworzenia operatorów. Zwróć uwagę, jak w listingu 12.10 ograniczono wartość zwrotną operatora indeksowania do `const char&`. Program działa i jest kompilowany nawet w przypadku braku wspomnianego słowa kluczowego. Powodem jego użycia jest chęć uniknięcia tego rodzaju kodu:

```
MyString sayHello("Witaj, świecie");
sayHello[2] = 'k';error: operator[] is const
```

Dzięki użyciu `const` wewnętrzny element składowy `MyString::Buffer` jest chroniony przed wprowadzaniem bezpośrednich modyfikacji z zewnątrz przy użyciu operatora indeksowania. Poza określeniem wartości zwrotnej jako `const`, nastąpiło także zdefiniowanie typu operatora jako `const` w celu zagwarantowania, że operator ten nie będzie mógł modyfikować atrybutów składowych klasy.

Ogólnie rzecz biorąc, stosuj maksymalne możliwe ograniczenia `const`, aby uniknąć przypadkowych modyfikacji danych i zwiększyć ochronę elementów składowych klasy.

Podczas implementacji operatorów indeksowania można usprawnić wersję przedstawioną w listingu 12.10. W wymienionym listingu znajduje się implementacja pojedynczego operatora indeksowania, który działa zarówno w trakcie odczytu, jak i zapisu danych w tablicy dynamicznej.

Istnieje jednak możliwość implementacji dwóch operatorów indeksowania, jednego jako funkcji typu `const` i drugiego jako funkcji innej niż `const`:

```
char& operator [] (int nIndex);  // Używany do zapisu lub zmiany bufora
                                 // we wskazanym indeksie.
char& operator [] (int nIndex) const; // Używany tylko w celu dostępu do znaku
                                      // we wskazanym indeksie.
```

Kompilator jest na tyle sprytny, że wywoła funkcję `const` podczas operacji odczytu, natomiast funkcję inną niż `const` podczas operacji zapisu w obiekcie `MyString`. Dlatego też możesz (jeśli chcesz) umieścić oddzielne funkcjonalności w dwóch funkcjach. Przykładowo jedna funkcja może rejestrować zapis do obiektu, podczas gdy druga może rejestrować odczyt z obiektu. Istnieją jeszcze inne operatory dwuargumentowe (wymieniono je w tabeli 12.2), które również mogą być przeciążone lub ponownie zdefiniowane, ale one nie będą omawiane w tej lekcji. Ich implementacja jest podobna do technik przedstawionych dotychczas.

Inne operatory, np. logiczne bądź bitowe, muszą być zaprogramowane, jeśli ich umieszczenie w klasie spowoduje zwiększenie jej możliwości. Oczywiście, klasa, taka jak `Date`, nie wymaga implementacji operatorów logicznych, podczas gdy klasa wykonująca operacje na ciągach tekstowych i liczbach może wymagać takich operatorów.

Podczas przeciążania operatorów i tworzenia nowych należy pamiętać o przeznaczeniu klasy i sposobach jej używania.

Funkcja operator()

Istnieją operatory, za pomocą których obiekty działają tak, jakby były funkcjami. Znajdują aplikację w bibliotece STL (ang. *Standard Template Library*, standardowa biblioteka wzorców) i zwykle są używane w algorytmach STL. Użycie tych funkcji może dotyczyć podejmowania decyzji (tego rodzaju obiekty funkcji są nazywane *predykatami jednoargumentowymi* lub *dwuargumentowymi*, w zależności od liczby operandów, z którymi działają). Przeanalizujemy

naprawdę prosty obiekt funkcji przedstawiony w listingu 12.11 w celu zrozumienia, co dają obiekty o tak intrygujących nazwach.

Listing 12.11. Obiekt funkcji zaimplementowany za pomocą funkcji operator()

```
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: class CDisplay
 6: {
 7: public:
 8:    void operator () (string Input) const
 9:    {
10:       cout << Input << endl;
11:    }
12: };
13:
14: int main ()
15: {
16:    CDisplay mDisplayFuncObject;
17:
18:    // Odpowiednik wywołania mDisplayFuncObject.operator () ("Wyświetl ten ciąg
       ↪tekstowy!");.
19:   mDisplayFuncObject ("Wyświetl ten ciąg tekstowy!");
20:
21:   return 0;
22: }
```

Wynik ▼

```
Wyświetl ten ciąg tekstowy!
```

Analiza ▼

W wierszach od 8. do 11. została zaimplementowana funkcja `operator()`, która następnie jest używana w funkcji `main()` w wierszu 18. Zwróć uwagę na to, jak kompilator pozwala na użycie `mDisplayFuncObject` jako funkcji w wierszu 18. poprzez jawną konwersję tego, co wygląda na wywołanie funkcji, na wywołanie do `operator()`.

Z tego powodu operator ten również nazywany jest funkcją `operator()`, a obiekt `CDisplay` jest także nazywany *obiektem funkcji* bądź *funktorem*.

Dokładne omówienie tego zagadnienia znajdziesz w lekcji 21., zatytułowanej „Zrozumienie obiektów funkcji”.

C++11

Użycie konstruktora przenoszącego i przenoszącego operatora przypisania w wysokowydajnym programowaniu

Konstruktor przenoszący i przenoszący operator przypisania to funkcje optymalizacji wydajności, które stały się częścią standardu C++11 i gwarantują, że wartości tymczasowe (r-wartości nieistniejące poza poleceniem) nie będą niepotrzebnie kopiowane. To jest szczególnie użyteczne podczas obsługi klas zarządzających dynamicznie alokowanymi zasobami, np. klasy tablicy dynamicznej lub klasy ciągu tekstowego.

Problem niechcianego kroku kopiowania

Spójrz na zaimplementowany w listingu 12.5 operator dodawania. Zauważ, że powoduje on utworzenie kopii i zwrócenie jej. To samo ma miejsce w przypadku operatora odejmowania. Co się stanie po utworzeniu nowego egzemplarza klasy `MyString` przy użyciu poniższej składni:

```
MyString Hello("Witaj, ");
MyString World("świecie");
MyString CPP(" języka C++");
MyString sayHello(Hello + World + CPP);   // Operator +, konstruktor kopiujący.
MyString sayHelloAgain ("nadpisz to");
sayHelloAgain = Hello + World + CPP;   // Operator +, konstruktor kopiujący,
                                       // kopiujący operator przypisania.
```

Ta prosta konstrukcja jest bardzo intuicyjnym sposobem połączenia trzech ciągów tekstowych, użyto w niej dwuargumentowego operatora dodawania (+) utworzonego następująco:

```
MyString operator+ (const MyString& AddThis)
{
   MyString NewString;

   if (AddThis.Buffer != NULL)
   {
      // Skopiowanie do NewString.
   }
```

```
   return NewString;   // Zwrot kopii przez wartość, wywołanie konstruktora kopiującego.
}
```

Wspomniany operator dodawania znacznie ułatwia łączenie ciągów tekstowych przy użyciu intuicyjnych wyrażeń, choć jednocześnie potencjalnie może spowodować problemy związane z wydajnością. Utworzenie `sayHello` wymaga dwukrotnego wykonania kodu operatora, a każde wykonanie operatora skutkuje utworzeniem kopii tymczasowej, ponieważ zwrot `MyString` następuje przez wartość, co powoduje uruchomienie kodu konstruktora kopiującego. Wspomniany konstruktor kopiujący tworzy głęboką kopię do wartości tymczasowej nieistniejącej po wyrażeniu. Dlatego też w trakcie przetwarzania wyrażenia tworzone są kopie tymczasowe (czyli r-wartości), które nie są wymagane nigdzie, poza danym poleceniem. W taki sposób na skutek działania C++ powstaje wąskie gardło. Na szczęście, mamy możliwość zastosowania odpowiedniego rozwiązania.

Przedstawiony problem został wreszcie rozwiązany w standardzie C++11, w którym kompilator rozpoznaje wartości tymczasowe i używa konstruktora przenoszącego oraz przenoszącego operatora przypisania, o ile będą zdefiniowane w klasie.

Zadeklarowanie konstruktora przenoszącego i przenoszącego operatora przypisania

Składnia konstruktora przenoszącego przedstawia się następująco:

```
Class MyClass
{
private:
   Type* PtrResource;

public:
   MyClass(); // Domyślny konstruktor.
   MyClass(const MyClass& CopySource); // Konstruktor kopiujący.
   MyClass& operator= (const MyClass& CopySource); // Kopiujący operator przypisania.

   MyClass(MyClass&& MoveSource) // Konstruktor przenoszący, zwróć uwagę na dwa
                                 // znaki &&.
   {
      PtrResource = MoveSource.PtrResource; // Przejęcie własności, rozpoczęcie
                                            // przeniesienia.
      MoveSource.PtrResource = NULL;
   }
```

```
   MyClass& operator= (MyClass&& MoveSource) // Przenoszący operator przypisania,
                                             // zwróć uwagę na dwa znaki &&.
   {
      if(this != &MoveSource)
      {
         delete [] PtrResource;  // Zwolnienie zasobów.
         PtrResource = MoveSource.PtrResource;  // Przejęcie własności, rozpoczęcie
                                                // przeniesienia.
         MoveSource.PtrResource = NULL;  // Zwolnienie zasobów.
      }
   }
};
```

Deklaracje konstruktora przenoszącego i operatora przypisania są inne niż zwykłego konstruktora kopiującego i kopiującego operatora przypisania, gdzie operator danych wejściowych jest typu `MyClass&&`. Ponadto, ponieważ parametr danych wejściowych jest przenoszonym zasobem, nie może być parametrem typu `const`, bo będzie modyfikowany. Wartość zwrotna pozostaje taka sama jak przeciążone wersje (odpowiednio) konstruktora i operatora przypisania.

Kompilatory zgodne ze standardem C++11 gwarantują, że użyte będą tymczasowe r-wartości konstruktora przenoszącego zamiast konstruktora kopiującego oraz nastąpi wywołanie przenoszącego operatora przypisania zamiast kopiującego operatora przypisania. W implementacji konstruktora przenoszącego i przenoszącego operatora przypisania gwarantujesz, że zamiast kopiowania zasobu nastąpi jego przeniesienie ze źródła do celu. Efekt zastosowania dwóch ostatnio wymienionych dodatków w standardzie C++11 do optymalizacji klasy `MyString` przedstawiono w listingu 12.12.

Listing 12.12. Klasa MyString wraz z konstruktorem przenoszącym, przenoszącym operatorem przypisania, konstruktorem kopiującym oraz kopiującym operatorem przypisania

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:    char* Buffer;
 7:
 8:    // Domyślny konstruktor prywatny.
 9:    MyString(): Buffer(NULL)
10:    {
11:       cout << "Wywołany został konstruktor domyślny" << endl;
```

```
12:    }
13:
14: public:
15:    // Destruktor.
16:    ~MyString()
17:    {
18:       if (Buffer != NULL)
19:          delete [] Buffer;
20:    }
21:
22:    int GetLength()
23:    {
24:       return strlen(Buffer);
25:    }
26:
27:    operator const char*()
28:    {
29:       return Buffer;
30:    }
31:
32:    MyString operator+ (const MyString& AddThis)
33:    {
34:       cout << "Wywołany został operator +: " << endl;
35:       MyString NewString;
36:
37:       if (AddThis.Buffer != NULL)
38:       {
39:          NewString.Buffer = new char[GetLength() +
           ↪strlen(AddThis.Buffer) + 1];
40:          strcpy(NewString.Buffer, Buffer);
41:          strcat(NewString.Buffer, AddThis.Buffer);
42:       }
43:
44:       return NewString;
45:    }
46:
47:    // Konstruktor.
48:    MyString(const char* InitialInput)
49:    {
50:       cout << "Wywołany został konstruktor dla: " << InitialInput <<
       ↪endl;
51:       if(InitialInput != NULL)
52:       {
53:          Buffer = new char [strlen(InitialInput) + 1];
54:          strcpy(Buffer, InitialInput);
55:       }
56:       else
57:          Buffer = NULL;
58:    }
```

```
59:
60:    // Konstruktor kopiujący.
61:    MyString(const MyString& CopySource)
62:    {
63:       cout<<"Konstruktor kopiujący w celu kopiowania z:
       ↪"<<CopySource.Buffer<<endl;
64:       if(CopySource.Buffer != NULL)
65:       {
66:          // Zapewnienie utworzenia głębokiej kopii przez alokację własnego bufora
          ↪w pierwszej kolejności.
67:          Buffer = new char [strlen(CopySource.Buffer) + 1];
68:
69:          // Operacja kopiowania źródła do bufora lokalnego.
70:          strcpy(Buffer, CopySource.Buffer);
71:       }
72:       else
73:          Buffer = NULL;
74:    }
75:
76:    // Kopiujący operator przypisania.
77:    MyString& operator= (const MyString& CopySource)
78:    {
79:       cout<<"Kopiujący operator przypisania w celu kopiowania z:
       ↪"<<CopySource.Buffer<< endl;
80:       if ((this != &CopySource) && (CopySource.Buffer != NULL))
81:       {
82:          if (Buffer != NULL)
83:           delete[] Buffer;
84:
85:          // Zapewnienie utworzenia głębokiej kopii przez alokację własnego bufora
          ↪w pierwszej kolejności.
86:          Buffer = new char [strlen(CopySource.Buffer) + 1];
87:
88:          // Operacja kopiowania źródła do bufora lokalnego.
89:          strcpy(Buffer, CopySource.Buffer);
90:       }
91:
92:      return *this;
93:    }
94:
95:    // Konstruktor przenoszący.
96:    MyString(MyString&& MoveSource)
97:    {
98:       cout << "Konstruktor przenoszący w celu przeniesienia z: " <<
       ↪MoveSource.Buffer << endl;
99:       if(MoveSource.Buffer != NULL)
100:       {
101:          Buffer = MoveSource.Buffer; // Przejęcie własności.
102:          MoveSource.Buffer = NULL;   // Zwolnienie zasobów.
```

```
103:       }
104:     }
105:
106:    // Przenoszący operator przypisania.
107:    MyString& operator= (MyString&& MoveSource)
108:    {
109:      cout<<"Przenoszący operator przypisania w celu przeniesienia z:
          ↪"<<MoveSource.Buffer<<endl;
110:      if((MoveSource.Buffer != NULL) && (this != &MoveSource))
111:      {
112:         delete Buffer; // Zwolnienie pamięci zajmowanej przez bufor.
113:
114:         Buffer = MoveSource.Buffer; // Przejęcie własności.
115:         MoveSource.Buffer = NULL;    // Zwolnienie zasobów.
116:      }
117:
118:      return *this;
119:   }
120: };
121:
122: int main()
123: {
124:    MyString Hello("Witaj, ");
125:    MyString World("świecie");
126:    MyString CPP(" języka C++");
127:
128:    MyString sayHelloAgain ("nadpisz to");
129:    sayHelloAgain = Hello + World + CPP;
130:
131:    return 0;
132: }
```

Wynik ▼

Dane wyjściowe bez użycia w programie konstruktora przenoszącego i przenoszącego operatora przypisania (po umieszczeniu w komentarzu wierszy od 95. do 119.):

```
Wywołany został konstruktor dla: Witaj,
Wywołany został konstruktor dla: świecie
Wywołany został konstruktor dla:  języka C++
Wywołany został konstruktor dla: nadpisz to
Wywołany został operator +
Wywołany został konstruktor domyślny
Konstruktor kopiujący w celu kopiowania z: Witaj, świecie
Wywołany został operator +
Wywołany został konstruktor domyślny
```

```
Konstruktor kopiujący w celu kopiowania z: Witaj, świecie języka C++
Kopiujący operator przypisania w celu kopiowania z: Witaj, świecie języka C++
```

Dane wyjściowe po włączeniu konstruktora przenoszącego i przenoszącego operatora przypisania:

```
Wywołany został konstruktor dla: Witaj,
Wywołany został konstruktor dla: świecie
Wywołany został konstruktor dla:  języka C++
Wywołany został konstruktor dla: nadpisz to
Wywołany został operator +
Wywołany został konstruktor domyślny
Konstruktor przenoszący w celu przeniesienia z: Witaj, świecie
Wywołany został operator +
Wywołany został konstruktor domyślny
Konstruktor przenoszący w celu przeniesienia z: Witaj, świecie języka C++
Przenoszący operator przypisania w celu przeniesienia z: Witaj, świecie
języka C++
```

Analiza ▼

To jest naprawdę długi listing, ale większość kodu została już zaprezentowana w poprzednich przykładach i lekcjach. Najważniejsza część listingu znajduje się w wierszach od 95. do 119. i zawiera implementację konstruktora przenoszącego oraz przenoszącego operatora przypisania. Fragmenty danych wyjściowych, na które wpływ mają nowe dodatki wprowadzone w standardzie C++11, zostały przedstawione pogrubioną czcionką. Zwróć uwagę na istotną zmianę danych wyjściowych po użyciu tej samej klasy, ale bez dodatków wprowadzonych w standardzie C++11. Jeżeli ponownie przyjrzysz się implementacji konstruktora przenoszącego i przenoszącego operatora przypisania, dostrzeżesz, że semantyka przeniesienia została zaimplementowana przez przejęcie własności zasobów ze źródła. Odbywa się to w konstruktorze przenoszącym (patrz wiersz 101.) i przenoszącym operatorze przypisania (patrz wiersz 114.). Po wymienionej operacji natychmiast (czyli w wierszach, odpowiednio, 102. i 115.) następuje przypisanie wartości `null` wskaźnikowi źródła. Dlatego też, nawet mimo zniszczenia źródła przeniesienia, operator `delete` wywoływany przez destruktor zdefiniowany w wierszach od 16. do 20. w zasadzie nie ma nic do zrobienia, ponieważ własność została przeniesiona do obiektu docelowego. Zauważ, że w przypadku braku konstruktora przenoszącego następuje wywołanie konstruktora kopiującego, który wykonuje głęboką kopię wskazanego ciągu tekstowego. Oczywiste jest więc, że konstruktor przenoszący pozwala na oszczędność znacznej ilości czasu przetwarzania na skutek eliminacji niechcianych kroków alokacji pamięci i kopiowania.

Utworzenie konstruktora przenoszącego i przenoszącego operatora przypisania jest zupełnie opcjonalne. W przeciwieństwie do konstruktora kopiującego i kopiującego operatora przypisania, kompilator nie dodaje za programistę ich domyślnej implementacji.

Wymienionych funkcji wprowadzonych w standardzie C++11 używa się do optymalizacji funkcjonowania klas dynamicznie alokowanych zasobów, które w przeciwnym razie będą kopiowane, nawet jeśli ich zasoby są potrzebne jedynie tymczasowo.

Operatory, których nie można ponownie zdefiniować

Pomimo całej elastyczności, jaką C++ daje programiście w dostosowywaniu operatorów do własnych potrzeb, aby klasy były łatwiejsze w użyciu, pewne karty pozostawia dla siebie i nie pozwala na zmianę zachowania konkretnych operatorów, które mają działać w sposób niezmienny. Operatory, których nie można ponownie definiować, zostały wymienione w tabeli 12.3.

Tabela 12.3. Operatory, których nie można przeciążać lub ponownie definiować

Operator	Nazwa
`.`	Wybór elementu składowego.
`.*`	Wybór wskaźnik-element składowy.
`::`	Zakres.
`? :`	Warunkowy operator trójargumentowy.
`sizeof`	Pobranie wielkości typu klasy/obiektu.

TAK	NIE
Przygotuj tyle operatorów, ile jest wymaganych w celu ułatwienia użycia danej klasy. Pamiętaj jednak, aby nie tworzyć więcej operatorów niż to konieczne.	Nie zapominaj, że kompilator dostarcza domyślny kopiujący operator przypisania i konstruktor kopiujący, o ile sam nie utworzysz ich implementacji. Jednak wersje domyślne nie tworzą głębokich kopii zwykłych wskaźników znajdujących się w klasie.

TAK	NIE
W klasach zawierających elementy składowe w postaci zwykłych wskaźników zawsze twórz kopiujący operator przypisania (wraz z konstruktorem kopiującym i destruktorem). Jeżeli korzystasz z kompilatora zgodnego ze standardem C++11, wtedy w klasach zarządzających dynamicznie alokowanymi zasobami, takimi jak tablice danych, zawsze twórz przenoszący operator przypisania (wraz z konstruktorem przenoszącym).	Nie zapominaj, że jeśli nie dostarczysz implementacji przenoszącego operatora przypisania lub konstruktora przenoszącego, kompilator nie utworzy ich za Ciebie. Zamiast tego użyte zostaną zwykły kopiujący operator przypisania i konstruktor kopiujący. Nie zapominaj, że tworzenie operatorów jest opcjonalne, choć znacznie zwiększają one użyteczność klasy. Nie zapominaj, że klasa sprytnego wskaźnika nie jest wskaźnikiem, o ile nie umieścisz w niej implementacji operatora dereferencji (*) i wyboru elementu składowego (->). Wskaźnik nie będzie również wystarczająco sprytny bez implementacji destruktora i dokładnego przeanalizowania przypadków kopiującego przypisania oraz konstrukcji kopiującej.

Podsumowanie

W tej lekcji dowiedziałeś się, że stosowanie operatorów powoduje ogromną różnicę w łatwości użycia klasy. Podczas tworzenia klasy zarządzającej zasobami, np. tablicą dynamiczną lub ciągiem tekstowym, jako minimum poza destruktorem konieczne jest zdefiniowanie konstruktora kopiującego i kopiującego operatora przypisania. Klasa narzędziowa zarządzająca tablicą dynamiczną może działać doskonale wraz z konstruktorem przenoszącym i przenoszącym operatorem przypisania, które gwarantują, że nie będzie tworzona głęboka kopia zasobu dla obiektów tymczasowych. Ponadto dowiedziałeś się, że operatory `.`, `.*`, `::`, `?:` i `sizeof` nie mogą być ponownie definiowane.

Pytania i odpowiedzi

Pytanie: Moja klasa hermetyzuje dynamiczną tablicę liczb całkowitych? Które funkcje i operatory powinny być zaimplementowane jako niezbędne minimum?

Odpowiedź: W trakcie tworzenia tego rodzaju klasy konieczne jest jasne zdefiniowanie zachowania w sytuacji, gdy egzemplarz będzie kopiowany

bezpośrednio do innego egzemplarza poprzez przypisanie lub pośrednio kopiowany przez przekazanie go funkcji za pomocą wartości. Zwykle trzeba zaimplementować konstruktor kopiujący, kopiujący operator przypisania i destruktor. Jeśli chcesz poprawić wydajność działania klasy w określonych przypadkach, należy zaimplementować także konstruktor przenoszący i przenoszący operator przypisania.

Pytanie: Mam egzemplarz obiektu klasy. Chcę zapewnić obsługę składni `cout << obiekt;`. Który operator powinienem zaimplementować?

Odpowiedź: Musisz zaimplementować operator konwersji pozwalający na interpretację klasy jako typu możliwego do obsługi przez `std::cout`. Jednym z możliwych rozwiązań jest zdefiniowanie operatora `char*()`, jak to zrobiono w listingu 12.2.

Pytanie: Chcę utworzyć własną klasę sprytnego wskaźnika. Które funkcje i operatory powinny być zaimplementowane jako niezbędne minimum?

Odpowiedź: Sprytny wskaźnik musi zapewniać możliwość jego użycia jako zwykłego wskaźnika, np.: `*pSmartPtr or pSmartPtr->Func()`. Konieczne jest więc zaimplementowanie operatorów dereferencji (*) i wyboru elementu składowego (->). Oprócz tego, aby wskaźnik był sprytny, należy zadbać o automatyczne zwalnianie pamięci zasobów przez odpowiednie przygotowanie destruktora. Trzeba jasno zdefiniować sposób działania operacji kopiowania i przypisania przez implementację konstruktora kopiującego i kopiującego operatora przypisania lub przez uniknięcie ich zadeklarowania jako prywatnych.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Czy wartość zwrotna operatora indeksowania [] jest typu `const`, czy innego?

   ```
   const Type& operator[](int Index);
   Type& operator[](int Index); // Czy to jest prawidłowe?
   ```

2. Czy kiedykolwiek zadeklarujesz prywatny konstruktor kopiujący lub kopiujący operator przypisania?
3. Czy ma sens zdefiniowanie konstruktora przenoszącego i przenoszącego operatora przypisania dla klasy `Date`?

Ćwiczenia

1. Utwórz operator konwersji dla klasy `Date`, który będzie konwertował przechowywaną datę na unikalną liczbę całkowitą.
2. Utwórz konstruktor przenoszący i przenoszący operator przypisania dla klasy `DynIntegers` hermetyzującej dynamicznie alokowaną tablicę w postaci prywatnego elementu składowego `int*`.

Lekcja 13

Operatory rzutowania

Rzutowanie to mechanizm, przy użyciu którego programista może tymczasowo bądź na stałe zmienić sposób interpretacji obiektu przez kompilator. Nie powoduje to rzeczywistej zmiany samego obiektu — to po prostu zmiana sposobu jego interpretacji. Operatory zmieniające interpretację obiektu nazywane są *operatorami rzutowania*.

Z tej lekcji dowiesz się:

- **do czego są wykorzystywane operatory rzutowania,**
- **dlaczego tradycyjne rzutowanie w stylu C nie jest popularne wśród niektórych programistów C++,**
- **jakie są cztery operatory rzutowania C++,**
- **czym jest koncepcja rzutowania w górę (ang. *upcasting*) i rzutowania w dół (ang. *downcasting*),**
- **dlaczego operatory rzutowania nie zawsze są najlepszym rozwiązaniem.**

Kiedy trzeba skorzystać z rzutowania?

W doskonałym świecie, w którym dobrze napisane aplikacje C++ składają się z bezpiecznych i ściśle określonych typów, nie byłoby konieczne przeprowadzanie rzutowania i stosowanie operatorów rzutowania. Jednak nie żyjemy w świecie idealnym, moduły są tworzone przez wielu różnych programistów, a producenci w pracy często wykorzystują odmienne środowiska. Aby nad tym wszystkim zapanować, kompilatory bardzo często muszą być instruowane o sposobie interpretacji danych, który pozwoli na kompilację kodu źródłowego i zapewni prawidłowe działanie aplikacji.

Rozważmy rzeczywisty przykład: wprawdzie pewne kompilatory C++ mogą obsługiwać `bool` jako ich rodzimy typ danych, jednak w użyciu nadal znajduje się wiele bibliotek, które zostały opracowane jeszcze w latach panowania języka C. Biblioteki te są przeznaczone dla kompilatorów C, więc do przechowywania danych boolowskich używają typu liczb całkowitych. Dlatego też w tych kompilatorach typ `bool` jest podobny do poniższej definicji:

```
typedef unsigned short BOOL;
```

Funkcja zwracająca dane boolowskie będzie więc zadeklarowana jako:

```
BOOL IsX();
```

Jeżeli tego rodzaju biblioteka zostanie użyta wraz z nową aplikacją utworzoną za pomocą najnowszej wersji kompilatora C++, programista musi znaleźć sposób, aby typ danych `bool` był zrozumiały jako funkcja z typem danych `BOOL` zarówno dla jego kompilatora C++, jak i dla biblioteki. W takim przypadku rozwiązaniem jest zastosowanie rzutowania:

```
bool bCPPResult = (bool)IsX (); // Rzutowanie w stylu języka C.
```

Ewolucja języka C++ obnażyła potrzebę powstania nowych operatorów rzutowania C++, co doprowadziło do podziału społeczności programistów C++. W tworzonych przez siebie aplikacjach C++ jedna grupa kontynuuje używanie rzutowania w stylu języka C, natomiast druga skrupulatnie przestawiła się na słowa kluczowe rzutowania wprowadzone przez kompilatory C++. Argumentem pierwszej grupy jest to, że rzutowanie w stylu C++ jest niewygodne w użyciu i jego funkcjonalność czasami różni się w tak małym stopniu od rzutowania w stylu C, iż wartość nowego stylu można uznać za jedynie teoretyczną. Natomiast druga grupa, wyraźnie składająca się z purystów składni C++, wskazuje niedociągnięcia rzutowania w stylu języka C.

Ponieważ rzeczywistość przynosi kod utworzony w obu stylach, warto po prostu zapoznać się z tą lekcją, poznać wady i zalety każdego stylu i na tej podstawie wyrobić sobie opinię.

Dlaczego rzutowanie w stylu C nie jest popularne wśród niektórych programistów C++?

Bezpieczeństwo typów to jeden z symboli, na które programiści C++ przysięgali, kiedy chwalili jakość nowego języka programowania. W rzeczywistości większość kompilatorów C++ nie pozwoli nawet na wykonanie poniższych instrukcji:

```
char* pszString = "Witaj, świecie!";
int* pBuf = pszString;     // Błąd: brak możliwości konwersji char* na int*.
```

... i bardzo słusznie!

Kompilatory C++ powinny nadal zapewniać wsteczną zgodność, aby pozwolić na budowanie starszego kodu. Dlatego też automatycznie zezwalają na zastosowanie składni takiej jak:

```
int* pBuf = (int*)pszString; // Rzutowanie jednego problemu powoduje powstanie
                             // jeszcze większego!
```

Jednak rzutowanie w stylu C wymusza na kompilatorze interpretację celu jako typu, który został wygodnie wybrany przez programistę. W takim przypadku programista nie musi przejmować się tym, czy kompilator zgłasza błąd ze słusznego powodu, po prostu „nakłada kaganiec" kompilatorowi i zmusza go do posłuszeństwa. To — oczywiście — nie eliminuje zagrożenia dostrzeganego przez programistów C++, którzy uznają, że proces rzutowania powoduje powstanie zagrożenia dla typów.

Operatory rzutowania C++

Samej koncepcji rzutowania, mimo wad, nie można odrzucać. W wielu sytuacjach rzutowanie to prawidłowe wymaganie pozwalające na rozwiązanie ważnych kwestii dotyczących zgodności. Język C++ dodatkowo oferuje nowy operator rzutowania, który nie istnieje w programowaniu C, a jest stosowany w sytuacjach wykorzystujących dziedziczenie.

Dostępne w C++ cztery operatory rzutowania to:

- `static_cast`,
- `dynamic_cast`,
- `reinterpret_cast`,
- `const_cast`.

Składnia stosowana podczas używania operatorów rzutowania jest spójna:

```
typ_docelowy wynik = typ_rzutowania <typ_docelowy< (obiekt_do_rzutowania);
```

Użycie static_cast

Operator `static_cast` to mechanizm, który może być używany do konwersji wskaźników między powiązanymi typami oraz do przeprowadzania jawnej konwersji typu dla standardowych typów danych, która w innym przypadku byłaby przeprowadzana automatycznie bądź niejawnie. Jeśli chodzi o wskaźniki, `static_cast` implementuje podstawowy proces sprawdzania przeprowadzany w trakcie kompilacji, który ma zagwarantować, że wskaźnik jest rzutowany na pokrewny z nim typ. Stanowi to usprawnienie w stosunku do stylu C, gdzie wskaźnik do jednego obiektu bez żadnych problemów mógł być rzutowany na zupełnie niepowiązany z nim typ. Gdy użyjemy `static_cast`, wskaźnik można rzutować w górę do typu bazowego lub w dół do typu pochodnego, co zostało pokazane na poniższym fragmencie kodu:

```
Base* pBase = new Derived ();      // Zbudowanie obiektu Derived.
Derived* pDerived = static_cast<Derived*>(pBase);      // OK!

// Obiekt CUnrelated nie jest powiązany z Base przez żadną hierarchię dziedziczenia.
CUnrelated* pUnrelated = static_cast<CUnrelated*>(pBase); // Błąd.

// Powyższe rzutowanie nie jest dozwolone, ponieważ oba typy nie są ze sobą w żaden sposób
// powiązane.
```

Uwaga

Rzutowanie `Derived*` na `Base*` jest nazywane rzutowaniem w górę i może być przeprowadzone bez wyraźnego użycia operatora rzutowania:

```
Derived objDerived;
Base* pBase = &objDerived;  // OK!
```

Rzutowanie `Base*` na `Derived*` jest nazywane rzutowaniem w dół i nie może być przeprowadzone bez wyraźnego użycia operatora rzutowania:

```
Derived objDerived;
Base* pBase = &objDerived; // Rzutowanie w górę -> OK!
Derived* pDerived = pBase; // Błąd: rzutowanie w dół wymaga wyraźnego użycia
                           // operatora rzutowania.
```

Jednak zwróć uwagę, że `static_cast` sprawdza tylko, czy typy wskaźników są ze sobą powiązane. W trakcie działania programu *nie następuje* żadna operacja sprawdzania. Dlatego też, korzystając ze `static_cast`, programista nadal może popełnić błąd poniższego rodzaju:

```
Base* pBase = new Base ();
Derived* pDerived = static_cast<Derived*>(pBase); // Kompilator wciąż
                                                  // nie znajduje błędów!
```

W powyższym kodzie `pDerived` w rzeczywistości prowadzi do obiektu częściowego `Derived`, ponieważ faktycznie obiekt jest typu `Base()`. Ponieważ `static_cast` przeprowadza sprawdzanie jedynie w trakcie kompilacji, upewniając się, że weryfikowane typy są ze sobą powiązane, i nie przeprowadza sprawdzania w trakcie działania programu, wywołanie `pDerived->SomeDerivedClassFunction()` zostanie skompilowane, choć prawdopodobnie będzie skutkowało nieprzewidzianym zachowaniem programu.

Poza pomocą podczas rzutowania w górę i dół w wielu sytuacjach `static_cast` pomaga w jawnych operacjach rzutowania niejawnego i koncentruje na nich uwagę programisty bądź osoby przeglądającej kod:

```
double dPi = 3.14159265;
int Num = static_cast<int>(dPi); // Rzutowanie jawne, które w innym przypadku
                                 // pozostaje niejawne.
```

W powyższym fragmencie kodu `Num = dPi` będzie działało równie doskonale i spowoduje osiągnięcie tego samego efektu. Jednak użycie `static_cast` skupia uwagę programisty na rzutowaniu i wskazuje (osobie znającej `static_cast`), że kompilator przeprowadził wymagane dostrojenie na podstawie informacji dostępnych w czasie kompilacji, potrzebnych do wykonania wymaganej konwersji typu.

Użycie dynamic_cast i identyfikacja typu w czasie działania

Rzutowanie dynamiczne, jak sama nazwa wskazuje, jest przeciwieństwem rzutowania statycznego i faktycznie zachodzi w czasie działania — tzn. w trakcie działania aplikacji. Wynik operacji `dynamic_cast` może być sprawdzony w celu weryfikacji, czy próba przeprowadzenia rzutowania zakończyła się powodzeniem. Typowa składnia użycia operatora `dynamic_cast` jest następująca:

```
typ_docelowy* pDest = dynamic_cast <typ_klasy*> (pSource);
if (pDest)     // Sprawdzenie, czy operacja rzutowania zakończyła się powodzeniem.
   pDest->CallFunc();
```

Przykładowo:

```
Base* pBase = new Derived();

// Przeprowadzenie rzutowania w dół.
Derived* pDerived = dynamic_cast <Derived*> (pBase);

if (pDerived)     // Sprawdzenie, czy operacja rzutowania zakończyła się powodzeniem.
   pDerived->CallDerivedClassFunction();
```

Jak pokazano w powyższym przykładzie, mając wskaźnik do obiektu klasy bazowej, programista może przy użyciu `dynamic_cast` sprawdzić typ obiektu docelowego, do którego prowadzi wskaźnik, przed faktycznym użyciem tego wskaźnika. Warto zwrócić uwagę, że w podanym fragmencie kodu obiekt docelowy *jest* typu `Derived`. Dlatego przykład służy jedynie do celów demonstracyjnych. Jednak nie zawsze tak musi być, np. kiedy wskaźnik typu `Derived*` jest przekazywany funkcji akceptującej typ `Base*`. Mając obiekt klasy bazowej, funkcja może użyć `dynamic_cast` w celu określenia typu, a następnie przeprowadzenia operacji odpowiedniej dla wykrytego typu. W takim przypadku `dynamic_cast` pomaga w ustaleniu typu podczas działania programu i pozwala na użycie rzutowanego wskaźnika, kiedy można to zrobić bezpiecznie. W kodzie przedstawionym w listingu 13.1 użyto hierarchii klas `Tuna` i `Carp` powiązanej z klasą bazową `Fish`, natomiast funkcja `DetectFishType()` dynamicznie wykrywa, czy wskaźnik `Fish*` jest typu `Tuna*`, czy `Carp*`.

Uwaga

> Ten mechanizm identyfikacji typu obiektu w trakcie działania aplikacji jest nazywany *Runtime Type Identification* (RTTI).

Listing 13.1. Rzutowanie dynamiczne pomaga w ustaleniu, czy obiekt Fish jest typu Tuna, czy Carp

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: public:
6:    virtual void Swim()
7:    {
8:       cout << "Ryba pływa w wodzie" << endl;
```

```
 9:    }
10:
11:    // Klasa bazowa zawsze powinna zawierać wirtualny destruktor.
12:    virtual ~Fish() {}
13: };
14:
15: class Tuna: public Fish
16: {
17: public:
18:    void Swim()
19:    {
20:       cout << "Tuńczyk pływa naprawdę szybko w morzu" << endl;
21:    }
22:
23:    void BecomeDinner()
24:    {
25:       cout << "Dzisiaj na kolację będzie sushi z tuńczyka" << endl;
26:    }
27: };
28:
29: class Carp: public Fish
30: {
31: public:
32:    void Swim()
33:    {
34:       cout << "Karp pływa naprawdę wolno w jeziorze" << endl;
35:    }
36:
37:    void Talk()
38:    {
39:       cout << "Karpie nie mają głosu" << endl;
40:    }
41: };
42:
43: void DetectFishType(Fish* InputFish)
44: {
45:    Tuna* pIsTuna = dynamic_cast <Tuna*>(InputFish);
46:    if (pIsTuna)
47:    {
48:       cout << "Wykryto tuńczyka. Tuńczyk będzie na kolację: " << endl;
49:       pIsTuna->BecomeDinner();    // Wywołanie metody Tuna::BecomeDinner.
50:    }
51:
52:    Carp* pIsCarp = dynamic_cast <Carp*>(InputFish);
53:    if(pIsCarp)
54:    {
55:       cout << "Wykryto karpia. Karp będzie na kolację: " << endl;
56:       pIsCarp->Talk();  // Wywołanie metody Carp::Talk.
57:    }
```

```
58:
59:    cout << "Sprawdzenie typu przy użyciu metody wirtualnej Fish::Swim: " <<
       ↪endl;
60:    InputFish->Swim(); // Wywołanie metody wirtualnej Swim.
61: }
62:
63: int main()
64: {
65:    Carp myLunch;
66:    Tuna myDinner;
67:
68:    DetectFishType(&myDinner);
69:    cout << endl;
70:    DetectFishType(&myLunch);
71:
72:    return 0;
73:}
```

Wynik ▼

```
Wykryto tuńczyka. Tuńczyk będzie na kolację:
Dzisiaj na kolację będzie sushi z tuńczyka
Sprawdzenie typu przy użyciu metody wirtualnej Fish::Swim:
Tuńczyk pływa naprawdę szybko w morzu

Wykryto karpia. Karp będzie na kolację:
Karpie nie mają głosu
Sprawdzenie typu przy użyciu metody wirtualnej Fish::Swim:
Karp pływa naprawdę wolno w jeziorze
```

Analiza ▼

W listingu użyto znanej z lekcji 10. hierarchii klas `Tuna` i `Carp` (ich klasa bazowa to `Fish`). W celu wyjaśnienia problemu obie wymienione klasy potomne nie tylko implementują metodę wirtualną `Swim()`, ale również zawierają metody charakterystyczne dla ich typów, czyli (odpowiednio) `Tuna::BecomeDinner()` i `Carp::Talk()`. Cechą specjalną omawianego przykładu jest fakt, że dany egzemplarz klasy bazowej `Fish*` ma możliwość dynamicznego określenia, czy podany wskaźnik prowadzi do obiektu klasy `Tuna`, czy `Carp`. Wspomniane dynamiczne wykrycie typu (czyli identyfikacji typu obiektu w trakcie działania aplikacji) odbywa się w funkcji `DetectFishType()` zdefiniowanej w wierszach od 43. do 61. W wierszu 45. operatora `dynamic_cast` użyto do sprawdzenia, czy wskaźnik klasy bazowej `Fish*` prowadzi do typu `Tuna*`. Jeżeli `Fish*` prowadzi do obiektu `Tuna`, operator zwraca prawidłowy adres, w przeciwnym razie

wartością zwrotną jest `null`. Wynik działania operatora `dynamic_cast` jest więc zawsze sprawdzany pod kątem poprawności. Gdy przeprowadzana w wierszu 46. operacja sprawdzania zakończy się powodzeniem, wiadomo, że wskaźnik `pIsTuna` prowadzi do prawidłowego obiektu `Tuna` i można wywołać metodę `Tuna::BecomeDinner()`, co przedstawiono w wierszu 49. W przypadku obiektu `Carp` wskaźnik zostaje użyty do wywołania metody `Carp::Talk()`, tak jak przedstawiono w wierszu 56. Przed zakończeniem działania metoda `DetectFishType()` przeprowadza sprawdzenie typu przy użyciu wywołania metody `Fish::Swim()` — to metoda wirtualna, która przekierowuje wszystkie wywołania do metod `Swim()` zaimplementowanych w klasach `Tuna` i `Carp`.

Ostrzeżenie

Wartość zwrotna operacji `dynamic_cast` zawsze powinna być sprawdzana pod kątem poprawności. Jeżeli rzutowanie zakończy się niepowodzeniem, wartością będzie `null`.

Użycie reinterpret_cast

Operator `reinterpret_cast` to operator rzutowania C++, który jest najbliższy rzutowaniu w stylu C. W rzeczywistości pozwala programiście na rzutowanie jednego typu obiektu do innego, niezależnie od tego, czy są one ze sobą powiązane. Oznacza to, że operator wymusza ponowną interpretację typu przy użyciu składni przedstawionej w poniższym przykładzie:

```
Base * pBase = new Base ();
CUnrelated * pUnrelated = reinterpret_cast<CUnrelated*>(pBase);
// Wprawdzie powyższy kod się kompiluje, ale nie jest dobrą praktyką programistyczną!
```

Powyższa operacja rzutowania faktycznie wymusza na kompilatorze zaakceptowanie sytuacji, na którą operator `static_cast` nigdy by nie pozwolił. Znajduje ona zastosowanie w określonych aplikacjach niskiego poziomu (np. w sterownikach), w których dane muszą być konwertowane na proste typy akceptowane przez API. Przykładowo niektóre API działają jedynie ze strumieniami `BYTE`, tzn. `unsigned char*`:

```
SomeClass* pObject = new SomeClass ();
// Konieczność wysłania obiektu jako strumienia bajtów.
unsigned char* pBytes = reinterpret_cast <unsigned char*>(pObject);
```

Rzutowanie zastosowane w powyższym fragmencie kodu nie spowodowało modyfikacji bitowej reprezentacji danych źródłowych, jedynie oszukało kompilator, pozwalając programiście na pobranie pojedynczych bajtów

znajdujących się w obiekcie typu `SomeClass`. Ponieważ żaden inny operator rzutowania C++ nie pozwala na tego rodzaju konwersję, `reinterpret_cast` wyraźnie ostrzega użytkownika o potencjalnym niebezpieczeństwie (i braku możliwości przeniesienia) przeprowadzanej konwersji.

Ostrzeżenie

> O ile to możliwe, w aplikacjach należy unikać stosowania `reinterpret_cast`, ponieważ operator ten pozwala kompilatorowi na traktowanie typu X jak niepowiązanego z nim typu Y. To naprawdę nie jest dobre rozwiązanie programistyczne.

Użycie const_cast

Operator `const_cast` pozwala na wyłączenie modyfikatora dostępu `const` w obiekcie. Jeżeli zastanawiasz się, czy taki rodzaj rzutowania w ogóle jest potrzebny, prawdopodobnie masz rację. W idealnej sytuacji programiści tworzą klasy w sposób prawidłowy i pamiętają o częstym używaniu słowa kluczowego `const` we właściwych miejscach. Niestety, praktyka daleko odbiega od ideału. Brakujące kwalifikatory `const`, jak przedstawiono w poniższym fragmencie kodu, są bardzo często spotykane:

```
class SomeClass
{
  public:
     // ...
     void DisplayMembers ();      // To powinien być element składowy wraz ze słowem
                                  // kluczowym const.
};
```

Dlatego też podczas tworzenia funkcji, takiej jak:

```
void DisplayAllData (const SomeClass& mData)
{
   mData.DisplayMembers ();  // Kompilacja nie powiedzie się.
   // Przyczyna porażki: wywołanie do elementu składowego innego niż const przy użyciu
   // odniesienia const.
}
```

masz rację, przekazując obiekt `mData` jako odniesienie typu `const`. Mimo wszystko, funkcja wyświetlająca dane powinna być tylko do odczytu i nie powinna mieć możliwości wywoływania elementów składowych innych niż typu `const`. Oznacza to, że nie powinna mieć możliwości wywołania funkcji, która modyfikuje stan obiektu. Jednak implementacja funkcji `DisplayMembers()`, która również powinna być typu `const`, niestety, taka nie jest. W takiej sytuacji,

dopóki klasa `SomeClass` należy do Ciebie i zachowujesz kontrolę nad kodem źródłowym, dopóty możesz wprowadzać odpowiednie poprawki w kodzie funkcji `DisplayMembers()`. Jednak w wielu przypadkach klasa może znajdować się w bibliotece firmy trzeciej i wprowadzenie tego rodzaju zmian nie będzie możliwe. W takich sytuacjach rozwiązaniem jest użycie `const_cast`.

Składnia wywołania funkcji `DisplayMemebers()` w takim przypadku jest następująca:

```
void DisplayAllData (const SomeClass& mData)
{
   SomeClass& refData = const_cast <SomeClass&>(mData);
   refData.DisplayMembers();     // Dozwolone!
}
```

Zwróć uwagę, że użycie `const_cast` w celu wywołania funkcji innej niż typu `const` powinno być ostatecznością. Ogólnie rzecz biorąc, pamiętaj, że używanie `const_cast` do modyfikacji obiektu `const` może prowadzić do wystąpienia nieprzewidzianego zachowania.

Warto także zwrócić uwagę na możliwość użycia operatora `const_cast` wraz ze wskaźnikami:

```
void DisplayAllData (const SomeClass* pData)
{
   // pData->DisplayMembers(); Błąd: próba wywołania funkcji innej niż typu const!
   SomeClass* pCastedData = const_cast <SomeClass*>(pData);
   pCastedData->DisplayMembers();     // Dozwolone!
}
```

Problemy z operatorami rzutowania C++

Nie każdy jest zadowolony ze wszystkich operatorów rzutowania C++, nawet zwolennicy C++. Powody tego mogą być różne, od niewygodnej i nieintuicyjnej składni, aż do uznania ich za zbędne.

Porównajmy po prostu przykładowy kod:

```
double dPi = 3.14159265;

// Rzutowanie w stylu C++: static_cast.
int Num = static_cast <int>(dPi);      // Wynik: Num wynosi 3.

// Rzutowanie w stylu C.
int Num2 = (int)dPi;                   // Wynik: Num wynosi 3.
```

```
// Pozostawienie kompilatorowi wyboru rzutowania.
int Num3 = dPi;              // Wynik: Num wynosi 3. Brak błędów!
```

We wszystkich trzech przypadkach programista osiągnął ten sam wynik. W praktyce, rozwiązanie drugie jest prawdopodobnie najbardziej rozpowszechnione, następne w kolejności jest trzecie. Niewielka grupa programistów stosuje pierwsze. W każdym przypadku kompilator jest na tyle inteligentny, aby prawidłowo skonwertować tego rodzaju typy. Składnia rzutowania sprawia więc wrażenie, że kod staje się trudniejszy w odczycie.

Inne przypadki użycia `static_cast` również są doskonale obsługiwane przez rzutowania w stylu C, które niezaprzeczalnie wyglądają na prostsze:

```
// Przy użyciu static_cast.
Derived* pDerived = static_cast <Derived*>(pBase);

// Jednak takie rozwiązanie również działa doskonale...
Derived* pDerivedSimple = (Derived*)pBase;
```

Dlatego też zalety używania `static_cast` są często przesłaniane przez niezręczną składnię tego operatora. Bjarne Stroustrup właściwie określił tę sytuację: „Ponieważ operator `static_cast` jest taki brzydki i względnie trudny do napisania, być może zastanowisz się dwa razy, zanim go użyjesz? To byłoby wskazane, ponieważ w nowoczesnym kodzie C++ powinno się unikać stosowania rzutowania". (Patrz dokument C++ Style and Technique FAQ napisany przez Stroustrupa i dostępny na stronie *http://www.stroustrup.com/bs_faq2.html*.)

Jeśli chodzi o inne operatory: `reinterpret_cast` wymusza zastosowanie Twojego rozwiązania, kiedy operator `static_cast` nie działa. Podobnie sprawa ma się z operatorem `const_cast`, gdy chodzi o modyfikację modyfikatora dostępu `const`. Dlatego też w nowoczesnych aplikacjach C++ unika się stosowania operatorów rzutowania innych niż `dynamic_cast`. Użycie innych operatorów rzutowania może być uzasadnione jedynie podczas spełniania wymagań przestarzałych aplikacji. W tego rodzaju sytuacjach preferowanie rzutowania w stylu C zamiast stylu C++ to najczęściej kwestia upodobań. Najważniejsze jest, aby unikać rzutowania, gdy tylko jest to możliwe. Kiedy trzeba zastosować rzutowanie, należy wiedzieć, co dzieje się w tle tego procesu.

TAK	NIE
Pamiętaj, że rzutowanie Derived* na Base* nosi nazwę rzutowania w górę i jest bezpieczne. Pamiętaj, że rzutowanie Base* na Derived* nosi nazwę rzutowania w dół i może być niebezpieczne, o ile nie zostanie użyty operator dynamic_cast. Pamiętaj, że celem tworzenia hierarchii dziedziczenia najczęściej jest przygotowanie funkcji wirtualnych. Wspomniane funkcje po wywołaniu używają wskaźników klasy bazowej gwarantujących wywołanie wersji funkcji dostępnych w klasach potomnych.	Nie zapominaj o sprawdzeniu poprawności wskaźnika po użyciu operatora dynamic_cast. Nie projektuj aplikacji w oparciu o mechanizm RTTI przy użyciu operatora dynamic_cast.

Podsumowanie

W tej lekcji poznałeś różne operatory rzutowania C++, a także argumenty za ich stosowaniem oraz przeciw niemu. Dowiedziałeś się także, że najlepiej unikać przeprowadzania rzutowania.

Pytania i odpowiedzi

Pytanie: Czy dopuszczalne jest modyfikowanie zawartości obiektu typu `const` poprzez rzutowanie wskaźnika/odniesienia do niego za pomocą operatora `const_cast`?

Odpowiedź: Zdecydowanie nie. Wynik tego rodzaju operacji jest niezdefiniowany i na pewno niepożądany.

Pytanie: Potrzebuję `Bird*`, ale pod ręką mam `Dog*`. Kompilator nie pozwala mi na użycie wskaźnika do obiektu `Dog` jako `Bird*`. Jednak kiedy użyję operatora `reinterpret_cast` w celu rzutowania `Dog*` do `Bird*`, kompilator nie zgłasza błędu. Wydaje się więc, że mogę użyć tego wskaźnika do wywołania funkcji składowej `Fly()` klasy `Bird`. Czy takie rozwiązanie jest dopuszczalne?

Odpowiedź: Ponownie, zdecydowanie nie. Operator `reinterpret_cast` zmienia jedynie sposób interpretacji wskaźnika, nie powoduje zmiany obiektu, do którego ten wskaźnik prowadzi (to nadal będzie `Dog`).

Wywołanie funkcji `Fly()` na obiekcie `Dog` nie przyniesie oczekiwanych wyników, a może doprowadzić do awarii aplikacji.

Pytanie: Mam obiekt `Derived` wskazywany przez `pBase`, który jest `Base*`. Jestem przekonany, że pBase wskazuje obiekt `Derived`, więc czy naprawdę muszę używać operatora `dynamic_cast`?

Odpowiedź: Ponieważ jesteś przekonany, że wskazywany obiekt to typ `Derived`, możesz poprawić wydajność programu w trakcie jego działania poprzez użycie operatora `static_cast`.

Pytanie: Język C++ dostarcza operatory rzutowania, ale doradza się, aby unikać ich stosowania, gdy to tylko możliwe. Dlaczego tak się dzieje?

Odpowiedź: Masz w domu aspirynę, ale przecież nie oznacza to, że stanowi ona podstawowy składnik Twojej diety, nieprawdaż? Używaj rzutowania, gdy będzie to konieczne.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Masz wskaźnik `pBase` prowadzący do klasy bazowej. Jakie zastosujesz rzutowanie w celu sprawdzenia, czy jest to typ `Derived1`, czy `Derived2`?
2. Masz referencję `const` do obiektu i próbujesz wywołać napisaną dla Ciebie publiczną funkcję składową. Kompilator nie pozwala na jej wywołanie, ponieważ funkcja nie jest typu `const`. Czy poprawisz tę funkcję, czy użyjesz operatora `const_cast`?
3. Operator `reinterpret_cast` powinien być używany tylko wtedy, gdy nie działa operator `static_const` oraz wiadomo, że rzutowanie jest konieczne i pozostaje bezpieczne. To prawda czy fałsz?

4. Czy to prawda, że wiele egzemplarzy konwersji bazujących na operatorze `static_cast`, zwłaszcza między prostymi typami danych, będzie automatycznie przeprowadzanych przez dobry kompilator C++?

Ćwiczenia

1. **Łowcy błędów:** Co jest nie tak z poniższym fragmentem kodu?

```
void DoSomething(Base* pBase)
{
   Derived* pDerived = dynamic_cast <Derived*>(pBase);
   pDerived->DerivedClassMethod();
}
```

2. Masz wskaźnik `pFish*` prowadzący do obiektu klasy `Tuna`.

```
Fish* pFish = new Tuna;
Tuna* pTuna = <what cast?>pFish;
```

   Którego rodzaju rzutowania użyjesz, aby wskaźnik `Tuna*` prowadził do obiektu typu `Tuna`? Zaprezentuj odpowiedni fragment kodu.

Lekcja 14

Wprowadzenie do makr i wzorców

Powinieneś mieć już opanowaną podstawową składnię C++. Programy napisane w tym języku powinny być dla Ciebie zrozumiałe. Najwyższy czas poznać funkcje języka, które będą pomocne w efektywnym tworzeniu aplikacji.

Z tej lekcji dowiesz się:

- **zapoznasz się z wprowadzeniem do preprocesora,**
- **poznasz słowo kluczowe #define i makra,**
- **zapoznasz się z wprowadzeniem do wzorców,**
- **dowiesz się, jak tworzyć wzorce funkcji i klas,**
- **poznasz różnice między makrami i wzorcami,**
- **dowiesz się, jak standard C++11 pomaga w użyciu `static_assert` podczas kompilacji.**

Preprocesor i kompilator

W lekcji 2., zatytułowanej „Anatomia programu C++", po raz pierwszy usłyszałeś o preprocesorze. Jak sama nazwa wskazuje, preprocesor jest uruchamiany przed rozpoczęciem kompilacji. Innymi słowy, na podstawie otrzymanych instrukcji preprocesor decyduje o tym, co będzie skompilowane. Wszystkie dyrektywy preprocesora rozpoczynają się od znaku #, np.:

```
// Nakazanie preprocesorowi wstawienia w tym miejscu zawartości pliku nagłówkowego iostream.
#include <iostream>

// Zdefiniowanie makra w postaci stałej.
#define ARRAY_LENGTH 25
int MyNumbers[ARRAY_LENGTH]; // Tablica zawierająca 25 liczb całkowitych.

// Zdefiniowanie makra w postaci funkcji.
#define SQUARE(x) ((x) * (x))
int TwentyFive = SQUARE(5);
```

W tej lekcji skoncentrujemy się na dwóch typach dyrektyw preprocesora przedstawionych w powyższym fragmencie kodu. W pierwszym typie użyto `#define` do zdefiniowana stałej, natomiast w drugim zastosowano `#define` do zdefiniowania makra w postaci funkcji. Niezależnie od odgrywanej roli, oba typy funkcji w rzeczywistości nakazują preprocesorowi zastąpienie każdego wystąpienia makra (`ARRAY_LENGTH` lub `SQUARE`) zdefiniowaną w nim wartością.

Uwaga

> Makra to nic innego jak zastępowanie tekstu. Preprocesor nie robi nic szczególnego, po prostu zastępuje wskazany identyfikator odpowiednim tekstem.

Użycie dyrektywy #define do definiowania stałych

Składnia użycia `#define` w celu zdefiniowania stałej jest całkiem prosta:

```
#define identyfikator wartość
```

Przykładowo stała `ARRAY_LENGTH` zastępowana przez wartość 25 będzie miała następującą postać:

```
#define ARRAY_LENGTH 25
```

Powyższy identyfikator będzie zastępowany wartością 25, za każdym razem gdy preprocesor napotka tekst `ARRAY_LENGTH`:

```
int MyNumbers [ARRAY_LENGTH] = {0};
double Radiuses [ARRAY_LENGTH] = {0.0};
std::string Names [ARRAY_LENGTH];
```

Po zakończeniu działania preprocesora powyższe polecenia będą miały następującą postać:

```
int MyNumbers [25] = {0};   // Tablica 25 liczb całkowitych.
double Radiuses [25] = {0.0};   // Tablica 25 liczb podwójnej precyzji.
std::string Names [25];   // Tablica 25 obiektów std::strings.
```

Operacja zastąpienia będzie przeprowadzona w każdej sekcji kodu, nawet w pętlach `for`, np. w takiej jak poniższa:

```
for(int Index = 0; Index < ARRAY_LENGTH; ++Index)
   MyNumbers[Index] = Index;
```

Ta pętla `for` jest widziana przez kompilator w następującej postaci:

```
for(int Index = 0; Index < 25; ++Index)
   MyNumbers[Index] = Index;
```

Aby dokładnie zobaczyć, jak działa makro, przeanalizuj listing 14.1.

Listing 14.1. Deklaracja i użycie makr definiujących stałe

```
 0: #include <iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: #define ARRAY_LENGTH 25
 5: #define PI 3.1416
 6: #define MY_DOUBLE double
 7: #define FAV_WHISKY "Jack Daniels"
 8:
 9: int main()
10: {
11:    int MyNumbers [ARRAY_LENGTH] = {0};
12:    cout << "Wielkość tablicy: " << sizeof(MyNumbers) / sizeof(int) << endl;
13:
14:    cout << "Podaj promień: ";
15:    MY_DOUBLE Radius = 0;
16:    cin >> Radius;
17:    cout << "Pole wynosi: " << PI * Radius * Radius << endl;
18:
19:    string FavoriteWhisky (FAV_WHISKY);
20:    cout << "Mój ulubiony drink to: " << FAV_WHISKY << endl;
21:
22:    return 0;
23: }
```

Wynik ▼

```
Wielkość tablicy: 25
Podaj promień: 2.1569
Pole wynosi: 14.7154
Mój ulubiony drink to: Jack Daniels
```

Analiza ▼

Makra ARRAY_LENGTH, PI, MY_DOUBLE i FAV_WHISKY to cztery makra w postaci stałych zdefiniowane w wierszach od 3. do 7. Jak możesz zobaczyć, pierwsze makro zostało użyte w definicji wielkości tablicy (patrz wiersz 11.) potwierdzonej pośrednio przez użycie operatora `sizeof()` w wierszu 12. Makra MY_DOUBLE użyto w deklaracji zmiennej `Radius` typu `double` (patrz wiersz 15.), podczas gdy PI użyto do obliczenia pola okręgu w wierszu 17. Wreszcie, makro FAV_WHISKY zostało użyte do inicjalizacji obiektu `std::string` (patrz wiersz 19.) i bezpośrednio w poleceniu `cout` w wierszu 20. Wszystkie wymienione makra pokazują, jak preprocesor po prostu przeprowadza operację zastępowania tekstu.

Tego rodzaju „zwykłe" zastępowanie teksu wszechobecne w listingu 14.1 ma jednak pewne wady.

Wskazówka

Kiedy preprocesor przeprowadza zwykłe zastępowanie tekstu, możesz tego rodzaju operację wymusić, choć kompilator nie zawsze ją wykona. Makro FAV_WHISKY w wierszu 7. listingu 14.1 można zdefiniować następująco:

```
#define FAV_WHISKY 42 // "Jack Daniels".
```

co skutkuje wygenerowaniem błędu kompilacji w wierszu 19. dla egzemplarza `std::string`. Jednak w przypadku tego egzemplarza kompilacja zakończy się powodzeniem, natomiast w trakcie działania programu wyświetlony będzie następujący komunikat:

```
Mój ulubiony drink to: 42
```

To — oczywiście — nie ma sensu, a, co najgorsze, błąd pozostaje niezauważony. Ponadto nie masz zbyt dużej kontroli nad sposobem definicji stałej PI: jej typem jest `double` czy `float`? Odpowiedź brzmi: żaden z dwóch wymienionych typów. PI to element, który przez preprocesor zostanie zastąpiony elementem tekstowym `3.1416`. Ten element nigdy nie został zdefiniowany jako typ danych.

Stałe lepiej definiować przy użyciu słowa kluczowego `const` wraz z typem danych. Dlatego też zaprezentowane poniżej rozwiązanie jest znacznie lepsze od przedstawionego powyżej:

```
const int ARRAY_LENGTH = 25;
const double PI = 3.1416;
const char* FAV_WHISKY = "Jack Daniels";
typedef double MY_DOUBLE;  // Użycie typedef jako aliasu typu.
```

Użycie makr do ochrony przed wielokrotnym dołączaniem

Programiści C++ najczęściej deklarują klasy i funkcje w plikach z rozszerzeniem *.h* nazywanych plikami nagłówkowymi. Z kolei definicje funkcji znajdują się w plikach *.cpp*, w których zawartość plików nagłówkowych jest wstawiana przy użyciu dyrektywy preprocesora `#include <plik nagłówkowy>`. Jeżeli plik nagłówkowy — nazwijmy go *class1.h* — zawiera deklarację klasy wraz z deklaracją innej klasy (*class2.h*), wtedy w pliku *class1.h* konieczne jest dołączenie pliku *class2.h*. Jeżeli projekt będzie skomplikowany i inna klasa będzie wymagała *class1.h*, w pliku *class2.h* znajdzie się polecenie wstawiające zawartość pliku *class1.h*.

Dla preprocesora dwa wymienione pliki nagłówkowe nawzajem wstawiające swoją zawartość to problem natury rekurencyjnej. Aby uniknąć tego rodzaju problemu, można użyć makr w połączeniu z dyrektywami preprocesora `#ifndef` i `#endif`.

Plik nagłówkowy *header1.h* wstawiający zawartość pliku *header2.h* przedstawia się następująco:

```
#ifndef HEADER1_H_  // Ochrona przed wielokrotnym dołączaniem:
#define HEADER1_H_  // ten i poniższe wiersze preprocesor odczyta jednokrotnie.
#include <header2.h>
class Class1
{
   // Elementy składowe klasy.
};
#endif   // Koniec pliku header1.h.
```

Z kolei plik *header2.h* jest podobny, ale zastosowano w nim inną definicję makra wstawiającego zawartość pliku *header1.h*:

```
<header1.h>:
#ifndef HEADER2_H_  // Ochrona przed wielokrotnym dołączaniem.
```

```
#define HEADER2_H_
#include <header1.h>
class Class2
{
    // Elementy składowe klasy.
};
#endif  // Koniec pliku header2.h.
```

Uwaga

> Dyrektywę `#ifndef` odczytuje się jako „jeśli nie zostało zdefiniowane”. To jest polecenie przetwarzania warunkowego, które nakazuje preprocesorowi działanie tylko wtedy, gdy identyfikator nie został zdefiniowany.
>
> Dyrektywa `#endif` oznacza dla preprocesora koniec polecenia przetwarzania warunkowego.

Dlatego też gdy na początku preprocesor zacznie przetwarzać plik *header1.h* i napotka dyrektywę `#ifndef`, dostrzeże, że makro `HEADER1_H_` nie zostało zdefiniowane i będzie kontynuować działanie. Pierwszy wiersz po dyrektywie `#ifndef` deklaruje makro `HEADER1_H_`, gwarantując, że drugie przetwarzanie tego pliki zakończy się już w wierszu pierwszym, ponieważ warunek `#ifndef` przyjmie wartość `false`. To samo dotyczy pliku nagłówkowego *header2.h*. Ten prosty mechanizm to bez wątpienia jedna z najczęściej stosowanych w świecie programowania C++ funkcjonalność oparta na makrach.

Użycie dyrektywy #define do definiowania funkcji

Możliwości preprocesora w zakresie prostego zastępowania elementów tekstowych identyfikowanych przez makro bardzo często skutkują tworzeniem prostych funkcji, np.:

```
#define SQUARE(x) ((x) * (x))
```

Przedstawiona powyżej funkcja służy do obliczenia kwadratu liczby. Podobnie, makro przeznaczone do obliczania pola okręgu może mieć następującą postać:

```
#define PI 3.1416
#define AREA_CIRCLE(r) (PI*(r)*(r))
```

Funkcje makro są bardzo często używane do tego rodzaju prostych obliczeń. W porównaniu ze zwykłymi wywołaniami funkcji, ich zaletą jest rozwinięcie funkcji w miejscu jej zdefiniowania przed kompilacją, co pozwala na zwiększenie

wydajności działania kodu w pewnych sytuacjach. Użycie funkcji makro przedstawiono w listingu 14.2.

Listing 14.2. Użycie funkcji makro obliczającej kwadrat liczby, pole okręgu oraz wskazującej większą i mniejszą liczbę spośród dwóch podanych

```
 0: #include <iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: #define SQUARE(x) ((x) * (x))
 5: #define PI 3.1416
 6: #define AREA_CIRCLE(r) (PI*(r)*(r))
 7: #define MAX(a, b) (((a) > (b)) ? (a) : (b))
 8: #define MIN(a, b) (((a) < (b)) ? (a) : (b))
 9:
10: int main()
11: {
12:    cout << "Podaj liczbę całkowitą: ";
13:    int Input1 = 0;
14:    cin >> Input1;
15:
16:    cout << "KWADRAT(" << Input1 << ") = " << SQUARE(Input1) << endl;
17:    cout << "Pole okręgu o promieniu " << Input1 << " wynosi: ";
18:    cout << AREA_CIRCLE(Input1) << endl;
19:
20:    cout << "Podaj inną liczbę całkowitą: ";
21:    int Input2 = 0;
22:    cin >> Input2;
23:
24:    cout << "MIN(" << Input1 << "" << Input2 << " = ";
25:    cout << MIN (Input1, Input2) << endl;
26:
27:    cout <<"MAX(" << Input1 << ", " << Input2 << ") = ";
28:    cout << MAX (Input1, Input2) << endl;
29:
30:    return 0;
31: }
```

Wynik ▼

```
Podaj liczbę całkowitą: 36
KWADRAT(36) = 1296
Pole okręgu o promieniu 36 wynosi: 4071.51
Podaj inną liczbę całkowitą: -101
MIN(36, -101) = -101
MAX(36, -101) = 36
```

Analiza ▼

W wierszach od 4. do 8. znajduje się kilka użytecznych funkcji makro, które zwracają kwadrat danej liczby, pole okręgu oraz mniejszą i większą liczbę spośród dwóch podanych. Zwróć uwagę, jak makro `AREA_CIRCLE` w wierszu 6. oblicza pole przy użyciu stałej `PI`, co oznacza, że jedno makro może używać innego makra. W końcu to przecież wydawane preprocesorowi zwykłe polecenia zastępowania tekstu. Przeanalizujmy wiersz 25. używający makra `MIN`:

```
cout << MIN (Input1, Input2) << endl;
```

Powyższy wiersz jest dostarczany kompilatorowi w następującym formacie (makro zostaje rozwinięte):

```
cout << (((Input1) < (Input2)) ? (Input1) : (Input2)) << endl;
```

Ostrzeżenie

> Pamiętaj, że wielkość liter w makrze nie ma znaczenia, a funkcje makro mogą być niebezpieczne. Przykładowo `AREA_CIRCLE` w idealnej sytuacji powinno być funkcją zwracającą wartość `double`, co zagwarantuje prawidłowy wynik obliczonego pola i zapewni funkcji niezależność od natury danych wejściowych (promienia).

Po co te wszystkie nawiasy?

Spójrz ponownie na makro obliczające pole okręgu:

```
#define AREA_CIRCLE(r) (PI*(r)*(r))
```

Powyższy wiersz ma ciekawą składnię ze względu na liczbę użytych nawiasów. Dla porównania spójrz na kod funkcji `Area()` przedstawiony w listingu 7.1 w lekcji 7., zatytułowanej „Funkcje”.

```
// Definicje funkcji (implementacje).
double Area(double InputRadius)
{
   return Pi * InputRadius * InputRadius;  // Zauważ, że polecenie nie zawiera nawiasów.
}
```

Dlaczego więc dla tego samego wzoru w makrze zastosowano nawiasy, natomiast w definicji funkcji już nie? Powód kryje się w sposobie przetwarzania makra jako obsługiwanego przez preprocesor mechanizmu zastępowania tekstu.

Przeanalizujmy wersję makra bez większości nawiasów:

```
#define AREA_CIRCLE(r) (PI*r*r)
```

Co się stanie, gdy powyższe makro zostanie wywołane w następującym poleceniu?

```
cout << AREA_CIRCLE (4+6);
```

Kompilator rozwinie polecenie do postaci:

```
cout << (PI*4+6*4+6);  // To nie odpowiada wyrażeniu PI*10*10.
```

Kierując się regułami określającymi pierwszeństwo operatorów, skoro mnożenie zachodzi przed dodawaniem, kompilator obliczy pole w następujący sposób:

```
cout << (PI*4+24+6);    // 42.5664 (wartość nieprawidłowa).
```

Z powodu braku nawiasów konwersja zwykłego tekstu dokonała zniszczeń w logice programu. Tego rodzaju problemu można uniknąć dzięki zastosowaniu nawiasów:

```
#define AREA_CIRCLE(r) (PI*(r)*(r))
cout << AREA_CIRCLE (4+6);
```

Po przeprowadzeniu operacji zastępowania tekstu kompilator otrzyma następujące polecenie:

```
cout << (PI*(4+6)*(4+6));  // PI*10*10, zgodnie z oczekiwaniami.
```

Nawiasy powodują prawidłowe obliczenie pola okręgu, a sam kod makra staje się niezależny od pierwszeństwa operatorów i wszelkich związanych z tym implikacji.

Użycie makra assert do sprawdzania wyrażeń

Wprawdzie dobrym rozwiązaniem jest sprawdzanie każdej ścieżki wykonywania kodu natychmiast po jego utworzeniu, ale to fizycznie niemożliwe. Można jednak wstawić kod sprawdzający wyrażenie i wartości zmiennych.

Do tego celu służy makro `assert`. Aby z niego skorzystać, należy dołączyć plik nagłówkowy *assert.h*, a następnie użyć poniższej składni:

```
assert (wyrażenie przyjmujące wartość true lub false);
```

Przykładowy fragment kodu korzystający z makra `assert()` do sprawdzenia zawartości wskaźnika przedstawia się następująco:

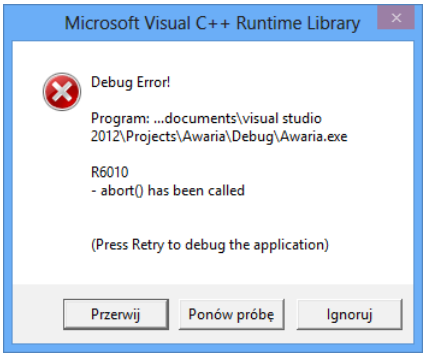
```
#include <assert.h>
int main()
{
```

```
    char* sayHello = new char [25];
    assert(sayHello != NULL); // Wyświetlenie komunikatu, jeśli wskaźnik ma wartość null.

    // Inny kod.

    delete [] sayHello;
    return 0;
}
```

Makro `assert()` gwarantuje wyświetlenie odpowiedniego komunikatu, jeśli wskaźnik jest nieprawidłowy. W celach demonstracyjnych zainicjalizowano `sayHello` z wartością `null`; po uruchomieniu programu w trybie debugowania w Visual Studio na ekranie natychmiast zostało wyświetlone okno dialogowe pokazane na rysunku 14.1.



RYSUNEK 14.1. Komunikat wyświetlony, gdy sprawdzony wskaźnik okazał się nieprawidłowy

Implementacja makra `assert()` w Microsoft Visual Studio pozwala na kliknięcie przycisku *Ponów próbę* i powrót do aplikacji, gdzie stos wywołań pokazuje wiersz, w którym test asercji zakończył się niepowodzeniem. Z tego powodu makro `assert()` to użyteczna funkcja podczas usuwania błędów, pomaga np. w sprawdzeniu parametrów danych wejściowych funkcji. Stosowanie tego makra jest zalecane, ponieważ w dłuższej perspektywie pomaga ono w poprawieniu jakości tworzonego kodu.

Uwaga

W większości środowisk programistycznych makro `assert()` zwykle pozostaje wyłączone w trybie Release, a komunikaty błędów lub inne informacje wyświetla jedynie po uruchomieniu programu w trybie debugowania.

Ponadto w pewnych środowiskach funkcjonalność wymienionego makra została zaimplementowana w postaci funkcji, a nie makra.

Ostrzeżenie

Asercja nie ma wpływu na kod aplikacji skompilowanej w trybie Release. Dlatego też trzeba się upewnić, że operacje sprawdzania o znaczeniu krytycznym dla aplikacji (np. wartość zwrotna operacji `dynamic_cast`) są nadal przeprowadzane przy użyciu konstrukcji `if`. Asercja pomaga w wykryciu problemu, ale nie zastępuje odpowiedniego kodu, np. sprawdzającego wskaźnik.

Wady i zalety użycia funkcji makro

Makro pozwala na ponowne wykorzystanie niektórych funkcji, niezależnie od rodzaju używanych zmiennych. Raz jeszcze spójrz na poniższy wiersz pochodzący z listingu 14.2:

```
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
```

Tej funkcji makro można użyć na liczbach całkowitych:

```
cout << MIN(25, 101) << endl;
```

To samo makro można również wykorzystać na liczbach typu `double`:

```
cout << MIN(0.1, 0.2) << endl;
```

Jeżeli `MIN()` to byłaby zwykła funkcja, wtedy konieczne byłoby przygotowanie jej dwóch wariantów: `MIN_INT()` (akceptuje parametry typu `int` i zwraca wartość `int`) oraz `MIN_DOUBLE()` (działa tak samo, ale na liczbach typu `double`). Zastosowana optymalizacja w postaci makra powoduje zmniejszenie ilości wierszy kodu programu i stanowi dla niektórych programistów zachętę do tworzenia makr definiujących proste funkcje narzędziowe. Funkcje te są rozwijane w miejscu ich zdefiniowana i przed kompilacją, a tym samym makro oferuje lepszą wydajność działania w porównaniu do zwykłego wywołania funkcji wykonującej takie samo zadanie. Wywołanie funkcji wymaga utworzenia stosu wywołania, przekazania argumentów funkcji itd. — obciążenie związane z tymi zadaniami często zabiera więcej czasu procesora niż same obliczenia w funkcji.

Pomimo wielu zalet, makra mają też wady, np. brak zapewnienia jakiegokolwiek bezpieczeństwa typu. Jakby było mało, debugowanie skompilowanego makra nie jest prostym zadaniem.

Jeżeli w programie potrzebujesz ogólnych funkcji niezależnych od typu i koniecznie zapewniających bezpieczeństwo typu, zamiast funkcji makro powinieneś przygotować funkcję wzorca. Gdy chcesz zwiększyć wydajność, rozważ użycie funkcji inline.

Tworzenie funkcji inline przy użyciu słowa kluczowego `inline` przedstawiono w listingu 7.10 w lekcji 7., zatytułowanej „Funkcje".

TAK	NIE
Jak najrzadziej staraj się tworzyć własne funkcje makro.	Nie zapominaj o umieszczeniu w nawiasie każdej zmiennej używanej w definicji funkcji makro.
Kiedy to tylko możliwe, zamiast makr stosuj zmienne typu const.	Nie zapominaj o stosowaniu w plikach nagłówkowych mechanizmu ochrony przed wielokrotnym wstawianiem w postaci dyrektyw #ifndef, #define i #endif.
Pamiętaj, że makra nie zapewniają bezpieczeństwa typów, a preprocesor nie przeprowadza sprawdzenia typu.	Nie zapominaj o umieszczeniu w kodzie poleceń assert() — nie mają one znaczenia w programie skompilowanym w trybie Release, ale pomagają w poprawieniu jakości tworzonego kodu.

Teraz możemy przejść do ogólnych praktyk programistycznych z użyciem wzorców.

Wprowadzenie do wzorców

Wzorce to prawdopodobnie jedna z najbardziej użytecznych funkcji języka C++, która jednocześnie pozostaje najmniej doceniana i rozumiana. Zanim przejdziemy do wzorców, zapoznajmy się z definicją wzorca (ang. *template*) znajdującą się w słowniku Webstera:

> Wymowa: \`tem-pl t\
>
> Funkcja: rzeczownik
>
> Etymologia: prawdopodobnie z języka francuskiego, zdrobnienie słowa temple (świątynia) prawdopodobnie pochodzi z łacińskiego słowa templum (świątynia)
>
> Data: 1677
>
> 1: krótki blok umieszczony poziomo w ścianie pod belką w celu rozłożenia jej wagi bądź ciśnienia (jak nad drzwiami)
>
> 2a: (1) miara, wzór lub forma (jak cienka płyta bądź deska) używana jako pomoc w formowaniu tworzonego przedmiotu; (2) cząsteczka (jak w przypadku DNA) służąca jako wzór do tworzenia innej cząsteczki (jak matrycowy kwas rybonukleinowy)
>
> 2b: przykrycie
>
> 3: coś nawiązującego lub służącego w charakterze wzoru

Ostatnia definicja prawdopodobnie jest najbliższa interpretacji słowa wzorzec, która jest używana w żargonie C++. Wzorce w C++ pozwalają na zdefiniowanie zachowania, które następnie będzie stosowane względem obiektów różnego typu. Niepokojąco blisko przypomina to możliwości oferowane przez makra (przypomnij sobie proste makro `MAX`, które z dwóch liczb określało większą), poza faktem, że makra nie zapewniają bezpieczeństwa typów, a wzorce tak.

Składnia deklaracji wzorca

Deklaracja wzorca rozpoczyna się od słowa kluczowego `template`, po którym znajduje się lista parametrów. Format takiej deklaracji jest następujący:

```
template <lista parametrów>
template funkcja | deklaracja klasy...
```

Słowo kluczowe `template` wskazuje początek deklaracji wzorca i poprzedza listę parametrów tego wzorca. Lista parametrów zawiera słowo kluczowe `typename`, które definiuje parametr wzorca `objectType`. W ten sposób następuje zarezerwowanie miejsca dla typu obiektu, dla którego będzie inicjalizowany wzorzec. Deklaracja wzorca zawiera wzór, który ma być zaimplementowany.

```
template <typename T1, typename T2 = T1>
bool TemplateFunction(const T1& param1, const T2& param2);

// Klasa wzorca.
template <typename T1, typename T2 = T1>
class Template
{
private:
   T1 m_Obj1;
   T2 m_Obj2;

public:
   T1 GetObj1() {return m_Obj1; }
   // Inne elementy składowe.
};
```

Powyżej przedstawiono wzorzec funkcji i wzorzec klasy, które pobierają dwa parametry wzorca `T1` i `T2`, gdzie typ domyślny `T2` jest taki sam jak `T1`.

Różne rodzaje deklaracji wzorca

Deklaracja wzorca może być:

- deklaracją lub definicją funkcji;
- deklaracją lub definicją klasy;
- definicją funkcji składowej lub elementu składowego klasy wzorca;
- definicją statycznych danych elementu składowego klasy wzorca;
- definicją statycznych danych elementu składowego klasy zagnieżdżonej wewnątrz klasy wzorca;
- definicją elementu składowego klasy wzorca.

Funkcje wzorca

Wyobraź sobie funkcję adaptującą się w celu dopasowania do zestawu parametrów różnych typów. Utworzenie tego rodzaju funkcji jest możliwe przy użyciu składni wzorca. Przeanalizujmy przykładową deklarację wzorca będącego odpowiednikiem wcześniej omówionego makra `MAX`, które zwraca większy parametr z dwóch dostarczonych:

```
template <typename objectType>
const objectType& GetMax (const objectType& value1, const objectType&
value2)
{
   if (value1 > value2)
      return value1;
   else
      return value2;
};
```

Przykładowy sposób użycia wzorca:

```
int Integer1 = 25;
int Integer2 = 40;
int MaxValue = GetMax <int> (Integer1, Integer2);
double Double1 = 1.1;
double Double2 = 1.001;
double MaxValue = GetMax <double> (Double1, Double2);
```

Zwróć uwagę na sposób użycia `<int>` w wywołaniu funkcji `GetMax()`. Skutkiem jest zdefiniowanie wzorca parametru `objectType` jako `int`. Powyższy kod prowadzi do tego, że kompilator wygeneruje dwie wersje funkcji wzorca `GetMax()`, które mogą być przedstawione następująco:

```
const int& GetMax (const int& value1, const int& value2)
{
   // ...
}
const double& GetMax (const double& value1, const double& value2)
{
   // ...
}
```

Jednak w rzeczywistości funkcje wzorca niekoniecznie muszą zawierać towarzyszący im specyfikator typu. Dlatego też przedstawione niżej wywołanie funkcji również będzie działało doskonale:

```
int MaxValue = GetMax (Integer1, Integer2);
```

W takich przypadkach kompilatory są na tyle inteligentne, aby zrozumieć, że funkcja wzorca jest wywoływana dla typu w postaci liczby całkowitej. Jednak w przypadku klasy wzorca konieczne jest wyraźne określenie typu, co przedstawiono w listingu 14.3.

Listing 14.3. Funkcja wzorca GetMax pomagająca w określeniu większej wartości spośród dwóch podanych

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: template <typename Type>
 5: const Type& GetMax (const Type& value1, const Type& value2)
 6: {
 7:     if (value1 > value2)
 8:         return value1;
 9:     else
10:         return value2;
11: }
12:
13: template <typename Type>
14: void DisplayComparison(const Type& value1, const Type& value2)
15: {
16:    cout << "GetMax(" << value1 << ", " << value2 << ") = ";
17:    cout << GetMax(value1, value2) << endl;
18: }
19:
20: int main()
21: {
22:    int Int1 = -101, Int2 = 2011;
23:    DisplayComparison(Int1, Int2);
24:
```

```
25:    double d1 = 3.14, d2 = 3.1416;
26:    DisplayComparison(d1, d2);
27:
28:    string Name1("Jack"), Name2("John");
29:    DisplayComparison(Name1, Name2);
30:
31:    return 0;
32: }
```

Wynik ▼

```
GetMax(-101, 2011) = 2011
GetMax(3.14, 3.1416) = 3.1416
GetMax(Jack, John) = John
```

Analiza ▼

W listingu znalazły się dwie funkcje wzorców: `GetMax()` zdefiniowana w wierszach od 4. do 11., która jest używana przez drugą funkcję wzorca `DisplayComparison()` zdefiniowaną w wierszach od 13. do 18. W funkcji `main()`, a dokładnie w wierszach 23., 26. i 29. pokazano, jak tej samej funkcji wzorca można wielokrotnie ponownie użyć z różnymi typami danych: `int`, `double` i `std::string`. Funkcje wzorca nie tylko mogą być wielokrotnie ponownie używane (podobnie jak ich odpowiedniki makr), ale są łatwiejsze do utworzenia i obsługi, a także zapewniają bezpieczeństwo typów.

Zwróć uwagę na możliwość wywołania funkcji `DisplayComparison()` wraz z wyraźnie podanym typem:

```
23:   DisplayComparison<int>(Int1, Int2);
```

Jednak podczas wywoływania funkcji wzorców jest to niepotrzebne. Nie ma konieczności podawania typu parametru wzorca, ponieważ kompilator potrafi go określić automatycznie. Z kolei w trakcie tworzenia klas wzorca trzeba podawać typ.

Wzorce i bezpieczeństwo typów

Przedstawione w listingu 14.3 funkcje wzorca `DisplayComparison()` i `GetMax()` są bezpieczne pod względem typu, a pozbawione sensu wywołania, np. tak:

```
DisplayComparison(Integer, "Dowolny ciąg tekstowy");
```

natychmiast spowodują powstanie błędu podczas kompilacji.

Klasy wzorca

W lekcji 10., „Klasy i obiekty”, dowiedziałeś się, że klasy to jednostki programistyczne hermetyzujące pewne atrybuty i metody działające z tymi atrybutami. Atrybuty najczęściej są prywatnymi elementami składowymi, takimi jak `int Age` w klasie `Human`. Klasy są projektowane jako wzorce, a rzeczywistym odzwierciedleniem klasy jest obiekt danej klasy. Dlatego też np. `Tomek` to obiekt klasy `Human` wraz z atrybutem `Age` przechowującym wartość 15. Co zrobić w sytuacji, jeśli chcesz, aby atrybut `Age` był typu `long long` dla pewnych osób żyjących długo, a `short` dla osób żyjących wyjątkowo krótko? W takim przypadku użyteczna jest klasa wzorca. Taka klasa wzorca jest wzorcową wersją klasy C++. Kiedy używasz klasy wzorca, masz możliwość podania „typu”, dla którego specjalizowana jest dana klasa. W ten sposób można utworzyć egzemplarze klasy `Human` wraz z parametrem `Age` typu `long long` i inne egzemplarze z parametrem `Age` typu `short`.

Prosta klasa wzorca zawierająca pojedynczy parametr `T` wzorca może być zapisana w taki sposób:

```
template <typename T>
class MyFirstTemplateClass
{
  public:
     void SetValue (const T& newValue) { Value = newValue; }
     const T& GetValue() const {return Value;}
  private:
     T Value;
};
```

Klasa `MyFirstTemplateClass` została zaprojektowana do przechowywania zmiennej typu `T` — typ ten będzie przypisany w chwili użycia wzorca. Spójrzmy więc na przykład użycia tej klasy wzorca:

```
MyFirstTemplateClass <int> mHoldInteger;   // Ustanowienie wzorca.
HoldInteger.SetValue (5);
std::cout << "Przechowywana wartość to: " << HoldInteger.GetValue() <<
↪std::endl;
```

Powyższa klasa wzorca została użyta w celu przechowywania i pobierania obiektu typu `int`. Oznacza to, że klasa `Template` została ustanowiona dla parametru wzorca typu `int`. Tę samą klasę można wykorzystać do współpracy z ciągami tekstowymi znaków, dokładnie w ten sam sposób:

```
MyFirstTemplateClass <char*> HoldString;
HoldInteger.SetValue ("Przykładowy ciąg tekstowy");
std::cout << "Przechowywana wartość to: " << HoldInteger.GetValue() <<
↪std::endl;
```

Klasa definiuje wzór, który jest ponownie wykorzystywany podczas implementacji różnych typów danych. Klasa `Human` umożliwiająca dostosowanie do własnych potrzeb i pozwalająca na wybór typu parametru `Age` będzie przedstawiała się następująco:

```
template <typename T>
class CustomizableHuman
{
public:
   void SetAge (const T& newValue) { Age = newValue; }
   const T& GetAge() const {return Age;}

private:
   T Age;    // T jest typem wybranym w celu dostosowania do własnych potrzeb tego wzorca!
};
```

W trakcie użycia wzorca możesz wskazać typ, stosując odpowiednią składnię tworzenia egzemplarza wzorca:

```
CustomizableHuman<int> NormalLifeSpan;   // Utworzenie egzemplarza dla typu int.
   NormalLifeSpan.SetAge(80);
CustomizableHuman<long long> LongLifeSpan;   // Utworzenie egzemplarza dla typu
                                             // long long.
   LongLifeSpan.SetAge(3147483647);
CustomizableHuman<short> ShortLifeSpan; // Utworzenie egzemplarza dla typu short.
   ShortLifeSpan.SetAge(40);
```

Ustanawianie i specjalizacja wzorca

Dla wzorców terminologia ulega drobnej zmianie. Słowo *ustanawianie* używane w kontekście klasy zwykle odnosi się do *obiektów* jako egzemplarzy *klas*.

Jednak w przypadku wzorców *ustanawianie* jest *procesem* tworzenia określonego typu z deklaracji wzorca oraz jednego lub wielu argumentów wzorca.

Dlatego też, jeśli spojrzymy na deklarację wzorca:

```
template <typename T>
class TemplateClass
{
   T m_member;
};
```

wówczas podczas używania tego wzorca kod zapiszemy tak:

```
TemplateClass <int> IntTemplate;
```

Określony typ utworzony jako wynik tego ustanawiania jest nazywany *specjalizacją*.

Deklarowanie wzorców z wieloma parametrami

Lista parametrów wzorca może być rozszerzona i pozwala na deklarację wielu parametrów rozdzielonych przecinkami. Dlatego też, jeżeli trzeba zadeklarować klasę ogólną przechowującą parę obiektów, które mogą być różnych typów, można użyć konstrukcji przedstawionej w poniższym przykładzie (wyświetla klasę wzorca z dwoma parametrami wzorca):

```
template <typename T1, typename T2>
class HoldsPair
{
  private:
     T1 Value1;
     T2 Value2;
  public:
     // Konstruktor inicjalizujący zmienne składowe.
     HoldsPair (const T1& value1, const T2& value2)
     {
        Value1 = value1;
        Value2 = value2;
     };
     // Deklaracje pozostałych funkcji.
};
```

W przedstawionym kodzie klasa `HoldsPair` akceptuje dwa parametry wzorca o nazwach `T1` i `T2`. Klasy tej można użyć w celu przechowywania dwóch obiektów tego samego typu bądź różnych typów, jak to zostało przedstawione poniżej:

```
// Ustanowienie wzorca, który paruje typ int z typem double.
HoldsPair <int, double> pairIntDouble (6, 1.99);
// Ustanowienie wzorca, który paruje typ int z typem int.
HoldsPair <int, int> pairIntDouble (6, 500);
```

Deklarowanie wzorców z parametrami domyślnymi

Poprzednią wersję `HoldsPair <...>` moglibyśmy zmodyfikować w celu zadeklarowania typu `int` jako domyślnego typu parametru wzorca.

```
template <typename T1=int, typename T2=int>
class HoldsPair
{
   // Deklaracje metod.
};
```

Konstrukcja jest podobna do funkcji definiujących wartości domyślne parametrów danych wejściowych z wyjątkiem faktu, że w omawianym przypadku definiowane są domyślne *typy*.

Kolejne użycie `HoldsPair` można więc skrócić do postaci:

```
// Ustanowienie wzorca, który paruje typ int z typem int (typ domyślny).
HoldsPair <> pairIntDouble (6, 500);
```

Przykład wzorca

Pora na rozbudowanie omawianego dotąd wzorca `HoldsPair`. W listingu 14.4 przedstawiono rozbudowaną wersję tego wzorca.

Listing 14.4. Klasa wzorca wraz z atrybutami składowymi

```
 0: // Deklaracja domyślnych typów parametrów: pierwszy to int, natomiast drugi to float.
 1: template <typename T1=int, typename T2=double>
 2: class HoldsPair
 3: {
 4: private:
 5:    T1 Value1;
 6:    T2 Value2;
 7: public:
 8:    // Konstruktor inicjalizujący zmienne składowe.
 9:    HoldsPair (const T1& value1, const T2& value2)
10:    {
11:       Value1 = value1;
12:       Value2 = value2;
13:    };
14:
15:    // Funkcje akcesora.
16:    const T1 & GetFirstValue () const
17:    {
18:       return Value1;
```

```
19:    };
20:
21:    const T2& GetSecondValue () const
22:    {
23:       return Value2;
24:    };
25: };
26:
27: #include <iostream>
28: using namespace std;
29:
30: int main ()
31: {
32:    // Utworzenie dwóch egzemplarzy klasy wzorca HoldsPair.
33:    HoldsPair <> mIntFloatPair (300, 10.09);
34:    HoldsPair<short,char*>mShortStringPair(25,"Poznaj wzorce,
       ↪pokochaj C++");
35:
36:    // Wartości danych wyjściowych znajdujące się w pierwszym obiekcie…
37:    cout << "Pierwszy obiekt zawiera -" << endl;
38:    cout << "Wartość 1: " << mIntFloatPair.GetFirstValue () <<  endl;
39:    cout << "Wartość 2: " << mIntFloatPair.GetSecondValue () << endl;
40:
41:    // Wartości danych wyjściowych znajdujące się w drugim obiekcie…
42:    cout << "Drugi obiekt zawiera -" << endl;
43:    cout << "Wartość 1: " << mShortStringPair.GetFirstValue () <<  endl;
44:    cout << "Wartość 2: " << mShortStringPair.GetSecondValue () << endl;
45:
46:    return 0;
47: }
```

Wynik ▼

```
Pierwszy obiekt zawiera -
Wartość 1: 300
Wartość 2: 10.09
Drugi obiekt zawiera -
Wartość 1: 25
Wartość 2: Poznaj wzorce, pokochaj C++
```

Analiza ▼

W tym prostym programie pokazano, jak można zadeklarować klasę wzorca `HoldsPair` w celu przechowywania pary wartości typów, które będą zależały od listy parametrów wzorca. Wiersz 1. zawiera listę parametrów wzorca definiującą dwa parametry `T1` i `T2` wraz z typami domyślnymi (odpowiednio)

int i double. Funkcje akcesorów GetFirstValue() i GetSecondValue() mogą być użyte do sprawdzenia wartości przechowywanych przez obiekt. Warto zwrócić uwagę na to, jak funkcje akcesorów GetFirstValue() i GetSecondValue() zostały dostosowane do zwrotu odpowiedniego typu obiektu na podstawie składni ustanawiania wzorca. Otrzymałeś kontrolę nad definicją wzoru w klasie HoldsPair, który można ponownie wykorzystać w celu dostarczenia tej samej logiki dla zmiennych różnego typu. W ten sposób wzorce przyczyniają się do zwiększenia możliwości ponownego wykorzystania kodu.

Klasy wzorców i statyczne elementy składowe

Wzorce są matrycami klas, które z kolei są matrycami dla obiektów. W jaki sposób statyczne elementy składowe funkcjonują w klasie wzorców? Z lekcji 9., zatytułowanej „Klasy i obiekty", wiesz, że zadeklarowanie statycznego elementu składowego klasy powoduje jego współdzielenie przez wszystkie egzemplarze danej klasy. Podobnie jest w klasie wzorców, statyczny element składowy klasy pozostaje współdzielony przez wszystkie egzemplarze klasy wzorców o tej samej specjalizacji. Dlatego też element statyczny X w klasie wzorców T pozostaje statyczny we wszystkich egzemplarzach T specjalizowanych dla int. Podobnie element X pozostanie statyczny dla wszystkich egzemplarzy T specjalizowanych dla double, niezależnie od innych specjalizacji dla int. Innymi słowy, można powiedzieć, że kompilator tworzy dwie wersje: X_int i X_double. Takie rozwiązanie zaprezentowano w listingu 14.5.

Listing 14.5. Efekt użycia zmiennych statycznych w klasie wzorców, a tym samym w jej egzemplarzach

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: template <typename T>
 4: class TestStatic
 5: {
 6: public:
 7:    static int StaticValue;
 8: };
 9:
10: // Inicjalizacja statycznego elementu składowego.
11: template<typename T> int TestStatic<T>::StaticValue;
12:
13: int main()
14: {
```

```
15:    TestStatic<int> Int_Year;
16:    cout << "Ustawienie StaticValue dla Int_Year jako 2011" << endl;
17:    Int_Year.StaticValue = 2011;
18:    TestStatic<int> Int_2;
19:
20:    TestStatic<double> Double_1;
21:    TestStatic<double> Double_2;
22:    cout << "Ustawienie StaticValue dla Double_2 jako 1011" << endl;
23:    Double_2.StaticValue = 1011;
24:
25:    cout << "Int_2.StaticValue = " << Int_2.StaticValue << endl;
26:    cout << "Double_1.StaticValue = " << Double_1.StaticValue << endl;
27:
28:    return 0;
29: }
```

Wynik ▼

```
Ustawienie StaticValue dla Int_Year jako 2011
Ustawienie StaticValue dla Double_2 jako 1011
Int_2.StaticValue = 2011
Double_1.StaticValue = 1011
```

Analiza ▼

W wierszach 17. i 21. ustawiono element składowy `StaticValue` dla egzemplarza klasy wzorca jako typu (odpowiednio) `int` i `double`. W funkcji `main()`, a dokładnie w wierszach 25. i 26., odczytano te wartości, ale przy użyciu innych elementów składowych egzemplarza: `Int_2` i `Double_1`. Dane wyjściowe pokazują, że otrzymujemy dwie odmienne wartości `StaticValue`: 2011 w przypadku egzemplarza specjalizowanego dla `int` oraz 1011 dla egzemplarza specjalizowanego dla `double`. To potwierdza, że kompilator zagwarantował niezmienne zachowanie statycznego elementu składowego dla klasy danego typu. Każda specjalizacja klasy wzorca ma własną zmienną statyczną.

Uwaga

W wierszu 11. listingu 14.5 możesz zobaczyć składnię tworzenia statycznego elementu składowego w klasie wzorca.

```
template<typename T> int TestStatic<T>::StaticValue;
```

Jest ona zgodna ze wzorem:

```
template<parametry wzorca> TypStatyczny NazwaKlasy<Argumenty
wzorca>::ZmiennaStatyczna;
```

C++11

Użycie static_assert do przeprowadzania operacji sprawdzania w trakcie kompilacji

Jest to funkcja standardu C++11 pozwalająca na zablokowanie kompilacji, jeśli pewne operacje sprawdzenia nie zostaną przeprowadzone. Wprawdzie brzmi to dość dziwnie, ale to całkiem użyteczna możliwość w klasach wzorców. Być może chcesz mieć pewność, że na podstawie klasy wzorca nie zostanie utworzony egzemplarz dla liczby całkowitej. `static_assert` to asercja w trakcie kompilacji, która wyświetla komunikaty w środowisku programistycznym (lub konsoli):

```
static_assert(sprawdzane wyrażenie, "Komunikat błędu w przypadku
niepowodzenia");
```

Aby zagwarantować, że klasa wzorca nie zostanie utworzona dla typu `int`, możesz użyć `static_assert` wraz z `sizeof(T)`, porównać z `sizeof(int)` i wyświetlić komunikat błędu w przypadku porównania zakończonego niepowodzeniem:

```
static_assert(sizeof(T) != sizeof(int), "Typ int jest niedozwolony!");
```

Tego rodzaju klasa wzorca używająca `static_assert` do uniemożliwienia kompilacji w przypadku inicjalizacji pewnych typów została zaprezentowana w listingu 14.6.

Listing 14.6. Klasa wzorca używająca static_assert do uniemożliwienia utworzenia egzemplarza dla typu int

```
 0: template <typename T>
 1: class EverythingButInt
 2: {
 3: public:
 4:    EverythingButInt()
 5:    {
 6:       static_assert(sizeof(T) != sizeof(int), "Typ int jest niedozwolony!");
 7:    }
 8: };
 9:
10: int main()
11: {
12:    EverythingButInt<int> test;   // Utworzenie egzemplarza dla typu int.
13:    return 0;
14: }
```

Wynik ▼

Jeżeli kompilacja zakończy się niepowodzeniem, nie będzie żadnych danych wyjściowych programu, a jedynie zdefiniowany komunikat błędu:

```
Błąd: Typ int jest niedozwolony!
```

Analiza ▼

Odpowiedni komunikat błędu został zdefiniowany w wierszu 6. W ten sposób `static_assert` to oferowany przez standard C++11 sposób pomagający w ochronie kodu wzorca przed utworzeniem egzemplarza określonego typu.

Użycie wzorców w praktycznym programowaniu C++

Najważniejsze i oferujące największe możliwości aplikacje wzorców znajdują się w standardowej bibliotece wzorców (STL, ang. *Standard Template Library*). Biblioteka STL składa się ze zbioru klas i funkcji wzorców zawierających ogólne klasy narzędziowe i algorytmy. Klasy wzorców STL pozwalają na implementację tablic dynamicznych, list, kontenerów par klucz-wartość, gdzie algorytmy, takie jak sortowanie, działają na tych kontenerach i przetwarzają znajdujące się w nich dane.

Zdobyta wcześniej wiedza na temat składni wzorców będzie przydatna podczas używania kontenerów i funkcji STL, które bardziej szczegółowo zostaną przedstawione w kolejnych lekcjach tej książki. Z kolei lepsze zrozumienie kontenerów STL i algorytmów pomoże w tworzeniu bardziej efektywnych aplikacji STL, które będą używały przetestowanych i sprawdzonych implementacji STL. W ten sposób unikniesz poświęcania czasu na tworzenie wszystkiego od zera.

TAK	NIE
Używaj wzorców do implementacji ogólnych koncepcji.	Nie zapominaj o regułach użycia const podczas tworzenia klas i funkcji wzorców.
Wybieraj wzorce zamiast makr.	Nie zapominaj, że statyczne elementy składowe znajdujące się w klasie wzorca pozostaną statyczne dla każdej specjalizacji typu danej klasy.

Podsumowanie

W tej lekcji poznałeś więcej szczegółów na temat pracy z preprocesorem. Za każdym razem, kiedy uruchamiasz kompilator, preprocesor działa jako pierwszy i przekształca dyrektywy, takie jak `#define`.

Preprocesor przeprowadza zastępowanie tekstowe, choć dzięki zastosowaniu makr zastąpienia te mogą być znacznie bardziej skomplikowane. Funkcje makr oferują znacznie bardziej złożone możliwości w zakresie zastępowania tekstu na podstawie argumentów przekazywanych do makra w trakcie kompilacji. Bardzo ważne jest umieszczanie nawiasów wokół każdego argumentu makra, co pozwala na zapewnienie prawidłowego przebiegu operacji zastępowania.

Wzorce pomagają w tworzeniu kodu, który można ponownie wykorzystać. Dostarczają programiście wzór możliwy do wykorzystania podczas pracy z różnymi typami danych. Zwykle stanowią bezpieczne dla typów zamienniki makr. Korzystając ze zdobytej w tej lekcji wiedzy na temat makr, możesz przystąpić do poznawania sposobu wykorzystania standardowej biblioteki wzorców!

Pytania i odpowiedzi

Pytanie: Dlaczego w plikach nagłówkowych powinienem stosować zabezpieczenie przed wielokrotnym dołączaniem kodu?

Odpowiedź: Tzw. wartownicy dołączania (ang. *inclusion guard*) wykorzystują dyrektywy `#ifndef`, `#define` i `#endif` do ochrony pliku nagłówkowego przed błędami związanymi z wielokrotnym lub rekurencyjnym dołączaniem innych plików nagłówkowych. Czasami mogą również skrócić czas kompilacji.

Pytanie: Dlaczego miałbym używać funkcji makro zamiast wzorców, skoro w obu konstrukcjach mogę umieścić taką samą funkcjonalność?

Odpowiedź: W idealnej sytuacji zawsze powinieneś wybierać wzorce, ponieważ pozwalają na zdefiniowanie ogólnej implementacji, a ponadto zapewniają bezpieczeństwo typów. Makra nie zapewniają bezpieczeństwa typów i lepiej ich unikać.

Pytanie: Czy podczas wywoływania funkcji wzorca muszę podawać argumenty wzorca?

Odpowiedź**:** Zwykle nie, ponieważ kompilator może je określić automatycznie, analizując argumenty użyte w wywołaniu funkcji.

Pytanie: Ile egzemplarzy zmiennych statycznych istnieje dla danej klasy wzorca?

Odpowiedź: To zależy wyłącznie od liczby typów, dla których dana klasa wzorca będzie ustanawiana. Dlatego też, jeśli klasa wzorca będzie ustanawiana dla typów `int`, `string` i własnego `X`, możesz oczekiwać dostępności trzech egzemplarzy zmiennej statycznej, po jednej dla każdej specjalizacji wzorca.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Co to jest wartownik dołączania?
2. Spójrz na poniższe makro:

   ```
   #define SPLIT(x) x / 5
   ```

   Jaki będzie wynik, jeśli makro zostanie wywołane wraz z wartością 20?
3. Jaki będzie wynik, gdy makro `SPLIT` z ćwiczenia 2. zostanie wywołane z wartością 10+10?
4. W jaki sposób można zmodyfikować makro `SPLIT`, aby uniknąć nieprawidłowych wyników?

Ćwiczenia

1. Utwórz makro, które mnoży dwie liczby.
2. Utwórz wzorzec, który będzie odpowiadał makru z ćwiczenia 1.
3. Zaimplementuj funkcję wzorca służącą do zamiany dwóch zmiennych.
4. **Łowcy błędów:** W jaki sposób możesz usprawnić poniższe makro obliczające jedną czwartą podanej wartości początkowej?

   ```
   #define QUARTER(x) (x / 4)
   ```

5. Utwórz prostą klasę wzorca przechowującą dwie tablice typów, które zostały zdefiniowane za pomocą listy parametrów klasy wzorca. Wielkość tablicy powinna wynosić 10, natomiast klasa wzorca powinna zawierać funkcje akcesorów pozwalających na przeprowadzanie operacji na elementach tablicy.

Część III

Poznajemy standardową bibliotekę wzorców (STL)

Lekcja 15

Wprowadzenie do standardowej biblioteki wzorców

Ujmując to najprościej, można stwierdzić, że standardowa biblioteka wzorców (STL) to zestaw klas i funkcji wzorcowych dostarczających programiście:

- **kontenery do przechowywania informacji;**
- **iteratory służące do uzyskiwania dostępu doprzechowywanych informacji;**
- **algorytmy pozwalające na manipulowanie treścią znajdującą się w kontenerach.**

W tej lekcji znajdziesz ogólny opis trzech wymienionych filarów STL.

Kontenery STL

Kontenery to klasy STL używane do przechowywania danych. Biblioteka STL dostarcza dwa rodzaje klas kontenerów:

- kontenery sekwencyjne;
- kontenery asocjacyjne.

Poza wymienionymi powyżej, biblioteka STL oferuje również klasy nazywane adapterami, które są wariantami tych samych kontenerów, ale z ograniczoną funkcjonalnością i przeznaczonych do ściśle określonego celu.

Kontenery sekwencyjne

Jak sama nazwa wskazuje, kontenery te są używane do przechowywania danych w sposób sekwencyjny, podobnie jak w przypadku tablic lub list. Kontenery sekwencyjne charakteryzują się krótkim czasem operacji wstawiania danych, ale są względnie powolne podczas przeprowadzania operacji wyszukiwania.

Kontenery sekwencyjne to:

- **`std::vector`** — działa jak tablica dynamiczna i rozrasta się na końcu. Wektor możesz potraktować jak półkę na książki, na końcu której dodajesz lub zabierasz książki.
- **`std::deque`** — działa podobnie jak kontener `std::vector`, ale pozwala na wstawianie nowych elementów również na początku.
- **`std::list`** — działa jak lista jednokierunkowa. Listę tę możesz traktować jak łańcuch, w którym obiekty są poszczególnymi ogniwami — w dowolnym miejscu łańcucha dodajesz lub usuwasz ogniwa, tzn. obiekty.
- **`std::forward_list`** — działa podobnie jak kontener `std::list`, ale z wyjątkiem faktu, że jest jednokierunkową listą elementów, w której iterację można przeprowadzać tylko w jednym kierunku.

Klasa STL `vector` jest podobna do tablicy i pozwala na uzyskanie dostępu do dowolnego elementu. Oznacza to możliwość uzyskania bezpośredniego dostępu do elementu klasy `vector` lub manipulowania nim za pomocą *operatora indeksowania* `[]` na podstawie jego pozycji (indeksu). Ponadto STL `vector` jest tablicą dynamiczną, może więc zmienić swoją wielkość w celu dopasowania do wymagań działającej aplikacji. Aby umożliwić swobodny dostęp do elementu tablicy, kiedy podana jest jego pozycja, większość

implementacji STL `vector` przechowuje wszystkie elementy w sąsiadujących położeniach. Dlatego też tablica `vector`, która musi zmienić swoją wielkość, bardzo często powoduje zmniejszenie wydajności aplikacji, w zależności od typu obiektów w niej zawartych. Wektor został wprowadzony po raz pierwszy w listingu 4.4 znajdującym się w lekcji 4., zatytułowanej „Tablice i ciągi tekstowe". Szczegółowe omówienie tego kontenera znajdziesz w lekcji 17., zatytułowanej „Dynamiczne klasy tablic w STL".

STL `list` można uznać za dostępną w bibliotece STL implementację zwykłej listy. Wprawdzie zapewniony jest swobodny dostęp do elementów znajdujących się w obiekcie `list` (podobnie jak w tablicy `vector`), jednak obiekt `list` pozwala na umieszczenie elementów w niesąsiadujących ze sobą sekcjach pamięci. Z tego powodu podczas pracy z obiektem `list` nie zachodzi spadek wydajności, który następuje w przypadku tablicy `vector`, jeśli ta tablica musi zmienić położenie swojej wewnętrznej tablicy. Szczegółowe omówienie klasy STL `list` znajdziesz w lekcji 18., zatytułowanej „Klasy STL list i forward_list".

Kontenery asocjacyjne

Kontenery asocjacyjne to takie, które przechowują dane w postaci posortowanej — podobnie jak słownik. W wyniku tego czas wstawiania elementu jest dłuższy, ale taki kontener pokazuje swoje zalety w trakcie operacji wyszukiwania.

Oto kontenery asocjacyjne dostarczane przez STL.

- **`std::set`** — posortowana lista unikalnych wartości po umieszczeniu w kontenerze o złożoności logarytmicznej.
- **`std::unordered_set`** — posortowana lista unikalnych wartości po umieszczeniu w kontenerze charakteryzującym się niemal stałym poziomem skompilowania. Kontener ten został wprowadzony w standardzie C++11.
- **`std::map`** — przechowuje pary klucz-wartość posortowane pod względem ich unikalnych kluczy po umieszczeniu w kontenerze o złożoności logarytmicznej.
- **`std::unordered_set`** — przechowuje pary klucz-wartość posortowane pod względem ich unikalnych kluczy po umieszczeniu w kontenerze charakteryzującym się niemal stałym poziomem skompilowania. Kontener ten został wprowadzony w standardzie C++11.

- **`std::multiset`** — podobny do zbioru. Ponadto obsługuje możliwość przechowywania wielu elementów posiadających tę samą wartość, zatem wartość nie musi być unikalna.
- **`std::unordered_set`** — podobny do `unordered_set`. Ponadto obsługuje możliwość przechowywania wielu elementów posiadających tę samą wartość, zatem wartość nie musi być unikalna. Kontener ten został wprowadzony w standardzie C++11.
- **`std::multimap`** — podobny do `map`. Ponadto obsługuje możliwość przechowywania par klucz-wartość, których klucze nie muszą być unikalne.
- **`std::unordered_set`** — podobny do `map`. Ponadto obsługuje możliwość przechowywania par klucz-wartość, których klucze nie muszą być unikalne. Kontener ten został wprowadzony w standardzie C++11.

Kryteria sortowania kontenerów STL można dostosować do własnych potrzeb za pomocą funkcji predykatu.

Wskazówka

Pewne implementacje STL zawierają również kontenery asocjacyjne, takie jak `hash_set`, `hash_multiset`, `hash_map` i `hash_multimap`. Są one podobne do kontenerów `unordered_*` obsługiwanych przez standard. W pewnych sytuacjach kontenery `hash_*` i `unordered_*` mogą być jeszcze lepsze podczas wyszukiwania elementu pod tym względem, że oferują stały czas dostępu, niezależny od wielkości kontenera. Wymienione kontenery zazwyczaj udostępniają także metody publiczne, które są identyczne z metodami oferowanymi przez kontenery standardowe, a więc pozostają bardzo łatwe w użyciu.

Warto zwrócić uwagę na pewien fakt — stosowanie kontenerów zgodnych ze standardami powoduje, że kod staje się łatwiejszy do przenoszenia między platformami i kompilatorami. Jest także możliwe, że logarytmiczny spadek wydajności w przypadku używania kontenera zgodnego ze standardami nie wpłynie znacząco na Twoją aplikację.

Wybór odpowiedniego kontenera

Twoja aplikacja niewątpliwie może mieć wymagania, które będą spełniane przez więcej niż tylko jeden kontener STL. Trzeba więc dokonać ważnego wyboru, ponieważ błędna decyzja może skutkować niewłaściwym funkcjonowaniem aplikacji.

Dlatego też przed podjęciem decyzji ważne jest przeanalizowanie zalet i wad poszczególnych kontenerów. Więcej informacji na ten temat znajduje się w tabeli 15.1.

Tabela 15.1. Właściwości klas kontenerów STL

Kontener	Zalety	Wady
`std::vector` (kontener sekwencyjny)	Szybkie (niezmienny czas) wstawianie elementu na końcu. Dostęp podobny do stosowanego w tablicy.	Zmiana wielkości może skutkować spadkiem wydajności. Czas wyszukiwania jest proporcjonalny do liczby elementów w kontenerze. Wstawianie elementów tylko na końcu.
`std::deque` (kontener sekwencyjny)	Wszystkie zalety kontenera `vector`, a ponadto możliwość wstawiania elementów (przy niezmiennym czasie) na początku kontenera.	Wymienione powyżej wady kontenera `vector` w zakresie wydajności i wyszukiwania dotyczą także kontenera deque. W przeciwieństwie do kontenera `vector`, specyfikacja `deque` nie wymaga funkcji `reserve()` pozwalającej programiście na zarezerwowanie pamięci używanej jako `vector` — funkcja ta unika częstej zmiany wielkości kontenera w celu poprawienia wydajności działania aplikacji.
`std::list` (kontener sekwencyjny)	Stały czas wstawiania elementów na początku, w środku i na końcu listy. Usuwanie elementów z listy zawsze zabiera taką samą ilość czasu, niezależnie od położenia elementu.	Nie ma możliwości uzyskania swobodnego dostępu do elementów na podstawie ich indeksu, jak ma to miejsce w tablicy. Wyszukiwanie może być wolniejsze niż w kontenerze `vector`, ponieważ elementy nie są układane w pamięci w sąsiadujących lokalizacjach.

Tabela 15.1. Właściwości klas kontenerów STL (cd.)

Kontener	Zalety	Wady
	Wstawianie lub usuwanie elementów nie powoduje unieważnienia iteratorów, które wskazują inne elementy listy.	Czas wyszukiwania jest proporcjonalny do liczby elementów znajdujących się w kontenerze.
`std::forward_list` (kontener sekwencyjny)	Klasa listy jednokierunkowej pozwalającej na przeprowadzanie iteracji tylko w jednym kierunku.	Wstawianie elementów jest możliwe tylko na początku listy przy użyciu metody `push_front()`.
`std::set` (kontener asocjacyjny)	Wyszukiwanie nie jest bezpośrednio proporcjonalne do liczby elementów w kontenerze, a raczej logarytmiczne do ich liczby, stąd często bywa znacznie szybsze niż w kontenerach sekwencyjnych.	Wstawianie elementów przebiega wolniej niż w kontenerach sekwencyjnych, ponieważ elementy są sortowane podczas wstawiania.
`std::unordered_set` (kontener asocjacyjny)	Wyszukiwanie, wstawianie i usuwanie elementów w tego rodzaju kontenerze jest niemal niezależne od liczby znajdujących się w nim elementów.	Ponieważ elementy praktycznie nie są posortowane, nie można polegać na ich względnych położeniach w kontenerze.
`std::multiset` (kontener asocjacyjny)	Powinien być używany, kiedy trzeba przechowywać również nieunikalne elementy.	Wstawianie elementów przebiega wolniej niż w kontenerach sekwencyjnych, ponieważ elementy są sortowane podczas wstawiania.
`std::unordered_multiset` (kontener asocjacyjny)	Ten kontener powinien być wybierany zamiast `unordered_set`, gdy trzeba przechowywać także wartości nieunikalne. Wydajność jest podobna do oferowanej przez kontener `unordered_set`; praktycznie taki sam czas wyszukiwania, wstawiania i usuwania elementów niezależnie od wielkości kontenera.	Ponieważ elementy praktycznie nie są posortowane, nie można polegać na ich względnych położeniach w kontenerze.

Tabela 15.1. Właściwości klas kontenerów STL (cd.)

Kontener	Zalety	Wady
`std::map` (kontener asocjacyjny)	Kontener par klucz-wartość, w którym wyszukiwanie jest bezpośrednio proporcjonalne do logarytmu liczby elementów w kontenerze, stąd często bywa znacznie szybsze niż w przypadku kontenerów sekwencyjnych.	Elementy (pary) są sortowane podczas ich wstawiania i dlatego operacja wstawiania będzie wolniejsza niż w przypadku par w kontenerze sekwencyjnym.
`std::unordered_map` (kontener asocjacyjny)	Zaletą jest praktycznie taki sam czas wyszukiwania, wstawiania i usuwania elementów, niezależnie od wielkości kontenera.	Elementy praktycznie nie są posortowane i dlatego ten kontener nie nadaje się w sytuacjach, gdy ważna jest kolejność elementów.
`std::multimap` (kontener asocjacyjny)	Wybierany zamiast `std::map`, kiedy trzeba użyć kontenera do przechowywania par klucz-wartość zawierających elementy o nieunikalnych kluczach.	Wstawianie elementów przebiega wolniej niż w kontenerach sekwencyjnych, ponieważ elementy są sortowane w trakcie ich wstawiania.
`std::unordered_multimap` (kontener asocjacyjny)	Wybierany zamiast `std::multimap`, kiedy trzeba użyć kontenera do przechowywania par klucz-wartość zawierających elementy o nieunikalnych kluczach. Zaletą jest praktycznie taki sam czas wyszukiwania, wstawiania i usuwania elementów, niezależnie od wielkości kontenera.	Elementy praktycznie nie są posortowane i dlatego ten kontener nie nadaje się w sytuacjach, gdy ważna jest względna kolejność elementów.

Adaptery

Adaptery to specjalne wersje kontenerów sekwencyjnych i asocjacyjnych, które mają ograniczoną funkcjonalność i są przeznaczone do ściśle określonych celów. Poniżej wymieniono najważniejsze kontenery specjalne.

- **`std::stack`** — przechowuje elementy w stosie typu LIFO (ang. *Last-In-First-Out*, ostatni na wejściu, pierwszy na wyjściu); pozwala na wstawienie (push) i usuwanie (pop) elementów na górze stosu.
- **`std::queue`** — przechowuje elementy w kolejce typu FIFO (ang. *First-In-First-Out*, pierwszy na wejściu, pierwszy na wyjściu); pozwala na usuwanie elementów w kolejności ich wstawiania.
- **`std::priority_queue`** — przechowuje posortowane elementy w taki sposób, że element o największej wartości zawsze jest pierwszym elementem kolejki.

Szczegółowo omówienie wymienionych powyżej kontenerów znajdziesz w lekcji 24., zatytułowanej „Kontenery adaptacyjne: stack i queue".

Iteratory STL

Najprostszym przykładem iteratora jest wskaźnik. Mając wskaźnik do pierwszego elementu tablicy, możesz go inkrementować i otrzymać wskaźnik do kolejnego elementu; w wielu przypadkach możesz też manipulować elementem w danym położeniu.

W bibliotece STL iteratory są wzorcami klas, które w pewien sposób stanowią uogólnienie wskaźników. To wzorce klas udostępniają programiście uchwyt, za pomocą którego może pracować z kontenerami STL, manipulować nimi oraz przeprowadzać na nich operacje. Warto zwrócić uwagę na fakt, że operacje mogą być algorytmami STL będącymi funkcjami wzorców. Iteratory stanowią pomost pozwalający tym funkcjom wzorców na współpracę z kontenerami będącymi wzorcami klas.

Oto ogólna klasyfikacja iteratorów dostarczanych przez STL.

- **Iteratory danych wejściowych** — iterator ten może wskazywać odniesienie do obiektu. Obiekt może być np. zbiorem. Najprostsze rodzaje iteratorów danych wejściowych gwarantują jedynie dostęp w celu odczytu danych.
- **Iteratory danych wyjściowych** — iterator ten pozwala programiście na zapisanie zbioru. Iteratory danych wyjściowych ścisłych typów gwarantują jedynie dostęp w celu zapisu danych.

A to dodatkowy podział wymienionych wyżej podstawowych rodzajów iteratorów.

- **Iterator poruszający się tylko do przodu** — ulepszony iterator danych wejściowych i wyjściowych pozwala zarówno na wejście, jak i wyjście

danych. Iteratory poruszające się tylko do przodu mogą być stałe, wtedy pozwalają jedynie na dostęp tylko do odczytu do obiektu wskazywanego przez iterator. W przeciwnym razie pozwalają zarówno na operacje odczytu, jak i zapisu, czyli są zmienne. Iterator poruszający się tylko do przodu najczęściej znajduje zastosowanie w liście jednokierunkowej.

- **Iterator dwukierunkowy** — ulepszony iterator poruszania się tylko do przodu, w którym wartość może być także dekrementowana, co pozwala iteratorowi na poruszanie się wstecz. Iterator dwukierunkowy najczęściej znajduje zastosowanie w liście dwukierunkowej.
- **Iterator swobodnego dostępu** — ogólnie rzecz biorąc, to ulepszona koncepcja iteratora dwukierunkowego, który pozwala na dodawanie i odejmowanie wartości przesunięcia lub na odjęcie jednego iteratora od innego w celu znalezienia względnego odstępu między dwoma obiektami w zbiorze. Iterator swobodnego dostępu najczęściej znajduje zastosowanie w tablicy.

Uwaga

Na poziomie implementacji *ulepszenie* można traktować jak *dziedziczenie* bądź *specjalizację*.

Algorytmy STL

Wyszukiwanie, sortowanie, odwracanie kolejności itp. to standardowe operacje programistyczne, które nie wymagają od programisty ponownego wynajdywania implementacji. W celu pomocy programiście w spełnieniu najczęściej spotykanych wymagań biblioteka STL dostarcza te funkcje w postaci algorytmów STL, doskonale działających z kontenerami za pomocą iteratorów.

Oto niektóre z najczęściej używanych algorytmów STL.

- **`std::find`** — pomaga w wyszukaniu wartości w zbiorze.
- **`std::find_if`** — pomaga w wyszukaniu wartości w zbiorze na podstawie zdefiniowanego przez użytkownika predykatu.
- **`std::reverse`** — odwraca kolejność w zbiorze.
- **`std::remove_if`** — pomaga w usunięciu elementu ze zbioru na podstawie zdefiniowanego przez użytkownika predykatu.
- **`std::transform`** — pomaga w zastosowaniu zdefiniowanej przez użytkownika funkcji przekształcenia względem elementów kontenera.

Wymienione powyżej algorytmy są funkcjami wzorcowymi w przestrzeni nazw `std` i wymagają dołączenia standardowego nagłówka `<algorithm>`.

Oddziaływania między kontenerami i algorytmami za pomocą iteratorów

Na podstawie przykładowego programu przekonamy się, jak można iteratory połączyć z kontenerami i algorytmami STL. W programie przedstawionym w listingu 15.1 użyto kontenera sekwencyjnego `std::vector`, który jest podobny do tablicy dynamicznej. Tablica ta jest stosowana do przechowywania wartości w postaci liczb całkowitych, natomiast program wyszukuje wartość w zbiorze za pomocą algorytmu `std::find`. Warto zwrócić uwagę na sposób, w jaki *iteratory* stanowią pomost łączący *algorytmy* i *kontenery*, na których operują. Nie przejmuj się skomplikowaną składnią lub funkcjonalnością programu. Szczegółowe omówienie kontenerów, takich jak `std::vector`, i algorytmów, takich jak `std::find`, znajdziesz w lekcjach (odpowiednio) 17., zatytułowanej „Dynamiczne klasy tablic w STL”, i 23., zatytułowanej „Algorytmy STL”. Jeżeli kod okaże się zbyt skomplikowany, pomiń teraz ten punkt.

Listing 15.1. Wyszukanie elementu i jego pozycji w kontenerze vector

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main ()
{
    // Tablica dynamiczna liczb całkowitych.
    vector <int> vecIntegerArray;

    // Umieszczenie w tablicy przykładowych liczb całkowitych.
    vecIntegerArray.push_back(50);
    vecIntegerArray.push_back(2991);
    vecIntegerArray.push_back(23);
    vecIntegerArray.push_back(9999);

    cout << "Zawartość kontenera vector jest następująca: " << endl;

    // Przejście przez kontener vector i odczytanie wartości za pomocą iteratora.
    vector <int>::iterator iArrayWalker = vecIntegerArray.begin();

    while (iArrayWalker != vecIntegerArray.end())
```

```
23:     {
24:         // Wyświetlenie wartości na ekranie.
25:         cout << *iArrayWalker << endl;
26:
27:         // Inkrementacja iteratora w celu uzyskania dostępu do kolejnego elementu.
28:         ++ iArrayWalker;
29:     }
30:
31:     // Wyszukanie elementu (powiedzmy 2991) w tablicy przy użyciu algorytmu 'find'...
32:     vector <int>::iterator iElement = find (vecIntegerArray.begin()
33:                             ,vecIntegerArray.end(), 2991);
34:
35:     // Sprawdzenie, czy wartość została znaleziona.
36:     if (iElement != vecIntegerArray.end())
37:     {
38:         // Wartość została znaleziona... Określenie jej pozycji w tablicy.
39:         int Position = distance (vecIntegerArray.begin(), iElement);
40:         cout << "Wartość " << *iElement;
41:         cout << " została znaleziona w kontenerze vector na pozycji: " <<
            ↪Position << endl;
42:     }
43:
44:     return 0;
45: }
```

Wynik ▼

```
Zawartość kontenera vector jest następująca:
50
2991
23
9999
Wartość 2991 została znaleziona w kontenerze vector na pozycji: 1
```

Analiza ▼

W listingu 15.1 przedstawiono użycie iteratorów podczas przejścia przez kontener `vector` oraz jako interfejsu pomagającego w połączeniu algorytmów, takich jak `find`, z kontenerami, takimi jak `vector`, zawierającymi dane, na których te algorytmy mają operować. Obiekt iteratora `isArrayWalker` został zadeklarowany w wierszu 20. i zainicjalizowany na początku kontenera, tzn. `vector` korzysta z wartości zwrotnej funkcji składowej `begin()`. W wierszach od 22. do 29. zademonstrowano użycie iteratora w pętli w celu zlokalizowania i wyświetlenia elementów znajdujących się w kontenerze `vector`. Odbywa się

to w sposób podobny do wyświetlenia zawartości tablicy statycznej. Użycie iteratora jest takie samo we wszystkich kontenerach STL. Wszystkie zawierają funkcję `begin()` wskazującą pierwszy element oraz funkcję `end()` wskazującą koniec kontenera *po* jego ostatnim elemencie. To wyjaśnia także, dlaczego pętla `while` w wierszu 22. zatrzymuje się na elemencie przed `end()` zamiast z `end()`. W wierszu 32. pokazano użycie `find` w celu zlokalizowania wartości w kontenerze `vector`. Wynik działania operacji `find` również jest iteratorem, a powodzenie operacji `find` jest sprawdzane za pomocą porównania iteratora z końcem kontenera, co przedstawiono w wierszu 36. Jeżeli element zostanie znaleziony, zostanie wyświetlony przez dereferencję tego iteratora (tak samo jak w przypadku dereferencji wskaźnika). Algorytm `distance` jest stosowany w obliczeniu wartości przesunięcia pozycji względem znalezionego elementu.

Jeżeli w listingu 15.1 wszystkie wystąpienia `vector` zastąpisz przez `deque`, tak otrzymany kod nadal zostanie skompilowany bez problemów i będzie doskonale działał. W taki oto sposób iteratory znacznie ułatwiają pracę z algorytmami i kontenerami.

C++11

Użycie słowa kluczowego auto pozwalającego kompilatorowi na definicję typu

W listingu 15.1 znalazło się kilka deklaracji iteratorów i wszystkie są podobne do poniższego wiersza kodu:

```
20: vector <int>::iterator iArrayWalker = vecIntegerArray.begin();
```

Powyższa deklaracja typu iteratora może wydawać się skomplikowana. Jeżeli korzystasz z kompilatora w standardzie C++11, tę deklarację możesz znacznie uprościć do postaci:

```
20: auto iArrayWalker = vecIntegerArray.begin(); // Kompilator wykrywa typ.
```

Zwróć uwagę, że zmienna zdefiniowana jako typ `auto` wymaga inicjalizacji, aby kompilator mógł wykryć typ, w zależności od wartości, z jaką jest inicjalizowana.

Klasy STL string

Biblioteka STL oferuje klasę wzorców przeznaczoną specjalnie do wykonywania operacji na ciągach tekstowych. `std::basic_string<T>` można najczęściej spotkać w dwóch specjalizacjach wzorca:

- **`std::string`** — oparta na `char` specjalizacja `std::basic_string` używana do operacji na prostych ciągach tekstowych znaków;
- **`std::wstring`** — oparta na `wchar` specjalizacja `std::basic_string` używana do operacji na większych ciągach tekstowych znaków.

Szczegółowe omówienie tej klasy narzędziowej znajdziesz w lekcji 16., zatytułowanej „Klasa string w STL”, gdzie przekonasz się jak naprawdę ułatwia ona pracę z ciągami tekstowymi.

Podsumowanie

W tej lekcji poznałeś koncepcję kontenerów STL, iteratorów i algorytmów. Wprowadzono także klasę `basic_string<T>`, która szczegółowo będzie omówiona w kolejnej lekcji. Kontenery, iteratory i algorytmy to prawdopodobnie najważniejsze składniki STL. Zrozumienie stojących za nimi koncepcji pomoże w efektywnym używaniu biblioteki STL w tworzonych aplikacjach. W lekcjach od 16. do 25. znacznie bardziej szczegółowo omówiono implementacje tych koncepcji oraz ich aplikacji.

Pytania i odpowiedzi

Pytanie: Muszę użyć tablicy, jednak nie wiem, jaką ilość elementów będzie miała. Z którego kontenera STL powinienem skorzystać?

Odpowiedź: W takim przypadku doskonale sprawdzą się kontenery `std::vector` i `std::deque`. Oba zarządzają pamięcią i mają możliwość dynamicznej zmiany wielkości wraz ze wzrostem wymagań stawianych przez aplikację.

Pytanie: Moja aplikacja wymaga częstego przeprowadzania wyszukiwania. Który rodzaj kontenera będzie najlepszy?

Odpowiedź: W przypadku częstych operacji wyszukiwania najbardziej odpowiednie będzie użycie kontenerów asocjacyjnych, takich jak `std::map` lub `std::set`, bądź ich nieuporządkowanych odpowiedników.

Pytanie: Muszę przechowywać pary klucz-wartość w celu ich szybkiego wyszukiwania. Jednak możliwe jest, że będzie istniało wiele kluczy, które nie będą unikalne. Który rodzaj kontenera powinienem wybrać?

Odpowiedź: W takim przypadku doskonale sprawdzi się kontener asocjacyjny `std::multimap`. Może on przechowywać nieunikalne pary klucz-wartość, a także oferuje szybkie wyszukiwanie, które charakteryzuje kontenery asocjacyjne.

Pytanie: Aplikacja musi mieć możliwość przeniesienia na różne platformy oraz kompilatory. Wymaganie aplikacji zakłada użycie kontenera pomagającego w szybkim wyszukiwaniu na podstawie klucza. Czy powinienem użyć `std::map`, czy `std::hash_map`?

Odpowiedź: Możliwość przeniesienia kodu jest dużym ograniczeniem i konieczne będzie zastosowanie kontenera zgodnego ze standardem. Jeżeli na wszystkich wymaganych platformach korzystasz z kompilatora zgodnego ze standardem C++11, możesz użyć kontenera `std::unordered_map`.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Jaki wybierzesz kontener, jeżeli ma zawierać tablicę elementów, a wstawianie elementów ma być możliwe na początku oraz na końcu?
2. Musisz przechowywać elementy w sposób pozwalający na ich szybkie wyszukanie. Który kontener wybierzesz?
3. Musisz przechowywać elementy w `std::set`, ale zachować możliwość zmiany kryteriów przechowywania i wyszukiwania na podstawie warunków, którymi niekoniecznie będą wartości elementów. Czy jest to możliwe?
4. Która część STL pomaga w połączeniu algorytmów z kontenerami w taki sposób, aby algorytmy mogły pracować na tych elementach?
5. Czy w aplikacji, która ma być przeniesiona na inne platformy oraz budowana za pomocą różnych kompilatorów C++, użyjesz kontenera `hash_set`?

Lekcja 16

Klasa string w STL

Standardowa biblioteka wzorców (STL) dostarcza programiście klasę kontenera, która pomaga w przeprowadzaniu operacji na ciągach tekstowych. Klasa string nie tylko dynamicznie zmienia swoją wielkość w celu spełnienia wymagań aplikacji, ale zapewnia również użyteczne funkcje lub metody pomocnicze pomagające w manipulowaniu ciągami tekstowymi i pracy z nimi. Dlatego też klasa ta dostarcza programiście możliwości użycia w tworzonych aplikacjach kodu zgodnego ze standardami, przenośnego i przetestowanego. W ten sposób programista może skoncentrować wysiłki na opracowywaniu funkcji, które mają krytyczne znaczenie dla budowanych przez niego aplikacji.

Z tej lekcji dowiesz się:

- **wymagania dotyczące używania klasy `string` służącej do manipulowania ciągami tekstowymi,**
- **sposoby pracy z klasą STL `string`,**
- **sposoby, na jakie STL pomaga w łatwym łączeniu, dołączaniu, wyszukiwaniu i przeprowadzaniu innych operacji na ciągach tekstowych,**
- **opartą na wzorcach implementację klasy STL `string`.**

Dlaczego potrzebna jest klasa służąca do manipulowania ciągami tekstowymi?

W języku C++ ciąg tekstowy (ang. *string*) to tablica znaków. Jak dowiedziałeś się w lekcji 4., najprostszą tablicę znaków można zdefiniować w następujący sposób:

```
char staticName [20];
```

Powyższa instrukcja jest deklaracją tablicy znaków (czyli *ciągu tekstowego*) składającej się ze stałej (stąd statycznej) liczby dwudziestu elementów. Jak widać, ten bufor może przechowywać *ciąg tekstowy* o określonej długości. Jeżeli nastąpi próba przechowania w nim większej liczby znaków, dojdzie do przepełnienia. Zmiana wielkości tej statycznie zaalokowanej tablicy jest niemożliwa. Aby pokonać to ograniczenie, język C++ dostarcza mechanizm dynamicznej alokacji danych. Dlatego też znacznie bardziej dynamiczna reprezentacja ciągu tekstowego tablicy ma postać:

```
char* dynamicName = new char [ArrayLength];
```

W ten sposób powstaje dynamicznie zaalokowana tablica znaków, która może być zainicjalizowana do długości wskazanej w `ArrayLength` i określanej w trakcie działania programu. Oznacza to, że może być zaalokowana do przechowywania danych o zmiennej wielkości. Jeżeli jednak w trakcie działania programu chcesz zmienić wielkość tablicy, na początek powinieneś zwolnić zaalokowaną wcześniej pamięć, a dopiero później zaalokować pamięć do przechowywania żądanych danych.

Wszystko komplikuje się jeszcze bardziej, jeżeli ciągi tekstowe implementowane jako tablice znaków są obecne w postaci atrybutów składowych klasy. W sytuacji, kiedy obiekt tej klasy będzie kopiowany do innego, z powodu braku doskonale zaprojektowanego konstruktora kopiującego i operatora przypisania prawdopodobnie zaistnieje sytuacja, w której obiekty klasy po skopiowaniu będą wskazywały adresy tego samego ciągu tekstowego. Kiedy w dwóch obiektach dwa wskaźniki ciągów tekstowych prowadzą do tego samego miejsca w pamięci, zniszczenie obiektu źródłowego spowoduje, że wskaźnik w obiekcie docelowym będzie nieprawidłowy. Taka sytuacja może doprowadzić do awarii aplikacji.

Użycie klasy ciągu tekstowego rozwiązuje te problemy. Klasy ciągu tekstowego w bibliotece STL, takie jak `std::string` (przeznaczona do obsługi zwykłych znaków) lub `std::wstring` (przeznaczona do obsługi znaków większych niż 8-bitowe), pomagają programiście następująco:

- zmniejszają wysiłek związany z tworzeniem ciągów tekstowych i manipulowaniem nimi;
- zwiększają stabilność aplikacji zaprojektowanej do wewnętrznego zarządzania szczegółami dotyczącymi alokacji pamięci;
- dostarczają konstruktor kopiujący i operator przypisania, co gwarantuje, że składowe ciągi tekstowe będą prawidłowo kopiowane;
- zapewniają użyteczne funkcje narzędziowe pomagające m.in. w kopiowaniu, skracaniu, wyszukiwaniu i usuwaniu ciągów tekstowych;
- dostarczają operatory pomagające podczas operacji porównywania ciągów tekstowych;
- pozwalają programiście skoncentrować wysiłki na podstawowych zadaniach aplikacji, nie musi zajmować się szczegółami dotyczącymi operacji na ciągach tekstowych.

Uwaga

Zarówno `std::string`, jak i `std::wstring` to w rzeczywistości specjalizacje wzorca tej samej klasy `std::basic_string<T>` dla typów (odpowiednio) `char` i `wchar_t`. Kiedy poznasz jedną z wymienionych klas, te same metody i operatory będziesz mógł używać w drugiej.

Wkrótce na przykładzie `std::string` poznasz niektóre użyteczne funkcje pomocnicze dostarczane przez klasę `string` biblioteki STL.

Praca z klasą STL string

Najczęściej używane funkcje ciągu tekstowego to:

- kopiowanie,
- konkatenacja (łączenie),
- wyszukiwanie znaków i podciągów tekstowych,
- skracanie,
- odwracanie zawartości ciągu tekstowego i zmiana wielkości znaków za pomocą algorytmów dostarczanych przez bibliotekę standardową.

Aby używać klasy STL `string`, trzeba dołączyć nagłówek `<string>`.

Ustanawianie obiektu STL string i tworzenie kopii

Klasa `string` zawiera wiele przeciążonych konstruktorów i dlatego może być ustanawiana i inicjalizowana na wiele różnych sposobów. Przykładowo prosta inicjalizacja i przypisanie ciągu tekstowego stałej znakowej do zwykłego obiektu STL przeprowadza się następująco:

```
const char* constCStyleString = "Witaj, ciągu tekstowy!";
std::string strFromConst (constCStyleString);
```

lub

```
std::string strFromConst = constCStyleString;
```

Ten wcześniejszy zapis jest podobny do poniższego:

```
std::string str2 ("Witaj, ciągu tekstowy!");
```

Już oczywiste jest, że ustanowienie obiektu `string` i jego inicjalizacja wraz z wartością nie wymagają podawania wielkości ciągu tekstowego ani innych informacji związanych z alokacją pamięci — konstruktor klasy STL `string` zajmuje się tym automatycznie.

Podobnie możliwe jest użycie jednego obiektu `string` w celu inicjalizacji innego:

```
std::string str2Copy (str2);
```

Można nawet poinstruować konstruktor obiektu `string`, aby akceptował tylko pierwsze *n* znaków z dostarczonego mu ciągu tekstowego:

```
// Inicjalizacja ciągu tekstowego z pierwszymi pięcioma znakami innego ciągu tekstowego.
std::string strPartialCopy (constCStyleString, 5);
```

Istnieje także możliwość zainicjalizowania obiektu `string` wraz z podaną liczbą egzemplarzy wskazanego znaku:

```
// Inicjalizacja obiektu string wraz z dziesięcioma znakami 'a'.
std::string strRepeatChars (10, 'a');
```

Techniki ustanawiania i kopiowania w klasie STL `string` zostały przedstawione w listingu 16.1.

Listing 16.1. Techniki ustanawiania i kopiowania obiektów klasy STL string

```
0: #include <string>
1: #include <iostream>
2:
```

```
 3: int main ()
 4: {
 5:    using namespace std;
 6:    const char* constCStyleString = "Witaj, ciągu tekstowy!";
 7:    cout << "Stała ciągu tekstowego ma wartość: " << constCStyleString <<
       ↪endl;
 8:
 9:    std::string strFromConst (constCStyleString); // Konstruktor.
10:    cout << "strFromConst: " << strFromConst << endl;
11:
12:    std::string str2("Witaj, ciągu tekstowy!");
13:    std::string str2Copy (str2);
14:    cout << "str2Copy: " << str2Copy << endl;
15:
16:    // Inicjalizacja ciągu tekstowego z pięcioma pierwszymi znakami innego.
17:    std::string strPartialCopy (constCStyleString, 5);
18:    cout << "strPartialCopy: " << strPartialCopy << endl;
19:
20:    // Inicjalizacja obiektu string wraz z dziesięcioma znakami 'a'.
21:    std::string strRepeatChars (10, 'a');
22:    cout << "strRepeatChars: " << strRepeatChars << endl;
23:
24:    return 0;
25: }
```

Wynik ▼

```
Stała ciągu tekstowego ma wartość: Witaj, ciągu tekstowy!
strFromConst: Witaj, ciągu tekstowy!
str2Copy: Witaj, ciągu tekstowy!
strPartialCopy: Witaj
strRepeatChars: aaaaaaaaaa
```

Analiza ▼

W przedstawionym powyżej fragmencie kodu pokazano techniki ustanawiania obiektu STL `string` i jego inicjalizacji wraz z innym ciągiem tekstowym. Istnieje więc możliwość skopiowania fragmentu innego ciągu tekstowego i utworzenia jego częściowej kopii lub inicjalizacji wraz z ustaloną liczbą powtarzających się znaków. W wierszu 6. został zainicjalizowany `constCStyleString`, czyli ciąg tekstowy znaków w stylu języka C zawierający przykładową wartość.
W wierszu 9. widzimy, jak łatwo za pomocą `std::string` utworzyć kopię, używając do tego konstruktora. W wierszu 12. następuje skopiowanie innego ciągu tekstowego do obiektu `std::string` o nazwie `str2`. Natomiast

w wierszu 13. zademonstrowano, że obiekt `std::string` ma inny przeciążony konstruktor pozwalający na skopiowanie obiektu `std::string` i otrzymanie obiektu `str2Copy`. W wierszu 17. pokazano uzyskanie kopii częściowej, w wierszu 21. obiekt `std::string` został ustanowiony i zainicjalizowany wraz z ustaloną liczbą powtarzających się znaków. Powyższy fragment kodu stanowi jedynie małą demonstrację tego, jak obiekt `std::string` i jego liczne konstruktory kopiujące ułatwiają programiście tworzenie ciągów tekstowych, kopiowanie ich i wyświetlanie.

Uwaga

Warto zwrócić uwagę, że jeżeli używasz ciągów tekstowych w stylu języka C do utworzenia kopii tego samego typu, odpowiednikiem kodu w wierszu 9. w listingu 16.1 będzie następujący fragment:

```
const char* constCStyleString = "Witaj, świecie!";

// W celu utworzenia kopii najpierw trzeba zaalokować pamięć…
char * pszCopy = new char [strlen (constCStyleString) + 1];
strcpy (pszCopy, constCStyleString);  // Właściwy krok kopiowania.

// Zwolnienie pamięci po użyciu pszCopy.
delete[] pszCopy;
```

Jak widzisz, ta wersja wymaga większej liczby wierszy kodu, zwiększa niebezpieczeństwo powstania błędu, a ponadto spycha na programistę kwestię zarządzania pamięcią i jej zwalniania. Klasa STL `string` zajmuje się tym wszystkim za programistę i oferuje jeszcze znacznie więcej możliwości!

Uzyskanie dostępu do obiektu string i jego zawartości

Dostęp do zawartości znakowej obiektu STL `string` może odbywać się z wykorzystaniem iteratorów lub za pomocą składni podobnej do tablicy z podaną wartością przesunięcia, przy użyciu operatora indeksowania `[]`. Reprezentację obiektu `string` w stylu języka C można pobrać za pomocą funkcji składowej `c_str()`. Przykład przedstawiono w listingu 16.2.

Listing 16.2. Dwa sposoby uzyskania dostępu do elementów znakowych obiektu STL string: operator indeksowania i iteratory

```
0: #include <string>
1: #include <iostream>
2:
3: int main ()
```

```
 4: {
 5:    using namespace std;
 6:
 7:     // Przykładowy ciąg tekstowy.
 8:    string strSTLString ("Witaj ciągu tekstowy");
 9:
10:     // Wyświetlenie zawartości ciągu tekstowego za pomocą składni tablicy.
11:    cout << "Wyświetlenie znaków przy użyciu składni tablicy: " << endl;
12:    for ( size_t nCharCounter = 0
13:        ; nCharCounter < strSTLString.length ()
14:        ; ++ nCharCounter )
15:     {
16:        cout << "Znak [" << nCharCounter << "] to: ";
17:        cout << strSTLString [nCharCounter] << endl;
18:     }
19:    cout << endl;
20:
21:     // Dostęp do zawartości ciągu tekstowego za pomocą iteratorów.
22:    cout << "Wyświetlenie znaków przy użyciu iteratorów: " << endl;
23:    int charOffset = 0;
24:    string::const_iterator iCharacterLocator;
25:    for ( iCharacterLocator = strSTLString.begin ()
26:        ; iCharacterLocator != strSTLString.end ()
27:        ; ++ iCharacterLocator )
28:    {
29:        cout << "Znak [" << charOffset ++ << "] to: ";
30:        cout << *iCharacterLocator << endl;
31:    }
32:    cout << endl;
33:
34:    // Dostęp do zawartości ciągu tekstowego jak do ciągu tekstowego w stylu języka C.
35:    cout << "Reprezentacja char* ciągu tekstowego to: ";
36:    cout << strSTLString.c_str () << endl;
37:
38:    return 0;
39: }
```

Wynik ▼

```
Wyświetlenie znaków przy użyciu składni tablicy:
Znak [0] to: W
Znak [1] to: i
Znak [2] to: t
Znak [3] to: a
Znak [4] to: j
Znak [5] to:
Znak [6] to: c
Znak [7] to: i
```

```
Znak [8] to: ą
Znak [9] to: g
Znak [10] to: u
Znak [11] to:
Znak [12] to: t
Znak [13] to: e
Znak [14] to: k
Znak [15] to: s
Znak [16] to: t
Znak [17] to: o
Znak [18] to: w
Znak [19] to: y

Wyświetlenie znaków przy użyciu iteratorów:
Znak [0] to: W
Znak [1] to: i
Znak [2] to: t
Znak [3] to: a
Znak [4] to: j
Znak [5] to:
Znak [6] to: c
Znak [7] to: i
Znak [8] to: ą
Znak [9] to: g
Znak [10] to: u
Znak [11] to:
Znak [12] to: t
Znak [13] to: e
Znak [14] to: k
Znak [15] to: s
Znak [16] to: t
Znak [17] to: o
Znak [18] to: w
Znak [19] to: y

Reprezentacja char* ciągu tekstowego to: Witaj ciągu tekstowy
```

Analiza ▼

W powyższym kodzie przedstawiono kilka sposobów uzyskania dostępu do zawartości obiektu `string`. Iteratory są ważne, ponieważ wiele funkcji składowych obiektu `string` zwraca wyniki właśnie w tej postaci. W wierszach od 12. do 18. znaki obiektu `string` są wyświetlane przy użyciu składni podobnej do tablicy poprzez operator indeksowania `[]`, zaimplementowany przez klasę `std::string`. Warto zwrócić uwagę na fakt, że operator ten wymaga podania wartości przesunięcia, jak to zostało pokazane w wierszu 17.

Dlatego też ważne jest, aby nie wykraczać poza zakres obiektu `string`, tzn. nie odczytywać znaków, dla których wartość przesunięcia wskazuje położenie znajdujące się poza obiektem `string`. W liniach od 25. do 31. również wyświetlono zawartość obiektu `string` znak po znaku, ale tym razem za pomocą iteratorów.

Łączenie ciągów tekstowych

Łączenie ciągów tekstowych można przeprowadzić, używając albo operatora `+=`, albo funkcji składowej `append()`.

```
string strSample1 ("Witaj,");
string strSample2 (" ciągu tekstowy!");
strSample1 += strSample2;     // Użycie std::string::operator +=.
// Alternatywne rozwiązanie to użycie std::string::append().
strSample1.append (strSample2);   // (Przeciążony również dla char*).
```

Przykłady obu wariantów pokazano w listingu 16.3.

Listing 16.3. Łączenie ciągów tekstowych przy użyciu operatora dodawania/przypisania (+=) lub metody append() obiektu STL string

```
 0: #include <string>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:
 7:    string strSample1 ("Witaj,");
 8:    string strSample2 (" ciągu tekstowy!");
 9:
10:    // Łączenie ciągów tekstowych.
11:    strSample1 += strSample2;
12:    cout << strSample1 << endl << endl;
13:
14:    string strSample3 (" Dobrze, że nie trzeba używać wskaźników!");
15:    strSample1.append (strSample3);
16:    cout << strSample1 << endl << endl;
17:
18:    const char* constCStyleString = " Jednak nadal możesz je stosować!";
19:    strSample1.append (constCStyleString);
20:    cout << strSample1 << endl;
21:
22:    return 0;
23: }
```

Wynik ▼

```
Witaj, ciągu tekstowy!
Witaj, ciągu tekstowy! Dobrze, że nie trzeba używać wskaźników!
Witaj, ciągu tekstowy! Dobrze, że nie trzeba używać wskaźników! Jednak
➥nadal możesz je stosować!
```

Analiza ▼

W wierszach 11., 15. oraz 19. listingu przedstawiono różne metody łączenia ciągów tekstowych w STL. Warto zwrócić uwagę na użycie operatora += oraz na możliwości funkcji `append()` — akceptuje inny obiekt `string` (jak pokazano w wierszu 11.) oraz ciąg tekstowy znaków w stylu języka C.

Wyszukiwanie znaku bądź podciągu tekstowego w obiekcie string

Obiekt STL `string` zawiera funkcję składową `find()` wraz z kilkoma przeciążonymi wersjami, która pomaga w wyszukiwaniu znaku bądź podciągu tekstowego w podanym obiekcie `string`.

```
// Wyszukanie podciągu tekstowego "dzień" w ciągu tekstowym strSample, wyszukiwanie
// rozpoczyna się w położeniu 0.
size_t charPos = strSample.find ("dzień", 0);
// Sprawdzenie, czy znaleziono podciąg tekstowy, porównanie względem string::npos.
if (charPos != string::npos)
   cout << "Pierwsze wystąpienie słowa \"dzień\" zostało znalezione
   ➥w położeniu " << charPos;
else
   cout << "Nie znaleziono podciągu tekstowego." << endl;
```

Sposób wykorzystania funkcji składowej `std::string::find` został przedstawiony w listingu 16.4.

Listing 16.4. Użycie funkcji string::find do wyszukania podciągu tekstowego lub znaku

```
 0: #include <string>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:
 7:     string strSample ("dzień dobry ciągu tekstowy! mamy dzisiaj piękny
        ➥dzień!");
```

```
 8:     cout << "Przykładowy ciąg tekstowy to: " << endl;
 9:     cout << strSample << endl << endl;
10:
11:     // Wyszukiwanie podciągu tekstowego "dzień" w tym ciągu tekstowym…
12:     size_t charPos = strSample.find ("dzień", 0);
13:
14:     // Sprawdzenie, czy podciąg tekstowy został znaleziony…
15:     if (charPos != string::npos)
16:         cout << "Pierwsze wystąpienie słowa \"dzień\" zostało
        ↪znalezione w położeniu " << charPos;
17:     else
18:         cout << "Nie znaleziono podciągu tekstowego." << endl;
19:
20:     cout << endl << endl;
21:
22:     cout << "Zlokalizowanie wszystkich wystąpień ciągu tekstowego
        ↪\"dzień\"" << endl;
23:     size_t SubstringPos = strSample.find ("dzień", 0);
24:
25:     while (SubstringPos != string::npos)
26:     {
27:         cout << "słowo \"dzień\" znalezione w położeniu " <<
            ↪SubstringPos << endl;
28:
29:         // Funkcja 'find' wyszukuje dalej, począwszy od kolejnego znaku.
30:         size_t nSearchPosition = SubstringPos + 1;
31:
32:         SubstringPos = strSample.find ("dzień", nSearchPosition);
33:     }
34:
35:     cout << endl;
36:
37:     cout << "Zlokalizowanie wszystkich wystąpień znaku 'a'" << endl;
38:     const char charToSearch = 'a';
39:     charPos = strSample.find (charToSearch, 0);
40:
41:     while (charPos != string::npos)
42:     {
43:         cout << "'" << charToSearch << "' znaleziono";
44:        cout << " w położeniu " << charPos << endl;
45:
46:         // Funkcja 'find' wyszukuje dalej, począwszy od kolejnego znaku.
47:         size_t charSearchPos = charPos + 1;
48:
49:     charPos = strSample.find(charToSearch, charSearchPos);
50:     }
51:
52:     return 0;
53: }
```

Wynik ▼

```
Przykładowy ciąg tekstowy to:
dzień dobry ciągu tekstowy! mamy dzisiaj piękny dzień!

Pierwsze wystąpienie słowa "dzień" zostało znalezione w położeniu 0

Zlokalizowanie wszystkich wystąpień ciągu tekstowego "dzień"
słowo "dzień" znalezione w położeniu 0
słowo "dzień" znalezione w położeniu 48

Zlokalizowanie wszystkich wystąpień znaku 'a'
'a' znaleziono w położeniu: 14
'a' znaleziono w położeniu: 29
'a' znaleziono w położeniu: 38
```

Analiza ▼

W wierszach od 12. do 18. pokazano najprostszy sposób użycia funkcji `find()` w celu upewnienia się, czy określony podciąg tekstowy znajduje się w obiekcie `string`. W tym celu wynik operacji `find()` jest porównywany z `std::string::npos` (tzn. w rzeczywistości –1) i wskazuje, że szukany element nie został znaleziony. Kiedy funkcja `find()` nie zwraca wartości `npos`, wówczas zwraca wartość przesunięcia wskazującą położenie podciągu tekstowego lub znaku w obiekcie `string`.

W powyższym kodzie pokazano użycie funkcji `find()` w pętli `while` w celu zlokalizowania wszystkich wystąpień znaku bądź podciągu tekstowego w obiekcie STL `string`. Użyta tutaj przeciążona wersja funkcji `find()` przyjmuje dwa parametry: wyszukiwany znak lub podciąg tekstowy oraz wartość przesunięcia wskazującą punkt, od którego funkcja `find()` ma rozpocząć wyszukiwanie.

Uwaga

> Obiekt STL `string` zawiera również inne funkcje, podobne w działaniu do `find()`, m.in. `find_first_of()`, `find_first_not_of()`, `find_last_of()` oraz `find_last_not_of()`, które służą programiście pomocą w innych zadaniach.

Skracanie obiektu STL string

Obiekt STL `string` zawiera funkcję o nazwie `erase()`, za pomocą której można usunąć:

- określoną liczbę znaków po podaniu położenia i liczby znaków do usunięcia

```
string strSample ("Witaj, ciągu tekstowy! Dzisiaj mamy wspaniały
↪dzień!");
strSample.erase (22, 30);  // Witaj, ciągu tekstowy!
```

- znak po jego podaniu wraz z iteratorem prowadzącym do tego znaku

```
strSample.erase (iCharS); // Iterator wskazuje określony znak.
```

- określony zakres znaków po podaniu dwóch iteratorów

```
strSample.erase (strSample.begin (), strSample.end ()); // Usunięcie
                                                        // wszystkich znaków.
```

W przedstawionym poniżej listingu 16.5 zademonstrowano różne wersje przeciążonej funkcji `string::erase()`.

Listing 16.5. Użycie funkcji składowej erase() do skracania ciągu tekstowego, począwszy od wskazanego położenia lub iteratora

```
 0: #include <string>
 1: #include <algorithm>
 2: #include <iostream>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:
 8:     string strSample ("Witaj ciągu tekstowy! Obudź się, piękny dzień
        ↪mamy!");
 9:     cout << "Początkowa postać przykładowego ciągu tekstowego: " <<
        ↪endl;
10:     cout << strSample << endl << endl;
11:
12:     // Usunięcie znaków z ciągu tekstowego na podstawie podanego położenia i liczby
        ↪znaków.
13:     cout << "Usunięcie drugiego zdania: " << endl;
14:     strSample.erase (21, 30);
15:     cout << strSample << endl << endl;
16:
17:     // Wyszukanie znaku 'c' w ciągu tekstowym przy użyciu algorytmu STL find.
18:     string::iterator iCharC = find ( strSample.begin ()
19:                                    , strSample.end (), 'c');
20:
21:     // Jeżeli znak zostanie znaleziony, funkcja erase() spowoduje usunięcie znaku.
22:     cout << "Usunięcie znaku 'c' z przykładowego ciągu tekstowego: " <<
        ↪endl;
23:     if (iCharC != strSample.end ())
24:         strSample.erase (iCharC);
25:
```

```
26:     cout << strSample << endl << endl;
27:
28:     // Usunięcie zakresu znaków za pomocą przeciążonej wersji funkcji erase().
29:     cout << "Usunięcie zakresu znaków od begin() do end(): " << endl;
30:     strSample.erase (strSample.begin (), strSample.end ());
31:
32:    // Sprawdzenie długości ciągu po wykonaniu powyższej funkcji erase().
33:     if (strSample.length () == 0)
34:         cout << "Ciąg tekstowy jest pusty" << endl;
35:
36:     return 0;
37: }
```

Wynik ▼

```
Początkowa postać przykładowego ciągu tekstowego:
Witaj ciągu tekstowy! Obudź się, piękny dzień mamy!

Usunięcie drugiego zdania
Witaj ciągu tekstowy!

Usunięcie znaku 'c' z przykładowego ciągu tekstowego
Witaj iągu tekstowy!

Usunięcie zakresu znaków od begin() do end():
Ciąg tekstowy jest pusty
```

Analiza ▼

W powyższym listingu pokazano użycie trzech wersji funkcji `erase()`. Pierwsza, przedstawiona w wierszu 14., powoduje usunięcie zbioru znaków po podaniu jej punktu początkowego i liczby znaków do usunięcia. Druga, przedstawiona w wierszu 24., powoduje usunięcie określonego znaku na podstawie iteratora prowadzącego do tego znaku. Trzecia i ostatnia, przedstawiona w wierszu 30., powoduje usunięcie zakresu znaków na podstawie iteratorów będących granicami usuwanego zakresu. Ponieważ granice tego zakresu zdefiniowane przez funkcje składowe `begin()` i `end()` obiektu `string` obejmują cały ciąg tekstowy, wywołanie funkcji `erase()` względem tego ciągu tekstowego powoduje usunięcie go. Warto zwrócić uwagę, że klasa `string` dostarcza także funkcję `clear()`, której zadaniem jest usunięcie wewnętrznego bufora i wyzerowanie obiektu `string`.

C++11

Uproszczenie deklaracji iteratora przy użyciu słowa kluczowego auto

Standard C++11 pomaga w uproszczeniu deklaracji iteratora zadeklarowanego w listingu 16.5 w następujący sposób:

```
18:     string::iterator iCharC = find ( strSample.begin ()
19:                                    , strSample.end (), 'c');
```

Aby uprościć deklarację, wprowadzono słowo kluczowe `auto`, z którym spotkałeś się już w lekcji 3., zatytułowanej „Zmienne i stałe”:

```
auto iCharC = find ( strSample.begin ()
                   , strSample.end (), 'c');
```

Kompilator automatycznie określi typ zmiennej `iCharC` na podstawie pobranych z `std::find` informacji o typie wartości zwrotnej.

Odwracanie zawartości ciągu tekstowego

Czasami ważne jest, aby odwrócić zawartość ciągu tekstowego. Powiedźmy, że zadaniem jest określenie, kiedy ciąg tekstowy wprowadzony przez użytkownika będzie palindromem. Jednym z rozwiązań jest odwrócenie kopii tego samego ciągu tekstowego, a następnie porównanie obu. Ciągi tekstowe STL można bardzo łatwo odwrócić za pomocą algorytmu `std::reverse`:

```
string strSample ("Witaj, ciągu tekstowy! Za chwilę zostaniesz odwrócony!");
reverse (strSample.begin (), strSample.end ());
```

Użycie algorytmu `std::reverse` zostało przedstawione w listingu 16.6.

Listing 16.6. Odwracanie ciągu tekstowego znajdującego się w obiekcie STL string

```
 0: #include <string>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:
 8:    string strSample ("Witaj, ciągu tekstowy! Zostaniesz odwrócony!");
 9:    cout << "Początkowa postać przykładowego ciągu tekstowego: " << endl;
```

```
10:    cout << strSample << endl << endl;
11:
12:    reverse (strSample.begin (), strSample.end ());
13:
14:    cout << "Po zastosowaniu algorytmu std::reverse: " << endl;
15:    cout << strSample << endl;
16:
17:    return 0;
18: }
```

Wynik ▼

```
Początkowa postać przykładowego ciągu tekstowego:
Witaj, ciągu tekstowy! Zostaniesz odwrócony!

Po zastosowaniu algorytmu std::reverse:
!ynocórwdo zseinatsoZ !ywotsket ugąic ,jatiW
```

Analiza ▼

Zastosowany w wierszu 12. algorytm `std::reverse` działa na ogranicznikach kontenera dostarczonych przez dwa parametry danych wejściowych. W omawianym przykładzie tymi ogranicznikami jest początek i koniec obiektu `string`, a więc następuje odwrócenie zawartości całego ciągu tekstowego. Istnieje także możliwość odwracania ciągu tekstowego we fragmentach poprzez określenie odpowiednich ograniczników jako parametrów wejściowych. Warto pamiętać, że ograniczniki nigdy nie powinny przekraczać wartości `end()`.

Konwersja wielkości znaków obiektu string

Konwersję wielkości znaków ciągu tekstowego można przeprowadzić za pomocą algorytmu `std::transform`, który względem każdego elementu zbioru stosuje funkcję zdefiniowaną przez użytkownika. W omawianym przykładzie zbiorem jest sam obiekt `string`. W przykładzie przedstawionym w listingu 16.7 pokazano, jak można zmienić wielkość znaków w obiekcie `string`.

Listing 16.7. Konwersja wielkości znaków w obiekcie STL string przeprowadzona za pomocą algorytmu std::transform

```
0: #include <string>
1: #include <iostream>
2: #include <algorithm>
3:
```

```
 4: int main ()
 5: {
 6:    using namespace std;
 7:
 8:    cout << "Proszę podać ciąg tekstowy, w którym nastąpi konwersja
       ↪wielkości znaków: " << endl;
 9:    cout << "> ";
10:
11:    string strInput;
12:    getline (cin, strInput);
13:    cout << endl;
14:
15:    transform (strInput.begin(),strInput.end(),strInput.begin(),toupper);
16:    cout << "Ciąg tekstowy, w którym znaki skonwertowano na wielkie: " <<
       ↪endl;
17:    cout << strInput << endl << endl;
18:
19:    transform (strInput.begin(),strInput.end(),strInput.begin(),tolower);
20:    cout << "Ciąg tekstowy, w którym znaki skonwertowano na małe: " <<
       ↪endl;
21:    cout << strInput << endl << endl;
22:
23:    return 0;
24: }
```

Wynik ▼

```
Proszą podać ciąg tekstowy, w którym nastąpi konwersja wielkości znaków:
> SkonwertuJ tEN ciĄG TekSTOwy!

Ciąg tekstowy, w którym znaki skonwertowano na wielkie:
SKONWERTUJ TEN CIĄG TEKSTOWY!

Ciąg tekstowy, w którym znaki skonwertowano na małe:
skonwertuj ten ciąg tekstowy!
```

Analiza ▼

W wierszach 15. i 19. pokazano, jak można efektywnie wykorzystać funkcję `std::transform` do zmiany wielkości znaków w ciągu tekstowym obiektu STL `string`.

Implementacja klasy STL string oparta na wzorcach

Klasa `std::string`, którą właśnie poznałeś, to w rzeczywistości specjalizacja wzorca klasy STL `std::basic_string<T>`. Deklaracja wzorca kontenera klasy `basic_string` jest następująca:

```
template<class _Elem,
   class _Traits,
   class _Ax>
   class basic_string
```

W powyższej definicji wzorca parametrem pierwszorzędnej wagi jest pierwszy: `_Elem`. Jest to typ określony przez obiekt `basic_string`. Dlatego też klasa `str::string` jest specjalizacją wzorca `basic_string` dla `_Elem=char`, podczas gdy `wstring` jest specjalizacją wzorca `basic_string` dla `_Elem=wchar_t`.

Innymi słowy, klasa STL `string` jest zdefiniowana jako:

```
typedef basic_string<char, char_traits<char>, allocator<char> >
   string;
```

natomiast klasa STL `wstring` jest zdefiniowana jako:

```
typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t> >
   string;
```

Tak więc wszystkie przedstawione dotąd możliwości i funkcje ciągu tekstowego są dostarczane przez `basic_string` i można je zastosować również w klasie STL `wstring`.

Wskazówka

Klasy `std::wstring` możesz używać w aplikacjach, w których konieczne jest zapewnienie lepszej obsługi znaków innych niż łacińskie, np. japońskich lub chińskich.

Podsumowanie

W tej lekcji dowiedziałeś się, że klasa STL `string` jest kontenerem dostarczanym przez standardową bibliotekę wzorców i pomaga programiście w przeprowadzaniu wielu operacji na ciągach tekstowych. Zaleta stosowania tej klasy polega na tym, że zadania obsługi zarządzania pamięcią, porównywania ciągów tekstowych i funkcje manipulowania na ciągach tekstowych są realizowane przy użyciu kontenera klasy dostarczanego przez strukturę STL.

Pytania i odpowiedzi

Pytanie: Muszę odwrócić zawartość ciągu tekstowego za pomocą funkcji `std::reverse`. Jaki nagłówek mam umieścić w programie, aby można było używać wymienionej funkcji?

Odpowiedź: Aby funkcja `std::reverse` była dostępna, należy w programie umieścić nagłówek `<algorithm>`.

Pytanie: Jaką rolę odgrywa `std::transform` podczas konwersji znaków ciągu tekstowego na małe za pomocą funkcji `tolower()`?

Odpowiedź: `std::transform` wywołuje funkcję `tolower()` dla znaków w obiekcie `string`, które znajdują się w zakresie dostarczonym funkcji przekształcającej znaki.

Pytanie: Dlaczego `std::wstring` i `std::string` mają dokładnie takie samo zachowanie i funkcje składowe?

Odpowiedź: Wynika to z faktu, że są specjalizacjami wzorca klasy `std::basic_string`.

Pytanie: Czy podczas generowania wyników za pomocą operatora porównania < klasy STL `string` wielkość znaków ma znaczenie?

Odpowiedź: Podczas generowania wyników wielkość znaków ma znaczenie.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Co specjalizuje klasa wzorca STL `std::string`?
2. W jaki sposób przeprowadzisz porównanie dwóch ciągów tekstowych bez uwzględniania wielkości znaków?
3. Czy ciągi tekstowe STL oraz w stylu języka C są do siebie podobne?

Ćwiczenia

1. Utwórz program sprawdzający, czy wprowadzone przez użytkownika dane wejściowe są palindromem. Przykładowo `KAJAK` to palindrom, ponieważ po odwróceniu słowo nie ulega zmianie.
2. Utwórz program informujący użytkownika o liczbie samogłosek w zdaniu.
3. Skonwertuj każdy co drugi znak ciągu tekstowego na wielki.
4. Twój program powinien mieć cztery obiekty ciągów tekstowych zainicjalizowanych z wartościami: „Uwielbiam”, „ciągi”, „tekstowe”, „STL”. Połącz je ze sobą, nie zapominając o spacjach między słowami, a następnie wyświetl całe zdanie.

Lekcja 17

Dynamiczne klasy tablic w STL

Tablice dynamiczne oferują programiście elastyczność w zakresie przechowywania danych bez konieczności dokładnego określenia wielkości tych danych już na etapie programowania aplikacji, jak ma to miejsce w przypadku tablic statycznych. Oczywiście, elastyczność taka bardzo często jest pożądana, a standardowa biblioteka wzorców (STL) dostarcza gotowe do zastosowania rozwiązanie w postaci klasy std::vector.

Z tej lekcji dowiesz się:

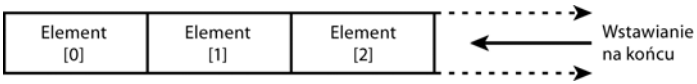
- **charakterystykę klasy std::vector,**
- **typowe operacje klasy vector,**
- **koncepcję wielkości i pojemności,**
- **klasę STL deque.**

Charakterystyka klasy std::vector

Klasa `vector` to wzorzec klasy, która oferuje ogólne funkcje tablicy dynamicznej i charakteryzuje się określonymi cechami. Oto one.

- Dodawanie elementów na końcu tablicy w niezmiennym czasie. Oznacza to, że czas potrzebny na wstawienie elementu na końcu tablicy nie zależy od wielkości tej tablicy. To samo dotyczy usuwania elementu z końca tablicy.
- Czas potrzebny na wstawienie lub usunięcie elementów ze środka tablicy jest wprost proporcjonalny do liczby elementów, które znajdują się po usuwanym elemencie.
- Liczba przechowywanych elementów jest zmienna, klasa `vector` zarządza użyciem pamięci.

Klasa `vector` to tablica dynamiczna, którą graficznie można przedstawić tak, jak pokazano na rysunku 17.1.



RYSUNEK 17.1. Wewnętrzna budowa klasy vector

Wskazówka

W celu użycia klasy `std::vector` w programie trzeba umieścić następujący nagłówek:

```
#include <vector>
```

Typowe operacje klasy vector

Specyfika zachowania i publiczne elementy składowe klasy `std::vector` zostały zdefiniowane w standardzie C++. Wskutek tego przedstawione w tej lekcji operacje na klasie `vector` są obsługiwane przez różne platformy programistyczne C++, które są zgodne ze standardem.

Ustanawianie klasy vector

Klasa `vector` to wzorzec klasy, który musi być ustanowiony zgodnie z przedstawionymi w lekcji 14. technikami ustanawiania wzorców.

Podczas procesu ustanawiania klasy `vector` trzeba określić typ obiektów, które będą *umieszczane* w tej tablicy dynamicznej.

```
std::vector<int> vecDynamicIntegerArray;  // Obiekt vector zawierający liczby całkowite.
std::vector<float> vecDynamicFloatArray;  // Obiekt vector zawierający liczby
                                          // zmiennoprzecinkowe.
std::vector<Tuna> vecDynamicTunaArray;  // Obiekt vector zawierający obiekty typu Tuna.
```

W celu zadeklarowania iteratora wskazującego element listy należy użyć polecenia, takiego jak poniższe:

```
std::list<int>::const_iterator iElementInSet;
```

Jeżeli potrzebny jest iterator, który będzie mógł być używany do modyfikacji wartości lub wywoływania funkcji innych niż `const`, wtedy w powyższym poleceniu zamiast `const_iterator` należy wykorzystać `iterator`.

Ponieważ klasa `std::vector` zawiera kilka przeciążonych konstruktorów, masz możliwość utworzenia obiektu z określoną liczbą elementów początkowych oraz przypisanymi im wartościami początkowymi. Oczywiście, jeden obiekt `vector` można wykorzystać w części lub w całości do utworzenia innego obiektu `vector`.

W listingu 17.1 zaprezentowano kilka utworzonych obiektów `vector`.

Listing 17.1. Różne formy tworzenia egzemplarzy klasy std::vector — o podanej wielkości, wartościach początkowych oraz wraz z wartościami skopiowanymi z innego obiektu vector

```
 0: #include <vector>
 1:
 2: int main ()
 3: {
 4:    std::vector <int> vecIntegers;;
 5:
 6:    // Ustanowienie tablicy wraz z dziesięcioma elementami (tablicę będzie można powiększyć).
 7:    std::vector <int> vecWithTenElements (10);
 8:
 9:    // Ustanowienie tablicy wraz z dziesięcioma elementami, każdy zainicjalizowany
      ↪z wartością 90.
10:    std::vector <int> vecWithTenInitializedElements (10, 90);
11:
12:    // Ustanowienie tablicy i zainicjalizowanie jej zawartością innej tablicy.
13:    std::vector <int> vecArrayCopy (vecWithTenInitializedElements);
14:
15:    // Ustanowienie tablicy wraz z pięcioma elementami pochodzącymi z innej tablicy.
16:    std::vector<int> vecSomeElementsCopied(vecWithTenElements.cbegin()
```

```
17:                                     , vecWithTenElements.cbegin () + 5);
18:
19:    return 0;
20: }
```

Analiza ▼

W powyższym listingu przedstawiono specjalizację klasy `vector` dla typu `integer` (liczby całkowite), innymi słowy, ustanowiono obiekt `vector` liczb całkowitych. Ten obiekt `vector` o nazwie `vecIntegers` używa konstruktora domyślnego, który jest całkiem użyteczny, kiedy nie wiadomo, jaka ma być minimalna wielkość kontenera, czyli gdy nieznana jest ilość liczb całkowitych przechowywanych w tym kontenerze. Pokazana w wierszach 10. i 13. druga i trzecia forma ustanawiania klasy `vector` jest stosowana, gdy programista wie, że tablica będzie zawierała przynajmniej dziesięć elementów. Warto zwrócić uwagę, że nie powoduje to ograniczenia maksymalnej wielkości kontenera, a raczej ustawienie wielkości inicjalizowanej. Czwarta forma w wierszach 16. i 17. jest stosowana, gdy zawartość tablicy będzie ustanawiana na podstawie innej. Oznacza to możliwość utworzenia jednego obiektu `vector` jako kopii innego lub jego części. Ta konstrukcja działa również we wszystkich kontenerach STL. W ostatniej formie wykorzystano iteratory. Tablica `vecSomeElementsCopied` zawiera pięć elementów z tablicy `vecWithTenElements`.

Uwaga

Czwarta konstrukcja może działać tylko z obiektami tego samego typu. Dlatego też możesz ustanowić `vecArrayCopy` — tablicę obiektów liczb całkowitych — używając innej tablicy obiektów liczb całkowitych. Jeżeli jeden z nich będzie tablicą, powiedzmy, obiektów typu `float`, kod nie zostanie skompilowany.

Wskazówka

Czy podczas kompilacji programu z metodami `cbegin()` i `cend()` występują błędy?

Jeżeli powyższy listing zechcesz skompilować w kompilatorze niezgodnym ze standardem C++11, zamiast metod `cbegin()` i `cend()` musisz użyć metod (odpowiednio) `begin()` i `end()`.

Metody `cbegin()` i `cend()` są inne (i lepsze), ponieważ zwracają iterator. Jednak wymienione metody nie są obsługiwane przez starsze wersje kompilatorów.

Wstawianie elementów na końcu przy użyciu push_back()

Po ustanowieniu tablicy liczb całkowitych kolejnym zadaniem będzie wstawienie do niej elementów (liczb całkowitych). Wstawianie elementów w obiekcie `vector` następuje na końcu tablicy, a elementy są tam umieszczane za pomocą metody składowej o nazwie `push_back()`:

```
vector <int> vecIntegers;   // Deklaracja obiektu vector typu int.
// Wstawienie do obiektu vector przykładowych liczb całkowitych:
vecIntegers.push_back (50);
vecIntegers.push_back (1);
```

W listingu 17.2 zaprezentowano użycie metody `push_back()` w celu dynamicznego wstawiania elementów do obiektu klasy `std::vector`.

Listing 17.2. Wstawianie elementów w obiekcie vector za pomocą metody push_back()

```
 0: #include <iostream>
 1: #include <vector>
 2: using namespace std;
 3:
 4: int main ()
 5: {
 6:     vector <int> vecIntegers;
 7:
 8:     // Wstawienie do obiektu vector przykładowych liczb całkowitych.
 9:     vecIntegers.push_back (50);
10:     vecIntegers.push_back (1);
11:     vecIntegers.push_back (987);
12:     vecIntegers.push_back (1001);
13:
14:     cout << "Obiekt vector zawiera ";
15:     cout << vecIntegers.size () << " elementy";
16:
17:     return 0;
18: }
```

Wynik ▼

```
Obiekt vector zawiera 4 elementy
```

Analiza ▼

Przedstawiona w wierszach od 9. do 12. publiczna metoda składowa `push_back()` klasy `vector` powoduje wstawienie elementów na końcu tablicy dynamicznej. Zwróć uwagę na użycie funkcji `size()` zwracającej liczbę elementów przechowywanych w tablicy.

C++11

Listy inicjalizacyjne

Standard C++11 oferuje listy inicjalizacyjne w postaci klasy `std::initialize_list<>`, która, jeśli jest obsługiwana, pozwala na utworzenie oraz inicjalizację elementów wektora, tak jak w tablicy statycznej:

```
vector<int> vecIntegers = {50, 1, 987, 1001};
// Rozwiązanie alternatywne:
vector<int> vecMoreIntegers {50, 1, 987, 1001};
```

Użycie powyższej składni spowoduje skrócenie listingu 17.2 o trzy wiersze. Nie wykorzystaliśmy tej możliwości, ponieważ listy inicjalizacyjne nie są obsługiwane przez implementację `std::vector` w kompilatorze Microsoft Visual C++ 2010.

Wstawianie elementów w określonym położeniu przy użyciu metody insert()

Metoda `push_back()` służy do wstawiania elementów na końcu wektora. Co zrobić w sytuacji, gdy element chcesz wstawić w środku? Wiele kontenerów STL, w tym m.in. `std::vector`, oferuje kilka przeciążonych metod `insert()`.

W jednej z wersji metody masz możliwość wskazania położenia, w którym ma zostać umieszczony wstawiany element:

```
// Wstawienie elementu na początku.
vecIntegers.insert (vecIntegers.begin (), 25);
```

W innej wersji metody również możesz wskazać położenie, a także podać liczbę elementów wraz z wartościami, które mają być wstawione:

```
// Wstawienie na końcu dwóch liczb o wartości 45.
vecIntegers.insert (vecIntegers.end (), 2, 45);
```

Istnieje również możliwość wstawienia zawartości jednego wektora we wskazanym położeniu drugiego wektora:

```
// Inny wektor zawierający dwa elementy o wartości 30.
vector <int> vecAnother (2, 30);

// Wstawienie w położeniu [1] dwóch elementów pochodzących z innego wektora.
vecIntegers.insert (vecIntegers.begin () + 1,
   vecAnother.begin (), vecAnother.end ());
```

Iterator najczęściej zwracany przez metody `begin()` lub `end()` można wykorzystać do wskazania metodzie `insert()` położenia, w którym mają być wstawione nowe elementy.

Wskazówka

Zwróć uwagę, że wspomniany iterator może być również wartością zwrotną algorytmu STL, np. funkcji `std::find()` używanej do wyszukiwania elementu, a następnie wstawiania innego w danym położeniu (operacja wstawiania spowoduje przesunięcie znalezionego elementu).

Wymienione postacie metod `vector::insert()` zaprezentowano w listingu 17.3.

Listing 17.3. Używanie vector::insert w celu wstawiania elementów we wskazanym położeniu

```
 0: #include <vector>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: void DisplayVector(const vector<int>& vecInput)
 5: {
 6:     for (auto iElement = vecInput.cbegin() // auto i cbegin() dla C++11.
 7:         ; iElement != vecInput.cend() // cend() to nowość w C++11.
 8:         ; ++ iElement )
 9:        cout << *iElement << ' ';
10:
11:     cout << endl;
12: }
13:
14: int main ()
15: {
16:     // Ustanowienie obiektu vector wraz z czterema elementami, każdy zainicjalizowany
        ↪z wartością 90.
17:     vector <int> vecIntegers (4, 90);
18:
19:     cout << "Początkowa zawartość obiektu vector jest następująca: ";
20:     DisplayVector(vecIntegers);
21:
```

```
22:     // Wstawienie wartości 25 na początku.
23:     vecIntegers.insert (vecIntegers.begin (), 25);
24:
25:     // Wstawienie na końcu dwóch liczb o wartości 45.
26:     vecIntegers.insert (vecIntegers.end (), 2, 45);
27:
28:     cout << "Zawartość tablicy vector po wstawieniu elementów
        ➥na początku i końcu: ";
29:     DisplayVector(vecIntegers);
30:
31:     // Inny obiekt vector zawierający dwa elementy o wartości 30.
32:     vector <int> vecAnother (2, 30);
33:
34:     // Wstawienie w pozycji [1] dwóch elementów pochodzących z innego kontenera.
35:     vecIntegers.insert (vecIntegers.begin () + 1,
36:                         vecAnother.begin (), vecAnother.end ());
37:
38:     cout << "Zawartość obiektu vector po wstawieniu w środku tablicy ";
39:     cout << "elementów innego obiektu:" << endl;
40:     DisplayVector(vecIntegers);
41:
42:     return 0;
43: }
```

Wynik ▼

```
Początkowa zawartość obiektu vector jest następująca: 90 90 90 90
Zawartość tablicy vector po wstawieniu elementów na początku i końcu:
➥25 90 90 90 90 45 45
Zawartość obiektu vector po wstawieniu w środku tablicy elementów innego obiektu:
25 30 30 90 90 90 90 45 45
```

Analiza ▼

W powyższym kodzie zademonstrowano użyteczne możliwości funkcji `insert()`, pozwalającej na wstawienie wartości w środku kontenera. Procedura rozpoczyna się w wierszu 17., gdy obiekt `vector` zawiera cztery elementy, wszystkie zainicjalizowane z wartościami 90. Wykorzystując ten wektor jako punkt wyjścia, używamy różnych przeciążonych wersji funkcji `vector::insert()`. W wierszu 23. zostaje wstawiony element na początku. W wierszu 26. użyto przeciążonej wersji metody w celu wstawienia na końcu dwóch elementów o wartości 45. Natomiast w wierszu 35. pokazano wstawienie w środku wektora (w omawianym przypadku to pozycja 1) elementów pochodzących z innego wektora.

Wprawdzie metoda `vector::insert` zapewnia ogromną elastyczność, ale preferowanym sposobem dodawania elementów do wektora pozostaje metoda `push_back()`.

Warto zwrócić uwagę, że metoda `insert()` to prawdopodobnie najmniej efektywny sposób dodawania elementów do obiektu `vector` (dodawanie elementu w pozycji, która nie znajduje się na końcu sekwencji). Wynika to z faktu, że dodawanie elementów na początku lub w środku obiektu powoduje przesunięcie wszystkich kolejnych elementów do tyłu (po zrobieniu na końcu tablicy miejsca dla tych elementów). Dlatego też, w zależności od typu obiektów znajdujących się w sekwencji, koszt operacji przesunięcia może być znaczny, zwłaszcza ze względu na użycie konstruktora kopiującego i operatora przypisania. W omawianym prostym przykładzie obiekt `vector` zawiera obiekty typu `int` (liczby całkowite), których przesunięcie jest stosunkowo mało wymagającą operacją. W wielu innych sytuacjach, gdzie używana będzie klasa `vector`, operacja przesunięcia może być bardzo kosztowna.

Wskazówka

Jeżeli w kontenerze trzeba bardzo często wstawiać elementy w środku, wtedy idealnym rozwiązaniem jest użycie kontenera `std::list`, którego szczegółowe omówienie znajduje się w lekcji 18., zatytułowanej „Klasy STL list i forward_list".

Ostrzeżenie

Czy używasz starszej wersji kompilatora C++?

Przedstawiona w listingu 17.3 funkcja `DisplayVector()` korzysta z wprowadzonego w standardzie C++11 słowa kluczowego `auto` do zdefiniowania typu iteratora (patrz wiersz 6.). Aby ten i kolejne przykłady skompilować w starszej wersji kompilatora nieobsługującej standardu C++11, musisz słowo kluczowe `auto` zastąpić konkretnym typem, w omawianym przypadku to `vector<int>::const_iterator`.

Dlatego też metoda `DisplayVector()` dla starszej wersji kompilatora będzie miała następującą postać:

```
// Wersja dla starszych wersji kompilatorów C++.
void DisplayVector(const vector<int>& vecInput)
{
   for (vector<int>::const_iterator iElement = vecInput.begin()
      ; iElement != vecInput.end ()
      ; ++ iElement )
     cout << *iElement << ' ';

   cout << endl;
}
```

Uzyskanie dostępu do elementów obiektu vector przy użyciu semantyki tablicy

Dostęp do elementów obiektu `vector` można uzyskać, wykorzystując następujące metody: semantykę tablicy za pomocą operatora indeksowania `[]`, funkcję składową `at()` lub iteratory.

W listingu 17.1 pokazano, jak egzemplarz `vector` może być utworzony wraz z zainicjalizowanymi dziesięcioma elementami:

```
std::vector <int> vecArrayWithTenElements (10);
```

Dostęp do poszczególnych elementów i ustawienie ich wartości jest możliwe przy użyciu składni stosowanej w tablicach:

```
vecArrayWithTenElements[3] = 2011; // Ustawienie czwartego elementu.
```

W listingu 17.4 zademonstrowano uzyskanie dostępu do elementów obiektu `vector` za pomocą operatora indeksowania `[]`.

Listing 17.4. Uzyskanie dostępu do elementów obiektu vector za pomocą semantyki tablicy

```
 0: #include <iostream>
 1: #include <vector>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:     vector <int> vecIntegerArray;
 7:
 8:     // Wstawienie do obiektu vector przykładowych liczb całkowitych.
 9:     vecIntegerArray.push_back (50);
10:     vecIntegerArray.push_back (1);
11:     vecIntegerArray.push_back (987);
12:     vecIntegerArray.push_back (1001);
13:
14:     for (size_t Index = 0; Index < vecIntegerArray.size (); ++Index)
15:     {
16:         cout << "Element[" << Index << "] = " ;
17:         cout << vecIntegerArray[Index] << endl;
18:     }
19:
20:     // Zmiana trzeciej liczby całkowitej z 987 na 2011.
21:     vecIntegerArray[2] = 2011;
22:     cout << "Po zastąpieniu: " << endl;
23:     cout << "Element[2] = " << vecIntegerArray[2] << endl;
24:
```

```
25:     return 0;
26: }
```

Wynik ▼

```
Element[0] = 50
Element[1] = 1
Element[2] = 987
Element[3] = 1001
Po zastąpieniu:
Element[2] = 2011
```

Analiza ▼

W wierszach 17., 21. i 23. w tym przykładowym fragmencie kodu pokazano, że w celu uzyskania dostępu do wartości tablicy za pomocą operatora indeksowania [] obiekt `vector` może być używany w taki sam sposób jak tablica statyczna. Operator indeksowania akceptuje położenie danego elementu wyrażone za pomocą liczonego od zera indeksu, podobnie jak w tablicach statycznych. Zwróć uwagę na pętlę `for` w wierszu 15. gwarantującą, że indeks nie wykroczy poza granice wektora — w tym celu przeprowadzane jest porównanie względem `vector::size()`.

Ostrzeżenie

Dostęp do elementów obiektu `vector` za pomocą operatora indeksowania [] jest najeżony tymi samymi niebezpieczeństwami, z którymi mamy do czynienia podczas uzyskiwania dostępu do elementów tablicy. Oznacza to, że nie wolno wykraczać poza zakres kontenera. Jeżeli używasz operatora indeksowania [] w celu uzyskania dostępu do elementu obiektu `vector` i podasz położenie spoza kontenera, wynik takiej operacji będzie nieprzewidywalny (wszystko może się zdarzyć, nawet pogwałcenie zasad dostępu).

Bezpieczniejszym rozwiązaniem jest użycie funkcji składowej `at()`:

```
// Pobranie elementu znajdującego się w położeniu 2.
cout << vecIntegerArray.at (2);
// Wersja vector::at() kodu przedstawionego w wierszu 17. listingu 17.4:
cout << vecIntegerArray.at (Index);
```

Funkcja `at()` przeprowadza w trakcie działania programu sprawdzenie wielkości kontenera i powoduje zgłoszenie wyjątku, jeśli nastąpi wykroczenie poza jego zakres (tego absolutnie nie należy robić).

Warto zwrócić uwagę, że operator indeksowania [] pozostaje bezpieczny w użyciu, kiedy jest wykorzystywany w sposób zapewniający spójność zakresu, co pokazano w powyższym fragmencie kodu.

Uzyskanie dostępu do elementów obiektu vector przy użyciu semantyki wskaźnika

Dostęp do elementów obiektu `vector` można również uzyskać za pomocą semantyki przypominającej wskaźniki, korzystając w tym celu z iteratorów, co przedstawiono w listingu 17.5.

Listing 17.5. Uzyskanie dostępu do elementów obiektu vector za pomocą semantyki wskaźników (iteratorów)

```
#include <iostream>
#include <vector>

int main ()
{
    using namespace std;
    vector <int> vecIntegers;

    // Wstawienie do obiektu vector przykładowych liczb całkowitych:
    vecIntegers.push_back (50);
    vecIntegers.push_back (1);
    vecIntegers.push_back (987);
    vecIntegers.push_back (1001);

    // Uzyskanie dostępu do elementów obiektu za pomocą iteratorów.
    vector<int>::iterator iElementLocator = vecIntegers.begin();
    // Iterator zadeklarowany przy użyciu wprowadzonego w C++11 słowa kluczowego
    ➥auto (usuń znak komentarza na początku kolejnego wiersza).
    // auto iElementLocator = vecIntegers.begin();

    while (iElementLocator != vecIntegers.end ())
    {
        size_t Index = distance (vecIntegers.begin (),
                        iElementLocator);

        cout << "Element w położeniu ";
        cout << Index << " to: " << *iElementLocator << endl;

        // Przejście do kolejnego elementu.
        ++ iElementLocator;
    }

    return 0;
}
```

Wynik ▼

```
Element w położeniu 0 to: 50
Element w położeniu 1 to: 1
Element w położeniu 2 to: 987
Element w położeniu 3 to: 1001
```

Analiza ▼

Iterator w powyższym przykładzie zachowuje się mniej lub bardziej jak wskaźnik. Natura jego użycia w powyższej aplikacji jest podobna do wskaźnika arytmetycznego, co można zauważyć w wierszu 25., w którym dostęp do wartości przechowywanej w wektorze odbywa się z wykorzystaniem operatora dereferencji (*), oraz w wierszu 29., gdzie iterator po inkrementacji za pomocą operatora ++ wskazuje kolejny element. Warto zwrócić uwagę na to, jak w wierszu 21. użyto funkcji `std::distance` w celu obliczenia wartości przesunięcia elementu. Iterator wskazujący element zostaje przekazany jako drugi parametr, pierwszym jest początek obiektu `vector` zwrócony przez iterator z funkcji `begin()`.

Usuwanie elementów z obiektu vector

Obiekt `vector` pozwala na umieszczanie elementów na końcu tablicy za pomocą metody `push_back()`, jednak oferuje również możliwość usunięcia elementu z końca za pomocą funkcji `pop_back()`. Usunięcie elementu z obiektu `vector` przy użyciu metody `pop_back()` jest operacją zabierającą stałą ilość czasu. Oznacza to, że wymagana ilość czasu na wykonanie tej operacji jest niezależna od liczby elementów przechowywanych w obiekcie `vector`. Kod przedstawiony w listingu 17.6 demonstruje użycie funkcji `pop_back()` w celu usunięcia elementów z końca obiektu `vector`.

Listing 17.6. Użycie metody pop_back() w celu usunięcia ostatniego elementu

```
0: #include <iostream>
1: #include <vector>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayVector(const vector<T>& vecInput)
6: {
7:     for (auto iElement = vecInput.cbegin() // auto i cbegin() dla C++11.
```

```
 8:          ; iElement != vecInput.cend() // cend() to nowość w C++11.
 9:          ; ++ iElement )
10:         cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     vector <int> vecIntegers;
18:
19:     // Wstawienie przykładowych liczb całkowitych do obiektu vector.
20:     vecIntegers.push_back (50);
21:     vecIntegers.push_back (1);
22:     vecIntegers.push_back (987);
23:     vecIntegers.push_back (1001);
24:
25:     cout << "Obiekt vector zawiera " << vecIntegers.size() <<
        ➥" elementy: ";
26:     DisplayVector(vecIntegers);
27:
28:     // Usunięcie ostatniego elementu.
29:     vecIntegers.pop_back ();
30:
31:     cout << "Po wywołaniu metody pop_back()" << endl;
32:     cout << "obiekt vector zawiera " << vecIntegers.size() <<
        ➥" elementy: ";
33:     DisplayVector(vecIntegers);
34:
35:     return 0;
36: }
```

Wynik ▼

```
Obiekt vector zawiera 4 elementy: 50 1 987 1001
Po wywołaniu metody pop_back()
obiekt vector zawiera 3 elementy: 50 1 987
```

Analiza ▼

Przedstawione dane wyjściowe wskazują, że funkcja `pop_back()` użyta w wierszu 29. zmniejszyła liczbę elementów obiektu `vector` poprzez usunięcie ostatniego elementu wstawionego do tego obiektu. W wierszu 32. ponownie wywołano funkcję `size()` w celu pokazania, że liczba elementów obiektu `vector` jest zmniejszona o jeden, co doskonale widać w danych wyjściowych.

Wskazówka

Funkcja `DisplayVector()` zdefiniowana w wierszach od 4. do 13. przybrała w listingu 17.6 formę wzorca, podczas gdy we wcześniejszym listingu 17.3 akceptowała jedynie wektor dla liczb całkowitych. W ten sposób funkcji wzorca można ponownie użyć dla wektora typu `float` (zamiast `int`):

```
vector <float> vecFloats;
DisplayVector(vecFloats); // To działa, ponieważ DisplayVector to ogólna funkcja.
```

Funkcja obsługuje teraz obiekty `vector` dowolnej klasy działającej z operatorem * i zwracającej wartość rozpoznawaną przez polecenie `cout`.

Zrozumienie koncepcji wielkości i pojemności

Wielkość (ang. *size*) obiektu `vector` to rzeczywista liczba przechowywanych w nim elementów. *Pojemność* (ang. *capacity*) obiektu `vector` to maksymalna liczba elementów, które potencjalnie mogą być przechowywane w tym obiekcie, zanim dojdzie do ponownej alokacji pamięci w celu pomieszczenia większej liczby elementów. Dlatego też *wielkość* obiektu `vector` jest mniejsza lub równa jego *pojemności*.

Wielkość obiektu `vector` można sprawdzić po wywołaniu metody `size()`:

```
cout << "Wielkość: " << vecIntegers.size ();
```

Natomiast pojemność można sprawdzić, wywołując metodę `capacity()`:

```
cout << "Pojemność: " <<  vecIntegers.capacity () << endl;
```

Obiekt `vector` może przyczyniać się do powstawania pewnych problemów związanych z wydajnością, kiedy często będzie przeprowadzana ponowna alokacja pamięci dla wewnętrznej tablicy dynamicznej. Te problemy mogą zostać w znacznym stopniu rozwiązane poprzez użycie funkcji składowej `reserve(liczba)`. Zadaniem funkcji `reserve()` jest zwiększenie ilości pamięci zaalokowanej dla wewnętrznej tablicy obiektu `vector`, tak aby możliwe było pomieszczenie wskazanej liczby elementów bez konieczności przeprowadzania ponownej alokacji. W zależności od typu obiektów przechowywanych w obiekcie `vector`, zmniejszenie liczby operacji ponownych alokacji powoduje również zmniejszenie liczby operacji kopiowania obiektów, a tym samym nie powoduje zmniejszania wydajności aplikacji. W przykładowym fragmencie kodu przedstawionym w poniższym listingu 17.7 pokazano różnicę między wielkością i pojemnością.

Listing 17.7. Demonstracja sposobów użycia funkcji size() i capacity()

```
 0: #include <iostream>
 1: #include <vector>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:
 7:     // Ustanowienie obiektu vector przechowującego pięć liczb całkowitych
        ↪z wartościami domyślnymi.
 8:     vector <int> vecIntegers (5);
 9:
10:     cout << "Obiekt vector został ustanowiony z parametrami " << endl;
11:     cout << "Wielkość: " << vecIntegers.size ();
12:     cout << ", Pojemność: " <<  vecIntegers.capacity () << endl;
13:
14:     // Wstawienie szóstego elementu do obiektu vector.
15:     vecIntegers.push_back (666);
16:
17:     cout << "Po wstawieniu elementu dodatkowego... " << endl;
18:     cout << "Wielkość: " << vecIntegers.size ();
19:     cout << ", Pojemność: " <<  vecIntegers.capacity () << endl;
20:
21:     // Wstawienie kolejnego elementu.
22:     vecIntegers.push_back (777);
23:
24:     cout << "Po wstawieniu kolejnego elementu... " << endl;
25:     cout << "Wielkość: " << vecIntegers.size ();
26:     cout << ", Pojemność: " <<  vecIntegers.capacity () << endl;
27:
28:     return 0;
29: }
```

Wynik ▼

```
Obiekt vector został ustanowiony z parametrami
Wielkość: 5, Pojemność: 5
Po wstawieniu elementu dodatkowego...
Wielkość: 6, Pojemność: 7
Po wstawieniu kolejnego elementu...
Wielkość: 7, Pojemność: 7
```

Analiza ▼

W wierszu 8. następuje ustanowienie obiektu `vector` zawierającego pięć liczb całkowitych z wartościami domyślnymi (0). W wierszach 12. i 13. zostaje wyświetlona (odpowiednio) wielkość i pojemność obiektu `vector`. W trakcie

ustanawiania obiektu obie wartości są takie same. W wierszu 9. do obiektu `vector` zostaje wstawiony szósty element. Ponieważ przed operacją wstawienia tego elementu pojemność obiektu `vector` wynosiła pięć, wewnętrzny bufor nie ma wystarczającej ilości pamięci do obsługi szóstego elementu. Innymi słowy, aby klasa `vector` zmieniła rozmiar i mogła przechowywać sześć elementów, musi ponownie przeprowadzić alokację wewnętrznego bufora. Implementacja logiki ponownej alokacji jest sprytna — w celu uniknięcia ponownej alokacji podczas wstawiania kolejnego elementu logika powoduje zaalokowanie większej pojemności, niż jest potrzebna w danej chwili.

Dane wyjściowe wyraźnie pokazują, że wstawienie szóstego elementu do obiektu `vector` o pojemności pięciu elementów powoduje podczas ponownej alokacji zwiększenie tej pojemności do siedmiu elementów. Funkcja `size()` zawsze odzwierciedla liczbę elementów w obiekcie `vector`, na tym etapie ma wartość 6. Dodanie siódmego elementu w wierszu 22. nie powoduje zwiększenia pojemności — aktualnie zaalokowana ilość pamięci pozwala na obsłużenie takiej liczby elementów. Na tym etapie wielkość i pojemność mają taką samą wartość. Wskazuje to na użycie całej pojemności przez obiekt `vector`. Dlatego też wstawienie kolejnego elementu spowoduje konieczność przeprowadzenia ponownej alokacji dla wewnętrznego bufora obiektu `vector` i skopiowania istniejących wartości przed wstawieniem nowej.

Uwaga

Wcześniejsze zwiększenie pojemności obiektu `vector` podczas ponownej alokacji pamięci dla wewnętrznego bufora nie jest zagwarantowane przez jakąkolwiek klauzulę w standardzie. To niezależne działanie używanego kontenera STL.

Klasa STL deque

Klasa deque (wymawiamy: *dek*) to klasa dynamicznej tablicy STL, której właściwości są podobne do oferowanych przez klasę `vector` z wyjątkiem tego, że pozwala na wstawianie i usuwanie elementów na początku i na końcu tablicy. Utworzenie egzemplarza tej klasy odbywa się w następujący sposób:

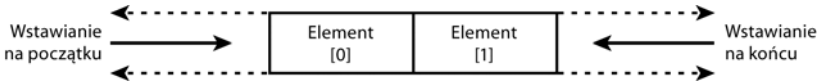
```
// Zdefiniowanie kolejki liczb całkowitych.
deque <int> dqIntegers;
```

Wskazówka

W celu użycia klasy `std::`deque w programie trzeba umieścić następujący nagłówek:

```
#include <deque>
```

W sposób graficzny klasę tę pokazano na rysunku 17.2.



RYSUNEK 17.2. Wewnętrzna budowa klasy deque

Klasa deque jest bardzo podobna do `vector`, pozwala na wstawianie i usuwanie elementów na końcu przy użyciu metod (odpowiednio) `push_back()` i `pop_back()`. Podobnie jak `vector`, także klasa deque umożliwia uzyskanie dostępu do elementów za pomocą semantyki tablicy, czyli operatora indeksowania (`[]`). Różnica pomiędzy klasami deque i `vector` polega na tym, że deque pozwala na wstawianie i usuwanie elementów na początku kolejki przy użyciu metod (odpowiednio) `push_front()` i `pop_front()`, co zaprezentowano w listingu 17.8.

Listing 17.8. Utworzenie egzemplarza klasy STL deque i użycie metod push_front() oraz pop_front() w celu (odpowiednio) wstawiania i usuwania elementów na początku kolejki

```
 0: #include <deque>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:
 8:     // Zdefiniowanie liczb całkowitych obiektu deque.
 9:     deque <int> dqIntegers;
10:
11:     // Wstawienie liczb całkowitych na końcu tablicy.
12:     dqIntegers.push_back (3);
13:     dqIntegers.push_back (4);
14:     dqIntegers.push_back (5);
15:
16:     // Wstawienie liczb całkowitych na początku tablicy.
17:     dqIntegers.push_front (2);
18:     dqIntegers.push_front (1);
19:     dqIntegers.push_front (0);
20:
21:     cout << "Zawartość obiektu deque po wstawieniu elementów ";
22:     cout << "na początku i na końcu to:" << endl;
23:
24:     // Wyświetlenie na ekranie zawartości obiektu.
25:     for ( size_t nCount = 0
26:         ; nCount < dqIntegers.size ()
```

```
27:         ; ++ nCount )
28:     {
29:         cout << "Element [" << nCount << "] = ";
30:         cout << dqIntegers [nCount] << endl;
31:     }
32:
33:     cout << endl;
34:
35:     // Usunięcie elementu z początku tablicy.
36:     dqIntegers.pop_front ();
37:
38:     // Usunięcie elementu z końca tablicy.
39:     dqIntegers.pop_back ();
40:
41:     cout << "Zawartość obiektu deque po usunięciu elementów";
42:     cout << "z początku i końca obiektu to:" << endl;
43:
44:     // Ponowne wyświetlenie zawartości obiektu, tym razem za pomocą iteratorów.
45:     // Jeśli używasz starszego kompilatora, usuń słowo auto i znak komentarza
       ↪w kolejnym wierszu.
46:     // deque <int>::iterator iElementLocator;
47:     for (auto iElementLocator = dqIntegers.begin ()
48:         ; iElementLocator != dqIntegers.end ()
49:         ; ++ iElementLocator )
50:     {
51:         size_t Offset = distance (dqIntegers.begin (), iElementLocator);
52:         cout<< "Element [" << Offset << "] = " << *iElementLocator <<
           ↪endl;
53:     }
54:
55:     return 0;
56: }
```

Wynik ▼

```
Zawartość obiektu deque po wstawieniu elementów na początku i na końcu to:
Element [0] = 0
Element [1] = 1
Element [2] = 2
Element [3] = 3
Element [4] = 4
Element [5] = 5
Zawartość obiektu deque po usunięciu elementów z początku i końca obiektu to:
Element [0] = 1
Element [1] = 2
Element [2] = 3
Element [3] = 4
```

Analiza ▼

W wierszu 10. ustanowiono obiekt deque wraz z liczbami całkowitymi. Zwróć uwagę na podobieństwo tej składni do ustanawiania obiektu `vector` wraz z liczbami całkowitymi. W wierszach od 13. do 16. widzimy użycie funkcji składowej `push_back()` obiektu deque, natomiast w wierszach od 18. do 20. funkcji składowej `push_front()`. Druga z wymienionych funkcji powoduje, że obiekt deque różni się od obiektu `vector`. Funkcja `pop_front()` jest jednak używana w taki sam sposób, jaki można zobaczyć w wierszu 37. Pierwszy mechanizm wyświetlania zawartości obiektu deque używa składni tablicy w celu uzyskania dostępu do elementów, podczas gdy w drugim mechanizmie użyto iteratorów. W przypadku tego drugiego (zobacz wiersze od 47. do 53.) algorytm `std::distance` został wykorzystany w celu obliczenia pozycji elementu w obiekcie deque. Działa to dokładnie w taki sam sposób, jaki widzieliśmy już w przykładzie używającym obiektu `vector` (zobacz listing 17.5).

TAK	NIE
Używaj tablic dynamicznych vector lub deque, kiedy nie znasz dokładnej liczby elementów, które trzeba będzie przechowywać.	Nie zapominaj, że metoda pop_back() usuwa ostatni element w kolekcji.
Pamiętaj, że obiekt vector może być zwiększany tylko przez wstawianie elementów na jego końcu przy użyciu metody push_pack().	Nie zapominaj, że metoda pop_front() usuwa pierwszy element w kolekcji.
Pamiętaj, że obiekt deque może być zwiększany przez wstawianie elementów zarówno na jego początku, jak i końcu przy użyciu metod (odpowiednio) push_front() i push_pack().	Nie próbuj uzyskać dostępu do tablicy dynamicznej, podając wartość indeksu wykraczającą poza granice tablicy.

Podsumowanie

W tej lekcji poznałeś podstawy używania obiektów `vector` i deque jako tablic dynamicznych. Wyjaśniona została także koncepcja wielkości i pojemności. Dowiedziałeś się, w jaki sposób użycie obiektu `vector` może być zoptymalizowane w celu zmniejszenia liczby operacji ponownej alokacji jego bufora wewnętrznego, które powodują kopiowanie obiektów, potencjalnie prowadząc do spadku wydajności aplikacji. Obiekt `vector` to najprostszy z kontenerów STL, jednocześnie najczęściej używany i bezsprzecznie najbardziej efektywny.

Pytania i odpowiedzi

Pytanie: Czy obiekt `vector` zmienia kolejność przechowywanych w nim elementów?

Odpowiedź: Obiekt `vector` to kontener sekwencyjny, więc elementy są przechowywane i udostępniane w kolejności, w której zostały wstawione.

Pytanie: Która funkcja jest używana podczas wstawiania elementów do obiektu `vector` i gdzie te elementy są umieszczane?

Odpowiedź: Funkcja składowa `push_back()` wstawia elementy na końcu obiektu `vector`.

Pytanie: Która funkcja podaje liczbę elementów przechowywanych w obiekcie `vector`?

Odpowiedź: Wartością zwracaną przez funkcję składową `size()` jest liczba elementów przechowywanych w obiekcie `vector`. Nawiasem mówiąc, funkcja ta jest stosowana we wszystkich kontenerach STL.

Pytanie: Czy wstawianie lub usuwanie elementów na końcu obiektu `vector` wymaga większej ilości czasu, kiedy obiekt `vector` zawiera większą liczbę elementów?

Odpowiedź: Nie. Czas wstawiania i usuwania elementów na końcu obiektu `vector` jest stały i nie zależy od liczby elementów.

Pytanie: Jakie są zalety używania funkcji składowej `reserve()`?

Odpowiedź: Funkcja składowa `reserve(...)` powoduje zaalokowanie przestrzeni dla wewnętrznego bufora obiektu `vector`. Z tego powodu wstawianie elementów nie będzie wymagało przeprowadzenia ponownej alokacji bufora obiektu `vector` i kopiowania jego istniejącej zawartości. W zależności od natury obiektów przechowywanych w obiekcie `vector`, zarezerwowanie przestrzeni dla obiektu `vector` może skutkować zwiększeniem wydajności działania programu.

Pytanie: Czy właściwości obiektu `deque` w jakiś sposób różnią się od właściwości obiektu `vector` w zakresie wstawiania elementów?

Odpowiedź: Nie, w zakresie wstawiania elementów właściwości obiektu deque są podobne do oferowanych przez vector. Oznacza to stały czas potrzebny na wstawienie elementów na końcu sekwencji oraz liniowy czas podczas wstawiania elementów w środku sekwencji. Jednak obiekt vector pozwala na wstawianie elementów jedynie na końcu (na dole) obiektu, podczas gdy obiekt deque pozwala na wstawianie elementów zarówno na końcu, jak i na początku (na dole i na górze) obiektu.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Czy można wstawiać elementy w środku bądź na początku obiektu vector w stałym czasie?
2. W moim obiekcie vector wartość funkcji size() wynosi 10, natomiast funkcji capacity() wynosi 20. Ile jeszcze mogę wstawić elementów, zanim klasa vector wymusi przeprowadzenie ponownej alokacji bufora?
3. Do czego służy funkcja pop_back()?
4. Jeżeli vector<int> to dynamiczna tablica liczb całkowitych, to jakiego typu jest tablica vector<CMammal>?
5. Czy możliwe jest uzyskanie swobodnego dostępu do elementów obiektu vector? Jeśli tak, to w jaki sposób?
6. Jaki typ iteratora pozwala na uzyskanie swobodnego dostępu do elementów obiektu vector?

Ćwiczenia

1. Napisz interaktywny program przyjmujący od użytkownika dane wejściowe w postaci liczb całkowitych i zapisujący je w obiekcie `vector`. Użytkownik powinien mieć w każdej chwili możliwość sprawdzenia wartości przechowywanej w obiekcie `vector` na podstawie podanego indeksu.
2. Rozbuduj program z ćwiczenia 1. w taki sposób, aby informował użytkownika, czy podawana wartość istnieje już w obiekcie `vector`.
3. Jacek sprzedaje produkty w portalu eBay. Aby pomóc mu w pakowaniu i wysyłce towaru, utwórz program, w którym będzie mógł podać wymiary każdego artykułu. Podane wymiary powinny być przechowywane w obiekcie `vector` oraz wyświetlane na ekranie.

Lekcja 18

Klasy STL list i forward_list

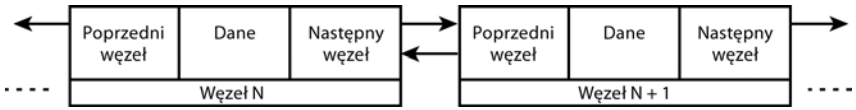
Standardowa biblioteka wzorców (STL) dostarcza programiście listę dwukierunkową w postaci klasy wzorca std::list. Podstawową zaletą listy jest stały i krótki czas wstawiania oraz usuwania elementów. W standardzie C++11 można również używać listy jednokierunkowej w postaci klasy std::forward_list pozwalającej na poruszanie się tylko w jednym kierunku.

Z tej lekcji dowiesz się:

- **sposoby tworzenia egzemplarzy klas list i forward_list,**
- **sposoby użycia klas STL obiektu list, m.in. wstawianie i usuwanie elementów,**
- **sposoby odwracania i sortowania elementów.**

Charakterystyka klasy std::list

Lista jednokierunkowa to zbiór węzłów, w którym każdy węzeł, poza tym, że zawiera wartość bądź obiekt, prowadzi również do kolejnego węzła. Oznacza to, że każdy węzeł prowadzi do poprzedniego i kolejnego, jak pokazano na rysunku 18.1.



RYSUNEK 18.1. Graficzne przedstawienie listy dwukierunkowej

Klasa STL `list` charakteryzuje się stałym czasem przeprowadzania operacji wstawiania elementów na początku, końcu bądź w środku sekwencji.

Wskazówka

W celu użycia klasy `std::list` w programie trzeba umieścić następujący nagłówek:

```
#include <list>
```

Podstawowe operacje klasy list

Zgodne ze standardem implementacje klasy STL `list` wymagają nagłówka pliku `<list>`, który programista musi dołączyć, aby można było używać tej klasy. Klasa wzorca `list` istniejąca w przestrzeni nazw `std` jest ogólną implementacją, która musi być ustanowiona przed rozpoczęciem używania jakichkolwiek jej funkcji składowych.

Ustanawianie obiektu std::list

Ustanowienie listy wymaga przeprowadzenia specjalizacji klasy wzorca `std::list` dla typu, który będzie przechowywany w danej liście. Inicjalizacja obiektu będzie więc odbywała się następująco:

```
std::list<int> listIntegers;   // Lista zawierająca liczby całkowite.
std::list<float> listFloats;   // Lista zawierająca liczby zmiennoprzecinkowe.
std::list<Tuna> listTunas;   // Lista zawierająca obiekty typu Tuna.
```

Aby zadeklarować iterator prowadzący do elementu listy, należy użyć poniższego polecenia:

```
std::list<int>::const_iterator iElementInSet;
```

Jeżeli potrzebujesz iteratora, który może być używany do modyfikacji wartości lub wywoływania funkcji innych niż `const`, w przedstawionym poleceniu musisz zastosować `iterator` zamiast `const_iterator`.

Implementacja `std::list` zawiera zestaw przeciążonych konstruktorów; masz nawet możliwość utworzenia listy zainicjalizowanej z wybraną liczbą elementów, a każdy z nich może mieć przypisaną wartość, co zostało przedstawione w listingu 18.1.

Listing 18.1. Różne formy ustanowienia obiektu STL list: wraz ze wskazaną liczbą elementów oraz z wartościami początkowymi

```
 0: #include <list>
 1: #include <vector>
 2:
 3: int main ()
 4: {
 5:    using namespace std;
 6:
 7:    // Inicjalizacja pustej listy.
 8:    list <int> listIntegers;
 9:
10:    // Inicjalizacja listy wraz z 10 liczbami całkowitymi.
11:    list<int> listWith10Integers(10);
12:
13:    // Inicjalizacja listy wraz z 4 liczbami całkowitymi, każda o wartości  99.
14:    list<int> listWith4IntegerEach99 (4, 99);
15:
16:    // Utworzenie dokładnej kopii istniejącej listy.
17:    list<int> listCopyAnother(listWith4IntegerEach99);
18:
19:    // Wektor z 10 liczbami całkowitymi, każda o wartości 2011.
20:    vector<int> vecIntegers(10, 2011);
21:
22:    // Utworzenie listy na podstawie wartości pochodzących z innego kontenera.
23:    list<int> listContainsCopyOfAnother(vecIntegers.cbegin(),
24:                                        vecIntegers.cend());
25:
26:    return 0;
27: }
```

Analiza ▼

Program nie generuje danych wyjściowych i pokazuje użycie różnych przeciążonych konstruktorów do budowania listy liczb całkowitych.

W wierszu 8. tworzona jest pusta lista, natomiast w wierszu 11. — lista zawierająca dziesięć liczb całkowitych. Z kolei w wierszu 14. powstaje lista o nazwie `listWith4IntegersEach99` zawierająca cztery liczby całkowite, każda o wartości 99. W wierszu 17. zaprezentowano utworzenie listy będącej dokładną kopią innej. Wiersze od 20. do 24. mogą być zaskakujące i ciekawe. Najpierw budowany jest wektor dziesięciu liczb całkowitych o wartości 2011 każda, a następnie (patrz wiersz 23.) zainicjalizowana zostaje lista zawierająca elementy skopiowane z wektora. W operacji użyto iteratorów typu `const` zwróconych przez metody `vector::cbegin()` i `vector::cend()`, które są nowością wprowadzoną w standardzie C++11. Listing 18.1 to również przykład pokazujący, jak iteratory mogą pomóc w zdefiniowaniu jednego kontenera na podstawie innego — podstawową funkcjonalność iteratorów można wykorzystać do utworzenia listy zawierającej wartości pobrane z wektora (patrz wiersze 23. i 24.).

Wskazówka

Czy użycie metod `cbegin()` i `cend()` powoduje błędy w trakcie kompilacji programu?

Jeżeli przedstawiony program próbujesz skompilować przy użyciu kompilatora niezgodnego ze standardem C++11, wtedy zamiast metod `cbegin()` i `cend()` użyj (odpowiednio) `begin()` i `end()`. Oferowane przez standard C++11 metody `cbegin()` i `cend()` są użyteczne, ponieważ zwracają iterator `const`, który nie może być wykorzystany do modyfikacji elementów.

Uwaga

Porównując listing 18.1 z przedstawionym w lekcji 17. listingiem 17.1, zauważysz podobieństwa w zakresie tworzenia kontenerów różnego typu. Im częściej będziesz korzystał z kontenerów STL, tym więcej znajdziesz wzorców do ponownego użycia, a samo pisanie kodu stanie się łatwiejsze.

Wstawianie elementów na początku obiektu list

Podobnie jak w klasie deque, wstawienie elementu na początku (tzn. w położeniu poprzedzającym istniejące elementy w sekwencji) odbywa się za pomocą metody składowej `push_front()` obiektu `list`. Z kolei wstawienie elementu na końcu umożliwia metoda `push_back()`. Obie metody akceptują pojedynczy parametr — wartość przeznaczoną do wstawienia:

```
listIntegers.push_back (-1);
listIntegers.push_front (2001);
```

W listingu 18.2 zaprezentowano efekt użycia wymienionych metod.

Listing 18.2. Wstawianie elementów w obiekcie list za pomocą metod push_front() i push_back()

```
 0: #include <list>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents(const T& Input)
 6: {
 7:     for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
 8:         ; iElement != Input.cend()
 9:         ; ++ iElement )
10:        cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     std::list <int> listIntegers;
18:
19:    listIntegers.push_front (10);
20:    listIntegers.push_front (2001);
21:    listIntegers.push_back (-1);
22:    listIntegers.push_back (9999);
23:
24:    DisplayContents(listIntegers);
25:
26:    return 0;
27: }
```

Wynik ▼

```
2011 10 -1 9999
```

Analiza ▼

W wierszach od 19. do 22. zaprezentowano użycie metod `push_front()` i `push_back()`. Wartość dostarczona jako argument metody `push_front()` pobiera położenie pierwszego elementu listy, natomiast dla `push_back()`

jest to położenie ostatniego elementu listy. Dane wyjściowe pokazują zawartość listy wyświetloną przy użyciu ogólnej funkcji wzorca `DisplayContents()` wskazującej kolejność wstawianych elementów (nie są one jednak przechowywane w kolejności wstawiania).

Ostrzeżenie

Czy użycie słowa kluczowego `auto` powoduje błędy w trakcie kompilacji?

W przedstawionej w listingu 18.2 funkcji `DisplayContents()` użyto wprowadzonego w standardzie C++11 słowa kluczowego `auto` do zdefiniowania typu iteratora (patrz wiersz 7.). Ponadto kod zawiera wywołania metod `cbegin()` i `cend()`, które są nowością w standardzie C++11 i zwracają wartość typu `const_iterator`.

Aby ten i kolejne przykłady skompilować w starszej wersji kompilatora, nieobsługującej standardu C++11, musisz słowo kluczowe `auto` zastąpić konkretnym typem.

Dlatego też metoda `DisplayContent()` dla starszej wersji kompilatora będzie miała następującą postać:

```
void DisplayContent(const T& Input)
{
   for (T::const_iterator iElement = Input.begin()
      ; iElement != Input.end ()
      ; ++ iElement )
     cout << *iElement << ' ';

   cout << endl;
}
```

Wskazówka

Funkcja `DisplayContents()` zdefiniowana w wierszach od 4. do 13. przybrała w listingu 18.2 formę bardziej ogólnej metody `DisplayVector()` z 17.6 (zwróć uwagę na zmienioną listę parametrów). Wprawdzie metoda `DisplayVector()` działała jedynie z wektorami, uogólnienie typu przechowywanych elementów powoduje, że metoda `DisplayContents()` naprawdę jest ogólną metodą obsługującą różne typy kontenerów.

Metodę `DisplayContents()` w listingu 18.2 możesz wywołać z argumentem `vector` lub `list` i nadal będzie ona działała doskonale.

Wstawianie elementów w środku obiektu list

Klasę `std::list` charakteryzuje możliwość wstawiania elementów w środku obiektu. Taka operacja przebiega w stałym czasie i jest przeprowadzana za pomocą funkcji składowej `insert()`.

Funkcja składowa `list::insert` dostępna jest w trzech postaciach:

- **Postać 1.**

  ```
  iterator insert(iterator pos, const T& x)
  ```

  W powyższej postaci funkcja `insert()` przyjmuje jako pierwszy parametr położenie, w którym ma zostać wstawiony element. Natomiast drugim parametrem jest wstawiana wartość. Funkcja zwraca iterator wskazujący element ostatnio wstawiony w obiekcie `list`.

- **Postać 2.**

  ```
  void insert(iterator pos, size_type n, const T& x)
  ```

  W powyższej postaci funkcja `insert()` przyjmuje jako pierwszy parametr położenie, w którym ma zostać wstawiony element. Natomiast wstawiana wartość jest ostatnim parametrem. Zmienna `n` wskazuje liczbę elementów.

- **Postać 3.**

  ```
  template <class InputIterator>
  void insert(iterator pos, InputIterator f, InputIterator l)
  ```

  Ten przeciążony wariant to funkcja wzorca, która — oprócz położenia wstawianych elementów — przyjmuje także dwa iteratory wejściowe ograniczające zbiór przeznaczony do wstawienia w obiekcie `list`. Warto zwrócić uwagę, że typ wejściowy `InputIterator` jest typem sparametryzowanego wzorca i dlatego może prowadzić do dowolnej kolekcji — może to być tablica, obiekt `vector` lub po prostu inny obiekt `list`.

W listingu 18.3 zademonstrowano użycie tych przeciążonych wariantów funkcji `list::insert`.

Listing 18.3. Różne sposoby wstawiania elementów do obiektu list

```
0: #include <list>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayContents(const T& Input)
```

```
 6: {
 7:     for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
 8:         ; iElement != Input.cend()
 9:         ; ++ iElement )
10:        cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     list <int> listIntegers1;
18:
19:     // Wstawianie elementów na początku…
20:     listIntegers1.insert (listIntegers1.begin (), 2);
21:     listIntegers1.insert (listIntegers1.begin (), 1);
22:
23:     // Wstawianie elementów na końcu…
24:     listIntegers1.insert (listIntegers1.end (), 3);
25:
26:     cout << "Zawartość obiektu list1 po wstawieniu elementów: " << endl;
27:     DisplayContents (listIntegers1);
28:
29:     list <int> listIntegers2;
30:
31:     // Wstawienie czterech elementów wraz z tą samą wartością 0…
32:     listIntegers2.insert (listIntegers2.begin (), 4, 0);
33:
34:     cout << "Zawartość obiektu list2 po wstawieniu '";
35:     cout << listIntegers2.size () << "' elementów o wartości:" << endl;
36:     DisplayContents (listIntegers2);
37:
38:     list <int> listIntegers3;
39:
40:     // Wstawienie na początku elementów pochodzących z innego obiektu list…
41:     listIntegers3.insert (listIntegers3.begin (),
42:                           listIntegers1.begin (), listIntegers1.end ());
43:
44:     cout << "Zawartość obiektu list3 po wstawieniu na początku ";
45:     cout << "zawartości obiektu list1:" << endl;
46:     DisplayContents (listIntegers3);
47:
48:     //  Wstawienie na końcu elementów pochodzących z innego obiektu list…
49:     listIntegers3.insert (listIntegers3.end (),
50:                           listIntegers2.begin (), listIntegers2.end ());
51:
52:     cout << "Zawartość obiektu list3 po wstawieniu na początku ";
53:     cout << "zawartości obiektu list2:" << endl;
```

```
54:     DisplayContents (listIntegers3);
55:
56:     return 0;
57: }
```

Wynik ▼

```
Zawartość obiektu list1 po wstawieniu elementów:
1 2 3

Zawartość obiektu list2 po wstawieniu '4' elementów o wartości:
0 0 0 0

Zawartość obiektu list3 po wstawieniu na początku zawartości obiektu list1:
1 2 3

Zawartość obiektu list3 po wstawieniu na początku zawartości obiektu list2:
1 2 3 0 0 0 0
```

Analiza ▼

W listingu 18.3 `begin()` i `end()` to funkcje składowe zwracające iteratory prowadzące do początku i końca obiektu `list`. Dotyczy to wszystkich kontenerów STL, w tym także `std::list`. Funkcja `list::insert` przyjmuje iterator wskazujący położenie, przed którym mają być wstawiane elementy. Iterator zwrócony przez funkcję `end()` prowadzi do położenia znajdującego się za ostatnim elementem obiektu `list`. Dlatego też w wierszu 24. następuje wstawienie wartości 3 przed końcem obiektu. Wiersz 32. pokazuje inicjalizację listy wraz z czterema elementami umieszczonymi na początku, każdy z nich ma wartość 0. W wierszach 41. i 42. pokazano, w jaki sposób zawartość jednego obiektu `list` może być umieszczona w innym. Wprawdzie w tym przykładzie następuje wstawienie obiektu `list` zawierającego liczby całkowite do innego obiektu `list`, jednak zakres wstawianych elementów jest ograniczony jedynie przez możliwości obiektu `vector` i — jak pokazano w listingu 18.1 — także zwykłych tablic statycznych.

Usuwanie elementów w obiekcie list

Funkcja składowa `erase()` obiektu `list` jest dostępna w dwóch przeciążonych formach. Pierwsza powoduje usunięcie jednego elementu na podstawie iteratora prowadzącego do tego elementu, natomiast druga akceptuje zakres

i dlatego może usunąć wskazany zakres elementów w danym obiekcie `list`. Funkcję `list::erase()` można w działaniu zobaczyć w listingu 18.4, w którym przedstawiono usunięcie zarówno pojedynczego elementu, jak i zakresu elementów w obiekcie `list`.

Listing 18.4. Usuwanie elementów w obiekcie list

```
 0: #include <list>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents(const T& Input)
 6: {
 7:     for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
 8:         ; iElement != Input.cend()
 9:         ; ++ iElement )
10:        cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     std::list <int> listIntegers;
18:
19:     // Wstawianie elementów na początku i końcu…
20:     listIntegers.push_back (4);
21:     listIntegers.push_front (3);
22:     listIntegers.push_back (5);
23:
24:     // Przechowywanie iteratora uzyskanego za pomocą funkcji insert.
25:     auto iValue2 = listIntegers.insert (listIntegers.begin (), 2);
26:
27:     cout << "Początkowa zawartość obiektu list:" << endl;
28:     DisplayContents(listIntegers);
29:
30:     listIntegers.erase (listIntegers.begin (), iValue2);
31:     cout << "Zawartość po usunięciu zakresu elementów:" << endl;
32:     DisplayContents(listIntegers);
33:
34:     cout << "Zawartość po usunięciu elementu '" << *iValue2 << "':" <<
      ↪endl;
35:     listIntegers.erase (iValue2);
36:     DisplayContents(listIntegers);
37:
38:     listIntegers.erase (listIntegers.begin (), listIntegers.end ());
39:     cout << "Liczba elementów po usunięciu zakresu:" << endl;
```

```
40:     cout << listIntegers.size() << endl;
41:
42:     return 0;
43: }
```

Wynik ▼

```
Początkowa zawartość obiektu list:
2 3 4 5

Zawartość po usunięciu zakresu elementów:
2 3 4 5

Zawartość po usunięciu elementu '2':
3 4 5

Liczba elementów po usunięciu zakresu: 0
```

Analiza ▼

Wiersze od 20. do 25. w funkcji `main()` pokazują użycie różnych metod w celu wstawienia liczb całkowitych do listy. Kiedy metoda `insert()` jest używana do wstawiania wartości, jej wartością zwrotną jest iterator do nowo wstawionego elementu. Wspomniany iterator prowadzi do wartości 2 przechowywanej w zmiennej `iValue2` (patrz wiersz 25.), która będzie w wierszu 35. użyta w trakcie wywołania `erase()` usuwającego ten element z listy. W wierszach od 30. do 38. zaprezentowano użycie metody `erase()` do usunięcia zakresu elementów. Najpierw usuwany jest zakres, począwszy od `begin()` do elementu zawierającego wartość 2 (ale bez tego elementu). Następnie usuwana jest zawartość zakresu od `begin()` do `end()`, czyli w praktyce cała lista.

Uwaga

W listingu 18.4 polecenie znajdujące się w wierszu 40. wyświetla liczbę elementów w `std::list` określoną przy użyciu `size()`, bardzo podobnie jak w wektorze. Tak samo można ustalić liczbę elementów we wszystkich klasach kontenerów STL.

Odwrócenie i sortowanie elementów w obiekcie list

Obiekt `list` ma specjalną właściwość polegającą na tym, że iteratory prowadzące do elementów obiektu `list` pozostają prawidłowe w przypadku zmiany kolejności elementów, wstawienia nowych itd. Aby pozostawić tę

właściwość nietkniętą, obiekt `list` zawiera funkcje `sort()` i `reverse()` działające jako metody składowe, chociaż standardowa biblioteka wzorców dostarcza je jako algorytmy, które działają z klasą `list`. Wersje składowe tych algorytmów gwarantują, że iteratory wskazujące elementy w obiekcie `list` nie zostaną unieważnione, kiedy zmianie ulegnie względne położenie tych elementów.

Odwracanie elementów

Obiekt `list` oferuje funkcję składową `reverse()`, która nie pobiera parametrów i odwraca kolejność zawartości obiektu `list`:

```
listIntegers.reverse(); // Odwrócenie kolejności elementów.
```

Przykład użycia tej funkcji pokazano w listingu 18.5.

Listing 18.5. Odwracanie elementów w obiekcie list

```
 0: #include <list>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents(const T& Input)
 6: {
 7:     for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
 8:         ; iElement != Input.cend()
 9:         ; ++ iElement )
10:        cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     std::list <int> listIntegers;
18:
19:     // Wstawianie elementów na początku i końcu.
20:     listIntegers.push_front (4);
21:     listIntegers.push_front (3);
22:     listIntegers.push_front (2);
23:     listIntegers.push_front (1);
24:     listIntegers.push_front (0);
25:     listIntegers.push_back (5);
26:
27:     cout << "Początkowa zawartość obiektu list:" << endl;
28:     DisplayContents(listIntegers);
```

```
29:
30:     listIntegers.reverse ();
31:
32:     cout << "Zawartość obiektu list po użyciu funkcji reverse():" <<
        ↪endl;
33:     DisplayContents(listIntegers);
34:
35:     return 0;
36: }
```

Wynik ▼

```
Początkowa zawartość obiektu list:
0 1 2 3 4 5
Zawartość obiektu list po użyciu funkcji reverse():
5 4 3 2 1 0
```

Analiza ▼

Jak pokazano w wierszu 30., funkcja `reverse()` po prostu odwraca kolejność elementów w obiekcie `list`. To proste wywołanie funkcji bez parametrów gwarantujące, że iteratory wskazujące elementy obiektu `list` pozostaną poprawne nawet po wykonaniu operacji odwrócenia.

Sortowanie elementów

Funkcja składowa `sort()` obiektu `list` jest dostępna w wersji niepobierającej parametrów:

```
listIntegers.sort();   // Sortowanie w kolejności rosnącej.
```

oraz w wersji, która jako parametr akceptuje funkcję z predykatem dwuargumentowym i przeprowadza sortowanie na podstawie kryteriów określonych w podanej funkcji:

```
bool SortPredicate_Descending (const int& lsh, const int& rsh)
{
   // Zdefiniowanie kryteriów dla list::sort: typ wartości zwrotnej dla sortowania w danej kolejności.
   return (lsh > rsh);
}
// Użycie predykatu w celu posortowania listy.
listIntegers.sort (SortPredicate_Descending);
```

Obie wersje zostały przedstawione w listingu 18.6.

Listing 18.6. Sortowanie obiektu list zawierającego liczby całkowite w kolejności rosnącej i malejącej przy użyciu metody list::sort()

```
 0: #include <list>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: bool SortPredicate_Descending (const int& lsh, const int& rsh)
 5: {
 6:     // Zdefiniowanie kryteriów dla list::sort: typ wartości zwrotnej dla sortowania
       ➥w danej kolejności.
 7:     return (rsh < lsh);
 8: }
 9:
10: template <typename T>
11: void DisplayContents(const T& Input)
12: {
13:     for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
14:         ; iElement != Input.cend()
15:         ; ++ iElement )
16:        cout << *iElement << ' ';
17:
18:     cout << endl;
19: }
20:
21: int main ()
22: {
23:     std::list <int> listIntegers;
24:
25:     // Wstawianie elementów na początku i końcu.
26:     listIntegers.push_front (444);
27:     listIntegers.push_front (2011);
28:     listIntegers.push_front (-1);
29:     listIntegers.push_front (0);
30:     listIntegers.push_back (-5);
31:
32:     cout << "Początkowa zawartość obiektu list -" << endl;
33:     DisplayContents(listIntegers);
34:
35:     listIntegers.sort ();
36:
37:     cout << "Kolejność elementów po użyciu funkcji sort():" << endl;
38:     DisplayContents(listIntegers);
39:
40:     listIntegers.sort (SortPredicate_Descending);
41:     cout << "Kolejność elementów po użyciu funkcji sort()
       ➥z predykatem:" << endl;
42:     DisplayContents(listIntegers);
```

```
43:
44:     return 0;
45: }
```

Wynik ▼

```
Początkowa zawartość obiektu list -
0 -1 2011 444 -5

Kolejność elementów po użyciu funkcji sort():
-5 -1 0 444 2011

Kolejność elementów po użyciu funkcji sort() z predykatem:
2011 444 0 -1 -5
```

Analiza ▼

W powyższym listingu pokazano sortowanie na przykładzie obiektu `list` zawierającego liczby całkowite. Kilka pierwszych linii kodu tworzy obiekt `list` i umieszcza w nim przykładowe wartości. W wierszu 35. pokazano użycie funkcji `sort()` bez parametrów, która domyślnie sortuje elementy w kolejności rosnącej, porównując liczby całkowite za pomocą operatora <, który w przypadku liczb całkowitych jest implementowany przez kompilator. Jeżeli programista chce zmienić to zachowanie domyślne, musi dostarczyć funkcję sortowania wraz z predykatem dwuargumentowym. Zdefiniowana w liniach od 4. do 8. funkcja `SortPredicate_Descending()` to predykat dwuargumentowy pomagający funkcji `sort()` obiektu `list` w określeniu, czy dany element jest mniejszy od poprzedniego. Jeżeli nie, ich pozycje zostają zamienione. Innymi słowy, wskazujesz liście, co powinno zostać zinterpretowano jako mniejsze (w omawianym przypadku pierwszy parametr jest większy od drugiego). Funkcja jest przekazywana jako parametr wariantowi funkcji `sort()`, co pokazano w wierszu 40. Predykat zwraca wartość `true`, jeżeli pierwsza wartość jest większa od drugiej. Dlatego też funkcja `sort()` używająca predykatu interpretuje, że pierwsza wartość (`lsh`) pod względem logicznym jest uznawana za mniejszą od drugiej (`rsh`) tylko wtedy, jeżeli wartość liczbowa tej pierwszej jest większa od drugiej. Na podstawie tej interpretacji następuje zamiana położeń w celu spełnienia wymagań predykatu.

Sortowanie i usuwanie elementów listy zawierających obiekty danej klasy

Co zrobić, jeśli lista jest typu klasy, a nie prostego typu wbudowanego, takiego jak `int`? Powiedzmy, że mamy listę danych kontaktowych, w której każdy element składa się z imienia i nazwiska, adresu itd. W jaki sposób zagwarantować posortowanie listy według nazwisk?

Odpowiedzią jest zastosowanie jednego z dwóch poniższych rozwiązań.

- Implementacja operatora < w danej klasie, której obiekty są przechowywane w liście.
- Dostarczenie *binarnego predykatu* sortowania — jest to funkcja pobierająca dane wejściowe w postaci dwóch wartości i zwracająca wartość boolowską wskazującą, czy pierwsza wartość jest mniejsza od drugiej.

W większości aplikacji praktycznych wykorzystujących kontenery STL rzadko używa się liczb całkowitych, są to raczej typy zdefiniowane przez użytkownika, np. *klasy* bądź *struktury*. W listingu 18.7 zademonstrowano użycie obiektu `list` przechowującego dane kontaktowe. W pierwszej chwili program wydaje się długi, ale w większości składa się z prostego kodu.

Listing 18.7. Obiekt list zawierający obiekty struktury: tworzenie danych kontaktowych

```
 0: #include <list>
 1: #include <string>
 2: #include <iostream>
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents(const T& Input)
 7: {
 8:     for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
 9:         ; iElement != Input.cend()
10:         ; ++ iElement )
11:        cout << *iElement << ' ';
12:
13:     cout << endl;
14: }
15:
16: struct ContactItem
17: {
18:     string strContactsName;
19:     string strPhoneNumber;
```

```
20:     string strDisplayRepresentation;
21:
22:     // Konstruktor i destruktor.
23:     ContactItem (const string& strName, const string & strNumber)
24:     {
25:         strContactsName = strName;
26:         strPhoneNumber = strNumber;
27:         strDisplayRepresentation = (strContactsName + ":
            ↪" + strPhoneNumber);
28:     }
29:
30:     // Operator używany przez list::remove().
31:     bool operator == (const ContactItem& itemToCompare) const
32:     {
33:         return (itemToCompare.strContactsName == this->strContactsName);
34:     }
35:
36:     // Operator używany przez list::sort() bez parametrów.
37:     bool operator < (const ContactItem& itemToCompare) const
38:     {
39:         return (this->strContactsName < itemToCompare.strContactsName);
40:     }
41:
42:    // Operator używany w DisplayContents() za pomocą polecenia cout.
43:    operator const char*() const
44:    {
45:      return strDisplayRepresentation.c_str();
46:    }
47: };
48:
49: bool SortOnPhoneNumber (const ContactItem& item1,
50:                         const ContactItem& item2)
51: {
52:    return (item1.strPhoneNumber < item2.strPhoneNumber);
53: }
54:
55: int main ()
56: {
57:    list <ContactItem> Contacts;
58:    Contacts.push_back(ContactItem("Jack Welsch", "+1 7889 879 879"));
59:    Contacts.push_back(ContactItem("Bill Gates", "+1 97 7897 8799 8"));
60:    Contacts.push_back(ContactItem("Angela Merkel", "+49 23456 5466"));
61:    Contacts.push_back(ContactItem("Vladimir Putin", "+7 6645 4564 797"));
62:    Contacts.push_back(ContactItem("Manmohan Singh", "+91 234 4564 789"));
63:    Contacts.push_back(ContactItem("Barack Obama", "+1 745 641 314"));
64:
65:    cout << "Lista w kolejności początkowej: " << endl;
```

```
66:    DisplayContents(Contacts);
67:
68:    Contacts.sort();
69:    cout << "Po sortowaniu w kolejności alfabetycznej przy użyciu
       ↪operatora <: " << endl;
70:    DisplayContents(Contacts);
71:
72:    Contacts.sort(SortOnPhoneNumber);
73:    cout << "Po sortowaniu według numeru telefonu przy użyciu predykatu: "<<
       ↪endl;
74:    DisplayContents(Contacts);
75:
76:    cout << "Po usunięciu Putina z listy: ";
77:    Contacts.remove(ContactItem("Vladimir Putin", ""));
78:    DisplayContents(Contacts);
79:
81: }
```

Wynik ▼

```
Lista w kolejności początkowej:
Jack Welsch: +1 7889 879 879
Bill Gates: +1 97 7897 8799 8
Angela Merkel: +49 23456 5466
Vladimir Putin: +7 6645 4564 797
Manmohan Singh: +91 234 4564 789
Barack Obama: +1 745 641 314

Po sortowaniu w kolejności alfabetycznej przy użyciu operatora <:
Angela Merkel: +49 23456 5466
Barack Obama: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
Jack Welsch: +1 7889 879 879
Manmohan Singh: +91 234 4564 789
Vladimir Putin: +7 6645 4564 797

Po sortowaniu według numeru telefonu przy użyciu predykatu:
Barack Obama: +1 745 641 314
Jack Welsch: +1 7889 879 879
Bill Gates: +1 97 7897 8799 8
Angela Merkel: +49 23456 5466
Vladimir Putin: +7 6645 4564 797
Manmohan Singh: +91 234 4564 789

Po usunięciu Putina z listy:
Barack Obama: +1 745 641 314
```

```
Jack Welsch: +1 7889 879 879
Bill Gates: +1 97 7897 8799 8
Angela Merkel: +49 23456 5466
Manmohan Singh: +91 234 4564 789
```

Analiza ▼

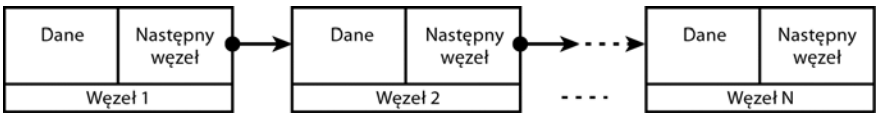
Na początek skoncentrujemy się na wierszach od 57. do 81. w funkcji `main()`. Utworzona zostaje lista elementów książki adresowej, które są typu `ContactItem` (patrz wiersz 57.). W wierszach od 58. do 63. następuje dodanie kilku elementów wraz z nieprawdziwymi numerami telefonów. Dane w książce adresowej prezentują kilka znanych osób, zawartość książki wyświetla polecenie w wierszu 66. W wierszu 68. zostaje użyta metoda `list::sort` bez funkcji predykatu. Z powodu braku predykatu ta funkcja sortująca szuka w strukturze `ContactItem` operatora < (został zdefiniowany w wierszach od 37. do 40.). Operator `ContactItem::operator<` pomaga `list::sort` w sortowaniu elementów w kolejności alfabetycznej (a nie numerów telefonów lub w zupełnie losowej kolejności). W celu posortowania listy w kolejności numerów telefonów konieczne jest użycie `list::sort()` i dostarczenie funkcji predykatu binarnego `SortOnPhoneNumber()`, jak pokazano w wierszu 72. Wymieniona funkcja została zaimplementowana w wierszach od 49. do 53. i gwarantuje, że argumenty danych wejściowych typu `ContactItem` są porównywane ze sobą na podstawie numerów telefonów, a nie nazwisk. Dlatego też `list::sort` pomaga w sortowaniu listy osób na podstawie ich numerów telefonów. Wreszcie, w wierszu 77. następuje użycie metody `list::remove()` do usunięcia elementu z książki adresowej. Jako parametr podawany jest element przeznaczony do usunięcia. Metoda `list::remove()` porównuje dany element z innymi elementami listy, używając zaimplementowanego w wierszach od 30. do 34. operatora `ContactItem::operator=`. Operator ten zwraca wartość `true` po dopasowaniu nazwisk i pomaga metodzie `list::remove()` w decyzji dotyczącej kryteriów dopasowania.

W powyższym przykładowym programie zaprezentowano nie tylko sposób, w jaki bazująca na wzorcu STL lista połączona może być używania w celu utworzenia obiektu `list` przechowującego obiekty dowolnego typu, ale również ogromną wagę operatorów i predykatów.

C++11

std::forward_list

W standardzie C++11 wprowadzono możliwość użycia `forward_list` zamiast listy dwukierunkowej `std::list`. Obiekt `std::forward_list` to lista jednokierunkowa, czyli pozwalająca na prowadzenie iteracji tylko w jednym kierunku, jak pokazano na rysunku 18.2.



RYSUNEK 18.2. Graficzne przedstawienie listy jednokierunkowej

Wskazówka

W celu użycia klasy `std::forward_list` w programie trzeba umieścić następujący nagłówek:

```
#include <forward_list>
```

Sposób użycia `forward_list` jest bardzo podobny do użycia `list`, ale iterator może poruszać się tylko w jednym kierunku. Dysponujesz metodą `push_front()` do wstawiania elementów na początku listy, nie masz natomiast metody `push_back()`. Oczywiście, zawsze można użyć metody `insert()` i jej przeciążonych wersji w celu wstawienia elementu w wybranym miejscu listy.

Użycie pewnych metod klasy `forward_list` zaprezentowano w listingu 18.8.

Listing 18.8. Podstawowe operacje wstawiania i usuwania elementów w klasie forward_list

```
 0: #include<forward_list>
 1: #include<iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents (const T& Input)
 6: {
 7:    for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
 8:       ; iElement != Input.cend ()
 9:       ; ++ iElement )
10:       cout << *iElement << ' ';
11:
12:    cout << endl;
```

```
13: }
14:
15: int main()
16: {
17:    forward_list<int> flistIntegers;
18:    flistIntegers.push_front(0);
19:    flistIntegers.push_front(2);
20:    flistIntegers.push_front(2);
21:    flistIntegers.push_front(4);
22:    flistIntegers.push_front(3);
23:    flistIntegers.push_front(1);
24:
25:    cout << "Zawartość obiektu forward_list: " << endl;
26:    DisplayContents(flistIntegers);
27:
28:    flistIntegers.remove(2);
29:    flistIntegers.sort();
30:    cout << "Zawartość obiektu po usunięciu wartości 2 i posortowaniu: " <<
       ↪endl;
31:    DisplayContents(flistIntegers);
32:
33:    return 0;
34: }
```

Wynik ▼

```
Zawartość obiektu forward_list:
1 3 4 2 2 0
Zawartość obiektu po usunięciu wartości 2 i posortowaniu:
0 1 3 4
```

Analiza ▼

Jak pokazano w przykładzie, klasa `forward_list` jest całkiem podobna do klasy `list`. Nie obsługuje iteracji w dwóch kierunkach, pozwala na użycie operatora `++`, ale już nie operatora `--`. W programie pokazano, że użycie funkcji `remove(2)` w wierszu 28. powoduje usunięcie wszystkich elementów o wartości 2. W wierszu 29. zastosowano funkcję `sort()` wraz z domyślnym predykatem sortowania stosującym `std::less<T>`.

Ponieważ klasa `forward_list` jest listą jednokierunkową, zużywa nieco mniej pamięci niż klasa `list` (poszczególne elementy muszą znać tylko element kolejny, a nie także poprzedni).

TAK	NIE
Wybieraj std::list zamiast std::vector, jeśli musisz często wstawiać lub usuwać elementy, zwłaszcza w środku listy. Wektor wymaga zmiany wewnętrznego bufora, aby korzystać z semantyki tablicy, a ponadto powoduje przeprowadzanie kosztowych operacji kopiowania. Z drugiej strony, lista po prostu łączy elementy lub ich nie łączy. Pamiętaj o możliwości wstawiania elementu na początku lub końcu listy przy użyciu metod (odpowiednio) push_front() i push_back(). Pamiętaj o utworzeniu implementacji operatorów < i == w klasach, które będą kolekcją w kontenerze STL (np. list). Ma to na celu zapewnienie domyślnego sortowania lub predykatu usuwania. Pamiętaj, że liczbę elementów listy można określić przy użyciu metody list::size(), podobnie jak w pozostałych klasach kontenerów STL. Pamiętaj o możliwości opróżnienia listy przy użyciu metody list::clear(), podobnie jak w pozostałych klasach kontenerów STL.	Nie używaj listy w rzadkich operacjach wstawiania i usuwania elementów na początku lub końcu kolekcji bądź też gdy nie trzeba wstawiać lub usuwać elementów w środku listy. W takich przypadkach klasy vector lub deque zapewniają lepszą wydajność. Nie zapomnij o zapewnieniu funkcji sortowania, jeśli chcesz sortować (sort()) listę lub usuwać (remove()) elementy, stosując kryteria inne niż domyślne.

Podsumowanie

W tej lekcji poznałeś właściwości obiektu `list` oraz różne operacje przeprowadzane przez ten obiekt. Teraz już znasz niektóre najbardziej użyteczne funkcje obiektu `list` i potrafisz tworzyć obiekty `list` przechowujące obiekty dowolnego typu.

Pytania i odpowiedzi

Pytanie: Dlaczego obiekt `list` dostarcza funkcje składowe, takich jak `sort()` i `remove()`?

Odpowiedź: Klasa STL `list` respektuje właściwość polegającą na tym, że iteratory wskazujące elementy obiektu `list` powinny pozostać prawidłowe bez względu na położenie tych elementów w samym obiekcie `list`. Wprawdzie algorytmy STL działają również w obiekcie `list`, jednak funkcje składowe obiektu `list` gwarantują, że wspomniana powyżej cecha obiektu `list` będzie respektowana. W ten sposób iteratory wskazujące elementy w obiekcie `list` przed operacją sortowania po zakończeniu tej operacji nadal będą wskazywały te same elementy.

Pytanie: Używam obiektu `list` typu `CAnimal`, który jest klasą. Które operatory klasy `CAnimal` powinny zostać zdefiniowane dla funkcji składowych obiektu `list`, aby umożliwić prawidłową pracę?

Odpowiedź: Każdej klasie, która będzie używana w kontenerach STL, trzeba dostarczyć domyślny operator porównywania == oraz domyślny operator <.

Pytanie: W jaki sposób zastąpić słowo kluczowe auto przez ściśle określoną deklarację typu w poniższym wierszu?

```
list<int> listIntegers(10);  // Lista dziesięciu liczb całkowitych.
auto iFirstElement = listIntegers.begin();
```

Odpowiedź: Jeżeli używasz starszej wersji kompilatora, który jest niezgodny ze standardem C++11, wtedy słowo kluczowe `auto` powinieneś zastąpić deklaracją zawierającą ściśle podany typ, np.:

```
list<int> listIntegers(10);  // Lista dziesięciu liczb całkowitych.
list<int>::iterator iFirstElement = listIntegers.begin();
```

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Czy w porównaniu do wstawiania elementów na początku lub na końcu obiektu STL `list` nastąpi duży spadek wydajności podczas wstawiania elementów do środka obiektu STL `list`?
2. Dwa iteratory wskazują dwa elementy obiektu STL `list`. Następnie między te elementy zostaje wstawiony inny element. Czy operacja wstawiania spowoduje unieważnienie iteratorów?
3. W jaki sposób można wyczyścić zawartość `std::list`?
4. Czy jest możliwe wstawienie wielu elementów do obiektu `list`?

Ćwiczenia

1. Napisz krótki program akceptujący liczby podane przez użytkownika, a następnie wstawiający je na początku obiektu `list`.
2. Używając krótkiego programu, zademonstruj, że iterator wskazujący element w obiekcie `list` pozostaje poprawny nawet po wstawieniu innego elementu przed elementem wskazanym przez dany iterator, mimo że operacja ta powoduje zmianę względnej pozycji elementu istniejącego już w obiekcie `list`.
3. Napisz program, który wstawia w obiekcie STL `list` zawartość obiektu `vector` za pomocą funkcji wstawiania obiektu `list`.
4. Napisz program, który sortuje i odwraca ciągi tekstowe przechowywane w obiekcie `list`.

Lekcja 19

Klasy STL set

Standardowa biblioteka wzorców (STL) dostarcza programiście klasy kontenerów, które pomagają w tworzeniu aplikacji wymagających częstego przeprowadzania szybkich operacji wyszukiwania. Klasy `std::set` i `std::multiset` są używane do przechowywania posortowanych kolekcji elementów i oferują możliwość wyszukiwania elementów w kontenerze o złożoności logarytmicznej. Odpowiedniki wymienionych klas, ale przechowujące nieposortowane elementy, oferują stały czas wstawiania i wyszukiwania elementów.

Z tej lekcji dowiesz się:

- **wprowadzenie do klas STL set, multiset, unordered_set i unordered_multiset,**
- **podstawowe operacje wstawiania, usuwania i wyszukiwania elementów,**
- **wady i zalety używania wymienionych kontenerów.**

Wprowadzenie do klas STL set

Set i multiset to kontenery ułatwiające szybkie wyszukiwanie przechowywanych w nich kluczy, tzn. klucze są wartościami przechowywanymi w jednowymiarowym kontenerze. Różnica między set i multiset polega na tym, że ten drugi pozwala na używanie powtarzających się wartości, podczas gdy ten pierwszy może przechowywać jedynie wartości unikalne.

Rysunek 19.1 to tylko demonstracja i ma za zadanie pokazać, że nazwy w obiekcie set są unikalne, podczas gdy obiekt multiset pozwala na duplikaty. Kontenery STL są ogólnymi wzorcami klas i dlatego mogą przechowywać ciągi tekstowe w dokładnie taki sam sposób jak liczby całkowite, struktury i klasy — to zależy od sposobu ustanowienia wzorca.

RYSUNEK 19.1. Graficzne przedstawienie nazw stosowanych w obiektach set i multiset

W celu ułatwienia szybkiego wyszukiwania implementacje obiektów STL set i multiset wewnętrznie są podobne do drzewa binarnego. Oznacza to, że elementy wstawione do obiektów set i multiset są sortowane w kolejności ich wstawiania, co przyśpiesza operacje wyszukiwania. Oznacza to również, że w przeciwieństwie do obiektu vector, w którym element w danym położeniu może być zastąpiony przez inny, w obiekcie set element w danym położeniu nie może być zastąpiony przez nowy element o innej wartości. Dzieje się tak, ponieważ obiekt set umieści go w innym położeniu, zgodnie z jego wartością, stosownie do elementów znajdujących się w wewnętrznym drzewie.

Wskazówka

W celu użycia klas std::set lub std::multiset w programie trzeba umieścić następujący nagłówek:

```
#include <set>
```

Podstawowe operacje klas STL set i multiset

STL `set` i `multiset` to klasy wzorców, które muszą być ustanowione, zanim będzie można przystąpić do używania ich funkcji składowych.

Ustanawianie obiektu std::set

Ustanowienie obiektu `set` lub `multiset` przechowującego liczby całkowite wymaga przeprowadzenia specjalizacji klasy wzorca `std::set` lub `std::multiset` dla odpowiedniego typu:

```
std::set <int> setIntegers;
std::multiset <int> msetIntegers;
```

Aby zdefiniować kontener `set` lub `multiset` przechowujący obiekty typu `Tuna`, konieczne jest użycie poniższych poleceń:

```
std::set <Tuna> setIntegers;
std::multiset <Tuna> msetIntegers;
```

Natomiast w celu zadeklarowania iteratora prowadzącego do elementu w kontenerze `set` lub `multiset` należy użyć poniższego fragmentu kodu:

```
std::set<int>::const_iterator iElementInSet;
std::multiset<int>::const_iterator iElementInMultiset;
```

Jeżeli potrzebujesz iteratora, który może być używany do modyfikacji wartości lub wywoływania funkcji innych niż `const`, w przedstawionym poleceniu musisz zastosować `iterator` zamiast `const_iterator`.

Ponieważ `set` i `multiset` są kontenerami sortującymi elementy w trakcie ich wstawiania, używają predykatu `std::less`, jeśli nie podasz kryteriów sortowania. To gwarantuje posortowanie elementów kontenera w kolejności rosnącej.

Utworzenie binarnego predykatu sortowania następuje przez zdefiniowanie klasy z operatorem `()` pobierającym jako dane wejściowe dwie wartości typu przechowywanego w kontenerze. W zależności od kryteriów, wartością zwrotną operatora jest `true`. W poniższym fragmencie kodu pokazano predykat sortowania, który przeprowadza sortowanie w kolejności malejącej:

```
// Poniższego fragmentu kodu można użyć w charakterze parametru wzorca podczas
// tworzenia egzemplarzy set lub multiset.
template <typename T>
struct SortDescending
{
   bool operator()(const T& lhs, const T& rhs) const
   {
      return (lhs > rhs);
   }
};
```

Następnie przygotowany predykat można wykorzystać w trakcie tworzenia egzemplarza `set` lub `multiset`:

```
// Kontener set lub multiset przechowujący liczby całkowite (użyty jest predykat sortowania).
set <int, SortDescending<int> > setIntegers;
multiset <int, SortDescending<int> > msetIntegers;
```

Kontenery `set` i `multiset`, poza przedstawionymi wariantami, zawsze można utworzyć na podstawie wartości pochodzących z innego kontenera lub podanego zakresu, tak jak przedstawiono w listingu 19.1.

Listing 19.1. Różne techniki tworzenia obiektów set lub multiset

```
 0: #include <set>
 1:
 2: // Poniższego fragmentu kodu można użyć w charakterze parametru wzorca podczas
    ↪tworzenia egzemplarzy set lub multiset.
 3: template <typename T>
 4: struct SortDescending
 5: {
 6:    bool operator()(const T& lhs, const T& rhs) const
 7:    {
 8:       return (lhs > rhs);
 9:    }
10: };
11:
12: int main ()
13: {
14:    using namespace std;
15:
16:    // Proste obiekty set i multiset przechowujące liczby całkowite (używają domyślnego
       ↪predykatu sortowania).
17:    set <int> setIntegers1;
18:    multiset <int> msetIntegers1;
19:
20:    // Obiekty set i multiset utworzone wraz ze zdefiniowanym przez programistę
       ↪predykatem sortowania.
```

```
21:    set<int, SortDescending<int> > setIntegers2;
22:    multiset<int, SortDescending<int> > msetIntegers2;
23:
24:    // Utworzenie jednego kontenera na podstawie innego lub jego części.
25:    set<int> setIntegers3(setIntegers1);
26:    multiset<int> msetIntegers3(setIntegers1.cbegin(),
       ↪setIntegers1.cend());
27:
28:    return 0;
29: }
```

Analiza ▼

Program nie generuje żadnych danych wyjściowych, ale pokazuje różne techniki tworzenia obiektów `set` i `multiset` specjalizowanych dla typu `int`. W wierszach 17. i 18. mamy najprostszą formę, w której zostały zignorowane parametry wzorca inne niż typ. Z tego powodu wykorzystany będzie domyślny predykat sortowania, zgodnie z jego implementacją w strukturze (klasie) `std::less<T>`. Jeżeli chcesz nadpisać domyślny rodzaj sortowania, musisz przygotować odpowiedni predykat, np. taki jak zdefiniowany w wierszach od 3. do 10. i użyty w wierszach 21. i 22. funkcji `main()`. Zdefiniowany predykat przeprowadza sortowanie w kolejności malejącej (domyślne sortowanie jest w kolejności rosnącej). Wreszcie, w wierszach 25. i 26. pokazano rozwiązanie, gdzie tworzony kontener jest kopią innego, natomiast obiekt `multiset` powstaje na podstawie zakresu wartości pobranych z innego kontenera `set` (to również może być obiekt `vector`, `list` lub inny kontener klasy STL, który zwraca iteratory i określa granice za pomocą metod `cbegin()` oraz `cend()`).

Wskazówka

Czy użycie metod `cbegin()` i `cend()` powoduje błędy w trakcie kompilacji programu?

Jeżeli przedstawiony program próbujesz skompilować przy użyciu kompilatora niezgodnego ze standardem C++11, wtedy zamiast metod `cbegin()` i `cend()` użyj (odpowiednio) `begin()` i `end()`. Oferowane przez standard C++11 metody `cbegin()` i `cend()` są użyteczne, ponieważ zwracają iterator `const`, który nie może być wykorzystany do modyfikacji elementów.

Wstawianie elementów do obiektów set lub multiset

Większość funkcji obiektów `set` i `multiset` działa w bardzo podobny sposób. Akceptują podobne parametry i zwracają wartości podobnych typów. Przykładowo w celu wstawienia elementów do obu rodzajów kontenerów można wykorzystać funkcję składową `insert()`, która przyjmuje wartość przeznaczoną do wstawienia:

```
setIntegers.insert (-1);
msetIntegers.insert (setIntegers.begin (), setIntegers.end ());
```

Wstawianie elementów do wymienionych kontenerów zaprezentowano w listingu 19.2.

Listing 19.2. Wstawianie elementów do obiektów STL set i multiset

```
 0: #include <set>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents(const T& Input)
 6: {
 7:     for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
 8:         ; iElement != Input.cend() // cend() to nowość w C++11.
 9:         ; ++ iElement )
10:        cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     set <int> setIntegers;
18:     multiset <int> msetIntegers;
19:
20:     setIntegers.insert (60);
21:     setIntegers.insert (-1);
22:     setIntegers.insert (3000);
23:     cout << "Wyświetlenie na ekranie zawartości obiektu set" << endl;
24:     DisplayContents(setIntegers);
25:
26:     msetIntegers.insert (setIntegers.begin (), setIntegers.end ());
27:     msetIntegers.insert (3000);
28:
```

```
29:     cout << "Wyświetlenie na ekranie zawartości obiektu multiset" <<
        ↪endl;
30:     DisplayContents(msetIntegers);
31:
32:     cout << "Liczba wystąpień elementu '3000' w obiekcie multiset
        ↪wynosi: '";
33:     cout << msetIntegers.count (3000) << "'" << endl;
34:
35:     return 0;
36: }
```

Wynik ▼

```
Wyświetlenie na ekranie zawartości obiektu set
-1 60 3000
Wyświetlenie na ekranie zawartości obiektu multiset
-1 60 3000 3000
Liczba wystąpień elementu '3000' w obiekcie multiset wynosi: '2'
```

Analiza ▼

W wierszach od 4. do 13. znajduje się deklaracja funkcji wzorca `DisplayContents()`, która została użyta również w lekcjach 17. oraz 18. i powoduje wyświetlenie na ekranie zawartości kontenera STL. W wierszach 17. i 18. zdefiniowano obiekty `set` i `multiset`, które już znasz. Natomiast w wierszach od 20. do 22. do obiektu `set` wstawiono wartości, używając do tego funkcji składowej `insert()`. W wierszu 26. pokazano, jak funkcję `insert()` można wykorzystać do wstawienia zawartości jednego kontenera (`set`) w inny (`multiset`). W omawianym przykładzie oznacza to wstawienie zawartości `setIntegers` do obiektu `multiset` o nazwie `msetIntegers`. Po wstawieniu zawartości obiektu `set` w `multiset` w wierszu 27. kod powoduje wstawienie elementu z wartością 3000, która już istnieje w obiekcie `multiset`. Dane wyjściowe programu pokazują jednak, że obiekt `multiset` może przechowywać wiele takich samych wartości. W wierszach 25. i 26. pokazano użyteczność funkcji składowej `multiset::count()`, która zwraca liczbę elementów obiektu `multiset` przechowujących określoną wartość.

Wskazówka

Metody `multiset::count()` używaj do ustalenia liczby elementów obiektu `multiset`, które mają taką samą wartość jak podana jako argument wymienionej metody.

Wskazówka

Czy użycie słowa kluczowego `auto` powoduje błędy w trakcie kompilacji?

Przedstawiona w listingu 19.2 funkcja `DisplayContents()` używa wprowadzonego w standardzie C++11 słowa kluczowego `auto` do zdefiniowania typu iteratora (patrz wiersz 7.). Ponadto kod zawiera wywołania metod `cbegin()` i `cend()`, które są nowością w standardzie C++11 i zwracają wartość typu `const_iterator`.

Aby ten i kolejne przykłady skompilować w starszej wersji kompilatora, nieobsługującej standardu C++11, musisz słowo kluczowe `auto` zastąpić konkretnym typem.

Dlatego też metoda `DisplayContent()` dla starszej wersji kompilatora będzie miała następującą postać:

```
template <typename T>
void DisplayContent(const T& Input)
{
   for (T::const_iterator iElement = Input.begin() // Ściśle określony typ.
      ; iElement != Input.end ()
      ; ++ iElement )
     cout << *iElement << ' ';

   cout << endl;
}
```

Wyszukiwanie elementów w obiektach STL set lub multiset

Kontenery asocjacyjne, takie jak `set`, `multiset`, `map` i `multimap`, zawierają funkcję `find()` — jest to metoda składowa pozwalająca na wyszukanie wartości podanego klucza:

```
auto iElementFound = setIntegers.find (-1);
// Czy znaleziono element?
if (iElementFound != setIntegers.end ())
   cout << "Element " << *iElementFound << " został znaleziony!" << endl;
else
   cout << "Element nie został znaleziony!" << endl;
```

Przykład użycia metody `find()` zademonstrowano w listingu 19.3. Dla obiektu `multiset` wymieniona funkcja wyszukuje pierwszą wartość, która będzie dopasowana do podanego klucza.

Listing 19.3. Używanie funkcji składowej find()

```
 0: #include <set>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main ()
 5: {
 6:     set<int> setIntegers;
 7:
 8:     // Wstawienie kilku losowo wybranych wartości.
 9:     setIntegers.insert (43);
10:     setIntegers.insert (78);
11:     setIntegers.insert (-1);
12:     setIntegers.insert (124);
13:
14:     // Wyświetlenie na ekranie zawartości obiektu.
15:     for (auto iElement = setIntegers.cbegin ()
16:         ; iElement != setIntegers.cend ()
17:         ; ++ iElement )
18:         cout << *iElement << endl;
19:
20:     // Próba wyszukania elementu.
21:     auto iElementFound = setIntegers.find (-1);
22:
23:     // Sprawdzenie, czy element został znaleziony...
24:     if (iElementFound != setIntegers.end ())
25:         cout << "Element " << *iElementFound << " został znaleziony!" <<
          ↪endl;
26:     else
27:         cout << "Element nie został znaleziony w obiekcie set!" <<
          ↪endl;
28:
29:     // Próba wyszukania innego elementu.
30:     auto iAnotherFind = setIntegers.find (12345);
31:
32:     // Sprawdzenie, czy element został znaleziony...
33:     if (iAnotherFind != setIntegers.end ())
34:         cout << "Element " << *iAnotherFind << " został znaleziony!" <<
          ↪endl;
35:     else
36:         cout << "Element 12345 nie został znaleziony w obiekcie set!" <<
          ↪endl;
37:
38:     return 0;
39: }
```

Wynik ▼

```
-1
43
78
124
Element -1 został znaleziony!
Element 12345 nie został znaleziony w obiekcie set!
```

Analiza ▼

W wierszach od 21. do 27. możemy zobaczyć sposób użycia funkcji składowej `find()`. Funkcja `find()` zwraca iterator, który musi być porównany z wartością zwracaną przez funkcję `end()`, jak pokazano w wierszu 24. Celem tego porównania jest sprawdzenie, czy dany element został znaleziony. Jeżeli iterator jest prawidłowy, do wskazywanej przez niego wartości można uzyskać dostęp przy użyciu `*iElementFound`.

Uwaga

> Przykład przedstawiony w listingu 19.3 będzie działał prawidłowo również w przypadku obiektu `multiset`. Oznacza to, że jeżeli w wierszu 6. będzie użyty obiekt `multiset` zamiast `set`, taka modyfikacja nie spowoduje zmiany sposobu działania aplikacji.

Usuwanie elementów z obiektów STL set lub multiset

Kontenery asocjacyjne, takie jak `set`, `multiset`, `map` i `multimap`, zawierają funkcję składową `erase()`, która pozwala na usunięcie wartości podanego klucza:

```
setObject.erase (klucz);
```

Inna postać funkcji `erase()` umożliwia usunięcie określonego elementu na podstawie prowadzącego do niego iteratora:

```
setObject.erase (iElement);
```

Istnieje również możliwość usunięcia zakresu elementów z obiektów `set` lub `multiset` przy użyciu iteratorów, które wskazują granice tego zakresu:

```
setObject.erase (iPoczatekZakresu, iKoniecZakresu);
```

W kodzie przedstawionym w listingu 19.4 zademonstrowano użycie funkcji `erase()` w celu usunięcia elementów z obiektu `set` i `multiset`.

Listing 19.4. Używanie funkcji składowej erase() w obiekcie multiset

```
 0: #include <set>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents(const T& Input)
 6: {
 7:     for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
 8:         ; iElement != Input.cend() // cend() to nowość w C++11.
 9:         ; ++ iElement )
10:        cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: typedef multiset <int> MSETINT;
16:
17: int main ()
18: {
19:     MSETINT msetIntegers;
20:
21:     // Wstawienie kilku losowo wybranych wartości.
22:     msetIntegers.insert (43);
23:     msetIntegers.insert (78);
24:     msetIntegers.insert (78);     // Duplikat.
25:     msetIntegers.insert (-1);
26:     msetIntegers.insert (124);
27:
28:     cout << "Obiekt multiset zawiera " << msetIntegers.size () <<
        ↪" element(y/ów).";
29:     cout << " Są nimi: " << endl;
30:
31:     // Wyświetlenie na ekranie zawartości obiektu multiset.
32:     DisplayContents(msetIntegers);
33:
34:     cout << "Proszę podać liczbę do usunięcia z obiektu set" << endl;
35:     int nNumberToErase = 0;
36:     cin >> nNumberToErase;
37:
38:     cout << "Usuwanie " << msetIntegers.count (nNumberToErase);
39:     cout << " egzemplarzy wartości " << nNumberToErase << endl;
40:
41:     // Próba wyszukania elementu.
42:     msetIntegers.erase (nNumberToErase);
43:
44:     cout << "Obiekt multiset zawiera " << msetIntegers.size () <<
        ↪" element(y/ów).";
45:     cout << " Są nimi: " << endl;
```

```
46:     DisplayContents(msetIntegers);
47:
48:     return 0;
49: }
```

Wynik ▼

```
Obiekt multiset zawiera 5 element(y/ów). Są nimi:
-1 43 78 78 124
Proszę podać liczbę do usunięcia z obiektu set
78
Usuwanie 2 egzemplarzy wartości 78
Obiekt multiset zawiera 3 element(y/ów). Są nimi:
-1 43 124
```

Analiza ▼

Zwróć uwagę na użycie typedef w wierszu 15. Z kolei w wierszu 38. zaprezentowano użycie metody `count()` w celu ustalenia liczby elementów o podanej wartości. Rzeczywista operacja usunięcia jest przeprowadzana w wierszu 42., w którym następuje usunięcie wszystkich elementów dopasowanych do podanej wartości.

Zwróć uwagę na fakt, że funkcja `erase()` jest przeciążona. Istnieje możliwość wywołania metody `erase()` względem iteratora, powiedzmy zwróconego przez funkcję `find()`, w celu usunięcia elementu zawierającego znalezioną wartość. Oto przykład:

```
MSETINT::iterator iElementFound = msetIntegers.find (nNumberToErase);
if (iElementFound != msetIntegers.end ())
   msetIntegers.erase (iElementFound);
else
   cout << "Element nie został znaleziony!" << endl;
```

Podobnie funkcję `erase()` można także wykorzystać do usunięcia zakresu elementów z obiektu `multiset`:

```
MSETINT::iterator iElementFound = msetIntegers.find (nValue);
if (iElementFound != msetIntegers.end ())
   msetIntegers.erase (msetIntegers.begin (), iElementFound);
```

Powyższy fragment kodu spowoduje usunięcie wszystkich elementów, od początku aż do elementu o wartości `nValue`, ale bez tej wartości. Zawartość zarówno obiektu `set`, jak i `multiset` może być opróżniona za pomocą funkcji składowej `clear()`.

Teraz, po przedstawieniu ogólnego opisu podstawowych funkcji obiektów `set` i `multiset`, warto zapoznać się z przykładem, który będzie praktyczną aplikacją używającą tej klasy kontenera. Program w listingu 19.5 to prosta implementacja książki telefonicznej bazującej na menu. Aplikacja pozwala użytkownikowi na wstawianie imion i numerów telefonów oraz ich wyszukiwanie, usuwanie i wyświetlanie.

Listing 19.5. Książka telefoniczna wykorzystująca obiekt STL set oraz funkcje find() i erase()

```
#include <set>
#include <iostream>
#include <string>
using namespace std;

template <typename T>
void DisplayContents(const T& Input)
{
    for (auto iElement = Input.cbegin() // auto i cbegin() dla C++11.
        ; iElement != Input.cend() // cend() to nowość w C++11.
        ; ++ iElement )
       cout << *iElement << endl;

    cout << endl;
}

struct ContactItem
{
    string strContactsName;
    string strPhoneNumber;
    string strDisplayRepresentation;

    // Konstruktor i destruktor.
    ContactItem (const string& strName, const string & strNumber)
    {
        strContactsName = strName;
        strPhoneNumber = strNumber;
        strDisplayRepresentation = (strContactsName + ":
        ↪" + strPhoneNumber);
    }

    // Operator używany przez std::find().
    bool operator == (const ContactItem& itemToCompare) const
    {
        return (itemToCompare.strContactsName == this->strContactsName);
    }

```

```
36:      // Operator używany jako predykat sortowania…
37:      bool operator < (const ContactItem& itemToCompare) const
38:      {
39:          return (this->strContactsName < itemToCompare.strContactsName);
40:      }
41:
42:     // Operator używany w DisplayContents() za pomocą polecenia cout.
43:     operator const char*() const
44:     {
45:       return strDisplayRepresentation.c_str();
46:     }
47: };
48:
49: int main ()
50: {
51:     set<ContactItem> setContacts;
52:     setContacts.insert(ContactItem("Jack Welsch", "+1 7889 879 879"));
53:     setContacts.insert(ContactItem("Bill Gates", "+1 97 7897 8799 8"));
54:     setContacts.insert(ContactItem("Angela Merkel", "+49 23456 5466"));
55:     setContacts.insert(ContactItem("Vladimir Putin", "+7 6645 4564 797"));
56:     setContacts.insert(ContactItem("Manmohan Singh", "+91 234 4564 789"));
57:     setContacts.insert(ContactItem("Barack Obama", "+1 745 641 314"));
58:     DisplayContents(setContacts);
59:
60:     cout << "Podaj osobę, której numer telefonu chcesz usunąć: ";
61:     string NameInput;
62:     getline(cin, NameInput);
63:
64:     auto iContactFound = setContacts.find(ContactItem(NameInput, ""));
65:     if(iContactFound != setContacts.end())
66:     {
67:        // Usunięcie elementu znalezionego w kolekcji.
68:        setContacts.erase(iContactFound);
69:        cout << "Wyświetlenie zawartości po operacji usunięcia: " <<
          ↪NameInput << endl;
70:        DisplayContents(setContacts);
71:     }
72:     else
73:        cout << "Nie znaleziono elementu" << endl;
74:
75:     return 0;
76: }
```

Wynik ▼

```
Angela Merkel: +49 23456 5466
Barack Obama: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
```

```
Jack Welsch: +1 7889 879 879
Manmohan Singh: +91 234 4564 789
Vladimir Putin: +7 6645 4564 797

Podaj osobę, której numer telefonu chcesz usunąć: Jack Welsch
Wyświetlenie zawartości po operacji usunięcia: Jack Welsch
Angela Merkel: +49 23456 5466
Barack Obama: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
Manmohan Singh: +91 234 4564 789
Vladimir Putin: +7 6645 4564 797
```

Analiza ▼

Przedstawiony przykład jest bardzo podobny do pokazanego w listingu 18.7 i sortującego obiekt `std::list` w kolejności alfabetycznej. W omawianym programie użyto obiektu `std::set`, a sortowanie jest przeprowadzane w chwili wstawiania elementu. Jak można zobaczyć w wyświetlonych danych wyjściowych, nie trzeba wywoływać żadnej funkcji sortującej, ponieważ elementy obiektu `set` są sortowane w trakcie ich wstawiania. Użytkownik otrzymuje możliwość usunięcia dowolnego numeru telefonu. W wierszu 64. zademonstrowano wywołanie metody `find()` w celu odszukania elementu, który następnie w wierszu 68. zostanie usunięty przy użyciu metody `erase()`.

Wskazówka

Przedstawiona implementacja książki telefonicznej bazuje na obiekcie `set`, a tym samym nie pozwala na używanie wielu wpisów posiadających taką samą wartość. Jeżeli trzeba zaimplementować książkę telefoniczną, w której można przechowywać osoby o takim samym imieniu (np. Tomek), należy wybrać obiekt STL `multiset`. Powyższy kod programu nadal będzie działał, jeśli obiekt `setContact` będzie typu `multiset`. Aby następnie użyć pojemności obiektu `multiset` w celu przechowywania wielu elementów o tej samej wartości, trzeba będzie wykorzystać funkcję składową `count()`, która poda liczbę elementów przechowujących określoną wartość. Takie rozwiązanie zademonstrowano w poprzednim fragmencie kodu. Podobne elementy są umieszczane w obiekcie `multiset` obok siebie, natomiast funkcja `find()` zwraca iterator prowadzący do pierwszej znalezionej wartości. Ten iterator może być inkrementowany przez ilość razy zwracaną przez funkcję `count()` w celu przejścia do kolejnych elementów.

Wady i zalety używania obiektów STL set i multiset

W aplikacjach, które muszą często przeprowadzać operacje wyszukiwania, obiekty STL `set` i `multiset` są bardzo użyteczne, ponieważ ich zawartość jest posortowana, a tym samym szybka do zlokalizowania. Aby jednak skorzystać z tej zalety, kontener musi sortować elementy w chwili ich wstawiania. To wiąże się z pewnym obciążeniem podczas wstawiania elementów, gdyż są od razu sortowane. Obciążenie to może być korzystnym kompromisem, jeśli w aplikacji trzeba użyć funkcji, takich jak `find()`.

Funkcja `find()` wykorzystuje wewnętrzną strukturę drzewa. Taka binarna struktura drzewa wiąże się z inną ukrytą wadą kontenerów sekwencyjnych, takich jak `vector`. W kontenerze `vector` element wskazywany przez iterator (powiedzmy, zwrócony przez operację `std::find()`) może być nadpisany nową wartością. Jednak w obiekcie `set` elementy są sortowane względem ich poszczególnych wartości, a więc nie należy nigdy nadpisywać elementu za pomocą iteratora, nawet jeśli będzie to możliwe.

C++11

Implementacje STL Hash set: std::unordered_set i std::unordered_multiset

Klasy `std::set` i `std::multiset` przeprowadzają sortowanie elementów (jednocześnie będących kluczami) na podstawie predykatu `std::less<T>` lub zdefiniowanego przez programistę. Operacja wyszukiwania w posortowanym kontenerze o złożoności logarytmicznej jest wykonywana szybciej niż w nieposortowanym, takim jak `std::vector`, a ponadto `std::sort`. Oznacza to, że czas wymagany na wyszukanie elementu w kontenerze `set` nie jest bezpośrednio związany z liczbą przechowywanych w nim elementów, a raczej stanowi ich logarytm. Dlatego też przeszukanie kontenera zawierającego 10000 elementów zwykle zajmuje dwa razy więcej czasu niż kontenera 100-elementowego (ponieważ 100^2 = 10000, log(10000) = 2x log(100)).

Jednak taka ogromna poprawa wydajności w stosunku do nieposortowanego kontenera (w którym czas wyszukiwania elementu wydłuża się proporcjonalnie do ich liczby) czasami okazuje się niewystarczająca.

Programiści i matematycy wolą rozwiązania pozwalające na zachowanie stałego czasu podczas wstawiania i wyszukiwania elementów. Jednym z takich rozwiązań jest użycie implementacji, w której funkcja hash jest stosowana w celu określenia indeksu sortowania. Elementy wstawiane do kontenera hash są najpierw poddawane działaniu funkcji hash generującej unikalny indeks wskazujący położenie danego elementu.

Biblioteka STL dostarcza klasę hash w postaci `std::unordered_set`.

Wskazówka

W celu użycia klas `std::unordered_set` lub `std::unordered_multiset` w programie trzeba umieścić następujący nagłówek:

```
#include <unordered_set>
```

Użycie wymienionej klasy nie różni się zbytnio od sposobu stosowania klasy `std::set`:

```
// Utworzenie egzemplarza:
unordered_set<int> usetInt;

// Wstawienie elementu.
usetInt.insert(1000);

// Użycie metody find():
auto iPairThousand = usetInt.find(1000);
if (iPairThousand != usetInt.end())
   cout << *iPairThousand << endl;
```

Bardzo ważną cechą obiektu `unordered_map` jest dostępność funkcji hash, która jest odpowiedzialna za kolejność sortowania:

```
unordered_set<int>::hasher HFn = usetInt.hash_function();
```

W listingu 19.6 zaprezentowano użycie najczęściej stosowanych metod dostarczanych przez `std::hash_set`.

Listing 19.6. Klasa std::unordered_set i użycie metod insert(), find(), size(), max_bucket_count(), load_factor() oraz max_load_factor()

```
 0: #include<unordered_set>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: template <typename T>
 5: void DisplayContents(const T& Input)
 6: {
 7:    cout << "Liczba elementów, size() = " << Input.size() << endl;
```

```
 8:    cout << "Maksymalna liczba kubełków = " << Input.max_bucket_count() <<
      ↪endl;
 9:    cout << "Współczynnik wypełniania: " << Input.load_factor() << endl;
10:    cout << "Maksymalny współczynnik wypełniania = " <<
      ↪Input.max_load_factor() << endl;
11:    cout << "Zawartość nieposortowanego obiektu: " << endl;
12:
13:    for(auto iElement = Input.cbegin() // auto, cbegin: C++11.
14:       ; iElement != Input.cend()   // cend() to nowość w C++11.
15:       ; ++ iElement )
16:       cout<< *iElement << ' ';
17:
18:    cout<< endl;
19: }
20:
21: int main()
22: {
23:    // Utworzenie obiektu unordered_set typu int:
24:    unordered_set<int> usetInt;
25:
26:    usetInt.insert(1000);
27:    usetInt.insert(-3);
28:    usetInt.insert(2011);
29:    usetInt.insert(300);
30:    usetInt.insert(-1000);
31:    usetInt.insert(989);
32:    usetInt.insert(-300);
33:    usetInt.insert(111);
34:    DisplayContents(usetInt);
35:    usetInt.insert(999);
36:    DisplayContents(usetInt);
37:
38:    // Metoda find():
39:    cout << "Podaj liczbę, której istnienie chcesz sprawdzić w kolekcji: ";
40:    int Key = 0;
41:    cin >> Key;
42:    auto iPairThousand = usetInt.find(Key);
43:
44:    if (iPairThousand != usetInt.end())
45:       cout << *iPairThousand << " znaleziono w kolekcji" << endl;
46:    else
47:       cout << Key << " nie znaleziono w kolekcji" << endl;
48:
49:    return 0;
50: }
```

Wynik ▼

```
Liczba elementów, size() = 8
Maksymalna liczba kubełków = 8
Współczynnik wypełniania: 1
Maksymalny współczynnik wypełniania = 1
Zawartość nieposortowanego obiektu:
1000 -3 2011 300 -1000 -300 989 111
Liczba elementów, size() = 9
Maksymalna liczba kubełków = 64
Współczynnik wypełniania: 0.140625
Maksymalny współczynnik wypełniania = 1
Zawartość nieposortowanego obiektu:
1000 -3 2011 300 -1000 -300 989 999 111
Podaj liczbę, której istnienie chcesz sprawdzić w kolekcji: -1000
-1000 znaleziono w kolekcji
```

Analiza ▼

W powyższym przykładzie tworzony jest obiekt `unordered_set` przechowujący osiem liczb całkowitych; następnie wyświetlona zostaje jego zawartość oraz kilka informacji statystycznych dostarczonych przez metody `max_bucket_count()`, `load_factor()` i `max_load_factor()`, co przedstawiono w wierszach od 8. do 10. Dane wyjściowe pokazują, że początkowa wielkość kubełka wynosi osiem, kontener zawiera osiem elementów, a więc wskaźnik wypełnienia wynosi 1, czyli ma wartość maksymalną. Po wstawieniu do obiektu `unordered_set` dziewiątego elementu następuje utworzenie 64 kubełków, ponowne zbudowanie tabeli hash, a wartość współczynnika wypełnienia zmniejsza się. Pozostała część kodu w funkcji `main()` pokazuje, że składnia wyszukiwania elementów w obiekcie `unordered_set` jest podobna do stosowanej w obiekcie `set`. Metoda `find()` zwraca iterator (patrz wiersz 42.), który trzeba sprawdzić, zanim będzie mógł zostać wykorzystany.

Uwaga

Ponieważ funkcja hash jest zwykle używana w tabeli hash do wyszukania wartości o podanym kluczu, zapoznaj się także z punktem poświęconym `std::unordered_map` w lekcji 20., zatytułowanej „Klasy STL map”.

`std::unordered_map` to wprowadzona w standardzie C++11 implementacja tabeli hash.

TAK	NIE
Pamiętaj, że kontenery STL set i multiset zostały zoptymalizowane do sytuacji, w której wymagane jest częste przeprowadzanie operacji wyszukiwania. Pamiętaj, że std::multiset pozwala na umieszczanie w kontenerze wielu elementów (kluczy) o takiej samej wartości, podczas gdy w std:set dozwolone są tylko wartości unikalne. Używaj metody multiset::count(wartość) w celu ustalenia liczby elementów o określonej wartości. Pamiętaj, że metody set::size() i multiset::size() podają liczbę elementów w kontenerze.	Nie zapominaj o implementacji operatorów < i == dla klas, których typy będą elementami przechowywanymi w kontenerach, takich jak set i multiset. Pierwszy z wymienionych staje się predykatem sortowania, podczas gdy drugi jest stosowany w funkcjach, takich jak set::find(). Nie używaj std::set lub std::multiset w sytuacjach, gdy trzeba często przeprowadzać operacje wstawiania i rzadko przeprowadzane jest wyszukiwanie. W takich przypadkach lepszym rozwiązaniem jest zwykle użycie klas std::vector lub std::list.

Podsumowanie

W tej lekcji dowiedziałeś się, w jaki sposób używać obiektów STL `set` i `multiset`, poznałeś ich ważne funkcje składowe oraz cechy charakterystyczne. Zobaczyłeś również, jak można je wykorzystać podczas tworzenia prostej książki telefonicznej bazującej na menu, która oferuje także funkcje wyszukiwania i usuwania kontaktów.

Pytania i odpowiedzi

Pytanie: W jaki sposób mogę zadeklarować obiekt set przechowujący liczby całkowite, aby były posortowane i przechowywane w kolejności malejącej?

Odpowiedź: Instrukcja `set <int>` definiuje obiekt `set` przechowujący liczby całkowite. Wykorzystuje domyślnie predykat sortowania `std::less <T>`, który powoduje sortowanie w kolejności rosnącej. Instrukcję tę można przedstawić również jako `set <int, less <int> >`. Aby zatem sortowanie przebiegało w kolejności malejącej, obiekt `set` należy zdefiniować jako `set <int, greater <int> >`.

Pytanie: Co się stanie, jeżeli w obiekcie `set` przechowującym ciągi tekstowe dwukrotnie wstawię ciąg tekstowy „`Jacek`"?

Odpowiedź: Obiekt `set` nie jest przystosowany do przechowywania wartości, które nie są unikalne. Dlatego też implementacja klasy `std::set` nie pozwala na wstawienie drugiej takiej samej wartości.

Pytanie: Co powinienem zmienić w poprzednim przykładzie, jeżeli będę chciał mieć dwa egzemplarze ciągu tekstowego „`Jacek`"?

Odpowiedź: Obiekt `set` z założenia przechowuje jedynie unikalne wartości. Rozwiązaniem będzie więc zastosowanie kontenera `multiset`.

Pytanie: Która funkcja składowa obiektu `multiset` zwraca liczbę elementów, które przechowują określoną wartość w kontenerze?

Odpowiedź: Ta funkcja to `count(wartość)`.

Pytanie: W obiekcie `set` wyszukałem wartość za pomocą funkcji `find()` i mam iterator prowadzący do niej. Czy mogę użyć tego iteratora, aby zmienić wartość, do której prowadzi?

Odpowiedź: Nie. Pewne implementacje STL mogą pozwalać użytkownikowi na zmianę wartości elementu w obiekcie `set` za pomocą iteratora zwróconego np. przez funkcję `find()`. Jednak to nie jest właściwy sposób działania. Iterator do elementu w obiekcie `set` powinien być używany jako `const` — nawet jeśli implementacja STL nie wymaga takiego zachowania.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Deklarujesz obiekt `set` przechowujący liczby całkowite (instrukcja `set <int>`). Która funkcja zapewnia kryteria sortowania?
2. Czy w obiekcie `multiset` znajdziesz powtarzające się elementy?
3. Która funkcja obiektów `set` lub `multiset` zwraca liczbę elementów przechowywanych w kontenerze?

Ćwiczenia

1. Przedstawiony w tej lekcji program książki telefonicznej rozbuduj w taki sposób, aby można było wyszukać imię osoby na podstawie podanego numeru telefonu bez zmiany struktury `ContactItem`. (Podpowiedź: zdefiniuj obiekt `set` wraz z binarnym predykatem, który powoduje sortowanie względem numerów telefonu, tym samym przesłaniając domyślne sortowanie bazujące na operatorze <).
2. Zdefiniuj obiekt `multiset` przechowujący słowa i ich znaczenie — tzn. obiekt `multiset` ma działać jak słownik. (Podpowiedź: obiekt `multiset` powinien mieć strukturę składającą się z dwóch ciągów tekstowych: słowo i jego znaczenie).
3. Zademonstruj za pomocą przykładowego programu, że obiekt `set` nie akceptuje powtarzających się elementów, podczas gdy obiekt `multiset` pozwala na stosowanie duplikatów.

Lekcja 20

Klasy STL map

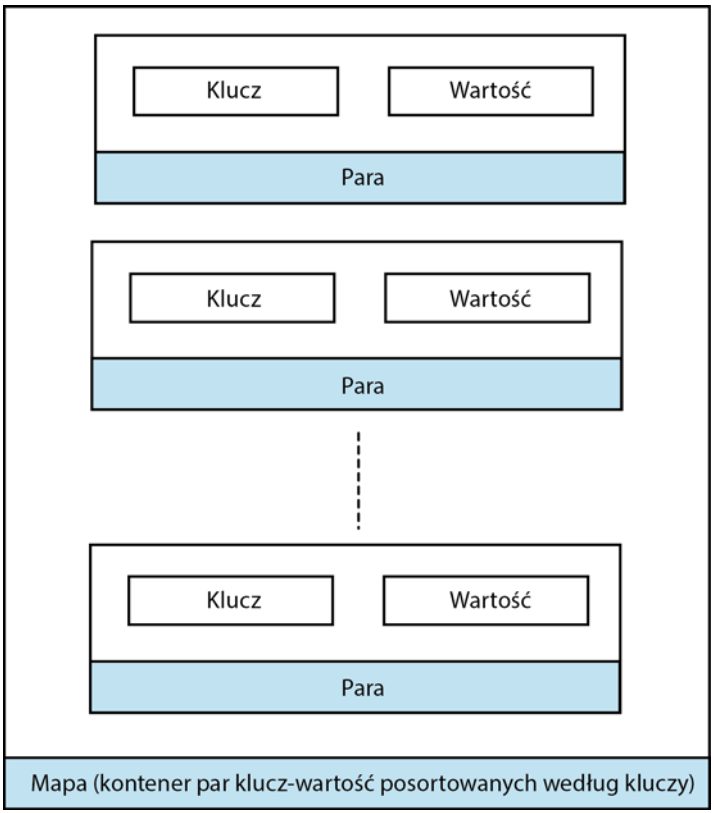
Standardowa biblioteka wzorców (STL) dostarcza programiście klasy kontenerów, które pomagają w tworzeniu aplikacji wymagających częstego przeprowadzania szybkich operacji wyszukiwania.

Z tej lekcji dowiesz się:

- **wprowadzenie do klas STL `map`, `multimap`, `unordered_map` i `unordered_multimap`,**
- **podstawowe operacje wstawiania, usuwania i wyszukiwania elementów,**
- **dostosowanie zachowania klas do własnych potrzeb przy użyciu predykatu sortowania,**
- **podstawy działania tabel hash.**

Krótkie wprowadzenie do klas STL map

Obiekty `Map` i `multimap` to kontenery par klucz-wartość, które pozwalają na przeprowadzanie wyszukiwania na podstawie kluczy, tak jak pokazano na rysunku 20.1.



RYSUNEK 20.1. Graficzne przedstawienie kontenera par typu klucz-wartość

Różnica między `map` i `multimap` polega na tym, że tylko ten drugi obiekt pozwala na stosowanie duplikatów, podczas gdy obiekt `map` może przechowywać jedynie klucze unikalne.

Aby ułatwić szybkie wyszukiwanie, implementacje obiektów STL `map` i `multimap` wewnętrznie są podobne do drzewa binarnego. Oznacza to, że elementy wstawione do obiektów `map` i `multimap` są sortowane w trakcie wstawiania. To również oznacza, że w przeciwieństwie do obiektu `vector`, w którym element w danym położeniu może być zastąpiony przez inny, w obiekcie `map` element w danym położeniu nie może być zastąpiony

przez nowy element o innej wartości. Dzieje się tak, ponieważ obiekt map umieści go w innym położeniu, zgodnie z jego wartością, stosownie do elementów znajdujących się w wewnętrznym drzewie.

Wskazówka

W celu użycia klas std::map lub std::multimap w programie trzeba umieścić następujący nagłówek:

```
#include<map>
```

Podstawowe operacje klas STL map i multimap

Klasy STL map i multimap to klasy wzorców, które muszą być ustanowione, zanim będzie można przystąpić do używania ich funkcji składowych.

Ustanawianie obiektu std::map lub std::multimap

Ustanowienie obiektu map lub multimap wraz z liczbami całkowitymi jako kluczami i ciągami tekstowymi jako wartościami wymaga przeprowadzenia specjalizacji klas wzorca std::map lub std::multimap. Ustanowienie wzorca klasy map wymaga od programisty podania typu klucza, typu wartości oraz opcjonalnie predykatu, który pomaga klasie map w sortowaniu elementów w trakcie ich wstawiania. Typowa składnia ustanawiania obiektu map jest podobna do przedstawionej poniżej:

```
#include <map>
using namespace std;
...
map <keyType, valueType, Predicate=std::less <keyType> > mapObject;
multimap <keyType, valueType, Predicate=std::less <keyType> > mmapObject;
```

Trzeci parametr wzorca jest opcjonalny. Po podaniu jedynie typu klucza i wartości, a zignorowaniu trzeciego parametru wzorca, klasy std::map i std::multimap będą domyślnie stosować std::less<> jako kryterium sortowania. Dlatego też obiekty map lub multimap mapujące liczby całkowite na ciągi tekstowe przedstawiają się następująco:

```
std::map<int, string> mapIntToString;
std::multimap<int, string> mmapIntToString;
```

W listingu 20.1 przedstawiono ustanowienie obiektu map w bardziej szczegółowy sposób.

Listing 20.1. Ustanowienie obiektów STL map i multimap (typ klucza: integer, typ wartości: string)

```
 0: #include<map>
 1: #include<string>
 2:
 3: template<typename KeyType>
 4: struct ReverseSort
 5: {
 6:    bool operator()(const KeyType& key1, const KeyType& key2)
 7:    {
 8:       return (key1 > key2);
 9:    }
10: };
11:
12: int main ()
13: {
14:    using namespace std;
15:
16:    //W obiektach map i multimap typem klucza jest int, natomiast typem wartości— string.
17:    map<int, string> mapIntToString1;
18:    multimap<int, string> mmapIntToString1;
19:
20:    // Obiekty map i multimap powstałe jako kopie innych obiektów.
21:    map<int, string> mapIntToString2(mapIntToString1);
22:    multimap<int, string> mmapIntToString2(mmapIntToString1);
23:
24:    // Obiekty map i multimap utworzone na podstawie części innego obiektu map lub multimap.
25:    map<int, string> mapIntToString3(mapIntToString1.cbegin(),
26:                                     mapIntToString1.cend());
27:
28:    multimap<int, string> mmapIntToString3(mmapIntToString1.cbegin(),
29:                                           mmapIntToString1.cend());
30:
31:    // Obiekty map i multimap wraz z predykatem odwracającym kolejność sortowania.
32:    map<int, string, ReverseSort<int> > mapIntToString4
33:       (mapIntToString1.cbegin(), mapIntToString1.cend());
34:
35:    multimap<int, string, ReverseSort<int> > mmapIntToString4
36:       (mapIntToString1.cbegin(), mapIntToString1.cend());
37:
38:    return 0;
39: }
```

Analiza ▼

Na początek koncentrujemy się na funkcji `main()` zdefiniowanej w wierszach od 12. do 39. Najprostsza postać obiektów `map` i `multimap` wraz z kluczami w postaci liczb całkowitych i wartościami w postaci ciągów tekstowych przedstawiono w wierszach 21. i 22. Z kolei w wierszach od 25. do 28. pokazano utworzenie obiektów `map` i `multimap` zainicjalizowanych wraz z zakresem wartości pochodzących z innego obiektu. Wiersze od 31. do 36. pokazują utworzenie obiektu `map` lub `multimap` wraz z własnym predykatem sortowania. Zwróć uwagę, że w sortowaniu domyślnym (w poprzednio utworzonych egzemplarzach) użyto `std::less<T>`, co powoduje sortowanie elementów w kolejności rosnącej. Jeżeli chcesz zmienić to zachowanie, musisz umieścić w klasie implementującej operator `()` definicję predykatu. Tego rodzaju predykat (patrz wiersze od 4. do 10.) został wykorzystany podczas tworzenia obiektów `map` (patrz wiersz 32.) i `multimap` (patrz wiersz 35.).

Wskazówka

Czy użycie metod `cbegin()` i `cend()` powoduje błędy w trakcie kompilacji programu?

Jeżeli przedstawiony program próbujesz skompilować przy użyciu kompilatora niezgodnego ze standardem C++11, wtedy zamiast metod `cbegin()` i `cend()` użyj (odpowiednio) `begin()` i `end()`. Oferowane przez standard C++11 metody `cbegin()` i `cend()` są użyteczne, ponieważ zwracają iterator `const`, który nie może być wykorzystany do modyfikacji elementów.

Wstawianie elementów do obiektów STL map lub multimap

Większość funkcji obiektów `map` i `multimap` działa w bardzo podobny sposób. Akceptują podobne parametry i zwracają wartości podobnych typów. Przykładowo w celu wstawienia elementów do obu rodzajów kontenerów można wykorzystać funkcję składową `insert()`:

```
std::map<int, std::string> mapIntToString1;
// Wstawienie pary klucz-wartość przy użyciu funkcji make_pair().
mapIntToString.insert (make_pair (-1, "Minus Jeden"));
```

Ponieważ dwa omawiane kontenery przechowują elementy w postaci par klucz-wartość, istnieje również możliwość bezpośredniego użycia obiektu `std::pair` zainicjalizowanego wraz z wstawianym kluczem i jego wartością:

```
mapIntToString.insert (pair <int, string>(1000, "Tysiąc"));
```

Alternatywnym rozwiązaniem jest zastosowanie składni tablicy, która jest bardzo przyjazna dla użytkownika i obsługiwana przy użyciu operatora indeksowania:

```
mapIntToString [1000000] = "Milion";
```

Obiekt `multimap` można utworzyć także jako kopię obiektu `map`:

```
std::multimap<int, std::string> mmapIntToString(mapIntToString.cbegin(),
mapIntToString.cend());
```

Różne metody wstawiania elementów pokazano w listingu 20.2.

Listing 20.2. Wstawianie elementów do obiektów STL map i multimap przy użyciu przeciążonych wersji metody insert() oraz semantyki tablicy z wykorzystaniem operatora indeksowania

```
 0: #include <map>
 1: #include <iostream>
 2: #include<string>
 3:
 4: using namespace std;
 5:
 6: // Zdefiniowanie typedef dla definicji obiektów map i multimap w celu zwiększenia czytelności.
 7: typedef map <int, string> MAP_INT_STRING;
 8: typedef multimap <int, string> MMAP_INT_STRING;
 9:
10: template <typename T>
11: void DisplayContents (const T& Input)
12: {
13:    for(auto iElement = Input.cbegin() // auto i cbegin(): C++11.
14:       ; iElement != Input.cend()   // cend() to nowość w C++11.
15:       ; ++ iElement )
16:       cout << iElement->first << " -> " << iElement->second << endl;
17:
18:    cout << endl;
19: }
20:
21: int main ()
22: {
23:     MAP_INT_STRING mapIntToString;
24:
25:     // Wstawienie pary klucz-wartość do obiektu map za pomocą  value_type.
26:     mapIntToString.insert (MAP_INT_STRING::value_type (3, "Trzy"));
27:
28:     // Wstawienie pary przy użyciu funkcji make_pair.
29:     mapIntToString.insert (make_pair (-1, "Minus Jeden"));
30:
31:     // Bezpośrednie wstawienie obiektu pary.
32:     mapIntToString.insert (pair <int, string> (1000, "Tysiąc"));
```

```
33:
34:     // Wstawienie pary klucz-wartość przy użyciu składni podobnej do tablicy.
35:     mapIntToString [1000000] = "Milion";
36:
37:     cout << "Obiekt map zawiera " << mapIntToString.size ();
38:     cout << " par(ę/y) klucz-wartość. Elementy w obiekcie map to:" <<
        ↪endl;
39:     DisplayContents(mapIntToString);
40:
41:    // Utworzenie obiektu multimap, który jest kopią obiektu map.
42:    MMAP_INT_STRING mmapIntToString(mapIntToString.cbegin(),
43:                              mapIntToString.cend());
44:
45:     // W obiekcie multimap funkcja insert() działa dokładnie w taki sam sposób.
46:     // Obiekt multimap może przechowywać duplikaty — wstawiamy jeden.
47:     mmapIntToString.insert (make_pair (1000, "Tysiąc"));
48:
49:     cout << endl << "Obiekt multimap zawiera " << mmapIntToString.size ();
50:     cout << " par(ę/y) klucz-wartość." << endl;
51:     cout << "Elementy w obiekcie multimap to: " << endl;
52:     DisplayContents(mmapIntToString);
53:
54:     // Obiekt multimap może również podać liczbę par o tym samym kluczu.
55:     cout << "Liczba par w obiekcie multimap o kluczu 1000: "
56:         << mmapIntToString.count (1000) << endl;
57:
58:     return 0;
59: }
```

Wynik ▼

```
Obiekt map zawiera 4 par(ę/y) klucz-wartość. Elementy w obiekcie map to:
-1 -> Minus Jeden
3 -> Trzy
1000 -> Tysiąc
1000000 -> Milion

Obiekt multimap zawiera 5 par(ę/y) klucz-wartość.
Elementy w obiekcie multimap to:
-1 -> Minus Jeden
3 -> Trzy
45 -> Czterdzieści pięć
1000 -> Tysiąc
1000 -> Tysiąc

Liczba par w obiekcie multimap o kluczu 1000: 2
```

Analiza ▼

W wierszach 7. i 8. znajduje się instrukcja typedef ustanawiająca obiekty map i multimap. Dzięki temu kod jest nieco prostszy. W wierszach od 10. do 19. znajduje się metoda DisplayContents() zaadaptowana na potrzeby obiektów map i multimap. Iterator jest używany w celu uzyskania dostępu do zmiennych first (wskazuje klucz) i second (wskazuje wartość). W wierszach od 26. do 32. pokazano różne sposoby wstawiania elementów do obiektu map przy użyciu przeciążonych wersji metody insert(). Natomiast w wierszu 35. pokazano użycie operatora indeksowania [] do wstawienia elementu za pomocą składni podobnej do tablicy. Wszystkie pozostałe metody można zastosować również dla obiektu multimap, co pokazano w wierszu 47., w którym następuje wstawienie duplikatu do obiektu multimap. Interesujące jest, że obiekt multimap został zainicjalizowany jako kopia obiektu map (patrz wiersze 42. i 43.). Dane wyjściowe pokazują, że w przypadku dwóch omawianych kontenerów wstawiane pary klucz-wartość są automatycznie sortowane w rosnącej kolejności kluczy. Dane wyjściowe pokazują również, że obiekt multimap może przechowywać dwie pary o takim samym kluczu (w omawianym przypadku to 1000). Z kolei w wierszu 56. pokazano użycie metody multimap::count() do sprawdzenia, ile elementów o podanym kluczu istnieje w kontenerze.

Wskazówka

Czy użycie słowa kluczowego auto powoduje błędy w trakcie kompilacji?

W przedstawionej w listingu 20.2 funkcji DisplayContents() użyto wprowadzonego w standardzie C++11 słowa kluczowego auto do zdefiniowania typu iteratora (patrz wiersz 13.). Ponadto kod zawiera wywołania metod cbegin() i cend(), które są nowością w standardzie C++11 i zwracają wartość typu const_iterator.

Aby ten i kolejne przykłady skompilować w starszej wersji kompilatora, nieobsługującej standardu C++11, musisz słowo kluczowe auto zastąpić konkretnym typem.

Dlatego też metoda DisplayContent() dla starszej wersji kompilatora będzie miała następującą postać:

```
template <typename T>
void DisplayContent(const T& Input)
{
   for (T::const_iterator iElement = Input.begin()
      ; iElement != Input.end ()
      ; ++ iElement )
     cout << iElement->first << " -> " << iElement->second << endl;

   cout << endl;
}
```

Wyszukiwanie elementów w obiekcie STL map

Kontenery asocjacyjne, takie jak `map` i `multimap`, zawierają funkcję `find()` — jest to metoda składowa pozwalająca na wyszukanie wartości podanego klucza. Wynikiem operacji wyszukiwania zawsze jest iterator:

```
multimap <int, string>::const_iterator iPairFound =
mapIntToString.find(Key);
```

Na początku trzeba sprawdzić otrzymany iterator (czy działanie metody `find()` zakończyło się powodzeniem), a dopiero później można uzyskać dostęp do znalezionej wartości:

```
if (iPairFound != mapIntToString.end())
{
   cout << "Klucz "" << iPairFound->first << " prowadzi do wartości: ";
   cout << iPairFound->second << endl;
}
else
   cout << "Przepraszamy, para o kluczu " << Key << " nie istnieje
w obiekcie" << endl;
```

Wskazówka

Jeżeli używasz kompilatora zgodnego ze standardem C++11, możesz uprościć deklarację iteratora, korzystając ze słowa kluczowego `auto`:

```
auto iPairFound = mapIntToString.find(Key);
```

Kompilator automatycznie określi typ iteratora na podstawie zadeklarowanej wartości zwrotnej metody `map::find()`.

Przykładowy program przedstawiony w listingu 20.3 pokazuje zastosowanie funkcji `map::find`.

Listing 20.3. Używanie funkcji składowej find() w obiekcie map do wyszukania pary klucz-wartość

```
 0: #include <map>
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents (const T& Input)
 7: {
 8:    for(auto iElement = Input.cbegin() // auto and cbegin(): C++11.
 9:       ; iElement != Input.cend()   // cend() to nowość w C++11.
10:       ; ++ iElement )
11:       cout << iElement->first << " -> " << iElement->second << endl;
12:
```

```
13:    cout << endl;
14: }
15:
16: int main ()
17: {
18:     map<int, string> mapIntToString;
19:
20:     mapIntToString.insert(make_pair (3, "Trzy"));
21:     mapIntToString.insert(make_pair (45, "Czterdzieści pięć"));
22:     mapIntToString.insert(make_pair (-1, "Minus Jeden"));
23:     mapIntToString.insert(make_pair (1000, "Tysiąc"));
24:
25:     cout << "Obiekt map zawiera " << mapIntToString.size ();
26:     cout << " par(ę/(y) klucz-wartość. Elementy w obiekcie map to: " <<
       ↪endl;
27:
28:     // Wyświetlenie na ekranie zawartości obiektu map.
29:     DisplayContents(mapIntToString);
30:
31:    cout << "Podaj klucz, który chcesz znaleźć: ";
32:    int Key = 0;
33:    cin >> Key;
34:
35:    auto iPairFound = mapIntToString.find(Key);
36:    if (iPairFound != mapIntToString.end())
37:    {
38:       cout << "Klucz " << iPairFound->first << " prowadzi do wartości: ";
39:       cout << iPairFound->second << endl;
40:    }
41:    else
42:       cout << "Przepraszamy, para o kluczu " << Key << " nie istnieje
         ↪w obiekcie" << endl;
43:
44:     return 0;
45: }
```

Wynik ▼

```
Obiekt map zawiera 4 par(ę/y) klucz-wartość. Elementy w obiekcie map to:
-1 -> Minus Jeden
3 -> Trzy
45 -> Czterdzieści pięć
1000 -> Tysiąc

Podaj klucz, który chcesz znaleźć: 45
Klucz 45 prowadzi do wartości: Czterdzieści pięć
```

Kolejne uruchomienie programu (metoda find() nie znajduje odpowiedniej wartości):

```
Obiekt map zawiera 4 par(ę/y) klucz-wartość. Elementy w obiekcie map to:
-1 -> Minus Jeden
3 -> Trzy
45 -> Czterdzieści pięć
1000 -> Tysiąc

Podaj klucz, który chcesz znaleźć: 2011
Przepraszamy, para o kluczu 2011 nie istnieje w obiekcie
```

Analiza ▼

W wierszach od 20. do 23. w funkcji `main()` następuje umieszczenie przykładowych par w obiekcie `map`, każda para zawiera mapowanie liczby całkowitej na ciąg tekstowy. Kiedy użytkownik wprowadzi klucz do wyszukania w obiekcie `map`, w wierszu 35. metoda `find()` zostaje użyta do znalezienia wskazanego klucza. Funkcja `find()` zawsze zwraca iterator prowadzący do znalezionego elementu. W tym przypadku elementem będzie para klucz-wartość. Aby określić, czy działanie funkcji `find()` zakończyło się powodzeniem, w pierwszej kolejności iterator powinien być porównany z drugim, zwróconym przez funkcję `end()`, jak pokazano w wierszu 36. Jeżeli iterator jest poprawny, wtedy następuje użycie elementu składowego `second` w celu uzyskania dostępu do wartości, co pokazano w wierszu 39. W trakcie drugiego uruchomienia programu podano nieistniejący w obiekcie klucz 2011, co doprowadziło do wyświetlenia użytkownikowi komunikatu błędu.

Ostrzeżenie

Wyniku operacji `find()` nigdy nie używaj bezpośrednio bez wcześniejszego sprawdzenia iteratora, czy operacja zakończyła się powodzeniem.

Wyszukiwanie elementów w obiekcie STL multimap

Gdyby w listingu 20.3 został użyty obiekt `multimap`, kontener mógłby przechowywać wiele par o takich samych kluczach i można by znaleźć wartości odpowiadające podanemu kluczowi. Dlatego też w obiekcie `multimap` do określenia liczby wartości odpowiadających podanemu kluczowi i inkrementacji iteratora konieczne jest użycie metody `multimap::count()`, aby uzyskać dostęp do kolejnych wartości.

```
auto iPairFound = mmapIntToString.find(Key);
// Sprawdzenie, czy działanie metody find() zakończyło się powodzeniem.
if(iPairFound != mmapIntToString.end())
{
   // Ustalenie liczby par o takim samym kluczu jak podany przez użytkownika.
   size_t nNumPairsInMap = mmapIntToString.count(1000);

   for( size_t nValuesCounter = 0
      ; nValuesCounter < nNumPairsInMap   // Pozostanie w zakresie.
      ; ++ nValuesCounter )
   {
      cout << "Klucz: " << iPairFound->first;  // Klucz.
      cout << ", Wartość [" << nValuesCounter << "] = ";
      cout << iPairFound->second << endl;     // Wartość.

      ++ iPairFound;
   }
}
else
   cout << "Element nie został znaleziony w obiekcie multimap";
```

Usuwanie elementów z obiektów STL map lub multimap

Obiekty `map` i `multimap` zawierają funkcję składową `erase()`, która pozwala na usunięcie elementu z kontenera. Funkcja `erase()` jest wywoływana wraz z parametrem w postaci klucza, który powoduje usunięcie wszystkich zawierających go par:

```
mapObject.erase (klucz);
```

Inna postać funkcji `erase()` umożliwia usunięcie określonego elementu na podstawie prowadzącego do niego iteratora:

```
mapObject.erase (iElement);
```

Istnieje również możliwość usunięcia zakresu elementów z obiektów `map` lub `multimap` przy użyciu iteratorów, które wskazują granice tego zakresu:

```
mapObject.erase (iPoczatekZakresu, iKoniecZakresu);
```

Kod przedstawiony w listingu 20.4 to demonstracja użycia funkcji `erase()`.

Listing 20.4. Usuwanie elementów z obiektu multimap

```
 0: #include <map>
 1: #include <iostream>
 2: #include <string>
```

```
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents (const T& Input)
 7: {
 8:    for(auto iElement = Input.cbegin() // auto and cbegin(): C++11.
 9:       ; iElement != Input.cend()   // cend() to nowość w C++11.
10:       ; ++ iElement )
11:       cout << iElement->first << " -> " << iElement->second << endl;
12:
13:    cout << endl;
14: }
15:
16: int main ()
17: {
18:     multimap<int, string> mmapIntToString;
19:
20:     // Wstawienie par klucz-wartość do obiektu multimap.
21:     mmapIntToString.insert (make_pair (3, "Trzy"));
22:     mmapIntToString.insert (make_pair (45, " Czterdzieści pięć"));
23:     mmapIntToString.insert (make_pair (-1, "Minus Jeden"));
24:     mmapIntToString.insert (make_pair (1000, "Tysiąc"));
25:
26:     // Wstawienie duplikatów do obiektu multimap.
27:     mmapIntToString.insert (make_pair (-1, "Minus Jeden"));
28:     mmapIntToString.insert (make_pair (1000, "Tysiąc"));
29:
30:     cout << "Obiekt multimap zawiera " << mmapIntToString.size ();
31:     cout << " par(ę/y) klucz-wartość. " << "Pary to: " << endl;
32:     DisplayContents(mmapIntToString);
33:
34:     // Usunięcie z obiektu multimap elementu z kluczem -1.
35:     auto NumPairsErased = mmapIntToString.erase(-1);
36:     cout<< "Usunięto " << NumPairsErased << " par(y) o kluczu -1."<<
        ↪endl;
37:
38:     // Usunięcie z obiektu multimap elementu, do którego prowadzi podany iterator.
39:     auto iPairLocator = mmapIntToString.find(45);
40:     if (iPairLocator != mmapIntToString.end ())
41:     {
42:         mmapIntToString.erase (iPairLocator);
43:         cout << "Usunięcie par z kluczem 45 za pomocą iteratora" << endl;
44:     }
45:
46:     // Usunięcie z obiektu multimap zakresu elementów...
47:     cout << "Usunięcie zakresu par z kluczem 1000." << endl;
48:     mmapIntToString.erase ( mmapIntToString.lower_bound (1000)
49:         , mmapIntToString.upper_bound (1000) );
```

```
50:
51:     cout << "Obiekt multimap zawiera teraz " << mmapIntToString.size ();
52:     cout << " par(ę/y) klucz-wartość." << "Pary to: " << endl;
53:     DisplayContents(mmapIntToString);
54:
55:     return 0;
56: }
```

Wynik ▼

```
Obiekt multimap zawiera 6 par(ę/y) klucz-wartość. Pary to:
-1 -> Minus Jeden
-1 -> Minus Jeden
3 -> Trzy
45 -> Czterdzieści pięć
1000 -> Tysiąc
1000 -> Tysiąc

Usunięto 2 par(y) o kluczu -1.
Usunięcie par z kluczem 45 za pomocą iteratora
Usunięcie zakresu par z kluczem 1000.
Obiekt multimap zawiera teraz 1 par(ę/y) klucz-wartość. Pary to:
3 -> Trzy
```

Analiza ▼

W wierszach od 21. do 28. do obiektu `multimap` zostają wstawione przykładowe wartości, niektóre z nich powtarzają się (ponieważ w przeciwieństwie do `map`, obiekt `multimap` pozwala na wstawienie powtarzających się wartości). Po wstawieniu par do obiektu `multimap` w kodzie usuwane są elementy przy użyciu wersji funkcji `erase()`, która akceptuje klucz i następnie usuwa wszystkie elementy o podanym kluczu (–1), co pokazano w wierszu 35. Wartością zwrotną `map::erase(klucz)` jest liczba usuniętych elementów, zostanie ona wyświetlona na ekranie. W wierszu 39. iterator zwrócony przez metodę `find(45)` jest użyty do usunięcia tej pary z obiektu. W wierszach 48. i 49. pokazano usunięcie par o kluczach z zakresu podanego przez funkcje `lower_bound()` i `upper_bound()`.

Dostarczanie własnego predykatu sortowania

Definicje wzorców `map` i `multimap` zawierają trzeci parametr, który akceptuje predykat sortowania dla obiektu `map` pozwalający obiektowi na poprawne funkcjonowanie. Ten trzeci parametr, gdy nie zostanie wymieniony (jak miało to miejsce we wcześniejszych przykładach), jest zastępowany przez domyślne kryteria sortowania dostarczane przez `std::less <>`. Oznacza to porównywanie dwóch obiektów za pomocą operatora <.

Aby podać inne kryterium sortowania, najczęściej należy umieścić w klasie binarny predykat w postaci operatora `()`:

```
template<typename KeyType>
struct Predicate
{
   bool operator()(const KeyType& key1, const KeyType& key2)
   {
      // Miejsce na kod sortowania.
   }
};
```

Obiekt `map` przechowujący elementy typu `std::string` jako klucze będzie miał domyślne kryterium sortowania bazujące na operatorze < zdefiniowanym przez klasę `std::string` (wywoływane przez domyślny predykat sortowania `std::less<T>`), a to oznacza rozróżnianie wielkości znaków. W wielu aplikacjach, np. w książce telefonicznej, ważne jest, aby operacje wstawiania i wyszukiwania nie rozróżniały wielkości znaków. Jedynym sposobem rozwiązania tego problemu jest użycie obiektu `map` wraz z predykatem sortowania zwracającego wartość albo `true`, albo `false` na podstawie porównania, które nie rozróżnia wielkości znaków:

```
map <keyType, valueType, Predicate> mapObject;
```

Przykład zastosowania takiego rozwiązania przedstawiono w listingu 20.5.

Listing 20.5. Dostarczenie własnego predykatu sortowania podczas budowania programu książki telefonicznej

```
0: #include <map>
1: #include <algorithm>
2: #include <string>
3: #include <iostream>
4: using namespace std;
```

```
 5:
 6: template <typename T>
 7: void DisplayContents (const T& Input)
 8: {
 9:    for(auto iElement = Input.cbegin() // auto and cbegin(): C++11.
10:       ; iElement != Input.cend()   // cend() to nowość w C++11.
11:       ; ++ iElement )
12:       cout << iElement->first << " -> " << iElement->second << endl;
13:
14:    cout << endl;
15: }
16:
17: struct PredIgnoreCase
18: {
19:     bool operator() (const string& str1, const string& str2) const
20:     {
21:         string str1NoCase (str1), str2NoCase (str2);
22:         std::transform (str1.begin(), str1.end(), str1NoCase.begin(),
          ↪tolower);
23:         std::transform (str2.begin(), str2.end(), str2NoCase.begin(),
          ↪tolower);
24:
25:         return (str1NoCase < str2NoCase);
26:     };
27: };
28:
29: typedef map <string, string> DIRECTORY_WITHCASE;
30: typedef map <string, string, PredIgnoreCase> DIRECTORY_NOCASE;
31:
32: int main ()
33: {
34:     // Nierozróżniająca wielkości znaków książka telefoniczna: wielkość znaków ciągu
      ↪tekstowego nie ma znaczenia.
35:     DIRECTORY_NOCASE dirCaseInsensitive;
36:
37:     dirCaseInsensitive.insert(make_pair("Jan", "2345764"));
38:     dirCaseInsensitive.insert(make_pair("JAN", "2345764"));
39:     dirCaseInsensitive.insert(make_pair("Sandra", "42367236"));
40:     dirCaseInsensitive.insert(make_pair("Jacek", "32435348"));
41:
42:     cout << "Wyświetlenie zawartości obiektu map, który nie rozróżnia
      ↪wielkości znaków:" << endl;
43:     DisplayContents(dirCaseInsensitive);
44:
45:     // Wielkość znaków ciągu tekstowego klucza ma wpływ na operacje wstawiania
      ↪i wyszukiwania.
46:     DIRECTORY_WITHCASE dirCaseSensitive(dirCaseInsensitive.begin()
47:        , dirCaseInsensitive.end());
48:
```

```
49:     cout << "Wyświetlenie zawartości obiektu map, który rozróżnia
        ↪wielkość znaków:" << endl;
50:     DisplayContents(dirCaseSensitive);
51:
52:     // Wyszukanie imienia w tych dwóch obiektach map i wyświetlenie wyników operacji.
53:     cout << "Proszę podać imię do wyszukania: " << endl << "> ";
54:     string strNameInput;
55:     cin >> strNameInput;
56:
57:     // Wyszukiwanie w obiekcie map…
58:     auto iPairInNoCaseDir = dirCaseInsensitive.find (strNameInput);
59:     if (iPairInNoCaseDir != dirCaseInsensitive.end())
60:     {
61:         cout << "Numer telefonu osoby " <<iPairInNoCaseDir->first<< "
            ↪w książce telefonicznej,";
62:         cout << " która nie rozróżnia wielkości znaków to: " <<
            ↪iPairInNoCaseDir->second << endl;
63:     }
64:     else
65:     {
66:         cout << "Numer telefonu osoby " << strNameInput << " nie został
            ↪znaleziony";
67:         cout << "w książce telefonicznej, która nie rozróżnia wielkości
            ↪znaków" << endl;
68:     }
69:
70:     // Wyszukiwanie w obiekcie map, który rozróżnia wielkość znaków…
71:     auto iPairInCaseSensDir = dirCaseSensitive.find (strNameInput);
72:     if (iPairInCaseSensDir != dirCaseSensitive.end())
73:     {
74:         cout << "Numer telefonu osoby " << iPairInCaseSensDir->first<<
            ↪" w książce telefonicznej,";
75:         cout << " która rozróżnia wielkość znaków to: " <<
            ↪iPairInCaseSensDir->second << endl;
76:     }
77:     else
78:     {
79:         cout << "Numer telefonu osoby " << strNameInput << " nie został
            ↪znaleziony";
80:         cout << " w ksiażce telefonicznej, która rozróżnia wielkość
            ↪znaków" << endl;
81:     }
82:
83:     return 0;
84: }
```

Wynik ▼

```
Wyświetlenie zawartości obiektu map, który nie rozróżnia wielkości znaków:
Jacek -> 32435348
Jan -> 2345764
Sandra -> 42367236

Wyświetlenie zawartości obiektu map, który rozróżnia wielkość znaków:
Jacek -> 32435348
Jan -> 2345764
Sandra -> 42367236

Proszę podać imię szukane w obu książkach telefonicznych:
> sandra
Numer telefonu osoby Sandra w ksiażce telefonicznej, która nie rozróżnia
↪wielkości znaków to: 2345764
Numer telefonu osoby sandra nie został znaleziony w książce telefonicznej,
↪która rozróżnia wielkość znaków
```

Analiza ▼

Program przedstawiony w listingu 20.5 ma dwie identyczne książki telefoniczne bazujące na obiektach `map` przechowujących pary zawierające i klucze, i wartości w postaci ciągów tekstowych. Jedna książka telefoniczna używa predykatu domyślnego (`std::less <>`) wywołującego operator < dla klucza, tzn. klasy `std::string`, która rozróżnia wielkość liter. Druga książka telefoniczna bazuje na obiekcie `map` zawierającym predykat `PredIgnoreCase` (patrz wiersze od 17. do 27.), który porównuje dwa ciągi tekstowe, niezależnie od wielkości znaków tego ciągu. Dane wyjściowe programu pokazują, że kiedy użytkownik podaje imię (tzn. klucz do wyszukania) istniejące w obu książkach telefonicznych, ale zapisane z użyciem różnej wielkości znaków, wersja nierozróżniająca wielkości znaków dzięki funkcji predykatu potrafi znaleźć to imię, natomiast wersja rozróżniająca wielkość znaków nie znajduje go.

Uwaga

> W listingu 20.5 struktura `PredIgnoreCase` może być klasą, jeśli dodasz słowo kluczowe `public` dla operatora `()`. Dla kompilatora C++ struktura jest zbliżona do klasy z (domyślnie) publicznymi elementami składowymi, które stosują dziedziczenie domyślne.

W powyższym programie pokazano, w jaki sposób można używać predykatów w celu dostosowania zachowania obiektu `map` do własnych potrzeb. Widzimy również, że klucz może być dowolnego typu, a programista ma możliwość

dostarczenia predykatu definiującego zachowanie obiektu `map` względem tego typu. Warto zwrócić uwagę, że predykat ma strukturę implementowaną przez `operator()`. To może być również *klasa*. Takiego rodzaju obiekty dublujące funkcje są nazywane *obiektami funkcji* lub *funktorami*. Szczegółowe omówienie tego zagadnienia będzie przedstawione w lekcji 21., „Zrozumienie obiektów funkcji".

Uwaga

Klasa `std::map` jest doskonale przystosowana do przechowywania par klucz-wartość, których można szukać pod kątem wartości dla danego klucza. Podczas przeprowadzania operacji wyszukiwania obiekt `map` gwarantuje lepszą wydajność od STL `vector` lub `list`. Wydajność zmniejsza się wraz ze wzrostem liczby elementów. Wydajność operacyjna obiektu `map` jest z natury logarytmiczna — tzn. proporcjonalna do liczby elementów znajdujących się w obiekcie.

To znacznie lepiej niż w przypadku złożoności liniowej, która charakteryzuje się spadkiem wydajności proporcjonalnie do liczby elementów znajdujących się w kontenerze (podobnie jak w nieposortowanym wektorze).

Wprawdzie złożoność logarytmiczna prezentuje się całkiem dobrze, ale należy pamiętać, że operacje wstawiania elementów do obiektu `map` (`multimap`, `set` i `multiset`) stają się coraz wolniejsze, ponieważ wymienione kontenery przeprowadzają sortowanie podczas wstawiania elementów. Dlatego też nadal trwają poszukiwania szybszych (w trakcie wyszukiwania) kontenerów, a programiści i matematycy próbują opracować kontener oferujący niezmienny czas wstawiania i wyszukiwania elementów. Tabela hash to jedno z tego rodzaju rozwiązań; kontener ten oferuje stały czas wstawiania elementu i niemalże stały czas wyszukiwania (w większości przypadków) danego klucza, niezależnie od wielkości kontenera.

C++11

Tabela hash w bibliotece STL, czyli kontener std::unorderd_map oparty na danych klucz-wartość

W standardzie C++11 biblioteka STL obsługuje mapę hash w postaci klasy `std::unordered_map`. Aby można było użyć tej klasy wzorca, konieczne jest dołączenie nagłówka:

```
#include<unordered_map>
```

Klasa `unordered_map` obiecuje zachowanie niemal stałego czasu wstawiania, usuwania i wyszukiwania dowolnych elementów w kontenerze.

Jak działa tabela hash?

Wprawdzie szczegółowe omówienie tabeli hash wykracza poza zakres tematyczny tej książki (często jest tematem prac naukowych), spróbuję przedstawić podstawy pokazujące sposób działania tabeli hash.

Tabela hash to kolekcja par klucz-wartość, gdzie dla danego klucza tabela może odnaleźć wartość. Różnica pomiędzy tabelą hash i zwykłym obiektem `map` polega na tym, że tabela hash przechowuje pary klucz-wartość w tzw. kubełkach (ang. *bucket*), a każdy kubełek ma indeks definiujący jego względne położenie w tabeli (to rozwiązanie zbliżone do tablicy). Wspomniany indeks jest tworzony przez funkcję hash wykorzystującą klucz jako dane wejściowe:

```
Index = HashFunction(Key, TableSize);
```

Podczas wykonywania metody `find()` dla danego klucza funkcja `HashFunction()` jest używana ponownie w celu określenia położenia elementu, a tabela zwraca wartość znajdującą się w wyliczonym położeniu. Działa to podobnie do tablicy zwracającej przechowywany w niej obiekt. Jeżeli funkcja hash nie jest przygotowana optymalnie, więcej niż tylko jeden element będzie miał ten sam indeks i znajdzie się w tym samym kubełku, który wewnętrznie jest listą elementów. W takich przypadkach (nazywanych *kolizjami*) wyszukiwanie będzie wolniejsze, a czas przeprowadzania tej operacji nie będzie stały.

Używanie tabel hash w C++11: unordered_map i unordered_multimap

Z punktu widzenia możliwości, opisane kontenery niewiele różnią się od `std::map` i `std::multimap`, i mogą przeprowadzać operacje tworzenia egzemplarzy, wstawiania i wyszukiwania elementów na następujące sposoby:

```
// Utworzenie obiektu unordered_map mapującego liczby całkowite na ciągi tekstowe:
unordered_map<int, string> umapIntToString;

// Metoda insert().
umapIntToString.insert(make_pair(1000, "Tysiąc"));

// Metoda find().
auto iPairThousand = umapIntToString.find(1000);
cout << iPairThousand->first << " –> " << iPairThousand->second << endl;

//  Wyszukanie wartości przy użyciu semantyki tablicy.:
cout << "umapIntToString[1000] = " << umapIntToString[1000] << endl;
```

Bardzo ważną cechą obiektu unordered_map jest dostępność funkcji hash, odpowiedzialnej za kolejność sortowania:

```
unordered_map<int, string>::hasher HFn =
   umapIntToString.hash_function();
```

Priorytet przypisany kluczowi można sprawdzić przez wywołanie funkcji hash dla danego klucza:

```
size_t HashingValue1000 = HFn(1000);
```

Obiekt unordered_map przechowuje pary klucz-wartość w kubełkach, przeprowadza automatyczne równoważenie, gdy liczba elementów w obiekcie osiągnie liczbę kubełków lub ma się z nią zrównać:

```
cout << "Współczynnik wypełniania: " << umapIntToString.load_factor() <<
↪endl;
cout << "Maksymalny współczynnik wypełniania = " <<
umapIntToString.max_load_factor() << endl;
cout << "Maksymalna liczba kubełków = " <<
umapIntToString.max_bucket_count() << endl;
```

Dane wyjściowe metody load_factor() informują, do jakiego stopnia kubełki w obiekcie unoredered_map zostały wypełnione. Kiedy wartość zwracana przez metodę load_factor() przekroczy wartość zwrotną metody max_load_factor() na skutek wstawienia elementu, obiekt map automatycznie zwiększy liczbę dostępnych kubełków, a następnie ponownie utworzy tabelę hash, co przedstawiono w listingu 20.6.

Wskazówka

Obiekt std::unordered_multimap jest podobny do std::unordered_map, ale może obsługiwać obecność wielu par o tym samym kluczu.

Użycie std::unordered_multimap jest całkiem podobne do obiektu std::multimap, ale z pewnymi funkcjami charakterystycznymi dla tabeli hash, co przedstawiono w listingu 20.6.

Listing 20.6. Implementacja tabeli hash w bibliotece STL w postaci obiektu unordered_map i przy użyciu metod insert(), find(), size(), max_bucket_count(), load_factor() oraz max_load_factor()

```
0: #include<iostream>
1: #include<string>
2: #include<unordered_map>
3: using namespace std;
4:
5: template <typename T1, typename T2>
```

```
 6: void DisplayUnorderedMap(unordered_map<T1, T2>& Input)
 7: {
 8:    cout << "Liczba par, size() = " << Input.size() << endl;
 9:    cout << "Maksymalna liczba kubełków = " << Input.max_bucket_count() <<
      ↪endl;
10:    cout << "Współczynnik wypełniania: " << Input.load_factor() << endl;
11:    cout << "Maksymalny współczynnik wypełniania = " <<
      ↪Input.max_load_factor() << endl;
12:    cout << "Zawartość nieposortowanego obiektu map: " << endl;
13:
14:    for(auto iElement = Input.cbegin() // auto, cbegin: C++11.
15:       ; iElement != Input.cend()   // cend() to nowość w C++11.
16:       ; ++ iElement )
17:       cout<< iElement->first<< " -> "<< iElement->second<< endl;
18: }
19:
20: int main()
21: {
22:    unordered_map<int, string> umapIntToString;
23:    umapIntToString.insert(make_pair(1, "Jeden"));
24:    umapIntToString.insert(make_pair(45, "Czterdzieści Pięć"));
25:    umapIntToString.insert(make_pair(1001, "Tysiąc Jeden"));
26:    umapIntToString.insert(make_pair(-2, "Minus Dwa"));
27:    umapIntToString.insert(make_pair(-1000, "Minus Tysiąc"));
28:    umapIntToString.insert(make_pair(100, "Sto"));
29:    umapIntToString.insert(make_pair(12, "Dwanaście"));
30:    umapIntToString.insert(make_pair(-100, "Minus Sto"));
31:
32:    DisplayUnorderedMap<int, string>(umapIntToString);
33:
34:    cout << "Wstawienie jeszcze jednego elementu" << endl;
35:    umapIntToString.insert(make_pair(300, "Trzysta"));
36:    DisplayUnorderedMap<int, string>(umapIntToString);
37:
38:    cout << "Podaj szukany klucz: ";
39:    int Key = 0;
40:    cin >> Key;
41:
42:    auto iElementFound = umapIntToString.find(Key);
43:    if (iElementFound != umapIntToString.end())
44:    {
45:       cout << "Znaleziono! Klucz " << iElementFound->first << "
      ↪prowadzi do wartości";
46:       cout << iElementFound->second << endl;
47:    }
48:    else
49:       cout << "Klucz nie ma odpowiadającej mu wartości w obiekcie map!" <<
      ↪endl;
```

```
50:
51:     return 0;
52: }
```

Wynik ▼

```
Liczba par, size() = 8
Maksymalna liczba kubełków = 8
Współczynnik wypełniania: 1
Maksymalny współczynnik wypełniania = 1
Zawartość nieposortowanego obiektu map:
-1000 -> Minus Tysiąc
1001 -> Tysiąc Jeden
1 -> Jeden
-100 -> Minus Sto
45 -> Czterdzieści pięć
-2 -> Minus Dwa
12 -> Dwanaście
100 -> Sto
Wstawienie jeszcze jednego elementu
Liczba par, size() = 9
Maksymalna liczba kubełków = 64
Współczynnik wypełniania: 0.140625
Maksymalny współczynnik wypełniania = 1
Zawartość nieposortowanego obiektu map:
1 -> Jeden
-1000 -> Minus Tysiąc
1001 -> Tysiąc Jeden
-100 -> Minus Sto
45 -> Czterdzieści pięć
-2 -> Minus Dwa
300 -> Trzysta
12 -> Dwanaście
100 -> Sto
100 -> Sto
Podaj szukany klucz: 300
Znaleziono! Klucz 300 prowadzi do wartości Trzysta
```

Analiza ▼

Spójrz na dane wyjściowe i zwróć uwagę, że obiekt `unordered_map` na początku zawiera osiem kubełków wypełnionych ośmioma parami klucz-wartość. Po wstawieniu dziewiątej pary następuje zwiększenie liczby dostępnych kubełków do 64. Zobacz, w jaki sposób w wierszach od 9. do 11. wykorzystano metody `max_bucket_count()`, `load_factor()` i `max_load_factor()`. Poza wymienionymi metodami pozostała część kodu praktycznie nie różni się

od kodu podobnego programu, w którym zastosowano obiekt `std::map`. W wierszu 42. pokazano użycie metody `find()`, która zwraca iterator. Podobnie jak w przypadku `std::map`, także w omawianym listingu otrzymany iterator trzeba sprawdzić przed użyciem.

Ostrzeżenie

Nie polegaj na kolejności elementów w obiekcie `unoredered_map` (stąd wywodzi się nazwa — mapa nieuporządkowana). Kolejność elementu względem innych elementów w obiekcie zależy od wielu czynników, m.in. od klucza, kolejności wstawiania oraz liczby kubełków.

Omawiane kontenery są zoptymalizowane pod kątem wyszukiwania, w trakcie iteracji nie można polegać na kolejności elementów.

Uwaga

Obiekt `std::unordered_map` gwarantuje, że operacje wstawiania i wyszukiwania (gdy nie występują kolizje) będą przeprowadzane w takim samym czasie, niezależnie od liczby elementów w kontenerze. Jednak to niekoniecznie oznacza, że obiekt `std::unordered_map` jest lepszy od `std::map` zapewniającego złożoność logarytmiczną we wszystkich sytuacjach. Stały czas operacji może być dłuższy niż w `std::map`, co spowoduje, że obiekt `std::unordered_map` będzie oferował mniejszą wydajność, gdy liczba elementów w obiekcie jest mniejsza.

Bardzo ważne jest, aby decyzję o wyborze typu kontenera podjąć po przeprowadzeniu testów symulujących faktyczne użycie kontenerów w danym programie.

TAK	NIE
Używaj obiektu map wtedy, gdy konieczne jest zapewnienie unikalności kluczy w parach klucz-wartość. Używaj obiektu multimap wtedy, gdy w parach klucz-wartość mogą powtarzać się klucze (np. w książce telefonicznej). Pamiętaj, że kontenery map i multimap, podobnie jak pozostałe kontenery STL, posiadają metodę size() informującą o liczbie par przechowywanych w obiekcie. Używaj obiektów unordered_map i unordered_multimap, kiedy stały czas wstawiania i wyszukiwania elementów ma znaczenie krytyczne (zwykle wtedy, gdy liczba elementów jest bardzo duża).	Nie zapominaj, że metoda multimap::count(klucz) może podać liczbę par zindeksowanych przy użyciu klucza dostępnego w kontenerze. Nie zapominaj o sprawdzeniu wyniku działania metody find() przez jego porównanie z wynikiem działania metody end().

Podsumowanie

W tej lekcji dowiedziałeś się, w jaki sposób używać obiektów STL `map` i `multimap`, poznałeś ich ważne funkcje składowe oraz cechy charakterystyczne. Przekonałeś się, że wymienione kontenery mają złożoność logarytmiczną, a biblioteka STL dostarcza tabele hash w postaci obiektów `unordered_map` i `unordered_multimap`. Cechą tabel hash jest niezmienna wydajność operacji `insert()` i `find()`, niezależnie od wielkości kontenera. Zobaczyłeś również, jak można dostosować do własnych potrzeb kryteria sortowania za pomocą *predykatu*, co przedstawiono w przykładzie programu prostej książki telefonicznej w listingu 20.5.

Pytania i odpowiedzi

Pytanie: W jaki sposób mogę zadeklarować obiekt `map` przechowujący liczby całkowite, aby elementy były posortowane/przechowywane w kolejności malejącej?

Odpowiedź: Instrukcja `map <int>` powoduje zdefiniowanie obiektu `map` przechowującego liczby całkowite. Wykorzystuje ona domyślnie predykat sortowania `std::less <T>`, który powoduje sortowanie w kolejności rosnącej. Instrukcję tę można przedstawić również jako `map <int, less <int> >`. Dlatego więc, aby sortowanie przebiegało w kolejności malejącej, obiekt `map` należy zdefiniować jako `map <int, greater <int> >`.

Pytanie: Co się stanie, jeżeli w obiekcie `map` przechowującym ciągi tekstowe dwukrotnie wstawię ciąg tekstowy „Jacek"?

Odpowiedź: Obiekt `map` nie jest przystosowany do przechowywania wartości, które nie są unikalne. Dlatego implementacja klasy `std::map` nie pozwala na wstawienie drugiej takiej samej wartości.

Pytanie: Co powinienem zmienić w poprzednim przykładzie, jeżeli nadal będę chciał mieć dwa egzemplarze ciągu tekstowego „Jacek"?

Odpowiedź: Obiekt `map` z założenia przechowuje jedynie unikalne wartości. Rozwiązaniem będzie więc zastosowanie kontenera `multimap`.

Pytanie: Która funkcja składowa obiektu `multimap` zwraca liczbę elementów przechowujących określoną wartość w kontenerze?

Odpowiedź: Ta funkcja to `count(wartość)`.

Pytanie: W obiekcie `map` wyszukałem element za pomocą funkcji `find()` i mam iterator prowadzący do niego. Czy mogę wykorzystać otrzymany iterator, aby zmienić wartość, do której prowadzi?

Odpowiedź: Nie. Pewne implementacje STL mogą pozwalać użytkownikowi na zmianę wartości elementu w obiekcie `map` za pomocą iteratora zwróconego np. przez funkcję `find()`. Jednak to nie jest właściwy sposób działania. Iterator do elementu w obiekcie `map` powinien być używany jako iterator typu `const` — nawet jeśli implementacja STL nie wymaga takiego zachowania.

Pytanie: Używam starszej wersji kompilatora, która nie obsługuje słowa kluczowego `auto`. W jaki sposób mogę zadeklarować zmienną przechowującą wartość zwrotną `map::find()`?

Odpowiedź: Iterator zawsze jest definiowany przy użyciu poniższej składni:

```
kontener<Typ>::iterator nazwaZmiennej;
```

Dlatego też deklaracja iteratora dla obiektu `map` przechowującego liczby całkowite przedstawia się następująco:

```
std::map<int>::iterator iPairFound = mapIntegers.find(1000);
if (iPairFound != mapIntegers.end())
   ; // Dowolna operacja.
```

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Deklarujesz obiekt map przechowujący liczby całkowite (instrukcja map <int>). Która funkcja zapewnia kryteria sortowania?
2. Czy w obiekcie multimap znajdziesz powtarzające się elementy?
3. Która funkcja obiektu map lub multimap zwraca liczbę elementów przechowywanych w kontenerze?
4. Gdzie znajdziesz powtarzające się elementy w obiekcie map?

Ćwiczenia

1. Utwórz aplikację działającą jak książka telefoniczna, w której imiona osób nie muszą być unikalne. Z którego kontenera skorzystasz w tym programie? Napisz definicję tego kontenera.
2. Poniżej przedstawiono definicję wzorca obiektu map w Twojej aplikacji książki telefonicznej:

```
map <wordProperty, string, fPredicate> mapWordDefinition;
gdzie word oznacza strukturę
struct wordProperty
{
   string strWord;
   bool bIsFromLatin;
};
```

   Zdefiniuj predykat dwuargumentowy fPredicate, który pomoże obiektowi map w sortowaniu kluczy typu wordProperty zgodnie z zawartym w nim atrybutem w postaci ciągu tekstowego.
3. Zademonstruj za pomocą przykładowego programu, że obiekt map nie akceptuje powtarzających się elementów, podczas gdy obiekt multimap pozwala na stosowanie duplikatów.

Część IV

Jeszcze więcej STL

Lekcja 21

Zrozumienie obiektów funkcji

Nazwa obiektu funkcji, czyli *funktor*, może brzmieć egzotycznie bądź przerażająco, ale oznacza koncepcję C++, z którą prawdopodobnie już się spotkałeś i której pewnie używałeś, nie zdając sobie z tego sprawy.

Z tej lekcji dowiesz się:

- **koncepcję obiektów funkcji,**
- **użycie obiektów funkcji jako predykatów,**
- **sposób implementacji predykatów jedno- i dwuargumentowych za pomocą obiektów funkcji.**

Koncepcja obiektów funkcji i predykatów

Na poziomie pojęciowym obiekty funkcji są obiektami działającymi jak funkcje. Jednak na poziomie implementacji obiekty funkcji to obiekty klasy implementującej `operator()`. Choć funkcje i wskaźniki funkcji również mogą być zaklasyfikowane jako obiekty funkcji, to pojemność obiektu klasy implementującej `operator()` w celu przechowywania informacji o stanie (tzn. wartości w elementach składowych klasy) okazuje się użyteczna podczas pracy z algorytmami biblioteki STL (ang. *Standard Template Library*).

Obiekty funkcji zwykle używane przez programistę C++ podczas pracy z biblioteką STL są klasyfikowane następująco.

- **Funkcja jednoargumentowa** — funkcja wywoływana z jednym argumentem, np. `f(x)`. Kiedy funkcja jednoargumentowa zwraca typ `bool`, jest nazywana *predykatem*.
- **Funkcja dwuargumentowa** — funkcja wywoływana z dwoma argumentami, np. `f(x, y)`. Kiedy funkcja dwuargumentowa zwraca typ `bool`, jest nazywana *predykatem dwuargumentowym*.

Obiekty funkcji zwracające typ `boolean` są naturalnym wyborem do użycia w algorytmach, w których muszą być podejmowane decyzje. Obiekt funkcji łączący dwa obiekty funkcji nosi nazwę *adaptacyjnego obiektu funkcji*.

Typowe aplikacje obiektów funkcji

W celu wyjaśnienia, czym są obiekty funkcji, można wykorzystać wiele stron rozważań teoretycznych. Inny sposób objaśnienia wyglądu i sposobu działania obiektów funkcji polega na użyciu małych aplikacji przykładowych. Wykorzystajmy więc podejście praktyczne i przejdźmy prosto do świata programowania C++ z wykorzystaniem obiektów funkcji, czyli funktorów.

Funkcje jednoargumentowe

Funkcje operujące na pojedynczym parametrze są nazywane funkcjami jednoargumentowymi. Funkcja jednoargumentowa może wykonywać bardzo proste zadanie, np. wyświetlać element na ekranie. Przykład tego rodzaju funkcji przedstawiono poniżej:

```
// Funkcja jednoargumentowa.
template <typename elementType>
void FuncDisplayElement (const elementType & element)
{
   cout << element << ' ';
};
```

Funkcja `FuncDisplayElement()` przyjmuje pojedynczy parametr typu `elementType`, który jest wyświetlany za pomocą instrukcji wyświetlania w konsoli `std::cout`. Ta sama funkcja może mieć również inną postać, w której implementacja funkcji w rzeczywistości znajduje się w `operator()` *klasy* bądź *struktury*:

```
// Struktura, która zachowuje się jak funkcja jednoargumentowa.
template <typename elementType>
struct DisplayElement
{
   void operator() (const elementType& element) const
   {
      cout << element << ' ';
   }
};
```

Wskazówka

Warto zwrócić uwagę na fakt, że `DisplayElement` jest strukturą. Jeżeli byłaby to klasa, funkcja `operator()` wymagałaby modyfikatora dostępu `public`. Struktura jest podobna do klasy, w której elementy składowe domyślnie są publiczne.

Dowolna z powyższych implementacji może być użyta wraz z algorytmem STL `for_each` w celu wyświetlenia na ekranie zawartości kolekcji, po jednym elemencie, jak przedstawiono w listingu 21.1.

Listing 21.1. Wyświetlenie na ekranie zawartości kolekcji za pomocą funkcji jednoargumentowej

```
 0: #include <algorithm>
 1: #include <iostream>
 2: #include <vector>
 3: #include <list>
 4:
 5: using namespace std;
 6:
 7: // Struktura, która zachowuje się jak funkcja jednoargumentowa.
 8: template <typename elementType>
 9: struct DisplayElement
10: {
```

```
11:     void operator () (const elementType& element) const
12:     {
13:         cout << element << ' ';
14:     }
15: };
16:
17: int main ()
18: {
19:     vector <int> vecIntegers;
20:
21:     for (int nCount = 0; nCount < 10; ++ nCount)
22:         vecIntegers.push_back (nCount);
23:
24:     list <char> listChars;
25:
26:     for (char nChar = 'a'; nChar < 'k'; ++nChar)
27:         listChars.push_back (nChar);
28:
29:     cout << "Wyświetlanie obiektu vector przechowującego liczby
       ↪całkowite: " << endl;
30:
31:     // Wyświetlanie tablicy liczb całkowitych.
32:     for_each ( vecIntegers.begin ()     // Początek zakresu.
33:           , vecIntegers.end ()         // Koniec zakresu.
34:           , DisplayElement <int> () ); // Obiekt funkcji jednoargumentowej.
35:
36:     cout << endl << endl;
37:     cout << "Wyświetlanie obiektu list przechowującego znaki: " <<
       ↪endl;
38:
39:     // Wyświetlenie znaków.
40:     for_each ( listChars.begin ()         // Początek zakresu.
41:           , listChars.end ()         // Koniec zakresu.
42:           , DisplayElement <char> () ); // Obiekt funkcji jednoargumentowej.
43:
44:     return 0;
45: }
```

Wynik ▼

```
Wyświetlanie obiektu vector przechowującego liczby całkowite:
0 1 2 3 4 5 6 7 8 9

Wyświetlanie obiektu list przechowującego znaki:
a b c d e f g h i j
```

Analiza ▼

W wierszach od 8. do 15. znajduje się obiekt funkcji `DisplayElement` implementujący funkcję `operator()`. Ten obiekt funkcji jest używany w algorytmie STL `std::for_each` w wierszach od 32. do 34.; `for_each` akceptuje trzy parametry: pierwszy to punkt początkowy zakresu, drugi to punkt końcowy zakresu, natomiast trzeci to funkcja wywoływana dla każdego elementu w podanym zakresie. Innymi słowy, dla każdego elementu w obiekcie `vector` o nazwie `vecIntegers` powyższy kod wywołuje funkcję `DisplayElement::operator()`. Warto zwrócić uwagę, że zamiast używania struktury `DisplayElement` można wykorzystać również funkcję `FuncDisplayElement()`, uzyskując taki sam efekt. W wierszach od 40. do 42. pokazano to samo zadanie wykonane względem obiektu `list` przechowującego znaki.

Wskazówka

W standardzie C++11 wprowadzono wyrażenia lambda będące obiektami funkcji anonimowych.

Struktura `DisplayElement<T>` z listingu 21.1 utworzona w postaci wyrażenia lambda składa się z mniejszej ilości kodu, zawiera definicję struktury i sposób jej użycia. To wszystko zajmuje trzy wiersze w funkcji `main()`, którymi należy zastąpić wiersze od 32. do 34:

```
// Wyświetlenie tablicy liczb całkowitych przy użyciu wyrażenia lambda.
for_each ( vecIntegers.begin ()     // Początek zakresu.
   , vecIntegers.end ()             // Koniec zakresu.
   , [](int& Element) {cout << element << ' '; } ); // Wyrażenie lambda.
```

Dlatego też wyrażenia lambda to fantastyczne usprawnienie wprowadzone do C++. Zostaną omówione w lekcji 22., zatytułowanej „Wyrażenia lambda w C++11". W listingu 22.1 pokazano użycie funkcji lambda w `for_each` w celu wyświetlenia zawartości kontenera zamiast zastosowania w tym celu obiektu funkcji, jak w listingu 21.1.

Prawdziwa zaleta używania obiektu funkcji zaimplementowanego w strukturze staje się oczywista, gdy możesz użyć tego obiektu struktury do przechowywania informacji. Funkcja `FuncDisplayElement()` nie może tego zrobić w sposób podobny do rozwiązania stosowanego przez strukturę, ponieważ atrybuty elementów składowych struktury są inne niż funkcji `operator()`. Nieco zmodyfikowana wersja używająca atrybutów elementów składowych została przedstawiona poniżej:

```
template <typename elementType>
struct DisplayElementKeepCount
{
   int Count;
   DisplayElementKeepCount()   // Konstruktor.
   {
      Count = 0;
   }
   void operator () (const elementType& element)
   {
      ++ Count;
      cout << element << ' ';
   }
};
```

W powyższym fragmencie kodu konstruktor `DisplayElementKeepCount()` to nieco zmodyfikowana wersja poprzedniego. Funkcja `operator()` nie jest już funkcją składową typu `const`, ponieważ powoduje inkrementację (a więc zmianę) elementu składowego `Count` przechowującego informacje o liczbie wywołań funkcji w celu wyświetlenia danych. Wspomniany licznik jest dostępny za pomocą publicznego elementu składowego `Count`. Zalety używania tego rodzaju obiektów funkcji, które mogą również przechowywać informacje o stanie, pokazano w listingu 21.2.

Listing 21.2. Używanie obiektu funkcji w celu przechowywania informacji o stanie

```
 0: #include <algorithm>
 1: #include <iostream>
 2: #include <vector>
 3: using namespace std;
 4:
 5: template <typename elementType>
 6: struct DisplayElementKeepCount
 7: {
 8:     int Count;
 9:
10:     // Konstruktor.
11:     DisplayElementKeepCount () : Count(0) {}
12:
13:     // Wyświetlenie elementu, uaktualnienie wartości licznika!
14:     void operator () (const elementType& element)
15:     {
16:         ++ Count;
17:         cout << element << ' ';
18:     }
19: };
```

```
20:
21: int main ()
22: {
23:     vector <int> vecIntegers;
24:     for (int nCount = 0; nCount < 10; ++ nCount)
25:         vecIntegers.push_back (nCount);
26:
27:     cout << "Wyświetlanie obiektu vector przechowującego liczby
        ↪całkowite: " << endl;
28:
29:     // Wyświetlenie tablicy liczb całkowitych.
30:     DisplayElementKeepCount <int> Result;
31:     Result = for_each ( vecIntegers.begin ()        // Początek zakresu.
32:                       , vecIntegers.end ()          // Koniec zakresu.
33:                       , DisplayElementKeepCount <int> () );
                          ↪// Obiekt funkcji.
34:
35:     cout << endl << endl;
36:
37:     // Użycie informacji o stanie przechowywanej wartości zwracanej dla każdego for_each!
38:     cout << "Wyświetlono '" << Result.Count << "' elementów!" << endl;
39:
40:     return 0;
41: }
```

Wynik ▼

```
Wyświetlanie obiektu vector przechowującego liczby całkowite:
0 1 2 3 4 5 6 7 8 9
Wyświetlono '10' elementów!
```

Analiza ▼

Największa różnica między tym przykładem a poprzednim, przedstawionym w listingu 21.1, polega na użyciu `DisplayElementKeepCount` jako wartości zwracanej `for_each`. Funkcja `operator()` wyświetlająca element i inkrementująca wewnętrzny licznik w trakcie wywołania względem każdego elementu przechowuje w `Count` informację o tym, ile razy obiekt został użyty. Po wykonaniu każdej instrukcji `for_each` w wierszu 38. używamy obiektu w celu wyświetlania informacji o tym, ile razy został użyty. Warto zwrócić uwagę na fakt, że w tym rozwiązaniu zastosowano zwykłą funkcję zamiast funkcji zaimplementowanej w strukturze, która nie byłaby w stanie dostarczyć omawianej funkcji w tak bezpośredni sposób.

Predykat jednoargumentowy

Funkcja jednoargumentowa, której wartością zwracaną jest typ `bool`, to *predykat*. Tego rodzaju funkcje pomagają w podejmowaniu decyzji algorytmom STL. W listingu 21.3 przedstawiono przykładowy predykat sprawdzający, czy element danych wejściowych jest wielokrotnością wartości początkowej.

Listing 21.3. Predykat jednoargumentowy sprawdzający, czy liczba jest wielokrotnością innej

```
// Struktura jako predykat jednoargumentowy.
template <typename numberType>
struct IsMultiple
{
    numberType Divisor;

    IsMultiple (const numberType& divisor)
    {
        Divisor = divisor;
    }

    bool operator () (const numberType& element) const
    {
        // Sprawdzenie, czy dzielna jest wielokrotnością dzielnika.
        return ((element % Divisor) == 0);
    }
};
```

Analiza ▼

W omawianym przykładzie funkcja `operator()` zwraca typ `bool` i działa jako predykat jednoargumentowy. Struktura zawiera konstruktor i jest zainicjalizowana z wartością dzielnika. Wartość ta, przechowywana w obiekcie, jest następnie używana w celu sprawdzenia, czy elementy przekazywane do porównania są podzielne przez tę wartość. Jak możesz zobaczyć w implementacji funkcji `operator()`, porównanie odbywa się za pomocą matematycznego operatora reszty z dzielenia (`%`, modulo), który zwraca resztę z operacji dzielenia. Predykat porównuje tę resztę z wartością zero i w ten sposób sprawdza, czy podana liczba jest wielokrotnością wartości początkowej.

W listingu 21.4 zastosowano predykat przedstawiony w listingu 21.3, ale tym razem użyto go do sprawdzenia, czy liczby podanego zbioru są wielokrotnością dzielnika podanego przez użytkownika.

Listing 21.4. Używanie predykatu jednoargumentowego IsMultiple wraz z std::find_if() w celu wyszukania w wektorze elementu, który jest wielokrotnością dzielnika podanego przez użytkownika

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3: using namespace std;
 4: // W tym miejscu wstawiamy definicję struktury IsMultiple z listingu 21.3.
 5: int main ()
 6: {
 7:     vector <int> vecIntegers;
 8:     cout << "Obiekt vector zawiera następujące wartości przykładowe: ";
 9:
10:     // Wstawienie wartości przykładowych: 25 – 31.
11:     for (int nCount = 25; nCount < 32; ++ nCount)
12:     {
13:         vecIntegers.push_back (nCount);
14:         cout << nCount << ' ';
15:     }
16:     cout << endl << "Podaj dzielnik (> 0): ";
17:    int Divisor = 2;
18:    cin >> Divisor;
19:
20:     // Wyszukanie w kolekcji pierwszego elementu, który będzie wielokrotnością
        ↪dzielnika podanego przez użytkownika.
21:     auto iElement = find_if ( vecIntegers.begin ()
22:                               , vecIntegers.end ()
23:                               , IsMultiple <int> (Divisor) );
24:
25:     if (iElement != vecIntegers.end ())
26:     {
27:         cout << "Pierwszy element obiektu vector, który jest
            ↪wielokrotnością " << Divisor;
28:         cout << "to: " << *iElement << endl;
29:     }
30:
31:     return 0;
32: }
```

Wynik ▼

```
Obiekt vector zawiera następujące wartości przykładowe: 25 26 27 28 29 30 31
Podaj dzielnik (> 0): 4
Pierwszy element obiektu vector, który jest wielokrotnościa 4 to: 28
```

Analiza ▼

Powyższy program rozpoczyna się od definicji przykładowego obiektu `vector` zawierającego liczby całkowite. W wierszach od 11. do 15. w wymienionym kontenerze umieszczono przykładowe liczby. Predykat jednoargumentowy został wykorzystany w algorytmie `find_if`, jak pokazano w wierszu 23. W algorytmie tym obiekt funkcji `IsMultiple` został zainicjalizowany z wartością dzielnika podaną przez użytkownika i przechowywaną w zmiennej `Divisor`. Działanie algorytmu `find_if` polega na wywołaniu predykatu jednoargumentowego `IsMultiple::operator()` dla każdego elementu w podanym zakresie. Kiedy funkcja `operator()` zwraca wartość `true` dla danego elementu (czyli wtedy, gdy w omawianym przypadku element jest bez reszty podzielny przez 4), wartością zwracaną przez `find_if` jest iterator `iElement` prowadzący do tego elementu. Wynik operacji `find_if` jest porównywany z wynikiem metody `end()` kontenera w celu potwierdzenia znalezienia elementu (zobacz wiersz 25.). Iterator `iElement` jest użyty do wyświetlenia znalezionej wartości, co pokazano w wierszu 28.

Wskazówka

Aby przekonać się, jak użycie funkcji lambda może zmniejszyć ilość wierszy kod w omawianym programie, spójrz na listing 22.3 w lekcji 22.

Predykaty jednoargumentowe znajdują zastosowanie w wielu algorytmach STL, takich jak `std::partition`, który może dzielić zakres na partycje, używając predykatu, bądź `stable_partition`, wykonującego to samo zadanie, ale z zachowaniem względnej kolejności partycjonowanych elementów. Inne algorytmy STL, w których stosowany jest predykat jednoargumentowy, to funkcje wyszukiwania (np. `std::find_if`) i funkcje pomagające w usuwaniu elementów, np. `std::remove_if` usuwająca elementy w zakresie spełniającym warunki predykatu.

Funkcje dwuargumentowe

Funkcje w rodzaju `f(x, y)` są szczególnie użyteczne, kiedy zwracają wartość na podstawie dostarczonych im danych wejściowych. Tego rodzaju funkcje dwuargumentowe mogą być stosowane do przeprowadzania operacji arytmetycznych obejmujących dwa operandy, np. dodawania, mnożenia, odejmowania itd. Przykładową funkcję dwuargumentową, która zwraca

wynik mnożenia argumentów podanych jako dane wejściowe, można zapisać w postaci:

```
template <typename elementType>
class CMultiply
{
public:
   elementType operator () (const elementType& elem1,
         const elementType& elem2)
   {
      return (elem1 * elem2);
   }
};
```

Interesująca nas implementacja ponownie zawiera się w funkcji `operator()`, która przyjmuje dwa argumenty i zwraca wynik ich pomnożenia. Tego rodzaju funkcje dwuargumentowe są stosowane w algorytmach, takich jak `std::transform`, których można użyć do pomnożenia zawartości dwóch kontenerów. W listingu 21.5 pokazano przykład użycia takiej funkcji dwuargumentowej w algorytmie `std::transform`.

Listing 21.5. Użycie funkcji dwuargumentowej w celu pomnożenia dwóch zakresów wartości

```
 0: #include <vector>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: template <typename elementType>
 5: class Multiply
 6: {
 7: public:
 8:     elementType operator () (const elementType& elem1,
 9:         const elementType& elem2)
10:     {
11:         return (elem1 * elem2);
12:     }
13: };
14:
15: int main ()
16: {
17:     using namespace std;
18:
19:     // Utworzenie dwóch przykładowych obiektów vector, z których każdy przechowuje
        ➥po dziesięć liczb całkowitych.
20:     vector <int> vecMultiplicand, vecMultiplier;
21:
```

```
22:     // Wstawienie wartości przykładowych od 0 do 9.
23:     for (int nCount1 = 0; nCount1 < 10; ++ nCount1)
24:         vecMultiplicand.push_back (nCount1);
25:
26:     // Wstawienie wartości przykładowych od 100 do 109.
27:     for (int nCount2 = 100; nCount2 < 110; ++ nCount2)
28:         vecMultiplier.push_back (nCount2);
29:
30:     // Trzeci kontener będzie przechowywał wyniki operacji mnożenia.
31:     vector <int> vecResult;
32:
33:     // Zrobienie miejsca dla wyników operacji mnożenia.
34:     vecResult.resize (10);
35:     transform ( vecMultiplicand.begin (), // Zakres mnożenia.
36:                 vecMultiplicand.end (), // Koniec zakresu.
37:                 vecMultiplier.begin (),  // Wartości mnożnika.
38:                 vecResult.begin (), // Zakres przechowujący wyniki.
39:                Multiply <int> () );     // Funkcja przeprowadzająca mnożenie.
40:
41:     cout << "Zawartość pierwszego obiektu vector: " << endl;
42:     for (size_t nIndex1 = 0; nIndex1 < vecMultiplicand.size (); ++
       ↪nIndex1)
43:         cout << vecMultiplicand [nIndex1] << ' ';
44:     cout << endl;
45:
46:     cout << "Zawartość drugiego obiektu vector: " << endl;
47:     for (size_t nIndex2 = 0; nIndex2 < vecMultiplier.size (); ++nIndex2)
48:         cout << vecMultiplier [nIndex2] << ' ';
49:     cout << endl << endl;
50:
51:     cout << "Wyniki operacji mnożenia: " << endl;
52:     for (size_t nIndex = 0; nIndex < vecResult.size (); ++ nIndex)
53:         cout << vecResult [nIndex] << ' ';
54:
55:     return 0;
56: }
```

Wynik ▼

```
Zawartość pierwszego obiektu vector:
0 1 2 3 4 5 6 7 8 9
Zawartość drugiego obiektu vector:
100 101 102 103 104 105 106 107 108 109
Wyniki operacji mnożenia:
0 101 204 309 416 525 636 749 864 981
```

Analiza ▼

W wierszach od 5. do 13. znajduje się kod predykatu `Multiply`, który został przedstawiony już w poprzednim listingu. W tym programie używamy algorytmu `std::transform` w celu pomnożenia zawartości dwóch zakresów i umieszczenia wyniku operacji w trzecim. W omawianym przykładzie wykorzystywane zakresy są przechowywane w obiektach `std::vector` jako `vecMultiplicand`, `vecMultiplier` oraz `vecResult`. Innymi słowy, w wierszach od 35. do 39. używamy algorytmu `std::transform` do pomnożenia każdego elementu `vecMultiplicand` przez odpowiadający mu element obiektu `vecMultiplier`, a wynik operacji mnożenia zostaje zapisany w obiekcie `vecResult`. Sama operacja mnożenia jest przeprowadzana przez funkcję dwuargumentową `Multiple::operator()`, która jest wywoływana dla każdego elementu w obiektach `vector` tworzących zakresy źródłowy i docelowy. Wartość zwracana funkcji `operator()` zostaje umieszczona w kontenerze `vecResult`.

Omówiony przykład demonstruje zastosowanie funkcji dwuargumentowych w przeprowadzaniu operacji arytmetycznych na elementach w kontenerach STL.

Predykat dwuargumentowy

Funkcja, która akceptuje dwa argumenty i zwraca typ `bool`, to predykat dwuargumentowy. Tego rodzaju funkcje znajdują zastosowanie w algorytmach STL, takich jak `std::sort`. W listingu 21.6 zademonstrowano użycie predykatu dwuargumentowego do porównania dwóch ciągów tekstowych po zmianie ich liter na małe. Tego rodzaju predykat może być również wykorzystany np. podczas przeprowadzania nierozróżniającego wielkości znaków sortowania kontenera przechowującego wartości `std::string`.

Listing 21.6. Predykat dwuargumentowy przeprowadzający sortowanie ciągów tekstowych bez rozróżniania wielkości znaków

```
0: #include <algorithm>
1: #include <string>
2: using namespace std;
3:
4: class CompareStringNoCase
5: {
6: public:
7:    bool operator () (const string& str1, const string& str2) const
```

```
 8:    {
 9:       string str1LowerCase;
10:
11:       // Zarezerwowanie miejsca.
12:       str1LowerCase.resize (str1.size ());
13:
14:       // Konwersja każdego znaku na mały.
15:       transform ( str1.begin (), str1.end (), str1LowerCase.begin ()
16:                   , tolower );
17:
18:       string str2LowerCase;
19:       str2LowerCase.resize (str2.size ());
20:       transform ( str2.begin (), str2.end (), str2LowerCase.begin ()
21:                   , tolower);
22:
23:       return (str1LowerCase < str2LowerCase);
24:    }
25: };
```

Analiza ▼

Na początku predykat dwuargumentowy implementowany w funkcji `operator()` konwertuje znaki wejściowych ciągów tekstowych na małe, używając do tego algorytmu `std::transform` (zobacz wiersze 15. i 20.). Dopiero później przeprowadza porównywanie ciągów tekstowych za pomocą operatora <.

Chociaż ten predykat dwuargumentowy może być użyty wraz z algorytmem `std::sort`, może być również dostarczony jako parametr predykatu kontenerów asocjacyjnych, takich jak `std::set`, co przedstawiono w listingu 21.7.

Listing 21.7. Użycie obiektu funkcji klasy CompareStringNoCase do przeprowadzenia nierozróżniającego wielkości znaków sortowania obiektu vector przechowującego ciągi tekstowe

```
 0: // W tym miejscu wstaw klasę CompareStringNoCase z listingu 21.6.
 1: #include <vector>
 2: #include <iostream>
 3:
 4: template <typename T>
 5: void DisplayContents (const T& Input)
 6: {
 7:    for(auto iElement = Input.cbegin() // auto, cbegin i cend: c++11.
 8:       ; iElement != Input.cend ()
 9:       ; ++ iElement )
```

```
10:       cout << *iElement << endl;
11: }
12:
13: int main ()
14: {
15:     // Zdefiniowanie zbioru ciągów tekstowych przechowujących imiona.
16:     vector<string> vecNames;
17:
18:     // Wstawienie do zbioru kilku przykładowych imion.
19:     vecNames.push_back ("jan");
20:     vecNames.push_back ("Jacek");
21:     vecNames.push_back ("Stefan");
22:     vecNames.push_back ("Anna");
23:
24:     cout << "Imiona w zbiorze wymienione w kolejności wstawiania: " <<
       ↪endl;
25:     DisplayContents(vecNames);
26:
27:    cout << "Imiona posortowane przy użyciu domyślnego predykatu
       ↪std::less<>: " << endl;
28:    sort(vecNames.begin(), vecNames.end());
29:    DisplayContents(vecNames);
30:
31:    cout << "Imiona posortowane przy użyciu domyślnego predykatu
       ↪ignorującego wielkość liter:" << endl;
32:    sort(vecNames.begin(), vecNames.end(), CompareStringNoCase());
33:    DisplayContents(vecNames);
34:
35:     return 0;
36: }
```

Wynik ▼

```
Imiona w zbiorze wymienione w kolejności wstawiania:
jan
Jacek
Stefan
Anna
Imiona posortowane przy użyciu domyślnego predykatu std::less<>:
Anna
Jacek
Stefan
jan
Imiona posortowane przy użyciu domyślnego predykatu ignorującego wielkość
liter:
Anna
Jacek
jan
Stefan
```

Analiza ▼

Dane wyjściowe pokazują zawartość kontenera na trzech etapach. Najpierw zostaje wyświetlona zawartość w kolejności wstawiania elementów. Następnie widzimy zawartość po jej posortowaniu (patrz wiersz 28.) przy użyciu domyślnego predykatu sortowania `less<T>`. Jak widzisz, element `jan` nie został umieszczony po elemencie `Jacek`, ponieważ sortowanie rozróżnia wielkość liter i jest przeprowadzane przy użyciu metody `string::operator<`. W wierszu 32. przeprowadzane jest sortowanie z wykorzystaniem predykatu `CompareStringNoCase<>` (zaimplementowanego w listingu 21.6), który gwarantuje, że element `jan` zostanie umieszczony po elemencie `Jacek`, niezależnie od wielkości liter w nazwach elementów.

Predykaty dwuargumentowe są wymagane przez wiele algorytmów STL, np. przez `std::unique`, który usuwa powielające się sąsiednie elementy, `std::sort`, który sortuje zawartość, `std::stable_sort`, który sortuje z zachowaniem względnej kolejności elementów, oraz `std::transform`, który przeprowadza operacje na dwóch zakresach będących algorytmami STL wymagającymi predykatu dwuargumentowego.

Podsumowanie

Dzięki tej lekcji uzyskałeś ogólny wgląd do świata funktorów, czyli obiektów funkcji. Dowiedziałeś się, że obiekty funkcji po ich zaimplementowaniu w strukturze lub klasie są znacznie bardziej użyteczne niż zwykłe funkcje, ponieważ mogą być wykorzystane również do przechowywania informacji. Otrzymałeś ogólne informacje na temat predykatów, które są specjalnymi klasami obiektów funkcji, a także zobaczyłeś kilka praktycznych przykładów ich użycia.

Pytania i odpowiedzi

Pytanie: Predykat to specjalna kategoria obiektu funkcji. Co powoduje, że jest specjalna?

Odpowiedź: Predykaty zawsze zwracają wartość typu `bool`.

Pytanie: Jakiego rodzaju obiektu funkcji powinienem użyć w wywołaniu funkcji, takiej jak `remove_if`?

Odpowiedź: Powinieneś użyć predykatu jednoargumentowego, który za pomocą konstruktora pobiera wartość do przetworzenia jako wartość początkową.

Pytanie: Jaki rodzaj obiekt funkcji powinienem zastosować podczas pracy z obiektem `map`?

Odpowiedź: Powinieneś zastosować predykat dwuargumentowy.

Pytanie: Czy jest możliwe, aby prosta funkcja, która nie zwraca wartości, mogła zostać użyta jako predykat?

Odpowiedź: Tak. Funkcja, która nie zwraca wartości, nadal może być bardzo użyteczna. Przykładowo może wyświetlać dane wejściowe.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Jakim mianem określa się funkcję jednoargumentową, która zwraca wartość w postaci typu `bool`?
2. Jak można wykorzystać obiekt funkcji, który ani nie modyfikuje danych, ani nie zwraca wartości typu `bool`? Czy możesz przedstawić wyjaśnienie, używając przykładu?
3. Jaka jest definicja pojęcia *obiekty funkcji*?

Ćwiczenia

1. Utwórz funkcję jednoargumentową, która może być użyta wraz z `std::for_each` w celu wyświetlenia podwojonej wartości parametru wejściowego.
2. Rozbuduj powyższy predykat w taki sposób, aby podawał także, ile razy został użyty.
3. Utwórz predykat dwuargumentowy pomagający w sortowaniu w kolejności rosnącej.

Lekcja 22

Wyrażenia lambda w C++11

Wyrażenia lambda pozwalają na zwięzłe definiowanie i konstruowanie obiektów funkcji anonimowych. Wyrażenia te są nowością wprowadzoną w standardzie C++11.

Z tej lekcji dowiesz się:

- **jak tworzyć wyrażenia lambda,**
- **jak używać wyrażeń lambda w charakterze predykatów,**
- **jak tworzyć wyrażenia lambda, które mogą przechowywać informacje o stanie i manipulować nimi.**

Czym są wyrażenia lambda?

Wyrażenie lambda można określić jako krótszą wersję struktury (lub klasy) anonimowej zawierającej publiczną implementację funkcji `operator()`.
W takim sensie wyrażenie lambda jest obiektem funkcji, takim jak przedstawione w lekcji 21., zatytułowanej „Zrozumienie obiektów funkcji”. Zanim przystąpimy do analizy tworzenia funkcji lambda, jeszcze na chwilę powrócimy do obiektu funkcji z listingu 21.1 znajdującego się w lekcji 21.:

```
// Struktura, która zachowuje się jak funkcja jednoargumentowa.
template <typename elementType>
struct DisplayElement
{
   void operator () (const elementType& element) const
   {
      cout << element << ' ';
   }
};
```

Przedstawiona funkcja wyświetla element na ekranie i najczęściej jest stosowana w algorytmach, takich jak `std::for_each`:

```
// Wyświetlenie tablicy liczb całkowitych.
for_each ( vecIntegers.begin ()     // Początek zakresu.
   , vecIntegers.end ()             // Koniec zakresu.
   , DisplayElement <int> () );     // Obiekt funkcji jednoargumentowej.
```

Wyrażenie lambda pozwala na zmniejszenie do trzech liczby wierszy całego kodu obejmującego także definicję obiektu funkcji:

```
// Wyświetlenie tablicy liczb całkowitych przy użyciu wyrażenia lambda.
for_each ( vecIntegers.begin ()     // Początek zakresu.
   , vecIntegers.end ()             // Koniec zakresu.
   , [](int& Element) {cout << element << ' '; } ); // Wyrażenie lambda.
```

Kiedy kompilator napotyka wyrażenie lambda, czyli w omawianym przypadku:

```
[](int Element) {cout << element << ' '; }
```

wtedy automatycznie rozwija je na postać podobną do struktury `DisplayElement<int>`:

```
struct NoName
{
   void operator () (const int& element) const
   {
      cout << element << ' ';
   }
};
```

W jaki sposób zdefiniować wyrażenie lambda?

Definicja wyrażenia lambda musi rozpoczynać się od nawiasów kwadratowych `[]`. Nawiasy informują kompilator o początku wyrażenia lambda. Następnie powinna znajdować się lista parametrów, dokładnie taka sama jak stosowana w implementacji funkcji `operator()`, w sytuacji gdy nie jest używane wyrażenie lambda.

Wyrażenie lambda dla funkcji jednoargumentowej

Wersja lambda funkcji `operator(Type)` pobierającej jeden argument będzie miała następującą postać:

```
[](Type paramName) {   // Kod wyrażenia lambda;   }
```

Zwróć uwagę na możliwość przekazania parametru przez referencję:

```
[](Type& paramName) {   // Kod wyrażenia lambda;   }
```

Przedstawiony poniżej listing 22.1 wykorzystaj do przestudiowania sposobu użycia funkcji lambda do wyświetlenia zawartości kontenera STL przy użyciu algorytmu `for_each`.

Listing 22.1. Wyświetlenie elementów kontenera przy użyciu algorytmu for_each() wywoływanego w wyrażeniu lambda zamiast w obiekcie funkcji

```
 0: #include <algorithm>
 1: #include <iostream>
 2: #include <vector>
 3: #include <list>
 4:
 5: using namespace std;
 6:
 7: int main ()
 8: {
 9:    vector <int> vecIntegers;
10:
11:    for (int nCount = 0; nCount < 10; ++ nCount)
12:       vecIntegers.push_back (nCount);
13:
14:    list <char> listChars;
```

```
15:    for (char nChar = 'a'; nChar < 'k'; ++nChar)
16:       listChars.push_back (nChar);
17:
18:    cout << "Użycie wyrażenia lambda do wyświetlenia zawartości obiektu
       ↪vector przechowującego liczby całkowite: " << endl;
19:
20:    // Wyświetlenie tablicy .
21:    for_each ( vecIntegers.begin ()     // Początek zakresu.
22:             , vecIntegers.end ()         // Koniec zakresu..
23:             , [](int& element) {cout << element << ' '; } );
                 ↪// Wyrażenie lambda.
24:
25:    cout << endl << endl;
26:    cout << "Użycie wyrażenia lambda do wyświetlenia zawartości obiektu
       ↪list przechowującego ciągi tekstowe: " << endl;
27:
28:    // Wyświetlenie listy znaków.
29:    for_each ( listChars.begin ()         // Początek zakresu.
30:             , listChars.end ()         // Koniec zakresu.
31:             , [](char& element) {cout << element << ' '; } );
                 ↪// Wyrażenie lambda.
32:
33:    cout << endl;
34:
35:    return 0;
36: }
```

Wynik ▼

```
Użycie wyrażenia lambda do wyświetlenia zawartości obiektu vector
przechowującego liczby całkowite:
0 1 2 3 4 5 6 7 8 9
Użycie wyrażenia lambda do wyświetlenia zawartości obiektu list
przechowującego ciągi tekstowe:
a b c d e f g h i j
```

Analiza ▼

Dwa interesujące nas wyrażenia lambda znajdują się w wierszach 23. i 31. Są podobne, ale różnią się typem parametru danych wejściowych, ponieważ zostały dostosowane do natury elementów w obsługiwanych kontenerach. Pierwsze wyrażenie lambda pobiera parametr typu `int` i jest używane do wyświetlenia elementów obiektu `vector`. Natomiast drugie pobiera parametr typu `char` i wyświetla elementy typu `char` przechowywane w obiekcie `std::list`.

Uwaga

Nie ma żadnego zbiegu okoliczności w tym, że dane wyjściowe wygenerowane przez listing 22.1 są takie same jak otrzymane po uruchomieniu programu z listingu 21.1. W rzeczywistości, omówiony powyżej program to po prostu wersja listingu 21.1, ale z funkcją `DisplayElement<T>` opartą na wyrażeniu lambda.

Jeśli porównasz obie wersje programu, przekonasz się, jak wielki potencjał drzemie w wyrażeniach lambda, dzięki którym kod C++ staje się prostszy i bardziej zwięzły.

Wyrażenie lambda dla predykatu jednoargumentowego

Predykat pomaga w podejmowaniu decyzji. Predykat jednoargumentowy to wyrażenie jednoargumentowe, które zwraca wartość boolowską, czyli `true` lub `false`. Wyrażenie lambda również może zwracać wartości. Przykładowo przedstawione poniżej wyrażenie lambda zwraca wartość `true` dla liczb parzystych:

```
[](int& Num) {return ((Num % 2) == 0); }
```

W tym przypadku natura wartości zwrotnej informuje kompilator, że wyrażenie lambda zwróciło wartość boolowską (`bool`).

Wyrażenia lambda będącego predykatem jednoargumentowym można użyć w algorytmach, takich jak `std::find_if()`, np. w celu wyszukania liczb parzystych w zbiorze. Tego rodzaju program przedstawiono w listingu 22.2.

Listing 22.2. Wyszukanie liczb parzystych w zbiorze przy użyciu wyrażenia lambda dla predykatu jednoargumentowego i algorytmu std::find_if()

```
 0: #include<algorithm>
 1: #include<vector>
 2: #include<iostream>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    vector<int> vecNums;
 8:    vecNums.push_back(25);
 9:    vecNums.push_back(101);
10:    vecNums.push_back(2011);
11:    vecNums.push_back(-50);
12:
```

```
13:    auto iEvenNum = find_if( vecNums.cbegin()
14:                           , vecNums.cend()   // Przeszukiwany zakres.
15:                           , [](const int& Num){return ((Num % 2) ==
                              ↪0); } );
16:
17:    if (iEvenNum != vecNums.cend())
18:       cout << "Liczba parzysta w zbiorze to: " << *iEvenNum << endl;
19:
20:    return 0;
21: }
```

Wynik ▼

```
Liczba parzysta w zbiorze to: -50
```

Analiza ▼

Wyrażenie lambda działające jako predykat jednoargumentowy zostało przedstawione w wierszu 15. Algorytm `find_if()` wywołuje predykat jednoargumentowy dla każdego elementu zbioru. Kiedy predykat zwróci wartość `true`, algorytm `find_if()` informuje o znalezieniu odpowiedniej liczby, zwracając iterator prowadzący do danego elementu. W omawianym przykładzie wyrażenie lambda zwraca wartość `true`, gdy algorytm `find_if()` będzie wywołany z parzystą liczbą całkowitą (tzn. reszta z dzielenia przez dwa wynosi zero).

Uwaga

W listingu 22.2 pokazano nie tylko wyrażenie lambda użyte w charakterze predykatu jednoargumentowego, ale także zastosowanie `const` w wyrażeniu lambda.

Pamiętaj o użyciu `const` względem parametrów danych wejściowych, zwłaszcza gdy są przekazywane przez referencję.

Wyrażenie lambda wraz ze stanem przy użyciu listy przechwytywania [...]

W listingu 22.2 utworzono predykat jednoargumentowy, który zwracał wartość `true`, kiedy liczba całkowita była podzielna bez reszty przez dwa, czyli była liczbą parzystą. Co zrobić w sytuacji, gdy chcesz otrzymać znacznie bardziej ogólną funkcję zwracającą wartość `true`, jeśli sprawdzana liczba jest bez reszty podzielna przez dzielnik podany przez użytkownika? Konieczne jest zapewnienie obsługi „stanu” — dzielnika — w wyrażeniu:

```
int Divisor = 2; // Wartość początkowa.
...
auto iElement = find_if ( początek zakresu
   , koniec zakresu
   , [Divisor](int dividend){return (dividend % Divisor) == 0; } );
```

Lista argumentów przekazywana jako zmienne stanu `[...]` jest nazywana także listą przechwytywania lambda.

Uwaga

Tego rodzaju wyrażenie lambda to jednowierszowy odpowiednik znajdującego się w listingu 21.3 fragmentu szesnastu wierszy kodu odpowiedzialnego za zdefiniowanie predykatu jednoargumentowego `struct IsMultiple<>`.

Wyraźnie widać, że wyrażenia lambda w ogromnym stopniu poprawiają wydajność programowania podczas stosowania standardu C++11.

W listingu 22.3 zaprezentowano wykorzystanie predykatu jednoargumentowego dla danej zmiennej stanu, używanego do wyszukania w zbiorze liczby, która bez reszty dzieli się przez dzielnik podany przez użytkownika.

Listing 22.3. Przykład użycia wyrażenia lambda przechowującego informacje o stanie i sprawdzającego, czy liczba dzieli się bez reszty przez inną liczbę

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3: using namespace std;
 4:
 5: int main ()
 6: {
 7:    vector <int> vecIntegers;
 8:    cout << "Obiekt vector zawiera następujące wartości przykładowe: ";
 9:
10:    // Wstawienie wartości przykładowych: 25 – 31.
11:    for (int nCount = 25; nCount < 32; ++ nCount)
12:    {
13:       vecIntegers.push_back (nCount);
14:       cout << nCount << ' ';
15:    }
16:    cout << endl << "Podaj dzielnik (> 0): ";
17:    int Divisor = 2;
18:    cin >> Divisor;
19:
20:    // Wyszukanie w kolekcji pierwszego elementu, który będzie wielokrotnością dzielnika
       ↪podanego przez użytkownika.
21:    vector <int>::iterator iElement;
```

```
22:    iElement = find_if ( vecIntegers.begin ()
23:                       , vecIntegers.end ()
24:           , [Divisor](int dividend){return (dividend % Divisor) == 0; }
);
25:
26:    if (iElement != vecIntegers.end ())
27:    {
28:       cout << "Pierwszy element obiektu vector, który jest
       ↪wielokrotnością " << Divisor;
29:       cout << "to: " << *iElement << endl;
30:    }
31:
32:    return 0;
33: }
```

Wynik ▼

```
Obiekt vector zawiera następujące wartości przykładowe: 25 26 27 28 29 30 31
Podaj dzielnik: 4
Pierwszy element obiektu vector, który jest wielokrotnością 4 to: 28
```

Analiza ▼

Wyrażenie lambda zawierające informacje o stanie i działające w charakterze predykatu zostało przedstawione w wierszu 24. Dzielnik to zmienna stanu, którą można porównać do `IsMultiple::Divisor` w listingu 21.3. Zmienne stanu są więc zbliżone do elementów składowych w obiekcie funkcji stosowanym przed wprowadzeniem standardu C++11. Obecnie masz możliwość przekazania wyrażeniu lambda informacji o stanie oraz dostosowania sposobu wykorzystania tych danych do własnych potrzeb.

Uwaga

> Listing 22.3 to oparty na wyrażeniu lambda odpowiednik listingu 21.4, ale bez użycia obiektu funkcji. Zmniejszenie kodu o szesnaście wierszy to zasługa użycia jednej z nowych funkcji wprowadzonych w standardzie C++11.

Ogólna składnia wyrażeń lambda

Wyrażenie lambda zawsze rozpoczyna się od nawiasów kwadratowych i może być skonfigurowane do pobierania wielu zmiennych stanu rozdzielonych przecinkami w liście przechwytywania `[...]`:

`[ZmiennaStanu1, ZmiennaStanu2](Typ& parametr) { // Kod wyrażenia lambda; }`

Jeżeli chcesz mieć pewność, że wspomniane zmienne stanu są modyfikowane w wyrażeniu lambda, powinieneś dodać słowo kluczowe `mutable`:

```
[ZmiennaStanu1, ZmiennaStanu2](Typ& parametr) mutable { // Kod wyrażenia lambda; }
```

Zauważ, że zmienne podawane w liście przechwytywania są modyfikowalne w wyrażeniu lambda, ale ich zmiana nie ma żadnego efektu poza wyrażeniem. Jeżeli chcesz mieć pewność, że modyfikacja wprowadzona w zmiennej stanu w wyrażeniu lambda będzie obowiązywała również na zewnątrz wyrażenia, powinieneś skorzystać z referencji:

```
[&ZmiennaStanu1, &ZmiennaStanu2](Typ& parametr) { // Kod wyrażenia lambda; }
```

Wyrażenia lambda mogą pobierać wiele rozdzielonych przecinkami parametrów danych wejściowych:

```
[ZmiennaStanu1, ZmiennaStanu2](Typ1& zmienna1, Typ2& zmienna2) {
↪// Kod wyrażenia lambda; }
```

Jeśli chcesz podać konkretny typ wartości zwrotnej, który bez żadnych wątpliwości zostanie zinterpretowany przez kompilator, użyj `->`, tak jak w poniższym przykładzie:

```
[Stan1, Stan2](Typ1 zmienna1, Typ2 zmienna2) -> TypWartościZwrotnej
{ return (wartość lub wyrażenie); }
```

Wreszcie, polecenia złożone {} mogą składać się z wielu poleceń, każde zakończone średnikiem, co przedstawiono poniżej:

```
[Stan1, Stan2](Typ1 zmienna1, Typ2 zmienna2) -> TypWartościZwrotnej
{ Polecenie 1; Polecenie 2; return (wartość lub wyrażenie); }
```

Uwaga

Jeżeli wyrażenie lambda obejmuje kilka wierszy, koniecznie musisz podać jego typ wartości zwrotnej.

W listingu 22.5 znajdującym się w dalszej części tej lekcji zaprezentowano wyrażenie lambda określające typ wartości zwrotnej oraz zdefiniowane w więcej niż jednym wierszu.

Funkcja lambda może być zwięzłym, w pełni funkcjonalnym odpowiednikiem obiektu funkcji, np. takiego jak przedstawiony poniżej:

```
template<typename Type1, typename Type2>
struct IsNowTooLong
{
   // Zmienne stanu.
   Typ1 zmieenna1;
```

```
   Typ2 zmienna2;

   // Konstruktor.
   IsNowTooLong(const Typ1& in1, Typ2& in2): zmienna1(in1), zmienna2(in2) {};


   // Właściwe polecenia.
   TypWartościZwrotnej operator()
   {
      Polecenie 1;
      Polecenie 2;
      return (wartość lub wyrażenie);
   }
};
```

Wyrażenie lambda dla funkcji dwuargumentowej

Funkcja dwuargumentowa pobiera dwa parametry oraz opcjonalnie zwraca wartość. Oparty na wyrażeniu lambda odpowiednik tego rodzaju funkcji przedstawia się następująco:

```
[...](Typ1& nazwaParametru1, Typ2& nazwaParametru2) {  // Kod wyrażenia lambda;  }
```

Funkcja lambda odpowiedzialna za mnożenie elementów dwóch wektorów o takiej samej wielkości przy użyciu `std::transform` i przechowująca wynik w trzecim wektorze została przedstawiona w listingu 22.4.

Listing 22.4. Wyrażenie lambda jako funkcja dwuargumentowa mnożąca elementy dwóch kontenerów i przechowująca wynik w trzecim

```
 0: #include <vector>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: int main ()
 5: {
 6:    using namespace std;
 7:
 8:    // Utworzenie dwóch przykładowych obiektów vector, z których każdy przechowuje
      ↪po dziesięć liczb całkowitych.
 9:    vector <int> vecMultiplicand, vecMultiplier;
10:
11:    // Wstawienie wartości przykładowych od 0 do 9.
12:    for (int nCount1 = 0; nCount1 < 10; ++ nCount1)
13:       vecMultiplicand.push_back (nCount1);
```

```
14:
15:    // Wstawienie wartości przykładowych od 100 do 109.
16:    for (int nCount2 = 100; nCount2 < 110; ++ nCount2)
17:       vecMultiplier.push_back (nCount2);
18:
19:    // Trzeci kontener będzie przechowywał wyniki operacji mnożenia.
20:    vector <int> vecResult;
21:
22:    // Zrobienie miejsca dla wyników operacji mnożenia.
23:    vecResult.resize (10);
24:
25:    transform ( vecMultiplicand.begin (), // Zakres mnożenia.
26:                vecMultiplicand.end (), // Koniec zakresu.
27:                vecMultiplier.begin (),  // Wartości mnożnika.
28:                vecResult.begin (), // Zakres przechowujący wyniki.
29:                [](int a, int b) {return a * b; } );  //Wyrażenie lambda.
30:
31:    cout << "Zawartość pierwszego obiektu vector: " << endl;
32:    for (size_t nIndex1 = 0; nIndex1 < vecMultiplicand.size (); ++ nIndex1)
33:       cout << vecMultiplicand [nIndex1] << ' ';
34:    cout << endl;
35:
36:    cout << "Zawartość drugiego obiektu vector: " << endl;
37:    for (size_t nIndex2 = 0; nIndex2 < vecMultiplier.size (); ++nIndex2)
38:       cout << vecMultiplier [nIndex2] << ' ';
39:    cout << endl << endl;
40:
41:    cout << "Wyniki operacji mnożenia: " << endl;
42:    for (size_t nIndex = 0; nIndex < vecResult.size (); ++ nIndex)
43:       cout << vecResult [nIndex] << ' ';
44:    cout << endl;
45:
46:    return 0;
47: }
```

Wynik ▼

```
Zawartość pierwszego obiektu vector:
0 1 2 3 4 5 6 7 8 9
Zawartość drugiego obiektu vector:
100 101 102 103 104 105 106 107 108 109
Wyniki operacji mnożenia:
0 101 204 309 416 525 636 749 864 981
```

Analiza ▼

Wyrażenie lambda znajduje się w wierszu 29. i jest parametrem dla `std::transform`. Algorytm pobiera dane wejściowe w postaci dwóch zakresów wartości, a następnie przekształca go zgodnie z algorytmem zdefiniowanym w funkcji dwuargumentowej. Wartość zwrotna wspomnianej funkcji jest przechowywana w kontenerze docelowym. Przedstawiona tutaj funkcja dwuargumentowa to wyrażenie lambda pobierające dane wejściowe w postaci dwóch liczb całkowitych i zwracające wynik mnożenia. Otrzymany wynik jest przez `std::transform` umieszczany w kontenerze `vecResult`. Dane wyjściowe programu pokazują zawartość dwóch kontenerów danych wejściowych oraz trzeci kontener, którego elementy powstały na skutek mnożenia liczb w pochodzących z dwóch pierwszych kontenerów.

Uwaga

> Listing 22.4 to oparty na wyrażeniu lambda odpowiednik listingu 21.5, w którym użyto obiektu funkcji `class Multiply<>`.

Wyrażenie lambda dla predykatu dwuargumentowego

Funkcja dwuargumentowa zwracająca wartość `true` lub `false` pomagająca w podjęciu decyzji nosi nazwę predykatu dwuargumentowego. Tego rodzaju predykaty są stosowane w algorytmach, takich jak `std::sort()`, który wywołuje predykat dwuargumentowy dla dwóch dowolnych wartości w kontenerze i ustala właściwą kolejność ich umieszczenia. Ogólna składnia predykatu dwuargumentowego jest następująca:

```
[...](Typ1& nazwaParametru1, Typ2& nazwaParametru2) {
↪// Zwraca wyrażenie typu bool; }
```

Użycie wyrażenia lambda w operacji sortowania przedstawiono w listingu 22.5.

Listing 22.5. Wyrażenie lambda jako predykat dwuargumentowy w std::sort() pozwalający na przeprowadzenie sortowania ignorującego wielkość znaków

```
0: #include <algorithm>
1: #include <string>
2: #include <vector>
3: #include <iostream>
4: using namespace std;
```

```
 5:
 6: template <typename T>
 7: void DisplayContents (const T& Input)
 8: {
 9:    for(auto iElement = Input.cbegin() // auto, cbegin i cend: c++11.
10:       ; iElement != Input.cend ()
11:       ; ++ iElement )
12:       cout << *iElement << endl;
13: }
14:
15: int main ()
16: {
17:    // Zdefiniowanie obiektu vector przechowującego ciągi tekstowe zawierające imiona.
18:    vector <string> vecNames;
19:
20:    // Wstawienie do zbioru kilku przykładowych imion.
21:    vecNames.push_back ("jan");
22:    vecNames.push_back ("Jacek");
23:    vecNames.push_back ("Stefan");
24:    vecNames.push_back ("Anna");
25:
26:    cout << "Imiona w zbiorze wymienione w kolejności wstawiania: " << endl;
27:    DisplayContents(vecNames);
28:
29:    cout << "Imiona posortowane przy użyciu domyślnego predykatu
       ↪std::less<>: " << endl;
30:    sort(vecNames.begin(), vecNames.end());
31:    DisplayContents(vecNames);
32:
33:    cout << "Imiona posortowane przy użyciu domyślnego predykatu
       ↪ignorującego wielkość liter:" << endl;
34:    sort(vecNames.begin(), vecNames.end(),
35:       [](const string& str1, const string& str2) -> bool  // Wyrażenie lambda.
36:          {
37:           string str1LowerCase;
38:
39:           // Zarezerwowanie miejsca.
40:           str1LowerCase.resize (str1.size ());
41:
42:           // Konwersja każdego znaku na mały.
43:           transform(str1.begin(), str1.end(),
              ↪str1LowerCase.begin(),tolower);
44:
45:           string str2LowerCase;
46:           str2LowerCase.resize (str2.size ());
47:           transform (str2.begin (), str2.end (), str2LowerCase.begin (),
48:                      tolower);
49:
50:           return (str1LowerCase < str2LowerCase);
```

```
51:           }   // Koniec wyrażenia lambda.
52:       );   // Koniec funkcji sortującej.
53:    DisplayContents(vecNames);
54:
55:    return 0;
56: }
```

Wynik ▼

```
Imiona w zbiorze wymienione w kolejności wstawiania:
jan
Jacek
Stefan
Anna
Imiona posortowane przy użyciu domyślnego predykatu std::less<>:
Anna
Jacek
Stefan
jan
Imiona posortowane przy użyciu domyślnego predykatu ignorującego wielkość
↪liter:
Anna
Jacek
jan
Stefan
```

Analiza ▼

W listingu zaprezentowano całkiem dużą funkcję lambda zdefiniowaną w wierszach od 35. do 51. Ta funkcja to trzeci parametr `std::sort()`; rozpoczyna się w wierszu 34., a kończy w 52. Jak możesz się przekonać, funkcja lambda może zawierać wiele poleceń, a wymaganie w postaci podania typu wartości zwrotnej zostało spełnione w wierszu 35. (określony typ `bool`). Dane wyjściowe pokazują zawartość obiektu `vector`, w którym element `jan` został wstawiony przed elementem `Jacek`. Zawartość obiektu `vector` po przeprowadzeniu sortowania domyślnego bez użycia zdefiniowanego wyrażenia lambda lub predykatu odbywa się w wierszu 30. Element `jan` został umieszczony po elemencie `Stefan`, ponieważ `string::operator<` rozróżnia wielkość liter. W wierszach od 34. do 52. mamy wyrażenie lambda ignorujące wielkość liter, a więc element `jan` został, zgodnie z oczekiwaniami, umieszczony po elemencie `Jacek`. Ta operacja sortowania wykorzystuje wyrażenie lambda zdefiniowane w wielu wierszach.

Uwaga

Zaprezentowane w listingu 22.5 ogromne wyrażenie lambda to oparta na wyrażeniu lambda wersja przedstawionej w listingu 21.6 klasy `CompareStringNoCase`, wykorzystanej w listingu 21.7.

Bez wątpienia to nie jest optymalne użycie wyrażenia lambda, ponieważ w tym przypadku obiekt funkcji można ponownie wykorzystać w wielu poleceniach `std::sort()` — jeśli potrzeba — a także w innych algorytmach wymagających predykatu dwuargumentowego.

Tak więc najlepiej używaj krótkich i efektywnych wyrażeń lambda.

TAK	NIE
Pamiętaj, że wyrażenia lambda zawsze zaczynają się od nawiasów kwadratowych [], np. [stan1, stan2, ...].	Nie zapominaj, że wyrażenia lambda to anonimowa postać klasy lub struktury wraz ze zdefiniowaną funkcją operator().
Pamiętaj, że domyślnie zmienne stanu podane w liście przechwytywania [...] są niemodyfikowalne. Jeśli chcesz mieć możliwość ich modyfikacji, musisz użyć słowa kluczowego mutable.	Nie zapominaj o prawidłowym użyciu typu const dla parametrów podczas tworzenia wyrażeń lambda: [](const T& value) { // *wyrażenie lambda;* }.
	Nie zapominaj o wyraźnym określeniu typu wartości zwrotnej wyrażenia lambda zawierającego wiele poleceń w bloku poleceń {}.
	Nie wybieraj wyrażeń lambda zamiast obiektu funkcji, gdy wyrażenie lambda staje się duże i zawiera wiele poleceń. W takich przypadkach polecenia będą musiały być ponownie definiowane przy każdym użyciu, co uniemożliwia ponowne wykorzystanie tego samego kodu.

Podsumowanie

W tej lekcji poznałeś bardzo ważną funkcję w standardzie C++11, czyli wyrażenia lambda. Przekonałeś się, że wyrażenia lambda to praktycznie anonimowe obiekty funkcji, które mogą pobierać parametry, przechowują informacje o stanie, mają możliwość zwrotu wartości i mogą być zdefiniowane w wielu wierszach. Zobaczyłeś, że w algorytmach STL wyrażenia lambda można używać zamiast obiektów funkcji, ponieważ pomagają w operacjach `find()`, `sort()` i `transform()`. Dzięki wyrażeniom lambda programowanie w C++ staje się szybkie i efektywne, więc powinieneś je wypróbować i stosować, gdy tylko istnieje taka możliwość.

Pytania i odpowiedzi

Pytanie: Czy zawsze powinienem stosować wyrażenie lambda zamiast obiektu funkcji?

Odpowiedź: Ogromne, zajmujące wiele wierszy wyrażenia lambda, takie jak użyte w listingu 22.5, nie pomagają w zwiększeniu wydajności programowania. Warto też pamiętać, że obiektów funkcji można ponownie wielokrotnie używać.

Pytanie: W jaki sposób w wyrażeniu lambda przekazywane są parametry stanu: przez wartość czy przez referencję?

Odpowiedź: Kiedy wyrażenie lambda zostało utworzone przy użyciu listy przechwytywania:

```
[Zmienna1, Zmienna2, ... N](Typ& Parametr1, ... ) { ...wyrażenie ;}
```

wtedy parametry stanu `Var1` i `Var2` są kopiowane, a nie dostarczane przez referencję. Jeżeli chcesz je przekazać przez referencję, musisz użyć poniższej składni:

```
[&Zmienna1, &Zmienna2, ... &N](Typ& Parametr1, ... ) { ...wyrażenie ;}
```

W takim przypadku konieczne jest zachowanie ostrożności, ponieważ wszelkie modyfikacje zmiennych stanu będą widoczne także poza samym wyrażeniem lambda.

Pytanie: Czy w funkcji znajdującej się w wyrażeniu lambda mogę używać zmiennych lokalnych?

Odpowiedź: Zmienne lokalne można przekazywać przy użyciu listy przechwytywania:

```
[Zmienna1, Zmienna2, ... N](Typ& Parametr1, ... ) { ...wyrażenie ;}
```

Jeżeli chcesz przechwycić wszystkie zmienne, wtedy należy użyć składni:

```
[=](Typ& Parametr1, ... ) { ...wyrażenie ;}
```

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Po czym kompilator rozpoznaje początek wyrażenia lambda?
2. W jaki sposób wyrażeniu lambda można przekazać zmienne stanu?
3. Jeśli zachodzi konieczność zdefiniowania wartości zwrotnej w wyrażeniu lambda, w jaki sposób możesz to zrobić?

Ćwiczenia

1. Utwórz predykat dwuargumentowy dla wyrażenia lambda, który pomoże w przeprowadzeniu sortowania w kolejności malejącej.
2. Utwórz funkcję lambda, która po użyciu w algorytmie `for_each` umieści w kontenerze, takim jak `vector`, wartość podaną przez użytkownika.

Lekcja 23

Algorytmy STL

Ważną częścią standardowej biblioteki wzorców (STL) jest zbiór ogólnych funkcji dostarczanych przy użyciu nagłówka <algorithm>, które pozwalają na manipulację zawartością kontenera oraz pracę z nią.

Z tej lekcji dowiesz się:

- **jak używać różnych algorytmów STL w celu zmniejszenia ilości tworzonego kodu,**
- **jak ogólne funkcje mogą pomóc w operacjach m.in. zliczania, znajdowania, wyszukiwania i usuwania elementów w kontenerach STL.**

Co to są algorytmy STL?

Wyszukiwanie, usuwanie i zliczanie to tylko kilka ogólnych zadań algorytmów wykorzystywanych w szerokiej gamie programów. Biblioteka STL rozwiązuje te i wiele innych problemów za pomocą wzorców ogólnych funkcji, które działają na kontenerach i używają iteratorów. W celu zastosowania algorytmów STL najpierw należy dołączyć nagłówek `<algorithm>`.

Uwaga

> Choć większość algorytmów działa na kontenerach za pomocą iteratorów, nie wszystkie algorytmy koniecznie muszą działać z kontenerami. Tak więc nie wszystkie algorytmy wymagają iteratorów. Przykładowo algorytm `swap()` po prostu akceptuje parę wartości i zamienia je miejscami. Algorytmy `min()` i `max()` również działają bezpośrednio na wartościach.

Klasyfikacja algorytmów STL

Algorytmy STL można zaklasyfikować do jednego dwóch rodzajów: mogą być niezmienne i zmienne.

Algorytmy niezmienne

Algorytmy, które nie zmieniają ani kolejności, ani zawartości kontenera, nazywane są algorytmami niezmiennymi. Niektóre z ważniejszych algorytmów niezmiennych zostały wymienione w tabeli 23.1.

Tabela 23.1. Algorytmy niezmienne

Algorytm	Opis
Algorytmy zliczające	
`count`	Wyszukuje w zakresie wszystkie elementy, których wartość odpowiada podanej wartości.
`count_if`	Wyszukuje w zakresie wszystkie elementy, których wartość spełnia podany warunek.
`search`	Wyszukuje pierwsze wystąpienie podanej sekwencji w zakresie docelowym. Wyszukiwanie odbywa się albo na podstawie równości elementu (tzn. używany jest operator ==), albo za pomocą wskazanego predykatu dwuargumentowego.

Tabela 23.1. Algorytmy niezmienne (cd.)

Algorytm	Opis
Algorytmy wyszukujące	
`search_n`	Wyszukuje w zakresie docelowym pierwsze wystąpienie *n* elementów podanej wartości lub tych, które spełniają podany predykat.
`find`	Wyszukuje w zakresie pierwszy element, który odpowiada podanej wartości.
`find_if`	Wyszukuje w zakresie pierwszy element, który spełnia podany warunek.
`find_end`	Wyszukuje ostatnie wystąpienie określonego podzakresu w podanym zakresie.
`find_first_of`	Wyszukuje w zakresie docelowym pierwsze wystąpienie dowolnego elementu zakresu źródłowego. Ewentualnie, w wersji przeciążonej powoduje wyszukanie pierwszego wystąpienia elementu, który spełnia podane kryteria wyszukiwania.
`adjacent_find`	Wyszukuje w kolekcji dwa elementy, które albo są równe, albo spełniają podany warunek.
Algorytmy porównania	
`equal`	Porównuje dwa elementy, sprawdzając, czy są równe. Ewentualnie, w celu sprawdzenia równości elementów używa podanego predykatu dwuargumentowego.
`mismatch`	Znajduje położenie pierwszej różnicy w dwóch zakresach elementów, używa do tego podanego predykatu dwuargumentowego.
`lexicographical_compare`	Porównuje elementy w dwóch sekwencjach w celu ustalenia, który z nich jest mniejszy.

Algorytmy zmienne

Algorytmy zmienne modyfikują zawartość bądź kolejność sekwencji, względem której przeprowadzają operacje. Niektóre z najbardziej użytecznych algorytmów zmiennych dostarczanych przez bibliotekę STL zostały wymienione w tabeli 23.2.

Tabela 23.2. Algorytmy zmienne

Algorytm	Opis
Algorytmy inicjalizacyjne	
`fill`	Przypisuje określoną wartość każdemu elementowi w podanym zakresie.
`fill_n`	Przypisuje określoną wartość pierwszym *n* elementom w podanym zakresie.
`generate`	Wartość zwracaną przez określony obiekt funkcji przypisuje każdemu elementowi w podanym zakresie.
`generate_n`	Wartość wygenerowana przez funkcję zostaje przypisana określonej liczbie wartości w podanym zakresie.
Algorytmy modyfikujące	
`for_each`	Przeprowadza operacje na każdym elemencie podanego zakresu. Kiedy określony argument modyfikuje zakres, algorytm `for_each` staje się algorytmem zmiennym.
`transform`	Powoduje zastosowanie wskazanej funkcji jednoargumentowej na każdym elemencie w podanym zakresie.
Algorytmy kopiujące	
`copy`	Kopiuje jeden zakres do innego.
`copy_backward`	Kopiuje jeden zakres do innego, elementy w zakresie docelowym są umieszczane w odwrotnej kolejności.
Algorytmy usuwające	
`remove`	Usuwa z podanego zakresu element o określonej wartości.
`remove_if`	Usuwa z podanego zakresu element, który spełnia określony predykat jednoargumentowy.
`remove_copy`	Kopiuje do zakresu docelowego wszystkie elementy zakresu źródłowego, z wyjątkiem elementów o określonej wartości.
`remove_copy_if`	Kopiuje do zakresu docelowego wszystkie elementy zakresu źródłowego, z wyjątkiem elementów, które spełniają określony predykat jednoargumentowy.
`unique`	Porównuje sąsiadujące elementy zakresu i usuwa duplikaty. Przeciążona wersja tego algorytmu działa przy użyciu predykatu dwuargumentowego.

Tabela 23.2. Algorytmy zmienne (cd.)

Algorytm	Opis
`unique_copy`	Kopiuje do podanego zakresu docelowego wszystkie elementy zakresu źródłowego poza sąsiadującymi duplikatami.
Algorytmy zastępujące	
`replace`	Każdy element w podanym zakresie dopasowany do określonej wartości zostanie zastąpiony wartością zastępującą.
`replace_if`	Każdy element w podanym zakresie dopasowany do wyniku działania predykatu jednoargumentowego zostanie zastąpiony wartością zastępującą.
Algorytmy sortujące	
`sort`	Sortuje elementy zakresu, używając do tego podanych kryteriów sortowania: predykatu dwuargumentowego, któremu trzeba podać tzw. ścisłe uporządkowanie słabe. Algorytm `sort` może również zmieniać względne położenie odpowiadających elementów.
`stable_sort`	Działanie tego algorytmu jest podobne do `sort`, ale powoduje zachowanie kolejności.
`partial_sort`	Sortuje określoną liczbę elementów w zakresie.
`partial_sort_copy`	Kopiuje elementy z określonego zakresu źródłowego do zakresu docelowego, który przechowuje je w kolejności sortowania.
Algorytmy partycjonujące	
`partition`	Mając podany zakres, algorytm powoduje podzielenie elementów na dwie części. W pierwszej znajdują się te, które spełniają predykat jednoargumentowy. Pozostałe elementy znajdują się w drugiej części. Względna kolejność elementów może nie zostać zachowana w zbiorze.
`stable_partition`	Zakres źródłowy zostaje podzielony na dwie części, podobnie jak w algorytmie `partition`, ale zostaje zachowana względna kolejność elementów.
Algorytmy, które działają jak posortowane kontenery	
`binary_search`	Ten algorytm jest używany w celu sprawdzenia, czy element istnieje w posortowanej kolekcji.

Tabela 23.2. Algorytmy zmienne (cd.)

Algorytm	Opis
`lower_bound`	Zwraca iterator wskazujący pierwsze położenie, w którym element może zostać ewentualnie umieszczony w posortowanej kolekcji. Ustalenie tego położenia następuje na podstawie wartości danego elementu bądź dostarczonego predykatu dwuelementowego.
`upper_bound`	Zwraca iterator wskazujący ostatnie położenie, w którym element może zostać ewentualnie umieszczony w posortowanej kolekcji. Ustalenie tego położenia następuje na podstawie wartości danego elementu bądź dostarczonego predykatu dwuelementowego.

Używanie algorytmów STL

Algorytmy wymienione w tabelach 23.1 i 23.2 najłatwiej poznać, używając ich bezpośrednio w przykładach. Zapoznaj się więc z przedstawionymi w poniższych fragmentach kodu szczegółowymi informacjami dotyczącymi używania wymienionych algorytmów, a następnie zacznij je stosować we własnych programach.

Znajdowanie elementów o podanej wartości lub warunku

Gdy mamy kontener, taki jak `vector`, algorytmy STL `find()` oraz `find_if()` pomagają w znajdowaniu elementów (odpowiednio) o dopasowanej wartości lub spełniających określone warunki. Użycie algorytmu `find()` odbywa się w następujący sposób:

```
auto iElementFound = find ( vecIntegers.cbegin () // Początek zakresu.
   , vecIntegers.cend ()   // Koniec zakresu.
   , NumToFind );          // Szukany element.
// Sprawdzenie, czy operacja zakończyła się powodzeniem.
if ( iElementFound != vecIntegers.cend ())
   cout << "Wynik: wartość została znaleziona!" << endl;
```

Algorytm `find_if()` działa podobnie i wymaga podania predykatu jednoargumentowego (funkcji jednoargumentowej, która zwraca wartość `true` lub `false`) jako trzeciego parametru.

```
auto iEvenNumber = find_if ( vecIntegers.cbegin () // Początek zakresu.
   , vecIntegers.cend ()   // Koniec zakresu.
   , [](int element) { return (element % 2) == 0; } );
if (iEvenNumber != vecIntegers.cend ())
   cout << "Wynik: wartość została znaleziona!" << endl;
```

Obie funkcje zwracają iterator, który trzeba porównać z wartością zwróconą przez metody `end()` lub `cend()` kontenera, aby w ten sposób potwierdzić, że operacja wyszukiwania zakończyła się powodzeniem. Jeśli sprawdzenie zakończy się powodzeniem, wtedy można używać iteratora. W listingu 23.1 w przykładowym fragmencie kodu pokazano praktyczne użycie funkcji `find()` w celu znalezienia wartości w obiekcie `vector` oraz funkcji `find_if()` do odszukania pierwszej parzystej wartości.

Listing 23.1. Użycie funkcji find() do odszukania liczby całkowitej w wektorze, find_if() do odszukania pierwszej liczby parzystej przy użyciu predykatu jednoargumentowego w wyrażeniu lambda

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:     vector <int> vecIntegers;
 8:
 9:     // Wstawienie przykładowych wartości, od –9 do 9.
10:     for (int SampleValue = -9; SampleValue < 10; ++ SampleValue)
11:         vecIntegers.push_back (SampleValue);
12:
13:    cout << "Podaj szukaną liczbę w zbiorze: ";
14:    int NumToFind = 0;
15:    cin >> NumToFind;
16:
17:     auto iElementFound = find ( vecIntegers.cbegin ()     // Początek zakresu.
18:                                 , vecIntegers.cend ()    // Koniec zakresu.
19:                                , NumToFind );            // Szukany element.
20:
21:     // Sprawdzenie, czy wyszukanie zakończyło się powodzeniem.
22:     if ( iElementFound != vecIntegers.cend ())
23:         cout << "Wynik: wartość " << *iElementFound << " została
           ↪znaleziona!" << endl;
24:     else
25:         cout << "Wynik: żaden element nie zawiera wartości " <<
           ↪NumToFind << endl;
26:
```

```
27:     cout << "Znajdowanie pierwszej liczby parzystej za pomocą algorytmu
        ↪find_if: " << endl;
28:
29:     auto iEvenNumber = find_if ( vecIntegers.cbegin () // Początek zakresu.
30:                           , vecIntegers.cend ()        // Koniec zakresu.
31:                           , [](int element) { return (element % 2) ==
                              ↪0; });
32:
33:     if (iEvenNumber != vecIntegers.cend ())
34:     {
35:         cout << "Liczba '" << *iEvenNumber << "' została znaleziona
            ↪w położeniu [";
36:         cout << distance (vecIntegers.cbegin (), iEvenNumber);
37:         cout << "]" << endl;
38:        }
39:
40:     return 0;
41: }
```

Wynik ▼

```
Podaj szukaną liczbę w zbiorze: 7
Wynik: wartość 7 została znaleziona!
Znajdowanie pierwszej liczby parzystej za pomocą algorytmu find_if:
Liczba '-8' została znaleziona w położeniu [1]
```

Kolejne uruchomienie programu

```
Podaj szukaną liczbę w zbiorze: 2011
Wynik: żaden element nie zawiera wartości 2011
Znajdowanie pierwszej liczby parzystej za pomocą algorytmu find_if:
Liczba '-8' została znaleziona w położeniu [1]
```

Analiza ▼

Działanie funkcji `main()` rozpoczyna się od utworzenia obiektu `vector` zawierającego liczby całkowite o wartościach od –9 do 9. W wierszach od 17. do 19. do wyszukania liczby podanej przez użytkownika używana jest metoda `find()`. Z kolei metoda `find_if()` pozwala na znalezienie pierwszej liczby parzystej we wskazanym zakresie (patrz wiersze od 29. do 31.). Wiersz 31. to predykat jednoargumentowy w postaci wyrażenia lambda. Wspomniane wyrażenie lambda zwraca wartość `true`, gdy sprawdzany element dzieli się przez dwa bez reszty.

Ostrzeżenie

Zwróć uwagę, że kod przedstawiony w listingu 23.1 zawsze sprawdza iterator zwracany przez metody `find()` i `find_if()`, porównując jego wartość z wartością zwrotną metody `cend()`. Tej operacji sprawdzenia nigdy nie należy pomijać, ponieważ pozwala przekonać się, czy operacja wyszukania została zakończona powodzeniem, bo przecież nie zawsze tak musi być.

Wskazówka

W listingu 17.5 w lekcji 17., zatytułowanej „Dynamiczne klasy tablic w STL", również zaprezentowano użycie `std::distance` (patrz wiersz 21.) w celu sprawdzenia pozycji znalezionego elementu względem początku obiektu `vector`.

Zliczanie elementów o podanej wartości lub warunku

Algorytmy `std::cout()` i `count_if()` to algorytmy pomagające w zliczaniu elementów we wskazanym zakresie. Algorytm `std::find()` pomaga w zliczeniu elementów o dopasowanej wartości (sprawdzonej przy użyciu operatora równości ==):

```
size_t nNumZeroes = count (vecIntegers.begin (),vecIntegers.end (),0);
cout << "Liczba wystąpień '0': " << nNumZeroes << endl << endl;
```

`std::count_if()` pomaga w zliczeniu liczby elementów spełniających predykat jednoargumentowy dostarczony jako parametr (może to być obiekt funkcji lub wyrażenie lambda):

```
// Predykat jednoargumentowy.
template <typename elementType>
bool IsEven (const elementType& number)
{
return ((number % 2) == 0); // Jeśli liczba jest parzysta, wartością zwrotną będzie true.
}
...
// Użycie algorytmu count_if wraz z predykatem jednoargumentowym IsEven:
size_t nNumEvenElements = count_if (vecIntegers.begin (),
   vecIntegers.end (), IsEven <int> );
cout << "Liczba elementów parzystych: " << nNumEvenElements << endl;
```

W kodzie przedstawionym w listingu 23.2 zademonstrowano użycie omówionych funkcji.

Listing 23.2. Użycie std::count() w celu ustalenia liczby elementów o określonej wartości oraz count_if() do ustalenia liczby elementów spełniających wskazany warunek

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3:
 4: // Predykat jednoargumentowy dla funkcji *_if.
 5: template <typename elementType>
 6: bool IsEven (const elementType& number)
 7: {
 8:    return ((number % 2) == 0); // Jeśli liczba jest parzysta, wartością zwrotną
       ↪będzie true.
 9: }
10:
11: int main ()
12: {
13:    using namespace std;
14:    vector <int> vecIntegers;
15:
16:    cout << "Wypełnienie obiektu vector<int> wartościami od -9 do 9" <<
       ↪endl;
17:    for (int nNum = -9; nNum < 10; ++ nNum)
18:       vecIntegers.push_back (nNum);
19:
20:    // Użycie algorytmu count do ustalenia liczby wystąpień '0' w obiekcie vector.
21:    size_t nNumZeroes = count (vecIntegers.begin (),vecIntegers.end (),0);
22:    cout << "Liczba wystąpień '0': " << nNumZeroes << endl << endl;
23:
24:    // Użycie algorytmu count_if wraz z predykatem jednoargumentowym IsEven:
25:    size_t nNumEvenElements = count_if (vecIntegers.begin (),
26:                              vecIntegers.end (), IsEven <int> );
27:
28:    cout << "Liczba elementów parzystych: " << nNumEvenElements << endl;
29:    cout << "Liczba elementów nieparzystych: ";
30:    cout << vecIntegers.size () - nNumEvenElements << endl;
31:
32:    return 0;
33: }
```

Wynik ▼

```
Wypełnienie obiektu vector<int> wartościami od -9 do 9
Liczba wystąpień '0': 1

Liczba elementów parzystych: 9
Liczba elementów nieparzystych: 10
```

Analiza ▼

W wierszu 21. użyto algorytmu `count()` do ustalenia liczby wystąpień zera w obiekcie `vector<int>`. Podobnie w wierszu 25. użyto algorytmu `count_if()` do określenia liczby elementów parzystych w obiekcie `vector`. Zwróć uwagę na trzeci parametr będący predykatem jednoargumentowym `IsEven()`, który został zdefiniowany w wierszach od 6. do 9. Liczba elementów nieparzystych w obiekcie `vector` jest obliczana jako różnica wartości zwrotnej algorytmu `count_if()` i wartości zwróconej przez `size()`, która podaje całkowitą liczbę elementów znajdujących się w wektorze.

Uwaga

W listingu 23.2 użyto funkcji predykatu `IsEven()` w algorytmie `count_if()`, podczas gdy w listingu 23.1 zastosowana była funkcja lambda wykonująca w algorytmie `find_if()` zadanie funkcji `IsEven()`.

Wersja oparta na wyrażeniu lambda jest krótsza, ale należy pamiętać, że po połączeniu obu wymienionych fragmentów kodu funkcji `IsEven()` będzie można użyć w algorytmach `find_if()` i `count_if()`, a tym samym zwiększyć poziom ponownego wykorzystania tego samego kodu.

Wyszukiwanie elementu lub zakresu w kolekcji

W listingu 23.1 pokazano, w jaki sposób można wyszukać element w kontenerze. Jednak czasami trzeba wyszukać zakres wartości. W takich sytuacjach, zamiast polegać na algorytmie wyszukiwania działającym na bazie elementu, lepiej zastosować funkcję `search()` lub `search_n()`, które zostały zaprojektowane do pracy z zakresami:

```
auto iRange = search ( vecIntegers.begin () // Początek zakresu.
   , vecIntegers.end ()      // Koniec przeszukiwanego zakresu.
   , listIntegers.begin ()   // Początek szukanego zakresu.
   , listIntegers.end () ); // Koniec szukanego zakresu.
```

Algorytm `search_n()` może być użyty do sprawdzenia, czy w kontenerze znajduje się *n* kolejnych wystąpień wskazanej wartości:

```
auto iPartialRange = search_n ( vecIntegers.begin () // Początek zakresu.
   , vecIntegers.end ()    // Koniec zakresu.
   , 3        // Liczba szukanych elementów.
   , 9 );     // Szukany element.
```

Obie funkcje zwracają iterator do pierwszego wystąpienia znalezionego wzorca. Przed użyciem wspomnianego iteratora należy go sprawdzić względem

wyniku wywołania metody `end()`. Przykład użycia algorytmów `search()` i `search_n()` pokazano w listingu 23.3.

Listing 23.3. Wyszukanie zakresu w kolekcji za pomocą funkcji search() i search_n()

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <list>
 3: #include <iostream>
 4: using namespace std;
 5:
 6: template <typename T>
 7: void DisplayContents (const T& Input)
 8: {
 9:    for(auto iElement = Input.cbegin() // auto i cbegin(): C++11.
10:       ; iElement != Input.cend()
11:       ; ++ iElement )
12:       cout << *iElement << ' ';
13:
14:    cout << endl;
15: }
16:
17: int main ()
18: {
19:     // Przykładowy kontener — obiekt vector przechowujący liczby całkowite od –9 do 9.
20:     vector <int> vecIntegers;
21:     for (int nNum = -9; nNum < 10; ++ nNum)
22:         vecIntegers.push_back (nNum);
23:
24:     // Wstawienie przykładowych wartości do obiektu vector.
25:     vecIntegers.push_back (9);
26:     vecIntegers.push_back (9);
27:
28:     // Inny przykładowy kontener — obiekt list przechowujący liczby całkowite.
29:     list <int> listIntegers;
30:     for (int nNum = -4; nNum < 5; ++ nNum)
31:         listIntegers.push_back (nNum);
32:
33:     cout << "Zawartość przykładowego obiektu vector to: " << endl;
34:     DisplayContents(vecIntegers);
35:
36:     cout << endl << "Zawartość przykładowego obiektu list to: " << endl;
37:     DisplayContents(listIntegers);
38:
39:     cout << "Wyszukanie zawartości obiektu 'list' w obiekcie 'vector'
       ↪za pomocą algorytmu 'search': " << endl;
40:     auto iRange = search ( vecIntegers.begin () // Początek zakresu.
```

```
41:                     , vecIntegers.end ()       // Koniec przeszukiwanego zakresu.
42:                     , listIntegers.begin ()   // Początek szukanego zakresu.
43:                     , listIntegers.end () ); // Koniec szukanego zakresu.
44:
45:     // Sprawdzenie, czy znaleziono dopasowanie.
46:     if (iRange != vecIntegers.end ())
47:     {
48:         cout << "Sekwencja obiektu list została dopasowana w obiekcie
        ↪vector w położeniu: ";
49:         cout << distance (vecIntegers.begin (), iRange) << endl;
50:     }
51:
52:     cout << "Wyszukiwanie elementów {9, 9, 9} w obiekcie vector
       ↪za pomocą algorytmu 'search_n': " << endl;
53:     auto iPartialRange = search_n ( vecIntegers.begin () // Początek zakresu.
54:                             , vecIntegers.end ()    // Koniec zakresu.
55:                             , 3         // Liczba szukanych elementów.
56:                             , 9 );      // Element do wyszukania.
57:
58:     if (iPartialRange != vecIntegers.end ())
59:     {
60:         cout << "Sekwencja {9, 9, 9} została dopasowana w obiekcie
           ↪vector w położeniu: ";
61:         cout << distance (vecIntegers.begin (), iPartialRange) << endl;
62:     }
63:
64:     return 0;
65: }
```

Wynik ▼

```
Zawartość przykładowego obiektu vector to:
-9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 9 9
Zawartość przykładowego obiektu list to:
-4 -3 -2 -1 0 1 2 3 4

Wyszukanie zawartości obiektu 'list' w obiekcie 'vector' za pomocą
↪algorytmu 'search':
Sekwencja obiektu list została dopasowana w obiekcie vector w położeniu: 5

Wyszukiwanie elementów {9, 9, 9} w obiekcie vector za pomocą algorytmu
↪'search_n':
Sekwencja {9, 9, 9} została dopasowana w obiekcie vector w położeniu: 18
```

Analiza ▼

Program rozpoczyna się od zdefiniowania dwóch przykładowych kontenerów (obiekty `vector` i `list`), które zostają zainicjalizowane wraz z wartościami w postaci liczb całkowitych. W wierszu 40. funkcja `search()` została użyta w celu wyszukania zawartości obiektu `list` w obiekcie `vector`. Parametry dostarczone funkcji `search()` to początek i koniec przeszukiwanego zakresu oraz początek i koniec szukanego zakresu. Ponieważ chcemy przeszukać całą zawartość obiektu `vector` pod kątem zawartości obiektu `list`, dostarczone zostały wartości zwrócone przez metody składowe `begin()` i `end()` odpowiednich obiektów. W tym programie pokazano w praktyce, jak doskonale można połączyć algorytmy z kontenerami, używając do tego celu iteratorów. Fizyczna charakterystyka kontenerów dostarczających iteratory nie ma dla algorytmów żadnego istotnego znaczenia. Funkcja `search_n()` użyta w wierszu 53. wyszukuje pierwsze wystąpienie serii {`9, 9, 9`} w obiekcie `vector`.

Inicjalizacja w kontenerze elementów wraz z określonymi wartościami

Funkcje `fill()` i `fill_n()` to algorytmy STL pomagające w ustawieniu określonej wartości zawartości kontenera. Funkcja `fill()` jest używana do nadpisania elementów w zakresie poprzez podanie zakresu oraz wartości przeznaczonej do wstawienia:

```
vector <int> vecIntegers (3);
// Wszystkie elementy w kontenerze otrzymają wartość 9.
fill (vecIntegers.begin (), vecIntegers.end (), 9);
```

Natomiast funkcja `fill_n()` wymaga podania pozycji początkowej, liczby elementów `n` oraz wartości użytej do wypełnienia kontenera.

```
fill_n (vecIntegers.begin () + 3, /* liczba elementów */ 3, /* wstawiana
➥wartość */ -9);
```

Przykład użycia wymienionych funkcji do łatwej inicjalizacji elementów w obiekcie `vector<int>` pokazano w listingu 23.4.

Listing 23.4. Użycie funkcji fill() i fill_n() w celu ustawienia wartości początkowych w kontenerze

```
 0: #include <algorithm>
 1: #include <vector>
```

```
 2: #include <iostream>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:
 8:     // Inicjalizacja przykładowego obiektu vector wraz z trzema elementami.
 9:     vector <int> vecIntegers (3);
10:
11:     // Wypełnienie wszystkich elementów kontenera wartością 9.
12:     fill (vecIntegers.begin (), vecIntegers.end (), 9);
13:
14:     // Zwiększenie pojemności obiektu vector tak, aby przechowywał 6 elementów.
15:     vecIntegers.resize (6);
16:
17:     // Wypełnienie wartością –9 trzech elementów, począwszy od położenia 3.
18:     fill_n (vecIntegers.begin () + 3, 3, -9);
19:
20:     cout << "Zawartość obiektu vector jest następująca: " << endl;
21:     for (size_t nIndex = 0; nIndex < vecIntegers.size (); ++ nIndex)
22:     {
23:         cout << "Element [" << nIndex << "] = ";
24:         cout << vecIntegers [nIndex] << endl;
25:     }
26:
27:     return 0;
28: }
```

Wynik ▼

```
Zawartość obiektu vector jest następująca:
Element [0] = 9
Element [1] = 9
Element [2] = 9
Element [3] = -9
Element [4] = -9
Element [5] = -9
```

Analiza ▼

Funkcje `fill()` i `fill_n()` zostały wykorzystane w listingu 23.4 w celu zainicjalizowania zawartości kontenera dwoma zbiorami wartości, co pokazano w wierszach 12. i 18. Warto zwrócić uwagę na użycie funkcji `resize()` przed wypełnieniem zakresu wartościami, zwłaszcza podczas tworzenia elementów, które następnie zostaną wypełnione wartościami.

Użycie std::generate() do inicjalizacji elementów wartościami wygenerowanymi w trakcie działania programu

Podobnie jak funkcje z rodziny `fill()` wypełniają kolekcję wartością zdefiniowaną przez programistę, algorytmy STL, takie jak `generate()` i `generate_n()`, mogą być użyte do zainicjalizowania kolekcji wraz z wartościami zwróconymi przez funkcję jednoargumentową.

Algorytm `generate()` można wykorzystać do wypełnienia zakresu wartościami wygenerowanymi przez funkcję `generate()`:

```
generate ( vecIntegers.begin (), vecIntegers.end ()      // Zakres.
   , rand );     // Wywoływana funkcja generatora.
```

Algorytm `generate_n()` działa podobnie do `generate()`, ale zamiast granicy zakresu podajesz liczbę elementów do wypełnienia:

```
generate_n (listIntegers.begin (), 5, rand);
```

Dwa wymienione algorytmy można wykorzystać do zainicjalizowania kolekcji wraz z zawartością, którą może być np. plik bądź losowe wartości, co przedstawiono w listingu 23.5.

Listing 23.5. Użycie funkcji generate() i generate_n() w celu inicjalizacji kolekcji wraz z losowymi wartościami

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <list>
 3: #include <iostream>
 4:
 5: int main ()
 6: {
 7:     using namespace std;
 8:
 9:     vector <int> vecIntegers (10);
10:     generate ( vecIntegers.begin (), vecIntegers.end ()     // Zakres.
11:               , rand );     // Wywołanie funkcji generatora.
12:
13:     cout << "Wartości elementów obiektu vector o pojemności " <<
       ↪vecIntegers.size();
14:     cout << " przypisane przez algorytm 'generate': " << endl << "{";
15:     for (size_t nCount = 0; nCount < vecIntegers.size (); ++ nCount)
16:         cout << vecIntegers [nCount] << " ";
17:
```

```
18:     cout << "}" << endl << endl;
19:
20:     list <int> listIntegers (10);
21:     generate_n (listIntegers.begin (), 5, rand);
22:
23:     cout << "Wartości elementów obiektu list o pojemności: " <<
        ↪listIntegers.size();
24:     cout << " przypisane przez algorytm 'generate_n': " << endl << "{";
25:     list <int>::const_iterator iElementLocator;
26:     for ( iElementLocator = listIntegers.begin ()
27:         ; iElementLocator != listIntegers.end ()
28:         ; ++ iElementLocator )
29:         cout << *iElementLocator << ' ';
30:
31:     cout << "}" << endl;
32:
33:     return 0;
34: }
```

Wynik ▼

```
Wartości elementów obiektu vector o pojemności 10 przypisane przez algorytm
↪'generate':
{41 18467 6334 26500 19169 15724 11478 29358 26962 24464 }

Wartości elementów obiektu list o pojemności 10 przypisane przez algorytm
↪'generate_n':
{5705 28145 23281 16827 9961 0 0 0 0 0 }
```

Analiza ▼

W listingu 23.5 wykorzystano funkcję `generate()` w celu wypełnienia wszystkich elementów obiektu `vector` losowymi wartościami wygenerowanymi przez funkcję `rand()`. Warto zwrócić uwagę na fakt, że funkcja `generate()` akceptuje dane wejściowe w postaci zakresu, a następnie wywołuje obiekt funkcji `rand()` względem każdego elementu podanego zakresu. Z kolei funkcja `generate_n()` akceptuje jedynie położenie początkowe. Następnie wywołuje podany obiekt funkcji — `rand()` — liczbę razy określoną przez parametr `count` w celu nadpisana zawartości wskazanej liczby elementów. Elementy w kontenerze, które znajdują się poza wskazanym położeniem, pozostają nienaruszone.

Przetwarzanie elementów w zakresie za pomocą for_each

Algorytm `for_each()` powoduje zastosowanie wskazanego obiektu funkcji jednoargumentowej względem każdego elementu w podanym zakresie. Sposób użycia algorytmu `for_each()` jest następujący:

```
unaryFunctionObjectType mReturn = for_each ( początek_zakresu
                                           , koniec_zakresu
                                           , obiektFunkcjiJednoargumentowej
);
```

Obiekt funkcji jednoargumentowej może być wyrażeniem lambda akceptującym pojedynczy parametr.

Wartość zwracana wskazuje, że algorytm `for_each()` zwraca obiekt funkcji (nazywany także *funktorem*) używany do przetworzenia każdego elementu w podanym zakresie. Konsekwencją tej specyfikacji jest to, że użycie struktury (`struct`) bądź klasy (`class`) jako obiektu funkcji może pomóc w przechowywaniu informacji o stanie, które następnie można wykorzystać po zakończeniu działania przez algorytm `for_each()`. Takie rozwiązanie zademonstrowano w listingu 23.6, w którym obiekt funkcji został użyty do wyświetlenia elementów zakresu oraz do obliczenia liczby wyświetlonych elementów.

Listing 23.6. Użycie funkcji for_each() w celu wyświetlenia zawartości kolekcji

```
 0: #include <algorithm>
 1: #include <iostream>
 2: #include <vector>
 3: #include <string>
 4: using namespace std;
 5:
 6: // Typ obiektu funkcji jednoargumentowej wywoływanej przez algorytm for_each.
 7: template <typename elementType>
 8: struct DisplayElementKeepCount
 9: {
10:     int Count;
11:
12:     // Konstruktor.
13:     DisplayElementKeepCount () : Count (0) {}
14:
15:     void operator () (const elementType& element)
16:     {
17:         ++ Count;
```

```
18:         cout << element << ' ';
19:     }
20: };
21:
22: int main ()
23: {
24:     vector <int> vecIntegers;
25:     for (int nCount = 0; nCount < 10; ++ nCount)
26:         vecIntegers.push_back (nCount);
27:
28:     cout << "Wyświetlenie obiektu vector przechowującego liczby
       ↪całkowite: " << endl;
29:
30:     // Wyświetlenie tablicy liczb całkowitych.
31:     DisplayElementKeepCount<int> Functor =
32:         for_each ( vecIntegers.begin ()    // Początek zakresu.
33:                  , vecIntegers.end ()        // Koniec zakresu.
34:                  , DisplayElementKeepCount<int> () ); // Funktor.
35:     cout << endl;
36:
37:     // Użycie informacji o stanie przechowywanej wartości zwracanej przez algorytm
       ↪for_each!
38:     cout << "W obiekcie vector wyświetlono '" << Functor.Count << "'
       ↪elementów!" << endl;
39:
40:     string Sample ("algorytm for_each i ciągi tekstowe!");
41:     cout << "Przykładowy ciąg tekstowy to: " << Sample << ", długość: " <<
       ↪Sample.length() << endl;
42:
43:     cout << "Ciąg tekstowy wyświetlony za pomocą wyrażenia lambda:" <<
       ↪endl;
44:    int NumChars = 0;
45:    for_each ( Sample.begin()
46:             , Sample.end ()
47:             , [&NumChars](char c) { cout << c << ' '; ++NumChars; } );
48:
49:     cout << endl;
50:     cout << "Wyświetlono '" << NumChars << "' znaków" << endl;
51:
52:     return 0;
53: }
```

Wynik ▼

```
Wyświetlenie obiektu vector przechowującego liczby całkowite:
0 1 2 3 4 5 6 7 8 9
W obiekcie vector wyświetlono '10' elementów!
Przykładowy ciąg tekstowy to: algorytm for_each i ciągi tekstowe!, długość: 35
```

```
Ciąg tekstowy wyświetlony za pomocą wyrażenia lambda:
a l g o r y t m  f o r _ e a c h  i  c i ą g i  t e k s t o w e !
Wyświetlono '35' znaków
```

Analiza ▼

Powyższy fragment kodu pokazuje nie tylko użyteczność algorytmu `for_each`, ale także jego charakterystykę polegającą na zwróceniu obiektu `Result`, który został zaprojektowany do przechowywania informacji, ile razy wywoływany był ten obiekt. W programie znajdują się dwa przykładowe zakresy: pierwszy to obiekt `vector` (`vecIntegers`) przechowujący liczby całkowite, natomiast drugi to obiekt `std::string` (`Sample`). Program wywołuje funkcję `for_each()` względem wymienionych zakresów (w wierszach, odpowiednio, 32. i 45.). Pierwsze wywołanie wykorzystuje funktor `DisplayElementKeepCount` jako predykat jednoargumentowy, natomiast drugie — wyrażenie lambda. Funkcja `for_each()` wywołuje względem każdego elementu w podanym zakresie funkcję `operator()`, która z kolei wyświetla element na ekranie oraz inkrementuje wartość wewnętrznego licznika. Obiekt funkcji jest zwracany po zakończeniu pracy przez `for_each()`, natomiast funkcja składowa `Count()` informuje, ile razy wykorzystywany był obiekt.

Możliwość przechowywania informacji (bądź stanu) w obiekcie zwracanym przez algorytm może być bardzo użyteczna podczas rozwiązywania rzeczywistych zagadnień programistycznych. Algorytm `for_each()` w wierszu 45. wykonuje dokładnie takie samo zadanie jak jego odpowiednik w wierszu 32. dla `std::string` przy użyciu wyrażenia lambda zamiast obiektu funkcji.

Przeprowadzanie transformacji zakresu za pomocą std::transform

Funkcje `for_each()` i `std::transform` są do siebie bardzo podobne pod tym względem, że obie wywołują obiekt funkcji dla każdego elementu w zakresie źródłowym. Jednak `std::transform` ma dwie wersje: pierwsza akceptuje funkcję jednoargumentową i bardzo często jest używana do konwersji znaków w ciągu tekstowym na wielkie lub małe przy użyciu funkcji `toupper()` lub `tolower()`:

```
string Sample ("TO jest PRZYkładowy ciąg tekstowy!");
transform ( Sample.begin ()           // Początek zakresu źródłowego.
   , Sample.end ()            // Koniec zakresu źródłowego.
   , strLowerCaseCopy.begin () // Początek zakresu docelowego.
   , tolower );          // Funkcja jednoargumentowa.
```

Natomiast druga wersja akceptuje funkcję dwuargumentową i pozwala algorytmowi `transform()` na przetworzenie również pary elementów pobranych z dwóch różnych zakresów:

```
// Dodanie elementów z dwóch zakresów i umieszczenie wyniku w trzecim.
transform ( vecIntegers1.begin ()    // Początek pierwszego zakresu źródłowego.
   , vecIntegers1.end ()      // Koniec pierwszego zakresu źródłowego.
   , vecIntegers2.begin ()    // Początek drugiego zakresu źródłowego.
   , dqResultAddition.begin () // Umieszczenie wyniku w obiekcie deque.
   , plus <int> () );         // Dwuargumentowa funkcja plus.
```

W przeciwieństwie do funkcji `for_each()` działającej tylko z pojedynczym zakresem, obie wersje funkcji `transform()` zawsze przypisują podanemu zakresowi docelowemu wynik działania określonej funkcji transformującej. Zastosowanie `std::transform` zostało zademonstrowane w listingu 23.7.

Listing 23.7. Użycie std::transform wraz z funkcjami jednoargumentową i dwuargumentową

```
 0: #include <algorithm>
 1: #include <string>
 2: #include <vector>
 3: #include <deque>
 4: #include <iostream>
 5: #include <functional>
 6:
 7: int main ()
 8: {
 9:     using namespace std;
10:
11:     string Sample ("TO jest TEstowy ciąg tekstowy!");
12:     cout << "Przykładowy ciąg tekstowy to: " << Sample << endl;
13:
14:     string strLowerCaseCopy;
15:     strLowerCaseCopy.resize (Sample.size ());
16:
17:     transform (Sample.begin ()           // Początek zakresu źródłowego.
18:                , Sample.end ()           // Koniec zakresu źródłowego.
19:                , strLowerCaseCopy.begin () // Początek zakresu docelowego.
20:                , tolower );         // Funkcja jednoargumentowa.
21:
22:     cout << "Wynik użycia algorytmu 'transform' wraz z funkcją
        ↪'tolower' względem ciągu tekstowego:" << endl;
23:     cout << "\"" << strLowerCaseCopy << "\"" << endl << endl;
24:
25:     // Dwa przykładowe obiekty vector przechowujące liczby całkowite...
```

```
26:     vector <int> vecIntegers1, vecIntegers2;
27:     for (int nNum = 0; nNum < 10; ++ nNum)
28:     {
29:         vecIntegers1.push_back (nNum);
30:         vecIntegers2.push_back (10 - nNum);
31:     }
32:
33:     // Zakres docelowy przeznaczony do przechowywania wyniku dodawania.
34:     deque <int> dqResultAddition (vecIntegers1.size ());
35:
36:     transform ( vecIntegers1.begin ()  // Początek pierwszego zakresu źródłowego.
37:               , vecIntegers1.end ()    // Koniec pierwszego zakresu źródłowego.
38:               , vecIntegers2.begin ()  // Początek drugiego zakresu źródłowego.
39:               , dqResultAddition.begin () // Koniec drugiego zakresu źródłowego.
40:               , plus <int> () );          // Funkcja dwuargumentowa.
41:
42:     cout << "Wynik użycia algorytmu 'transform' wraz z funkcją
       ↪dwuargumentową 'plus': " << endl;
43:     cout <<endl << "Indeks   Vector1 + Vector2 = Wynik (w Deque)" <<
       ↪endl;
44:     for (size_t nIndex = 0; nIndex < vecIntegers1.size (); ++ nIndex)
45:     {
46:         cout << nIndex << "    \t " << vecIntegers1 [nIndex]  << "\t+   ";
47:         cout << vecIntegers2 [nIndex]  << " \t  =    ";
48:
49:         cout << dqResultAddition [nIndex] << endl;
50:     }
51:
52:     return 0;
53: }
```

Wynik ▼

```
Przykładowy ciąg tekstowy to: TO jest TEstowy ciąg tekstowy!
Wynik użycia algorytmu 'transform' wraz z funkcją 'tolower' względem ciągu
↪tekstowego:
"to jest testowy ciąg tekstowy!"

Wynik użycia algorytmu 'transform' wraz z funkcją dwuargumentową 'plus':

Indeks    Vector1 + Vector2 = Wynik (w Deque)
0         0      +   10    =   10
1         1      +   9     =   10
2         2      +   8     =   10
3         3      +   7     =   10
4         4      +   6     =   10
5         5      +   5     =   10
```

```
6       6     +   4     =   10
7       7     +   3     =   10
8       8     +   2     =   10
9       9     +   1     =   10
```

Analiza ▼

W powyższym programie pokazano zastosowanie obu wersji `std::transform`. Pierwsza działa względem pojedynczego zakresu, używając funkcji jednoargumentowej `tolower()`, jak widać w wierszu 20. Natomiast druga działa względem dwóch zakresów i używa funkcji dwuargumentowej `plus()`, co pokazano w wierszu 40. Pierwsza wersja powoduje zmianę wielkości znaków ciągu tekstowego — zmienia je na małe, znak po znaku. Jeżeli zamiast `tolower()` zostałaby użyta funkcja `toupper()`, konwersja znaków nastąpiłaby w przeciwnym kierunku, tzn. zaszłaby zamiana wszystkich znaków na duże. Pokazana w wierszach od 36. do 40. druga wersja `std::transform` działa na elementach pobranych z dwóch zakresów źródłowych (w tym przypadku to dwa obiekty `vector`). Ta wersja używa predykatu dwuargumentowego w postaci funkcji STL `plus()` (dostarczanej poprzez nagłówek `<functional>`) i dodaje pobrane elementy. `std::transform` pobiera jednorazowo po jednej parze elementów, przekazuje je funkcji dwuargumentowej `plus()`, a obliczony wynik przypisuje elementowi w zakresie docelowym — należy on do `std::deque`. Warto zwrócić uwagę na fakt, że zmiana w kontenerze używanym do przechowywania wyników służy jedynie do celów demonstracyjnych. Wyświetla tylko ilość iteratorów użytych do abstrakcji kontenerów oraz ich implementację z algorytmów STL. Algorytm `transform` działa względem zakresów i naprawdę nie musi znać żadnych szczegółów dotyczących kontenerów implementujących te zakresy. Tak więc, jeżeli zakresy źródłowe będą obiektami `vector`, a zakresy docelowe obiektami `deque`, wszystko nadal będzie działało doskonale — przynajmniej do chwili, kiedy ograniczniki definiujące zakres (dostarczane algorytmowi `transform` jako parametry wejściowe) pozostaną prawidłowe.

Operacje kopiowania i usuwania

Biblioteka STL dostarcza trzy wiodące funkcje kopiowania: `copy()`, `copy_if()` i `copy_backward()`. Funkcja `copy()` może przepisać zawartość zakresu źródłowego do zakresu docelowego w zwykłej kolejności:

```
auto iLastPos = copy ( listIntegers.begin ()  // Początek zakresu źródłowego.
   , listIntegers.end ()      // Koniec zakresu źródłowego.
   , vecIntegers.begin () ); // Początek zakresu docelowego.
```

Funkcja `copy_if()` kopiuje element tylko wtedy, gdy predykat jednoargumentowy dostarczony przez programistę zwraca wartość `true`:

```
// Kopiowanie liczb nieparzystych z obiektu list do obiektu vector.
copy_if ( listIntegers.begin(), listIntegers.end()
   , iLastPos
   , [](int element){return ((element % 2) == 1);});
```

Uwaga

> W standardzie C++11 `copy_if()` jest algorytmem w przestrzeni nazw `std`. Jeżeli używasz starszej wersji kompilatora, niezgodnej ze standardem C++11, możesz mieć problemy z użyciem algorytmu `copy_if()`.

Algorytm `copy_backward()` powoduje przepisanie zawartości w zakresie docelowym; stosuje przy tym odwrotną kolejność:

```
copy_backward ( listIntegers.begin ()
   , listIntegers.end ()
   , vecIntegers.end () );
```

Funkcja `remove()` usuwa z kontenera elementy, których wartość została dopasowana:

```
// Usunięcie wszystkich wystąpień '0' i zmiana wielkości obiektu vector przy użyciu metody
erase().
auto iNewEnd = remove (vecIntegers.begin (), vecIntegers.end (), 0);
vecIntegers.erase (iNewEnd, vecIntegers.end ());
```

Z kolei funkcja `remove_if()` używa predykatu jednoargumentowego i usuwa z kontenera te elementy, dla których zdefiniowany predykat przyjmuje wartość `true`:

```
// Usunięcie wszystkich liczb nieparzystych z obiektu  vector przy użyciu algorytmu remove_if.
iNewEnd = remove_if (vecIntegers.begin (), vecIntegers.end (),
   [](int element) {return ((element % 2) == 1);} ); // Predykat.
vecIntegers.erase (iNewEnd , vecIntegers.end ());  // Zmiana wielkości.
```

Użycie funkcji kopiowania i usuwania elementów przedstawiono w listingu 23.8.

Listing 23.8. Demonstracja użycia funkcji copy(), copy_backward(), remove() i remove_if() w celu skopiowania elementów obiektu list do vector oraz usunięcia zer i liczb parzystych

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <list>
 3: #include <iostream>
 4: using namespace std;
 5:
 6: template <typename T>
 7: void DisplayContents (const T& Input)
 8: {
 9:    for(auto iElement = Input.cbegin() // auto i cbegin(): C++11.
10:       ; iElement != Input.cend() // cend() to nowość w C++11.
11:       ; ++ iElement )
12:       cout << *iElement << ' ';
13:
14:    cout << " | Liczba elementów: " << Input.size() << endl;
15: }
16: int main ()
17: {
18:     list <int> listIntegers;
19:     for (int nCount = 0; nCount < 10; ++ nCount)
20:         listIntegers.push_back (nCount);
21:
22:     cout << "Elementy w obiekcie źródłowym (list) są następujące: " <<
        ↪endl;
23:     DisplayContents(listIntegers);
24:
25:     // Inicjalizacja obiektu vector przeznaczonego do przechowywania dwukrotnie
        ↪większej liczby elementów niż list.
26:     vector <int> vecIntegers (listIntegers.size () * 2);
27:
28:     auto iLastPos = copy ( listIntegers.begin ()  // Początek zakresu
        ↪źródłowego.
29:               , listIntegers.end ()       // Koniec zakresu źródłowego.
30:               , vecIntegers.begin () ); // Początek zakresu docelowego.
31:
32:     // Kopiujemy liczby nieparzyste z obiektu list do obiektu vector.
33:    copy_if ( listIntegers.begin(), listIntegers.end()
34:            , iLastPos
35:            , [](int element){return ((element % 2) == 1);});
36:
37:     cout << "Elementy w obiekcie docelowym (vector) po operacji
        ↪kopiowania są następujące: " << endl;
38:     DisplayContents(vecIntegers);
39:
```

```
40:     // Usunięcie wszystkich wystąpień elementu '0' i zmiana wielkości obiektu vector
        ↪przy użyciu algorytmu erase().
41:     auto iNewEnd = remove (vecIntegers.begin (), vecIntegers.end (), 0);
42:     vecIntegers.erase (iNewEnd, vecIntegers.end ());
43:
44:     // Usunięcie z obiektu vector wszystkich liczb nieparzystych za pomocą funkcji
        ↪remove_if().
45:     iNewEnd = remove_if (vecIntegers.begin (), vecIntegers.end (),
46:               [](int element) {return ((element % 2) == 1); });
                  ↪// Predykat.
46:               [](int element) {return ((element % 2) == 1); // Predykat.
47:
48:     vecIntegers.erase (iNewEnd , vecIntegers.end ()); // Zmiana wielkości.
49:
50:     cout << "Elementy w obiekcie docelowym (vector) po operacji
        ↪usunięcia są następujące: " << endl;
51:     DisplayContents(vecIntegers);
52:
53:     return 0;
54: }
```

Wynik ▼

```
Elementy w obiekcie źrodłowym (list) są następujące:
0 1 2 3 4 5 6 7 8 9 | Liczba elementów: 10

Elementy w obiekcie docelowym (vector) po operacji kopiowania są następujące:
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 | Liczba elementów: 20

Elementy w obiekcie docelowym (vector) po operacji usunięcia są
następujące:
2 4 6 8 | Liczba elementów: 4
```

Analiza ▼

W wierszu 28. można zobaczyć sposób użycia funkcji `copy()` do skopiowania zawartości obiektu `list` do obiektu `vector`. Funkcja `copy_if()` została użyta w wierszu 33. i spowodowała skopiowanie wszystkich liczb nieparzystych z zakresu źródłowego `listIntegers` do zakresu docelowego `vecIntegers` od pozycji wskazywanej przez iterator `iLasPos` zwrócony przez funkcję `copy()`. W wierszu 41. funkcja `remove()` została użyta w celu usunięcia wszystkich wystąpień elementu 0 w kolekcji `vecIntegers`. Z kolei funkcja `remove_if()` zastosowana w wierszu 45. usunęła wszystkie liczby nieparzyste.

Ostrzeżenie

W listingu 23.8 pokazano, że funkcje `copy()` i `remove()` zwracają pozycje końcowe iteratorów. Jednak wielkość kontenera `vecIntegers` jeszcze nie została zmieniona. Elementy zostały usunięte przez algorytm, pozostałe elementy przesunięte do przodu, ale wielkość kontenera pozostała bez zmian, więc wartości znajdują się na końcu wektora. Aby zmienić wielkość kontenera (to jest ważne z powodu niechcianych wartości na jego końcu), konieczne jest użycie iteratora zwróconego przez algorytmy `remove()` lub `remove_if()` w kolejnym wywołaniu `erase()`, tak jak pokazano w wierszach 42. i 48.

Zastępowanie wartości oraz zastępowanie elementu na podstawie danego warunku

Algorytmy STL `replace()` i `replace_if()` oferują możliwość zastępowania elementów kolekcji, które — kolejno — mają określoną wartość lub spełniają podany warunek. Funkcja `replace()` powoduje zastępowanie elementów na podstawie wartości zwracanej przez operator porównania `==`:

```
cout << "Użycie 'std::replace' do zastąpienia wartości 5 wartością 8" << endl;
replace (vecIntegers.begin (), vecIntegers.end (), 5, 8);
```

Z kolei `replace_if()` wymaga zdefiniowanego przez programistę predykatu jednoargumentowego, który zwraca wartość `true` dla każdego elementu przeznaczonego do zastąpienia.

```
cout << "Użycie 'std::replace' do zastąpienia wartości parzystych
➥wartością -1" << endl;
replace_if (vecIntegers.begin (), vecIntegers.end ()
   , [](int element) {return ((element % 2) == 0); }, -1);
```

Użycie wymienionych funkcji zademonstrowano w listingu 23.9.

Listing 23.9. Użycie funkcji replace() i replace_if() do zastępowania wartości w zdefiniowanym zakresie

```
 0: #include <iostream>
 1: #include <algorithm>
 2: #include <vector>
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents (const T& Input)
 7: {
 8:    for(auto iElement = Input.cbegin() // auto i cbegin(): C++11.
 9:       ; iElement != Input.cend() // cend() to nowość w C++11.
10:       ; ++ iElement )
```

```
11:       cout << *iElement << ' ';
12:
13:    cout << " | Liczba elementów: " << Input.size() << endl;
14: }
15: int main ()
16: {
17:     vector <int> vecIntegers (6);
18:
19:     // Wypełnienie pierwszych trzech elementów wartością 8, ostatnich trzech wartością 5.
20:     fill (vecIntegers.begin (), vecIntegers.begin () + 3, 8);
21:     fill_n (vecIntegers.begin () + 3, 3, 5);
22:
23:     // Umieszczenie elementów kontenera w losowej kolejności.
24:     random_shuffle (vecIntegers.begin (), vecIntegers.end ());
25:
26:     cout << "Początkowa zawartość obiektu vector jest następująca: " <<
       ↪endl;
27:     DisplayContents(vecIntegers);
28:
29:     cout << endl << "Użycie funkcji 'std::replace' w celu zastąpienia
       ↪wartości 5 przez wartość 8" << endl;
30:     replace (vecIntegers.begin (), vecIntegers.end (), 5, 8);
31:
32:     cout << "Użycie funkcji 'std::replace_if' w celu zastąpienia liczb
       ↪parzystych przez wartość -1" << endl;
33:     replace_if (vecIntegers.begin (), vecIntegers.end ()
34:        , [](int element) {return ((element % 2) == 0); }, -1);
35:
36:     cout << endl << "Zawartość obiektu vector po operacjach
       ↪zastępowania:" << endl;
37:     DisplayContents(vecIntegers);
38:
39:     return 0;
40: }
```

Wynik ▼

```
Początkowa zawartość obiektu vector jest następująca:
5 8 5 8 8 5 | Liczba elementów: 6

Użycie funkcji 'std::replace' w celu zastąpienia wartości 5 przez wartość 8
Użycie funkcji 'std::replace_if' w celu zastąpienia liczb parzystych przez
↪wartość -1

Zawartość obiektu vector po operacjach zastępowania:
-1 -1 -1 -1 -1 -1 | Liczba elementów: 6
```

Analiza ▼

Ten przykładowy program powoduje wypełnienie obiektu `vector` (`vecIntegers`) wartościami przykładowymi, a następnie za pomocą algorytmu STL `std::random_shuffle` elementy obiektu są umieszczone w kolejności losowej, co pokazano w wierszu 24. Użycie funkcji `replace()` w celu zastąpienia wartości 5 przez wartość 8 pokazano w wierszu 30. Dlatego też w wierszu 33. funkcja `replace_if()` zastępuje wszystkie liczby parzyste wartością –1. Jak można zobaczyć w danych wyjściowych, kolekcja końcowa zawiera sześć elementów i wszystkie mają taką samą wartość –1.

Sortowanie i przeszukiwanie posortowanej kolekcji oraz usuwanie duplikatów

Sortowanie i przeszukiwanie posortowanego zakresu (dla zachowania wydajności) to wymagania, które bardzo często pojawiają się w rzeczywistych aplikacjach. Nierzadko zdarza się sytuacja, kiedy masz tablicę informacji, które muszą zostać posortowane — np. w celu prezentacji. Do posortowania kontenera można użyć algorytmu STL `sort()`:

```
sort (vecIntegers.begin (), vecIntegers.end ()); // Kolejność rosnąca.
```

Przedstawiona wersja algorytmu `sort()` korzysta z `std:less<>` — predykatu binarnego wykorzystującego operator < zaimplementowany przez typ w wektorze. Istnieje możliwość dostarczenia własnego predykatu w celu zmiany kolejności sortowania przy użyciu przeciążonej wersji:

```
sort (vecIntegers.begin (), vecIntegers.end (),
   [](int lhs, int rsh) {return (lhs > rhs);} ); // Kolejność malejąca.
```

Duplikaty muszą być usunięte przed wyświetleniem zawartości kolekcji. Aby usunąć położone obok powtarzające się wartości, należy użyć algorytmu `unique()`:

```
auto iNewEnd = unique (vecIntegers.begin (), vecIntegers.end ());
vecIntegers.erase (iNewEnd, vecIntegers.end ());  // Zmiana wielkości.
```

W celu przeprowadzenia szybkiego wyszukiwania biblioteka STL oferuje algorytm `binary_search()`, który działa efektywnie jedynie w posortowanym kontenerze:

```
bool bElementFound = binary_search (vecIntegers.begin (), vecIntegers.end
↪(), 2011);
```

```
if (bElementFound)
   cout << "Element został znaleziony w wektorze!" << endl;
```

W programie przedstawionym w listingu 23.10 pokazano użycie algorytmów STL: `std::sort` sortującego zakres, `std::binary_search` przeszukującego posortowany zakres oraz `std::unique` usuwającego powtarzające się sąsiednie elementy (czyli takie, które zostały sąsiadami po przeprowadzeniu operacji sortowania).

Listing 23.10. Użycie algorytmów std::sort, binary_search oraz unique

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <string>
 3: #include <iostream>
 4: using namespace std;
 5:
 6: template <typename T>
 7: void DisplayContents (const T& Input)
 8: {
 9:    for(auto iElement = Input.cbegin() // auto i cbegin(): C++11.
10:       ; iElement != Input.cend() // cend() to nowość w C++11.
11:       ; ++ iElement )
12:       cout << *iElement << endl;
13: }
14: int main ()
15: {
16:     vector<string> vecNames;
17:     vecNames.push_back ("Cezary Pazura");
18:     vecNames.push_back ("Marek Kondrat");
19:     vecNames.push_back ("Marek Perepeczko");
20:     vecNames.push_back ("Anna Dereszowska");
21:
22:     // Wstawienie duplikatu do obiektu vector.
23:     vecNames.push_back ("Marek Kondrat");
24:
25:     cout << "Początkowa zawartość obiektu vector jest następująca:" <<
       ↪endl;
26:     DisplayContents(vecNames);
27:
28:     cout << "Posortowany obiekt vector zawiera elementy w następującej
       ↪kolejności:" << endl;
29:     sort (vecNames.begin (), vecNames.end ());
30:     DisplayContents(vecNames);
31:
32:     cout << "Wyszukanie elementu \"Cezary Pazura\" za pomocą algorytmu
       ↪'binary_search':" << endl;
```

```
33:     bool bElementFound = binary_search (vecNames.begin (),
        ↪vecNames.end (),
34:                                         "Cezary Pazura");
35:
36:     if (bElementFound)
37:         cout << "Wynik: element \"Cezary Pazura\" został znaleziony
            ↪w obiekcie vector!" << endl;
38:     else
39:         cout << "Element nie został znaleziony " << endl;
40:
41:     // Usunięcie sąsiadujących ze sobą duplikatów.
42:     auto iNewEnd = unique (vecNames.begin (), vecNames.end ());
43:     vecNames.erase (iNewEnd, vecNames.end ());
44:
45:     cout << "Zawartość obiektu vector po użyciu algorytmu 'unique'
        ↪jest następująca:" << endl;
46:     DisplayContents(vecNames);
47:
48:     return 0;
49: }
```

Wynik ▼

```
Początkowa zawartość obiektu vector jest następująca:
Cezary Pazura
Marek Kondrat
Marek Perepeczko
Anna Dereszowska
Marek Kondrat

Posortowany obiekt vector zawiera elementy w następującej kolejności:
Anna Dereszowska
Cezary Pazura
Marek Kondrat
Marek Kondrat
Marek Perepeczko

Wyszukanie elementu "Cezary Pazura" za pomocą algorytmu 'binary_search':
Wynik: element "Cezary Pazura" został znaleziony w obiekcie vector!

Zawartość obiektu vector po użyciu algorytmu 'unique' jest następująca:
Anna Dereszowska
Cezary Pazura
Marek Kondrat
Marek Perepeczko
```

Analiza ▼

W przedstawionym powyżej programie na początek posortowano przykładowy obiekt `vector` (`vecNames`) w wierszu 29., a dopiero później użyto algorytmu `binary_search` w celu wyszukania elementu „Cezary Pazura” w obiekcie (wiersz 33.). Podobnie w wierszu 42. funkcja `std::unique` służy do usunięcia drugiego wystąpienia sąsiadujących ze sobą duplikatów. Zwróć uwagę, że `unique()`, podobnie jak `remove()`, nie powoduje zmiany wielkości kontenera. Elementy w kontenerze będą przesunięte, ale nie zmniejszy się ich całkowita liczba. Aby zagwarantować, że kontener nie będzie zawierał na końcu niechcianych lub nieznanych wartości, po wywołaniu algorytmu `unique()` zawsze wywołuj `vector::erase()`, używając iteratora zwróconego przez `unique()`, tak jak to przedstawiono w wierszach 42. i 43.

Ostrzeżenie

> Algorytmy, takie jak `binary_search()`, są efektywne jedynie w posortowanych kontenerach. Użycie tego rodzaju algorytmu w nieposortowanym kontenerze może spowodować niepożądane konsekwencje.

Uwaga

> Sposób użycia algorytmu `stable_sort` jest taki sam jak pokazanego w powyższym programie algorytmu `sort`. Wymieniony algorytm `stable_sort` gwarantuje, że względna kolejność posortowanych elementów zostanie zachowana. Zachowanie tej względnej kolejności wiąże się z pewnym kosztem dotyczącym wydajności działania — trzeba o tym pamiętać zwłaszcza wtedy, kiedy względna kolejność elementów nie ma istotnego znaczenia.

Partycjonowanie zakresu

Algorytm `std::partition` pozwala na partycjonowanie zakresu źródłowego na dwie części: część spełniającą predykat jednoargumentowy i część, która nie spełnia tego predykatu:

```
bool IsEven (const int& nNumber)  // Predykat jednoargumentowy.
{
   return ((nNumber % 2) == 0);
}
...
partition (vecIntegers.begin(), vecIntegers.end(), IsEven);
```

Jednak algorytm `std::partition` nie gwarantuje zachowania względnej kolejności elementów w ramach każdej partycji. W celu zachowania względnej kolejności elementów należy użyć algorytmu `std::stable_partition`:

```
stable_partition (vecIntegers.begin(), vecIntegers.end(), IsEven);
```

W listingu 23.11 zademonstrowano sposób użycia wymienionych algorytmów.

Listing 23.11. Użycie algorytmów partition i stable_partition w celu partycjonowania zakresu przechowującego liczby całkowite na części przechowujące liczby parzyste i nieparzyste

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3: using namespace std;
 4:
 5: bool IsEven (const int& nNumber)
 6: {
 7:     return ((nNumber % 2) == 0);
 8: }
 9:
10: template <typename T>
11: void DisplayContents (const T& Input)
12: {
13:    for(auto iElement = Input.cbegin() // auto i cbegin(): C++11.
14:       ; iElement != Input.cend() // cend() to nowość w C++11.
15:       ; ++ iElement )
16:       cout << *iElement << ' ';
17:
18:    cout << " | Liczba elementów: " << Input.size() << endl;
19: }
20: int main ()
21: {
22:     vector <int> vecIntegers;
23:
24:     for (int nNum = 0; nNum < 10; ++ nNum)
25:         vecIntegers.push_back (nNum);
26:
27:     cout << "Zawartość początkowa: " << endl;
28:     DisplayContents(vecIntegers);
29:
30:     vector <int> vecCopy (vecIntegers);
31:
32:     cout << "Zawartość obiektu vector po użyciu algorytmu 'partition'
       ↪jest następująca:" << endl;
33:     partition (vecIntegers.begin (), vecIntegers.end (), IsEven);
34:     DisplayContents(vecIntegers);
35:
```

```
36:     cout << "Efekt użycia algorytmu 'stable_partition' jest
        ↪następujący: " << endl;
37:     stable_partition (vecCopy.begin (), vecCopy.end (), IsEven);
38:     DisplayContents(vecIntegers);
39:
40:     return 0;
41: }
```

Wynik ▼

```
Zawartość początkowa:
0 8 2 6 4 5 3 7 1 9 | Liczba elementów: 10

Zawartość obiektu vector po użyciu algorytmu 'partition' jest następująca:
0 8 2 6 4 5 3 7 1 9 | Liczba elementów: 10

Efekt użycia algorytmu 'stable_partition' jest następujący:
0 2 4 6 8 1 3 5 7 9 | Liczba elementów: 10
```

Analiza ▼

Powyższy kod powoduje podział na partycje zakresu liczb całkowitych przechowywanych w obiekcie `vector` (`vecIntegers`) na wartości parzyste i nieparzyste. Operacja podziału na partycje jest po raz pierwszy przeprowadzana w wierszu 33. za pomocą algorytmu `std::partition`, a następnie w wierszu 37. przy użyciu algorytmu `stable_partition`. Aby zachować możliwość porównania, zakres przykładowy `vecIntegers` zostaje skopiowany do `vecCopy`. Ten pierwszy został partycjonowany algorytmem `std::partition`, podczas gdy ten drugi algorytmem `std::stable_partition`. Efekt użycia algorytmu `stable_partition` zamiast `partition` jest widoczny w wyświetlonych danych wyjściowych. Algorytm `stable_partition` powoduje zachowanie względnej kolejności elementów w każdej partycji. Warto pamiętać, że zachowanie tej kolejności wiąże się z pewnym kosztem, który w zależności od typu obiektów przechowywanych w zakresie może być mały (jak ma to miejsce w omawianym przykładzie) bądź znaczący.

Uwaga

> Algorytm `stable_partition` jest wolniejszy w działaniu niż `partition` i dlatego powinien być używany tylko wtedy, gdy zachowanie względnej kolejności elementów ma istotne znaczenie w programie.

Wstawianie elementów do posortowanej kolekcji

Bardzo często ważne jest, aby elementy wstawiane do posortowanej kolekcji były umieszczane we właściwych położeniach. Biblioteka STL oferuje funkcje, takie jak `lower_bound()` i `upper_bound()`, pomagające w spełnieniu tego rodzaju wymagań:

```
auto iMinInsertPos = lower_bound ( listNames.begin(), listNames.end()
   , "Marek Kondrat" );
// Rozwiązanie alternatywne:
auto iMaxInsertPos = upper_bound ( listNames.begin(), listNames.end()
   , "Marek Kondrat" );
```

Funkcje `lower_bound()` i `upper_bound()` zwracają minimalne i maksymalne położenie w posortowanym zakresie, w którym element może zostać umieszczony bez naruszenia kolejności sortowania.

W listingu 23.12 zademonstrowano sposób użycia funkcji `lower_bound()` podczas wstawiania elementu w pozycji minimalnej w posortowanym obiekcie `list`.

Listing 23.12. Używanie funkcji lower_bound() i upper_bound() w posortowanej kolekcji

```
 0: #include <algorithm>
 1: #include <list>
 2: #include <string>
 3: #include <iostream>
 4: using namespace std;
 5:
 6: template <typename T>
 7: void DisplayContents (const T& Input)
 8: {
 9:    for(auto iElement = Input.cbegin() // auto i cbegin(): C++11.
10:       ; iElement != Input.cend() // cend() to nowość w C++11.
11:       ; ++ iElement )
12:       cout << *iElement << endl;
13: }
14: int main ()
15: {
16:     list<string> listNames;
17:
18:     // Wstawienie wartości przykładowych.
19:     listNames.push_back ("Cezary Pazura");
20:     listNames.push_back ("Robert Gonera");
```

```
21:     listNames.push_back ("Marek Kondrat");
22:     listNames.push_back ("Marek Perepeczko");
23:     listNames.push_back ("Anna Dereszowska");
24:
25:     cout << "Posortowany obiekt list zawiera elementy w następującej
        ↪kolejności: " << endl;
26:     listNames.sort();
27:     DisplayContents(listNames);
28:
29:     cout << "Najniższa wartość indeksu, w którym można umieścić element
        ↪\"Robert Gonera\" to: ";
30:     auto iMinInsertPos = lower_bound ( listNames.begin (),
        ↪listNames.end ()
31:        , "Robert Gonera" );
32:     cout << distance (listNames.begin (), iMinInsertPos) << endl;
33:
34:     cout << "Najwyższa wartość indeksu, w którym można umieścić element
        ↪\"Robert Gonera\" to: ";
35:     auto iMaxInsertPos = upper_bound ( listNames.begin (),
        ↪listNames.end ()
36:        , "Robert Gonera" );
37:     cout << distance (listNames.begin (), iMaxInsertPos) << endl;
38:
39:     cout << endl;
40:
41:     cout << "Obiekt list po wstawieniu elementu Robert Gonera: " << endl;
42:     listNames.insert (iMinInsertPos, "Robert Gonera");
43:
44:     DisplayContents(listNames);
45:     return 0;
46: }
```

Wynik ▼

```
Posortowany obiekt list zawiera elementy w następującej kolejności:
Anna Dereszowska
Cezary Pazura
Marek Kondrat
Marek Perepeczko
Robert Gonera

Najniższa wartość indeksu, w którym można umieścić element "Robert
↪Gonera" to: 4
Najwyższa wartość indeksu, w którym można umieścić element "Robert
↪Gonera" to: 5

Obiekt list po wstawieniu elementu Robert Gonera:
Anna Dereszowska
```

```
Cezary Pazura
Marek Kondrat
Marek Perepeczko
Robert Gonera
Robert Gonera
```

Analiza ▼

W posortowanej kolekcji element może być wstawiony w dwóch położeniach. Pierwsze jest położenie najniższe (czyli najbliższe początkowi kolekcji), zwracane przez funkcję `lower_bound()`. Natomiast drugie jest iteratorem zwracanym przez funkcję `upper_bound()` i jest najwyższe (czyli najdalsze od początku kolekcji). W przypadku programu pokazanego w listingu 23.12, w którym wstawiany do kolekcji ciąg tekstowy (`string`) „Robert Gonera" już w nim istnieje (wstawiony w wierszu 20.), wartości najniższa i najwyższa są różne (gdyby wstawiany ciąg tekstowy nie istniał w kolekcji, wówczas byłyby takie same). Użycie omawianych funkcji zostało pokazane w wierszach (odpowiednio) 30. i 35. Jak można zobaczyć w danych wyjściowych, kiedy iterator zwrócony przez funkcję `lower_bound()` jest używany podczas wstawiania ciągu tekstowego do obiektu `list` (wiersz 42.), powoduje, że lista zachowuje swój posortowany stan. Oznacza to możliwość wstawienia elementu w takim położeniu kolekcji, które nie powoduje naruszenia posortowanej natury zawartości kolekcji. Użycie iteratora zwróconego przez funkcję `upper_bound()` odbywa się w taki sam sposób.

TAK	NIE
Pamiętaj o użyciu metody `erase()` kontenera po algorytmach, takich jak `remove()`, `remove_if()` lub `unique()`. Metoda `erase()` powoduje zmianę wielkości kontenera. Zawsze sprawdzaj poprawność iteratora zwróconego przez metody `find()`, `find_if()`, `search()` lub `search_if()`. Sprawdzenie przez porównanie iteratora względem wartości zwrotnej metody `end()` powinno nastąpić przed użyciem iteratora.	Nie zapominaj o posortowaniu kontenera przy użyciu metody `sort()` przed wywołaniem `unique()` w celu usunięcia powtarzających się sąsiadujących wartości. Metoda `sort()` gwarantuje, że wszystkie elementy o takich samych wartościach będą ułożone obok siebie, co oznacza efektywne działanie metody `unique()`.

TAK	NIE
Wybieraj algorytm `stable_partition()` zamiast `partition()` i `stable_sort()` zamiast `sort()` tylko wtedy, gdy względna kolejność posortowanych elementów jest ważna. Wersje `stable_*` algorytmów mogą zmniejszyć wydajność działania aplikacji.	Nie zapominaj, że algorytm `binary_search()` powinien być używany tylko w posortowanych kontenerach.

Podsumowanie

W tej lekcji poznałeś jeden z najważniejszych i jednocześnie oferujący największe możliwości aspekt STL, czyli algorytmy. Dokładnie przeanalizowałeś różne rodzaje algorytmów, a dzięki przykładowym fragmentom kodu zobaczyłeś, jak można je stosować w tworzonych aplikacjach.

Pytania i odpowiedzi

Pytanie: Czy mogę używać algorytmu zmiennego, takiego jak `std::transform`, względem kontenera asocjacyjnego, np. `std::set`?

Odpowiedź: Nawet jeśli byłoby to możliwe, nie powinno się tego robić. Zawartość kontenera asocjacyjnego powinna być traktowana jako stała. Wynika to z tego, że kontenery asocjacyjne sortują elementy podczas ich wstawiania, a więc względna kolejność elementów odgrywa ważną rolę nie tylko w funkcjach, takich jak `find()`, ale ma również znaczenie w wydajności działania kontenera. Z tego powodu algorytmy zmienne, takie jak `std::transform`, nie powinny być używane względem obiektów STL `set`.

Pytanie: Muszę ustawić określoną wartość każdemu kolejnemu elementowi kontenera. Czy do takiej operacji powinienem wykorzystać algorytm `std::transform`?

Odpowiedź: Wprawdzie algorytmu `std::transform` można użyć do wykonania takiej operacji, jednak znacznie odpowiedniejsze będzie zastosowanie funkcji `fill()` lub `fill_in()`.

Pytanie: Czy funkcja `copy_backward()` powoduje odwrócenie kolejności elementów w kontenerze docelowym?

Odpowiedź: Nie, nie powoduje. Algorytm STL `copy_backward()` powoduje odwrócenie kolejności, w jakiej elementy są kopiowane, ale nie kolejności przechowywania tych elementów. Oznacza to, że proces kopiowania rozpoczyna się od końca zakresu i posuwa w kierunku jego początku. W celu odwrócenia kolejności elementów kolekcji należy użyć funkcji `std::reverse()`.

Pytanie: Czy powinienem używać algorytmu `std::sort` względem obiektu `list`?

Odpowiedź: Algorytm `std::sort` może być używany względem obiektu `list` dokładnie w taki sam sposób, jak w przypadku dowolnego kontenera sekwencyjnego. Jednak obiekt `list` wymaga zachowania specjalnej właściwości polegającej na tym, że operacja na nim nie spowoduje unieważnienia istniejących operatorów — tej właściwości algorytm `std::sort` nie jest w stanie zagwarantować. Z tego powodu obiekt STL `list` dostarcza algorytm `sort` w postaci funkcji składowej `list::sort`, którą należy wykorzystywać. Gwarantuje ona pozostawienie nienaruszonych iteratorów w obiekcie `list`, nawet jeśli ich względne położenie w obiekcie `list` zostanie zmienione.

Pytanie: Dlaczego tak ważne jest używanie funkcji, takich jak `lower_bound()` lub `upper_bound()`, podczas wstawiania elementów do posortowanego zakresu?

Odpowiedź: Wymienione funkcje podają (odpowiednio) pierwsze i ostatnie położenie, w którym element może być wstawiony do posortowanej kolekcji bez naruszania kolejności sortowania.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Masz za zadanie usunąć z obiektu `list` elementy spełniające określony warunek. Której funkcji użyjesz: `std::remove_if()` czy `list::remove_if()`?
2. Masz obiekt `list` klasy `ContactItem`. W jaki sposób funkcja `list::sort()` posortuje elementy w tym obiekcie, jeśli nie zostanie wyraźnie zdefiniowany predykat dwuargumentowy?
3. Jak często algorytm STL `generate` wywołuje funkcję `generator()`?
4. Co odróżnia algorytm `std::transform` od `std::for_each`?

Ćwiczenia

1. Utwórz predykat dwuargumentowy, który będzie akceptował ciągi tekstowe jako parametry wejściowe i zwracał wartość na podstawie porównania tych ciągów tekstowych bez uwzględniania wielkości ich znaków.
2. Przy użyciu operacji kopiowania między dwiema sekwencjami przechowywanymi w dwóch odmiennych kontenerach zademonstruj, jak algorytmy STL, takie jak `copy`, używają iteratorów w celu wykonywania swoich zadań bez konieczności poznawania natury kolekcji docelowej.
3. Tworzysz aplikację zapisującą cechy charakterystyczne gwiazd pojawiających się na horyzoncie w kolejności pojawiania się. W astronomii wielkość gwiazdy ma ogromne znaczenie, podobnie jak informacje o jej pojawieniu się i kierunku poruszania. Podczas sortowania tej kolekcji gwiazd względem ich wielkości użyjesz algorytmu `std::sort` czy `std::stable_sort`?

Lekcja 24

Kontenery adaptacyjne: stack i queue

Standardowa biblioteka wzorców (STL) zawiera kontenery adaptujące, które umożliwiają symulację zachowania stosu i kolejki. Tego rodzaju kontenery, które wewnętrznie używają innych kontenerów i oferują odmienny sposób zachowania, nazywane są *kontenerami adaptacyjnymi*.

Z tej lekcji dowiesz się:

- **jakie są cechy charakterystyczne zachowania obiektów stack i queue,**
- **jak używać obiektu STL stack,**
- **jak korzystać z obiektu STL queue,**
- **jak stosować obiekt STL `priority_queue`.**

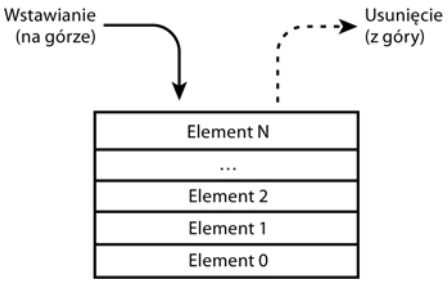
Cechy charakterystyczne zachowania stosów i kolejek

Stosy i kolejki są bardzo podobne do tablic i list, ale zawierają ograniczenia dotyczące sposobu wstawiania elementów, usuwania ich i uzyskiwania dostępu do nich. Cechy charakterystyczne ich zachowania są dokładnie ustalane i zależą od umieszczenia elementów podczas wstawiania bądź położenia elementu, który może być usunięty z kontenera.

Stosy

Stosy są systemami typu LIFO (ang. *Last-In-First-Out*, ostatni na wejściu, pierwszy na wyjściu), w którym elementy mogą być wstawiane lub usuwane z góry kontenera. Wizualnie stos można przedstawić jak talerze umieszczone jeden na drugim. Ostatni talerz na stosie będzie pierwszym talerzem zdjętym z tego stosu (patrz rysunek 24.1). Talerzy znajdujących się w środku bądź na spodzie stosu nie można kontrolować.

RYSUNEK 24.1. Operacje na stosie



Takie zachowanie stosu talerzy jest symulowane za pomocą ogólnego kontenera STL `std::stack`.

Wskazówka

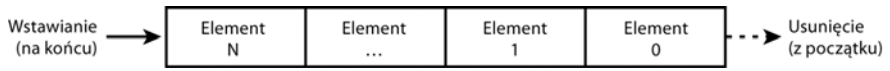
W celu użycia klasy `std::stack` w programie trzeba umieścić następujący nagłówek:

```
#include <stack>
```

Kolejki

Kolejki są systemami typu FIFO (ang. *First-In-First-Out*, pierwszy na wejściu, pierwszy na wyjściu), w którym elementy mogą być wstawiane za kolejnymi, a element wstawiony jako pierwszy będzie usunięty również jako pierwszy.

Wizualnie kolejkę można przedstawić jako kolejkę osób czekających do okienka na poczcie — osoby, które ustawiły się w kolejce wcześniej, będą wcześniej obsłużone (patrz rysunek 24.2).



RYSUNEK 24.2. Operacje na kolejce

Takie zachowanie kolejki jest symulowane za pomocą ogólnego kontenera STL `std::queue`.

Wskazówka

W celu użycia klasy `std::queue` w programie trzeba umieścić następujący nagłówek:

```
#include <queue>
```

Używanie klasy STL stack

STL `stack` to klasa wzorca, która do działania wymaga dołączenia w programie nagłówka `<stack>`. Jest to klasa ogólna pozwalająca na wstawianie i usuwanie elementów na górze stosu. Nie ma żadnych możliwości kontrolowania bądź uzyskania dostępu do elementów znajdujących się w środku stosu. Pod tym względem kontener `std::stack` bardzo przypomina zachowanie stosu talerzy.

Ustanawianie obiektu stack

Przez niektóre implementacje STL klasa `std::stack` jest definiowana następująco:

```
template <
   class elementType,
   class Container=deque<Type>
> class stack;
```

Parametr *`elementType`* jest typem obiektu odkładanego na stosie. Drugi parametr wzorca, *`Container`*, to domyślna implementacja klasy kontenera; `std::deque` jest wartością domyślną dla wewnętrznego magazynu danych stosu i może zostać zastąpiona przez `std::vector` lub `std::list`. Dlatego też utworzenie stosu liczb całkowitych odbywa się następująco:

```
std::stack <int> stackInts;
```

Jeżeli chcesz utworzyć stos obiektów dowolnego typu, np. `Tuna`, musisz użyć poniższej składni:

```
std::stack <Tuna> stackTunas;
```

W celu utworzenia stosu opartego na innym kontenerze użyj polecenia:

```
std::stack <double, vector <double> > stackDoublesInVector;
```

W listingu 24.1 zademonstrowano ustanawianie wzorca dla `std::stack`.

Listing 24.1. Ustanawianie obiektu STL stack

```
 0: #include <stack>
 1: #include <vector>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:
 7:     // Stos liczb całkowitych.
 8:     stack <int> stackInts;
 9:
10:     // Stos liczb typu double.
11:     stack <double> stackDoubles;
12:
13:     // Stos liczb typu double przechowywanych w obiekcie vector.
14:     stack <double, vector <double> > stackDoublesInVector;
15:
16:     // Inicjalizacja stosu na podstawie innego.
17:     stack <int> stackIntsCopy(stackInts);
18:
19:     return 0;
20: }
```

Analiza ▼

Powyższy program nie wyświetla danych wyjściowych; pokazano w nim ustanawianie wzorca obiektu STL `stack`. W wierszach 8. i 11. następuje ustanowienie dwóch obiektów `stack` przechowujących elementy typu (odpowiednio) `int` i `double`. W wierszu 14. również ustanowiono obiekt `stack` przechowujący obiekty typu `double`, ale tym razem zastosowany został drugi parametr wzorca. Typ klasy kolekcji, której obiekt `stack` powinien wewnętrznie używać, ustawiono jako `vector`. Jeżeli drugi parametr wzorca nie zostanie podany, w stosie automatycznie będzie użyty `std::deque`. Wreszcie, w wierszu 17. zaprezentowano sposób utworzenia obiektu stosu jako kopii innego.

Funkcje składowe klasy stack

Kontener `stack` adaptujący inne kontenery, np. deque, `list` lub `vector`, implementuje swoją funkcjonalność poprzez ograniczenie sposobu, w jaki elementy mogą być wstawiane bądź usuwane. W ten sposób zapewnia zachowanie, które jest oczekiwane od mechanizmu działającego jak stos. W tabeli 24.1 wymieniono publiczne funkcje składowe klasy `stack` oraz zademonstrowano przykłady ich użycia dla stosu liczb całkowitych.

Tabela 24.1. Funkcje składowe kontenera std::stack

Funkcja	Opis
`push()`	Wstawia element na górze stosu. `stackIntegers.push(25);`
`pop()`	Usuwa element z góry stosu. `stackIntegers.pop();`

`empty()`	Sprawdza, czy stos jest pusty. Zwraca wartość boolowską.
	`if (stackIntegers.empty ())`
	`  DoSomething();`

`size()`	Zwraca liczbę elementów na stosie. `size_t nNumElements = stackIntegers.size();`

`top()`	Pobiera odniesienie do elementu umieszczonego najwyżej na stosie.
	`cout << "Element znajdujący się na górze stosu to = " <<`
	`↪stackIntegers.top();`

Jak widać w powyższej tabeli, publiczne funkcje składowe stosu udostępniają jedynie te metody, które pozwalają na wstawianie i usuwanie elementów w położeniach zgodnych ze sposobem działania stosu. Oznacza to, że jeśli nawet stosowanym kontenerem będzie deque, `vector` lub `list`, cechy charakterystyczne tego kontenera będą ograniczone w celu wymuszenia zachowania charakterystycznego dla stosu.

Wstawianie i usuwanie elementów z góry stosu przy użyciu metod push() i pop()

Wstawianie elementów odbywa się z wykorzystaniem metody składowej `stack<T>::push()`:

```
stackInts.push (25); // Element 25. znajduje się na górze stosu.
```

Stos z definicji pozwala na uzyskanie dostępu do elementów znajdujących się na górze stosu; używa się do tego funkcji składowej `top()`:

```
cout << stackInts.top() << endl;
```

Jeżeli chcesz usunąć element znajdujący się na górze stosu, musisz zastosować metodę `pop()`:

```
stackInts.pop (); // Metoda pop: usunięcie elementu znajdującego się na samej górze stosu.
```

W listingu 24.2 zademonstrowano wstawianie i usuwanie elementów ze stosu przy użyciu metod (odpowiednio) `push()` i `pop()`.

Listing 24.2. Praca ze stosem liczb całkowitych

```
 0: #include <stack>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:     stack <int> stackInts;
 7:
 8:     // Umieszczenie wartości przykładowych na górze stosu.
 9:     cout << "Umieszczenie liczb {25, 10, -1, 5} na stosie:" << endl;
10:     stackInts.push (25);
11:     stackInts.push (10);
12:     stackInts.push (-1);
13:     stackInts.push (5);
14:
15:     cout << "Stos zawiera " << stackInts.size () << " elementy" << endl;
16:     while (stackInts.size () != 0)
17:     {
18:         cout << "Usunięcie ze stosu elementu: " << stackInts.top() <<
        ↪endl;
19:         stackInts.pop (); // Metoda pop:  usunięcie elementu znajdującego się
            ↪na samej górze stosu.
20:     }
21:
22:     if (stackInts.empty ()) // Wartość true ze względu na wcześniejsze wywołania
        ↪metody pop.
23:         cout << "Stos jest teraz pusty!" << endl;
24:
25:     return 0;
26: }
```

Wynik ▼

```
Umieszczenie liczb {25, 10, -1, 5} na stosie:
Stos zawiera 4 elementy

Usunięcie ze stosu elementu: 5
Usunięcie ze stosu elementu: -1
Usunięcie ze stosu elementu: 10
Usunięcie ze stosu elementu: 25

Stos jest teraz pusty!
```

Analiza ▼

W powyższym programie pierwszym krokiem jest umieszczenie przykładowych liczb na stosie (`stackInts`) za pomocą funkcji `stack::push()`, jak pokazano w wierszach od 9. do 13. Następnie kod usuwa elementy z tego stosu przy użyciu metody `stack::pop()`. Ponieważ stos umożliwia dostęp jedynie do elementu znajdującego się na górze, dostęp do tego elementu odbywa się z wykorzystaniem funkcji składowej `stack::top()`, co pokazano w wierszu 18. Elementy ze stosu można usuwać pojedynczo za pomocą funkcji `stack::pop()`, jak pokazano w wierszu 19. Zastosowana w tym miejscu pętla `while` gwarantuje wykonywanie operacji `pop()`, aż do chwili opróżnienia stosu. Jak można zobaczyć na podstawie kolejności usuwanych elementów, elementy wstawione jako pierwsze zostały usunięte jako pierwsze. To typowe zachowanie LIFO (ang. *Last-In-First-Out*) stosu.

W listingu 24.2 zademonstrowano wszystkie pięć funkcji składowych klasy `stack`. Warto zwrócić uwagę, że funkcje `push_back()` i `insert()`, które są dostępne we wszystkich kontenerach sekwencyjnych STL używanych w tle jako kontenery klasy `stack`, są niedostępne w postaci funkcji składowych obiektu `stack`. To samo dotyczy iteratorów pomagających w używaniu elementów, które nie znajdują się na górze kontenera. Obiekt `stack` udostępnia jedynie element znajdujący się na samej górze, nic więcej.

Używanie klasy STL queue

STL queue to klasa wzorca, która do działania wymaga dołączenia w programie nagłówka <queue>. Jest to klasa ogólna pozwalająca na wstawianie elementów jedynie na końcu obiektu i usuwanie elementów jedynie z początku obiektu.

Nie ma żadnych możliwości kontrolowania bądź uzyskania dostępu do elementów znajdujących się w środku kolejki, dostęp można uzyskać tylko do elementów znajdujących się na początku i końcu kolejki. Pod tym względem kontener `std::queue` bardzo przypomina zachowanie osób stojących w kolejce do kasy w supermarkecie!

Ustanawianie obiektu queue

Klasa `std::queue` jest definiowana następująco:

```
template <
   class elementType,
   class Container = deque<Type>
> class queue;
```

Tutaj parametr *`elementType`* jest typem elementu umieszczanego w obiekcie queue. Drugi parametr wzorca, *`Container`*, to typ kolekcji używanej przez klasę `std::queue` do obsługi danych. Wartością domyślną jest deque, ale może ona zostać zastąpiona przez `std::vector` lub `std::list`.

Kolejkę liczb całkowitych najprościej można utworzyć następująco:

```
std::queue <int> qIntegers;
```

Jeżeli chcesz zbudować kolejkę zawierającą obiekty typu double w kontenerze `std::list` (zamiast w domyślnym deque), wtedy musisz użyć następującej składni:

```
std::queue <double, list <double> > qDoublesInList;
```

Podobnie jak stos, kolejka również może zostać utworzona na podstawie innej:

```
std::queue<int> qCopy(qIntegers);
```

W listingu 24.3 zademonstrowano ustanawianie wzorca dla `std::queue`.

Listing 24.3. Ustanowienie obiektu STL queue

```
 0: #include <queue>
 1: #include <list>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:
```

```
 7:     // Kolejka liczb całkowitych.
 8:     queue <int> qIntegers;
 9:
10:     // Kolejka liczb typu double.
11:     queue <double> qDoubles;
12:
13:     // Kolejka liczb typu double przechowywanych w obiekcie list.
14:     queue <double, list <double> > qDoublesInList;
15:
16:     // Kolejka utworzona na podstawie innej.
17:     queue<int> qCopy(qIntegers);
18:
19:     return 0;
20: }
```

Analiza

W powyższym programie zademonstrowano, jak ogólna klasa STL queue może być ustanowiona w celu utworzenia kolejki liczb całkowitych (`integers`), co pokazano w wierszu 8., oraz kolejki liczb typu double, tak jak w wierszu 11. W wierszu 14. również ustanowiono obiekt queue (`qDoublesInList`), w którym wyraźnie określono, że kontenerem zaadaptowanym w tle przez queue do wewnętrznego przechowywania danych ma być `std::list`. Wskazuje na to drugi parametr wzorca. Jeżeli drugi parametr wzorca nie zostanie podany, jak ma to miejsce w przypadku dwóch pierwszych kolejek, domyślnym kontenerem używanym w tle przez obiekt queue do przechowywania jego zawartości będzie `std::deque`.

Funkcje składowe klasy queue

Podobnie jak `std::stack`, także implementacja `std::queue` bazuje na kontenerze STL, np. deque, `list` lub `vector`. Obiekt queue udostępnia kilka funkcji składowych, które implementują zachowanie charakterystyczne dla kolejki. W tabeli 24.2 wymieniono publiczne funkcje składowe, używające obiektu `qIntegers`, który w listingu 24.3 jest obiektem queue przechowującym liczby całkowite.

Tabela 24.2. Funkcje składowe kontenera std::queue

Funkcja	Opis
`push()`	Wstawia element na końcu kolejki, tzn. w ostatniej pozycji. `qIntegers.push(10);`
`pop()`	Usuwa element z początku kolejki, tzn. z pierwszej pozycji. `qIntegers.pop();`
`front()`	Zwraca odniesienie do elementu znajdującego się na początku kolejki. `cout << "Element znajdujący się na początku kolejki to: " << qIntegers.front();`
`back()`	Zwraca odniesienie do elementu znajdującego się na końcu kolejki, tzn. do ostatniego wstawionego elementu. `cout << "Element znajdujący się na końcu kolejki to: " << qIntegers.back();`

`empty()`	Sprawdza, czy kolejka jest pusta. Zwraca wartość boolowską.
	`if (qIntegers.empty ())`
	`  cout << "Kolejka jest pusta!";`

`size()`	Zwraca liczbę elementów w kolejce. `size_t nNumElements = qIntegers.size();`

Kontener STL queue nie zawiera funkcji, takich jak `begin()` i `end()`, które są dostarczane przez większość kontenerów STL, łącznie z `deque`, `vector` lub `list` używanymi w tle przez klasę `queue`. Jest to celowe, aby względem obiektu `queue` można było przeprowadzić tylko te operacje, które są zgodne z cechami charakterystycznymi zachowania kolejki.

Wstawianie na końcu i usuwanie na początku kolejki przy użyciu metod push() i pop()

W kolejce elementy są wstawianie na końcu przy użyciu funkcji składowej `push()`:

```
qIntegers.push (5);  // Wstawione elementy są umieszczane na końcu kolejki.
```

Z kolei elementy są usuwane z początku kolejki przy użyciu funkcji składowej `pop()`:

```
qIntegers.pop ();  // Usunięcie elementu na początku kolejki.
```

W przeciwieństwie do `stack`, obiekt queue pozwala na obsługę elementów po obu stronach. Oznacza to możliwość obsługi elementów znajdujących się na początku i końcu kontenera za pomocą funkcji `front()` i `back()`:

```
cout << "Element na początku kolejki: " << qIntegers.front() << endl;
cout << "Element na końcu kolejki: " << qIntegers.back () << endl;
```

Wstawianie, usuwanie oraz przeglądanie kolejki zostało zademonstrowane w listingu 24.4.

Listing 24.4. Wstawianie, usuwanie i przeglądanie elementów w obiekcie queue przechowującym liczby całkowite

```
 0: #include <queue>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:     queue <int> qIntegers;
 7:
 8:     cout << "Wstawianie elementów {10, 5, -1, 20} do kolejki" << endl;
 9:     qIntegers.push (10);
10:     qIntegers.push (5);
11:     qIntegers.push (-1);
12:     qIntegers.push (20);
13:
14:     cout << "Kolejka zawiera " << qIntegers.size () << " elementy" <<
        ↪endl;
15:     cout << "Element na początku: " << qIntegers.front() << endl;
16:     cout << "Element na końcu: " << qIntegers.back () << endl;
17:
18:     while (qIntegers.size () != 0)
19:     {
20:         cout << "Usuwanie elementu " << qIntegers.front () << endl;
21:         qIntegers.pop (); // Usunięcie elementu na początku kolejki.
22:     }
23:
24:     if (qIntegers.empty ()) // Wartość true ze względu na wcześniejsze wywołania
        ↪metody pop.
25:         cout << "Kolejka jest teraz pusta!";
26:
27:     return 0;
28: }
```

Wynik ▼

```
Wstawianie elementów {10, 5, -1, 20} do kolejki
Kolejka zawiera 4 elementy
Element na początku: 10
Element na końcu: 20
Usuwanie elementu 10
Usuwanie elementu 5
Usuwanie elementu -1
Usuwanie elementu 20
Kolejka jest teraz pusta!
```

Analiza ▼

Elementy do kolejki `qIntegers` zostały dodane za pomocą funkcji `push()`, która umieszcza je na końcu kolejki (patrz wiersze od 9. do 12.). Funkcje `front()` i `back()` są używane w celu odwołania się do elementów znajdujących się w położeniu początkowym i końcowym kolejki, jak pokazano w wierszach 15. i 16. Pętla `while` w wierszach od 18. do 22. powoduje wyświetlenie elementu znajdującego się na początku kolejki, a następnie jego usunięcie za pomocą operacji `pop()` pokazanej w wierszu 21. Pętla ta kontynuuje działanie, aż do opróżnienia kolejki. Dane wyjściowe programu pokazują, że elementy są usuwane z kolejki w tej samej kolejności, w jakiej zostały wstawione, ponieważ elementy są wstawiane na końcu kolejki, ale usuwane na jej początku.

Używanie klasy STL priority_queue

STL `priority_queue` to klasa wzorca, która do działania wymaga dołączenia w programie nagłówka <queue>. Klasa `priority_queue` różni się od klasy `queue` pod tym względem, że element o największej wartości (lub wartości uznawanej za najwyższą przez predykat dwuargumentowy) jest umieszczony na początku kolejki i tylko na początku kolejki można przeprowadzać operacje.

Ustanawianie obiektu priority_queue

Oto definicja klasy `std::priority_queue`:

```
template <
   class elementType,
   class Container=vector<Type>,
      class Compare=less<typename Container::value_type>
>
class priority_queue
```

Tutaj parametr *elementType* jest parametrem wzorca przekazującym typ elementów umieszczanych w obiekcie `priority_queue`. Drugi parametr wzorca określa klasę kolekcji wewnętrznie używaną przez `priority_queue` do przechowywania danych. Natomiast trzeci parametr pozwala programiście na zdefiniowanie predykatu dwuargumentowego pomagającego kolejce w ustaleniu elementu znajdującego się na górze. W przypadku braku tego predykatu dwuargumentowego klasa `priority_queue` będzie domyślnie korzystała z `std::less`, który porównuje dwa obiekty za pomocą operatora <.

Najprostsza postać utworzenia obiektu `priority_queue` zawierającego liczby całkowite przedstawia się następująco:

```
std::priority_queue <int> pqIntegers;
```

Jeżeli chcesz utworzyć kolejkę priorytetową zawierającą obiekty typu double w kontenerze `std::deque`, musisz użyć następującej składni:

```
priority_queue <int, deque <int>, greater <int> > pqIntegers_Inverse;
```

Podobnie jak stos, kolejka również może zostać utworzona na podstawie innej:

```
std::priority_queue<int> pqCopy(pqIntegers);
```

Ustanowienie obiektu `priority_queue` zostało zademonstrowane w listingu 24.5.

Listing 24.5. Ustanowienie obiektu STL priority_queue

```
 0: #include <queue>
 1: #include <functional>
 2: int main ()
 3: {
 4:     using namespace std;
 5:
 6:     // Obiekt priority_queue przechowujący liczby całkowite posortowane za pomocą
        ➥std::less <> (domyślnie).
 7:     priority_queue <int> pqIntegers;
 8:
 9:     //  Obiekt priority_queue przechowujący liczby typu double.
10:     priority_queue <double> pqDoubles;
11:
12:     // Obiekt priority_queue przechowujący liczby całkowite posortowane za pomocą
        ➥std::greater <>.
13:     priority_queue <int, deque <int>, greater <int> > pqIntegers_Inverse;
14:
```

```
15:     // Kolejka utworzona na podstawie innej.
16:     priority_queue<int> pqCopy(pqIntegers);
17:
18:     return 0;
19: }
```

Analiza ▼

W wierszach 7. i 10. następuje utworzenie dwóch obiektów `priority_queue`. Pierwszy służy do przechowywania elementów typu `int`, natomiast drugi do przechowywania elementów typu `double`. Brak któregokolwiek innego parametru wzorca powoduje użycie `std::vector` jako wewnętrznego kontenera danych oraz domyślnych kryteriów porównywania dostarczanych przez `std::less`. Utworzone w programie kolejki mają priorytet nadawany w taki sposób, że najwyższa wartość jest udostępniana na początku kolejki priorytetów (za pomocą funkcji `front()`). Jednak `pqIntegers_Inverse` dostarcza `deque` jako wartość drugiego parametru wskazującego wewnętrzny kontener oraz `std::greater` jako predykat. Zastosowanie tego predykatu powoduje, że na początku kolejki znajduje się element o najmniejszej wartości.

Efekt użycia predykatu `std::greater<T>` stanie się jasny w listingu 24.7, który znajduje się dalej w tej lekcji.

Funkcje składowe klasy priority_queue

Dostępne w klasie `queue` funkcje składowe `front()` i `back()` są niedostępne w klasie `priority_queue`. W tabeli 24.3 wymieniono funkcje składowe udostępniane przez klasę `priority_queue`.

Jak pokazano w poniższej tabeli, dostęp do elementów obiektu `priority_queue` może odbywać się tylko za pomocą funkcji `top()`, która zwraca element o najwyższej wartości. Jest on obliczany za pomocą predykatu zdefiniowanego przez użytkownika bądź `std::less` w przypadku braku takiego predykatu.

Tabela 24.3. Funkcje składowe klasy std::priority_queue

Funkcja	Opis
`push()`	Wstawia element do kolejki priorytetów. `pqIntegers.push(10);`
`pop()`	Usuwa element z góry kolejki, tzn. największy element w kolejce. `pqIntegers.pop();`

`top()`	Zwraca odniesienie do największego elementu w kolejce (który jednocześnie zajmuje najwyższą pozycję).
	`cout << "Największy element w kolejce to: " <<`
	`↪pqIntegers.top();`

`empty()`	Sprawdza, czy kolejka priorytetów jest pusta. Zwraca jest wartość boolowską.
	`if (pqIntegers.empty ())`
	`cout << "Kolejka jest pusta!";`

`size()`	Zwraca liczbę elementów w kolejce priorytetów. `size_t nNumElements = pqIntegers.size();`

Wstawianie na końcu i usuwanie na początku kolejki priorytetowej przy użyciu metod push() i pop()

W kolejce priorytetowej elementy są wstawianie na końcu przy użyciu funkcji składowej `push()`:

```
pqIntegers.push (5);  // Wstawione elementy są umieszczane na końcu kolejki priorytetowej.
```

Z kolei elementy są usuwane z początku kolejki priorytetowej przy użyciu funkcji składowej `pop()`:

```
pqIntegers.pop ();  // Usunięcie elementu na początku kolejki priorytetowej.
```

Użycie funkcji składowych klasy `priority_queue` zostało zademonstrowane w listingu 24.6.

Listing 24.6. Praca z obiektem priority_queue z wykorzystaniem metod push(), top() i pop()

```
0: #include <queue>
1: #include <iostream>
2:
3: int main ()
4: {
```

```
 5:     using namespace std;
 6:
 7:     priority_queue <int> pqIntegers;
 8:     cout << "Wstawianie elementów {10, 5, -1, 20} do obiektu
        ↪priority_queue" << endl;
 9:     pqIntegers.push (10);
10:     pqIntegers.push (5);
11:     pqIntegers.push (-1);
12:     pqIntegers.push (20);
13:
14:     cout << "Usunięcie: " << pqIntegers.size () << " elementów" << endl;
15:     while (!pqIntegers.empty ())
16:     {
17:         cout << "Usuwanie elementu " << pqIntegers.top () << endl;
18:         pqIntegers.pop ();
19:     }
20:
21:     return 0;
22: }
```

Wynik ▼

```
Wstawianie elementów {10, 5, -1, 20} do obiektu priority_queue
Usunięcie 4 elementów
Usuwanie elementu 20
Usuwanie elementu 10
Usuwanie elementu 5
Usuwanie elementu -1
```

Analiza ▼

W powyższym programie następuje wstawienie przykładowych liczb całkowitych do obiektu `priority_queue` (patrz wiersze od 9. do 12.), a następnie usunięcie elementu znajdującego się na górze (początku) kolejki. Usunięcie obywa się za pomocą funkcji `pop()`, jak przedstawiono w wierszu 18. Dane wyjściowe pokazują, że element o największej wartości jest dostępny na początku kolejki. Dlatego też użycie funkcji `priority_queue::pop()` powoduje usunięcie elementu, który został uznany za element o największej wartości spośród wszystkich znajdujących się w kontenerze. Wspomniany element jest dostępny przy użyciu metody `top()`, jak pokazano w wierszu 17. Jeżeli nie został zdefiniowany predykat nadawania priorytetu, wtedy kolejka automatycznie sortuje elementy w kolejności malejącej (element o największej wartości znajduje się na górze).

W kolejnym przykładzie przedstawionym w listingu 24.7 pokazano użycie obiektu `priority_queue`, w którym jako predykat zastosowano `std::greater <int>`. Predykat powoduje, że obiekt queue uznaje najmniejszą liczbę za największą wartość, która następnie jest umieszczana na początku kolejki i dostępna za pomocą funkcji `front()`.

Listing 24.7. Obiekt priority_queue wraz z najmniejszą wartością na początku kolejki dzięki użyciu odpowiedniego predykatu

```
 0: #include <queue>
 1: #include <iostream>
 2: #include <functional>
 3: int main ()
 4: {
 5:     using namespace std;
 6:
 7:     // Zdefiniowanie obiektu priority_queue wraz z predykatem w postaci greater <int>.
 8:     priority_queue <int, vector <int>, greater <int> > pqIntegers;
 9:
10:     cout << "Wstawianie elementów {10, 5, -1, 20} do obiektu
        ↪priority_queue" << endl;
11:     pqIntegers.push (10);
12:     pqIntegers.push (5);
13:     pqIntegers.push (-1);
14:     pqIntegers.push (20);
15:
16:     cout << "Usunięcie: " << pqIntegers.size () << " elementów" << endl;
17:     while (!pqIntegers.empty ())
18:     {
19:         cout << "Usuwanie elementu " << pqIntegers.top () << endl;
20:         pqIntegers.pop ();
21:     }
22:
23:     return 0;
24: }
```

Wynik ▼

```
Wstawianie elementów {10, 5, -1, 20} do obiektu priority_queue
Usunięcie 4 elementów
Usuwanie elementu -1
Usuwanie elementu 5
Usuwanie elementu 10
Usuwanie elementu 20
```

Analiza ▼

Większość kodu oraz wszystkie wartości dostarczone obiektowi `priority_queue` w tym przykładzie celowo są takie same jak w poprzednim programie (patrz listing 24.6). Jednak dane wyjściowe pokazują, że te kolejki zachowują się w zupełnie odmienny sposób. Obiekt `priority_queue` porównuje przechowywane elementy za pomocą predykatu `greater <int>`, co pokazano w wierszu 8. W wyniku zastosowania tego predykatu najmniejsza liczba jest uznawana za liczbę o największej wartości i umieszczana w najwyższej pozycji (dostępna za pomocą funkcji `top()`). Stąd funkcja `top()` użyta w wierszu 19. zawsze wyświetla najmniejszą liczbę przechowywaną w obiekcie `priority_queue`, która następnie w wierszu 20. jest usuwana za pomocą funkcji `pop()`.

Z tego powodu podczas usuwania elementów ten obiekt `priority_queue` będzie usuwał coraz większe liczby.

Podsumowanie

W tej lekcji wyjaśniono używanie trzech kluczowych kontenerów adaptacyjnych — STL `stack`, `queue` oraz `priority_queue`. Wymienione kontenery adaptują kontenery sekwencyjne i wykorzystują je wewnętrznie do przechowywania danych. Mimo to, udostępniane przez nie funkcje składowe powodują, że cechy charakterystyczne w zachowaniu tych kontenerów są unikalne dla stosów i kolejek.

Pytania i odpowiedzi

Pytanie: Czy można modyfikować element znajdujący się w środku stosu?

Odpowiedź: Nie, taka możliwość byłaby niezgodna z zachowaniem stosu.

Pytanie: Czy mogę przeprowadzić iterację przez wszystkie elementy kolejki?

Odpowiedź: Kolejka nie udostępnia iteratorów, możliwy jest dostęp jedynie do elementów znajdujących się na początku i końcu kolejki.

Pytanie: Czy algorytmy STL działają wraz z kontenerami adaptacyjnymi?

Odpowiedź: Algorytmy STL działają za pomocą iteratorów. Ponieważ ani klasa `stack`, ani klasa queue nie dostarczają iteratorów oznaczających koniec zakresu, użycie algorytmów STL wraz z tymi kontenerami jest niemożliwe.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Czy możesz zmienić zachowanie obiektu `priority_queue` względem określonego elementu, np. w taki sposób, aby element o największej wartości został usunięty jako ostatni?
2. Masz obiekt `priority_queue` klasy `Coins`. Jaki operator składowy będziesz musiał zdefiniować w tej klasie, aby przedstawić na pozycji początkowej monetę o większej wartości?
3. Masz stos klasy `Coins`, na którym umieściłeś sześć elementów. Czy możesz uzyskać dostęp do pierwszej monety umieszczonej na stosie bądź ją usunąć?

Ćwiczenia

1. Kolejka osób (klasa `CPerson`) czeka do okienka na poczcie. Klasa `CPerson` zawiera atrybuty składowe przechowujące wiek i płeć osoby zdefiniowane jako:

```
class CPerson
{
  public:
    int Age;
    bool IsFemale;
};
```

Utwórz predykat dwuargumentowy dla obiektu `priority_queue`, który będzie pomagał w obsłużeniu osób starszych i kobiet (w tej kolejności) jako pierwszych.

2. Utwórz program odwracający podany przez użytkownika ciąg tekstowy, używając do tego klasy `stack`.

Lekcja 25

Praca z opcjami bitowymi za pomocą STL

Bity mogą być bardzo efektywnym sposobem przechowywania ustawień oraz opcji. Biblioteka STL obsługuje klasy pomagające w organizacji informacji bitowych i manipulacji nimi.

Z tej lekcji dowiesz się:

- **klasa bitset,**
- **klasa vector<bool>.**

Klasa bitset

Klasa `std::bitset` to klasa STL zaprojektowana do obsługi informacji przechowywanych w bitach oraz opcji bitowych. Klasa ta nie jest klasyfikowana jako klasa kontenera STL, ponieważ nie ma możliwości samodzielnej zmiany wielkości. To zdecydowanie klasa narzędziowa, która została zoptymalizowana do pracy z sekwencjami bitów o długości znanej już *w trakcie kompilacji*.

Wskazówka

W celu użycia klasy `std::bitset` w programie trzeba umieścić następujący nagłówek:

```
#include <bitset>
```

Ustanowienie klasy std::bitset

Ta klasa wzorca wymaga dołączenia w programie nagłówka `<bitset>` oraz potrzebuje jednego parametru wzorca dostarczającego liczbę bitów, którymi będzie zarządzał dany egzemplarz klasy:

```
bitset <4> fourBits;  // 4 zainicjalizowane jako 0000.
```

Możesz również zainicjalizować obiekt `bitset` jako bitową sekwencję umieszczoną w dosłownym ciągu tekstowym typu `char*`:

```
bitset <5> fiveBits ("10101"); // 5 bitów 10101.
```

Utworzenie obiektu `bitset` na podstawie innego obiektu tego typu także jest proste:

```
bitset <8> eightBitsCopy(eightbits);
```

W listingu 25.1 przedstawiono przykładowe ustanowienie klasy wzorca `std::bitset`.

Listing 25.1. Ustanowienie klasy std::bitset

```
0: #include <bitset>
1: #include <iostream>
2: #include <string>
3:
4: int main ()
5: {
6:     using namespace std;
7:
```

```
 8:     // Ustanowienie obiektu bitset przeznaczonego do przechowywania czterech bitów.
 9:     bitset <4> fourBits; // 4 bity zainicjalizowane jako 0000.
10:     cout << "Początkowa zawartość obiektu fourBits: " << fourBits <<
        ↪endl;
11:
12:     bitset <5> fiveBits (string ("10101")); // 5 bitów 10101.
13:     cout << "Początkowa zawartość obiektu fiveBits: " << fiveBits <<
        ↪endl;
14:
15:     bitset <8> eightBits (255); // 8 bitów zainicjalizowanych jako long int 255.
16:     cout << "Początkowa zawartość obiektu eightBits: " << eightBits <<
        ↪endl;
17:
18:     // Utworzenie obiektu bitset na podstawie innego obiektu tego typu.
19:     bitset <8> eightBitsCopy(eightBits);
20:
21:     return 0;
22: }
```

Wynik ▼

```
Początkowa zawartość obiektu fourBits: 0000
Początkowa zawartość obiektu fiveBits: 10101
Początkowa zawartość obiektu eightBits: 11111111
```

Analiza ▼

W powyższym przykładzie zademonstrowano cztery różne sposoby konstruowania obiektu `bitset`. Pierwszy za pomocą konstruktora domyślnego, który zainicjalizował sekwencję bitową z wartością 0, jak pokazano w wierszu 9. Drugi przy użyciu ciągu tekstowego w stylu C zawierającego reprezentację żądanej sekwencji bitów w postaci ciągu tekstowego, jak pokazano w wierszu 12. Trzeci za pomocą wartości `unsigned long` przechowującej wartość dziesiętną sekwencji bitowej, jak pokazano w wierszu 15. Czwarty z wykorzystaniem konstruktora kopiującego, jak pokazano w wierszu 19. Warto zwrócić uwagę, że w każdym z wymienionych sposobów trzeba jako parametr wzorca podać liczbę bitów, które mają być przechowywane przez obiekt `bitset`. Ta liczba jest stała, definiowana w trakcie kompilacji, a nie dynamiczna. W przeciwieństwie do wstawiania elementów do obiektu `vector`, w obiekcie `bitset` nie można umieścić większej liczby bitów niż zadeklarowana w kodzie.

Używanie klasy std::bitset i jej elementów składowych

Klasa `bitset` dostarcza funkcje składowe, które pomagają w przeprowadzaniu operacji wstawiania do obiektu `bitset`, ustawiania bądź zerowania jego zawartości, odczytu lub zapisu do strumienia. Klasa oferuje także operatory pomagające w wyświetlaniu zawartości obiektu `bitset` oraz m.in. w przeprowadzaniu logicznych operacji bitowych.

Operatory std::bitset

Operatory poznałeś już w lekcji 12., „Typy operatorów i ich przeciążanie", w której dowiedziałeś się również, że najważniejszą rolą operatorów jest zwiększenie użyteczności klasy. W tabeli 25.1 wymieniono wiele użytecznych operatorów dostarczanych przez klasę `std::bitset`, które bardzo ułatwiają stosowanie klasy. Operatory w tabeli zostały objaśnione za pomocą przykładowego obiektu `bitset` o nazwie `fourBits` wykorzystywanego w listingu 25.1.

Tabela 25.1. Operatory obsługiwane przez klasę std::bitset

Operator	Opis
<<	W strumieniu wyjściowym wstawia tekstową reprezentację sekwencji bitowej. `cout << fourBits;`
>>	Wstawia ciąg tekstowy do obiektu bitset. `"0101" >> fourBits;`
&	Wykonuje bitową operację AND. `bitset <4> result (fourBits1 & fourBits2);`
|	Wykonuje bitową operację OR. `bitset <4> result (fourBits1 | fourBits2);`
^	Wykonuje bitową operację XOR. `bitset <4> result (fourBits1 ^ fourBits2);`
~	Wykonuje bitową operację NOT. `bitset <4> result (~fourBits1);`
>>=	Wykonuje bitowe przesunięcie w prawo. `fourBits >>= (2); // Dwa bity zostają przesunięte w prawo.`
<<=	Wykonuje bitowe przesunięcie w lewo. `fourBits <<= (2); // Dwa bity zostają przesunięte w lewo.`

Tabela 25.1. Operatory obsługiwane przez klasę std::bitset (cd.)

Operator	Opis

`[N]`	Zwraca odniesienie do bitu (N+1) w sekwencji bitowej.
	`fourBits [2] = 0; // Trzeciemu bitowi zostaje ustawiona wartość 0.`
	`bool bNum = fourBits [2]; // Odczytanie trzeciego bitu.`

Oprócz wymienionych, klasa `std::bitset` oferuje także operatory, takie jak `|=`, `&=`, `^=` oraz `~=`, które przeprowadzają operacje bitowe na obiekcie `bitset`.

Metody składowe klasy std::bitset

Bit może przechowywać jeden z dwóch stanów — ustawiony (1) lub wyzerowany (0). Aby pomóc w manipulowaniu zawartością obiektu `bitset`, programista może wykorzystać funkcje składowe wymienione w tabeli 25.2. Zadaniem tych funkcji jest pomoc w trakcie pracy z bitem bądź wszystkimi bitami w obiekcie `bitset`.

Tabela 25.2. Metody składowe oferowane przez klasę std::bitset

Metoda	Opis
`set()`	Wszystkie bity w sekwencji otrzymują wartość 1. `fourBits.set (); // Sekwencja zawiera teraz bity: '1111'.`
`set(N, val = 1)`	Bitowi (N+1) zostaje ustawiona wartość określona przez `val` (domyślnie to 1). `fourBits.set (2, 0); // Ustawienie trzeciemu bitowi // wartości 0.`
`reset()`	Wszystkie bity w sekwencji otrzymują wartość 0. `fourBits.reset (); // Sekwencja zawiera teraz bity: '0000'.`
`reset(N)`	Wyzerowanie bitu znajdującego się w położeniu (N+1). `fourBits.reset (2); // Trzeci bit ma teraz wartość 0.`
`flip()`	Odwraca wartości wszystkich bitów w sekwencji. `fourBits.flip (); // 0101 zmienia się w 1010.`
`size()`	Zwraca liczbę bitów w sekwencji. `size_t nNumBits = fourBits.size (); // Zwraca liczbę 4.`

`count()`	Zwraca liczbę bitów, które zostały ustawione.
	`size_t nNumBitsSet = fourBits.count ();`
	`size_t nNumBitsReset = fourBits.size () – fourBits.count ();`

Sposób użycia wymienionych metod i operatorów został zademonstrowany w listingu 25.2.

Listing 25.2. Przeprowadzanie operacji logicznych za pomocą obiektu bitset

```
 0: #include <bitset>
 1: #include <string>
 2: #include <iostream>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:     bitset <8> inputBits;
 8:     cout << "Podaj ośmiobitową sekwencję: ";
 9:
10:     cin >> inputBits; // Umieszczenie w obiekcie bitset danych wejściowych
       ↪użytkownika.
11:
12:     cout << "Liczba jedynek w podanej sekwencji: " << inputBits.count () <<
       ↪endl;
13:     cout << "Liczba zer w podanej sekwencji: ";
14:     cout << inputBits.size () - inputBits.count () << endl;
15:
16:     bitset <8> inputFlipped (inputBits); // Kopia.
17:     inputFlipped.flip (); // Odwrócenie bitów.
18:
19:     cout << "Odwrócona wersja sekwencji jest następująca: " <<
       ↪inputFlipped << endl;
20:
21:     cout << "Wynik przeprowadzenia operacji AND, OR oraz XOR między
       ↪dwiema podanymi sekwencjami:" << endl;
22:     cout << inputBits << " & " << inputFlipped << " = ";
23:     cout << (inputBits & inputFlipped) << endl; // Bitowe AND.
24:
25:     cout << inputBits << " | " << inputFlipped << " = ";
26:     cout << (inputBits | inputFlipped) << endl; // Bitowe OR.
27:
28:     cout << inputBits << " ^ " << inputFlipped << " = ";
29:     cout << (inputBits ^ inputFlipped) << endl; // Bitowe XOR.
30:
31:     return 0;
32: }
```

Wynik ▼

```
Podaj ośmiobitową sekwencję: 10110101
Liczba jedynek w podanej sekwencji: 5
Liczba zer w podanej sekwencji: 3
Odwrócona wersja sekwencji jest następująca: 01001010
Wynik przeprowadzenia operacji AND, OR oraz XOR między dwiema podanymi
➥sekwencjami:
10110101 & 01001010 = 00000000
10110101 | 01001010 = 11111111
10110101 ^ 01001010 = 11111111
```

Analiza ▼

Powyższy przykład to interaktywny program, który demonstruje nie tylko łatwość, z jaką można przeprowadzać operacje bitowe między dwiema sekwencjami bitowymi za pomocą klasy `std::bitset`, ale również użyteczność jej operatorów strumieni. Operatory >> i << to te, dzięki którym można sekwencję bitową wyświetlić na ekranie i pobrać od użytkownika w formie ciągu tekstowego. Obiekt `inputBits` zawiera sekwencję bitową podaną przez użytkownika w wierszu 10. Użyta w wierszu 12. funkcja `count()` informuje o liczbie jedynek i zer w sekwencji. Jak można zobaczyć w wierszu 14., liczby te są obliczane jako różnica między wartością zwracaną przez funkcję `size()`, która podaje liczbę bitów w obiekcie `bitset`, oraz wartością zwracaną przez funkcję `count()`. Obiekt `inputFlipped` to na początku kopia obiektu `inputBits`, której zawartość jest następnie odwrócona za pomocą funkcji `flip()`, co pokazano w wierszu 17. Po operacji odwrócenia obiekt zawiera sekwencję, której poszczególne bity zostały odwrócone (tzn. jedynki stają się zerami i na odwrót). Pozostała część programu pokazuje wynik przeprowadzenia bitowych operacji `AND`, `OR` i `XOR` na podanych sekwencjach bitowych.

Uwaga

Warto w tym miejscu zwrócić uwagę na względną wadę tej klasy STL, jaką jest brak możliwości dynamicznej zmiany wielkości obiektu. Oznacza to, że obiekt `bitset` może przechowywać jedynie liczbę bitów zdefiniowaną *w trakcie kompilacji*.

Biblioteka STL dostarcza programiście klasę `vector<bool>` (w pewnych implementacjach STL nazywaną również `bit_vector`), która pozwala na obejście wymienionego ograniczenia.

Klasa vector<bool>

Klasa `vector<bool>` to częściowa specjalizacja klasy `std::vector`, która została przeznaczona do przechowywania danych boolowskich. Klasa ta ma możliwość dynamicznej zmiany wielkości obiektu, więc *w trakcie kompilacji* programista nie musi znać liczby opcji boolowskich, które będzie chciał przechowywać w utworzonym obiekcie.

Wskazówka

W celu użycia klasy `std::vector<bool>` w programie trzeba umieścić następujący nagłówek:

```
#include <vector>
```

Ustanowienie klasy vector<bool>

Utworzenie obiektu `vector<bool>` jest podobne do tworzenia wektora, dostępne są pewne wygodne metody przeciążone:

```
vector <bool> vecBool1;
```

Możesz np. utworzyć obiekt `vector` wraz z dziesięcioma wartościami boolowskimi, którym przypisano 1 (tzn. `true`):

```
vector <bool> vecBool2 (10, true);
```

Istnieje również możliwość utworzenia obiektu jako kopii innego:

```
vector <bool> vecBool2Copy (vecBool2);
```

W listingu 25.3 przedstawiono przykładowe ustanowienie klasy `vector<bool>`.

Listing 25.3. Ustanowienie klasy vector<bool>

```
 0: #include <vector>
 1:
 2: int main ()
 3: {
 4:     using namespace std;
 5:
 6:     // Ustanowienie obiektu za pomocą konstruktora domyślnego.
 7:     vector <bool> vecBool1;
 8:
 9:     // Obiekt vector przechowujący dziesięć elementów o wartości true.
10:     vector <bool> vecBool2 (10, true);
11:
12:     // Ustanowienie obiektu jako kopii innego.
```

```
13:     vector <bool> vecBool2Copy (vecBool2);
14:
15:     return 0;
16: }
```

Analiza ▼

W powyższym przykładzie zaprezentowano niektóre sposoby ustanowienia obiektu `vector<bool>`. W wierszu 7. został użyty konstruktor domyślny. Z kolei w wierszu 10. zademonstrowano utworzenie obiektu, który został zainicjalizowany wraz z dziesięcioma opcjami boolowskimi, każda o wartości `true`. Natomiast w wierszu 13. pokazano, jak obiekt `vector<bool>` można utworzyć jako kopię innego obiektu tego typu.

Używanie klasy vector<bool>

Klasa `vector<bool>` zawiera funkcję `flip()`, która odwraca stan wartości boolowskiej w sekwencji; działa ona podobnie do funkcji `bitset<>::flip()`.

Pomijając tę funkcję, klasa jest całkiem podobna do klasy `std::vector` pod tym względem, że można np. umieszczać opcje w sekwencji za pomocą funkcji `push_back()`. Sposób używania tej klasy przedstawiono bardziej szczegółowo w listingu 25.4.

Listing 25.4. Używanie klasy vector<bool>

```
 0: #include <vector>
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main ()
 5: {
 6:     vector <bool> vecBoolFlags (3); // Utworzenie obiektu przechowującego
        ➥trzy opcje.
 7:     vecBoolFlags [0] = true;
 8:     vecBoolFlags [1] = true;
 9:     vecBoolFlags [2] = false;
10:
11:     vecBoolFlags.push_back (true); // Wstawienie czwartej opcji na końcu.
12:
13:     cout << "Zawartość obiektu vector jest następująca: " << endl;
14:     for (size_t nIndex = 0; nIndex < vecBoolFlags.size (); ++ nIndex)
15:         cout << vecBoolFlags [nIndex] << ' ';
16:
```

```
17:     cout <<  endl;
18:     vecBoolFlags.flip ();
19:
20:     cout << "Odwrócona zawartość obiektu vector jest następująca: " <<
        ↪endl;
21:     for (size_t nIndex = 0; nIndex < vecBoolFlags.size (); ++ nIndex)
22:         cout << vecBoolFlags [nIndex] << ' ';
23:
24:     cout << endl;
25:
26:     return 0;
27: }
```

Wynik ▼

```
Zawartość obiektu vector jest następująca:
{1 1 0 1 }

Odwrócona zawartość obiektu vector jest następująca:
{0 0 1 0 }
```

Analiza ▼

W powyższym przykładzie dostęp do opcji boolowskich w obiekcie `vector` odbywa się za pomocą składni operatora indeksowania `[]`, jak pokazano w wierszach od 7. do 9. Jest to podobne do sposobu używanego w zwykłym obiekcie `vector`. Użyta w wierszu 18. funkcja `flip()` powoduje odwrócenie zawartości bitów, czyli konwersję zer na jedynki i na odwrót. Zwróć uwagę na zastosowanie metody `push_back()` w wierszu 11. Wprawdzie w wierszu 6. obiekt `vecBoolFlags` zainicjalizowano do przechowywania trzech opcji boolowskich, to jego wielkość można zmienić dynamicznie, co pokazano w wierszu 11., w którym dodano czwartą opcję. Dodanie liczby opcji większej, niż zdefiniowana w trakcie kompilacji, nie jest możliwe w klasie `std::bitset`.

Podsumowanie

W tej lekcji poznałeś najbardziej efektywne narzędzie służące do obsługi sekwencji bitowych oraz opcji bitowych, czyli klasę `std::bitset`. Dowiedziałeś się również o istnieniu klasy `vector<bool>`, która pozwala na przechowywanie opcji boolowskich — w trakcie kompilacji programu nie musisz znać liczby tych opcji.

Pytania i odpowiedzi

Pytanie: W sytuacji, w której można wykorzystać zarówno klasę `std::bitset`, jak i `vector<bool>`, której z wymienionych klas lepiej użyć do przechowywania opcji binarnych?

Odpowiedź: Obiekt `bitset` jest lepiej dostosowany do spełnienia tego rodzaju wymagań.

Pytanie: Mam obiekt `std::bitset` o nazwie `myBitSeq` zawierający określoną liczbę bitów. W jaki sposób mogę obliczyć liczbę bitów przechowujących wartość `0` (czyli `false`)?

Odpowiedź: Funkcja `bitset::count()` podaje liczbę bitów, które mają wartość 1. Po odjęciu tej liczby od wartości zwróconej przez funkcję `bitset::size()`, podającej całkowitą liczbę przechowywanych bitów, otrzymasz liczbę bitów, które przechowują wartość 0.

Pytanie: Czy mogę wykorzystać iteratory w celu uzyskania dostępu do poszczególnych elementów obiektu `vector<bool>`?

Odpowiedź: Tak. Ponieważ `vector<bool>` to częściowa specjalizacja klasy `std::vector`, iteratory są obsługiwane.

Pytanie: Czy w trakcie kompilacji mogę określić liczbę elementów, które mają być przechowywane w obiekcie `vector<bool>`?

Odpowiedź: Tak, odbywa się to poprzez podanie liczby w przeciążonym konstruktorze lub użycie funkcji `vector<bool>::resize` w egzemplarzu klasy.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Czy obiekt `bitset` może rozszerzyć swój wewnętrzny bufor, tak aby przechowywać zmienną liczbę elementów?
2. Dlaczego obiekt `bitset` nie jest klasyfikowany jako klasa kontenera STL?
3. Czy użyjesz obiektu `std::vector` do przechowywania stałej liczby bitów, która jest znana już w trakcie kompilacji?

Ćwiczenia

1. Utwórz klasę `bitset` zawierającą cztery bity. Zainicjalizuj ją wraz z liczbą, wyświetl wynik, a następnie dodaj do innego obiektu `bitset`. (Haczyk: obiekty `bitset` nie pozwalają na użycie `bitsetA = bitsetX + bitsetY`).
2. Zademonstruj, w jaki sposób możesz odwrócić (tzn. zamienić) bity w obiekcie `bitset`.

Część V

Zaawansowane koncepcje C++

Lekcja 26

Sprytne wskaźniki

Podczas zarządzania pamięcią w stercie (lub stosie) programiści C++ niekoniecznie muszą używać prostych typów wskaźników. Zamiast nich mogą skorzystać ze sprytniejszego rozwiązania.

Z tej lekcji dowiesz się:

- **dowiesz się, czym są sprytne wskaźniki oraz dlaczego możesz ich potrzebować,**
- **dowiesz się, jak są implementowane sprytne wskaźniki,**
- **poznasz różne typy sprytnych wskaźników,**
- **dowiesz się, dlaczego nie powinieneś używać przestarzałej klasy std::auto_ptr,**
- **poznasz klasę sprytnego wskaźnika std::unique_ptr wprowadzoną w C++11,**
- **dowiesz się, jakie są popularne biblioteki sprytnych wskaźników.**

Czym są sprytne wskaźniki?

Ujmując najprościej, *sprytny wskaźnik* w C++ to klasa z przeciążonymi operatorami, która zachowuje się, jak wskaźnik konwencjonalny, ale dostarcza wartość dodatkową, zapewniając na czas właściwe zwolnienie dynamicznie zaalokowanych danych. Ponadto może zawierać implementację strategii zarządzania cyklem życiowym obiektu.

Na czym polega problem związany z używaniem wskaźników konwencjonalnych?

W przeciwieństwie do innych nowoczesnych języków programowania, C++ zapewnia programiście pełną elastyczność w zakresie alokacji pamięci, zwalniania jej oraz zarządzania nią. Niestety, elastyczność ta wiąże się także z pewnymi wadami. Z jednej strony powoduje, że C++ to język programowania o potężnych możliwościach. Jednak z drugiej strony oznacza to, że programista może doprowadzić do powstania problemów związanych z pamięcią, np. z wyciekami pamięci, kiedy obiekty alokowane dynamicznie nie są prawidłowo zwalniane.

Przykładowo:

```
CData *pData = mObject.GetData ();
/*
Pytania: Czy obiekt wskazywany przez wskaźnik pData został zaalokowany dynamicznie?
Kto przeprowadzi operację zwalniania, gdy zajdzie taka potrzeba: wywołujący czy wywoływany?
Odpowiedź: Nie wiadomo!
*/
pData->Display ();
```

Analizując powyższy fragment kodu, nie można z całą pewnością stwierdzić, do którego miejsca w pamięci prowadzi wskaźnik `pData`.

- Czy pamięć dla obiektu została zaalokowana w stercie (`heap`) i dlatego powinna zostać *zwolniona*?
- Czy za proces *zwalniania pamięci* odpowiada wywołujący?
- Czy obiekt zostanie automatycznie zniszczony przez *destruktor* obiektu?

Chociaż pewne niejasności można częściowo wyeliminować poprzez wstawienie komentarzy i wymuszenie stosowania odpowiednich praktyk programowania, mechanizmy te są zbyt słabe, aby efektywnie pomagać

w unikaniu wszelkich błędów spowodowanych przez nadużycie dynamicznie alokowanych danych i wskaźników.

W jaki sposób sprytne wskaźniki mogą pomóc?

Znając problemy związane z używaniem wskaźników konwencjonalnych oraz konwencjonalnych technik zarządzania pamięcią, należy zauważyć, że programista C++ nie jest zmuszony do ich stosowania, kiedy trzeba zarządzać danymi w stercie i na stosie. Programista może zdecydować się na wybór znacznie sprytniejszego sposobu alokacji i zarządzania danymi dynamicznymi poprzez zaadaptowanie w tworzonych programach sprytnych wskaźników:

```
smart_pointer<CData> spData = mObject.GetData ();
// Użycie sprytnego wskaźnika w taki sam sposób jak wskaźnika konwencjonalnego!
spData->Display ();
(*spData).Display ();
// Nie należy się przejmować kwestią zwalniania pamięci
// (destruktor sprytnego wskaźnika zajmie się tym za programistę).
```

Tak oto sprytne wskaźniki zachowują się w taki sam sposób jak wskaźniki konwencjonalne (od tej chwili będziemy je nazywać również *zwykłymi wskaźnikami*), ale za pomocą *przeciążonych operatorów* i *destruktorów* dostarczają użyteczne funkcje w celu zagwarantowania, że zaalokowane dane będą usunięte w samą porę.

W jaki sposób są implementowane sprytne wskaźniki?

To pytanie na chwilę może być uproszczone do takiego: W jaki sposób sprytny wskaźnik `spData` będzie funkcjonował jak wskaźnik konwencjonalny?. Oto odpowiedź: Klasa sprytnego wskaźnika przeciąża operatory * (operator dereferencji) i -> (operator wyboru elementu składowego), aby pozwolić programiście na użycie ich jako wskaźników konwencjonalnych. Przeciążanie operatorów zostało już omówione w lekcji 12., „Typy operatorów i ich przeciążanie".

Aby ponadto pozwolić programiście na zarządzanie wybranym typem w stercie, niemal wszystkie dobre klasy sprytnych wskaźników są klasami wzorców, które zawierają ogólną implementację ich funkcji. Jako wzorce są

na tyle uniwersalne, że mogą być specjalizowane w celu zarządzania obiektem dowolnie wybranego typu.

Przykładowa implementacja prostej klasy sprytnego wskaźnika została przedstawiona w listingu 26.1.

Listing 26.1. Minimalna ilość istotnych komponentów składających się na klasę sprytnego wskaźnika

```
 0: template <typename T>
 1: class smart_pointer
 2: {
 3: private:
 4:     T* m_pRawPointer;
 5: public:
 6:     smart_pointer (T* pData) : m_pRawPointer (pData) {}     // Konstruktor.
 7:     ~smart_pointer () {delete pData;};                      // Destruktor.
 8:
 9:     // Konstruktor kopiujący.
10:     smart_pointer (const smart_pointer & anotherSP);
11:     // Operator przypisania.
12:     smart_pointer& operator= (const smart_pointer& anotherSP);
13:
14:     T& operator* () const         // Operator dereferencji.
15:     {
16:         return *(m_pRawPointer);
17:     }
18:
19:     T* operator-> () const         // Operator wyboru elementu składowego.
20:     {
21:         return m_pRawPointer;
22:     }
23: };
```

Analiza ▼

Powyższa klasa sprytnego wskaźnika pokazuje implementację dwóch operatorów: dereferencji (*) oraz wyboru elementu składowego (->), co przedstawiono w wierszach od 14. do 17. oraz od 19. do 22. Pomagają one klasie w funkcjonowaniu jak „wskaźnik”, w konwencjonalnym znaczeniu tego słowa. Jeżeli przykładowo będziesz miał klasę Tuna, będziesz mógł użyć sprytnego wskaźnika względem obiektu typu Tuna w następujący sposób:

```
smart_pointer <Tuna> pSmartTuna (new Tuna);
pSmartTuna->Swim();
```

```
// Alternatywnie:
(*pSmartTuna).Swim();
```

Klasa `smart_pointer` wciąż ani nie pokazuje, ani nie implementuje żadnej funkcji, która powodowałaby, że ta klasa wskaźnika stanie się sprytna i będzie pokazywała swoją przewagę nad użyciem wskaźnika konwencjonalnego. Jak można zobaczyć w wierszu 6., *konstruktor* akceptuje wskaźnik zapisany w klasie sprytnego wskaźnika jako wewnętrzny obiekt wskaźnika. Z kolei *destruktor* zwalnia pamięć zaalokowaną przez ten wskaźnik, a więc przeprowadza automatyczne zwalnianie pamięci.

Uwaga

Implementacje, dzięki którym sprytny wskaźnik naprawdę staje się „sprytny", to implementacje *konstruktora kopiującego*, *operatora przypisania* oraz *destruktora*. Definiują one zachowanie obiektu sprytnego wskaźnika, kiedy jest przekazywany pomiędzy funkcjami, przypisywany bądź gdy wykracza poza zakres (tzn. jest usuwany, podobnie jak każdy inny obiekt klasy). Tak więc przed zapoznaniem się z pełną implementacją sprytnego wskaźnika trzeba zrozumieć pewne typy sprytnych wskaźników.

Typy sprytnych wskaźników

Zarządzanie zasobami pamięci (tzn. kiedy zaimplementowany został własnościowy model) jest czynnikiem wyróżniającym klasę sprytnego wskaźnika. W trakcie operacji kopiowania i przypisywania sprytne wskaźniki decydują o tym, co zostanie zrobione z zasobami. Najprostsza implementacja bardzo często wiąże się z problemami dotyczącymi wydajności, podczas gdy najszybsze mogą nie być dopasowane do wszystkich aplikacji. Ogólnie rzecz biorąc, do programisty należy zrozumienie sposobu działania sprytnych wskaźników przed podjęciem decyzji o ich zastosowaniu w tworzonej aplikacji.

Klasyfikacja sprytnych wskaźników to w rzeczywistości klasyfikacja stosowanych przez nie strategii zarządzania zasobami pamięci. Mamy więc:

- kopiowanie głębokie,
- kopiowanie przy zapisie (ang. *copy-on-write*, COW),
- licznik odniesień,
- wskaźniki powiązane z licznikiem odniesień,
- kopiowanie destrukcyjne.

Przed analizą sprytnego wskaźnika dostarczanego przez bibliotekę standardową C++ (`std::unique_ptr`) zapoznamy się z wszystkimi wymienionymi powyżej strategiami.

Kopiowanie głębokie

W sprytnym wskaźniku implementującym *kopiowanie głębokie* każdy egzemplarz sprytnego wskaźnika przechowuje pełną kopię *zarządzanego* obiektu. Zawsze wtedy, gdy sprytny wskaźnik będzie kopiowany, obiekt, do którego prowadzi ten wskaźnik, również zostanie skopiowany (stąd nazwa kopiowanie głębokie). Kiedy sprytny wskaźnik wykracza poza zakres, zwalnia pamięć, do której prowadzi (za pomocą destruktora).

Wprawdzie wydaje się, że sprytny wskaźnik bazujący na kopiowaniu głębokim nie oferuje żadnej dodatkowej wartości, poza przekazywaniem obiektów poprzez wartość, jego zalety stają się widoczne podczas pracy z obiektami polimorficznymi. Pokazano to w poniższym fragmencie kodu, w którym można uniknąć *segmentowania*:

```
// Przykład segmentowania podczas przekazywania obiektów polimorficznych poprzez wartość.
// Fish to klasa bazowa dla Tuna oraz Carp, natomiast Fish::Swim() to metoda wirtualna.
void MakeFishSwim (Fish aFish)      // Zwróć uwagę na typ parametru.
{
   aFish.Swim(); // Funkcja wirtualna.
}
// ... Dowolna funkcja.
Carp freshWaterFish;
MakeFishSwim (freshWaterFish); // Unikamy segmentowania.
// Segmentowanie: tylko część Fish obiektu Carp będzie wysłana do MakeFishSwim().
Tuna marineFish;
MakeFishSwim(marineFish);      // Segmentowanie raz jeszcze.
```

Kwestie związane z segmentowaniem zostają rozwiązane, kiedy programista wybiera sprytny wskaźnik bazujący na kopiowaniu głębokim, tak jak przedstawiono w listingu 26.2.

Listing 26.2. Używanie sprytnego wskaźnika bazującego na kopiowaniu głębokim w celu przekazywania obiektów polimorficznych poprzez ich typ bazowy

```
template <typename T>
class deepcopy_smart_pointer
{
private:
    T* m_pObject;
```

```
 5: public:
 6:      // ... Inne funkcje.
 7:
 8:      // Konstruktor kopiujący wskaźnika deepcopy.
 9:      deepcopy_smart_pointer (const deepcopy_smart_pointer& source)
10:      {
11:           // Użycie wirtualnej funkcji klonującej zdefiniowanej w klasie pochodnej
                 ➥w celu pobrania pełnej kopii obiektu.
12:           m_pObject = source->Clone ();
13:      }
14:
15:      // Kopiujący operator przypisania.
16:      deepcopy_smart_pointer& operator= (const deepcopy_smart_pointer&
         ➥source)
17:      {
18:         if (m_pObject)
19:            delete m_pObject;
20:
21:         m_pObject = source->Clone ();
22:      }
23:
24: };
```

Analiza ▼

Jak widać, w wierszach od 9. do 13. sprytny wskaźnik `deepcopy_smart_pointer` implementuje *konstruktor kopiujący* pozwalający na przeprowadzenie operacji kopiowania głębokiego obiektu polimorficznego za pomocą funkcji `Clone()`, którą ten obiekt musi zaimplementować. Natomiast w wierszach od 16. do 22. został zaimplementowany kopiujący operator przypisania. W celu zachowania prostoty omawianego przykładu przyjęto założenie, że funkcja wirtualna implementowana przez klasę bazową `Fish` nosi nazwę `Clone()`. Zazwyczaj sprytne wskaźniki implementujące model kopiowania głębokiego będą miały tę funkcję dostarczoną albo jako parametr wzorca, albo jako obiekt funkcji.

Stąd, kiedy sprytny wskaźnik jest przekazywany jako wskaźnik klasie bazowej typu `Fish`:

```
deepcopy_smart_ptr <Carp> freshWaterFish (new Carp);
MakeFishSwim (freshWaterFish);      // Brak problemów z segmentowaniem.
```

zaimplementowany w konstruktorze sprytnego wskaźnika model kopiowania głębokiego zostaje wykonany w celu zagwarantowania, że przekazywany obiekt nie będzie poddany segmentowaniu. Odbywa się to nawet wtedy, gdy funkcja docelowa `MakeFishSwim()` wymaga tylko części obiektu.

Wadą mechanizmu bazującego na kopiowaniu głębokim jest spadek wydajności. W przypadku niektórych aplikacji to nie będzie miało znaczenia, natomiast w wielu innych ta wada będzie skutecznie powstrzymywała programistów przed używaniem sprytnych wskaźników. Wówczas funkcjom, takim jak `MakeFishSwim()`, są po prostu przekazywane wskaźniki typu bazowego (wskaźnik konwencjonalny, np. `Fish *`). Inne typy wskaźników próbują na różne sposoby poradzić sobie z problemem spadku wydajności.

Mechanizm kopiowania przy zapisie (COW)

Mechanizm *kopiowania przy zapisie* (ang. *Copy on Write*, popularnie nazywany COW) próbuje zoptymalizować wydajność wskaźników modelu kopiowania głębokiego poprzez współdzielenie wskaźników, aż do pierwszej próby zapisu obiektu. W trakcie pierwszej próby wywołania funkcji innej niż typu `const`, wskaźnik COW zazwyczaj tworzy kopię obiektu, względem którego następuje wywołanie funkcji innej niż `const`. Natomiast pozostałe egzemplarze wskaźnika kontynuują współdzielenie obiektu źródłowego.

Wskaźniki COW mają wielu fanów. Dla programistów używających wskaźników COW implementacja operatorów * oraz -> w tworzonych funkcjach `const` i innych niż `const` jest podstawowym kluczem funkcjonalności wskaźników COW. Ten drugi powoduje utworzenie kopii.

Sedno tkwi w tym, aby po wybraniu implementacji wskaźnika działającego według filozofii COW upewnić się o zrozumieniu szczegółów implementacji przed rozpoczęciem korzystania z tej implementacji. W przeciwnym razie można znaleźć się w sytuacji, kiedy kopii jest albo zbyt mało, albo zbyt wiele.

Sprytne wskaźniki zliczania odniesień

Ogólnie rzecz biorąc, zliczanie odniesień to mechanizm, który przechowuje liczbę bieżących użytkowników obiektu. Kiedy licznik spadnie do zera, obiekt zostaje usunięty, a zajmowana przez niego pamięć zwolniona. Tak więc licznik odniesień to bardzo dobry mechanizm współdzielenia obiektów bez konieczności ich kopiowania. Jeżeli kiedykolwiek pracowałeś z technologią firmy Microsoft o nazwie COM, przynajmniej raz musiałeś się spotkać z koncepcją licznika odniesień[1].

[1] Ten mechanizm jest stosowany również w języku Objective-C wykorzystywanym do tworzenia oprogramowania na platformach iOS i OS X — *przyp. tłum.*

Podczas kopiowania tego rodzaju sprytnego wskaźnika trzeba inkrementować licznik odniesień kopiowanego obiektu. Istnieją co najmniej dwa popularne sposoby obsługi licznika odniesień:

- licznik odniesień obsługiwany przez obiekt;
- licznik odniesień obsługiwany przez klasę wskaźnika we współdzielonym obiekcie.

Pierwszy z wymienionych sposobów nosi nazwę *inwazyjnego licznika odniesień*, ponieważ obiekt musi zostać zmodyfikowany poprzez obsługę, inkrementację i dostarczanie licznika odniesień każdej klasie sprytnego wskaźnika, która nim zarządza. Nawiasem mówiąc, takie podejście zostało zastosowane w technologii COM. Natomiast drugi z wymienionych sposobów to mechanizm, w którym klasa sprytnego wskaźnika może przechowywać na stosie licznik odniesień (np. dynamicznie alokowaną liczbę całkowitą). W takim przypadku podczas kopiowania konstruktor kopiujący będzie inkrementował tę wartość.

A więc mechanizm licznika odniesień jest istotny dla programisty pracującego ze sprytnymi wskaźnikami jedynie podczas używania obiektu. Rozwiązanie polegające na użyciu sprytnego wskaźnika zarządzającego obiektem i zwykłego wskaźnika prowadzącego do tego obiektu jest złym pomysłem, ponieważ sprytny wskaźnik (sprytnie) usunie obiekt i zwolni zajmowaną przez niego pamięć, kiedy obsługiwany przez niego licznik odniesień spadnie do zera. Jednak zwykły wskaźnik nadal będzie wskazywał miejsce w pamięci, które już nie należy do danej aplikacji. Podobnie, licznik odniesień może spowodować powstanie dziwnych problemów w pewnych sytuacjach: dwa obiekty przechowujące wskaźniki do siebie nawzajem nigdy nie zostaną usunięte, ponieważ wzajemna zależność powoduje, że oba obiekty utrzymują licznik odniesień o wartości minimum 1.

Sprytne wskaźniki powiązane z licznikiem odniesień

Sprytne wskaźniki *powiązane z licznikiem odniesień* to takie, które nie zliczają aktywnie liczby użytkowników korzystających z danego obiektu. Zamiast tego wskaźniki te muszą po prostu wiedzieć, kiedy wartość licznika spadnie do zera. Wtedy następuje usunięcie obiektu i zwolnienie zajmowanej przez niego pamięci.

Wskaźniki takie są nazywane *powiązanymi z licznikiem odniesień*, ponieważ ich implementacja bazuje na liście dwukierunkowej. Kiedy następuje utworzenie

nowego sprytnego wskaźnika poprzez skopiowanie istniejącego, zostaje on dołączony do listy. Gdy sprytny wskaźnik wykracza poza zakres, zostaje zniszczony, a destruktor usuwa ten sprytny wskaźnik z listy. Podobnie jak w przypadku wskaźników licznika odniesień, także i te dotyka problem wzajemnej zależności wskaźników.

Kopiowanie destrukcyjne

Kopiowanie destrukcyjne to mechanizm, w którym sprytny wskaźnik podczas kopiowania przenosi pełne prawa własności obsługiwanego obiektu na obiekt docelowy, a następnie zeruje się:

```
destructive_copy_smartptr <SomeClass> pSmartPtr (new SomeClass ());
SomeFunc (pSmartPtr);      // Prawa własności przekazane na SomeFunc.
// W wywołującym nie używamy już więcej wskaźnika pSmartPtr!
```

Wprawdzie ten mechanizm nie jest całkiem intuicyjny w użyciu, jednak zaletą oferowaną przez sprytne wskaźniki modelu kopiowania destrukcyjnego jest fakt, że w każdej chwili tylko jeden aktywny wskaźnik prowadzi do danego obiektu. Jest to więc dobry mechanizm dla wskaźników przekazywanych z wywołanej funkcji, a także w sytuacjach, kiedy ich „destrukcyjna" natura stanowi zaletę.

Jak można się przekonać na podstawie listingu 26.3, implementacja sprytnych wskaźników modelu kopiowania destrukcyjnego odbiega od standardu i zalecanych technik programowania w C++.

Ostrzeżenie

> Dotychczas klasa `std::auto_ptr` to najpopularniejszy (lub cieszący się złą sławą, zależy jak na to spojrzysz) wskaźnik stosujący model kopiowania destrukcyjnego. Taki sprytny wskaźnik staje się bezużyteczny po przekazaniu funkcji lub skopiowaniu do innego.
>
> W standardzie C++11 klasa `std::auto_ptr` jest uznawana za przestarzałą. Zamiast niej należy używać klasy `std::unique_ptr`. Wskaźnika `std::unique_ptr` nie można przekazać przez wartość ze względu na prywatny konstruktor kopiujący i kopiujący operator przypisania — wskaźnik tej klasy może być przekazany tylko przez referencję.

Listing 26.3. Przykład sprytnego wskaźnika modelu kopiowania destrukcyjnego

```
0: template <typename T>
1: class destructivecopy_pointer
2: {
```

```
 3: private:
 4:     T* pObject;
 5: public:
 6:     destructivecopy_pointer(T* pInput):pObject(pInput) {}
 7:     ~destructivecopy_pointer() { delete pObject; }
 8:
 9:     // Konstruktor kopiujący.
10:     destructivecopy_pointer(destructivecopy_pointer& source)
11:     {
12:         // Przeniesienie praw własności na kopię.
13:         pObject = source.pObject;
14:
15:         // Usunięcie źródła.
16:         source.pObject = 0;
17:     }
18:
19:     // Kopiujący operator przypisania.
20:     destructivecopy_pointer& operator= (destructivecopy_pointer& rhs)
21:     {
22:         if (pObject != source.pObject)
23:         {
24:             delete pObject;
25:             pObject = source.pObject;
26:            source.pObject = 0;
27:        }
28:     }
29: };
30:
31: int main()
32: {
33:    destructivecopy_pointer<int> pNumber (new int);
34:    destructivecopy_pointer<int> pCopy = pNumber;
35:
36:    // Wskaźnik pNumber jest teraz nieprawidłowy.
37:    return 0;
38: }
```

Analiza ▼

W listingu 26.3 przedstawiono najważniejszą część implementacji sprytnego wskaźnika bazującego na modelu kopiowania destrukcyjnego. W wierszach od 10. do 17. oraz od 20. do 28. znajduje się konstruktor kopiujący oraz kopiujący operator przypisania. W kodzie widać, że wymienione funkcje podczas tworzenia kopii w rzeczywistości unieważniają źródło. Oznacza to, że konstruktor kopiujący po operacji kopiowania ustawia wartość zero na liczniku wskaźnika źródłowego, co wyjaśnia nazwę kopiowania destrukcyjnego.

Operator przypisania działa w taki sam sposób. Dlatego też wskaźnik `pNumber` jest unieważniany w wierszu 34. po przypisaniu do innego wskaźnika. Takie zachowanie jest sprzeczne z operacją przypisania.

Ostrzeżenie

Konstruktor kopiujący i kopiujący operator przypisania mają znaczenie krytyczne w implementacji pokazanego w listingu 26.3 sprytnego wskaźnika modelu kopiowania destrukcyjnego; są również przedmiotem ogromnej krytyki. Możesz zauważyć, że w przeciwieństwie do większości implementacji C++, ta klasa sprytnego wskaźnika nie ma konstruktora kopiującego i kopiującego operatora przypisywania, które akceptowałyby odniesienia typu `const`. To wynika z oczywistego powodu — unieważnienia źródła po jego skopiowaniu. Takie rozwiązanie stanowi odstępstwo od tradycyjnej semantyki konstruktora kopiującego i operatora przypisania. Niewielu programistów spodziewa się zniszczenia źródła kopii lub źródła przypisania po operacji kopiowania lub przypisania. Niszczenie źródła przez ten sprytny wskaźnik powoduje, że nie nadaje się on do używania w kontenerach STL, takich jak `std::vector`, oraz innych dynamicznych kolekcjach klas, które programista może stosować. Wymienione kontenery muszą wewnętrznie kopiować zawartość, co doprowadziłoby do unieważnienia wskaźników.

Tak więc z wymienionych powodów wielu programistów unikających stosowania sprytnych wskaźników modelu kopiowania destrukcyjnego uważa je za plagę.

Wskazówka

Wskaźnik `auto_ptr` był obsługiwanym dotąd przez standard C++ sprytnym wskaźnikiem stosującym kopiowanie destrukcyjne. Wreszcie, w standardzie C++11 został określony jako przestarzały i zamiast niego powinieneś używać `std::unique_ptr`.

C++11

Używanie klasy std::unique_ptr

Obiekt `std::unique_ptr` to wprowadzony w standardzie C++11 sprytny wskaźnik, który jest nieco inny od `auto_ptr`, ponieważ nie pozwala na kopiowanie lub przypisanie.

Wskazówka

W celu użycia obiektu `std::unique_ptr` trzeba najpierw dołączyć odpowiedni nagłówek:

```
#include <memory>
```

Obiekt `unique_ptr` to prosty obiekt sprytnego wskaźnika, podobny do przedstawionego w listingu 26.1, ale zawierający prywatny konstruktor kopiujący i operator przypisania. To uniemożliwia kopiowanie, np. przez przekazanie wskaźnika jako argumentu funkcji lub przez operację przypisania. Przykład użycia `unique_ptr` zaprezentowano w listingu 26.4.

Listing 26.4. Użycie obiektu std::unique_ptr

```
0: #include <iostream>
1: #include <memory>   // Ten nagłówek jest wymagany do użycia std::unique_ptr.
2: using namespace std;
3:
4: class Fish
5: {
6: public:
7:     Fish() {cout << "Fish: utworzono!" << endl;}
8:     ~Fish() {cout << "Fish: zniszczono!" << endl;}
9:
10:     void Swim() const {cout << "Ryba pływa w wodzie" << endl;}
11: };
12:
13: void MakeFishSwim(const unique_ptr<Fish>& inFish)
14: {
15:    inFish->Swim();
16: }
17:
18: int main()
19: {
20:    unique_ptr<Fish> smartFish (new Fish);
21:
22:    smartFish->Swim();
23:    MakeFishSwim(smartFish); // OK, MakeFishSwim akceptuje referencje.
24:
25:    unique_ptr<Fish> copySmartFish;
26:    // copySmartFish = smartFish; // Błąd: operator= jest prywatny.
27:
28:    return 0;
29: }
```

Wynik ▼

```
Fish: utworzono!
Ryba pływa w wodzie
Ryba pływa w wodzie
Fish: zniszczono!
```

Analiza▼

Jak widać w wyświetlonych danych wyjściowych, program wykonuje sekwencję tworzenia i usuwania obiektu. Widać również, że chociaż obiekt wskazywany przez `smartFish` był, zgodnie z oczekiwaniami, utworzony w funkcji `main()`, został usunięty (i to automatycznie) nawet bez konieczności wywołania operatora `delete`. Takie jest zachowanie obiektu `unique_ptr`, gdy wskaźnik wykraczający poza zakres powoduje również usunięcie (za pomocą destruktora) obiektu, do którego się odnosi. Zwróć uwagę na możliwość przekazania w wierszu 23. `smartFish` jako argumentu `MakeFishSwim()`. To nie jest krok kopiowania, ponieważ `MakeFishSwim()` akceptuje parametr przez referencję, jak pokazano w wierszu 13. Jeżeli usuniesz symbol referencji w wierszu 13., natychmiast otrzymasz błąd kompilacji wywołany przez prywatny konstruktor kopiujący. Podobnie, przypisanie obiektu `unique_ptr` innemu, jak przedstawiono w wierszu 26. jest niedozwolone ze względu na istnienie prywatnego operatora przypisania.

Podsumowując, możemy stwierdzić, że klasa `unique_ptr` jest bezpieczniejsza niż `auto_ptr` (która w standardzie C++11 została uznana za przestarzałą), ponieważ nie powoduje unieważnienia źródła wskaźnika podczas operacji kopiowania lub przypisania. Ponadto upraszcza zarządzanie pamięcią, bo w trakcie niszczenia obiektu zwalnia zajmowaną przez niego pamięć.

Popularne biblioteki sprytnych wskaźników

Jest oczywiste, że wersja sprytnego wskaźnika dostarczana wraz z biblioteką standardową C++ nie spełnia wszystkich wymagań programisty. Z tego powodu powstało wiele innych bibliotek sprytnych wskaźników.

Boost (*http://www.boost.org/*) wśród wielu innych użytecznych klas narzędziowych dostarcza również doskonale przetestowane i świetnie udokumentowane klasy sprytnych wskaźników. Więcej informacji na temat sprytnych wskaźników Boost oraz łącze pozwalające na pobranie biblioteki znajdziesz na stronie: *http://www.boost.org/doc/libs/1_54_0/libs/smart_ptr/smart_ptr.htm*.

Programiści zajmujący się tworzeniem aplikacji COM dla platformy Windows powinni rozpocząć korzystanie z efektywnych klas sprytnych wskaźników struktury szkieletowej ATL. Przykładowe klasy to m.in. `CComPtr` i `CComQIPtr` pozwalające na zarządzanie obiektami COM zamiast stosowania wskaźników zwykłego interfejsu.

Podsumowanie

W tej lekcji dowiedziałeś się, jak prawidłowo używać sprytnych wskaźników, które mogą pomóc w tworzeniu kodu wykorzystującego wskaźniki. Podczas tworzenia takiego kodu programista jest zwolniony z samodzielnego zajmowania się alokacją pamięci i kwestiami związanymi z prawami własności do obiektów. Poznałeś również różne typy sprytnych wskaźników i wiesz, jak ważne jest zrozumienie sposobu zachowania klasy sprytnych wskaźników przed wykorzystaniem jej we własnej aplikacji. Na koniec dowiedziałeś się, że nie powinieneś używać sprytnego wskaźnika `std::auto_ptr`, ponieważ unieważnia on źródło w trakcie operacji kopiowania lub przypisania. Wiesz także, że nowsza klasa sprytnego wskaźnika wprowadzona w standardzie C++ to `std::unique_ptr`.

Pytania i odpowiedzi

Pytanie: Potrzebuję obiektu `vector` przechowującego wskaźniki. Czy jako typ elementów przechowywanych w obiekcie `vector` powinienem wybrać `auto_ptr`?

Odpowiedź: Nie, nigdy nie powinieneś używać klasy `std::auto_ptr`, ponieważ jest uznawana za przestarzałą. Pojedyncza operacja kopiowania lub przypisania elementu w obiekcie `vector` spowoduje unieważnienie danego elementu.

Pytanie: Które dwa operatory klasa zawsze musi zawierać, aby była nazywana klasą sprytnych wskaźników?

Odpowiedź: Operatory * i `->`. Pomagają one w używaniu obiektów klasy wraz z semantyką zwykłych wskaźników.

Pytanie: Mam aplikację, w której `Class1` i `Class2` przechowują elementy składowe będące obiektami innego typu. Czy w takiej sytuacji powinienem używać wskaźnika licznika odniesień?

Odpowiedź: Prawdopodobnie nie będziesz mógł, ponieważ wspólna zależność uniemożliwi licznikowi odniesień zejście do zera. W efekcie obiekty dwóch klas na stałe pozostaną w stercie.

Pytanie: Ile sprytnych wskaźników jest na świecie?

Odpowiedź: Tysiące. Nie, może nawet miliony. Powinieneś używać tylko tych sprytnych wskaźników, które mają doskonale udokumentowany sposób działania oraz pochodzą z zaufanych źródeł, takich jak np. Boost.

Pytanie: Klasa ciągu tekstowego również dynamicznie zarządza tablicami znaków w stercie. Czy z tego powodu klasę ciągu tekstowego można nazwać sprytnym wskaźnikiem?

Odpowiedź: Nie, nie można. Tego rodzaju klasy zwykle nie implementują operatorów * i ->, a przez to nie są klasyfikowane jako klasy sprytnych wskaźników.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Gdzie zajrzysz w pierwszej kolejności przed rozpoczęciem tworzenia własnego sprytnego wskaźnika dla aplikacji?
2. Czy sprytny wskaźnik może spowodować znaczące spowolnienie działania Twojej aplikacji?
3. Gdzie sprytne wskaźniki licznika odniesień będą przechowywały dane licznika odniesień?
4. Czy używany przez sprytne wskaźniki powiązane z licznikiem odniesień mechanizm listy może być jedno-, czy dwukierunkowy?

Ćwiczenia

1. **Łowcy błędów:** Wskaż błąd znajdujący się w poniższym kodzie:

   ```
   std::auto_ptr<SomeClass> pObject(new SomeClass());
   std::auto_ptr<SomeClass> pAnotherObject(pObject);
   pObject->DoSomething();
   pAnotherObject->DoSomething();
   ```

2. Użyj klasy `unique_ptr` do ustanowienia klasy `Carp` dziedziczącej po `Fish`. Przekaż obiekt jako wskaźnik `Fish` i skomentuj segmentowanie, o ile takie wystąpi.

3. **Łowcy błędów:** Wskaż błąd znajdujący się w poniższym kodzie:

   ```
   std::unique_ptr<Tuna> myTuna (new Tuna);
   unique_ptr<Tuna> copyTuna;
   copyTuna = myTuna;
   ```

Lekcja 27

Użycie strumieni w operacjach wejścia-wyjścia

W tej książce strumienie są wykorzystywane od samego początku, czyli od lekcji 1., w której wyświetlono komunikat *Witaj, świecie* przy użyciu std::cout. Warto poświęcić nieco więcej uwagi tej części C++ i poznać strumienie z praktycznego punktu widzenia.

Z tej lekcji dowiesz się:

- **dowiesz się, czym są strumienie i jak się ich używa,**
- **dowiesz się, jak odczytywać i zapisywać pliki za pomocą strumieni,**
- **poznasz użyteczne operacje z użyciem strumieni C++.**

Koncepcja strumieni

Tworzysz program, który odczytuje dane z dysku, zapisuje dane przeznaczone do wyświetlenia, odczytuje dane wprowadzone przez użytkownika za pomocą klawiatury i zapisuje dane na dysku. Czy nie byłoby wygodnie, gdyby można było wszystkie operacje odczytu i zapisu przeprowadzać przy użyciu podobnego wzorca, niezależnie od urządzenia lub położenia, z którego pochodzą dane lub do którego są przekazywane? Dokładnie taką funkcję w języku C++ pełnią strumienie.

Strumienie C++ to ogólna implementacja logiki odczytu i zapisu (innymi słowy, danych wejściowych i wyjściowych), która pozwala na stosowanie tych samych wzorców podczas odczytu i zapisu danych. Wspomniane wzorce pozostają niezmienne, niezależnie od miejsca pochodzenia odczytywanych (dysk lub klawiatura) bądź zapisywanych (dysk lub wyświetlenie na ekranie) danych. Zadaniem programisty jest użycie odpowiedniej klasy strumienia, a implementacja wybranej klasy zajmuje się obsługą wszelkich szczegółów związanych z urządzeniem lub systemem operacyjnym.

Powróćmy do ważnego wiersza w pierwszym programie C++ przedstawionym w listingu 1.1, w lekcji 1.:

```
std::cout << "Witaj, świecie!" << std::endl;
```

Jak łatwo zgadnąć, `std::cout` to obiekt klasy strumienia `ostream` przeznaczonej do wyświetlania danych w konsoli. W celu użycia `std::cout` w programie umieszczono nagłówek `<iostream>` dostarczający tej, a także innych funkcji, np. `std::cin` do odczytu danych ze strumienia.

Możesz zadawać sobie pytanie, co miałem na myśli, pisząc, że strumienie pozwalają na dostęp stały i charakterystyczny dla urządzenia? Jeżeli komunikat *Witaj, świecie* chciałbyś zapisać w pliku tekstowym, wtedy musisz użyć poniższej składni wykorzystującej obiekt strumienia `fsHW`:

```
fsHW << "Witaj, świecie!" << endl;  // Umieszczenie komunikatu "Witaj, świecie!"
                                    // w pliku.
```

Jak widzisz, po wybraniu odpowiedniej klasy strumienia polecenie zapisu komunikatu *Witaj, świecie* do pliku nie różni się niczym od polecenia wyświetlającego komunikat na ekranie.

Wskazówka

Operator << użyty podczas zapisu danych w strumieniu nosi nazwę operatora strumienia wstawiania. Stosujesz go podczas kierowania danych na ekran, do pliku itd.

Operator >> użyty podczas zapisu strumienia w zmiennej jest nazywany operatorem strumienia wyodrębniania. Stosujesz go podczas odczytu danych wejściowych z klawiatury, pliku itd.

W tej lekcji przeanalizujemy strumienie z praktycznego punktu widzenia.

Ważne klasy i obiekty strumieni C++

Język C++ oferuje zestaw standardowych klas i nagłówków, które pomagają w przeprowadzaniu pewnych ważnych i często wykonywanych operacji wejścia-wyjścia. W tabeli 27.1 wymieniono najczęściej używane klasy.

Tabela 27.1. Popularne klasy strumieni C++ dostępne w przestrzeni nazw std

Klasa/Obiekt	Opis
`cout`	Strumień standardowego wyjścia, zwykle przekierowany do konsoli.
`cin`	Strumień standardowego wejścia, zwykle używany do umieszczania danych w zmiennych.
`cerr`	Strumień standardowych komunikatów błędów.
`fstream`	Klasa strumieni wejścia i wyjścia dla operacji przeprowadzanych na plikach, dziedziczy po `ofstream` i `ifstream`.
`ofstream`	Klasa strumienia wyjścia dla operacji przeprowadzanych na plikach, używana do tworzenia plików.
`ifstream`	Klasa strumienia wyjścia dla operacji przeprowadzanych na plikach, używana do odczytu plików.
`stringstream`	Klasa strumieni wejścia i wyjścia dla operacji przeprowadzanych na ciągach tekstowych, dziedziczy po `istringstream` i `ostringstream`, zwykle używana do przeprowadzania konwersji z (lub na) ciąg tekstowy i inne typy.

Uwaga

Obiekty `cout`, `cin` i `cerr` to obiekty globalne klas strumieni (odpowiednio) `ostream`, `istream` i `ostream`. Ponieważ są obiektami globalnymi, ich inicjalizacja następuje jeszcze przed wywołaniem funkcji `main()`.

Podczas używania klasy strumienia masz możliwość wskazania manipulatorów przeprowadzających pewne operacje: `std::endl` to jeden z takich manipulatorów, który jest używany do wstawiania znaku nowego wiersza:

```
std::cout  << "Ten wiesz się tutaj kończy" << std::endl;
```

W tabeli 27.2 wymieniono inne manipulatory i flagi.

Tabela 27.2. Dostępne w przestrzeni nazw std manipulatory najczęściej używane podczas pracy ze strumieniami

Manipulatory wyjścia	Opis
`endl`	Wstawia znak nowego wiersza.
`ends`	Wstawia znak `null`.
Manipulatory podstaw liczb	**Opis**
`dec`	Nakazuje strumieniowi interpretację danych wejściowych lub wyświetlenie danych wyjściowych jako liczb dziesiętnych.
`hex`	Nakazuje strumieniowi interpretację danych wejściowych lub wyświetlenie danych wyjściowych jako liczb szesnastkowych.
`oct`	Nakazuje strumieniowi interpretację danych wejściowych lub wyświetlenie danych wyjściowych jako liczb ósemkowych.
Manipulatory liczb zmiennoprzecinkowych	**Opis**
`fixed`	Nakazuje strumieniowi wyświetlenie danych wyjściowych przy użyciu zapisu zmiennoprzecinkowego.
`scientific <iomanip>`	Nakazuje strumieniowi wyświetlenie danych wyjściowych przy użyciu zapisu wykładniczego.
Manipulatory	**Opis**
`setprecision`	Ustawienie precyzji liczby dziesiętnej jako parametru.
`setw`	Ustawienie szerokości pola jako parametru.
`setfill`	Ustalenie znaku wypełnienia jako parametru.
`setbase`	Ustawienie podstawy (użycie zbliżone do `dec`, `hex` lub `oct`) jako parametru.

Tabela 27.2. Dostępne w przestrzeni nazw std manipulatory najczęściej używane podczas pracy ze strumieniami (cd.)

Manipulatory wyjścia	Opis
`setiosflag`	Ustawienie flag przy użyciu maski parametru danych wejściowych typu `std::ios_base::fmtflags`.
`resetiosflag`	Przywrócenie wartości domyślnych dla określonego typu wskazanego przez `std::ios_base::fmtflags`.

Użycie std::cout do zapisu w konsoli sformatowanych danych

Strumień `std::cout` stosowany w celu zapisu standardowego strumienia danych wyjściowych to prawdopodobnie jeden z najczęściej używanych w tej książce. Pora więc powrócić do `cout` i wykorzystać niektóre z dostępnych manipulatorów do zmiany sposobu wyrównania i wyświetlania danych.

Użycie std::cout do zmiany formatu wyświetlanych liczb

Istnieje możliwość użycia `cout` do wyświetlenia liczby w formacie szesnastkowym lub ósemkowym. W listingu 27.1 zaprezentowano przykład wykorzystania `cout` w celu wyświetlenia w różnych formatach liczby podanej przez użytkownika.

Listing 27.1. Wyświetlenie liczby w zapisie dziesiętnym, ósemkowym i szesnastkowym przy użyciu cout i flag <iomanip>

```
 0: #include <iostream>
 1: #include <iomanip>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Podaj liczbę całkowitą: ";
 7:    int Input = 0;
 8:    cin >> Input;
 9:
10:    cout << "Liczba w formacie ósemkowym: " << oct << Input << endl;
11:    cout << "Liczba w formacie szesnastkowym: " << hex << Input << endl;
12:
```

```
13:    cout << "Liczba w formacie szesnastkowym (zapis podstawowy): ";
14:    cout<<setiosflags(ios_base::hex|ios_base::showbase|
       ↪ios_base::uppercase);
15:    cout << Input << endl;
16:
17:    cout << "Liczba po wyzerowaniu flag I/O: ";
18:    cout<<resetiosflags(ios_base::hex|ios_base::showbase|
       ↪ios_base::uppercase);
19:    cout << Input << endl;
20:
21:    return 0;
22: }
```

Wynik ▼

```
Podaj liczbę całkowitą: 253
Liczba w formacie ósemkowym: 375
Liczba w formacie szesnastkowym: fd
Liczba w formacie szesnastkowym (zapis podstawowy): 0XFD
Liczba po wyzerowaniu flag I/O: 253
```

Analiza ▼

W powyższym fragmencie kodu wykorzystano manipulatory wymienione w tabeli 27.2 do zmiany sposobu wyświetlania przez `cout` tego samego obiektu liczby całkowitej `Input` podanej przez użytkownika. Zwróć uwagę na użycie w wierszach 10. i 11. manipulatorów `oct` i `hex`. W wierszu 14. zastosowano `setiosflags()` do wyświetlenia liczby szesnastkowej przy użyciu wielkich liter, co spowodowało, że polecenie `cout` wyświetliło liczbę całkowitą 253 w postaci `0XFD`. Z kolei wykorzystanie `resetiosflags()` w wierszu 18. pokazuje, że liczba całkowita zostaje przez `cout` ponownie wyświetlona w postaci dziesiętnej. Poniżej zaprezentowano inny sposób zmiany podstawy wyświetlanej liczby (tutaj na liczbę dziesiętną):

```
cout << dec << Input << endl; // Wyświetlenie przy użyciu zapisu dziesiętnego.
```

Istnieje również możliwość określenia sposobu, w jaki `cout` wyświetla liczby, takie jak Pi — w tym celu można wskazać precyzję, czyli ilość cyfr po przecinku dziesiętnym lub też nakazać wyświetlenie liczby przy użyciu zapisu wykładniczego. Tego rodzaju rozwiązanie zaprezentowano w listingu 27.2.

Listing 27.2. Zastosowanie cout do wyświetlenia Pi i pola okręgu przy użyciu zapisu zmiennoprzecinkowego i wykładniczego

```
 0: #include <iostream>
 1: #include <iomanip>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    const double Pi = (double)22.0 / 7;
 7:    cout << "Pi = " << Pi << endl;
 8:
 9:    cout << endl << "Ustalenie precyzji na 7: " << endl;
10:    cout << setprecision(7);
11:    cout << "Pi = " << Pi << endl;
12:    cout << fixed << "Zmiennoprzecinkowe Pi = " << Pi << endl;
13:    cout << scientific << "Wykładnicze Pi = " << Pi << endl;
14:
15:    cout << endl << "Ustalenie precyzji na 10: " << endl;
16:    cout << setprecision(10);
17:    cout << "Pi = " << Pi << endl;
18:    cout << fixed << "Zmiennoprzecinkowe Pi = " << Pi << endl;
19:    cout << scientific << "Wykładnicze Pi = " << Pi << endl;
20:
21:    cout << endl << "Podaj promień: ";
22:    double Radius = 0.0;
23:    cin >> Radius;
24:    cout << "Pole okręgu: " << 2*Pi*Radius*Radius << endl;
25:
26:    return 0;
27: }
```

Wynik ▼

```
Pi = 3.14286

Ustalenie precyzji na 7:
Pi = 3.142857
Zmiennoprzecinkowe Pi = 3.1428571
Wykładnicze Pi = Pi = 3.1428571e+000

Ustalenie precyzji na 10:
Pi = 3.1428571429e+000
Zmiennoprzecinkowe Pi = 3.1428571429
Wykładnicze Pi = 3.1428571429e+000

Podaj promień: 9.99
Pole okręgu: 6.2731491429e+002
```

Analiza ▼

Dane wyjściowe pokazują, jak zwiększenie w wierszu 10. precyzji do 7 cyfr oraz w wierszu 16. do 10 cyfr powoduje zmianę wyświetlanej wartości Pi. Zwróć uwagę jak manipulator `scentific` wpływa na wyświetlenie obliczonego pola okręgu w postaci `6.2731491429e+002`.

Wyrównanie tekstu i ustawienie szerokości pola przy użyciu std::cout

Do ustawienia wyrażonej w znakach szerokości pola możesz użyć manipulatora `setw()`. Wszystkie dane wstawiane do strumienia są wyrównane do prawej strony w zdefiniowanej szerokości pola. Z kolei manipulator `setfill()` pozwala na wskazanie znaku, który wypełni puste miejsca w polu. Przykład zaprezentowano w listingu 27.3.

Listing 27.3. Ustawienie szerokości pola przy użyciu setw() i znaku wypełnienia przy użyciu setfill()

```
 0: #include <iostream>
 1: #include <iomanip>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Hej - wartości domyślne!" << endl;
 7:
 8:    cout << setw(35);   // Ustawienie szerokości pola na 35 kolumn.
 9:    cout << "Hej - wyrównane do prawej!" << endl;
10:
11:    cout << setw(35) << setfill('*');
12:    cout << "Hej - wyrównane do prawej!" << endl;
13:
14:    cout << "Hej - powrót do wartości domyślnych!" << endl;
15:
16:    return 0;
17: }
```

Wynik ▼

```
Hej - wartości domyślne!
        Hej - wyrównane do prawej!
********Hej - wyrównane do prawej!
Hej - powrót do wartości domyślnych!
```

Analiza ▼

Dane wyjściowe pokazują efekt wywołania w wierszu 8. manipulatora `setw(35)` dla polecenia `cout` oraz użycia w wierszu 11. manipulatorów `setfill('*')` i `setw(35)`. Jak możesz zobaczyć, w tym drugim przypadku wolne miejsca w polu poprzedzające wyświetlany tekst zostały wypełnione gwiazdkami, zgodnie ze znakiem wskazanym w manipulatorze `setfill()`.

Użycie std::cin dla danych wejściowych

Strumień `std::cin` to wszechstronny strumień pozwalający na wczytywanie danych wejściowych i umieszczanie ich w typach danych, takich jak `int`, `double` i `char*` (ciągi tekstowe w stylu C). Ponadto daje możliwość odczytywania wierszy lub znaków z ekranu przy użyciu metod, takich jak `getline()`.

Użycie std::cin w celu umieszczenia danych wejściowych w zmiennych typu POD

Liczby całkowite, podwójnej precyzji i znaki można przy użyciu `cin` pobierać bezpośrednio ze standardowego wejścia. W listingu 27.4 zaprezentowano użycie `cin` w prostym programie odczytującym dane wprowadzone przez użytkownika.

Listing 27.4. Użycie cin w celu umieszczenia danych wejściowych w zmiennej typu int, liczby zmiennoprzecinkowej w formacie wykładniczym, zmiennej typu double oraz trzech liter w zmiennej typu char

```
 0: #include<iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Podaj liczbę całkowitą: ";
 6:    int InputInt = 0;
 7:    cin >> InputInt;
 8:
 9:    cout << "Podaj wartość Pi: ";
10:    double Pi = 0.0;
11:    cin >> Pi;
12:
13:    cout << "Podaj trzy znaki rozdzielone spacjami: " << endl;
```

```
14:    char Char1 = '\0', Char2 = '\0', Char3 = '\0';
15:    cin >> Char1 >> Char2 >> Char3;
16:
17:    cout << "Zapisano następujące wartości zmiennych: " << endl;
18:    cout << "InputInt: " << InputInt << endl;
19:    cout << "Pi: " << Pi << endl;
20:    cout << "Trzy litery: " << Char1 << Char2 << Char3 << endl;
21:
22:    return 0;
23: }
```

Wynik ▼

```
Podaj liczbę całkowitą: 32
Podaj wartość Pi: 0.314159265e1
Podaj trzy znaki rozdzielone spacjami:
c + +
Zapisano następujące wartości zmiennych:
InputInt: 32
Pi: 3.14159
Trzy litery: c++
```

Analiza ▼

W powyższym programie najbardziej interesujące jest to, że wartość `Pi` została wprowadzona przy użyciu zapisu wykładniczego, natomiast polecenie `cin` umieściło tę wartość w zmiennej `double Pi`. Zwróć uwagę na możliwość przypisania znaków trzem zmiennym za pomocą tylko jednego polecenia, co pokazano w wierszu 15.

Użycie std::cin::get w celu umieszczenia danych wejściowych w buforze typu char w stylu C

Podobnie jak `cin` pozwala na zapisanie danych wejściowych bezpośrednio w zmiennej `int`, tak samo można je zapisać w tablicy typu `char` w stylu C:

```
cout << "Podaj wiersz: " << endl;
char CStyleStr [10] = {0};   // Może zawierać maksymalnie 10 znaków.
cin >> CStyleStr;   // Uwaga: użytkownik może wprowadzić więcej niż 10 znaków.
```

Podczas zapisu danych w buforze ciągu tekstowego stylu C bardzo ważne jest, aby nie wykroczyć poza granice bufora w celu uniknięcia awarii aplikacji lub

powstania luki w zabezpieczeniach. Dlatego też lepszy sposób umieszczenia danych wejściowych w buforze char stylu C przedstawia się następująco:

```
cout << "Podaj wiersz: " << endl;
char CStyleStr[10] = {0};
cin.get(CStyleStr, 9);  // Zatrzymanie wstawiania w dziewiątym znaku.
```

Ten bezpieczniejszy sposób wstawiania tekstu do bufora stylu C zademonstrowano w listingu 27.5.

Listing 27.5. Wstawienie znaków do bufora w stylu C bez przekroczenia jego granic

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Podaj wiersz: " << endl;
 7:    char CStyleStr[10] = {0};
 8:    cin.get(CStyleStr, 9);
 9:    cout << "CStyleStr: " << CStyleStr << endl;
10:
11:    return 0;
12: }
```

Wynik ▼

```
Podaj wiersz:
Mogę wykroczyć poza granice bufora?
CStyleStr: Mogę wykr
```

Analiza ▼

Jak widzisz w wygenerowanych danych wejściowych, ze względu na użycie cin::get w wierszu 8. w buforze w stylu C zostało umieszczonych jedynie pierwszych dziewięć znaków danych wejściowych wprowadzonych przez użytkownika. To jest najbezpieczniejszy sposób pracy z ciągiem tekstowym w stylu C.

Jeśli to możliwe, nie używaj ciągów tekstowych w stylu C i tablic char. Zamiast z char* korzystaj z std::string, gdy tylko będzie taka możliwość.

Wskazówka
Wskazówka

Użycie std::cin w celu umieszczenia danych wejściowych w std::string

Narzędzie `cin` to bardzo wszechstronne narzędzie i można je wykorzystać nawet do pobrania ciągu tekstowego od użytkownika i bezpośredniego umieszczenia w `std::string`:

```
std::string Input;
cin >> Input;  // Zatrzymanie wstawiania na pierwszej spacji.
```

Użycie `cin` do wstawienia danych wejściowych w `std::string` zaprezentowano w listingu 27.6.

Listing 27.6. Wstawianie tekstu do obiektu std::string przy użyciu cin

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Podaj imię: ";
 7:    string Name;
 8:    cin >> Name;
 9:    cout << "Witaj, " << Name << endl;
10:
11:    return 0;
12: }
```

Wynik ▼

```
Podaj imię: Jan Kowalski
Witaj, Jan
```

Analiza ▼

Wyświetlone dane wyjściowe pokazują, że program nie wyświetlił nazwiska, a tylko samo imię. Możesz oczekiwać, że w wierszu 8. zmienna `Name` otrzyma od `cin` dane w postaci pełnego imienia i nazwiska, a nie tylko imię. Co więc się stało? Wstawianie znaków zostało zatrzymane po napotkaniu pierwszego znaku odstępu.

Aby umożliwić użytkownikowi podanie pełnego imienia i nazwiska wraz ze spacjami, konieczne jest użycie metody `getline()`:

```
string Name;
getline(cin, Name);
```

Użycie metody `getline()` wraz z `cin` zaprezentowano w listingu 27.7.

Listing 27.7. Odczyt całego wiersza danych wejściowych użytkownika przy użyciu getline() i cin

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Podaj imię: ";
 7:    string Name;
 8:    getline(cin, Name);
 9:    cout << "Witaj, " << Name << endl;
10:
11:    return 0;
12: }
```

Wynik ▼

```
Podaj imię: Jan Kowalski
Witaj, Jan Kowalski
```

Analiza ▼

Użyta w wierszu 8. metoda `getline()` dobrze wykonała swoje zadania i zagwarantowała uwzględnienie w danych wejściowych także znaków odstępu. Wygenerowane przez program dane wyjściowe zawierają teraz pełny wiersz wprowadzony przez użytkownika.

Użycie std::fstream do obsługi pliku

Klasa `std::fstream` to klasa dostarczana przez C++ w celu zapewnienia (względnie) niezależnego od platformy dostępu do pliku. Klasa `std::fstream` dziedziczy po `std::ofstream` w zakresie zapisu pliku oraz po `std::ofstream` w zakresie odczytu pliku.

Innymi słowy, `std::fstream` zapewnia funkcje pozwalające na odczyt i zapis pliku.

Wskazówka

W celu użycia klasy `std::fstream` lub jej klas bazowych w programie trzeba umieścić następujący nagłówek:

```
#include <fstream>
```

Otwieranie i zamykanie pliku przy użyciu metod open() i close()

Aby użyć klasy `fstream`, `ofstream` lub `ifstream`, konieczne jest otworzenie pliku przy użyciu metody `open()`:

```
fstream myFile;
myFile.open("HelloFile.txt",ios_base::in|ios_base::out|ios_base::trunc);
if (myFile.is_open())
{
   // Operacje odczytu lub zapisu.
   myFile.close();
}
```

Metoda `open()` pobiera dwa argumenty. Pierwszy to ścieżka dostępu i nazwa otwieranego pliku (jeśli nie podasz ścieżki dostępu, zostanie przyjęte założenie, że plik znajduje się w katalogu roboczym). Drugi to tryb, w którym ma zostać otworzony plik. Wybrany tryb pozwala nawet na utworzenie nieistniejącego pliku (`ios_base::trunc`), a także odczyt i zapis w pliku (`in | out`).

Zwróć uwagę na użycie metody `is_open()` w celu sprawdzenia, czy wywołanie metody `open()` zakończyło się powodzeniem.

Ostrzeżenie

Pamiętaj, że zamknięcie strumienia pliku jest ważne w celu zapisania jego zawartości.

Istnieje jeszcze inny sposób otworzenia strumienia pliku, tym razem przy użyciu konstruktora:

```
fstream myFile("HelloFile.txt",ios_base::in|ios_base::out|ios_base::trunc);
```

Jeżeli chcesz otworzyć plik jedynie w celu zapisu, możesz skorzystać z poniższego polecenia:

```
ofstream myFile("HelloFile.txt", ios_base::out);
```

Natomiast otworzenie pliku tylko do odczytu jest możliwe przy użyciu poniższego polecenia:

```
ifstream myFile("HelloFile.txt", ios_base::in);
```

Wskazówka

Niezależnie od zastosowanego sposobu (konstruktor lub metoda składowa `open()`), zaleca się, aby wynik operacji otworzenia pliku sprawdzić przy użyciu metody `is_open()` przed kontynuacją pracy z odpowiednim obiektem strumienia pliku.

Poniżej wymieniono różne tryby, w których można otworzyć strumień pliku:

- `ios_base::app` — dołączenie danych na końcu istniejącego pliku zamiast jego skracania,
- ios_base::ate — umieszczenie danych na końcu pliku, choć istnieje możliwość ich umieszczenia w dowolnym miejscu pliku,
- ios_base::trunc — skrócenie istniejącego pliku — to jest tryb domyślny,
- ios_base::binary — utworzenie pliku binarnego (domyślnie jest tworzony plik tekstowy),
- ios_base::in — otworzenie pliku tylko w celu przeprowadzenia operacji odczytu,
- ios_base::`out` — otworzenie pliku tylko w celu przeprowadzenia operacji zapisu.

Tworzenie i zapisywanie tekstu w pliku przy użyciu metody open() i operatora <<

Po otworzeniu strumienia pliku można zapisać w nim dane przy użyciu operatora <<, jak zaprezentowano w listingu 27.8.

Listing 27.8. Utworzenie nowego pliku i umieszczenie w nim tekstu przy użyciu ofstream

```
 0: #include<fstream>
 1: #include<iostream>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    ofstream myFile;
 7:    myFile.open("HelloFile.txt", ios_base::out);
 8:
 9:    if (myFile.is_open())
10:    {
11:       cout << "Otworzenie pliku zakończyło się powodzeniem" << endl;
12:
```

```
13:       myFile << "Mój pierwszy plik tekstowy!" << endl;
14:       myFile << "Witaj z pliku!";
15:
16:       cout << "Zakończono zapis w pliku, plik zostanie zamknięty" <<
          ↪endl;
17:       myFile.close();
18:    }
19:
20:    return 0;
21: }
```

Wynik ▼

```
Otworzenie pliku zakończyło się powodzeniem
Zakończono zapis w pliku, plik zostanie zamknięty
```

Zawartość pliku *HelloFile.txt*:

```
Mój pierwszy plik tekstowy!
Witaj z pliku!
```

Analiza ▼

W wierszu 7. plik został otworzony w trybie `ios_base::out`, tzn. wyłącznie dla operacji zapisu. Wiersz 9. pokazuje użycie metody `is_open()` w celu sprawdzenia, czy operacja otworzenia pliku zakończyła się powodzeniem. Następnie (patrz wiersze 13. i 14.) przeprowadzany jest zapis danych w strumieniu pliku przy użyciu operatora <<. Wreszcie, w wierszu 17. plik zostaje zamknięty.

Uwaga

W listingu 27.8 pokazano, że zapis danych w strumieniu pliku przeprowadza się w dokładnie taki sam sposób jak ich przekazanie do standardowego wyjścia (konsoli) przy użyciu `cout`.

Przekonałeś się więc, że strumienie C++ pozwalają na obsługę różnych urządzeń w podobny sposób. Tekst jest wyświetlany na ekranie przy użyciu `cout` i tak samo zapisywany w pliku za pomocą `ofstream`.

Odczyt tekstu z pliku przy użyciu metody open() i operatora >>

Aby odczytać zawartość pliku, konieczne jest wykorzystanie klasy `fstream` i otworzenie pliku z użyciem flagi `ios_base::in` lub też użycie klasy `ifstream`.

W listingu 27.9 zaprezentowano odczyt pliku *HelloFile.txt* utworzonego w listingu 27.8.

Listing 27.9. Odczyt tekst z pliku HelloFile.txt utworzonego w listingu 27.8

```
 0: #include<fstream>
 1: #include<iostream>
 2: #include<string>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    ifstream myFile;
 8:    myFile.open("HelloFile.txt", ios_base::in);
 9:
10:    if (myFile.is_open())
11:    {
12:       cout << "Otworzenie pliku zakończyło się powodzeniem,
          ↪jego zawartość:" << endl;
13:       string fileContents;
14:
15:       while (myFile.good())
16:       {
17:          getline (myFile, fileContents);
18:         cout << fileContents << endl;
19:       }
20:
21:       cout << "Zakończono odczyt z pliku, plik zostanie zamknięty" << endl;
22:       myFile.close();
23:    }
24:    else
25:       cout << "Błąd open(): sprawdź, czy plik znajduje się
          ↪w odpowiednim katalogu" << endl;
26:
27:    return 0;
28: }
```

Wynik ▼

```
Otworzenie pliku zakończyło się powodzeniem, jego zawartość:
Mój pierwszy plik tekstowy!
Witaj z pliku!
Zakończono odczyt z pliku, plik zostanie zamknięty
```

Uwaga

> Ponieważ kod przedstawiony w listingu 27.9 odczytuje plik tekstowy *HelloFile.txt* utworzony w listingu 27.8, konieczne jest umieszczenie wymienionego pliku w katalogu roboczym bieżącego projektu lub połączenie tego kodu z poprzednim.

Analiza

Jak zawsze, wywołana zostaje metoda `is_open()` w celu sprawdzenia, czy wywołanie `open()` w wierszu 8. zakończyło się powodzeniem. Zwróć uwagę na użycie operatora wyodrębniania >> podczas odczytu zawartości pliku i jej bezpośredniego umieszczenia w zmiennej typu `string` wyświetlanej następnie w wierszu 18. przez `cout`. W omawianym programie użyto metody `getline()` do odczytu danych wejściowych ze strumienia pliku. W dokładnie taki sam sposób jak w listingu 27.7 odczytano dane wejściowe wprowadzone przez użytkownika, czyli jednorazowo po jednym pełnym wierszu.

Zapis i odczyt pliku binarnego

Rzeczywisty proces odczytu i zapisu danych w pliku binarnym nie różni się zbyt od tego samego procesu dla pliku tekstowego. Bardzo ważne jest użycie flagi `ios_base::binary` jako maski podczas otwierania pliku. Zwykle stosowane są metody `ofstream::write` lub `ifstream::read`, co zaprezentowano w listingu 27.10.

Listing 27.10. Zapis struktury w pliku binarnym i jej rekonstrukcja z pliku binarnego

```
 0: #include<fstream>
 1: #include<iomanip>
 2: #include<string>
 3: #include<iostream>
 4: using namespace std;
 5:
 6: struct Human
 7: {
 8:    Human() {};
 9:    Human(const char* inName, int inAge, const char* inDOB) : Age(inAge)
10:    {
11:       strcpy(Name, inName);
12:       strcpy(DOB, inDOB);
13:    }
14:
15:    char Name[30];
```

```
16:    int Age;
17:    char DOB[20];
18: };
19:
20: int main()
21: {
22:    Human Input("Jan Kowalski", 101, "maj 1910");
23:
24:    ofstream fsOut ("MyBinary.bin", ios_base::out | ios_base::binary);
25:
26:    if (fsOut.is_open())
27:    {
28:      cout << "Zapis obiektu Human w pliku binarnym" << endl;
29:       fsOut.write(reinterpret_cast<const char*>(&Input),
          ↪sizeof(Input));
30:       fsOut.close();
31:    }
32:
33:    ifstream fsIn ("MyBinary.bin", ios_base::in | ios_base::binary);
34:
35:    if(fsIn.is_open())
36:    {
37:       Human somePerson;
38:       fsIn.read((char*)&somePerson, sizeof(somePerson));
39:
40:       cout << "Odczyt informacji z pliku binarnego: " << endl;
41:       cout << "Imię i nazwisko = " << somePerson.Name << endl;
42:       cout << "Wiek = " << somePerson.Age << endl;
43:       cout << "Data urodzenia = " << somePerson.DOB << endl;
44:    }
45:
46:    return 0;
47: }
```

Wynik ▼

```
Zapis obiektu Human w pliku binarnym
Odczyt informacji z pliku binarnego:
Imię i nazwisko = Jan Kowalski
Wiek = 101
Data urodzenia = maj 1910
```

Analiza ▼

W wierszach od 22. do 31. następuje utworzenie egzemplarza struktury `Human` zawierającej atrybuty `Name`, `Age` i `DOB`, które następnie przy użyciu klasy `ofstream` są zapisywane na dysku w pliku binarnym o nazwie *MyBinary.bin*.

Zapisane informacje w dalszej części programu (patrz wiersze od 33. do 44.) są odczytywane i umieszczane w innym obiekcie strumienia. Dane wyjściowe atrybutów, takich jak `Name`, zostały pobrane z pliku binarnego. Przykład pokazuje również użycie klas `ifstream` i `ofstream` w celu odczytu i zapisu pliku za pomocą (odpowiednio) `ifstream::read` i `ofstream::write`. Zwróć uwagę na użycie `reinterpret_cast` w wierszu 29. w celu wymuszenia na kompilatorze interpretacji struktury jako `char*`. W wierszu 38. użyto rzutowania w stylu C jako odpowiednika polecenia w wierszu 29.

Uwaga

Jeżeli przedstawiony program nie byłby przeznaczony do celów demonstracyjnych, strukturę `Human` wraz z jej wszystkimi atrybutami lepiej zapisać w pliku XML. Wspomniany XML to oparty na znacznikach tekstowy format przechowywania danych charakteryzujący się dużą elastycznością i skalowalnością w zakresie sposobu, w jakim będą przechowywane dane.

Jeżeli struktura `Human` będzie używana w przedstawionej powyżej postaci, a następnie zechcesz dodać do niej nowe atrybuty (np. `NumChildren`), wtedy musisz zadbać, aby metoda `ifstream::read` prawidłowo odczytała dane binarne utworzone przy użyciu starszej wersji struktury.

Użycie std::stringstream do konwersji ciągu tekstowego

Masz ciąg tekstowy zawierający wartość 45. W jaki sposób można ten ciąg teksty skonwertować na liczbę całkowitą o wartości 45 lub na odwrót? Jednym z najbardziej użytecznych narzędzi dostarczanych przez C++ jest klasa `stringstream` pozwalająca na przeprowadzanie różnego rodzaju konwersji.

Wskazówka

W celu użycia klasy `std::stringstream` w programie trzeba umieścić następujący nagłówek:

```
#include <sstream>
```

W listingu 27.11 zaprezentowano przykładowe operacje wykonywane przy użyciu klasy `stringstream`.

Listing 27.11. Konwersja liczby całkowitej na postać ciągu tekstowego i na odwrót przy użyciu klasy std::stringstream

```
0: #include<fstream>
1: #include<sstream>
```

```
 2: #include<iostream>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    cout << "Podaj liczbę całkowitą: ";
 8:    int Input = 0;
 9:    cin >> Input;
10:
11:    stringstream converterStream;
12:    converterStream << Input;
13:    string strInput;
14:    converterStream >> strInput;
15:
16:    cout << "Podana liczba całkowita = " << Input << endl;
17:    cout << "Ciąg tekstowy utworzony na podstawie liczby całkowitej,
       ↪strInput = " << strInput << endl;
18:
19:    stringstream anotherStream;
20:    anotherStream << strInput;
21:    int Copy = 0;
22:    anotherStream >> Copy;
23:
24:    cout << "Liczba całkowita utworzona na podstawie ciągu tekstowego,
       ↪Copy = " << Copy << endl;
25:
26:    return 0;
27: }
```

Wynik ▼

```
Podaj liczbę całkowitą: 45
Podana liczba całkowita = 45
Ciąg tekstowy utworzony na podstawie liczby całkowitej, strInput = 45
Liczba całkowita utworzona na podstawie ciągu tekstowego, Copy = 45
```

Analiza ▼

Użytkownik zostaje poproszony o podanie liczby całkowitej. Podana liczba całkowita jest najpierw przy użyciu operatora << umieszczana w obiekcie `stringstream`, jak pokazano w wierszu 12. Następnie w wierszu 14. z wykorzystaniem operatora wyodrębniania >> ta liczba całkowita jest konwertowana na postać ciągu tekstowego. Otrzymany w ten sposób ciąg tekstowy stanowi punkt wyjścia do uzyskania na jego podstawie liczby całkowitej (`Copy`).

TAK	NIE
Używaj klasy ifstream, jeśli zamierzasz jedynie przeprowadzać operacje odczytu z pliku. Używaj klasy ofstream, jeśli zamierzasz jedynie przeprowadzać operacje zapisu w pliku. Pamiętaj o użyciu metody is_open() do sprawdzenia, czy otworzenie strumienia pliku zakończyło się powodzeniem. Sprawdzenie należy przeprowadzić przed wstawieniem lub pobraniem danych ze strumienia.	Nie zapominaj o zamknięciu strumienia pliku za pomocą metody close() po zakończeniu pracy z nim. Nie zapominaj, że wyodrębnianie danych z cin do ciągu tekstowego przez cin >> strData; zwykle oznacza umieszczenie w strData tekstu jedynie do napotkania pierwszego znaku odstępu, a nie całego wiersza. Nie zapominaj, że funkcja getline(cin, strData); powoduje pobranie całego wiersza ze strumienia danych wejściowych, łącznie ze znakami odstępu.

Podsumowanie

W tej lekcji poznałeś strumienie C++ z praktycznego punktu widzenia. Przekonałeś się, że strumienie wejścia-wyjścia `cout` i `cin` są od samego początku używane w tej książce. Potrafisz już utworzyć proste pliki tekstowe, a także odczytywać i zapisywać w nich dane. Dowiedziałeś się, jak klasa `stringstream` może pomóc w konwersji prostych typów, takich jaki liczby całkowite, na postać ciągów tekstowych i na odwrót.

Pytania i odpowiedzi

Pytanie: Jak widzę, klasy `fstream` mogę używać zarówno do zapisu, jak i odczytu danych z pliku. Kiedy zatem powinienem stosować `ofstream` i `ifstream`?

Odpowiedź: Jeżeli kod lub moduł wymaga jedynie możliwości odczytu danych z pliku, wtedy powinieneś użyć klasy `ifstream`. Podobnie, jeśli wymagana jest jedynie możliwość zapisu danych z pliku, powinieneś skorzystać z klasy `ofstream`. W obu wymienionych sytuacjach klasa `fstream` sprawdzi się doskonale, ale w celu zapewnienia spójności danych i kodu lepszym rozwiązaniem jest zachowanie restrykcyjnej polityki, podobnej do użycia `const`, co jednak nie jest obowiązkowe.

Pytanie: Kiedy powinienem używać `cin.get()`, a kiedy `cin.getline()`?

Odpowiedź: Metoda `cin.getline()` gwarantuje przechwycenie całego (łącznie ze znakami odstępu) wiersza danych wejściowych wprowadzonych przez użytkownika. Z kolei `cin.get()` pomaga w pobieraniu po jednym znaku z danych wejściowych użytkownika.

Pytanie: Kiedy powinienem używać klasy `stringstream`?

Odpowiedź: Klasa `stringstream` zapewnia wygodny sposób konwersji liczb całkowitych oraz innych prostych typów danych na postać ciągu tekstowego oraz na odwrót, jak to zaprezentowano w listingu 27.11.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Musisz nadpisać plik. Którego strumienia użyjesz?
2. W jaki sposób możesz użyć `cin` w celu pobrania pełnego wiersza ze strumienia danych wejściowych?
3. Musisz zapisać obiekty `std::string` w pliku. Czy wybierzesz tryb `ios_base::binary`?
4. Otworzyłeś strumień przy użyciu metody `open()`. Dlaczego powinieneś jeszcze skorzystać z metody `is_open()`?

Ćwiczenia

1. **Łowcy błędów:** Wskaż błąd znajdujący się w poniższym kodzie:

```
myFile.open("HelloFile.txt", ios_base::out);
myFile << "Witaj z pliku!";
myFile.close();
```

2. **Łowcy błędów:** Wskaż błąd znajdujący się w poniższym kodzie:

```
ifstream MyFile("SomeFile.txt");
if(MyFile.is_open())
{
   MyFile << "To jest dowolny tekst" << endl;
   MyFile.close();
}
```

Lekcja 28

Obsługa wyjątków

Tytuł lekcji mówi wszystko: obsługa wyjątkowych sytuacji, które powodują zakłócenie działania programu. W dotychczasowych lekcjach przyjęliśmy przesadnie pozytywne podejście i założenie, że wszelkie operacje zarządzania pamięcią zakończą się powodzeniem, używane pliki będą dostępne itd. Rzeczywistość jest jednak zupełnie inna.

Z tej lekcji dowiesz się:

- **czym jest wyjątek,**
- **jak obsługiwać wyjątki,**
- **jak obsługa wyjątków pomaga w tworzeniu stabilnych aplikacji C++.**

Czym jest wyjątek?

Tworzony przez Ciebie program alokuje pamięć, odczytuje i zapisuje dane, tworzy pliki — po prostu działa. Wszystkie wymienione operacje są przeprowadzane bez żadnych problemów w środowisku testowym, jesteś nawet dumny, że choć program zarządza gigabajtem pamięci, nie cierpi z powodu wycieków pamięci. Wydajesz aplikację, a klient instaluje ją w tysiącu stacji roboczych, niektóre z jego komputerów mają po dziesięć lat i ledwo działające dyski twarde. Naprawdę nie upłynie zbyt wiele czasu, a otrzymasz pierwsze wiadomości e-mail ze skargami na program. Część skarg będzie dotyczyła błędu typu *Access Violation*, z kolei inne wiązać się będą z „nieobsłużonym wyjątkiem".

Tu Cię mam — „nieobsłużony" i „wyjątek". Skoro aplikacja doskonale działała w środowisku programisty, co mogło się stać?

Świat zewnętrzny jest bardzo heterogeniczny. Dwa komputery nawet w takiej samej konfiguracji sprzętowej nie będą jednakowe. Wynika to z działającego w nich oprogramowania, a stan komputera decyduje o ilości zasobów dostępnych w danej chwili. Istnieje więc duże prawdopodobieństwo, że operacja alokacji pamięci, która doskonale działa w środowisku programisty, zakończy się niepowodzeniem w innym środowisku.

Tego rodzaju sytuacje są rzadkie, ale jednak się zdarzają. Takie niepowodzenie skutkuje zgłoszeniem „wyjątku".

Wyjątki zakłócają normalne działanie programu. W końcu, jeśli nie będzie dostępnej wolnej pamięci, aplikacja nie może wykonać zadania, do którego została przygotowana. Program może jednak obsłużyć wyjątek, wyświetlić użytkownikowi przyjazny komunikat, wykonać minimalne operacje ratunkowe i elegancko zakończyć działanie.

Obsługa wyjątków pomaga w uniknięciu otrzymywania od użytkowników skarg związanych ze wspomnianymi wcześniej błędami typu *Access Violation* lub z nieobsłużonymi wyjątkami. Sprawdzimy teraz, jakie narzędzia język C++ dostarcza w celu obsługi nieoczekiwanych zdarzeń i sytuacji.

Co powoduje zgłoszenie wyjątku?

Wyjątki są powodowane przez czynniki zewnętrzne, takie jak niewystarczająca ilość zasobów systemowych lub czynniki wewnętrzne w aplikacji, np. wskaźnik zawierający nieprawidłową wartość bądź też dzielenie przez zero. Niektóre moduły zostały zaprojektowane do informowania o błędach przez zgłaszanie wyjątków wywołującemu.

> Aby chronić kod przed wyjątkami, musisz zaimplementować ich „obsługę", a tym samym zagwarantować, że kod „zapewni bezpieczeństwo wyjątków".

Uwaga

Implementacja bezpieczeństwa wyjątków przy użyciu try i catch

W języku C++ `try` i `catch` to najważniejsze słowa kluczowe stosowane w implementacji zapewniającej bezpieczeństwo wyjątków. Polecenia zapewniające bezpieczeństwo wyjątków umieszcza się w bloku `try`, a obsługą wyjątków zajmują się polecenia w bloku `catch`.

```
void SomeFunc()
{
   try
   {
      int* pNumber = new int;
      *pNumber = 999;
      delete pNumber;
   }
   catch(...)   // Przechwycenie wszystkich wyjątków.
   {
      cout << "Wyjątek w funkcji SomeFunc(), zamknięcie programu" << endl;
   }
}
```

Użycie catch(...) do obsługi wszystkich wyjątków

W lekcji 8., zatytułowanej „Wskaźniki i referencje", wspomniano, że domyślna forma operatora `new` zwraca prawidłowy wskaźnik do położenia w pamięci, gdy alokacja zakończy się powodzeniem; w przeciwnym razie następuje zgłoszenie wyjątku. W listingu 28.1 przedstawiono rozwiązanie zapewniające bezpieczeństwo wyjątków podczas alokacji pamięci przy użyciu operatora `new` i obsługę sytuacji, gdy nie będzie możliwości alokacji żądanej pamięci.

Listing 28.1. Użycie poleceń try i catch w celu zagwarantowania bezpieczeństwa wyjątku w alokacji pamięci

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Podaj ilość liczb całkowitych, które chcesz zarezerwować: ";
 6:    try
 7:    {
 8:       int Input = 0;
 9:       cin >> Input;
10:
11:       // Żądanie pamięci dla zarezerwowanych liczb.
12:       int* pReservedInts = new int [Input];
13:       delete[] pReservedInts;
14:    }
15:    catch (...)
16:    {
17:       cout << "Zgłoszono wyjątek. Przepraszamy, ale nastąpi zamknięcie
          ➥programu!" << endl;
18:    }
19:    return 0;
20: }
```

Wynik ▼

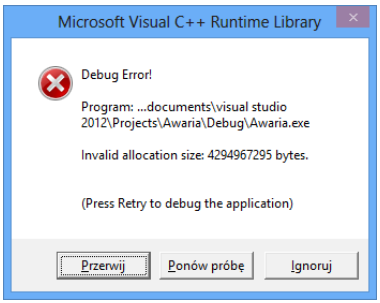
```
Podaj ilość liczb całkowitych, które chcesz zarezerwować: -1
Zgłoszono wyjątek. Przepraszamy, ale nastąpi zamknięcie programu!
```

Analiza ▼

W omawianym przykładzie następuje próba zarezerwowania -1 liczb całkowitych. Wprawdzie te dane wejściowe są absurdalne, ale podczas pracy z aplikacjami użytkownicy bardzo często postępują nie tylko nieracjonalnie, a wręcz sprzecznie z wszelkimi zdroworozsądkowymi regułami. W przypadku braku procedury obsługi wyjątków wspomniana próba spowoduje nagłe zakończenie działania programu. Jednak dzięki obsłudze wyjątków aplikacja wyświetli użytkownikowi użyteczny komunikat informujący o problemie (`Przepraszamy, ale nastąpi zamknięcie programu!`).

Uwaga

Jeżeli omawiany program uruchomisz w Visual Studio, możesz otrzymać komunikat wygenerowany przez tryb debugowania, taki jaki pokazano na rysunku 28.1.



RYSUNEK 28.1. Asercja spowodowana błędem nieprawidłowej alokacji pamięci

Kliknięcie przycisku *Ignoruj* spowoduje uruchomienie procedury obsługi wyjątków. Pokazane okno dialogowe zawiera komunikat trybu debugowania, ale procedura obsługi wyjątków pomaga w eleganckim zakończeniu działania programu w trybie Release.

W kodzie w listingu 28.1 zaprezentowano użycie bloków `try` i `catch`. Podobnie jak funkcja, także `catch()` pobiera parametry, ten blok akceptuje wszystkie rodzaje wyjątków. Jednak w omawianym przykładzie chcemy specjalnie wyizolować wyjątki typu `std::bad_alloc`, ponieważ są one zgłaszane, gdy działanie operatora `new` kończy się niepowodzeniem. Przechwycenie wyjątku określonego typu pomaga w jego lepszej obsłudze, np. wyświetla użytkownikowi dokładną informację o zaistniałym problemie.

Przechwytywanie wyjątku określonego typu

Wyjątek w listingu 28.1 był zgłaszany przez bibliotekę standardową C++. Tego rodzaju wyjątki są znanych typów. Przechwycenie określonego typu pozwala dokładnie określić powód zgłoszenia wyjątku, lepiej obsłużyć wyjątek, a przynajmniej wyświetlić użytkownikowi komunikat precyzyjnie informujący o danym problemie, co zaprezentowano w listingu 28.2.

Listing 28.2. Przechwytywanie wyjątków typu std::bad_alloc

```
0: #include <iostream>
1: #include<exception>  // Dołącz ten nagłówek, aby przechwytywać wyjątki typu bad_alloc.
2: using namespace std;
3:
4: int main()
5: {
```

```
 6:    cout << "Podaj ilość liczb całkowitych, które chcesz zarezerwować: ";
 7:    try
 8:    {
 9:       int Input = 0;
10:       cin >> Input;
11:
12:       // Żądanie pamięci dla zarezerwowanych liczb.
13:       int* pReservedInts = new int [Input];
14:       delete[] pReservedInts;
15:    }
16:    catch (std::bad_alloc& exp)
17:    {
18:       cout << "Zgłoszono wyjątek: " << exp.what() << endl;
19:       cout << "Przepraszamy, ale nastąpi zamknięcie programu!" << endl;
20:    }
21:    catch(...)
22:    {
23:       cout << "Zgłoszono wyjątek. Przepraszamy, ale nastąpi zamknięcie
       ↪programu!" << endl;
24:    }
25:    return 0;
26: }
```

Wynik ▼

```
Podaj ilość liczb całkowitych, które chcesz zarezerwować: -1
Zgłoszono wyjątek: bad allocation
Przepraszamy, ale nastąpi zamknięcie programu!
```

Analiza ▼

Porównaj dane wyjściowe listingów 28.1 i 28.2. Jak możesz się przekonać, dane wyjściowe listingu 28.2 zawierają dokładny komunikat, który precyzyjnie informuje użytkownika o powodzie zakończenia działania aplikacji (tutaj: `bad_allocation`). Wynika to z umieszczenia w kodzie (patrz wiersze od 16. do 20.) dodatkowego bloku `catch` (tak, kod zawiera dwa bloki `catch`) przeznaczonego do obsługi wyjątków typu `catch(bad_alloc&)`, który jest zgłaszany przez operator `new`.

Wskazówka

Ogólnie rzecz biorąc, w kodzie możesz umieścić dowolną liczbę bloków `catch()`, jeden po drugim, w zależności od wyjątków, które chcesz obsługiwać za ich pomocą.

Blok `catch(...)` zaprezentowany w listingu 28.2 przechwytuje wszystkie typy wyjątków, które nie zostały wcześniej obsłużone przez inne bloki `catch`.

Użycie throw do zgłoszenia wyjątku określonego typu

Podczas przechwytywania wyjątku `std::bad_alloc` w listingu 28.2 naprawdę przechwytywany był obiekt klasy `std::bad_alloc` zgłoszony przez operator `new`. Istnieje możliwość zgłoszenia wyjątku wybranego typu. Do tego konieczne jest użycie polecenia `throw`:

```
void DowolnaMetoda()
{
   if(coś_niechcianego)
      throw Wartość;
}
```

Przeanalizujmy zastosowanie polecenia `throw` we własnym wyjątku, co zademonstrowano w listingu 28.3, którego kod przeprowadza operację dzielenia dwóch liczb.

Listing 28.3. Zgłoszenie własnego wyjątku w trakcie próby operacji dzielenia przez zero

```
 0: #include<iostream>
 1: using namespace std;
 2:
 3: double Divide(double Dividend, double Divisor)
 4: {
 5:    if(Divisor == 0)
 6:       throw "Dzielenie przez zero jest zabronione";
 7:
 8:    return (Dividend / Divisor);
 9: }
10:
11: int main()
12: {
13:    cout << "Podaj dzielną: ";
14:    double Dividend = 0;
15:    cin >> Dividend;
16:    cout << "Podaj dzielnik: ";
17:    double Divisor = 0;
18:    cin >> Divisor;
19:
20:    try
21:    {
22:       cout << "Wynik dzielenia: " << Divide(Dividend, Divisor);
23:    }
24:    catch(char* exp)
25:    {
```

```
26:       cout << "Wyjątek: " << exp << endl;
27:       cout << "Przepraszamy, nie można kontynuować pracy!" << endl;
28:    }
29:
30:    return 0;
31: }
```

Wynik ▼

```
Podaj dzielną: 2011
Podaj dzielnik: 0
Wyjątek: Dzielenie przez zero jest zabronione
Przepraszamy, nie można kontynuować pracy!
```

Analiza ▼

Powyższy kod demonstruje nie tylko możliwość przechwycenia wyjątków typu `char*`, jak widać w wierszu 24., ale również możliwość przechwycenia wyjątku zgłoszonego przez funkcję `Divide()` w wierszu 6. Zwróć uwagę, że w bloku `try{}` nie znajdują się wszystkie polecenia normalnie umieszczane w funkcji `main()`, a tylko te, które mają spowodować zgłoszenie wyjątku. To jest dobra praktyka, ponieważ obsługa wyjątków może zmniejszyć wydajność wykonywania kodu.

Jak działa obsługa wyjątków?

W listingu 28.3 następuje zgłoszenie nowego wyjątku typu `char*` w funkcji `Divide()`, który następnie jest przechwytywany przez blok `catch(char*)` w funkcji `main()`.

Kiedy wyjątek zostaje zgłoszony przy użyciu polecenia `throw`, kompilator wstawia kod odpowiedzialny za dynamiczne wyszukiwanie odpowiedniego bloku `catch(Type)` przeznaczonego do obsługi danego wyjątku. Logika obsługi wyjątku najpierw sprawdza, czy wiersz zgłaszający wyjątek znajduje się w bloku `try`. Jeżeli tak, następnie szukany jest odpowiedni blok `catch(Type)`, który potrafi obsłużyć wyjątek podanego typu. Gdy polecenie `throw` nie znajduje się w bloku `try` lub nie zostanie znaleziony odpowiedni blok `catch()` dla danego wyjątku, wtedy logika zgłoszenia wyjątku szuka tych samych danych w funkcji wywołującej. Dlatego też logika obsługi wyjątku „wspina” się po stosie, z jednej funkcji do kolejnej i szuka odpowiedniego bloku `catch(Type)`, który

będzie potrafił obsłużyć wyjątek. W trakcie każdego kroku w procedurze poruszania się po stosie zmienne lokalne funkcji są niszczone w kolejności odwrotnej do ich tworzenia. Przykład zaprezentowano w listingu 28.4.

Listing 28.4. Kolejność niszczenia obiektów lokalnych w przypadku zgłoszenia wyjątku

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: struct StructA
 4: {
 5:    StructA() {cout << "Utworzono strukturę A" << endl; }
 6:    ~StructA() {cout << "Zniszczono strukturę A" << endl;  }
 7: };
 8:
 9: struct StructB
10: {
11:    StructB() {cout << "Utworzono strukturę B" << endl; }
12:    ~StructB() {cout << "Zniszczono strukturę B" << endl; }
13: };
14:
15: void FuncB()   // Zgłoszenie wyjątku.
16: {
17:       cout << "W Func B" << endl;
18:       StructA objA;
19:       StructB objB;
20:       cout << "Nastąpi zgłoszenie wyjątku!" << endl;
21:       throw "A niech to...";
22: }
23:
24: void FuncA()
25: {
26:    try
27:    {
28:       cout << "W Func A" << endl;
29:       StructA objA;
30:       StructB objB;
31:       FuncB();
32:      cout << "FuncA: powrót do wywołującego" << endl;
33:    }
34:    catch(const char* exp)
35:    {
36:       cout << "FuncA: zgłoszono wyjątek, komunikat: " << exp << endl;
37:       cout << "FuncA: wyjątek został obsłużony, nie nastąpi zgłoszenie
          ↪wywołującemu" << endl;
38:       // throw;  // Usuń znak komentarza, aby zgłosić wyjątek do funkcji main().
39:    }
```

```
40: }
41:
42: int main()
43: {
44:    cout << "main(): rozpoczęcie działania" << endl;
45:    try
46:    {
47:       FuncA();
48:    }
49:    catch(const char* exp)
50:    {
51:       cout << "Wyjątek: " << exp << endl;
52:    }
53:    cout << "main(): eleganckie zakończenie działania" << endl;
54:    return 0;
55: }
```

Wynik ▼

```
main(): rozpoczęcie działania
W Func A
Utworzono strukturę  A
Utworzono strukturę B
W Func B
Utworzono strukturę A
Utworzono strukturę B
Nastąpi zgłoszenie wyjątku
Zniszczono strukturę B
Zniszczono strukturę A
Zniszczono strukturę B
Zniszczono strukturę A
FuncA: zgłoszono wyjątek, komunikat: A niech to...
FuncA: wyjątek został obsłużony, nie nastąpi zgłoszenie wywołującemu
main(): eleganckie zakończenie działania
```

Analiza ▼

W listingu 28.4 funkcja `main()` wywołuje funkcję `FuncA()` wywołującą `FuncB()`, która w wierszu 21. zgłasza wyjątek. Obie funkcje wywołujące, czyli `FuncA()` i `main()`, zapewniają bezpieczeństwo wyjątków, ponieważ zawierają zaimplementowany blok `catch(const char*)`. Funkcja `FuncB()` zgłaszająca wyjątek nie ma bloków `catch()`, a tym samym zdefiniowany w wierszach od 34. do 39. blok `catch()` w `FuncA()` jest pierwszą procedurą obsługi wyjątku zgłoszonego w `FuncB()`, ponieważ `FuncA()` jest funkcją wywołującą `FuncB()`.

Zwróć uwagę, że znajdująca się w `FuncA()` procedura obsługi tego wyjątku nie uznała wyjątku za niebezpieczny i nie przekazała go funkcji `main()`. Dlatego też działanie funkcji `main()` odbywa się normalnie, jakby nie wystąpił żaden problem. Jeżeli usuniesz znak komentarza na początku wiersza 38., wyjątek będzie zgłoszony wywołującemu `FuncB()`, czyli zostanie przekazany również funkcji `main()`.

Dane wyjściowe pokazują kolejność tworzenia obiektów (taka sama jak zdefiniowana w kodzie) oraz kolejność ich usuwania po zgłoszeniu wyjątku (to jest kolejność odwrotna do ich tworzenia). To zachodzi nie tylko w funkcji `FuncB()` zgłaszającej wyjątek, ale również w `FuncA()` wywołującej `FuncB()` i zajmującej się obsługą wyjątku.

Ostrzeżenie

W listingu 28.4 zademonstrowano wywoływanie destruktorów obiektów lokalnych po zgłoszeniu wyjątku.

Jeżeli destruktor obiektu wywołany na skutek wyjątku również zgłosi wyjątek, wtedy nastąpi nagłe i nieeleganckie zakończenie działania programu.

Wyjątki klasy std::exception

W trakcie przechwytywania wyjątku `std::bad_alloc` w listingu 28.2 następuje przechwycenie obiektu klasy `std::bad_alloc` zgłoszonego przez operator `new`. Wymieniona `std::bad_alloc` to klasa dziedzicząca po standardowej klasie C++ o nazwie `std::exception` zadeklarowanej w nagłówku `<exception>`.

Klasa `std::exception` jest klasą bazową dla wymienionych poniżej ważnych klas wyjątków:

- `bad_alloc` — wyjątek tej klasy jest zgłaszany, gdy alokacja pamięci przy użyciu operatora new zakończy się niepowodzeniem,
- bad_cast — wyjątek tej klasy jest zgłaszany przez dynamic_cast, gdy następuje próba rzutowania nieprawidłowego typu (czyli typu nieposiadającego relacji dziedziczenia),
- ios_base`::failure` — wyjątek tej klasy jest zgłaszany przez funkcje i metody w bibliotece `iostream`.

Klasa `std::exception` jest klasą bazową obsługującą bardzo użyteczną i zarazem ważną metodę wirtualną `what()` dostarczającą znacznie bardziej szczegółowy opis natury problemu, który doprowadził do zgłoszenia danego

wyjątku. W listingu 28.2 wywołanie `exp.what()` w wierszu 18. powoduje wygenerowanie komunikatu `bad allocation` informującego o przyczynie problemu. Klasę `std::exception` możesz wykorzystać jako klasę bazową dla wielu typów wyjątków oraz utworzyć jeden blok `catch(const exception&)`, który będzie przechwytywał wszystkie wyjątki klasy bazowej `std::exception`:

```
void SomeFunc()
{
   try
   {
      // Kod zapewniający bezpieczeństwo wyjątku.
   }
   catch (const std::exception& exp) // Przechwycenie bad_alloc, bad_cast itd.
   {
      cout << "Zgłoszono wyjątek: " << exp.what() << endl;
   }
}
```

Własna klasa wyjątku wywodząca się z std::exception

Istnieje możliwość zgłoszenia wyjątku dowolnego typu. Jednak zaletą jest utworzenie klasy dziedziczącej po `std::exception`, ponieważ wszystkie istniejące procedury obsługi wyjątków, takie jak `catch(const std::exception&)` i działające dla `bad_alloc`, `bad_cast` itd., zostaną automatycznie przeskalowane do przechwytywania wyjątków nowej klasy. Powód takiego zachowania jest prosty — mają wspólną klasę bazową. Takie rozwiązanie zaprezentowano w listingu 28.5.

Listing 28.5. Klasa CustomException, która dziedziczy po klasie std::exception

```
 0: #include <exception>
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: class CustomException: public std::exception
 6: {
 7:    string Reason;
 8: public:
 9:    // Konstruktor, wymaga Reason.
10:    CustomException(const char* why):Reason(why) {}
11:
```

```
12:     // Ponowna definicja funkcji wirtualnej, aby zwracała 'Reason'.
13:     virtual const char* what() const throw()
14:     {
15:        return Reason.c_str();
16:     }
17: };
18:
19: double Divide(double Dividend, double Divisor)
20: {
21:     if(Divisor == 0)
22:        throw CustomException("CustomException: Dzielenie przez zero jest
           ↪zabronione");
23:
24:     return (Dividend / Divisor);
25: }
26:
27: int main()
28: {
29:     cout << "Podaj dzielną: ";
30:     double Dividend = 0;
31:     cin >> Dividend;
32:     cout << "Podaj dzielnik: ";
33:     double Divisor = 0;
34:     cin >> Divisor;
35:     try
36:     {
37:        cout << "Wynik dzielenia: " << Divide(Dividend, Divisor);
38:     }
39:     catch(exception& exp) // Przechwycenie CustomException, bad_alloc itd.
40:     {
41:        cout << exp.what() << endl;
42:        cout << "Przepraszamy, nie można kontynuować pracy!" << endl;
43:     }
44:
45:     return 0;
46: }
```

Wynik ▼

```
Podaj dzielną: 2011
Podaj dzielnik: 0
Wyjątek: Dzielenie przez zero jest zabronione
Przepraszamy, nie można kontynuować pracy!
```

Analiza ▼

To jest wersja listingu 28.3 zgłaszającego prosty wyjątek typu `char*` na skutek operacji dzielenia przez zero. Jednak omawiana wersja powoduje utworzenie obiektu klasy `CustomException` zdefiniowanej w wierszach od 5. do 17. i dziedziczącej po klasie `std::exception`. Zauważ, że własna klasa wyjątku implementuje w wierszach od 13. do 16. funkcję wirtualną `what()`, która w zasadzie podaje powód zgłoszenia wyjątku. Zdefiniowana w wierszach od 39. do 43. logika `catch(exception&)` w funkcji `main()` obsługuje nie tylko wyjątki klasy `CustomException`, ale również inne wyjątki, np. `bad_alloc`, ponieważ mają one taką samą klasę bazową `std::exception`.

Uwaga

Zwróć uwagę na deklarację metody wirtualnej `CustomException::what()` w wierszu 13. listingu 28.5:

```
virtual const char* what() const throw()
```

Na końcu tej metody znajduje się wywołanie `throw()`, co oznacza, że ta metoda sama nie jest przeznaczona do zgłoszenia wyjątku. To bardzo ważne i istotne ograniczenie w klasie używanej jako obiekt wyjątku. Jeśli mimo to umieścisz polecenie `throw` w kodzie tej funkcji, możesz spodziewać się wyświetlenia ostrzeżenia przez kompilator.

Jeżeli funkcja kończy się wywołaniem `throw(int)`, oznacza to, że jest przygotowana do zgłaszania wyjątku typu `int`.

TAK	NIE
Pamiętaj o przechwytywaniu wyjątków typu std::exception.	Nie zgłaszaj wyjątków z poziomu destruktorów.
Pamiętaj, że jeśli tworzysz własną klasę obsługi wyjątków, najlepiej jest, aby dziedziczyła ona po klasie std::exception.	Nie przyjmuj założenia, że operacja alokacji pamięci zawsze zakończy się powodzeniem. Kod odpowiedzialny za obsługę operatora new zawsze powinien być bezpieczny i wykonywany w bloku try wraz z catch(std::exception&).
Pamiętaj o rozsądnym korzystaniu z wyjątków. Nie powinny być zamiennikami dla wartości zwrotnych, takich jak true i false.	W bloku catch() nie umieszczaj żadnej ogromnej logiki lub operacji alokacji zasobów. Nie chcesz przecież zgłoszenia wyjątku w trakcie obsługi innego zgłoszonego wyjątku.

Podsumowanie

W tej lekcji poznałeś bardzo ważny obszar praktycznego programowania w języku C++. Zapewnienie stabilności aplikacji poza własnym środowiskiem programistycznym jest bardzo ważne z punktu widzenia satysfakcji klienta i zapewnienia mu najlepszych wrażeń podczas używania aplikacji. Do tego celu wyjątki nadają się doskonale. Przekonałeś się, że działanie kodu odpowiedzialnego za alokację zasobów lub pamięci może zakończyć się niepowodzeniem i stąd konieczność bezpiecznego zgłoszenia wyjątku. Poznałeś klasę `std::exception` zapewniającą obsługę wyjątków w C++ i dowiedziałeś się, że podczas tworzenia własnej klasy wyjątków idealnym rozwiązaniem jest, aby dziedziczyła po wymienionej klasie.

Pytania i odpowiedzi

Pytanie: Dlaczego powinienem zgłaszać wyjątek zamiast zwracać błąd?

Odpowiedź: Nie zawsze masz przywilej zwrócenia błędu. Jeżeli wywołanie operatora `new` zakończy się niepowodzeniem, konieczne jest zapewnienie obsługi wyjątku, aby uniknąć awarii aplikacji. Jeśli ponadto błąd jest poważny i uniemożliwia dalsze działanie aplikacji, wtedy należy rozważyć zgłoszenie wyjątku.

Pytanie: Dlaczego moja klasa wyjątku powinna dziedziczyć po `std::exception`?

Odpowiedź: Oczywiście, nie jest to obowiązkowe, ale pomaga w ponownym użyciu bloków `catch()` przechwytujących wyjątki typu `std::exception`. Możesz utworzyć własną klasę wyjątku, która nie będzie dziedziczyła po żadnej innej, ale wtedy we wszystkich odpowiednich miejscach musisz umieścić nowe polecenia `catch(MyNewExceptionType&)`.

Pytanie: Mam funkcję zgłaszającą wyjątek. Czy wspomniany wyjątek musi być przechwytywany przez tę samą funkcję?

Odpowiedź: Niekoniecznie. Po prostu upewnij się, że zgłaszany typ wyjątku jest przechwytywany przez jedną z funkcji ze stosu funkcji.

Pytanie: Czy konstruktor może zgłosić wyjątek?

Odpowiedź: Tak naprawdę konstruktor nie ma wyjścia! Ponieważ konstruktor nie ma wartości zwrotnej, zgłoszenie wyjątku jest najlepszym sposobem zademonstrowania braku zgody na pewną operację.

Pytanie: Czy destruktor może zgłosić wyjątek?

Odpowiedź: Technicznie tak. To jednak jest niewłaściwa praktyka, ponieważ destruktory są wywoływane także, gdy stos pozostaje nietknięty na skutek wyjątku. Dlatego też destruktor wywołany wskutek wyjątku sam zgłosi wyjątek, co jest niekorzystne dla niestabilnej aplikacji, która próbuje elegancko zakończyć działanie.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Co to jest `std::exception`?
2. Jaki typ wyjątku zostanie zgłoszony, gdy alokacja pamięci przy użyciu operatora `new` zakończy się niepowodzeniem?
3. Czy blok procedury obsługi wyjątków (a dokładnie blok `catch`) jest odpowiednim miejscem na alokację miliona liczb całkowitych w celu utworzenia dla egzemplarza kopii istniejących danych?
4. Jak można przechwycić obiekt wyjątku klasy `MyException`, który dziedziczy po `std::exception`?

Ćwiczenia

1. **Łowcy błędów:** Wskaż błąd znajdujący się w poniższym kodzie:

```
class SomeIntelligentStuff
{
   bool StuffGoneBad;
public:
   ~ SomeIntelligentStuff()
   {
      if(StuffGoneBad)
         throw "Wystąpił duży problem w klasie";
   }
};
```

2. **Łowcy błędów:** Wskaż błąd znajdujący się w poniższym kodzie:

```
int main()
{
   int* pMillionIntegers = new int [1000000];
   // Operacja przeprowadzana na milionie liczb całkowitych.
   delete []pMillionIntegers;
}
```

3. **Łowcy błędów:** Wskaż błąd znajdujący się w poniższym kodzie:

```
int main()
{
   try
   {
      int* pMillionIntegers = new int [1000000];
      // Operacja przeprowadzana na milionie liczb całkowitych.
      delete []pMillionIntegers;
   }
   catch(exception& exp)
   {
      int* pAnotherMillionIntegers = new int [1000000];
      // Wykonanie kopii pMillionIntegers i zapisanie jej na dysku.
   }
}
```

Lekcja 29

Co dalej?

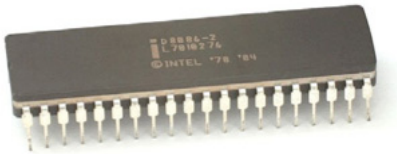
Poznałeś podstawy programowania w C++, a naprawdę wykroczyłeś poza teoretyczne granice podstaw użycia standardowej biblioteki wzorców (STL) czy innych wzorców, a już sama standardowa biblioteka może pomóc w tworzeniu efektywnego i zwięzłego kodu. Nadeszła pora, aby przyjrzeć się wydajności i najlepszym praktykom w programowaniu.

Z tej lekcji dowiesz się:

- **dowiesz się, czym różnią się obecnie stosowane procesory,**
- **dowiesz się, jak w Twojej aplikacji C++ najlepiej wykorzystać możliwości oferowane przez procesor,**
- **poznasz wątki i wielowątkowość,**
- **poznasz najlepsze praktyki stosowane podczas programowania w C++,**
- **dowiesz się, jak poprawić swoje umiejętności w zakresie programowania w C++.**

Czym wyróżniają się obecnie stosowane procesory?

Jeszcze do niedawna komputery stawały się coraz szybsze w wyniku stosowania w nich szybszych procesorów o częstotliwości zegara mierzonej w hercach (Hz), megahercach (MHz) lub gigahercach (GHz). Przykładowo pokazany na rysunku 29.1 procesor Intel 8086 to procesor 16-bitowy o częstotliwości zegara 10 MHz, wprowadzony na rynek w 1978 roku.



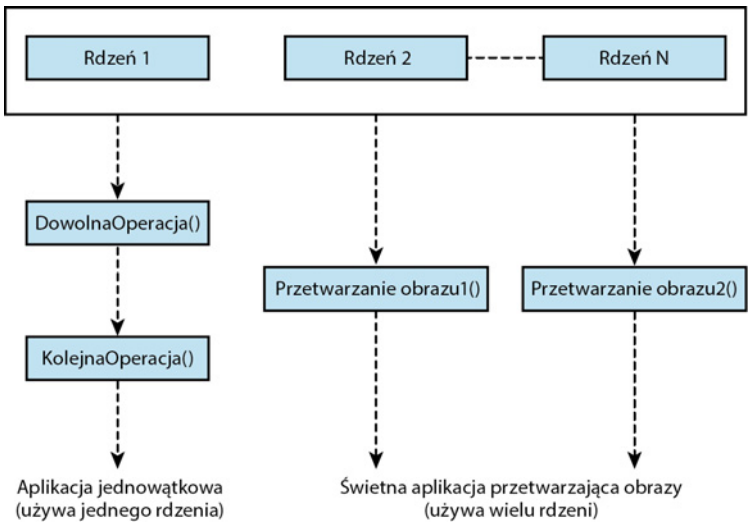
RYSUNEK 29.1. Mikroprocesor Intel 8086

W tamtym okresie, gdy procesor był znacznie szybszy od poprzednika, aplikacja utworzona w C++ również działała dużo szybciej. Wystarczyło więc poczekać na wprowadzenie procesorów o lepszej wydajności, aby dostrzec poprawę wydajności budowanej aplikacji. Wprawdzie obecnie wprowadzane procesory także są coraz szybsze, ale prawdziwa innowacja wynika z liczby używanych w nich rdzeni. W trakcie pisania tej książki firma Intel sprzedawała 64-bitowy mikroprocesor wyposażony w sześć rdzeni działających z częstotliwością zegara 3.2 GHz, a tendencją jest zwiększanie liczby rdzeni w procesorze. Na rysunku 29.2 pokazano przykład procesora obecnej generacji. Nawet smartfony są wyposażane w procesory wielordzeniowe.

RYSUNEK 29.2. Wielordzeniowy procesor firmy Intel

Procesor wielordzeniowy składa się z jednego fizycznego układu, w którym umieszczono wiele jednocześnie działających procesorów. Każdy z nich ma własną pamięć podręczną pierwszego poziomu (L1) oraz może działać niezależnie od pozostałych.

Logiczne jest, że szybszy procesor implikuje większą szybkość działania aplikacji. Możesz się jednak zastanawiać, jak liczba rdzeni procesora wpływa na wydajność aplikacji. Każdy rdzeń ma możliwość uruchomienia aplikacji, ale to przecież nie spowoduje jej szybszego działania. Dotychczas w tej książce przedstawiałem jednowątkowe aplikacje C++, a one nie potrafią wykorzystać pełni możliwości oferowanych przez procesory wielordzeniowe. Aplikacja działa w jednym wątku i dlatego jest nazywana jednowątkową, co pokazano na rysunku 29.3.



RYSUNEK 29.3. Jednowątkowa aplikacja w wielordzeniowym procesorze

Jeżeli aplikacja zawsze wykonuje szeregowo wszystkie operacje, wtedy system operacyjny prawdopodobnie przydziela jej tylko tyle czasu procesora, ile pozostałym aplikacjom w kolejce, a sam program działa tylko w jednym rdzeniu procesora. Innymi słowy, aplikacja działa w procesorze wielordzeniowym w taki sam sposób jak wiele lat temu.

Jak lepiej używać wielu rdzeni?

Kluczem jest wielowątkowość. Wszystkie wątki działają równocześnie, co pozwala systemowi operacyjnemu na ich uruchamianie w wielu rdzeniach. Wprawdzie dokładne omówienie działania wielowątkowego wykracza poza ramy tej książki, ale spróbuję przybliżyć temat i pokazać, jak tworzyć aplikacje o wysokiej wydajności działania.

Czym jest wątek?

Kod aplikacji zawsze działa w wątku. Wątek to synchronicznie wykonywana jednostka kodu, a polecenia w wątku są wykonywane kolejno, jedno po drugim. Kod znajdujący się w funkcji `main()` jest uznawany za wykonywany w wątku głównym aplikacji. We wspomnianym wątku głównym można tworzyć nowe wątki, działające równolegle z pozostałymi. Tego rodzaju aplikacje składające się z wątków działających jednocześnie obok wątku głównego są nazywane aplikacjami wielowątkowymi.

System operacyjny określa sposób tworzenia wątków, natomiast programista może je tworzyć bezpośrednio przy użyciu API dostarczanego przez system operacyjny.

Wskazówka

Standard C++11 definiuje funkcje wątków, które zajmują się za programistę odpowiednimi wywołaniami API, co powoduje, że tworzona aplikacja wielowątkowa jest jeszcze bardziej przenośna.

Jeżeli planujesz utworzenie aplikacji tylko dla jednego systemu operacyjnego, sprawdź w danym systemie operacyjnym wywołania API pozwalające na tworzenie aplikacji wielowątkowych.

Uwaga

Rzeczywista operacja utworzenia wątku to funkcja systemu operacyjnego. W standardzie C++11 podjęto próbę dostarczenia niezależnego od platformy mechanizmu w postaci klasy `std::thread` zdefiniowanej w nagłówku `<thread>`.

Na razie większość kompilatorów niezbyt dobrze obsługuje tego rodzaju rozwiązanie. Poza tym, jeśli tworzysz aplikację dla jednej platformy, najlepszym rozwiązaniem jest wykorzystanie funkcji wątków oferowanych przez dany system operacyjny.

Jeżeli w przenośnej aplikacji C++ potrzebujesz wątków, zainteresuj się bibliotekami Boost Thread Libraries, które znajdziesz w witrynie *http://www.boost.org/*.

Dlaczego należy tworzyć aplikacje wielowątkowe?

Wielowątkowość jest stosowana w aplikacjach wymagających jednoczesnego wykonywania wielu sesji określonej działalności. Wyobraź sobie, że jesteś jednym z 10 tysięcy użytkowników próbujących zakupić coś w sklepie internetowym. Oczywiście, taki sklep nie może pozwolić sobie, aby 9999 użytkowników czekało na możliwość wykonania zakupów. Dlatego też serwer

WWW tworzy wiele wątków w celu jednoczesnej obsługi jak największej liczby użytkowników. Jeśli serwer WWW działa w procesorze wielordzeniowym (a zwykle tak jest), wątki wykorzystują maksymalną liczbę rdzeni, aby zapewnić użytkownikowi optymalną wydajność działania aplikacji sieciowej.

Inny, często spotykany przykład aplikacji wielowątkowej to program wykonujący pewną operację (np. wyświetlenie paska postępu) oprócz prowadzenia interakcji z użytkownikiem. Tego rodzaju aplikacje zwykle składają się z wątku interfejsu użytkownika wyświetlającego i uaktualniającego interfejs użytkownika oraz akceptującego dane wejściowe od użytkownika, a także z wątków roboczych wykonujących operacje w tle. Do tego typu aplikacji można zaliczyć narzędzia przeprowadzające defragmentację dysku twardego. Po naciśnięciu przycisku rozpoczynającego pracę następuje utworzenie wątku roboczego, który zaczyna operację skanowania i defragmentacji. W tym samym czasie wątek interfejsu użytkownika wyświetla informacje o postępie operacji i daje możliwość jej przerwania. Zwróć uwagę na pewien fakt: aby wątek interfejsu użytkownika mógł wyświetlać informacje o postępie aplikacji, musi regularnie pobierać tego rodzaju informacje od wątku roboczego. Aby — podobnie — wątek roboczy mógł przerwać działanie na żądanie użytkownika, wątek interfejsu użytkownika musi mu dostarczyć odpowiednie polecenie.

Uwaga

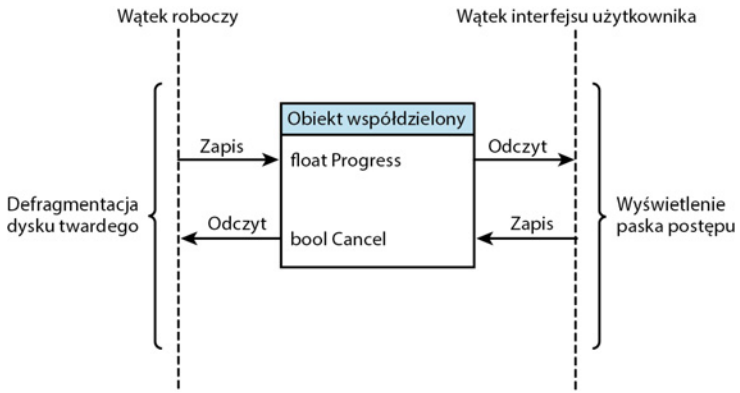
W aplikacjach wielowątkowych poszczególne wątki często muszą się komunikować, aby umożliwić aplikacjom działanie jako całości, a nie kolekcji wątków wykonujących operacje zupełnie niezależnie od pozostałych.

Kolejność wykonywania operacji również jest ważna. Nie chcesz przecież, aby wątek interfejsu użytkownika zakończył działanie, zanim wątek roboczy skończy defragmentację dysku twardego. Zdarzają się sytuacje, w których jeden wątek musi zaczekać na zakończenie działania w innym wątku. Przykładowo wątek odczytujący dane z bazy danych powinien zaczekać, aż pracę zakończy wątek zapisujący dane w bazie danych.

Proces oczekiwania wątku na zakończenie działania innego nosi nazwę *synchronizacji wątku*.

Jak wątki wymieniają dane?

Wątki mogą współdzielić zmienne i mają dostęp do danych globalnych. Wątek można utworzyć wraz ze wskaźnikiem do obiektu współdzielonego (struktury lub klasy) zawierającego dane, co pokazano na rysunku 29.4.



RYSUNEK 29.4. Wątki interfejsu użytkownika i roboczy współdzielą dane

Poszczególne wątki mogą się komunikować przez uzyskanie dostępu do danych lub zapis danych przechowywanych w położeniu w pamięci, do którego mają dostęp wszystkie wątki, co powoduje, że to miejsce jest współdzielone. W przykładzie aplikacji defragmentacji dysku, gdzie wątek roboczy zna postęp operacji, a wątek interfejsu użytkownika musi otrzymać te informacje, wątek roboczy może nieustannie przechowywać informacje o postępie jako wartość procentową w zmiennej typu liczba całkowita, a wątek interfejsu użytkownika używać tej zmiennej do wyświetlenia informacji o postępie operacji.

To jest bardzo prosty przykład: jeden wątek tworzy informacje, natomiast drugi je wykorzystuje. Co się stanie, jeśli wiele wątków jednocześnie będzie zapisywało lub odczytywało informacje z tego samego położenia w pamięci? Pewne wątki mogą rozpocząć odczyt danych, zanim inne wątki zakończą zapis danych. Taka sytuacja może doprowadzić do uszkodzenia danych.

Teraz już wiesz, dlaczego ważne jest zapewnienie synchronizacji wątków.

Użycie muteksu i semaforów do synchronizacji wątków

Wątki są jednostkami systemu operacyjnego, więc obiekty używane do ich synchronizacji są dostarczane przez system operacyjny. Większość systemów operacyjnych zapewnia semafory i muteksy pozwalające na przeprowadzanie synchronizacji wątków.

Za pomocą muteksu gwarantujesz, że tylko jeden wątek ma dostęp do określonego fragmentu kodu w danym czasie. Innymi słowy, mutekst jest używany do wskazania sekcji kodu, na wykonanie którego wątek musi

poczekać, aż inny wątek zakończy jego wykonywanie i zwolni muteks. Kolejny wątek przechwytuje muteks, wykonuje swoją operację, a następnie zwalnia muteks.

Przy użyciu semaforów można kontrolować liczbę wątków wykonujących sekcję kodu. Semafor pozwalający tylko jednemu wątkowi jednocześnie na uzyskanie dostępu do sekcji kodu jest nazywany semaforem binarnym.

Uwaga

Do dyspozycji programisty, w zależności od systemu operacyjnego, może istnieć więcej obiektów synchronizacji wątków. Przykładowo w systemie Windows znajdują się sekcje o znaczeniu krytycznym, które mogą być wykonywane tylko przez jeden wątek w danym czasie.

Problemy powodowane przez wielowątkowość

Wielowątkowość wraz z koniecznością zapewnienia dobrej synchronizacji wątków może również być przyczyną wielu nieprzespanych nocy, gdy synchronizacja nie będzie efektywna (czytaj: zawiera błędy). Oto dwa problemy najczęściej występujące w aplikacjach wielowątkowych.

- **Stan wyścigu** — dwa wątki lub więcej wątków próbuje zapisać te same dane. Kto wygra? W jaki stanie będzie znajdował się obiekt?
- **Zakleszczenie** — dwa wątki czekają na zakończenie działania przez ten drugi wątek. Oba znajdują się więc w stanie „oczekiwania", a działanie aplikacji jest wstrzymane.

Dobra synchronizacja pozwala na uniknięcie sytuacji wyścigu. Ogólnie rzecz biorąc, gdy wątki mają możliwość przeprowadzania zapisu w obiekcie współdzielonym, trzeba podjąć dodatkowe środki ostrożności i zagwarantować, że:

- tylko jeden wątek w danym czasie przeprowadza operację zapisu,
- żaden wątek nie może odczytywać obiektu, aż do chwili zakończenia operacji przez wątek zapisujący dane.

Zakleszczeń można uniknąć, gdy zagwarantuje się, że w żadnej sytuacji dwa wątki nie będą oczekiwały na zakończenie operacji wykonywanej przez ten drugi wątek. Możesz zdefiniować wątek główny synchronizujący wątki robocze lub też utworzyć program w taki sposób, aby zadania były rozproszone między

wątkami i powstał jasny podział wykorzystania wątków. Wątek A może czekać na B, ale B nigdy nie powinien czekać na A.

Tworzenie aplikacji wielowątkowych samo w sobie jest sztuką. Dlatego ten temat wykracza poza zakres tej książki, ponieważ na tych kilku stronach nie da się go szczegółowo wyjaśnić w interesujący i ekscytujący sposób. Jednak w internecie znajdziesz szczegółową dokumentację poświęconą temu tematowi, a ponadto programowania wielowątkowego możesz nauczyć się, po prostu tworząc aplikacje. Po opanowaniu tej sztuki Twoje aplikacje C++ będą zapewniały optymalną wydajność działania w procesorach wielowątkowych, które zostaną opracowane w przyszłości.

Tworzenie doskonałego kodu C++

Język C++ nie tylko ewoluuje od początku istnienia, ale podejmowane przez twórców najważniejszych kompilatorów wysiłki na rzecz jego standaryzacji i dostępności narzędzi oraz funkcji pomagają programistom w tworzeniu zwięzłego i czystego kodu C++. Tworzenie czytelnych i niezawodnych aplikacji C++ naprawdę jest bardzo łatwe.

Poniżej wymieniono krótką listę najlepszych praktyk, które pomogą w tworzeniu dobrych aplikacji C++.

- Zmiennym nadawaj nazwy, które są sensowne zarówno dla Ciebie, jak i innych pracujących nad danym kodem. Warto poświęcić sekundę lub dwie i nadawać zmiennym lepsze nazwy.
- Zawsze inicjalizuj zmienne, takie jak `int`, `float` itd.
- Zawsze inicjalizuj wartości wskaźników — **`null`** albo poprawny adres, np. zwrócony przez operatora `new`.
- Podczas tworzenia tablic nigdy nie przekraczaj granic bufora tablicy. To nosi nazwę przepełnienia bufora i może stanowić lukę w zabezpieczeniach.
- Nie używaj buforów ciągów tekstowych w stylu C (`char*`) lub funkcji, takich jak `strlen()` i `strcpy()`. Klasa `std::string` to bezpieczniejsze rozwiązanie, a ponadto dostarcza wiele metod, m.in. pozwalających na określanie długości ciągu tekstowego, jego kopiowanie, dołączanie danych itd.

- Tablic statycznych używaj tylko wtedy, gdy znasz liczbę przechowywanych w nich elementów. Jeśli nie jesteś pewien liczby elementów, lepiej wybierz tablicę dynamiczną, taką jak `std::vector`.
- Podczas deklarowania i definiowania funkcji pobierających dane wejściowe typów innych niż POD (ang. *Plain Old Data*), rozważ deklarację parametrów jako referencji, aby w ten sposób uniknąć niepotrzebnego kroku kopiowania w trakcie wywołania funkcji.
- Jeżeli klasa zawiera zwykłe wskaźniki (lub elementy składowe), zastanów się, jak powinno wyglądać zarządzanie własnością zasobu w pamięci na wypadek jego skopiowania lub przypisania. Powinieneś rozważyć utworzenie konstruktora kopiującego i kopiującego operatora przypisania.
- Podczas budowania klasy narzędziowej zarządzającej tablicą dynamiczną lub podobną konstrukcją pamiętaj o utworzeniu konstruktora przenoszącego i przenoszącego operatora przypisania, co zapewni lepszą wydajność działania.
- Pamiętaj o prawidłowym stosowaniu słowa kluczowego `const` w kodzie. Idealnie będzie, jeśli funkcja `get()` zostanie pozbawiona możliwości modyfikacji elementów składowych klasy, a więc powinna być typu `const`. Podobnie, parametry funkcji powinny być referencjami typu `const`, o ile nie chcesz zmieniać przechowywanych przez nie wartości.
- Unikaj stosowania zwykłych wskaźników. Jeśli istnieje możliwość, wybieraj odpowiedni sprytny wskaźnik.
- Podczas tworzenia klasy narzędziowej zdobądź się na wysiłek i przygotuj wszystkie wymagane operatory, które ułatwią używanie tej klasy.
- Mając taką możliwość, wybieraj wersję wzorca zamiast makro. Wzorce zapewniają bezpieczeństwo typów i mają ogólną postać.
- Jeżeli tworzona klasa ma być obiektem w kontenerze, takim jak wektor lub lista, bądź ma być używana jako element kluczowy w mapie, pamiętaj o zapewnieniu obsługi operatora `<`, co pomoże w zdefiniowaniu domyślnych kryteriów sortowania.
- Jeśli funkcja lambda stanie się zbyt wielka, rozważ utworzenie zamiast niej obiektu funkcji, czyli klasy wraz z operatorem `()`, ponieważ funktor można wielokrotnie wykorzystać, a sam kod będziesz musiał obsługiwać tylko w jednym miejscu.

- Nigdy nie przyjmuj założenia, że działanie operatora `new` musi zakończyć się powodzeniem. Kod przeprowadzający alokację zasobów zawsze powinien zapewniać bezpieczeństwo wyjątków — umieść go w bloku `try` wraz odpowiednimi blokami `catch()`.
- Nigdy nie wywołuj polecenia `throw` z poziomu destruktora klasy.

To na pewno nie jest pełna lista, ale zawiera najważniejsze punkty, których przestrzeganie pomoże w tworzeniu dobrego i łatwego w obsłudze kodu C++.

Ucz się C++. Nie poprzestań na tym, czego się tu dowiedziałeś!

Gratulacje, poczyniłeś ogromne postępy w nauce języka C++. Najlepszym sposobem na kontynuację nauki jest tworzenie coraz większej ilości kodu.

C++ to złożony język programowania. Im więcej będziesz tworzył kodu, tym lepiej zrozumiesz język i sposób jego działania. Środowiska programistyczne, takie jak Visual Studio wraz z funkcją listy typu Intellisense, pomagają w tworzeniu kodu i zaspokajają ciekawość, np. wyświetlają elementy składowe klasy, których dotąd nie znałeś. Czas na kontynuację nauki przez ćwiczenia praktyczne!

Dokumentacja w internecie

Jeżeli chcesz dowiedzieć się więcej na temat sygnatur kontenerów STL, ich metod, algorytmów i szczegółów funkcjonowania, odwiedź witrynę MSDN (*http://msdn.microsoft.com/pl-pl/*), w której znajdziesz wiele użytecznych informacji na temat standardowej biblioteki wzorców (STL).

Wskazówka
Wskazówka

> W trakcie lektury dokumentacji STL w witrynie MSDN pamiętaj o wybraniu odpowiedniej wersji Visual Studio, ponieważ obsługa standardu C++11 jest oferowana jedynie przez wydania Visual Studio 2010 i nowsze.

Gdy pisałem tę książkę, większość kompilatorów C++ nie oferowała pełnej obsługi wszystkich funkcji wprowadzonych w C++11. Przykładowo Visual Studio 2010 nie zawierało obsługi wzorców o zmiennej liczbie argumentów. Z kolei kompilator GNU GCC w wersji 4.6 miał nieprawidłowo działającą implementację `std::thread`. Ogólnie dobrym pomysłem jest przejrzenie dokumentacji kompilatora i sprawdzenie, które funkcje standardu C++11

są obsługiwane lub będą wkrótce. Zespół Visual Studio pod adresem *http://blogs.msdn.com/b/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx* umieścił wpis bloga zatytułowany „C++11 Core Language Feature Support", natomiast tego rodzaju strona dla kompilatora GCC znajduje się pod adresem *http://gcc.gnu.org/projects/cxx0x.html*.

> Zwróć uwagę, że w trakcie pisania tej książki Visual Studio i GCC były dwoma ważnymi kompilatorami, które zapewniały obsługę sporej liczby funkcji zalecanych przez standard C++11. Możesz być spokojny, wiedząc, że kod zaprezentowany w książce został przetestowany w obu wymienionych kompilatorach.

Uwaga

Społeczności, w których możesz uzyskać porady i pomoc

Język C++ ma bogate i tętniące życiem społeczności w internecie. Zarejestruj się w witrynach, takich jak CodeGuru (*http://www.codeguru.com/*) i CodeProject (*http://www.codeproject.com/*), a zyskasz szansę, że na zadane przez Ciebie pytanie otrzymasz odpowiedź.

Kiedy będziesz czuł się na siłach, sam możesz wnieść swój wkład w wymienione społeczności. Przekonasz się, że udzielanie odpowiedzi na wymagające pytania i nauka mają wiele wspólnego.

Podsumowanie

Ta lekcja kończąca książkę to pierwszy krok na Twojej drodze ku lepszemu poznaniu języka C++! Skoro dotarłeś tak daleko, zdążyłeś już poznać podstawy i bardziej zaawansowane koncepcje języka. W tej lekcji poznałeś teoretyczne podstawy programowania wielowątkowego. Dowiedziałeś się, że jedynym sposobem na maksymalne wykorzystanie procesorów wielordzeniowych jest umieszczenie logiki w wątkach i zastosowanie przetwarzania równoległego. Poznałeś pułapki czyhające na programistów tworzących aplikacje wielowątkowe i wiesz, jak ich unikać. Wreszcie poznałeś pewne, najważniejsze i najlepsze, praktyki w programowaniu C++. Wiesz, że tworzenie dobrego kodu C++ to nie tylko stosowanie zaawansowanych koncepcji, ale również nadawanie zmiennym nazw zrozumiałych dla innych programistów, obsługa wyjątków, używanie klas narzędziowych, takich jak sprytne wskaźniki zamiast zwykłych, itd. Bez wątpienia jesteś gotowy, aby wkroczyć do świata profesjonalnego programowania w języku C++.

Pytania i odpowiedzi

Pytanie: Jestem całkiem zadowolony z wydajności działania mojej aplikacji? Czy mimo wszystko powinienem implementować możliwości wielowątkowe?

Odpowiedź: Nic podobnego. Nie wszystkie aplikacje muszą być wielowątkowe. Wielowątkowość jest zalecana przede wszystkim w aplikacjach, które muszą jednocześnie wykonywać wiele zadań lub służyć wielu użytkownikom.

Pytanie: Najważniejsze kompilatory nie oferują pełnej obsługi standardu C++11. Dlaczego więc nie powinienem stosować programowania w starym stylu?

Odpowiedź: Przede wszystkim, dwa najważniejsze kompilatory (Microsoft Visual C++ i GNU GCC) oferują obsługę większości ważnych funkcji C++11 i nie obsługują tylko niewielkiej części funkcji wprowadzonych w standardzie C++11. Ponadto standard C++11 znacznie ułatwia programowanie. Słowa kluczowe, takie jak `auto`, oszczędzają konieczności żmudnego tworzenia długich deklaracji iteratorów, a funkcje lambda pozwalają na zmniejszenie kodu konstrukcji `for_each()` i eliminację z niej obiektu funkcji. Korzyści wynikające ze stosowania standardu C++11 powinny być więc oczywiste.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D.

Quiz

1. Moja aplikacja przetwarzania obrazów nie reaguje podczas korekcji kontrastu. Co powinienem zrobić w tej sytuacji?
2. Moja wielowątkowa aplikacja pozwala na wyjątkowo szybki dostęp do bazy danych. Jednak czasami pobrane dane są uszkodzone. Co robię źle?

Dodatki

Dodatek A

Praca z liczbami: dwójkowo i szesnastkowo

Zrozumienie sposobu działania systemów liczb dwójkowych i szesnastkowych ma znaczenie krytyczne nie tylko w tworzeniu lepszych aplikacji C++, ale pozwala także na jeszcze lepsze poznanie wewnętrznego sposobu działania aplikacji.

System liczb dziesiętnych

Numery, którymi codziennie się posługujemy, mieszczą się w zakresie od 0 do 9. Ten zestaw numerów nosi nazwę systemu liczb dziesiętnych. Ponieważ system składa się z dziesięciu unikalnych cyfr, jego podstawą jest właśnie dziesiątka.

Skoro podstawą systemu jest dziesiątka, liczone od zera położenie każdej kolejnej cyfry określa potęgę 10 dla danej cyfry:

$957 = 9 \times 10^2 + 5 \times 10^1 + 7 \times 10^0 = 9 \times 100 + 5 \times 10 + 7$

W liczbie 957 liczona od zera pozycja cyfry 7 to 0, cyfry 5 to 1, natomiast cyfry 9 to 2. Wymienione indeksy położenia stają się potęgami liczby 10, co przedstawiono w przykładzie. Pamiętaj, że dowolna liczba podniesiona do potęgi 0 daje 1, czyli $10^0 = 1000^0 = 1$.

Uwaga

W systemie dziesiętnym potęgi liczby 10 są najważniejsze. Cyfry w liczbach są mnożone przez 10, 100, 1000 itd. w celu określenia rzędu wielkości liczby.

System liczb dwójkowych

System liczb o podstawie 2 jest nazywany systemem dwójkowym. Ponieważ pozwala na stosowanie jedynie dwóch stanów, jest reprezentowany przez liczby 0 i 1. Wymienione liczby w C++ zwykle przyjmują wartość `false` i `true` (`true` to wartość niezerowa).

Podobnie jak liczby w systemie dziesiętnym są obliczane jako potęgi podstawy 10, w systemie dwójkowym podstawą potęg jest 2.

$101\ (\text{dwójkowo}) = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5\ (\text{dziesiętnie})$

Tak więc dziesiętnym odpowiednikiem liczby dwójkowej 101 jest 5.

Uwaga

Cyfry w liczbie dwójkowej są mnożone przez potęgi liczby 2, np. 4, 8, 16, 32 itd. w celu określenia rzędu wielkości liczby. Potęga jest określana przez liczone od zera położenie danej cyfry w liczbie.

Aby lepiej zrozumieć system liczb dwójkowych, przeanalizuj tabelę A.1, w której wymieniono różne potęgi liczby 2.

Tabela A.1. Potęgi liczby 2

Potęga	Wartość	Postać dwójkowa
0	$2^0 = 1$	1
1	$2^1 = 2$	10
2	$2^2 = 4$	100
3	$2^3 = 8$	1000
4	$2^4 = 16$	10000
5	$2^5 = 32$	100000
6	$2^6 = 64$	1000000
7	$2^7 = 128$	10000000

Dlaczego w komputerach używany jest system dwójkowy?

System dwójkowy jest względnie nowy i stosowany przede wszystkim przez producentów elektroniki i osprzętu komputerowego. Ewolucja elektroniki i komponentów elektronicznych spowodowała opracowanie systemu określającego stan komponentu jako WŁĄCZONY (duża potencjalna różnica w napięciu) lub WYŁĄCZONY (brak napięcia lub mała potencjalna różnica w napięciu).

Stany WŁĄCZONY i WYŁĄCZONY są wygodnie interpretowane jako 1 i 0, w pełni przedstawiają liczbę dwójkową i stały się wybieraną metodą do przeprowadzania obliczeń arytmetycznych. Omówione w lekcji 5., zatytułowanej „Wyrażenia, instrukcje i operatory", operacje logiczne, takie jak NOT, AND, OR i XOR, były łatwo obsługiwane przez bramy elektroniczne. To spowodowało, że system dwójkowy wszedł w powszechne użycie, gdy przetwarzanie warunkowe stało się łatwiejsze.

Czym są bity i bajty?

Bit to podstawowa jednostka w systemie komputerowym i zawiera stan dwójkowy. Mówi się, że bit jest „ustawiony", gdy zawiera 1, lub „wyzerowany", gdy zawiera 0. Zestaw bitów tworzy bajt. Teoretycznie liczba bitów w bajcie nie jest stała i zależy od sprzętu komputerowego.

Jednak w większości systemów komputerowych przyjęto założenie, że bajt tworzy osiem bitów — dla prostoty i wygody wybrano 8, ponieważ to potęga liczby 2. Osiem bitów w bajcie pozwala również na transmisję maksymalnie 2^8 różnych wartości (czyli 255 wartości). Te 255 różnych wartości wystarcza do wyświetlenia lub transakcji wszystkich znaków w zbiorze ASCII.

Ile bajtów jest w kilobajcie?

1024 bajty (2^{10} bajtów) tworzą kilobajt. Podobnie 1024 kilobajty tworzą megabajt, 1024 megabajty to gigabajt, natomiast 1024 gigabajty to terabajt.

System liczb szesnastkowych

W systemie liczb szesnastkowych stosowana jest podstawa wynosząca 16. Cyfra w systemie szesnastkowym może być z zakresu od 0 do 9 i od A do F. Dlatego też dziesiętne 10 to szesnastkowe A, natomiast dziesiętne 15 to szesnastkowe F (patrz tabela A.2).

Tabela A.2. Cyfry w systemie szesnastkowym

Dziesiętnie	Szesnastkowo	Dziesiętnie	Szesnastkowo
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

Podobnie jak liczby w systemie dziesiętnym są określane jako potęgi podstawy 10, w systemie dwójkowym to potęgi podstawy 2, natomiast w systemie szesnastkowym to potęgi podstawy 16:

$0x31F = 3 \times 16^2 + 1 \times 16^1 + F \times 16^0 = 3 \times 256 + 16 + 15 = 799$ (dziesiętnie)

Uwaga

Według konwencji liczby szesnastkowe są poprzedzane prefiksem 0x.

Dlaczego potrzebujemy systemu szesnastkowego?

Komputery działają w systemie dwójkowym. Stan każdej jednostki pamięci w komputerze można określić jako 0 lub 1. Jeśli jednak człowiek pracuje z komputerem bądź też programuje pewne informacje jako ciąg zer i jedynek, potrzebuje ogromnej ilości przestrzeni na wprowadzenie niewielkiej ilości informacji. Dlatego też zamiast zapisać 1111 dwójkowo, znacznie efektywniej zapiszemy F szesnastkowo.

Jasno więc widać, że postać szesnastkowa pozwala na bardzo efektywne przedstawienie stanu 4 bitów w cyfrze, a użycie maksymalnie dwóch cyfr szesnastkowych umożliwia przedstawienie stanu bajta.

Uwaga

Rzadziej używanym systemem liczb jest system ósemkowy. To jest system o podstawie 8 i wykorzystuje liczby od 0 do 7.

Konwersja na inną podstawę

Podczas pracy z liczbami od czasu do czasu musimy przedstawić tę samą liczbę w innej postaci, np. wartości liczby dwójkowej w postaci dziesiętnej lub liczby dziesiętnej jako szesnastkowej.

W poprzednich przykładach przekonałeś się, jak liczby mogą być konwertowane z postaci dwójkowej lub szesnastkowej na dziesiętną. Spójrz teraz na konwersję liczb dwójkowych i szesnastkowych na dziesiętne.

Ogólny proces konwersji

Podczas konwersji liczby pomiędzy różnymi podstawami następuje kolejne dzielenie podstawy, począwszy od liczby przeznaczonej do konwersji. Każda reszta z dzielenia wypełnia miejsca w docelowym systemie liczbowym, począwszy od miejsca najmniejszego. Kolejna operacja dzielenia używa ilorazu poprzedniej operacji dzielenia wraz z podstawą jako dzielnikiem.

Proces jest kontynuowany do chwili, gdy iloraz w docelowym systemie liczbowym wynosi zero, i nosi nazwę *procesu rozbioru*.

Konwersja liczby dziesiętnej na dwójkową

Aby skonwertować liczbę dziesiętną 33 na postać dwójkową, należy odjąć najwyższą możliwą potęgę liczby 2 (tutaj to 32):

```
Miejsce 1: 33/2 = iloraz 16, reszta 1
Miejsce 2: 16 / 2 = iloraz 8, reszta 0
Miejsce 3: 8/2 = iloraz 4, reszta 0
Miejsce 4: 4/2 = iloraz 2, reszta 0
Miejsce 5: 2/2 = iloraz 1, reszta 0
Miejsce 6: 1/2 = iloraz 0, reszta 1
```

Dwójkowy odpowiednik liczby 33 (odczytując wartość reszty w miejscach) to 100001.

Podobnie, dwójkowy odpowiednik liczby 156 to:

```
Miejsce 1: 156/2 = iloraz 78, reszta 1
Miejsce 2: 78/2 = iloraz 39, reszta 0
Miejsce 3: 39/2 = iloraz 19, reszta 1
Miejsce 4: 19/2 = iloraz 9, reszta 1
Miejsce 5: 9/2 = iloraz 4, reszta 1
Miejsce 6: 4/2 = iloraz 2, reszta 0
Miejsce 7: 2/2 = iloraz 1, reszta 0
Miejsce 8: 1/0 = iloraz 0, reszta 1
```

Tak więc dwójkowy odpowiednik liczby 156 to 10011100.

Konwersja liczby dziesiętnej na szesnastkową

Proces przebiega tak samo jak w przypadku liczby dwójkowej, ale dzielenie odbywa się przez podstawę 16 zamiast 2.

Skonwertowanie liczby dziesiętnej 5211 na szesnastkową:

```
Miejsce 1: 5211/16 = iloraz 325, reszta B (11 w systemie dziesiętnym
to B w szesnastkowym)
Miejsce 2: 325/16 = iloraz 20, reszta 5
Miejsce 3: 20/16 = iloraz 1, reszta 4
Miejsce 4: 1/16 = iloraz 0, reszta 1
```

Liczba 5211 w systemie dziesiętnym to 145B w systemie szesnastkowym.

Wskazówka

> Aby jeszcze lepiej zrozumieć działanie różnych systemów liczb, możesz utworzyć w C++ prosty program, podobny do przedstawionego w listingu 27.1 w lekcji 27., zatytułowanej „Użycie strumieni w operacjach wejścia-wyjścia". Program w wymienionym listingu wykorzystuje `std::cout` wraz z różnymi manipulatorami do wyświetlania liczb całkowitych w systemach o podstawie szesnastkowej, dziesiętnej i ósemkowej.

W celu wyświetlenia liczby całkowitej w formacie dwójkowym użyj klasy `std::bitset` omówionej w lekcji 25., zatytułowanej „Praca z opcjami bitowymi za pomocą STL", i zaczerpnij inspirację z listingu 25.1.

Dodatek B

Słowa kluczowe C++

Słowa kluczowe to zarezerwowane przez kompilator symbole języka. Nie można ich używać jako nazw klas, zmiennych czy funkcji.

```
asm                 false               signed
auto                final               sizeof
bool                float               static
break               for                 static_assert
case                friend              static_cast
catch               goto                struct
char                if                  switch
class               inline              template
const               int                 this
constexpr           long                throw
const_cast          long long int       true
continue            mutable             try
decltype            namespace           typedef
default             new                 typeid
delete              operator            typename
do                  override            union
double              private             unsigned
dynamic_cast        protected           using
else                public              virtual
enum                register            void
explicit            reinterpret_cast    volatile
export              return              wchar_t
extern              short               while
```

Oprócz wymienionych, zarezerwowane są jeszcze następujące słowa:

```
and                 compl               or_eq
and_eq              not                 xor
bitand              not_eq              xor_eq
bitor               or
```

Dodatek C

Kolejność operatorów

Dobrą praktyką jest używanie nawiasów, które wyraźnie dzielą operację na poszczególne fragmenty. Kiedy brakuje nawiasów, kompilator stosuje wbudowaną kolejność, w jakiej stosowane są operatory. Wspomniana kolejność operatorów (wymieniona w tabeli C.1) jest używana przez kompilator w przypadku jakichkolwiek niejasności.

Tabela C.1. Priorytety operatorów

Pozycja	Nazwa	Operator
1	operator zakresu	`::`

2	wybór składowych, indeksowanie, wywołania funkcji, inkrementacja i dekrementacja postfiksowa	`. ->`
		`()`
		`++ --`

3	`sizeof`, inkrementacja i dekrementacja prefiksowa, negacja, `and`, `not`, jednoargumentowy minus i plus, adres i wyłuskanie, `new`, `new[]`, `delete`, `delete[]`, rzutowanie, `sizeof()`	`++ --`
		`^ !`
		`- +`
		`& *`
		`()`

4	wybór składowej dla wskaźnika	`.* ->*`
5	mnożenie, dzielenie, modulo	`* / %`
6	dodawanie, odejmowanie	`+ -`
7	przesunięcie (w lewo, w prawo)	`<< >>`
8	relacje większości i mniejszości	`< <= > >=`
9	równe, nierówne	`== !=`
10	bitowe `AND`	`&`
11	bitowe `XOR`	`^`
12	bitowe `OR`	`|`
13	logiczne `AND`	`&&`
14	logiczne `OR`	`||`
15	operator warunkowy	`?:`

16	operatory przypisania	`= *= /= %=`
		`+= -+ <<=`
		`>>=`
		`&= |= ^=`

17	przecinek	`,`

Dodatek D

Odpowiedzi

Lekcja 1. Zaczynamy

Quiz

1. Interpreter to narzędzie interpretujące Twój kod (oraz pośredni kod bajtowy) i wykonujące pewne określone zadania. Z kolei kompilator pobiera Twój kod jako dane wejściowe i generuje plik obiektu. W języku C++ po fazach kompilacji i łączenia otrzymujesz plik wykonywalny, który może być bezpośrednio uruchamiany przez procesor bez konieczności jakiejkolwiek dalszej interpretacji.
2. Kompilator pobiera plik kodu C++ jako dane wejściowe i generuje plik obiektu w języku maszynowym. Bardzo często zdarza się, że kod zawiera zależności w postaci bibliotek i funkcji w innych plikach kodu. Utworzenie tych powiązań i wygenerowanie pliku wykonywalnego, który bezpośrednio i pośrednio integruje wszystkie zależności, jest zadaniem linkera.
3. Utworzenie kodu. Kompilacja, aby utworzyć plik obiektu. Użycie linkera do utworzenie pliku wykonywalnego. Uruchomienie programu w celu jego przetestowania. Usunięcie znalezionych błędów i ponowne wykonanie wymienionych kroków.
4. Standard C++11 obsługuje model przenośnych wątków, który pozwala programiście na utworzenie wielowątkowych aplikacji przy użyciu standardowych funkcji wątków C++11. W ten sposób wielordzeniowy procesor może być optymalnie wykorzystany, bo jednocześnie przez poszczególne rdzenie procesora wykonywane są różne wątki w aplikacji .

Ćwiczenia

1. Ten program wyświetla wynik odjęcia *y* od *x*, pomnożenia wartości wymienionych zmiennych oraz ich dodania.
2. Dane wyjściowe powinny być w postaci:

   ```
   2 48 14
   ```

3. Umieszczone w wierszu pierwszym polecenie preprocesora pozwalające na dodanie pliku nagłówkowego `iostream` powinno rozpoczynać się od znaku #.
4. Program wyświetla w konsoli komunikat `Witaj, świecie z błędami`.

Lekcja 2. Anatomia programu C++

Quiz

1. Kod w języku C++ rozróżnia wielkość liter. Dlatego też Int nie zostanie przez kompilator rozpoznane jako słowo kluczowe oznaczające liczbę całkowitą (`int`).
2. Tak.

   ```
   /* Jeżeli użyjesz składni w stylu C dla komentarza,
   wtedy komentarz może obejmować więcej niż tylko jeden wiersz. */
   ```

Ćwiczenia

1. Niepowodzenie wynika z faktu, że C++ rozróżnia wielkość liter, więc kompilator nie rozpoznaje polecenia `std::Cout` i nie wie, dlaczego znajdujący się po nim ciąg tekstowy nie został ujęty w cudzysłów. Ponadto deklaracja funkcji `main()` zawsze powinna zwracać wartość typu `int`.
2. Poniżej przedstawiono jedno z możliwych rozwiązań:

   ```
   #include <iostream>
   int main()
   {
      std::cout << "Czy tutaj jest błąd?"; // Nie, już nie.
      return 0;
   }
   ```

3. Poniżej przedstawiono jedno z możliwych rozwiązań:

```
#include <iostream>
using namespace std;

// Deklaracja funkcji.
int DemoConsoleOutput();

int main()
{
   // Wywołanie funkcji.
   DemoConsoleOutput();

   return 0;
}

// Definicja funkcji.
int DemoConsoleOutput()
{
   cout << "Operacja odejmowania 10 - 5 = " << 10 - 5 << endl;
   cout << "Operacja mnożenia 10 * 5 = " << 10 * 5 << endl;

   return 0;
}
```

Wynik ▼

```
Operacja odejmowania 10 – 5 = 5

Operacja mnożenia 10*5 = 50
```

Lekcja 3. Zmienne i stałe

Quiz

1. W liczbie całkowitej ze znakiem (typ `signed`) najbardziej znaczący bit (MSB) pełni funkcję bitu znaku i określa, czy wartość jest liczbą całkowitą dodatnią, czy ujemną. Z kolei liczba całkowita bez znaku (typ `unsigned`) jest używana do przechowywania jedynie dodatnich liczb całkowitych.
2. `#define` to dyrektywa preprocesora, która nakazuje kompilatorowi zamianę wszystkich wystąpień danego tekstu przez zdefiniowaną wartość. Takie rozwiązanie jednak nie zapewnia bezpieczeństwa typów

i stanowi prymitywny sposób definiowania stałych. Dlatego też należy unikać jego stosowania.

3. Aby zagwarantować, że będą zawierały konkretną wartość, a nie zupełnie przypadkową.
4. 2
5. Nazwa nie jest opisowa i wskazuje typ. Wprawdzie taki kod zostanie skompilowany, ale staje się trudny w odczycie i obsłudze dla innych programistów. Należy unikać tworzenia w taki sposób. Liczbę całkowitą lepiej zadeklarować, używając nazwy wskazującej jej przeznaczenie, np. tak:

```
int Age = 0;
```

Ćwiczenia

1. Istnieje kilka sposobów rozwiązania tego ćwiczenia:

```
enum KARTY {AS = 43, WALET, DAMA, KRÓL};
// AS ma wartość 43, WALET ma wartość 44, DAMA ma wartość 45, KRÓL ma wartość 46.
// Alternatywne rozwiązanie.
enum KARTY {AS, WALET, DAMA = 45, KRÓL};
// AS ma wartość 43, WALET ma wartość 44, DAMA ma wartość 45, KRÓL ma wartość 46.
```

2. Przejrzyj listing 3.4, a następnie przystosuj go (skróć), aby uzyskać rozwiązanie tego ćwiczenia.
3. Poniżej przedstawiono program, który prosi o podanie promienia okręgu, a następnie oblicza jego pole i obwód:

```
#include <iostream>
using namespace std;
int main()
{
   const double Pi = 3.1416;

   cout << "Podaj promień okręgu: ";
   double Radius = 0;
   cin >> Radius;

   cout << "Pole = " << Pi * Radius * Radius << endl;
   cout << "Obwód = " << 2 * Pi * Radius << endl;

   return 0;
}
```

Wynik ▼

```
Podaj promień okręgu: 4
Pole = 50.2656
Obwód: 25.1328
```

4. Jeżeli wynik obliczenia pola i obwodu zostanie umieszczony w zmiennej typu liczba całkowita, wtedy otrzymasz ostrzeżenie (nie błąd) w trakcie kompilacji, a dane wyjściowe będą miały następującą postać:

Wynik ▼

```
Podaj promień okręgu: 4
Pole = 50
Obwód: 25
```

5. `auto` to konstrukcja pozwalająca kompilatorowi na automatyczne określenie typu zmiennej, w zależności od wartości, z jaką jest inicjalizowana. Polecenie wymienione w ćwiczeniu nie inicjalizuje wartości, więc kompilacja zakończy się niepowodzeniem.

Lekcja 4. Tablice i ciągi tekstowe

Quiz

1. 0 i 4 to indeksy pierwszego i ostatniego elementu w tablicy składającej się z pięciu elementów.
2. Nie, ciągi tekstowe w stylu C są niebezpieczne, zwłaszcza podczas obsługi danych wejściowych pochodzących od użytkownika, ponieważ pozwalają mu na wprowadzenie ciągu tekstowego większego od pojemności tablicy.
3. Jeden znak `null`.
4. To zależy od sposobu użycia tego ciągu tekstowego. W poleceniu `cout` logika wyświetlająca komunikat będzie odczytywać kolejne znaki w poszukiwaniu znaku `null`. To może doprowadzić do wykroczenia poza tablicę, a tym samym prawdopodobnie do awarii aplikacji.
5. To jest proste i polega na zastąpieniu `int` w deklaracji wektora typem `char`.

```
vector<char> DynArrChars (3);
```

Ćwiczenia

1. Poniżej przedstawiono jedno z możliwych rozwiązań. Wprawdzie poniższy program przeprowadza inicjalizację jedynie wież, ale pokazuje ogólną koncepcję.

```
int main()
{
   enum SQUARE
   {
      NIC = 0,
      PIONEK,
      WIEŻA,
      SKOCZEK,
      GONIEC,
      KRÓL,
      KRÓLOWA
   };

   SQUARE ChessBoard[8][8];

   // Inicjalizacja planszy wraz z wieżami.
   ChessBoard[0][0] = ChessBoard[0][7] = WIEŻA;
   ChessBoard[7][0] = ChessBoard[7][7] = WIEŻA;

   return 0;
}
```

2. Aby ustawić piąty element tablicy, musisz uzyskać dostęp do elementu `MyNumbers[4]`, ponieważ indeks rozpoczyna się od zera.
3. Dostęp do czwartego elementu tablicy następuje bez jego wcześniejszej inicjalizacji lub przypisania mu wartości. Wynik tego rodzaju operacji jest trudny do przewidzenia. Zawsze należy inicjalizować zmienne i tablice, ponieważ w przeciwnym razie zawierają ostatnie wartości znajdujące się w danym adresie pamięci w chwili tworzenia zmiennej lub tablicy.

Lekcja 5. Wyrażenia, instrukcje i operatory

Quiz

1. Typ w postaci liczby całkowitej nie może zawierać wartości dziesiętnych, które prawdopodobnie są istotne dla użytkownika

przeprowadzającego operację dzielenia dwóch liczb. Dlatego też powinieneś użyć typu `float`.

2. Ponieważ kompilator interpretuje liczby jako całkowite, wynikiem jest 4.
3. Ponieważ dzielna wynosi 32.0, a nie 32, kompilator uznaje operację za przeprowadzaną na liczbach zmiennoprzecinkowych i wynik o wartości około 4.571 jest typu `float`.
4. Nie, `sizeof` jest operatorem i nie może być przeciążony.
5. Polecenie nie działa zgodnie z oczekiwaniami, ponieważ operator dodawania ma pierwszeństwo przed operatorem przesunięcia, więc nastąpi przesunięcie 6 (1+5) bitów zamiast tylko jednego.
6. Wynikiem operacji XOR jest fałsz (`false`), zgodnie z tabelą 5.5.

Ćwiczenia

1. Poniżej przedstawiono jedno z możliwych rozwiązań:
   ```
   int Result = ((number << 1) + 5) << 1; // Niejasne nawet dla człowieka.
   ```
2. Wynik zawiera wartość `number` przesuniętą o siedem bitów w lewo, ponieważ operator dodawania (+) ma pierwszeństwo przed operatorem przesunięcia (<<).
3. Poniżej przedstawiono program przechowujący dwie wartości boolowskie podane przez użytkownika. W programie pokazano wynik przeprowadzenia operacji bitowej na podanych wartościach.

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Podaj pierwszą boolowską wartość true(1) lub false(0): ";
   bool Value1 = false;
   cin >> Value1;

   cout << "Podaj drugą boolowską wartość true(1) lub false(0): ";
   bool Value2 = false;
   cin >> Value2;

   cout << "Wynik przeprowadzenia operacji bitowych na podanych
operandach: " << endl;
   cout << "Bitowe AND: " << (Value1 & Value2) << endl;
   cout << "Bitowe OR: " << (Value1 | Value2) << endl;
```

```
    cout << "Bitowe XOR: " << (Value1 ^ Value2) << endl;

    return 0;
}
```

Wynik ▼

```
Podaj pierwszą boolowską wartość true(1) lub false(0): 1
Podaj drugą boolowską wartość true(1) lub false(0): 0
Wynik przeprowadzenia operacji bitowych na podanych operandach:
Bitowe AND: 0
Bitowe OR: 1
Bitowe XOR: 1
```

Lekcja 6. Sterowanie przebiegiem działania programu

Quiz

1. Wcięcia stosujesz nie ze względu na kompilator, ale dla siebie i innych osób, które być może będą czytać dany kod lub próbować go zrozumieć.
2. Powinieneś unikać stosowania `goto` w kodzie, ponieważ to polecenie powoduje powstanie nieintuicyjnego i trudnego w obsłudze kodu.
3. Spójrz na kod w odpowiedzi do ćwiczenia 1. Kod ten wykorzystuje operator dekrementacji.
4. Ponieważ warunek polecenia `for` nie jest spełniony, pętla nie zostanie wykonana ani razu, a tym samym znajdujące się w bloku pętli polecenie `cout` nigdy nie będzie wykonane.

Ćwiczenia

1. Pamiętaj, że indeks ma wartości liczone od zera, natomiast ostatni element ma indeks o wartości (`liczba_elementów – 1`):

```
#include <iostream>
using namespace std;

int main()
{
    const int ARRAY_LEN = 5;
    int MyNumbers[ARRAY_LEN]= {-55, 45, 9889, 0, 45};
```

```
    for (int nIndex = ARRAY_LEN - 1; nIndex >= 0; —nIndex)
       cout<<"MyNumbers[" << nIndex << "] = "<<MyNumbers[nIndex]<<endl;

    return 0;
}
```

Wynik ▼

```
MyNumbers[4] = 45
MyNumbers[3] = 0
MyNumbers[2] = 9889
MyNumbers[1] = 45
MyNumbers[0] = -55
```

2. Poniżej przedstawiono jedno z możliwych rozwiązań — odpowiednik pętli zagnieżdżonej z listingu 6.13, która w odwrotnej kolejności dodaje elementy z dwóch tablic:

```
#include <iostream>
using namespace std;

int main()
{
   const int ARRAY1_LEN = 3;
   const int ARRAY2_LEN = 2;

   int MyInts1[ARRAY1_LEN] = {35, -3, 0};
   int MyInts2[ARRAY2_LEN] = {20, -1};

   cout << "Dodanie każdej liczby typu int w MyInts1 do każdej
   ↪w MyInts2:" << endl;

   for(int Array1Index=ARRAY1_LEN-1;Array1Index>=0;--Array1Index)
      for(int Array2Index=ARRAY2_LEN-1;Array2Index>=0;--Array2Index)
         cout<<MyInts1[Array1Index]<<" + "<<MyInts2[Array2Index] \
         << " = " << MyInts1[Array1Index] + MyInts2[Array2Index] << endl;

   return 0;
}
```

Wynik ▼

```
Dodanie każdej liczby typu int w MyInts1 do każdej w MyInts2:
0 + -1 = -1
0 + 20 = 20
-3 + -1 = -4
-3 + 20 = 17
35 + -1 = 34
35 + 20 = 55
```

3. Konieczne jest zastąpienie stałej w postaci liczby całkowitej z określoną wartością kodem, który zadaje użytkownikowi pytanie o ilość liczb ciągu Fibonacciego do obliczenia:

```
cout << "Ile liczb ciągu Fibonacciego chcesz obliczyć: ";
int NumsToCal = 0;
cin >> NumsToCal;
```

Konstrukcja `switch-case`, w której użyto stałych typu wyliczeniowego pozwala na określenie, czy kolor zalicza się do kolorów tęczy:

```
#include <iostream>
using namespace std;

int main()
{
   enum COLORS
   {
      VIOLET = 0,
      INDIGO,
      BLUE,
      GREEN,
      YELLOW,
      ORANGE,
      RED,
      CRIMSON,
      BEIGE,
      BROWN,
      PEACH,
      PINK,
      WHITE,
   };

   cout << "Poniżej wymieniono dostępne kolory: " << endl;
   cout << "Fioletowy: " << VIOLET << endl;
   cout << "Indygo: " << INDIGO << endl;
   cout << "Niebieski: " << BLUE << endl;
   cout << "Zielony: " << GREEN << endl;
   cout << "Żółty: " << YELLOW << endl;
   cout << "Pomarańczowy: " << ORANGE << endl;
   cout << "Czerwony: " << RED << endl;
   cout << "Purpurowy: " << CRIMSON << endl;
   cout << "Szary: " << BEIGE << endl;
   cout << "Brązowy: " << BROWN << endl;
   cout << "Brzoskwiniowy: " << PEACH << endl;
   cout << "Różowy: " << PINK << endl;
   cout << "Biały: " << WHITE << endl;

   cout << "Wybierz jeden kolor i podaj jego kod: ";
   int YourChoice = BLUE; // Wartość początkowa.
```

```
    cin >> YourChoice;

    switch (YourChoice)
    {
       case VIOLET:
       case INDIGO:
       case BLUE:
       case GREEN:
       case YELLOW:
       case ORANGE:
       case RED:
          cout << "Bingo, wybrany kolor zalicza się do kolorów tęczy!" <<
          ↪endl;
          break;
       default:
          cout << "Wybrany kolor nie zalicza się do kolorów tęczy" <<
          ↪endl;
          break;
    }

    return 0;
}
```

Wynik ▼

```
Poniżej wymieniono dostępne kolory:
Fioletowy: 0
Indygo: 1
Niebieski: 2
Zielony: 3
Żółty: 4
Pomarańczowy: 5
Czerwony: 6
Purpurowy: 7
Szary: 8
Brązowy: 9
Brzoskwiniowy: 10
Różowy: 11
Biały: 12
Wybierz jeden kolor i podaj jego kod: 4
Bingo, wybrany kolor zalicza się do kolorów tęczy!
```

4. Programista przypadkowo przypisał wartość 10 w warunku pętli `for`.
5. Po instrukcji `while` znajduje się średnik kończący polecenie. Dlatego też kod pętli `while` nigdy nie zostanie wykonany. Skoro zmienna `LoopCounter` nie będzie inkrementowana, pętla `while` nie ma końca i jej polecenia nigdy nie będą wykonane.
6. Brakuje polecenia `break` (zawsze zostanie wykonany blok `default`).

Lekcja 7. Funkcje

Quiz

1. Zasięg zmiennych zadeklarowanych w funkcji obejmuje daną funkcję w całym okresie jej istnienia.
2. `SomeNumber` to odniesienie do zmiennej w funkcji wywołującej. To odniesienie nie jest kopią zmiennej.
3. Funkcja rekurencyjna.
4. Funkcje przeciążone.
5. Na górze. Wyobraź sobie stos talerzy: znajdujący się na górze możesz bez problemów wziąć, do tego odnosi się wskaźnik stosu.

Ćwiczenia

1. Prototypy funkcji mogą przedstawiać się następująco:

   ```
   double Area (double Radius); // Okrąg.
   double Area (double Radius, double Height); // Walec.
   ```

   Implementacje (definicje) funkcji korzystają z odpowiednich wzorów podanych w pytaniu, a obliczone pole jest zwracane wywołującemu jako wartość zwrotna.

2. Niech listing 7.8 będzie dla Ciebie inspiracją. Prototyp funkcji może być następujący:

   ```
   void ProcessArray(double Numbers[], int Length);
   ```

3. Aby funkcja `Area()` działała efektywnie, parametr `Result` powinien być referencją:

   ```
   void Area(double Radius, double &Result)
   ```

4. Parametr domyślny powinien być wymieniony na końcu lub wszystkie parametry powinny mieć zdefiniowane wartości domyślne.
5. Wygenerowane dane wyjściowe funkcja powinna przekazywać wywołującemu przy użyciu referencji:

   ```
   void Area (double Radius, double &Area, double &Circumference)
   {
      Area = 3.14 * Radius * Radius;
      Circumference = 2 * 3.14 * Radius;
   }
   ```

Lekcja 8. Wskaźniki i referencje

Quiz

1. Gdyby kompilator pozwalał na takie zachowanie, byłoby to całkowicie sprzeczne z przeznaczeniem referencji typu `const`: tych danych nie wolno modyfikować.
2. Nie, to są operatory.
3. To adres w pamięci.
4. Operator dereferencji (*).

Ćwiczenia

1. 40.
2. W pierwszej przeciążonej wersji argumenty są kopiowane do wywoływanej funkcji. Natomiast w drugiej wersji nie są kopiowane, ponieważ to referencje do zmiennych w funkcji wywołującej i nie można ich zmieniać. W trzeciej wersji użyto wskaźników, które — w przeciwieństwie do referencji — mogą mieć wartość `null` lub być nieprawidłowe, ich poprawność trzeba zagwarantować w tego rodzaju systemie.
3. Użyj słowa kluczowego `const`:

   ```
   1: const int* pNum1 = &Number;
   ```

4. Bezpośrednie przypisanie liczby całkowitej wskaźnikowi (tzn. nadpisanie liczbą całkowitą przechowywanego we wskaźniku adresu w pamięci):

   ```
   *pNumber = 9; // Poprzednio: pNumber = 9;
   ```

5. Dwukrotne wywołanie operatora `delete` względem tego samego adresu w pamięci zwróconego przez `new` dla `pNumber` i powielonego dla `pNumberCopy`. Usuń jeden z nich.
6. 30.

Lekcja 9. Klasy i obiekty

Quiz

1. W puli wolnej pamięci. To tak samo jakbyś alokował pamięć dla zmiennej typu `int` podczas użycia operatora `new`.
2. Operator `sizeof()` oblicza wielkość klasy na podstawie zadeklarowanych elementów składowych. Ponieważ `sizeof(wskaźnik)` to stała, niezależna od wielkości wskazywanych danych, a `sizeof(klasa)` zawiera tego rodzaju wskaźniki, zatem również pozostaje stałą.
3. Nikt poza metodami składowymi tej samej klasy.
4. Tak, możesz.
5. Konstruktor jest zwykle używany do inicjalizacji danych składowych i zasobów.
6. Destruktor jest najczęściej stosowany do zwolnienia zasobów i zaalokowanej wcześniej pamięci.

Ćwiczenia

1. Język C++ rozróżnia wielkość liter. Deklaracja klasy powinna zaczynać się od słowa kluczowego `class`, a nie `Class`, i kończyć średnikiem, tak jak przedstawiono w poniższym fragmencie kodu:

```
class Human
{
   int Age;
   string Name;

public:
   Human() {}
};
```

2. Ponieważ `Human::Age` to prywatny element składowy (pamiętaj, że w przeciwieństwie do struktury (`struct`) elementy klasy (`class`) są domyślnie prywatne). Nie istnieje więc publiczna funkcja akcesora, a tym samym użytkownik klasy nie może uzyskać dostępu do elementu `Age`.
3. Poniżej przedstawiono wersję klasy `Human` wraz z listą inicjalizacyjną w konstruktorze:

```
class Human
{
   int Age;
   string Name;

public:
   Human(string InputName, int InputAge)
      : Name(InputName), Age(InputAge) {}
};
```

4. Zwróć uwagę, jak Pi nie jest udostępniane na zewnątrz klasy:

```
#include <iostream>
using namespace std;

class Circle
{
   const double Pi;
   double Radius;

public:
   Circle(double InputRadius) : Radius(InputRadius), Pi(3.1416) {}

   double GetCircumference()
   {
      return 2*Pi*Radius;
   }

   double GetArea()
   {
      return Pi*Radius*Radius;
   }
};

int main()
{
   cout << "Podaj promień: ";
   double Radius = 0;
   cin >> Radius;

   Circle MyCircle(Radius);

   cout << "Obwód = " << MyCircle.GetCircumference() << endl;
   cout << "Pole = " << MyCircle.GetArea() << endl;

   return 0;
}
```

Lekcja 10. Dziedziczenie

Quiz

1. Użyj specyfikatora dostępu `protected`, aby zagwarantować, że element składowy klasy bazowej będzie dostępny dla klasy potomnej, ale nie dla egzemplarza tej samej klasy.
2. Obiekt klasy potomnej ulegnie segmentowaniu i przez wartość zostanie przekazana tylko część odpowiadająca klasie bazowej. Zachowanie takiego obiektu jest nieprzewidywalne.
3. W takim przypadku największą elastyczność zapewnia kompozycja.
4. Słowo kluczowe `using` pozwala na udostępnienie metod klasy bazowej.
5. Nie, ponieważ pierwsza klasa specjalizująca `Base` — tzn. klasa `Derived` — stosuje dziedziczenie prywatne po klasie `Base`. Tym samym publiczne elementy składowe klasy `Base` są prywatne dla klasy `SubDerived`, czyli pozostają niedostępne.

Ćwiczenia

1. Konstrukcja w kolejności wymienionej w deklaracji klasy (`Mammal`, `Bird`, `Reptile`, `Platypus`), natomiast destrukcja w odwrotnej kolejności. Poniżej przedstawiono kod programu:

```
#include <iostream>
using namespace std;

class Mammal
{
public:
   void FeedBabyMilk()
   {
      cout << "Ssak: młode musi dostać mleko!" << endl;
   }

   Mammal()
   {
      cout << "Mammal: konstruktor" << endl;
   }
   ~Mammal()
   {
      cout << "Mammal: destruktor" << endl;
   }
```

```
};

class Reptile
{
public:
   void SpitVenom()
   {
      cout << "Gad: przegonić wroga, pluć jadem!"  << endl;
   }

   Reptile()
   {
      cout << "Reptile: konstruktor" << endl;
   }
   ~Reptile()
   {
      cout << "Reptile: destruktor" << endl;
   }
};

class Bird
{
public:
   void LayEggs()
   {
      cout << "Ptak: muszę złożyć jaja!" << endl;
   }

   Bird()
   {
      cout << "Bird: konstruktor" << endl;
   }
   ~Bird()
   {
      cout << "Bird: destruktor" << endl;
   }
};

class Platypus: public Mammal, public Bird, public Reptile
{
public:
   Platypus()
   {
      cout << "Platypus: konstruktor" << endl;
   }
   ~Platypus()
   {
   cout << "Platypus: destruktor" << endl;
   }
```

```
};

int main()
{
   Platypus realFreak;
   //realFreak.LayEggs();
   //realFreak.FeedBabyMilk();
   //realFreak.SpitVenom();

   return 0;
}
```

2. Poniżej przedstawiono kod:

```
class Shape
{
   // Elementy składowe klasy Shape.
};

class Polygon: public Shape
{
   // Elementy składowe klasy Polygon.
}

class Triangle: public Polygon
{
   // Elementy składowe klasy Triangle.
}
```

3. Dziedziczenie pomiędzy klasą `D1` i `Base` powinno być prywatne, aby uniemożliwić klasie `D2` uzyskanie dostępu do publicznych metod klasy `Base`.
4. Klasy domyślnie stosują dziedziczenie prywatne. Jeżeli `Derived` to byłaby struktura, wtedy dziedziczenie byłoby publiczne.
5. `SomeFunc` przez wartość pobiera parametr typu `Base`. Oznacza to, że wywołanie wymienionego typu spowoduje segmentowanie, co z kolei doprowadzi do niestabilności i trudnych do przewidzenia danych wyjściowych:

```
Derived objectDerived;
SomeFunc(objectDerived); // Ulegnie segmentowaniu.
```

Lekcja 11. Polimorfizm

Quiz

1. Zadeklaruj abstrakcyjną klasę bazową Shape wraz z funkcjami czysto wirtualnymi `Area()` i `Print()`, co wymusi na klasach `Circle` i `Triangle` również ich implementację. Wymienione klasy będą zmuszone do zachowania zgodności z wymaganymi kryteriami, aby zapewnić obsługę funkcji `Area()` i `Print()`.
2. Nie. Tabela funkcji wirtualnych jest tworzona jedynie dla klas zawierających funkcje wirtualne, dotyczy to również klas potomnych.
3. Tak, ale nie można utworzyć jej egzemplarza. Dopóki klasa będzie zawierała przynajmniej jedną funkcję czysto wirtualną, dopóty pozostanie abstrakcyjną klasą bazową, niezależnie od obecności lub nieobecności innych w pełni zdefiniowanych funkcji bądź parametrów.

Ćwiczenia

1. Poniżej przedstawiono jedno z możliwych rozwiązań w zakresie hierarchii dziedziczenia abstrakcyjnej klasy bazowej Shape dla klas `Circle` i `Triangle`:

```
#include<iostream>
using namespace std;

class Shape
{
public:
   virtual double Area() = 0;
   virtual void Print() = 0;
};

class Circle
{
   double Radius;
public:
   Circle(double inputRadius) : Radius(inputRadius) {}

   double Area()
   {
      return 3.1415 * Radius * Radius;
   }
```

```
   void Print()
   {
      cout << "Witaj w okręgu!" << endl;
   }
};

class Triangle
{
   double Base, Height;
public:
   Triangle(double inputBase, double inputHeight) : Base(inputBase),
      Height(inputHeight) {}

   double Area()
   {
      return 0.5 * Base * Height;
   }

   void Print()
   {
      cout << "Witaj w trójkącie!" << endl;
   }
};

int main()
{
   Circle myRing(5);
   Triangle myWarningTriangle(6.6, 2);

   cout << "Pole okręgu: " << myRing.Area() << endl;
   cout << "Pole trójkąta: " << myWarningTriangle.Area() << endl;

   myRing.Print();
   myWarningTriangle.Print();

   return 0;
}
```

2. Brakuje wirtualnego destruktora!
3. Bez wirtualnego destruktora sekwencja konstruktora będzie następująca: `Vehicle()` i później `Car()`, natomiast niewirtualny destruktor powoduje wywołanie jedynie `~Car()`.

Lekcja 12. Typy operatorów i ich przeciążanie

Quiz

1. Nie, język C++ nie pozwala, aby dwie funkcje o takiej samej nazwie miały różne wartości zwrotne. Możesz przygotować dwie implementacje operatora [] wraz z identycznymi typami zwrotnymi, jeden typ zdefiniowany jako funkcja `const`, natomiast drugi nie. W takim przypadku kompilator C++ wybierze wersję inną niż `const` podczas operacji przypisania i wersję `const` w pozostałych operacjach:

   ```
   Type& operator[](int Index) const;
   Type& operator[](int Index);
   ```

2. Tak, ale tylko wtedy, gdy nie chcesz zezwolić klasie na kopiowanie lub przypisywanie.

3. Ponieważ w klasie `Date` nie ma dynamicznie przypisywanych zasobów, które powodowałyby niepotrzebne cykle alokacji i zwalniania pamięci w konstruktorze kopiującym lub kopiującym operatorze przypisania, wymieniona klasa nie jest dobrym kandydatem dla konstruktora przenoszącego lub przenoszącego operatora przypisania.

Ćwiczenia

1. Operator konwersji `int()` został przedstawiony poniżej:

   ```
   class Date
   {
      int Day, Month, Year;
   public:
      operator int()
      {
         return ((Year * 10000) + (Month * 100) + Day);
      }

      // Konstruktor itd.
   };
   ```

2. Konstruktor przenoszący i przenoszący operator przypisania zostały przedstawione w poniższym fragmencie kodu:

   ```
   class DynIntegers
   {
   ```

```
private:
   int* pIntegers;

public:
   // Konstruktor przenoszący.
   DynIntegers(DynIntegers&& MoveSource)
   {
      pIntegers = MoveSource.pIntegers;  // Przejęcie własności.
      MoveSource.pIntegers = NULL;  // Zwolnienie własności w źródle.
   }

   // Przenoszący operator przypisania.
   DynIntegers& operator= (DynIntegers&& MoveSource)
   {
      if(this != &MoveSource)
      {
         delete [] pIntegers;  // Zwolnienie zasobów.
         pIntegers = MoveSource.pIntegers;
         MoveSource.pIntegers = NULL;
      }
      return *this;
   }

   ~DynIntegers() {delete[] pIntegers;}  // Destruktor.

   // Implementacja konstruktora domyślnego, konstruktora kopiującego,
   // operatora przypisania.
};
```

Lekcja 13. Operatory rzutowania

Quiz

1. `dynamic_cast`
2. Oczywiście, trzeba poprawić funkcję. Ogólnie rzecz biorąc, `const_cast` oraz operatory rzutowania powinny być stosowane w ostateczności.
3. Prawda.
4. Tak, to prawda.

Ćwiczenia

1. Wynik operacji rzutowania dynamicznego zawsze powinien być sprawdzony, czy jest poprawny:

```
void DoSomething(Base* pBase)
{
   Derived* pDerived = dynamic_cast <Derived*>(pBase);

   if(pDerived) // Sprawdzenie pod kątem poprawności.
      pDerived->DerivedClassMethod();
}
Użyj static_cast, ponieważ wiesz, że wskazywany obiekt jest typu
➥Tuna. Jako punkt wyjścia wykorzystaj listing 13.1, poniżej
➥przedstawiono funkcję main():
int main()
{
   Fish* pFish = new Tuna;
   Tuna* pTuna = static_cast<Tuna*>(pFish);

   // Tuna::BecomeDinner będzie działało jedynie z poprawnym wskaźnikiem Tuna*.
   pTuna->BecomeDinner();

   // Wirtualny destruktor w klasie Fish gwarantuje wywołanie ~Tuna().
   delete pFish;

   return 0;
}
```

Lekcja 14. Wprowadzenie do makr i wzorców

Quiz

1. Wartownicy dołączania są używani w celu ochrony plików nagłówkowych przed dołączeniem do programu więcej niż tylko jeden raz.
2. 4.
3. Wynik to `10 + 10 / 5`, czyli `10 + 2`, czyli 12.
4. Należy użyć nawiasów:

   ```
   #define SPLIT(x) ((x) / 5)
   ```

Ćwiczenia

1. Makro `MULTIPLY` może przedstawiać się następująco:

   ```
   #define MULTIPLY(a,b) ((a)*(b))
   ```

2. Wersja makra MULTIPLY w postaci wzorca może przedstawiać się następująco:

```
template <typename T>
T Split (const T& input)
{
   return (input / 5);
}
```

3. Wersja funkcji swap() w postaci wzorca może przedstawiać się następująco:

```
template <typename T>
void Swap (T& x, T& y)
{
   T temp = x;
   x = y;
   y = temp;
}
```

4. `#define QUARTER(x) ((x)/ 4)`

5. Definicja klasy wzorca może przedstawiać się następująco:

```
template <typename Array1Type, typename Array2Type>
class TwoArrays
{
   private:
      Array1Type Array1 [10];
      Array2Type Array2 [10];
   public:
      Array1Type& GetArray1Element(int Index){return Array1[Index];}
      Array2Type& GetArray2Element(int Index){return Array2[Index];}
};
```

Lekcja 15. Wprowadzenie do standardowej biblioteki wzorców

Quiz

1. Kontener deque. Tylko ten kontener zapewnia stały czas wstawiania elementów na początku oraz na końcu kontenera.
2. Kontener std::set lub std::map w przypadku par klucz-wartość. Jeżeli przechowywane elementy mają się powtarzać, należy wybrać kontener std::multiset bądź std::multimap.

3. Tak. Podczas ustanawiania wzorca `std::set` masz możliwość dostarczenia opcjonalnego drugiego parametru. Parametr ten jest predykatem dwuargumentowym, którego klasa `set` będzie używała jako kryterium sortowania. Zaprojektuj wspomniany predykat dwuargumentowy w taki sposób, aby zawierał kryteria spełniające Twoje wymagania. Musi być zgodny ze ścisłym uporządkowaniem słabym.
4. Iteratory budują most między algorytmami i kontenerami, dzięki któremu iteratory (które są ogólne) mogą współdziałać z kontenerami bez konieczności znajomości (lub dostosowywania do) każdego możliwego typu kontenera.
5. `hash_set` to nie jest kontener zgodny ze standardem C++. Dlatego też nie powinieneś go stosować w żadnej aplikacji, która wymaga przeniesienia na inne platformy. Zamiast tego kontenera należy użyć `std::map`.

Lekcja 16. Klasa string w STL

Quiz

1. `std::basic_string <T>`
2. Skopiuj dwa ciągi tekstowe do dwóch obiektów. Następnie skonwertuj skopiowane ciągi tekstowe albo na małe, albo na duże znaki. Wartością zwracaną powinien być wynik porównania skonwertowanych ciągów tekstowych.
3. Nie, nie są. Ciągi tekstowe w stylu języka C to zwykłe wskaźniki podobne do tablicy znaków. Natomiast obiekt `string` biblioteki STL to klasa implementująca różne operatory i funkcje składowe w maksymalnym stopniu ułatwiające obsługę oraz przeprowadzanie operacji na ciągach tekstowych.

Ćwiczenia

1. W programie należy użyć funkcji `std::reverse()`:

```
#include <string>
#include <iostream>
```

```
#include <algorithm>

int main ()
{
   using namespace std;

   cout << "Proszę podać słowo, które będzie sprawdzone pod kątem
   ↪tego, czy jest palindromem:" << endl;
   string strInput;
   cin >> strInput;

   string strCopy (strInput);
   reverse (strCopy.begin (), strCopy.end ());

   if (strCopy == strInput)
      cout << strInput << " to jest palindrom!" << endl;
   else
      cout << strInput << " to nie jest palindrom." << endl;
   return 0;
}
```

2. W programie powinno się użyć funkcji `std::find()`:

```
#include <string>
#include <iostream>

using namespace std;

// Znajdź liczbę znaków 'chToFind' w ciągu tekstowym "strInput".
int GetNumCharacters (string& strInput, char chToFind)
{
   int nNumCharactersFound = 0;

   size_t nCharOffset = strInput.find (chToFind);
   while (nCharOffset != string::npos)
   {
      ++ nNumCharactersFound;
      nCharOffset = strInput.find (chToFind, nCharOffset + 1);
   }
   return nNumCharactersFound;
}

int main ()
{
   cout << "Proszę podać ciąg tekstowy:" << endl << "> ";
   string strInput;
   getline (cin, strInput);

   int nNumVowels = GetNumCharacters (strInput, 'a');
   nNumVowels += GetNumCharacters (strInput, 'e');
```

```
   nNumVowels += GetNumCharacters (strInput, 'i');
   nNumVowels += GetNumCharacters (strInput, 'o');
   nNumVowels += GetNumCharacters (strInput, 'u');

   // Zrób to sam: zajmij się obsługą również wielkich liter.

   cout << "Liczba samogłosek w podanym zdaniu wynosi: " << nNumVowels;
   return 0;
}
```

3. W programie trzeba użyć funkcji `toupper()`:

```
#include <string>
#include <iostream>
#include <algorithm>

int main ()
{
   using namespace std;

   cout << "Proszę podać ciąg tekstowy, w którym ma być przeprowadzona
   ↪konwersja wielkości znaków:" << endl;
   cout << "> ";

   string strInput;
   getline (cin, strInput);
   cout << endl;

   for ( size_t nCharIndex = 0
      ; nCharIndex < strInput.length ()
      ; nCharIndex += 2)
      strInput [nCharIndex] = toupper (strInput [nCharIndex]);

   cout << "Ciąg tekstowy, w którym znaki skonwertowano na wielkie: " <<
   ↪endl;
   cout << strInput << endl << endl;

   return 0;
}
```

4. To można zaprogramować w bardzo prosty sposób:

```
#include <string>
#include <iostream>

int main ()
{
   using namespace std;

   const string str1 = "Uwielbiam";
   const string str2 = "ciągi";
```

```
    const string str3 = "tekstowe";
    const string str4 = "STL.";

    string strResult = str1 + " " + str2 + " " + str3 + " " + str4;

    cout << "Całe zdanie brzmi:" << endl;
    cout << strResult;

    return 0;
}
```

Lekcja 17. Dynamiczne klasy tablic w STL

Quiz

1. Nie, nie można. W stałej ilości czasu elementy mogą być dodawane jedynie na końcu sekwencji obiektu `vector`.
2. Można wstawić dodatkowe dziesięć. Po wstawieniu jedenastego nastąpi wymuszenie przeprowadzenia ponownej alokacji bufora.
3. Do usunięcia ostatniego elementu, tzn. usunięcia elementu znajdującego się na końcu.
4. Będzie typu `CMammal`.
5. Za pomocą: a) operatora indeksowania `[]` b) funkcji `at()`.
6. Iterator swobodnego dostępu.

Ćwiczenia

1. Poniżej przedstawiono jedno z możliwych rozwiązań:

```
#include <vector>
#include <iostream>

using namespace std;

char DisplayOptions ()
{
    cout << "Co chcesz teraz zrobić?" << endl;
    cout << "Opcja 1: Podanie liczby całkowitej" << endl;
    cout << "Opcja 2: Pobranie wartości z podanego indeksu" << endl;
    cout << "Opcja 3: Wyświetlenie obiektu vector" << endl << "> ";
    cout << "Opcja 4: Zakończenie pracy!" << endl << "> ";
```

```
   char ch;
   cin >> ch;

   return ch;
}

int main ()
{
   vector <int> vecData;

   char chUserChoice = '\0';
   while ((chUserChoice = DisplayOptions ()) != '4')
   {
      if (chUserChoice == '1')
      {
        cout << "Proszę podać liczbę całkowitą, która ma być wstawiona: ";
         int nDataInput = 0;
         cin >> nDataInput;
         vecData.push_back (nDataInput);
      }
      else if (chUserChoice == '2')
      {
         cout << "Proszę wskazać indeks z zakresu od 0 do ";
         cout << (vecData.size () - 1) << ": ";
         int nIndex = 0;
         cin >> nIndex;

         if (nIndex < (vecData.size ()))
         {
            cout<<"Element ["<<nIndex<<"] = "<<vecData [nIndex];
            cout << endl;
         }
      }
      else if (chUserChoice == '3')
      {
         cout << "Zawartość obiektu vector jest następująca: ";
         for (size_t nIndex = 0; nIndex < vecData.size (); ++ nIndex)
            cout << vecData [nIndex] << ' ';
         cout << endl;
      }
   }
   return 0;
}
```

2. Użyj algorytmu `std::find`:

```
vector <int>::iterator iElementFound = std::find (vecData.begin (),
   vecData.end (), nDataInput);
```

3. Wykorzystaj kod źródłowy przedstawiony w rozwiązaniu do ćwiczenia 1., akceptujący dane wejściowe użytkownika, a następnie wyświetl zawartość obiektu `vector`.

Lekcja 18. Klasy STL list i forward_list

Quiz

1. Elementy mogą być wstawiane zarówno w środku listy, jak również na początku i na końcu. Pozycja wstawianego elementu nie ma żadnego wpływu na wydajność działania programu.
2. Cechą listy jest to, że operacje, takie jak wymienione, nie powodują unieważnienia istniejących iteratorów.
3. `mList.clear ();`

   lub

   `mList.erase (mList.begin(), mList.end());`
4. Tak, przeciążona funkcja `insert()` pozwala na wstawianie wielu obiektów do obiektu `list`.

Ćwiczenia

1. Rozwiązanie tego ćwiczenia jest podobne do rozwiązania ćwiczenia 1. dotyczącego obiektu `vector`. Jedyną zmianą jest użycie funkcji wstawiania obiektu `list`:

   ```
   mList.insert (mList.begin(),nDataInput);
   ```
2. Trzeba przechowywać iteratory do dwóch elementów obiektu `list`. Wstaw element w środku listy, używając funkcji wstawiania obiektu `list`. Wykorzystaj iteratory w celu zademonstrowania, że nadal mogą pobierać wartości, do których prowadziły jeszcze przed operacją wstawiania.
3. Poniżej przedstawiono jedno z możliwych rozwiązań:

   ```
   #include <vector>
   #include <list>
   #include <iostream>

   using namespace std;
   ```

```
int main ()
{
   vector <int> vecData (4);
   vecData [0] = 0;
   vecData [1] = 10;
   vecData [2] = 20;
   vecData [3] = 30;

   list <int> listIntegers;

   // Wstawienie zawartości obiektu vector na początku obiektu list.
   listIntegers.insert (listIntegers.begin (),
      vecData.begin (), vecData.end());

   cout << "Zawartość obiektu list jest następująca: ";

   list <int>::const_iterator iElement;
   for ( iElement = listIntegers.begin ()
      ; iElement != listIntegers.end ()
      ; ++ iElement)
      cout << *iElement << " ";

   return 0;
};
```

4. Poniżej przedstawiono jedno z możliwych rozwiązań:

```
#include <list>
#include <string>
#include <iostream>

using namespace std;

int main ()
{
   list <string> listNames;
   listNames.push_back ("Jacek");
   listNames.push_back ("Jan");
   listNames.push_back ("Anna");
   listNames.push_back ("Sławek");

   cout << "Zawartość obiektu list jest następująca: ";

   list <string>::const_iterator iElement;
   for (iElement = listNames.begin(); iElement!=listNames.end();
   ↪++iElement)
      cout << *iElement << " ";
   cout << endl;

   cout << "Zawartość obiektu list po sortowaniu jest następująca : ";
   listNames.sort ();
```

```
   for (iElement = listNames.begin(); iElement!=listNames.end();
   ↪++iElement)
      cout << *iElement << " ";
   cout << endl;

   return 0;
};
```

Lekcja 19. Klasy STL set

Quiz

1. Domyślne kryterium sortowania jest określane przez predykat sortowania `std::less<>`, w którym w celu porównania dwóch liczb całkowitych używa się operatora <. Jeżeli pierwsza liczba jest mniejsza od drugiej, zwracaną wartością będzie `true`.
2. Tak, umieszczone razem, jeden za drugim.
3. To funkcja `size()`, podobnie jak w przypadku wszystkich kontenerów STL.

Ćwiczenia

1. Predykat dwuargumentowy może mieć następującą postać:

```
struct FindContactGivenNumber
{
   bool operator()(const CContactItem& lsh,const CContactItem& rsh)
   ↪const
   {
      return (lsh.strPhoneNumber < rsh.strPhoneNumber);
   }
};
```

2. Strukturę oraz definicję obiektu `multiset` można przedstawić następująco:

```
#include <set>
#include <iostream>
#include <string>

using namespace std;

struct PAIR_WORD_MEANING
{
   string strWord;
   string strMeaning;
```

```
   PAIR_WORD_MEANING (const string& sWord, const string& sMeaning)
         : strWord (sWord), strMeaning (sMeaning) {}

   bool operator< (const PAIR_WORD_MEANING& pairAnotherWord) const
   {
      return (strWord < pairAnotherWord.strWord);
   }
};

int main ()
{
   multiset <PAIR_WORD_MEANING> msetDictionary;
   PAIR_WORD_MEANING word1 ("C++", "Język programowania");
   PAIR_WORD_MEANING word2 ("Programista", "Maniak komputerowy!");

   msetDictionary.insert (word1);
   msetDictionary.insert (word2);

   return 0;
}
```

3. Poniżej przedstawiono jedno z możliwych rozwiązań:

```
#include <set>
#include <iostream>

using namespace std;

template <typename T>
void DisplayContent (const T& sequence)
{
   T::const_iterator iElement;

   for (iElement = sequence.begin(); iElement!=sequence.end();
   ↪++iElement)
      cout << *iElement << " ";
}

int main ()
{
   multiset <int> msetIntegers;

   msetIntegers.insert (5);
   msetIntegers.insert (5);
   msetIntegers.insert (5);

   set <int> setIntegers;
   setIntegers.insert (5);
   setIntegers.insert (5);
   setIntegers.insert (5);
```

```
    cout << "Wyświetlanie zawartości obiektu multiset: ";
    DisplayContent (msetIntegers);
    cout << endl;

    cout << "Wyświetlanie zawartości obiektu set: ";
    DisplayContent (setIntegers);
    cout << endl;

    return 0;
}
```

Lekcja 20. Klasy STL map

Quiz

1. Domyślne kryterium sortowania jest określane przez predykat sortowania `std::less<>`.
2. Tak, obok siebie.
3. Funkcja `size()`.
4. W obiekcie `map` nie znajdziesz powtarzających się elementów!

Ćwiczenia

1. Kontener asocjacyjny, który pozwala na przechowywanie powtarzających się elementów, np.:

```
std::multimap.
std::multimap <string, string> multimapPeopleNamesToNumbers;
```

2. Kontener asocjacyjny, który pozwala na przechowywanie powtarzających się elementów:

```
struct fPredicate
{
   bool operator< (const wordProperty& lsh, const wordProperty& rsh)
   ↪const
   {
      return (lsh.strWord < rsh. strWord);
   }
};
```

3. Wykorzystaj podpowiedź z rozwiązania bardzo podobnego ćwiczenia dla obiektów `set` i `multiset`.

Lekcja 21. Zrozumienie obiektów funkcji

Quiz

1. Predykat jednoargumentowy.
2. Może wyświetlać dane lub po prostu zliczać elementy.
3. W języku C++ wszystko to, co istnieje w trakcie działania aplikacji, nazywamy obiektami. W takim przypadku nawet struktury i klasy można zmusić do pracy w charakterze funkcji, stąd termin *obiekty funkcji*. Warto zwrócić uwagę na fakt, że funkcje również mogą być dostępne za pomocą wskaźników funkcji — takie funkcje także są obiektami.

Ćwiczenia

1. Poniżej przedstawiono jedno z możliwych rozwiązań:

```
template <typename elementType=int>
struct Double
{
   void operator () (const elementType element) const
   {
      cout << element * 2 << ' ';
   }
};
```

Ten predykat jednoargumentowy można wykorzystać następująco:

```
int main ()
{
   vector <int> vecIntegers;

   for (int nCount = 0; nCount < 10; ++ nCount)
         vecIntegers.push_back (nCount);

   cout << "Wyświetlanie obiektu vector przechowującego liczby
   ↪całkowite: " << endl;

   // Wyświetlenie tablicy liczb całkowitych.
   for_each ( vecIntegers.begin ()          // Początek zakresu.
         , vecIntegers.end ()                  // Koniec zakresu.
         , Double <> () ); // Obiekt funkcji jednoargumentowej.

   return 0;
}
```

2. Dodaj element składowy w postaci liczby całkowitej, która będzie inkrementowana podczas każdego użycia funkcji `operator()`:

```
template <typename elementType=int>
struct Double
{
   int m_nUsageCount;

   // Konstruktor.
   Double () : m_nUsageCount (0) {};

   void operator () (const elementType element)
   {
      ++ m_nUsageCount;
      cout << element * 2 << ' ';
   }
};
```

3. Predykat dwuargumentowy ma następującą postać:

```
template <typename elementType>
class CSortAscending
{
   public:
      bool operator () (const elementType& num1,
            const elementType& num2) const
      {
         return (num1 < num2);
      }
};
```

Ten predykat dwuargumentowy można wykorzystać następująco:

```
int main ()
{
   std::vector <int> vecIntegers;

   // Wstawienie przykładowych liczb: 100, 90... 20, 10.
   for (int nSample = 10; nSample > 0; -- nSample)
      vecIntegers.push_back (nSample * 10);

   std::sort ( vecIntegers.begin (), vecIntegers.end (),
      CSortAscending<int> () );

   for ( size_t nElementIndex = 0;
      nElementIndex < vecIntegers.size ();
      ++ nElementIndex )
      cout << vecIntegers [nElementIndex] << ' ';

   return 0;
}
```

Lekcja 22. Wyrażenia lambda w C++11

Quiz

1. Wyrażenia lambda zawsze zaczynają się od nawiasów kwadratowych [].
2. Przy użyciu listy przechwytywania: `[Zmienna1, Zmienna2, ...](Typ& parametr) { ...; }`.
3. Przykładowo w następujący sposób:

```
[Zmienna1, Zmienna2, ...](Typ& parametr) -> TypZwrotny { ...; }
```

Ćwiczenia

1. Poniżej przedstawiono przykładową wersję wyrażenia lambda:

```
sort(vecNumbers.begin(), vecNumbers.end(),
   [](int Num1, int Num2) {return (Num1 > Num2); } );
```

2. Poniżej przedstawiono przykładową wersję wyrażenia lambda:

```
cout << "Liczba, którą chcesz dodać do wszystkich elementów: ";
int NumInput = 0;
cin >> NumInput;

for_each(vecNumbers.begin(), vecNumbers.end(),
   [=](int& element) {element += NumInput;} );
```

Zaprezentowany poniżej przykładowy program zawiera rozwiązania ćwiczeń 1. i 2.

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;

template <typename T>
void DisplayContents (const T& Input)
{
   for(auto iElement = Input.cbegin() // auto, cbegin i cend: c++11
      ; iElement != Input.cend ()
      ; ++ iElement )
      cout << *iElement << ' ';
   cout << endl;
}

int main()
{
   vector<int> vecNumbers;
   vecNumbers.push_back(25);
```

```
    vecNumbers.push_back(-5);
    vecNumbers.push_back(122);
    vecNumbers.push_back(2011);
    vecNumbers.push_back(-10001);
    DisplayContents(vecNumbers);

    sort(vecNumbers.begin(), vecNumbers.end());
    DisplayContents(vecNumbers);

    sort(vecNumbers.begin(), vecNumbers.end(),
       [](int Num1, int Num2) {return (Num1 > Num2); } );
    DisplayContents(vecNumbers);

    cout << "Liczba, którą chcesz dodać do wszystkich elementów: ";
    int NumInput = 0;
    cin >> NumInput;

    for_each(vecNumbers.begin(), vecNumbers.end(),
       [=](int& element) {element += NumInput;} );

    DisplayContents(vecNumbers);

    return 0;
}
```

Wynik ▼

```
25 -5 122 2011 -10001
-10001 -5 25 122 2011
2011 122 25 -5 -10001
Liczba, którą chcesz dodać do wszystkich elementów: 5
2016 127 30 0 -9996
```

Lekcja 23. Algorytmy STL

Quiz

1. Użyj funkcji `std::list::remove_if`, ponieważ gwarantuje ona, że istniejące iteratory do elementów w obiekcie `list` (które nie zostały usunięte) nadal pozostaną ważne.
2. W przypadku wyraźnie podanego predykatu sortowania funkcja `list::sort` (lub nawet `std::sort`) przeprowadza sortowanie za pomocą predykatu `std::less<>`, który do posortowania obiektów w kolekcji wykorzystuje operator <.

3. Jednokrotnie względem każdego elementu w podanym zakresie.
4. Oba wymienione algorytmy zwracają obiekt funkcji.

Ćwiczenia

1. Poniżej przedstawiono jedno z możliwych rozwiązań:

```
struct CaseInsensitiveCompare
{
   bool operator() (const string& str1, const string& str2) const
   {
      string str1Copy (str1), str2Copy (str2);

      transform (str1Copy.begin (),
            str1Copy.end(), str1Copy.begin (), tolower);
      transform (str2Copy.begin (),
            str2Copy.end(), str2Copy.begin (), tolower);

      return (str1Copy < str2Copy);
   }
};
```

2. Poniżej przedstawiono jedno z możliwych rozwiązań. Warto zwrócić uwagę na fakt, że algorytm `std::copy` działa bez konieczności poznawania natury kolekcji. Algorytm ten działa przy użyciu jedynie iteratorów klasy.

```
#include <vector>
#include <algorithm>
#include <list>
#include <string>
#include <iostream>

using namespace std;

int main ()
{
   list <string> listNames;
   listNames.push_back ("Jacek");
   listNames.push_back ("Jan");
   listNames.push_back ("Anna");
   listNames.push_back ("Sławek");

   vector <string> vecNames (4);
   copy (listNames.begin (), listNames.end (), vecNames.begin ());

   vector <string> ::const_iterator iNames;
   for (iNames = vecNames.begin (); iNames != vecNames.end (); ++ iNames)
```

```
        cout << *iNames << ' ';

    return 0;
}
```

3. Różnica między algorytmami `std::sort` i `std::stable_sort` polega na tym, że ten drugi w trakcie sortowania gwarantuje zachowanie względnych pozycji obiektów. Ponieważ aplikacja musi przechowywać dane w sekwencji ich podawania, wybrałbym użycie algorytmu `stable_sort` w celu zachowania względnej kolejności wydarzeń zachodzących na niebie.

Lekcja 24. Kontenery adaptacyjne: stack i queue

Quiz

1. Tak, poprzez podanie predykatu.
2. Klasa `Coins` musi implementować operator <.
3. Nie, można uzyskać dostęp jedynie do elementu znajdującego się na górze stosu. Nie ma więc możliwości uzyskania dostępu do monety znajdującej się na dole stosu.

Ćwiczenia

1. Predykatem dwuargumentowym może być operator <:

```
class Person
{
   public:
      int Age;
      bool IsFemale;

      bool operator<  (const Person& anotherPerson) const
      {
         bool bRet = false;
         if (Age > anotherPerson.Age)
            bRet = true;
         else if (IsFemale && anotherPerson.IsFemale)
            bRet = true;
         return bRet;
      }
};
```

2. Po prostu umieść ten ciąg tekstowy na stosie. W trakcie usuwania danych nastąpi odwrócenie zawartości, ponieważ stos to typ kontenera „ostatni na wejściu, pierwszy na wyjściu”.

Lekcja 25. Praca z opcjami bitowymi za pomocą STL

Quiz

1. Nie. Liczba bitów przechowywanych przez obiekt `bitset` jest ustalana w trakcie kompilacji.
2. Ponieważ ten obiekt nie jest kontenerem STL. W przeciwieństwie do innych kontenerów nie ma możliwości dynamicznej zmiany swojej wielkości oraz nie obsługuje iteratorów w sposób, w jaki obsługiwane są przez inne kontenery.
3. Nie. Do tego celu najlepiej nadaje się kontener `std::bitset`.

Ćwiczenia

1. Nie. Do tego celu najlepiej nadaje się kontener `std::bitset`.

```
#include <bitset>
#include <iostream>

int main()
{
   // Inicjalizacja obiektu bitset z wartością 1001.
   std::bitset <4> fourBits (9);

   std::cout << "fourBits: " << fourBits << std::endl;

   // Inicjalizacja innego obiektu bitset z wartością 0010.
   std::bitset <4> fourMoreBits (2);

   std::cout << "fourMoreBits: " << fourMoreBits << std::endl;

   std::bitset<4> addResult(fourBits.to_ullong() +
    ↪fourMoreBits.to_ullong());
   std::cout << "Wynik dodawania jest następujący: " << addResult;

   return 0;
}
```

2. Wywołaj funkcję odwracającą bity względem dowolnego obiektu `bitset` z poprzedniego przykładu:

```
addResult.flip ();
std::cout << "Wynik odwrócenia bitów jest następujący: " << addResult <<
➥std::endl;
```

Lekcja 26. Sprytne wskaźniki

Quiz

1. Zajrzysz na witrynę *http://www.boost.org/*. Mam nadzieję, że właśnie tam!
2. Nie, dobrze zaprojektowany (i prawidłowo wybrany) sprytny wskaźnik nie wpływa na spowolnienie działania aplikacji.
3. Inwazyjnie w obiektach, do których prowadzą. W przeciwnym razie mogą przechowywać te informacje w obiektach współdzielonych umieszczonych na stosie.
4. Lista musi umożliwiać komunikację w dwóch kierunkach, a więc musi być listą dwukierunkową.

Ćwiczenia

1. Instrukcja `pObject->DoSomething();` jest błędna, ponieważ wskaźnik utracił prawa własności podczas etapu kopiowania. Ta instrukcja spowoduje awarię programu (albo wykona coś bardzo niepożądanego).
2. Poniżej przedstawiono przykładowy kod.

```
#include <memory>
#include <iostream>
using namespace std;
class Fish
{
public:
   Fish() {cout << "Fish: utworzono!" << endl;}
   ~Fish() {cout << "Fish: zniszczono!" << endl;}

   void Swim() const {cout << "Ryba pływa w wodzie" << endl;}
};

class Carp: public Fish
{
};
```

```
void MakeFishSwim(const unique_ptr<Fish>& inFish)
{
   inFish->Swim();
}

int main ()
{
   unique_ptr<Fish> myCarp (new Carp); // Zwróć uwagę na ten wiersz.
   MakeFishSwim(myCarp);

   return 0;
}
```

Ponieważ nie jest wykonywany krok kopiowania, a metoda `MakeFishSwim()` akceptuje argument przez referencję, nie ma niebezpieczeństwa segmentowania. Ponadto zwróć uwagę na składnię tworzenia zmiennej `myCarp`.

3. Wskaźnik `unique_ptr` nie pozwala na kopiowanie lub przypisywanie, ponieważ zarówno konstruktor kopiujący, jak i kopiujący operator przypisania zostały zdefiniowane jako prywatne.

Lekcja 27. Użycie strumieni w operacjach wejścia-wyjścia

Quiz

1. Strumienia `ofstream` używaj jedynie w celu zapisu danych w pliku.
2. Powinieneś użyć `cin.getline()`. Spójrz na listing 27.7.
3. Nie powinieneś, ponieważ obiekt `std::string` zawiera informacje tekstowe, a więc możesz pozostać z trybem domyślnym, który obsługuje tekst (nie trzeba korzystać z pliku binarnego).
4. Sprawdź, czy wywołanie metody `open()` zakończyło się powodzeniem.

Ćwiczenia

1. Otworzyłeś plik, ale przed użyciem strumienia lub zamknięciem pliku nie sprawdziłeś przy użyciu `is_open()`, czy wywołanie metody `open()` zakończyło się powodzeniem.

2. Nie możesz wstawić danych do `ifstream`, ponieważ ten strumień jest przeznaczony dla danych wejściowych, a nie wyjściowych, i nie zapewnia obsługi operatora wstawiania do strumienia (<<).

Lekcja 28. Obsługa wyjątków

Quiz

1. Klasa jak każda inna, ale utworzona jako klasa bazowa dla innych klas wyjątków, takich jak `bad_alloc`.
2. `std::bad_alloc`.
3. To kiepski pomysł, istnieje możliwość, że wyjątek zostanie zgłoszony w pierwszej kolejności na skutek braku wolnej pamięci.
4. Używając tego samego polecenia `catch(std::exception& exp)`, jakie jest wykorzystywane dla typu `bad_alloc`.

Ćwiczenia

1. Nigdy nie należy zgłaszać wyjątków w destruktorze.
2. Zapomniałeś o zapewnieniu bezpieczeństwa wyjątków w kodzie (pominięty blok `try-catch`).
3. Nie przeprowadzaj alokacji pamięci w bloku `catch`! Nie dla miliona liczb całkowitych. Przyjmij założenie, że dane zaalokowane w `try` zostały utracone i kontynuuj działanie, kontrolując szkody.

Lekcja 29. Co dalej?

Quiz

1. Wydaje się, że wszystkie operacje aplikacji są przeprowadzane w jednym wątku. Dlatego też jeśli zadanie przetwarzania obrazu (korekcji kontrastu) stanowi duże obciążenie dla procesora, wtedy interfejs użytkownika z opóźnieniem reaguje na polecenia wydawane przez użytkownika. Te dwie aktywności należy podzielić na dwa wątki, a system operacyjny będzie się przełączał pomiędzy nimi, przydzielając

czas procesora zarówno interfejsowi użytkownika, jak i wątkowi roboczemu wykonującemu korekcję obrazu.

2. Prawdopodobnie wątki są kiepsko zsynchronizowane. Jednocześnie przeprowadzane operacje odczytu i zapisu obiektu powodują, że dane są niespójne lub uszkodzone. Umieść w kodzie binarny semafor i zagwarantuj, że dana tabela będzie niedostępna w trakcie przeprowadzania jej modyfikacji.

Dodatek E

Kody ASCII

Komputery działają, używając bitów i bajtów, czyli w zasadzie liczb. Aby przedstawić znak w systemie liczbowym, opracowano standard ASCII (ang. *American Standard Code for Information Interchange*), który jest powszechnie stosowany. W standardzie ASCII ośmiobitowe kody liczbowe odpowiadają znakom alfabetu łacińskiego, czyli literom od A do Z, od a do z, cyfrom od 0 do 9, pewnym specjalnym kombinacjom (np. DEL) i znakom (np. Backspace).

Siedem bitów pozwala na uzyskanie 128 kombinacji, z których pierwsze 32 (od 0 do 31) są zarezerwowane jako znaki kontrolne używane do współpracy z urządzeniami peryferyjnymi, takimi jak drukarki itd.

Tabela ASCII znaków wyświetlanych na ekranie

Kody ASCII od 32 do 127 są używane do określenia znaków wyświetlanych na ekranie, takich jak od 0 do 9, od A do Z, od a do z i innych, np. spacji. W przedstawionej poniżej tabeli wymieniono dziesiętne i szesnastkowe wartości zarezerwowane dla wspomnianych znaków.

Znak	Wartość dziesiętna	Wartość szesnastkowa	Nazwa
	32	20	Spacja
!	33	21	Wykrzyknik
"	34	22	Znak cudzysłowu
#	35	23	Znak liczby
$	36	24	Znak dolara
%	37	25	Znak procentu
&	38	26	Znak ampersand
'	39	27	Apostrof
(	40	28	Lewy nawias (otwierający)
)	41	29	Prawy nawias (zamykający)
*	42	2A	Gwiazdka
+	43	2B	Znak plus
,	44	2C	Przecinek
-	45	2D	Minus
.	46	2E	Kropka
/	47	2F	Ukośnik (slash)
0	48	30	Cyfra zero
1	49	31	Cyfra jeden
2	50	32	Cyfra dwa
3	51	33	Cyfra trzy
4	52	34	Cyfra cztery
5	53	35	Cyfra pięć
6	54	36	Cyfra sześć
7	55	37	Cyfra siedem

Znak	Wartość dziesiętna	Wartość szesnastkowa	Nazwa
8	56	38	Cyfra osiem
9	57	39	Cyfra dziewięć
:	58	3A	Dwukropek
;	59	3B	Średnik
<	60	3C	Znak mniejszości (ostry nawias otwierający)
=	61	3D	Znak równości
>	62	3E	Znak większości (ostry nawias zamykający)
?	63	3F	Znak zapytania
@	64	40	Znak at
A	65	41	Wielka litera A
B	66	42	Wielka litera B
C	67	43	Wielka litera C
D	68	44	Wielka litera D
E	69	45	Wielka litera E
F	70	46	Wielka litera F
G	71	47	Wielka litera G
H	72	48	Wielka litera H
I	73	49	Wielka litera I
J	74	4A	Wielka litera J
K	75	4B	Wielka litera K
L	76	4C	Wielka litera L
M	77	4D	Wielka litera M
N	78	4E	Wielka litera N
O	79	4F	Wielka litera O
P	80	50	Wielka litera P
Q	81	51	Wielka litera Q
R	82	52	Wielka litera R
S	83	53	Wielka litera S
T	84	54	Wielka litera T
U	85	55	Wielka litera U

Znak	Wartość dziesiętna	Wartość szesnastkowa	Nazwa
V	86	56	Wielka litera V
W	87	57	Wielka litera W
X	88	58	Wielka litera X
Y	89	59	Wielka litera Y
Z	90	5A	Wielka litera Z
[	91	5B	Lewy nawias kwadratowy (nawias otwierający)
\	92	5C	Ukośnik (backslash)
]	93	5D	Prawy nawias kwadratowy (nawias zamykający)
^	94	5E	Daszek
_	95	5F	Podkreślenie
`	96	60	Akcent
a	97	61	Mała litera A
b	98	62	Mała litera B
c	99	63	Mała litera C
d	100	64	Mała litera D
e	101	65	Mała litera E
f	102	66	Mała litera F
g	103	67	Mała litera G
h	104	68	Mała litera H
i	105	69	Mała litera I
j	106	6A	Mała litera J
k	107	6B	Mała litera K
l	108	6C	Mała litera L
m	109	6D	Mała litera M
n	110	6E	Mała litera N
o	111	6F	Mała litera O
p	112	70	Mała litera P
q	113	71	Mała litera Q
r	114	72	Mała litera R

Znak	Wartość dziesiętna	Wartość szesnastkowa	Nazwa
s	115	73	Mała litera S
t	116	74	Mała litera T
u	117	75	Mała litera U
v	118	76	Mała litera V
w	119	77	Mała litera W
x	120	78	Mała litera X
y	121	79	Mała litera Y
z	122	7A	Mała litera Z
{	123	7B	Lewy nawias klamrowy (nawias otwierający)
|	124	7C	Pionowa kreska
}	125	7D	Prawy nawias klamrowy (nawias zamykający)
~	126	7E	Tylda
DEL	127	7F	Delete

Skorowidz

A

B

C

D

E

F

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z