



DO NOWEJ
PODSTAWY PROGRAMOWEJ

Część 2

Programowanie obiektowe

Kwalifikacja INF.04

Projektowanie, programowanie
i testowanie aplikacji



**Podręcznik do nauki zawodu
technik programista**

Piotr Siewniak

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicielami.

Autor oraz wydawca dolożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Joanna Zaręba

Projekt okładki: Jan Paluch

Ilustracja na okładce została wykorzystana za zgodą Shutterstock.

Helion S.A.

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)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie?inf042_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-8767-6

Copyright © Helion S.A. 2021

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	7
Rozdział 1. Wprowadzenie do programowania	9
1.1. Podstawowe pojęcia	9
1.2. Podstawy algorytmiki	13
1.3. Pytania i zadania kontrolne	27
Rozdział 2. Środowiska programistyczne	29
2.1. Code::Blocks	30
2.2. CodeLite	31
2.3. Eclipse	33
2.4. NetBeans	33
2.5. Visual Studio	35
2.6. CLion	36
2.7. Pytania i zadania kontrolne	37
Rozdział 3. Podstawy programowania	39
3.1. Podstawowe elementy języka C++	39
3.2. Podejmowanie decyzji w programie	72
3.3. Pętle programowe	85
3.4. Typ wyliczeniowy	95
3.5. Pytania i zadania kontrolne	98
Rozdział 4. Programowanie z użyciem wskaźników	103
4.1. Operator adresu	103
4.2. Wskaźniki	106
4.3. Dynamiczna alokacja pamięci	114
4.4. Pytania i zadania kontrolne	119
Rozdział 5. Tablice i wektory	121
5.1. Tablice statyczne	121
5.2. Tablice i wskaźniki	132
5.3. Tablice dynamiczne i wektory	137

5.4. Pętla foreach	142
5.5. Pytania i zadania kontrolne	144
Rozdział 6. Łańcuchy znaków	147
6.1. C-napisy	148
6.2. Łańcuchy typu string	157
6.3. Pytania i zadania kontrolne	168
Rozdział 7. C-struktury i C-unie	171
7.1. C-struktury	171
7.2. C-unie	184
7.3. Pytania i zadania kontrolne	190
Rozdział 8. Funkcje	193
8.1. Deklarowanie i definiowanie funkcji	194
8.2. Wywołanie funkcji	196
8.3. Parametry funkcji	198
8.4. Zmienne globalne i lokalne	224
8.5. Funkcje przeciążone	233
8.6. Funkcje inline	235
8.7. Funkcje rekurencyjne	236
8.8. Pytania i zadania kontrolne	238
Rozdział 9. Preprocesor	241
9.1. Dyrektywa #include	241
9.2. Dyrektywa #define	245
9.3. Dyrektywy komplikacji warunkowej	251
9.4. Pytania i zadania kontrolne	253
Rozdział 10. Funkcje biblioteczne	255
10.1. Funkcje matematyczne	256
10.2. Funkcje znakowe	258
10.3. Konwersja typu danych	260
10.4. Funkcje wejścia/wyjścia	261
10.5. Przykłady innych funkcji bibliotecznych	263
10.6. Pytania i zadania kontrolne	264

Rozdział 11. Klasy i obiekty	267
11.1. Wprowadzenie do programowania obiektowego	267
11.2. Definiowanie klas	273
11.3. Deklarowanie zmiennych obiektowych	278
11.4. Odwołania do elementów członkowskich obiektów	279
11.5. Statyczne elementy członkowskie klas	283
11.6. Funkcje członkowskie typu inline	288
11.7. Wskaźniki do obiektów	290
11.8. Przekazywanie obiektów jako parametrów funkcji	292
11.9. Struktury w języku C++	296
11.10. Pytania i zadania kontrolne	301
Rozdział 12. Tworzenie i inicjowanie obiektów	303
12.1. Konstruktory	303
12.2. Inicjalizacja obiektów	315
12.3. Konstruktor kopiący	328
12.4. Delegowanie konstruktorów	332
12.5. Destruktory	337
12.6. Pytania i zadania kontrolne	340
Rozdział 13. Hermetyzacja i ukrywanie danych	343
13.1. Hermetyzacja danych	343
13.2. Ukrywanie danych	345
13.3. Pytania i zadania kontrolne	348
Rozdział 14. Mechanizm dziedziczenia	351
14.1. Definicja relacji dziedziczenia	352
14.2. Rodzaje dziedziczenia	358
14.3. Dziedziczenie a konstruktory	362
14.4. Pytania i zadania kontrolne	365
Rozdział 15. Polimorfizm	367
15.1. Polimorfizm statyczny	367
15.2. Polimorfizm dynamiczny	373
15.3. Pytania i zadania kontrolne	381

Rozdział 16. Mechanizm abstrakcji	383
16.1. Klasy abstrakcyjne	384
16.2. Interfejsy	389
16.3. Abstrakcja danych	393
16.4. Mechanizm abstrakcji a pliki nagłówkowe	396
16.5. Pytania i zadania kontrolne	399
Rozdział 17. Funkcje i klasy zaprzyjaźnione	401
17.1. Funkcje zaprzyjaźnione	401
17.2. Klasy zaprzyjaźnione	407
17.3. Pytania i zadania kontrolne	410
Rozdział 18. Szablony funkcji i klas	413
18.1. Szablony funkcji	413
18.2. Szablony klas	420
18.3. Szablony a polimorfizm	428
18.4. Pytania i zadania kontrolne	429
Rozdział 19. Obsługa błędów i wyjątków	431
19.1. System komunikatów i kodów zwrotnych	432
19.2. Mechanizm obsługi wyjątków	440
19.3. Pytania i zadania kontrolne	456
Rozdział 20. Przykłady implementacji wybranych algorytmów	459
20.1. Wyznaczenie największego wspólnego dzielnika	459
20.2. Sortowanie tablic	463
20.3. Wyszukiwanie binarne	467
20.4. Pytania i zadania kontrolne	470
Bibliografia	471
Skorowidz	472

Wstęp

Niniejsza publikacja jest podręcznikiem do nauki zawodu technik programista, a w szczególności do nauki *programowania obiektowego* (ang. *object oriented programming*).

Zakres omawianego materiału obejmuje wszystkie efekty kształcenia wymienione w dziale *INF.04.4. Programowanie obiektowe* kwalifikacji *INF.04. Projektowanie, programowanie i testowanie aplikacji* w podstawie programowej dla zawodu technik programista. Dotyczy to również kryteriów weryfikacji osiągnięć edukacyjnych ucznia.

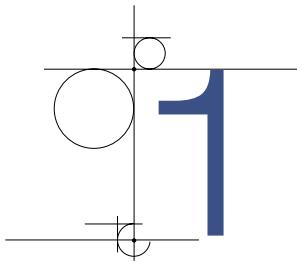
Podręcznik składa się z 20 rozdziałów, które można podzielić umownie na dwie części. W pierwszej części, obejmującej rozdziały od 1. do 10., przedstawiono podstawy programowania w języku C++. W drugiej zaś, w rozdziałach od 11. do 20., omówiono zagadnienia dotyczące programowania zorientowanego obiektowo.

W podręczniku uwzględniono wszystkie najważniejsze zasady (paradygmaty) programowania: zarówno programowanie imperatywne, proceduralne i strukturalne (rozdziały od 1. do 10.), jak też programowanie obiektowe, tj. abstrakcję, enkapsulację, mechanizm dziedziczenia i polimorfizm (rozdziały od 11. do 20.).

Duży nacisk położono na umiejętności praktyczne ucznia. Dlatego też podręcznik zawiera wiele gotowych przykładów praktycznych ilustrujących omawiane zagadnienia. Każdy z przykładów został obszernie skomentowany i wyjaśniony. Nowo zdobytą wiedzę pozwalającą ugruntować ćwiczenia praktyczne dotyczące omawianej tematyki, przeznaczone do samodzielnego wykonania przez ucznia na zajęciach dydaktycznych w szkole lub w domu, a także zamieszczone na końcu każdego rozdziału pytania i zadania kontrolne związane z zaprezentowanym materiałem.

Wybór języka C++ do nauki programowania obiektowego jest podkutowany terminologią użytą w podstawie programowej dla zawodu w dziale *INF.04.4. Programowanie obiektowe* w opisie efektów kształcenia i kryteriów ich weryfikacji. W szczególności dotyczy to niektórych pojęć i terminów, charakterystycznych wyłącznie dla języka C++. Przykładem jest pojęcie *klasy zaprzyjaźnionej* (ang. *friend class*). W innych językach można oczywiście stosować znaną z C++ ideę klas zaprzyjaźnionych, ale tylko przez użycie równoważnych lub przybliżonych funkcjonalnie zamienników, np. *klas zagnieżdżonych* (ang. *nested classes*) lub modyfikatora *internal* w C# czy też *pakietów* (ang. *packages*) w Javie.

Każdy uczeń kształcący się w zawodzie technik programista musi być w pełni świadomy tego, że bez gruntowej znajomości i umiejętności programowania obiektowego będzie mu bardzo trudno programować aplikacje desktopowe, mobilne i internetowe, a tym samym osiągnąć główne cele kształcenia w zakresie kwalifikacji zawodowej INF.04.



Wprowadzenie do programowania

W terminologii programistycznej można się często spotkać z określeniem **inżynieria oprogramowania** (ang. *software engineering*). Znaczenie tego terminu jest bardzo szerokie, ponieważ obejmuje różne fazy tworzenia oprogramowania komputerowego — począwszy od zdefiniowania i opisania problemu do rozwiązania (analiza i określenie wymagań), poprzez jego planowanie, kodowanie, testowanie, dokumentowanie i wdrożenie, a skończywszy na konserwacji i obsłudze technicznej.

W zakres inżynierii oprogramowania wchodzi faza, którą można określić ogólnym terminem **projektowanie oprogramowania** (ang. *software design*). Termin ten oznacza zdefiniowanie i opisanie rozwiązania zadanego problemu oraz czynności związane z planowaniem tego rozwiązania. Następną fazę tworzenia programu można nazwać procesem jego kodowania lub bardziej ogólnie **implementacją** (ang. *implementation*). W ogólności implementacja może obejmować kilka działań (czynności), które są ze sobą wzajemnie powiązane, np. kodowanie, weryfikację czy też walidację poszczególnych elementów składowych (modułów) programu. Po zintegrowaniu ze sobą poszczególnych modułów gotowy i kompletny produkt (program) należy poddać gruntownym, systematycznym testom. Równolegle z innymi fazami należy program dokumentować. **Dokumentacja programu** (ang. *program documentation*) zawiera część techniczną i część eksploatacyjną, które są bardzo pomocne podczas jego wdrożenia i późniejszej konserwacji.

1.1. Podstawowe pojęcia

Programowanie komputerowe (ang. *computer programming*) to proces tworzenia programu komputerowego. Przy czym **program komputerowy** (ang. *computer program*) jest zestawem instrukcji (komend, poleceń) dla komputera, przygotowanym w celu uzyskania określonego rezultatu lub wykonania postawionego zadania.

Proces tworzenia programu komputerowego obejmuje następujące etapy:

1. Zdefiniowanie problemu do rozwiązania.
2. Planowanie rozwiązania.
3. Kodowanie programu.
4. Testowanie programu.
5. Dokumentowanie programu.

Zdefiniowanie problemu do rozwiązania polega na zidentyfikowaniu i określeniu wejścia oraz wyjścia programu. Wejście programu opisuje dane wejściowe, które program ma przetwarzać. Opis ten powinien uwzględniać analizę danych wejściowych zarówno pod względem jakościowym (np. rodzaje — typy danych, źródła danych), jak i ilościowym (np. liczba danych należących do poszczególnych grup tematycznych). Wyjście programu zaś powinno stanowić odpowiedź na pytanie: co ma być wynikiem (wynikami) działania programu? W tym przypadku również należy przeprowadzić analizę jakościową i ilościową.

Proces definiowania problemu jest zadaniem bardzo trudnym. Niejednokrotnie wymaga wielu konsultacji ze zleceniodawcą (użytkownikiem końcowym programu) i żmudnej, czasochłonnej analizy. Rezultatem tego etapu może być pisemna umowa, w której możliwie dokładnie i szczegółowo opisuje się wymagane wejście i wyjście programu.

Planowanie rozwiązania problemu to drugi etap procesu tworzenia programu komputerowego. Jest ono zwykle realizowane na jeden z dwóch sposobów. Pierwszy polega na sporządzeniu schematu blokowego algorytmu rozwiązania problemu. Przy czym **algorytm** (ang. *algorithm*) należy rozumieć jako ciąg (zestaw) czynności lub sposób postępowania prowadzący do rozwiązania problemu albo wykonania określonego zadania w skończonej liczbie kroków (skończonym czasie), a **schemat blokowy** (ang. *block diagram*) to mapa graficzna opisująca jednoznacznie logikę i funkcjonalność programu (algorytmu): jego poszczególne kroki, wzajemne uwarunkowania i zależności, metody rozwiązania problemów częściowych itd.

UWAGA

Podstawy algorytmiki, w tym pojęcia algorytmu i schematu blokowego, zostały omówione szczegółowo w dalszej części rozdziału — w podrozdziale 1.2.

Drugi z wymienionych sposobów planowania rozwiązania problemu polega na wykorzystaniu tzw. **pseudokodu** (ang. *pseudocode*). Pseudokod pozwala precyzyjnie i szczegółowo zapisać algorytm programu za pomocą konstrukcji programistycznych zapożyczonych z wybranego popularnego języka programowania (np. struktur warunkowych, pętli, podprogramów), zgodnych z określoną konwencją i określonym stylem, założonymi i przyjętymi w projekcie. Z drugiej strony pseudokod — z założenia — nie jest zgodny z regułami składniowymi żadnego konkretnego języka programowania. Jego forma i styl są zasadniczo dowolne. Pseudokod

jest przystosowany do czytania przez człowieka — programistę, a nie przez komputer. Ważne jest to, żeby algorytm zapisany w pseudokodzie był zrozumiały dla możliwie dużej liczby programistów, niezależnie od tego, jakie języki programowania preferują.

Proces planowania rozwiązania postawionego problemu programistycznego może obejmować również inne elementy, np. wybór platformy sprzętowej, na której program będzie pracował, wybór systemu operacyjnego i środowiska programistycznego, przydział zadań dla poszczególnych programistów w zespole itp.

Następny etap — **kodowanie programu** — polega na zapisaniu algorytmu programu (w postaci schematu blokowego i/lub pseudokodu) w konkretnym języku (językach) programowania. **Język programowania** (ang. *computer language*) najprościej można określić jako zbiór reguł (zasad), które zapewniają skutecną komunikację programisty z maszyną (komputerem) w celu realizacji przez komputer wymaganych zadań — operacji. Operacje te są opisane w sposób jednoznaczny w programie komputerowym, który tworzy (pisze) programista.

Języki programowania można podzielić według różnych kryteriów. Na przykład języki programowania do tworzenia aplikacji internetowych można podzielić na takie, które zapewniają interfejs (interakcję) aplikacji z użytkownikiem po stronie przeglądarki (klienta), m.in. JavaScript, oraz takie, które odpowiadają za funkcjonalność aplikacji po stronie serwera, choćby C++, Java, C#. Inne kryterium podziału języków programowania jest związane ze sposobem konwersji — translacji (ang. *translation*) kodu źródłowego (ang. *source code*) programu napisanego w języku wysokiego poziomu (ang. *high-level computer language*), zrozumiałym dla programisty, na kod wynikowy maszynowy (ang. *machine code*), zrozumiałym dla maszyny (komputera). Według tego kryterium języki programowania dzielą się na komplilowane i interpretowane. **Język komplilowany** (ang. *compiled language*) to taki, w którym wspomniany powyżej proces translacji polega na komplikacji. W czasie komplikacji program źródłowy (traktowany jako całość) jest tłumaczyony na kod pośredni, który jest kodem obiektowym (ang. *object code*), a następnie integrowany z innymi elementami składowymi aplikacji w procesie nazywanym linkowaniem, inaczej łączeniem (ang. *linking*). Rezultatem komplikacji jest program wykonywalny (ang. *executable program*), który można wielokrotnie wykonywać bez potrzeby jego ponownej komplikacji. Za komplikację programu odpowiada program nazywany **kompilatorem** (ang. *compiler*). Do popularnych kompilatorów języka C++ należą m.in. g++ (GCC) i Microsoft Visual C++. Z drugiej strony **języki interpretowane** (ang. *interpreted languages*) pozwalają na translację kodu źródłowego programu przez jego interpretowanie — za co odpowiadają interpretery. **Interpreter** (ang. *interpreter*) tłumaczy kod programu instrukcja po instrukcji i wykonuje każdą z nich zaraz po przetłumaczeniu. Proces ten jest realizowany krok po kroku aż do końca programu. Kolejne wykonanie programu jest związane z jego ponowną interpretacją instrukcja po instrukcji. Do najbardziej popularnych języków interpretowanych należą JavaScript i PHP.

Program może się składać z większej liczby elementów — modułów funkcjonalnych. Moduły te mogą być napisane w różnych językach programowania, pozwalających modułom wymieniać między sobą informacje (np. dane). Najprostszym przykładem ilustrującym taką

sytuację są aplikacje bazodanowe, w których do wykonania interfejsu aplikacja-użytkownik wykorzystuje się jakiś uniwersalny język programowania, np. C++, PHP, a przetwarzanie danych w bazie jest realizowane za pomocą języka specjalistycznego, np. SQL.

Wybór języka (języków) programowania jest ściśle powiązany z doborem odpowiedniego środowiska programistycznego, które programista będzie wykorzystywał do tworzenia aplikacji. Zintegrowane środowisko programistyczne, **IDE** (ang. *integrated development environment*), to narzędzie — program (lub grupa programów) — służące do tworzenia, modyfikowania, testowania i konserwacji oprogramowania. Jako przykłady popularnych komercyjnych zintegrowanych środowisk programistycznych można wymienić Microsoft Visual Studio Professional i JetBrains IDEA IntelliJ Ultimate.

Po zakończeniu etapu kodowania programu w konkretnym języku (językach) programowania należy go poddać gruntownym testom. Jednakże testowanie programu jest często prowadzone równolegle z kodowaniem jego poszczególnych modułów. Dotyczy to zwłaszcza sprawdzania poprawności działania algorytmu, funkcjonalności poszczególnych modułów oraz poprawności interakcji pomiędzy modułami.

W ogólności **testowanie programu** obejmuje jego weryfikację oraz walidację. **Weryfikacja programu** (ang. *program verification*) polega na udowodnieniu, że spełnia on założone wymagania dotyczące funkcjonalności, które są zazwyczaj określone w formalnej specyfikacji. Innymi słowy, weryfikacja programu pozwala odpowiedzieć na pytanie, czy praca została wykonana rzetelnie. **Walidacja programu** (ang. *program validation*) zaś daje odpowiedź na pytanie, czy program jest w pełni dostosowany do potrzeb zamawiającego (użytkownika końcowego).

Testy programów można umownie podzielić na kilka poziomów:

- testy jednostkowe,
- testy akceptacyjne,
- testy systemowe,
- testy integracyjne i systemowe elementów (komponentów) składowych.

Z etapem kodowania i testowania jest ściśle powiązany proces **debugowania** (ang. *debugging*). Pojęcie to oznacza wykrywanie, lokalizowanie, eliminowanie i korygowanie błędów logicznych w programie. Debugowanie jest realizowane w praktyce za pomocą programów nazywanych **debuggerami** (ang. *debuggers*). Niektóre środowiska programistyczne mają wbudowane debuggery, w innych trzeba korzystać z debuggerów zewnętrznych (np. GDB), które należy zintegrować z wykorzystywanym środowiskiem i skonfigurować pod kątem własnych potrzeb.

Ważnym elementem procesu tworzenia programu jest również **optymalizacja programu** (ang. *program optimization*). Celem optymalizacji może być zwiększenie prędkości działania programu (np. wyszukiwania danych) i/lub zmniejszenie jego objętości (rozmiaru). W ogólności optymalizacja programu poprawia jego wydajność (efektywność) w odniesieniu do wybranego kryterium (kryteriów).

Istnieje kilka poziomów optymalizacji programu, m.in.:

- optymalizacja na poziomie projektowania (ang. *design level optimization*),
- optymalizacja algorytmów (ang. *algorithms optimization*),
- optymalizacja struktur danych (ang. *data structures optimization*),
- optymalizacja kodu źródłowego (ang. *source code optimization*).

Dokumentowanie jest ostatnim etapem tworzenia programu. To pisemny, szczegółowy opis procesu tworzenia programu oraz samego programu. Zasadniczo dokumentacja programu składa się z dwóch części:

- dokumentacji technicznej,
- dokumentacji użytkownika końcowego.

Dokumentacja techniczna (ang. *technical documentation*) programu obejmuje opis problemu, schematy blokowe i/lub pseudokod, opisy rekordów (struktur) danych, listę modułów, opis ważniejszych podprogramów, wyniki przeprowadzonych testów itp. W skład dokumentacji technicznej programu wchodzą także komentarze zawarte przez programistę (programistów) w jego kodzie źródłowym.

Dokumentacja użytkownika końcowego (ang. *end-user documentation*) składa się zazwyczaj ze szczegółowej instrukcji obsługi programu w postaci papierowej i/lub elektronicznej wraz z opisem konkretnych przykładów zastosowań. Ponadto ta część dokumentacji może obejmować wymagania techniczne (sprzętowe i programowe), opis procesu instalacji, opis procesu archiwizacji danych itp.

Dokumentacja programu to jego bardzo istotny element, ponieważ umożliwia zapoznanie się z całym programem i jego poszczególnymi elementami składowymi członkom zespołu programistów i/lub innej instytucji, która np. otrzymała zlecenie na rozbudowę i/lub modyfikację programu. To samo dotyczy jego innych użytkowników końcowych.

1.2. Podstawy algorytmiki

Algorytmika to dział informatyki, który obejmuje tematykę projektowania i analizy algorytmów. Przy czym termin „algorytm” oznacza ciąg (zestaw) czynności lub sposób postępowania prowadzący do rozwiązania problemu albo wykonania określonego zadania w skończonej liczbie kroków (skończonym czasie). Była o tym już mowa w poprzednim podrozdziale. Algorytm można zaimplementować w postaci programu komputerowego.

Zadaniem algorytmu jest zmiana — przeprowadzenie zadanego stanu początkowego, opisanego w programie np. za pomocą wartości początkowych określonych zmiennych — w wymagany stan końcowy, reprezentowany np. przez wartości końcowe tych zmiennych. Wspomniane zmienne mogą być zorganizowane w strukturach danych zdefiniowanych w programie.

Każdy algorytm powinien spełniać następujące kryteria:

- dane wejściowe powinny być określone w sposób jednoznaczny, z uwzględnieniem ich liczby oraz typów (rodzajów),
- sterowanie przebiegiem działania programu (algorytmu) powinno zapewniać uzyskanie wyniku (wyników),
- liczba kroków zmierzających do uzyskania wyniku (rezultatu) musi być skończona,
- każdy krok (operacja) powinien być zdefiniowany w sposób jednoznaczny.

Algorytmy programów można zapisywać w różny sposób, np. w postaci:

- schematu blokowego,
- pseudokodu,
- opisu słownego,
- listy kroków,
- drzewa (grafu)

oraz oczywiście wybranego języka programowania.

1.2.1. Schematy blokowe algorytmów

Schemat blokowy (ang. *block diagram, flowchart, flow chart*) algorytmu składa się z **bloków** (ang. *blocks*) funkcjonalnych reprezentujących operacje (polecenia, instrukcje). Różnym grupom poleceń — np. operacjom wejścia/wyjścia, strukturom warunkowym — odpowiadają różne kształty bloków. Ponadto każdy schemat blokowy zawiera **linie — strzałki** (ang. *arrowheads, flowlines*), które określają wzajemne powiązania funkcjonalne pomiędzy blokami (instrukcjami) oraz kolejność ich wykonywania.

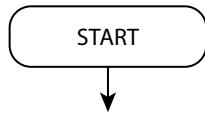
Do najważniejszych bloków, z których budowane są schematy blokowe, należą:

- blok początkowy,
- blok końcowy,
- blok wejścia/wyjścia,
- blok operacji (przetwarzania),
- blok decyzyjny,
- blok podprogramu,
- łączniki,
- punkty koncentracji,
- blok komentarza.

Kształty i zasady używania wymienionych powyżej symboli na schematach blokowych algorytmów programów komputerowych są znormalizowane — standaryzowane. Opisuje je norma ISO 5807 z 1985 roku, określona przez ANSI (American National Standards Institute).

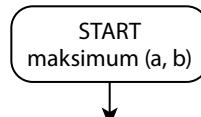
Blok początkowy

Blok początkowy (ang. *start block*, *start terminal*) reprezentuje na schemacie blokowym początek algorytmu programu lub podprogramu, np. funkcji. Wewnątrz bloku znajduje się słowo START, jeśli algorytm dotyczy całego programu (rysunek 1.1), albo słowo START wraz z nazwą podprogramu oraz ewentualnie jego parametrami/argumentami, jeśli algorytm opisuje podprogram (rysunek 1.2).



Rysunek 1.1.

Blok początkowy algorytmu programu



Rysunek 1.2.

Przykład bloku początkowego algorytmu podprogramu

Na rysunku 1.2 pokazano przykładowy blok początkowy algorytmu podprogramu o nazwie *maksimum* z parametrami/argumentami *a* i *b*.

Schemat blokowy algorytmu podprogramu opisuje definicję tego podprogramu. Schemat ten może być w pełni niezależny od innych schematów blokowych lub stanowić część (element składowy) większego, złożonego schematu. W pierwszym przypadku w nawiasach okrągłych podaje się nazwy parametrów formalnych podprogramu, które są w pełni niezależne od miejsca jego wywołania (otoczenia). Jeśli zaś schemat blokowy algorytmu podprogramu stanowi część składową innego schematu blokowego, wówczas w nawiasach okrągłych należy wypisać argumenty wywołania tego podprogramu.



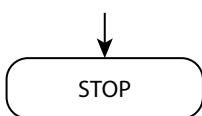
UWAGA

Podprogramy w języku C++ — funkcje, w tym zasady ich definiowania i wywoływanie oraz parametry i argumenty — zostały omówione szczegółowo w rozdziale 8. podręcznika.

Blok początkowy może występować w schemacie blokowym algorytmu programu (oraz niezależnego od innych schematu blokowego algorytmu definicji podprogramu) wyłącznie raz.

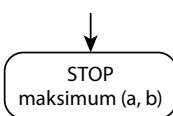
Blok końcowy

Blok końcowy (ang. *stop block*, *stop terminal*) oznacza koniec schematu blokowego algorytmu programu (rysunek 1.3) albo algorytmu podprogramu (funkcji) (rysunek 1.4 i rysunek 1.5). Blok końcowy może występować w schemacie blokowym programu (podprogramu) więcej niż raz.



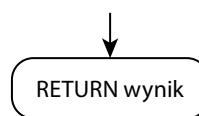
Rysunek 1.3.

Blok końcowy algorytmu programu



Rysunek 1.4.

Przykład bloku końcowego algorytmu podprogramu



Rysunek 1.5.

Przykład bloku końcowego na schemacie algorytmu podprogramu

Na rysunku 1.4 pokazano przykład bloku końcowego na schemacie algorytmu podprogramu o nazwie *maksimum* z parametrami/argumentami *a*, *b*.

Jeżeli schemat blokowy dotyczy algorytmu podprogramu i jest niezależny od innych schematów blokowych, wówczas w nawiasach okrągłych należy podać nazwy parametrów tego podprogramu. Jeśli zaś schemat blokowy algorytmu podprogramu stanowi część (element składowy) innego schematu blokowego, wówczas w nawiasach okrągłych należy wypisać argumenty wywołania tego podprogramu.

Jeśli podprogram zwraca na zewnątrz wartość „za pośrednictwem swojej nazwy”, wówczas koniec jego algorytmu może być oznaczony za pomocą słowa RETURN wraz z określeniem zwracanej wartości (rysunek 1.5). Jest to informacja o powrocie do otoczenia podprogramu, np. do programu głównego.

Na rysunku 1.5 zaprezentowano przykład bloku końcowego na schemacie algorytmu podprogramu, który zwraca na zewnątrz „za pośrednictwem swojej nazwy” wartość przechowywaną w zmiennej o nazwie *wynik*.

UWAGA

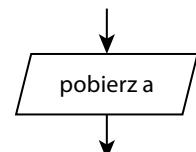
Ujmując rzecz najprościej, zmienna to element programu komputerowego, w którym przechowywana jest wartość określonego typu, np. liczba lub napis. Zmienna może zmieniać swoją wartość w trakcie wykonywania programu.

Blok wejścia/wyjścia

Blok wejścia/wyjścia (ang. *input/output block*) reprezentuje na schemacie blokowym algorytmu operacje pobierania — odczytu danych z urządzenia wejściowego (np. z klawiatury), oraz prezentacji — wyświetlenia danych na urządzeniu wyjściowym (np. na ekranie monitora).

Na rysunku 1.6 pokazano przykład bloku wejściowego na schemacie blokowym algorytmu programu (lub podprogramu) ilustrujący pobranie z urządzenia wejściowego wartości zmiennej o nazwie *a*.

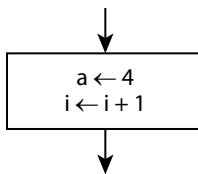
Blok wejścia/wyjścia może być również wykorzystany w celu wyświetlania komunikatu informacyjnego dla użytkownika, np. dotyczącego błędu.



Rysunek 1.6.
Przykład bloku wejściowego

Blok operacji

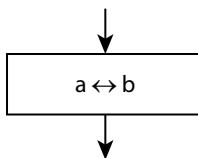
Blok operacji (ang. *process block, action block*) oznacza na schemacie blokowym algorytmu operacje przetwarzania danych (np. instrukcje przypisania), które nie należą do grupy operacji wejścia/wyjścia. Do oznaczenia przypisania używa się strzałki blokowej skierowanej od wyrażenia znajdującego się z prawej strony do zmiennej z lewej strony. Zilustrowano to na rysunku 1.7.



Rysunek 1.7. Przykład bloku przetwarzania reprezentującego instrukcję przypisania

W przykładzie na rysunku 1.7 do zmiennej o nazwie a jest wpisywana wartość **4**, a do zmiennej i przypisywana jest jej wartość poprzednia zwiększona o **1**.

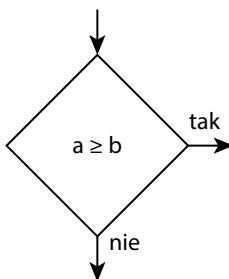
Operacje zamiany wartości zmiennych są w blokach przetwarzania oznaczane za pomocą strzałki dwukierunkowej (z grotami po obu stronach). Zilustrowano to na rysunku 1.8 na przykładzie zamiany wartości zmiennych a oraz b .



Rysunek 1.8. Przykład bloku przetwarzania z zamianą wartości zmiennych

Blok decyzyjny

Blok decyzyjny (ang. *decision block*) (rysunek 1.9) reprezentuje na schemacie blokowym warunek. Blok ten ma jedno wejście oraz dwa wyjścia. Jedno z tych wyjść jest oznaczane słowem TAK i odpowiada spełnieniu warunku (**TRUE**, **1**), a drugie jest opisywane słowem NIE i odpowiada niespełnieniu warunku (**FALSE**, **0**).



Rysunek 1.9. Przykład bloku decyzyjnego

W przykładzie na rysunku 1.9 sprawdzany jest warunek, czy wartość zmiennej o nazwie a jest większa lub równa wartości zmiennej b .

W blokach decyzyjnych nie należy używać operatorów relacyjnych znanych z kodów źródłowych programów, takich jak: $==$ (równy), $!=$ (różny), $<=$ (mniejszy lub równy), $>=$ (większy lub równy). Na schematach blokowych warunki należy formułować przy użyciu standardowego, znormalizowanego zapisu matematycznego, a więc przy wykorzystaniu operatorów: $=$, \neq , $<$, $>$, \leq , \geq . To samo dotyczy operacji sumy i iloczynu logicznego: \vee , \wedge .

W praktyce bloki decyzyjne występują na schematach blokowych zawierających opis instrukcji warunkowych (np. `if`, `if-else`, `switch`) oraz pętli programowych (np. `while`, `do-while`).

UWAGA

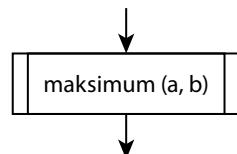
Zarówno instrukcje warunkowe, jak i pętle programowe zostały omówione w rozdziale 3. podręcznika.

Blok podprogramu

Blok podprogramu (ang. *subroutine block*, *predefined process*) (rysunek 1.10) oznacza na schemacie blokowym algorytmu wywołanie określonego podprogramu (np. funkcji). Przy tym podprogram traktuje się wówczas jako „czarną skrzynkę” — bez zajmowania się szczegółami dotyczącymi jego algorytmu. W bloku podprogramu należy podać nazwę wywoływanego podprogramu wraz z listą argumentów jego wywołania.

W bloku funkcjonalnym na rysunku 1.10 wywoływany jest podprogram o nazwie *maksimum*, którego argumentami są zmienne *a* i *b*.

W razie potrzeby algorytm podprogramu wywoływanego jako blok opisuje się (definiuje) na osobnym schemacie blokowym.

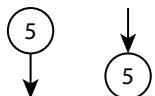


Rysunek 1.10.
Przykład bloku wywołania podprogramu

Łączniki

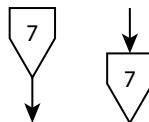
Dany schemat blokowy może być mało przejrzysty i czytelny, jeśli jest przedstawiony na pojedynczej stronie, np. z powodu dużej złożoności i/lub występowania przecinających się linii. Aby tego uniknąć, można przenieść jego wybrane fragmenty w inne miejsce na tej samej stronie lub na inną, dodatkową stronę.

W pierwszym ze wspomnianych przypadków pomocne będą **łączniki wewnętrzne**, inaczej **wewnętrzstronicowe** (ang. *on-page connectors*) (rysunek 1.11), a w drugim **łączniki zewnętrzne**, inaczej **międzystronicowe** (ang. *off-page connectors*) (rysunek 1.12). Łączniki wewnętrzne reprezentują łączenie w obrębie jednej strony, a łączniki zewnętrzne — w ramach dwóch stron.



Rysunek 1.11.

Przykład łącznika wewnętrznego (od lewej: źródłowy i docelowy)



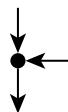
Rysunek 1.12.

Przykład łącznika zewnętrzne (od lewej: źródłowy i docelowy)

Każdy łącznik składa się z dwóch bloków — źródłowego i docelowego. Łączniki numeruje się liczbami całkowitymi. Dla każdego łącznika źródłowego istnieje jeden łącznik docelowy o tym samym numerze porządkowym. Strzałka jest skierowana do łącznika źródłowego oraz od łącznika docelowego.

Punkty koncentracji

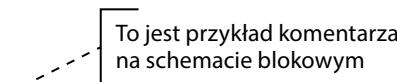
Bloki funkcjonalne na schematach blokowych są połączone za pomocą linii, które są zakończone strzałkami. Strzałki te określają kolejność wykonywania działań zdefiniowanych za pomocą poszczególnych bloków. Jeżeli co najmniej dwa groty strzałek koncentrują się w jednym punkcie na schemacie, wówczas należy ten punkt oznaczyć jako punkt koncentracji. Zilustrowano to na rysunku 1.13.



Rysunek 1.13. Przykład punktu koncentracji

Komentarze

Blok komentarza (ang. *comment block*, *annotation block*) (rysunek 1.14) jest używany do skomentowania (wyjaśnienia) znaczenia instrukcji zawartych w wybranym bloku (blokach). Tekst komentarza należy wpisać pomiędzy kreskami pionowymi. Kreska ukośna powinna dotykać komentowanego bloku.



Rysunek 1.14. Przykład bloku komentarza

Zasady tworzenia schematów blokowych

- Schemat blokowy powinien być zrozumiały dla programistów wykorzystujących różne języki programowania. Dlatego też nie należy stosować składni ani operatorów charakterystycznych dla żadnego języka programowania. Preferowane jest użycie standardowych operatorów matematycznych.
- Jeśli schemat blokowy algorytmu jest przedstawiony na wielu stronach (arkuszach), wówczas poszczególne strony należy numerować zgodnie z przyjętą konwencją. W takim przypadku zalecane jest sporządzenie spisu wszystkich stron wchodzących w skład schematu blokowego wraz z niezbędnym opisem, co zawiera każda ze stron.
- Schemat blokowy powinien być przejrzysty i czytelny. Jeśli problem do rozwiązania jest złożony, należy go podzielić na części — elementy składowe, i dla każdego z nich utworzyć osobny schemat. Schematy blokowe cząstkowe można ze sobą łączyć za po-

mocą łączników na schemacie blokowym ogólnym, zamieszczonym na osobnej stronie (arkuszu). Innym rozwiązaniem jest wykorzystanie na schemacie blokowym ogólnym bloków wywołania podprogramów. Schematy blokowe algorytmów poszczególnych podprogramów można umieszczać na osobnych stronach (arkuszach).

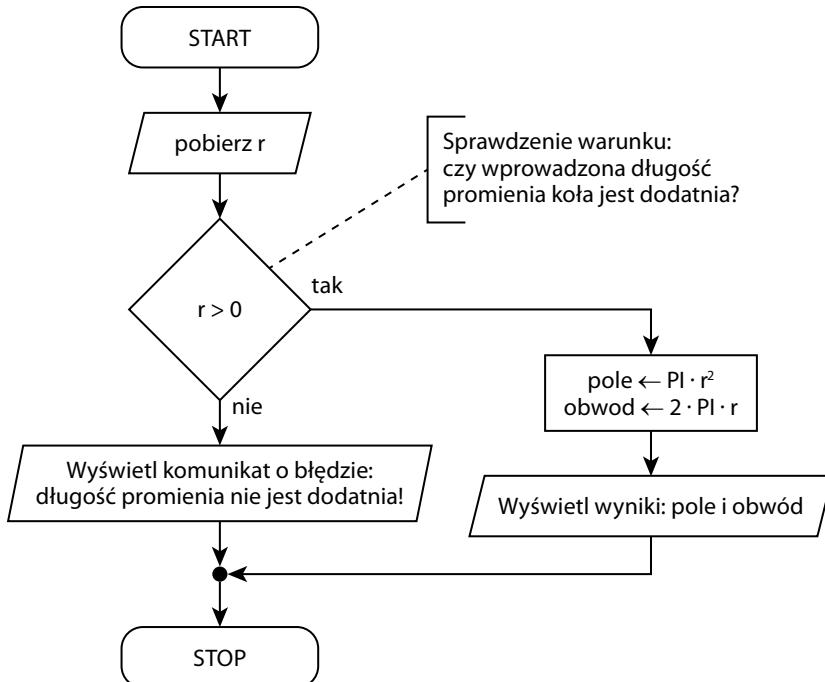
- Nie należy zbytnio zagęszczać bloków funkcjonalnych na schemacie blokowym w celu umożliwienia jego ewentualnej modyfikacji i/lub poprawy, np. dorysowania nowych bloków funkcjonalnych.
- Groty strzałek na schematach blokowych powinny być skierowane albo w dół, albo na prawo oraz pod kątem prostym.
- Zalecane jest, aby na schematach blokowych unikać stosowania ścieżek (linii), które się przecinają. Wynika to z faktu, że przecinające się linie znacznie zmniejszają przejrzystość i czytelność schematu. W razie konieczności należy wykorzystać łączniki wewnętrzstwonne do podzielenia schematu na mniejsze części.
- W zależności od wymagań zewnętrznych (zamawiającego) i/lub wewnętrznych (programistów) każdy blok funkcjonalny na schemacie blokowym powinien reprezentować przyjętą (stałą) liczbę odpowiadających mu operacji. Na schemacie blokowym nie powinno być tak, że np. dany blok odpowiada pojedynczej operacji prostej, a inny blok reprezentuje kilka operacji złożonych.
- Opis algorytmu programu w postaci schematu blokowego można łączyć z innymi formami zapisu schematów blokowych, np. opisem słownym, pseudokodem.
- Schemat blokowy powinien być właściwie i wyczerpująco skomentowany. Komentarze można wstawiać bezpośrednio na schemacie, za pomocą bloków komentarza odpowiadających poszczególnym lub wybranym blokom funkcjonalnym, albo na dodatkowych arkuszach, na których zamieszcza się szczegółowy słowny opis danego bloku/bloków.

Przykłady schematów blokowych

Przykład 1.1

Na rysunku 1.15 przedstawiono schemat blokowy programu obliczającego pole i obwód koła. Długość promienia jest wprowadzana z klawiatury, a wyniki (wartości pola i obwodu koła) są prezentowane na ekranie monitora. Obliczenia są wykonywane tylko wtedy, gdy wprowadzona długość promienia koła jest dodatnia. W przeciwnym razie — gdy długość promienia nie jest dodatnia — na ekranie monitora powinien zostać wyświetlony komunikat o błędzie.

Omawiany program nie zawiera żadnych podprogramów.



Rysunek 1.15. Schemat blokowy algorytmu programu z przykładu 1.1

W schemacie blokowym przedstawionym na rysunku 1.15 zilustrowano praktyczne wykorzystanie następujących bloków: początkowego i końcowego, wejścia/wyjścia, operacji, decyzyjnego i komentarza. Blok wejściowy wykorzystuje się do pobrania długości promienia koła *r* z klawiatury. Zadaniem bloku decyzyjnego jest ustalenie, czy *r* jest dodatnie. Jeśli tak, wówczas wykonywane są niezbędne obliczenia, a następnie wyniki (wartości zmiennych *pole* i *obwód*) są wyświetlane na ekranie monitora. W przeciwnym razie na ekranie wyświetlany jest stosowny komunikat o błędzie.

Przykład 1.2

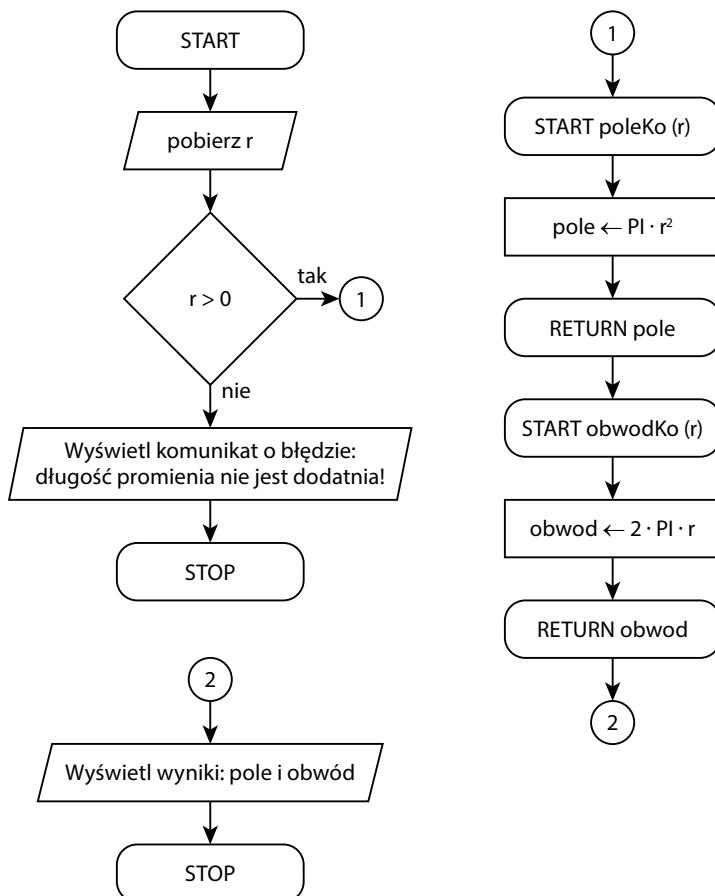
W przykładzie zaprezentowano dwa warianty schematu blokowego programu, którego treść została przedstawiona w przykładzie 1.1.

Program tutaj różni się od programu z przykładu 1.1 wykorzystaniem dwóch podprogramów o nazwach: *poleKo* i *obwodKo*. Zadaniem podprogramu *poleKo* jest obliczenie pola koła, a *obwodKo* — jego obwodu. Do każdego z tych podprogramów dostarczane są pojedyncze dane wejściowe — długość promienia koła reprezentowana przez zmienną *r*. Wyniki działania opisywanych podprogramów — wartości zmiennych *pole* i *obwód* — są przekazywane do programu głównego za pośrednictwem nazw tych podprogramów.

W wariancie pierwszym (rysunek 1.16) definicje podprogramów są zawarte bezpośrednio na schemacie blokowym ogólnym, dotyczącym całego programu. Bloki początkowe wspo-

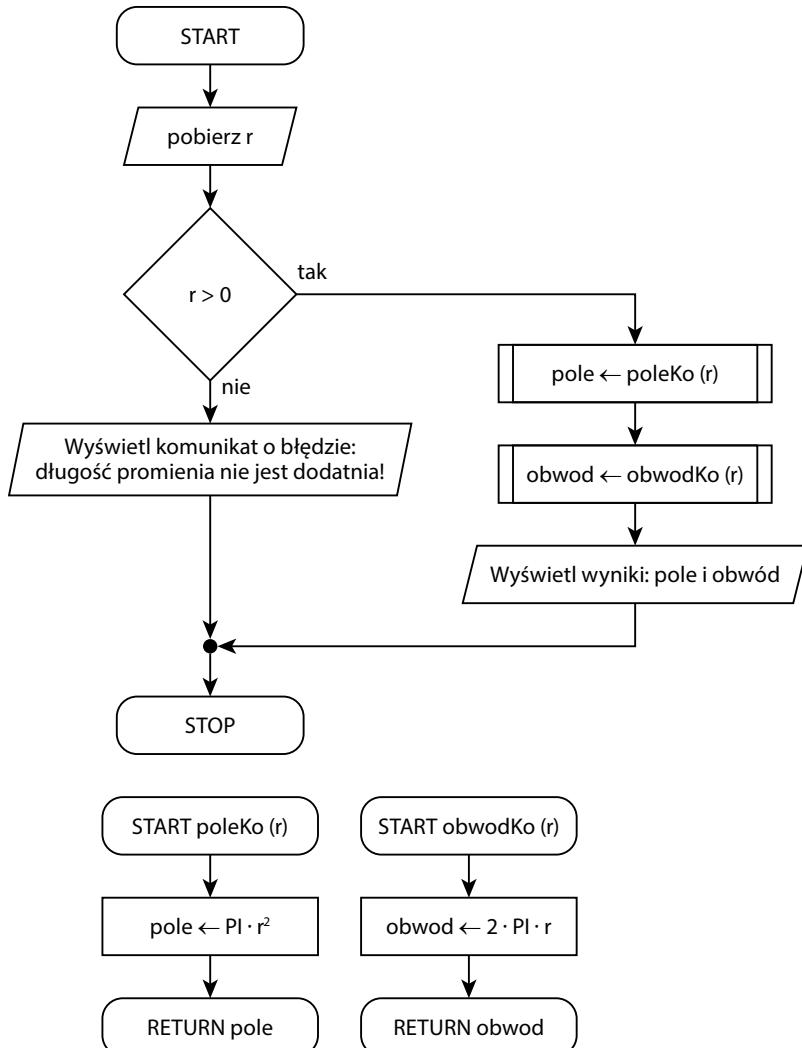
mianych definicji zawierają nazwy podprogramów (*poleKo* i *obwodKo*) oraz ich argument *r*, który jest równoważny parametrowi wejściowemu. Bloki końcowe zaś są opisane za pomocą słowa RETURN oraz zmiennych *pole* i *obwod*, w których przechowywane są wyniki działania podprogramów.

Ponadto na omawianym schemacie blokowym zaprezentowano praktyczne wykorzystanie łączników wewnętrznych (wewnętrznych stronicowych).



Rysunek 1.16. Wariant pierwszy schematu blokowego algorytmu programu z przykładu 1.2

W wariantie drugim schematu blokowego pokazanym na rysunku 1.17 definicje podprogramów *poleKo* i *obwodKo* są określone za pomocą odrębnych schematów blokowych — niezależnych od schematu ogólnego. Każdy ze wspomnianych schematów ma jednoznacznie określony początek i koniec.



Rysunek 1.17. Wariant drugi schematu blokowego algorytmu programu z przykładu 1.2

Schemat blokowy algorytmu całego programu zawiera jedynie bloki wywołania podprogramów *poleKo* i *obwodKo*, a nie ich kompletne definicje — jak na rysunku 1.16. We wspomnianych blokach wywołań podprogramów są zawarte ich nazwy (*poleKo*, *obwodKo*) wraz z argumentem wywołania *r*, a także przypisanie wyników ich działania do konkretnych zmiennych, odpowiednio *pole* i *obwod*.

Schemat blokowy przedstawiony na rysunku 1.17 jest bardziej czytelny i przejrzysty od schematu na rysunku 1.16, pomimo że oba dotyczą algorytmu tego samego programu. W celu przekazania programiście możliwe dużej ilości informacji schemat na rysunku 1.17 można byłoby uzupełnić o zestaw komentarzy ogólnych, dotyczących całych algorytmów, jak również szczegółowych, związanych z ważniejszymi blokami funkcjonalnymi.

1.2.2. Pseudokod

Algorytmy programów oraz ich części (elementów) składowych, np. podprogramów, można zapisywać również przy użyciu tzw. **pseudokodu** (ang. *pseudocode*). Podobnie jak schemat blokowy, pseudokod to opis algorytmu niezależny od systemu operacyjnego, języka programowania czy środowiska programistycznego.

Najważniejszą różnicą pomiędzy algorytmem programu określonym za pomocą pseudokodu a tym samym algorytmem zapisanym w konkretnym języku programowania (czyli programem komputerowym) jest to, że pseudokod jest zrozumiały także dla kogoś, kto nie umie programować w żadnym języku. W pseudokodzie reprezentowane są wszystkie elementy składowe algorytmów, np. operacje wejścia/wyjścia, operacje przypisania i struktury sterujące, takie jak konstrukcje warunkowe i pętle programowe.

W ogólności pseudokod to logiczne połączenie elementów konkretnego języka naturalnego (np. języka polskiego) z elementami języka programowania. W przeciwieństwie do języków programowania pseudokod nie ma standardu. Mimo to, pisząc pseudokod, należy się stosować do ogólnie przyjętych konwencji i zaleceń dotyczących tej formy zapisu algorytmów.

Zasady zapisu algorytmów w postaci pseudokodu

- Pseudokod powinien być zrozumiały zarówno dla doświadczonych programistów, jak i początkujących. Dlatego też zaleca się, aby w pseudokodzie wykorzystywać jedynie powszechnie znane konstrukcje językowe (np. struktury warunkowe *if*, *if-else*, pętle programowe *while*, *do-while*), występujące w większości popularnych języków programowania.
- Algorytm zapisany w pseudokodzie powinien być czytelny. Dlatego w celu uzyskania jak największej przejrzystości można stosować „wcięcia” — podobnie jak wcina się w kodzie źródłowego programu instrukcje składowe konstrukcji warunkowych (np. *if*) oraz pętli programowych (np. *while*). Takie podejście pomaga programistom zrozumieć wykorzystane mechanizmy sterujące, np. procesy decyzyjne w programie.
- Łatwość zrozumienia i analizy algorytmu zapisanego w pseudokodzie można w niektórych przypadkach zwiększyć przez zastosowanie numeracji linii.
- W całym pseudokodzie należy stosować te same konwencje nazewnicze dotyczące nazw stałych, zmiennych, podprogramów, parametrów/argumentów podprogramów itp. Przykładowo nazwy stałych w programie można zapisywać dużymi literami (np. *PI*), zmiennych małymi (np. *r*), a do nazw podprogramów można wykorzystać styl camelCase (np. *obliczPole*).
- Pseudokod powinien być kompletny. Innymi słowy, żadna wykorzystana konstrukcja nie powinna być dla programisty zagadką przez zbyt dużą ogólność zapisu. Wszystkie operacje powinny być opisane na tyle szczegółowo, aby programista mógł je zinterpretować w sposób jednoznaczny.

- Jeśli to potrzebne, poszczególne lub wybrane operacje w algorytmie powinny być skomentowane. W tym celu można wykorzystać styl pisania komentarzy obowiązujący w popularnych językach programowania, np. C++ lub Java, tj. używać symbolu `//`.
- Zaleca się, aby algorytmy podprogramów rozpoczynać od podania nazwy podprogramu oraz jego parametrów, np. `poleKo (r)`. Jeżeli podprogram zwraca na zewnątrz (do swojego otoczenia) jakąś wartość, wówczas można go zakończyć słowem `return` wraz z okresem wyrażenia, którego wartość jest zwracana przez podprogram (np. `return pole`).
- Dobrym nawykiem jest, aby na początku każdego algorytmu opisać problem (zadanie), który należy za jego pomocą rozwiązać.

Przykłady pseudokodu

Przykład 1.3

```

// Obliczenie pola i obwodu koła dla długości promienia wprowadzonego z klawiatury.
// Wyniki (pole i obwód koła) są prezentowane na ekranie monitora.
Wprowadź z klawiatury długość promienia koła r
if r > 0
    then
        // Obliczenie pola koła:
        pole ← PI · r · r
        // Obliczenie obwodu koła:
        obwod ← 2 · PI · r
        // Wyświetlenie wyników na ekranie monitora:
        wyświetl pole
        wyświetl obwod
    else
        wyświetl komunikat o błędzie: długość promienia nie jest dodatnia!

```

W powyższym pseudokodzie opisano algorytm programu, którego treść została przedstawiona w przykładzie 1.1. Cel i zadania programu są opisane w komentarzu na początku pseudokodu. Konstrukcja warunkowa `if-then-else` została wykorzystana w celu sprawdzenia, czy długość promienia koła `r` wprowadzona z klawiatury jest dodatnia. Jeśli tak, wówczas obliczane są pole i obwód koła. Wyniki zostają zapamiętane w zmiennych `pole` i `obwod`, a następnie wyświetcone na ekranie monitora. W przeciwnym razie na ekranie monitora wyświetlany jest komunikat o błędzie.

Zaprezentowany w przykładzie pseudokod jest w pełni zgodny logicznie (funkcjonalnie) ze schematem blokowym przedstawionym na rysunku 1.15.

Przykład 1.4

```

// Obliczenie pola i obwodu koła dla długości promienia wprowadzonego z klawiatury.
// Wyniki (pole i obwód koła) są prezentowane na ekranie monitora.
// W programie wykorzystano dwa podprogramy: poleKo i obwodKo.

Wprowadź z klawiatury długość promienia koła r
if r > 0
then
    // Wywołanie podprogramu poleKo z argumentem r.
    // Wynik działania podprogramu jest podstawiany do zmiennej o nazwie pole:
    pole ← poleKo (r)
    // Wywołanie podprogramu obwodKo z argumentem r.
    // Wynik jest przypisywany do zmiennej obwod:
    obwod ← obwodKo (r)
    // Wyświetlenie wyników (wartości zmiennych pole i obwod) na ekranie monitora:
    wyświetl pole
    wyświetl obwod
else
    wyświetl komunikat o błędzie: długość promienia nie jest dodatnia!

```

Pseudokod opisujący algorytm podprogramu *poleKo*, który jest wywoływany w programie głównym:

```

// POCZĄTEK DEFINICJI PODPROGRAMU poleKo z parametrem r
poleKo (r)
    pole ← PI · r · r
    return pole
// KONIEC DEFINICJI PODPROGRAMU poleKo

```

Pseudokod opisujący algorytm podprogramu o nazwie *obwodKo*, który jest wywoływany w programie głównym:

```

// POCZĄTEK DEFINICJI PODPROGRAMU obwodKo z parametrem r
obwodKo (r)
    obwod ← 2 · PI · r
    return obwod
// KONIEC DEFINICJI PODPROGRAMU obwodKo

```

Zaprezentowany algorytm opisuje program, którego treść przedstawiono w przykładzie 1.2. Podobnie jak tam, w celu obliczenia pola i obwodu koła zostały zdefiniowane dwa podprogramy: *poleKo* i *obwodKo*. Każdy z tych podprogramów zwraca obliczoną wartość na zewnątrz, do swojego otoczenia, „za pośrednictwem nazwy”.

Algorytm przedstawiony tutaj za pomocą pseudokodu odpowiada funkcjonalnie schematowi blokowemu pokazanemu na rysunku 1.17.



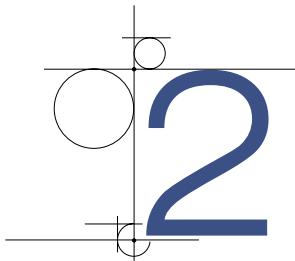
1.3. Pytania i zadania kontrolne

1.3.1. Pytania

1. Omów poszczególne etapy tworzenia programu komputerowego.
2. Podaj definicję algorytmu.
3. Podaj definicję programu komputerowego.
4. Co to jest kompilator i jakie są jego zadania?
5. Wymień i opisz najważniejsze bloki funkcjonalne wchodzące w skład schematu blokowego.
6. Omów zasady zapisu algorytmów w postaci pseudokodu.

1.3.2. Zadania

1. Zaprojektuj i narysuj schemat blokowy algorytmu programu pozwalającego na obliczenie pola i obwodu prostokąta. Dane wejściowe do programu mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora.
2. Treść jak w zadaniu 1. Do obliczenia pola i obwodu prostokąta wykorzystaj podprogramy.
3. Zapisz w formie pseudokodu algorytm programu pozwalającego na obliczenie objętości, pola powierzchni bocznej oraz łącznej długości wszystkich krawędzi prostopadłościanu. Dane wejściowe do programu mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora.
4. Treść jak w zadaniu 3. Do obliczenia objętości, pola powierzchni i długości krawędzi prostopadłościanu wykorzystaj podprogramy.
5. Zapisz w formie listy kroków algorytm programu pozwalającego na obliczenie pola i obwodu koła. Dane wejściowe do programu mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora.
6. Treść jak w zadaniu 5. Do obliczenia pola i obwodu koła wykorzystaj podprogramy.



Środowiska programistyczne

Efektywne pisanie kodów źródłowych programów komputerowych wymaga stosowania odpowiednich metod, stylów i technik programowania. Wymienione elementy można zaliczyć do grupy zagadnień określanych wspólnie jako metodologia programowania. To samo dotyczy wyboru języka programowania optymalnego do rozwiązania postawionego zadania.

Tworzenie programu komputerowego jest procesem złożonym, który składa się z kilku faz (etapów). Można wymienić np. etapy projektowania programu (określanie problemu do rozwiązania, planowanie rozwiązania), edycji kodu źródłowego, debugowania, kompilowania, uruchamiania, testowania czy dokumentowania programu. Zostało to szczegółowo omówione w poprzednim rozdziale, w podrozdziale 1.1.

Wiele z wymienionych etapów tworzenia programu można zrealizować przy użyciu **zintegrowanego środowiska programistycznego** (ang. *integrated development environment*) — **IDE**. IDE jest środowiskiem kodowania, które zawiera zarówno edytor, jak i zestaw wbudowanych narzędzi pomocniczych, specyficznych dla danego języka lub języków programowania. Do wspomnianych narzędzi można zaliczyć np. kompilator (ang. *compiler*), debugger (ang. *debugger*), moduł umożliwiający podświetlanie składni języka, refaktoryzację kodu itd.

Na rynku jest dostępnych co najmniej kilka IDE umożliwiających tworzenie aplikacji w języku C++, w tym programów obiektowych konsolowych. Są to m.in.:

- Code::Blocks,
- CodeLite,
- Eclipse,
- NetBeans,
- Visual Studio,
- CLion.



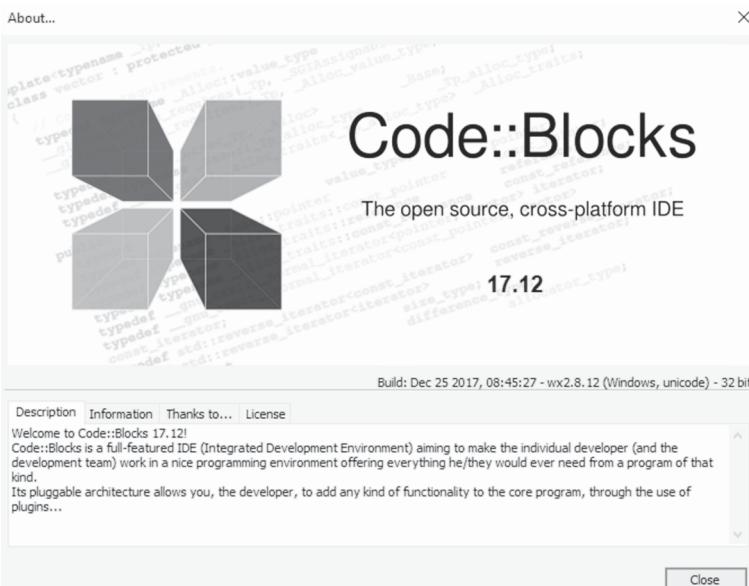
2.1. Code::Blocks

Code::Blocks to IDE **wieloplatformowe** (ang. *cross-platform IDE*), które umożliwia tworzenie programów w językach C i C++ (oraz Fortran) w różnych systemach operacyjnych, tj. Windows, Linux i macOS. Code::Blocks należy do grupy **oprogramowania otwartego** (ang. *open source software*). Jest zatem w pełni darmowe.

Łącznie do pobrania najnowszej wersji Code::Blocks (tj. Code::Blocks 20.03) można znaleźć na oficjalnej stronie programu, <http://www.codeblocks.org>. Do wyboru są różne wersje, np. dla środowiska Windows można pobrać zarówno 64-bitową, jak i 32-bitową.

Ponadto istnieje możliwość pobrania instalatora Code::Blocks wraz z darmową kolekcją kompilatorów GCC (*GNU Compiler Collection*), zawierającą m.in. kompilator C++ (g++), oraz również bezpłatnym debuggerem GDB (*GNU Projekt Debugger*), wchodząącymi w skład **środowiska wykonawczego** (ang. *runtime environment*) MinGW (*Minimalist GNU for Windows*). Pobranie i instalacja Code::Blocks wraz z MinGW jest rozwiązaniem bardzo wygodnym zwłaszcza dla początkujących programistów, mających niewielkie doświadczenie w instalacji kompilatorów i debuggerów C++. Oprócz tego dostępny jest instalator Code::Blocks zawierający dodatkowo zestaw wszystkich dostępnych wtyczek — **pluginów** (ang. *plugins*), dzięki którym można rozszerzać i/lub modyfikować funkcjonalność środowiska.

Starsze wersje Code::Blocks można pobrać ze strony <https://sourceforge.net>. Okienko *About...* środowiska Code::Blocks przedstawiono na rysunku 2.1.



Rysunek 2.1. Okienko *About...* środowiska Code::Blocks

IDE Code::Blocks może współpracować z wieloma kompilatorami języka C++, np. kompilatorem g++ (zawartym m.in. w kolekcji kompilatorów GNU GCC), jak również Microsoft Visual C++ oraz Borland C++, co jest bardzo dużą zaletą tego środowiska. Dla szkół oraz innych placówek oświatowych szczególnie godny polecenia jest kompilator C++ zawarty w GCC, ponieważ jest w pełni darmowy i obsługuje wiele dialektów języka C++ odpowiadających opublikowanym standardom ISO, a w szczególności:

- C++98 (standard ISO/IEC 14882:1998),
- C++03 (ISO/IEC 14882:2003),
- C++11 (ISO/IEC 14882:2011),
- C++14 (ISO/IEC 14882:2014),
- C++17 (ISO/IEC 14882:2017)

oraz, eksperymentalnie, ostatniemu standardowi: C++20 (ISO/IEC 14882:2020).

Inną korzyścią z wyboru IDE Code::Blocks IDE jest rozbudowany **graficzny interfejs użytkownika** (ang. *graphical user interface*) — w skrócie **GUI**. Inne zalety tego IDE to edytor z wbudowanym systemem podświetlania składni języka, narzędzia ułatwiające tworzenie złożonych projektów i bibliotek, jak również dostępność debugera i wygoda w jego stosowaniu.

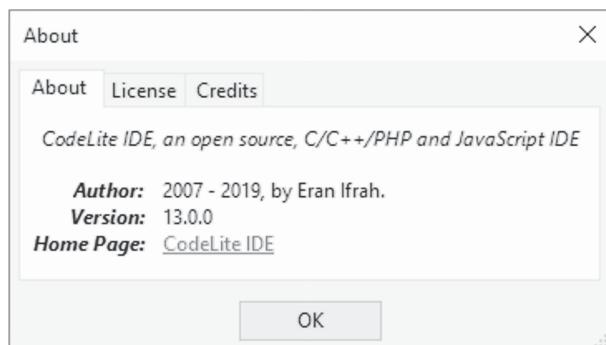
Wbudowany edytor kodu źródłowego można łatwo konfigurować, np. dostosować jego wygląd i czcionki do własnych potrzeb i upodobań. To samo dotyczy systemu podświetlenia składni języka, opcji kompilatora, debugera itd.

Środowisko Code::Blocks nadal jest rozwijane. Wielu programistów traktuje je jako naturalnego następcę bardzo popularnego (jeszcze kilka lat temu) i wygodnego środowiska Dev-C++, które obecnie nie jest już wspierane.

2.2. CodeLite

CodeLite to również IDE wieloplatformowe należące do oprogramowania otwartego. Jest w pełni darmowe. CodeLite wspomaga tworzenie programów komputerowych w języku C++, a także innych, np. JavaScript czy PHP. Tym samym może być atrakcyjną propozycją dla szkół kształcących w zawodzie technika programisty. Najnowszą wersję tego IDE jest CodeLite 14.0, opublikowana w 2020 roku.

CodeLite można pobrać z oficjalnej strony tego programu: <https://downloads.codelite.org>. Starsze wersje tego IDE są dostępne na stronie archiwum: <https://downloads.codelite.org/ReleaseArchive.php>. Okienko *About* środowiska CodeLite jest pokazane na rysunku 2.2.



Rysunek 2.2. Okienko About środowiska CodeLite

CodeLite dobrze współpracuje z darmowym kompilatorem C++ zawartym w pakiecie MinGW, który można pobrać ze strony <http://www.mingw.org>, i debuggerem GDB, dostępnym na stronie <https://www.gnu.org/software/gdb/download>.

UWAGA

Domyślna instalacja środowiska programistycznego CodeLite nie zawiera kompilatora (kompilatorów) języka C++. Kompilator należy zainstalować oddzielnie, najlepiej przed instalacją CodeLite. To samo dotyczy debugera.

Środowisko CodeLite zawiera rozbudowany, łatwo konfigurowalny edytor kodu źródłowego. Ponadto zapewnia możliwość uruchamiania i testowania tworzonego programu komputerowego przy użyciu interaktywnego debugera (opartego na GDB). Konfiguracja środowiska CodeLite oraz jego elementów składowych jest realizowana przy wykorzystaniu odpowiednich opcji menu.

CodeLite oferuje szereg przydatnych funkcjonalności, np.:

- podświetlanie składni języka,
- automatyczne uzupełnianie kodu,
- refaktoryzację kodu oraz możliwość generowania komentarzy,
- zarządzanie projektami i plikami,
- mechanizm wyszukiwania i zamiany w plikach.

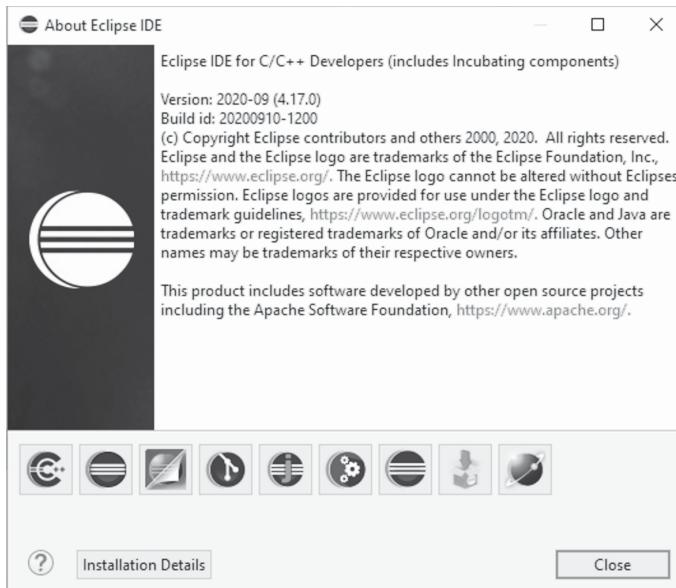
Funkcjonalność CodeLite można rozszerzać za pomocą systemu wtyczek.

Podobnie jak Code::Blocks, CodeLite jest polecane początkującym, niedoświadczonym programistom języka C++, szczególnie uczniom.

2.3. Eclipse

Projekt **Eclipse CDT** (ang. *C/C++ Development Tooling*) zapewnia zintegrowane środowisko programistyczne przeznaczone do tworzenia programów w języku C++, oparte na ogólnej, uniwersalnej platformie Eclipse. Eclipse CDT to IDE wieloplatformowe. Umożliwia tworzenie programów w C++ w systemie Windows, Linux i macOS. Jest to bardzo rozbudowane środowisko, które wciąż jest modyfikowane i rozwijane.

Najnowszą wersją Eclipse CDT jest 10.1.0 z grudnia 2020 roku. Można ją pobrać z oficjalnej strony Eclipse: <https://www.eclipse.org>. Łącza do starszych wersji można znaleźć na stronie <https://www.eclipse.org/cdt/downloads.php>. Okienko *About Eclipse IDE* przedstawiono na rysunku 2.3.



Rysunek 2.3. Okienko About Eclipse IDE

2.4. NetBeans

Zintegrowane środowisko programistyczne NetBeans jest przeznaczone do tworzenia aplikacji internetowych, desktopowych i mobilnych w języku Java. Jednakże wspomaga również programowanie aplikacji w języku C/C++, włącznie z obsługą popularnego framework'a o nazwie Qt (<https://www.qt.io>). Oprócz tego za pomocą NetBeans można tworzyć aplikacje internetowe frontendowe (ang. *front-end applications*) — w HTML5 wraz z CSS3 oraz JavaScript, jak też backendowe (ang. *back-end applications*) — np. w PHP.

Ostatnia wspierana wersja omawianego środowiska to NetBeans 8.2. Można ją pobrać z oficjalnej strony programu: <https://netbeans.org/downloads/old/8.2>. Okienko *About* środowiska NetBeans przedstawiono na rysunku 2.4.



Rysunek 2.4. Okienko About środowiska NetBeans

Następcą IDE NetBeans jest środowisko o nazwie Apache NetBeans. Ostatnia wersja tego IDE to Apache NetBeans 12.2 z grudnia 2020 roku. Można ją pobrać ze strony pod adresem: <https://netbeans.apache.org/download>.

NetBeans jest środowiskiem wieloplatformowym (Windows, Linux, macOS). Należy do oprogramowania otwartego. Jest całkowicie bezpłatne.

Do najważniejszych zalet środowiska Apache NetBeans, z punktu widzenia programisty aplikacji obiektowych w języku C++, należą:

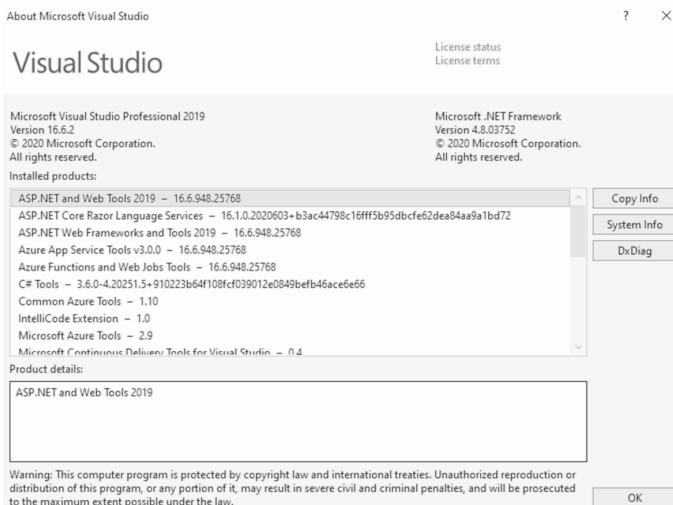
- dobra współpraca z kompilatorem g++ zawartym w pakiecie MinGW oraz możliwość dostosowania opcji kompilatora do własnych potrzeb (np. ustalenia standardu języka na C++11),
- możliwość integracji z debuggerem GNU GDB,
- bogaty interfejs użytkownika (GUI),
- łatwość konfigurowania środowiska,
- rozbudowany edytor kodu z podświetlaniem składni C++, autouzupełnianiem kodu, automatycznym formatowaniem i wcięciami, efektywną nawigacją pomiędzy składnikami programu itp.,
- refaktoryzacja plików,
- możliwość zarządzania złożonymi projektami,
- możliwość tworzenia statycznych i dynamicznych bibliotek,

- tworzenie i wykonywanie testów bezpośrednio z poziomu IDE,
- integralna współpraca z zasobami frameworka Qt.

IDE NetBeans i Apache NetBeans mogą być z powodzeniem wykorzystywane w szkołach kształcących w zawodzie technika programisty jako narzędzie wspomagające zarówno programowanie aplikacji w HTML/CSS, JavaScript i PHP (kwalifikacja INF.03), jak i (m.in.) programowanie obiektowe w języku C++ (kwalifikacja INF.04).

2.5. Visual Studio

Visual Studio jest w pełni profesjonalnym środowiskiem programistycznym firmy Microsoft. Umożliwia ono tworzenie zaawansowanych biznesowych aplikacji webowych, desktopowych i mobilnych w różnych językach programowania, np. C#, C++, Python, JavaScript. Okienko [About Microsoft Visual Studio](#) przedstawiono na rysunku 2.5.



Rysunek 2.5. Okienko About Microsoft Visual Studio

Standardowo Visual Studio nie wspomaga programowania w języku PHP. Jest to możliwe dopiero po zainstalowaniu dodatkowego, komercyjnego pakietu [PHP Tools for Visual Studio](#) firmy DEVSENSE. Jednakże możliwe jest pozyskanie przez szkołę licencji edukacyjnej na ten pakiet, która jest nieodpłatna i obowiązuje przez rok. Po jego upływie można ją (również bezpłatnie) odnowić. Visual Studio nie wspomaga również programowania w języku Java.

Microsoft preferuje język programowania C#, ewentualnie Python. Jeśli jednak spojrzeć z perspektywy programowania obiektowego, Visual Studio wspomaga także w pełni tworzenie aplikacji w języku C++, w tym aplikacji konsolowych.

W domyślnych ustawieniach instalacji Visual Studio w wersji 2013 automatycznie instalowane są składniki niezbędne do tworzenia aplikacji w języku C++. Jednakże, począwszy od wersji 2015, aby można było pisać, kompilować i debugować aplikacje C++, należy już podczas

instalacji dołączyć do Visual Studio niezbędne składniki związane z C++ albo doinstalować je po zakończeniu instalacji domyślnej.

Na rynku dostępnych jest kilka odmian środowiska Visual Studio. Visual Studio Professional oraz Visual Studio Enterprise to środowiska komercyjne. Visual Studio Community jest natomiast bezpłatne. Funkcjonalność środowiska Visual Studio Community odpowiada, z niewielkimi wyjątkami, odmianie Professional. Według postanowień licencyjnych bezpłatne wykorzystanie Visual Studio Community w instytucjach (np. w szkołach publicznych) jest możliwe wyłącznie w celach edukacyjnych. Natomiast użytkownicy indywidualni (np. nauczyciele i uczniowie) mogą nieodpłatnie wykorzystywać odmianę Community nawet w celach komercyjnych.

Wymagania sprzętowe Visual Studio, szczególnie najnowszych wersji, 2017 i 2019, są znaczne. Dotyczy to zwłaszcza wielkości pamięci operacyjnej komputera oraz pojemności (i prędkości) dysku twardego. Praca w Visual Studio w wersji 2017 lub 2019 na komputerze wyposażonym w powolny dysk twardy i mniej niż 8 GB RAM jest mało komfortowa. Dlatego też, jeśli szkoła ma do dyspozycji starszy sprzęt komputerowy, być może wygodniej byłoby skorzystać z mniej wymagającego oprogramowania, np. jednego z narzędzi niekomercyjnych.

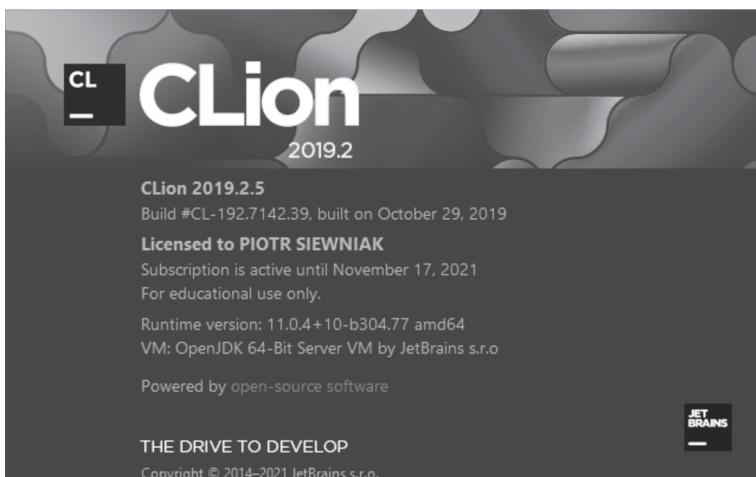
Visual Studio Code to kolejny bezpłatny i wieloplatformowy produkt firmy Microsoft, który może wspomagać tworzenie aplikacji w języku C++ (po zainstalowaniu rozszerzenia *Microsoft C/C++*). Visual Studio Code nie jest wyposażone domyślnie ani w kompilator języka C++, ani wbudowany debugger wspomagający proces tworzenia aplikacji w C++. Dlatego też należy osobno zainstalować wybrany kompilator (np. GNU g++) oraz debugger (np. GDB).

2.6. CLion

CLion jest rozbudowanym środowiskiem programistycznym firmy JetBrains. Jest to IDE wieloplatformowe, ale komercyjne. Jednakże szkoły publiczne mogą uzyskać od JetBrains bezpłatną roczną licencję na wykorzystanie oprogramowania do celów edukacyjnych. Po upływie roku licencję tę można (również bezpłatnie) odnowić na kolejny rok. Okienko z informacjami o środowisku CLion przedstawiono na rysunku 2.6.

CLion, podobnie jak Visual Studio firmy Microsoft, to bardzo potężne narzędzie. Ma rozbudowany, profesjonalny edytor oferujący analizę kodu, system podświetlania składni języka, podpowiedzi, autouzupełnianie kodu i wiele innych udogodnień. Zapewnia też zintegrowany debugger, ułatwiający proces uruchomienia i testów roboczych programu.

Wymagania sprzętowe środowiska CLion, podobnie jak wymagania Visual Studio, są znaczne. Dlatego też pomimo niewątpliwych ogromnych zalet tego IDE szkoły jako instytucje, jak również nauczyciele i uczniowie mogą rozważyć jako alternatywną opcję inne, „lżejsze” IDE, np. niekomercyjne.



Rysunek 2.6. Okienko z informacjami o środowisku CLion

2.7. Pytania i zadania kontrolne

2.7.1. Pytania

1. Co oznacza skrót IDE? Opisz najważniejsze funkcje IDE na przykładzie Code::Blocks, CodeLite lub NetBeans.
2. Wymień nazwy kilku niekomercyjnych, darmowych kompilatorów języka C++.
3. W jakim celu przeprowadza się debugowanie programu?
4. Jakimi cechami powinien się charakteryzować profesjonalny edytor kodu?
5. Co oznacza, że dany program należy do oprogramowania otwartego (ang. *open source*)?
6. Które IDE wspomagają nieodpłatnie programowanie aplikacji w C++, a także dodatkowo w językach HTML wraz z CSS oraz JavaScript i PHP?

2.7.2. Zadania

1. Zapoznaj się z kompilatorami języka C++ polecanymi przez B. Stroustrupą, twórcę C++, na jego stronie domowej: <https://www.stroustrup.com/compilers.html>.
2. Pobierz i zainstaluj pakiety MinGW i Cygwin w ich najnowszych dostępnych wersjach.
3. Pobierz i zainstaluj środowisko programistyczne Code::Blocks.
4. Zainstaluj środowisko CodeLite. Skonfiguruj wybrany kompilator i debugger (np. GNU GDB).
5. Zainstaluj Visual Studio Code wraz z rozszerzeniem Microsoft C/C++. Skonfiguruj wybrany kompilator i debugger.
6. Zapoznaj się z postanowieniami licencyjnymi środowiska Visual Studio Community.

3

Podstawy programowania



3.1. Podstawowe elementy języka C++

Program komputerowy w języku C++ składa się m.in. ze zbioru plików tekstowych zawierających kod źródłowy z deklaracjami (ang. *declarations*), definicjami (ang. *definitions*), wyrażeniami (ang. *expressions*), instrukcjami (ang. *statements*) i innymi konstrukcjami tego języka, np. komentarzami (ang. *comments*). Wspomniane pliki tekstowe to pliki źródłowe (ang. *source files*) i pliki nagłówkowe (ang. *header files*). Zostały omówione, podobnie jak deklaracje, definicje, wyrażenia, instrukcje itd., w dalszej części tego podrozdziału.

W kodzie źródłowym programu wykorzystuje się różne słowa. Znaczenie niektórych z nich jest zastrzeżone dla określonych elementów składowych języka. Są to słowa kluczowe (words). Na przykład słowo kluczowe `if` to nazwa instrukcji, słowo kluczowe `double` to nazwa typu danych, a `true` to wartość logicznej prawdy.

Pozostałych słów można używać do nadawania nazw — tzw. **identyfikatorów** (ang. *identifiers*) — elementom składowym programu zdefiniowanym przez programistę: stałym, zmennym, funkcjom, klasom, obiektom itp. Identyfikator w C++ składa się z dowolnego ciągu liter, cyfr oraz kresek podkreślenia (ang. *underscores*) i musi rozpoczynać się od litery. Na przykład poprawnymi identyfikatorami w języku C++ są: `bok1`, `poleProstokata`, `dane_wejsciowe`.



UWAGA

W języku C++ małe i duże litery są rozróżnialne. Przykładowo identyfikatory `zmienna` i `Zmienna` to nazwy dwóch różnych elementów (np. zmiennych) w programie.

Kod źródłowy programu komputerowego w języku C++ musi być zgodny z określonymi regułami (zasadami) syntaktycznymi ustalonymi dla tego języka. Przy czym wspomniane reguły

syntaktyczne dotyczą składni języka, czyli jego „gramatyki” i „ortografii”. Na przykład ważną regułą składniową jest zasada, że każdą instrukcję w języku C++ kończy się średnikiem (;).

UWAGA

Kompletną dokumentację — specyfikację (ang. *specification*) — języka C++ można znaleźć na stronie cppreference.com. Dotyczy ona wszystkich najważniejszych standardów języka — od C++98 (ISO/IEC 14882:1998) po C++20 (ISO/IEC 14882:2020).

W kodzie źródłowym oprócz słów kluczowych, reprezentujących zastrzeżone elementy języka, oraz identyfikatorów skojarzonych z elementami programu zdefiniowanymi przez programistę mogą się znaleźć tzw. **komentarze** (ang. *comments*). Komentarze to informacje dla programisty (programistów) dotyczące różnych aspektów kodu — począwszy od opisu ogólnej koncepcji zastosowanego rozwiązań, a skończonej na szczegółowych informacjach związanych z pojedynczymi instrukcjami.

W języku C++ można stosować komentarze dwojakiego rodzaju:

- **komentarze jednoliniowe** (ang. *single-line comments*),
- **komentarze wieloliniowe** (ang. *multi-line comments*).

Komentarz jednoliniowy obejmuje ciąg znaków od symbolu // do końca linii, np.

..... // to jest komentarz jednoliniowy

Komentarz wieloliniowy zaś rozpoczyna się symbolem /*, a kończy */ i może obejmować wiele linii, np.

.....

```
/* To jest komentarz wieloliniowy,
który może obejmować wiele linii.
*/
```

.....

Komentarze są ignorowane przez kompilator i jeśli zostały użyte zgodnie z regułami języka, nie mają żadnego wpływu na proces komplikacji programu. Nie są też dodawane do pliku wynikowego.

3.1.1. Zmienne, typy danych, stałe, operatory, wyrażenia

Deklarowanie zmiennych

Zmienne (ang. *variables*) są wykorzystywane w programie do przechowywania w pamięci operacyjnej komputera wartości, które mogą się zmieniać w trakcie jego działania. Każda zmienna w programie powinna być zadeklarowana.

Postać ogólna **deklaracji zmiennej** (ang. *variable declaration*) jest następująca:

typ_zmiennej nazwa_zmiennej;

gdzie:

- typ_zmiennej — typ danych (ang. *data type*), do którego należy zmienna, np. `double`,
- nazwa_zmiennej — identyfikator zmiennej, np. `bok1`.

Jeśli trzeba zadeklarować kilka zmiennych należących do tego samego typu, należy użyć składni:

typ_zmiennej nazwa_zmiennej_1, nazwa_zmiennej_2, nazwa_zmiennej_3;

gdzie `nazwa_zmiennej_1`, `nazwa_zmiennej_2`, `nazwa_zmiennej_3` to identyfikatory zmiennych.

Typy danych

Każda zmienna zadeklarowana w programie może przechowywać wartości typu określonego w jej deklaracji. Typy danych w języku C++ można podzielić umownie na:

- typy predefiniowane,
- typy zdefiniowane przez użytkownika.

Predefiniowane typy danych (ang. *predefined data types*) to typy wewnętrzne kompilatora (ang. *compiler internal data types*). Dlatego też nazywa się je często **typami wbudowanymi** (ang. *built-in data types*). Do wbudowanych typów danych języka C++ należą:

- typy całkowite,
- typy zmiennoprzecinkowe,
- typ znakowy,
- typ logiczny,
- typ `void`.

Z wbudowanych typów danych można korzystać w programie bez potrzeby ich definiowania.

Do **typów danych zdefiniowanych przez użytkownika** (ang. *user-defined data types*) zalicza się wszystkie pozostałe typy danych, które albo określa samodzielnie programista, albo zostały zdefiniowane wcześniej przez innego programistę (programistów) i są zawarte np. w konkretnej bibliotece. Aby można było w programie skorzystać ze zdefiniowanego typu danych, należy zapewnić dostęp do jego definicji w sposób bezpośredni lub za pośrednictwem np. wspomnianej wcześniej biblioteki.

Inny podział typów danych wynika z rodzaju „bazy” będącej podstawą konstrukcji określonego typu danych. Według tego kryterium typy danych w języku C++ można podzielić na:

- typy podstawowe (pierwotne),
- typy pochodne.

Podstawowe typy danych (ang. *fundamental data types*) są typami elementarnymi. Zawierają one określone zbiory wartości, których nie można podzielić na elementy składowe. Do podstawowych typów danych należą wszystkie typy wbudowane, które wymieniono wcześniej.

Typy podstawowe są stosowane jako bazy konstrukcyjne lub elementy składowe innych — **pochodnych typów danych** (ang. *derived data types*). Dlatego typy podstawowe nazywa się również **typami pierwotnymi** (ang. *primitive data types*).

Patrząc na omawiane zagadnienie z drugiej strony, pochodne typy danych są oparte na typach podstawowych albo zawierają elementy składowe należące do typów podstawowych. Jednakże to nie jest kompletna definicja pochodnego typu danych. W ogólności należy również uwzględnić przypadek, w którym określony pochodny typ danych jest konstruowany na podstawie innych typów pochodnych.

Biorąc pod uwagę powyższe rozważania, można zauważyć, że pochodne typy danych należą do typów zdefiniowanych przez użytkownika, o których była mowa wcześniej.

W literaturze często można spotkać jeszcze jeden podział typów danych, wynikający ze stopnia ich złożoności. Mianowicie, jeżeli wartości należące do określonego typu danych są niepodzielne, tzn. nie można ich dzielić na elementy funkcjonalne, taki typ nazywamy **typem prostym** (ang. *basic data type*). W przeciwnym razie, tj. jeżeli wartości należące do typu danych można podzielić na składniki funkcjonalne, o takim typie danych mówi się, że jest to **typ złożony** (ang. *compound data type*).

Typy całkowite

Typy całkowite (ang. *integer types*) umożliwiają przechowywanie w zmiennych wartości liczb całkowitych. Typ danych o nazwie `int` pełni funkcję podstawowego (bazowego) typu całkowitego.

Na przykład deklaracja zmiennej o nazwie `zmiennaInt` typu całkowitego `int` ma postać: `int zmiennaInt;`. W procesie komplikacji programu kompilator zarezerwuje w pamięci operacyjnej dla tej zmiennej co najmniej 4 bajty. Zakres (ang. *scope*) dopuszczalnych wartości, jakie może przyjąć zmienna typu `int`, obejmuje przedział liczbowy od -2 147 483 648 do +2 147 483 647.

Typ bazowy `int` może być modyfikowany. Służą do tego modyfikatory (ang. *modifiers*): `short`, `long`, `signed` i `unsigned`. W wyniku użycia nazwy typu bazowego `int` wraz z odpowiednim modyfikatorem lub połączeniem wybranych modyfikatorów można uzyskać inne typy danych całkowitych. Typy te różnią się od siebie zakresem dopuszczalnych wartości oraz ilością miejsca zajmowanego w pamięci operacyjnej. Na przykład użycie modyfikatora `long` określa typ `long int`, a połączenie modyfikatorów `unsigned` i `long` daje w rezultacie typ `unsigned long int`.

UWAGA

Szczegółowe informacje dotyczące podstawowych typów danych w C++ można znaleźć w dokumentacji języka, w artykule *Fundamental types* na stronie pod adresem: <https://en.cppreference.com/w/cpp/language/types>.

Typy zmiennoprzecinkowe

Typy zmiennoprzecinkowe (ang. *floating-point types*) są również nazywane **typami zmiennopozycyjnymi**. Pozwalają one na przechowywanie w zmiennych wartości należących do zbioru liczb rzeczywistych. Do typów zmiennoprzecinkowych zalicza się: float, double oraz long double. W zmiennych typu float można zapamiętać liczby o pojedynczej precyzji (ang. *single precision*), double — podwójnej precyzji (ang. *double precision*), long double — rozszerzonej precyzji (ang. *extended precision*). Stąd zmienna typu float zajmuje w pamięci operacyjnej 4 bajty, zmienna typu double — 8 bajtów, a typu long double — 10 bajtów. Na przykład deklaracja zmiennej o nazwie (identyfikatorze) zmiennaDouble typu double ma postać: double zmiennaDouble;.

Typ znakowy

Typ znakowy (ang. *character type*) jest oznaczany za pomocą słowa kluczowego char. Jest on wykorzystywany w deklaracjach zmiennych, w których pamiętane są liczby całkowite z przedziału od -128 do 127 lub pojedyncze znaki z przedziału od 0 do 255. Wraz z nazwą typu char można używać modyfikatorów signed oraz unsigned, np. unsigned char. Zmienne typu char są przechowywane w pamięci operacyjnej w obszarach (blokach) o pojemności 1 bajta.

Typ logiczny

Nazwa wbudowanego typu logicznego (ang. *boolean type*) to bool. Do typu bool należą dwie predefiniowane wartości: true (prawda) oraz false (fałsz). Zmienna typu bool zajmuje w pamięci 1 bajt.

Typ void

Typ void reprezentuje zbiór pusty, czyli taki, który nie zawiera żadnych wartości.

UWAGA

Nie można zadeklarować zmiennej typu void. Typ void jest używany w deklaracjach wskaźników ogólnych — generycznych (ang. *generic pointer declarations*), oraz w deklaracjach funkcji (ang. *function declarations*), które nie zwracają do swojego otoczenia żadnej wartości „za pośrednictwem swojej nazwy”. Zagadnienia te zostały omówione w dalszej części podręcznika: w rozdziale 4., dotyczącym wskaźników, oraz w rozdziale 8., w którym zaprezentowano funkcje.

Przykład 3.1

```
char znak;
long int bok1, bok2;
double a, b, c;
```

W przedstawionym kodzie zadeklarowano zmienną o nazwie `znak` należącą do predefiniowanego typu podstawowego `char`. W następnej linii zadeklarowano dwie zmienne typu `long int`. Identyfikator pierwszej z nich to `bok1`, a drugiej — `bok2`. W ostatniej linii zadeklarowano zmienne `a`, `b` i `c` należące do typu `double`.

Ćwiczenie 3.1

Opisz szczegółowo znaczenie kodu w kolejnych liniach w przykładowym fragmencie programu:

```
char c;
int liczba1, liczba2;
float x, y, z;
```

Dopisz dwie dodatkowe linie kodu. W pierwszej z nich powinna się znaleźć deklaracja zmiennej o nazwie `promien` należąca do typu całkowitego o standardowej długości (zakresie), ale bez znaku. Z kolei w drugiej linii zadeklaruj zmienną o nazwie `sterowanie` typu całkowitego krótkiego bez znaku.

Aliases nazw predefiniowanych typów danych

W języku C++ programista ma możliwość określenia własnych nazw (identyfikatorów) dla predefiniowanych typów danych podstawowych. Innymi słowy, można tworzyć aliasy (ang. *aliases*) nazw, czyli inne, zastępcze nazwy dla już istniejących typów danych. Można to zrealizować zasadniczo na dwa sposoby:

- przy użyciu **specyfikatora** (ang. *specifier*) `typedef`,
- za pomocą **słowa kluczowego** (ang. *keyword*) `using`.

Ogólna postać użycia specyfikatora `typedef` jest następująca:

```
typedef typ_istniejący alias_nazwy;
```

gdzie `typ_istniejący` jest nazwą istniejącego, predefiniowanego typu danych, a `alias_nazwy` to jego inna, nowa, zastępca nazwa.

Ogólna składnia użycia słowa kluczowego `using`, która pozwala określić alias nazwy istniejącego typu danych, ma postać:

```
using alias_nazwy = typ_istniejący;
```

gdzie `alias_nazwy` i `typ_istniejący` mają takie samo znaczenie jak w przypadku słowa kluczowego `typedef`.

Przykład 3.2

```
typedef unsigned char byte;
using word = unsigned int;

byte zmiennaByte;
word zmiennaWord;
```

W kodzie zdefiniowano aliasy nazw typów danych: `unsigned char` — jako `byte`, oraz `unsigned int` — jako `word`. W pierwszym przypadku wykorzystano słowo kluczowe `typedef`, a w drugim `using`. Następnie zadeklarowano dwie zmienne całkowite: `zmiennaByte` i `zmiennaWord`. Pierwsza z nich należy do typu `byte`, a druga do typu `word`.



UWAGA

Słowo kluczowe `typedef` można wykorzystać również w celu zdefiniowania nowego typu danych. Będzie o tym mowa w dalszej części tego rozdziału.

Ćwiczenie 3.2

Zinterpretuj następujący fragment kodu programu:

```
typedef float real;
typedef double extended;
```

Zmodyfikuj przedstawiony kod — zamiast słowa kluczowego `typedef` zastosuj słowo kluczowe `using`. Znaczenie kodu przed modyfikacją i po niej powinno być identyczne.

Inicjalizacja zmiennych

Zmiennej w programie można nadać wartość początkową już podczas jej deklarowania. Operacja ta nazywa się **inicjalizacją zmiennej** (ang. *variable initialization*). Ogólna postać deklaracji zmiennej należącej do jednego z typów podstawowych połączona z jej inicjalizacją jest następująca:

```
typ_zmiennej nazwa_zmiennej = wyrażenie;
```

lub

```
typ_zmiennej nazwa_zmiennej (wyrażenie);
```

lub

```
typ_zmiennej nazwa_zmiennej {};
typ_zmiennej nazwa_zmiennej {wyrażenie};
typ_zmiennej nazwa_zmiennej = {wyrażenie};
```

gdzie:

- **typ_zmiennej** — typ danych, do którego należy zmienna,
- **nazwa_zmiennej** — identyfikator zmiennej,
- **wyrażenie** (ang. *expression*) — sekwencja zmiennych, stałych i operatorów zgodna z regułami języka C++, określająca operacje, które dają w rezultacie wartość typu zgodnego z **typem_zmiennej**.

UWAGA

Zarówno **state**, jak i **operatory** i **wyrażenia** zostały omówione szczegółowo w dalszej części tego podrozdziału.

Pierwszy z wymienionych wcześniej sposobów inicjalizacji zmiennej został odziedziczony po języku C. Dlatego też można go nazwać **C-inicjalizacją** (ang. *C-like initialization*). Sposób ten jest również nazywany **inicjalizacją kopiującą** (ang. *copy initialization*).

Drugi sposób jest nazywany **inicjalizacją bezpośrednią** (ang. *direct initialization*) albo **inicjalizacją konstruktorską** (ang. *constructor initialization*).

Trzeci sposób, wprowadzony w specyfikacji C++11, jest nazywany **inicjalizacją jednolitą** (ang. *uniform initialization*) lub **inicjalizacją klamrową** (ang. *brace initialization*). Inicjalizacja jednolita może występować w programie w różnych postaciach (formach):

- Jeżeli podczas inicjalizacji zmiennej metodą jednolitą zostanie użyta postać: **typ_zmiennej nazwa_zmiennej {};**, zmienna zostanie zainicjowana wartością zerową (lub pustą) w zależności od **typu_zmiennej**. Taka inicjalizacja jest nazywana **inicjalizacją zerową** (ang. *zero initialization*).
- Kolejna postać inicjalizacji jednolitej, **typ_zmiennej nazwa_zmiennej {wyrażenie};**, odpowiada **inicjalizacji jednolitej bezpośredniej** (ang. *direct uniform initialization*).
- Ostatnia postać, **typ_zmiennej nazwa_zmiennej = {wyrażenie};**, to **inicjalizacja jednolita kopiująca** (ang. *copy uniform initialization*).

Zalecanym sposobem inicjalizacji zmiennej jest inicjalizacja jednolita bezpośrednią.

Przykład 3.3

```
int zmienna1 = 1;
int zmienna2 (zmienna1 + 1);
int zmienna3 {};
int zmienna4 {zmienna3 + 1};
int zmienna5 = {zmienna4 + 1};
```

We fragmencie kodu zadeklarowano pięć zmiennych. Każda z nich została zainicjowana podczas deklaracji. Zmiennej **zmienna1** nadano wartość początkową równą 1 przy użyciu inicjalizacji kopiującej. Zmienna **zmienna2** została zainicjowana wartością wyrażenia **zmienna1 + 1**, które daje w rezultacie wartość 2. W tym przypadku zastosowano inicjalizację

bezpośrednią konstruktorową. Zmiennej `zmienna3` nadano wartość początkową równą 0, co wynika z wykorzystanej inicjalizacji zerowej. Zmienną `zmienna4` zainicjowano wartością wyrażenia `zmienna3 + 1` (`= 2`) przy użyciu inicjalizacji jednolitej bezpośredniej. Ostatnią zmienną (`zmienna5`) zainicjowano wartością wyrażenia `zmienna4 + 1`, które daje w rezultacie liczbę 2 — za pomocą inicjalizacji jednolitej kopiącej. W przykładzie wykorzystano operator dodawania arytmetycznego `+`.

Ćwiczenie 3.3

Zinterpretuj kolejne linie kodu we fragmencie programu:

```
float bok1 {1};  
float bok2 = {2};  
float pole, obwod;
```

Zmodyfikuj przedstawiony kod — zastosuj inny sposób inicjalizacji zmiennych `bok1` i `bok2`, np. styl odziedziczony po języku C.

Deklaracja zmiennej z dedukcją typu

Deklaracja zmiennej może być połączona z **dedukcją typu danych** (ang. *data type deduction*), do którego kompilator zaliczy tę zmienną. W tym celu wykorzystuje się słowa kluczowe `decltype` oraz `auto`.

Specyfikator `decltype` może być użyty w deklaracji zmiennej w celu wydedukowania jej typu na podstawie typu innej zmiennej lub typu wyniku zadanego wyrażenia. Postać ogólna użycia specyfikatora `decltype` jest następująca:

```
decltype(zmienna_wzorcowa) nazwa_zmiennej;
```

lub

```
decltype(wyrażenie) nazwa_zmiennej;
```

gdzie:

- `nazwa_zmiennej` — identyfikator deklarowanej zmiennej,
- `zmienna_wzorcowa` — zmienna, na podstawie której zostanie wydedukowany typ zmiennej deklarowanej,
- `wyrażenie` — wyrażenie, na podstawie którego kompilator wydedukuje typ deklarowanej zmiennej.

Należy zwrócić uwagę na to, że użycie specyfikatora `decltype` w deklaracji zmiennej nie powoduje jej automatycznej inicjalizacji.

Przykład 3.4

```
int zmienna1 {1};  
decltype(zmienna1) zmienna2;  
zmienna2 = 2;  
  
decltype(zmienna2 + 1) zmienna3;  
zmienna3 = 3;
```

W przedstawionym fragmencie programu zadeklarowano zmienną o nazwie `zmienna1` typu całkowitego `int`. Zmienna ta została zainicjowana wartością 1. Następnie została zadeklarowana zmienna `zmienna2` — ale bez inicjalizacji. Jej typ (`int`) został wydedukowany przez kompilator na podstawie typu zmiennej `zmienna1`. Zmiennej tej nadano w osobnej instrukcji wartość równą 2. Na końcu zadeklarowano zmienną o nazwie `zmienna3`. Typ zmiennej `zmienna3` (`int`) został wydedukowany na podstawie wyniku wyrażenia `zmienna2 + 1`.

Ćwiczenie 3.4

Zmodyfikuj kod przedstawiony w przykładzie 3.4 — zadeklaruj i zainicjuj zmienne `zmienna2` i `zmienna3` bez dedukcji typu tych zmiennych, tj. z jawnie określonymi typami danych, do których zmienne te należą.

Drugi z wymienionych wcześniej specyfikatorów, `auto`, może być użyty w deklaracji zmiennej połączonej z jej inicjalizacją. Za pomocą wartości początkowej kompilator może wydedukować typ zmiennej, dlatego wymagane jest podanie wartości początkowej. Postać ogólna użycia specyfikatora `auto` jest następująca:

`auto nazwa_zmiennej = wyrażenie;`

lub

`auto nazwa_zmiennej (wyrażenie);`

lub

`auto nazwa_zmiennej {wyrażenie};`

gdzie:

- `nazwa_zmiennej` — identyfikator zmiennej,
- `wyrażenie` — dowolne wyrażenie, na podstawie którego kompilator wydedukuje typ deklarowanej zmiennej. Zmiennej `nazwa_zmiennej` zostanie przypisana wartość początkowa równa wynikowi `wyrażenia`.

Przykład 3.5

```
int zmienna1 = 1;
auto zmienna2 = zmienna1 + 1;
auto zmienna3 (zmienna2 + 1);
auto zmienna4 {zmienna3 + 1};
```

Zmienna o nazwie `zmienna1` została zadeklarowana jako należąca do typu `int` oraz zainicjowana wartością 1. Następnie w kodzie występują deklaracje i inicjalizacje zmiennych: `zmienna2`, `zmienna3` i `zmienna4`. Typ każdej z nich, `int`, został wydedukowany przez kompilator na podstawie typu wartości wyrażeń, odpowiednio: `zmienna1 + 1`, `zmienna2 + 1`, `zmienna3 + 1`.

Ćwiczenie 3.5

Zmodyfikuj kod przedstawiony w przykładzie 3.5 — zadeklaruj zmienne: `zmienna2`, `zmienna3` i `zmienna4` bez dedukcji typu tych zmiennych, tj. z jawnie określonymi typami danych, do których zmienne te należą. Inicjalizację wymienionych zmiennych zrealizuj innym sposobem niż w kodzie z przykładu 3.5.

Stałe

W ujęciu ogólnym **stałe** (ang. *constants*) to wyrażenia (ang. *expressions*) mające ustaloną wartość, która się nie zmienia w trakcie działania programu. Na przykład stałą jest liczba całkowita o wartości dziesiętnej równej 1, jak również wyrażenie `1 + 1`.

Literały

Wyrażenia określające wartości stałych w kodzie źródłowym programu można formułować przy użyciu **literałów** (ang. *literals*). W zależności od typu wyrażanej wartości literały można podzielić na:

- całkowite,
- zmiennoprzecinkowe,
- znakowe,
- łańcuchowe,
- logiczne.

Literały całkowite (ang. *integer literals*) definiują stałe całkowite. Literały całkowite można zapisywać w systemie dziesiętnym (ang. *decimal*), ósemkowym (ang. *octal*) i szesnastkowym (ang. *hexadecimal*), zakończone opcjonalnie przyrostkiem — modyfikatorem (ang. *modifier*) `u` (lub `U`), `L` (lub `l`) albo `LL` (lub `ll`). Wybrane modyfikatory można ze sobą łączyć.

Na przykład `25` to literal całkowity zapisany w systemie dziesiętnym, `025` — literal całkowity zapisany w systemie ósemkowym, `0x25` — literal całkowity zapisany w systemie szesnastkowym, `25L` — literal całkowity zapisany w systemie dziesiętnym traktowany jako należący do

typu `long` (zamiast domyślnie `int`), `25UL` — literał całkowity zapisany w systemie dziesiętnym traktowany jako `unsigned long`.

Literaly zmiennoprzecinkowe (ang. *floating point literals*) wyrażają liczby rzeczywiste. Literaly zmiennoprzecinkowe można zapisywać w dwojakim sposobie: przez podanie albo jedynie części całkowitej i ułamkowej liczby, np. `1.2345`, albo dodatkowo wykładnika potęgi liczby 10 , oznaczanego jako `e` lub `E`. Na przykład literał `1.2345e-5` oznacza liczbę $1,2345 \cdot 10^{-5}$.

Literaly zmiennoprzecinkowe mogą być zakończone modyfikatorami przyrostkowymi `F` (lub `f`) oraz `U` (lub `u`). Na przykład literał `1.2345F` jest traktowany przez kompilator jako należący do typu `float` zamiast (domyślnie) do typu `double`.

Literaly znakowe (ang. *character literals*) reprezentują pojedyncze znaki. Literaly znakowe zapisuje się w pojedynczych apostrofach (ang. *single quotes*), `'`, np. `'A'`, `'a'`, `';'`.

Do literałów znakowych zalicza się również **znaki specjalne** (ang. *special characters*). Są one poprzedzone znakiem `\` (ang. *backslash*). Znaki specjalne mają przypisane określone znaczenie. Na przykład znak `'\n'` oznacza nową linię, a `'\t'` to znak tabulacji.

Literaly łańcuchowe (ang. *string literals*) obejmują napisy ujęte w podwójne apostrofy (ang. *double quotes*), `"`, np. `"Język C++"`, `"C"`.

Literaly logiczne (ang. *logical literals*) reprezentują określona wartość logiczną: albo prawda — `true`, albo fałsz — `false`.

Stałe nazwane

W programowaniu często się zdarza, że trzeba wiele razy użyć pewnej ustalonej wartości, która w przypadku ogólnym może być określona za pomocą wyrażenia. Wówczas można za każdym razem wpisywać do kodu to wyrażenie albo przypisać mu unikatowy identyfikator i zamiast wyrażenia wykorzystywać ten zdefiniowany identyfikator.

Stała, która została skojarzona z wybranym unikatowym identyfikatorem, to **stała nazwana** (ang. *named constant*). Ogólna postać definicji takiej stałej jest następująca:

```
const typ_stalej nazwa_stalej = wyrażenie;
```

lub

```
const typ_stalej nazwa_stalej (wyrażenie);
```

lub

```
const typ_stalej nazwa_stalej {wyrażenie};
```

gdzie:

- `nazwa_stalej` — identyfikator stałej,
- `typ_stalej` — typ danych, do którego należy stała,
- `wyrażenie` — dowolne wyrażenie, którego wartość kompilator skojarzy z `nazwą_stalej`.

Zamiast słowa kluczowego określającego typ_stałej można użyć specyfikatora auto. Wówczas kompilator sam wydedukuje typ definiowanej stałej.

Przykład 3.6

```
const int stala1 = 1;
const int stala2 (stala1 + 1);
const auto stala3 {stala2 + 1};
```

Na początku zdefiniowano stałą o nazwie `stala1` typu `int` i przypisano jej wartość 1. Następnie zdefiniowano stałą `stala2`, której została przypisana wartość wyrażenia `stala1 + 1`. Na końcu znajduje się definicja stałej `stala3`. Jej typ zostanie wydedukowany przez kompilator, co wynika z zastosowania specyfikatora `auto`. Stała `stala3` przyjmuje wartość wyrażenia `stala2 + 1`.



UWAGA

Stałe można definiować również przy użyciu dyrektywy preprocesora `#define`. Szczegółowe informacje dotyczące preprocesora są zawarte w dalszej części podręcznika, w rozdziale 9.

Ćwiczenie 3.6

Podaj formalną nazwę każdego z elementów składowych definicji stałej nazwanej:

```
const float PI = 3.14159;
```

Jaką funkcję spełniają poszczególne elementy składowe w tej definicji?

Zdefiniuj stałą o nazwie `liczbaEulera` równą podstawie logarytmu naturalnego `e` (liczbie Eulera), która wynosi 2,71828.

Operatory

Operator (ang. *operator*) stanowi symbol jedno- lub wieloznakowy, który informuje kompilator, jaką operację ten ma wykonać, np. operację dodawania arytmetycznego dwóch liczb. Argumentami — **operandami** (ang. *operands*) — operatorów mogą być zmienne oraz stałe.

Język C++ oferuje wiele **operatorów wbudowanych** (ang. *built-in operators*), które można podzielić na kilka grup:

- operatory przypisania,
- operatory arytmetyczne,
- operatory bitowe,
- operatory porównania,
- operatory logiczne

i inne.

Jeżeli dany operator jest umiejscowiony przed operandem (np. `++zmienna`), to nazywa się go **operatorem prefiksowym** (ang. *prefix operator*), inaczej przedrostkowym. Jeżeli położenie operatora jest odwrotne (np. `zmienna++`), jest on nazywany **operatorem postfiksowym** (ang. *postfix operator*), czyli przyrostkowym.

Operatory przypisania

Zadaniem **operatorów przypisania** (ang. *assignment operators*) jest modyfikacja wartości zmiennej. Język C++ jest wyposażony w:

- operator przypisania prostego,
- operatory przypisania złożonego.

Operator przypisania prostego (ang. *simple assignment operator*) umożliwia „przypisanie” wartości określonego wyrażenia do zmiennej. Postać ogólna wyrażenia zawierającego operator przypisania prostego jest następująca:

`nazwa_zmiennej = wyrażenie;`

lub

`nazwa_zmiennej = (wyrażenie);`

lub

`nazwa_zmiennej = {wyrażenie};`

gdzie `nazwa_zmiennej` oznacza zmienną, której należy przypisać wartość wyniku wyrażenia z prawej strony operatora. Typ wyniku wyrażenia powinien być zgodny z typem zmiennej.

Przykład 3.7

```
int zmienna1, zmienna2, zmienna3;
zmienna1 = 1;
zmienna2 = (zmienna1 + 1);
zmienna3 = {zmienna2 + 1};
```

W pierwszej linii kodu zadeklarowano trzy zmienne całkowite (`int`): `zmienna1`, `zmienna2` oraz `zmienna3`. W kolejnych trzech liniach wykorzystano operator przypisania prostego `=`. Do zmiennej `zmienna1` przypisano wartość równą 1, do `zmienna2` — wynik wyrażenia `zmienna1 + 1` (który wynosi 2), a do `zmienna3` — wynik wyrażenia `zmienna2 + 1` (czyli 3).

Ćwiczenie 3.7

Zmodyfikuj kod źródłowy przedstawiony w przykładzie 3.7 — zadeklaruj zmienne: `zmienna1`, `zmienna2` oraz `zmienna3`, wykorzystując dedukcję ich typów na podstawie typu wartości początkowej i/lub typu innych zmiennych. Wartości końcowe wymienionych zmiennych powinny być takie same jak w przykładzie 3.7.

Operatory przypisania złożonego (ang. *compound assignment operators*) z kolei pozwalają zmodyfikować wartość zmiennej przez wykonanie na niej określonej operacji, np. operacji dodania do niej wartości innej zmiennej. Jest to równoważne przypisaniu wyniku wykonywanej operacji do modyfikowanej zmiennej. Wybrane operatory przypisania złożonego przedstawiono w tabeli 3.1.

Tabela 3.1. Wybrane operatory przypisania złożonego

Nazwa	Symbol	Przykład	Przykład równoważny
Przypisanie dodawania	<code>+=</code>	<code>zmienna += 10</code>	<code>zmienna = zmienna + 10</code>
Przypisanie odejmowania	<code>-=</code>	<code>zmienna -= 10</code>	<code>zmienna = zmienna - 10</code>
Przypisanie mnożenia	<code>*=</code>	<code>zmienna *= 10</code>	<code>zmienna = zmienna * 10</code>
Przypisanie dzielenia	<code>/=</code>	<code>zmienna /= 1</code>	<code>zmienna = zmienna / 10</code>
Przypisanie modulo	<code>%=</code>	<code>zmienna %= 1</code>	<code>zmienna = zmienna % 10</code>

Przykład 3.8

```
int zmienna1 = 1, zmienna2 = 2, zmienna3 = 3;
zmienna1 += 1; //2
zmienna2 *= zmienna1 + 1; //2 * 3 = 6
zmienna3 %= 2; //3 / 2 = 1 reszta 1
```

W pierwszej linii kodu zostały zadeklarowane i zainicjowane trzy zmienne: `zmienna1`, `zmienna2` i `zmienna3`. Następnie wykorzystano operator przypisania złożonego `+=`. Zmiennej `zmienna1` została przypisana jej bieżąca wartość zwiększoną o 1, czyli została wykonana instrukcja `zmienna1 = zmienna1 + 1;`. W kolejnej instrukcji bieżąca wartość zmiennej `zmienna2` (równa 2) została pomnożona przez wartość wyrażenia `zmienna1 + 1`, czyli przez 3. Jest to równoważne instrukcji: `zmienna2 = zmienna2 * (zmienna1 + 1);`. Wartość końcowa zmiennej `zmienna3` po wykonaniu ostatniej linii kodu to 1, ponieważ reszta z dzielenia wartości 3 przez 2 wynosi 1.

Ćwiczenie 3.8

Szczegółowo przeanalizuj następujący fragment kodu:

```
int x = 1;
int y = 1;
y += x;
x += y;
int z = y = x;
```

Jakie wartości osiągną zmienne `x`, `y` i `z` po wykonaniu całości kodu?

Operatory arytmetyczne

Operatory arytmetyczne (ang. *arithmetic operators*) mają za zadanie obliczenie wyniku określonej operacji arytmetycznej. Można je podzielić w zależności od liczby argumentów na:

- jednoargumentowe,
- dwuargumentowe.

Wykaz operatorów jednoargumentowych (ang. *unary operators*) przedstawiono w tabeli 3.2.

Tabela 3.2. Operatory arytmetyczne jednoargumentowe

Nazwa	Symbol	Przykład
Operator znaku plus	+	+a
Operator znaku minus	-	-a
Operator inkrementacji preinkrementacja postinkrementacja	++	++zmienna zmienna++
Operator dekrementacji predekrementacja postdekrementacja	--	--zmienna zmienna--

Jeżeli operator **preinkrementacji** (ang. *pre-incrementation*) lub **predekrementacji** (ang. *pre-decrementation*) danej zmiennej jest częścią składową wyrażenia, to wartość tego wyrażenia jest obliczana dopiero po zwiększeniu lub zmniejszeniu o 1 wartości zmiennej. W przypadku **postinkrementacji** (ang. *post-incrementation*) i postdekrementacji (ang. *post-decrementation*) sytuacja jest odwrotna — najpierw obliczana jest wartość wyrażenia, a dopiero po zakończeniu tej operacji wartość zmiennej jest zwiększana lub zmniejszana o 1.

Przykład 3.9

```
int zmienna1 = 1;
int zmienna2 = 2;
int wynik = ++zmienna1 + zmienna2;
```

Na początku w kodzie występuje deklaracja i inicjalizacja zmiennych `zmienna1` i `zmienna2`. Następną operacją jest preinkrementacja zmiennej `zmienna1`. Jej wartość po tej operacji wynosi 2. Następnie obliczana jest wartość wyrażenia `zmienna1 + zmienna2` i jego wynik (równy 4) jest na końcu podstawiany do zmiennej `wynik`.

Ćwiczenie 3.9

Zmodyfikuj kod źródłowy zawarty w przykładzie 3.9 — zamiast preinkrementacji zmiennej `zmienna1` użyj postinkrementacji tej zmiennej. Jaka będzie wartość zmiennej `wynik` po wykonaniu całości zmodyfikowanego kodu?

Operatory dwuargumentowe „operują” na dwóch argumentach (operandach), np. `zmienna1 + zmienna2`. Wykaz operatorów dwuargumentowych zawiera tabela 3.3.

Tabela 3.3. Operatory arytmetyczne dwuargumentowe

Nazwa	Symbol	Przykład
Operator dodawania	+	a + b
Operator odejmowania	-	a - b
Operator mnożenia	*	a * b
Operator dzielenia	/	a / b
Operator modulo	%	a % b

W ogólności typy argumentów operatorów wymienionych w tabeli 3.3 mogą być zarówno rzeczywiste (np. `float`, `double`), jak i całkowite (np. `int`, `long`). Jednakże operator `modulo`, który pozwala na obliczenie reszty z dzielenia, wymaga użycia argumentów (operandów) całkowitych.

W danym wyrażeniu może występować wiele operatorów (w tym operatorów arytmetycznych) oraz wiele argumentów (operandów), na których są wykonywane zadane operacje. Na przykład w wyrażeniu `w = ++a + b % c;` występują trzy operatory: inkrementacji `++`, dodawania `+` i modulo `%`, oraz trzy argumenty: `a`, `b` i `c`.

Priorytet operatorów

Operatory, które odpowiadają za wykonanie określonych operacji, mają różnych **priorytet** (ang. *operator priority*) — czyli **pierwszeństwo w wykonaniu** (ang. *operator precedence*). Operatory o wyższym priorytecie mają pierwszeństwo w wykonaniu nad operatorami o niższym priorytecie. Na przykład operatory inkrementacji i dekrementacji mają wyższy priorytet od operatorów mnożenia, dzielenia i modulo, a operatory mnożenia, dzielenia i modulo mają wyższy priorytet od operatorów dodawania i odejmowania. Tym samym operacje inkrementacji i dekrementacji zostaną wykonane przed operacjami mnożenia, dzielenia i modulo, a operacje mnożenia, dzielenia i modulo zostaną wykonane przed operacjami dodawania i odejmowania.

Jeżeli w danym wyrażeniu występują wyłącznie operatory o tym samym priorytecie (np. operatory mnożenia i dzielenia), to działania są wykonywane w kolejności od lewej do prawej strony tego wyrażenia.

Przykład 3.10

```
int a = 99, b = 20, c = 3;
int w = ++a + b % c;
```

Po zadeklarowaniu i inicjalizacji zmiennych `a`, `b` i `c` powinna zostać obliczona wartość wyrażenia `++a + b % c`. W tym celu na początku zostanie poddana inkrementacji zmienna `a`. Po tej operacji wartość `a` wynosi `100`. Następnie zostanie obliczona wartość wyrażenia `b % c`. Wynik tej operacji to `2` (reszta z dzielenia całkowitego `20 / 3`). Kolejna operacja to dodawanie `100 + 2`, która daje w rezultacie wartość `102`. Wartość ta zostaje na końcu przypisana do zmiennej `w`.

Ćwiczenie 3.10

Przeanalizuj kod źródłowy:

```
int a = 1, b = 2, c = 3;
int w = a++ + --b % c--;
```

Jaka wartość zostanie przypisana do zmiennej `w` po obliczeniu wyrażenia `a++ + --b % c--`?

Jeżeli w danym wyrażeniu występują operatory o różnym priorytecie, to kolejność działań wynikającą z pierwszeństwa w wykonaniu można zmienić przez zastosowanie nawiasów okrągłych ().

Przykład 3.11

```
int a = 99, b = 20, c = 3;
int w = (++a + b) % c;
```

Na początku zostanie poddana inkrementacji zmieniona `a` — jak w przykładzie 3.10. Jednakże jako druga zostanie wykonana operacja dodawania wartości zmiennej `a` (100) do wartości `b` równej 20, a nie dzielenie *modulo*, co wynika z priorytetu operatorów. Wynik tej operacji zostanie podzielony modulo przez wartość zmiennej `c`. Reszta z dzielenia całkowitego 120 modulo 3 to 0. I ta ostatnia wartość (0) zostanie na końcu przypisana do zmiennej `w`.

Ćwiczenie 3.11

Przeprowadź szczegółową analizę fragmentu kodu:

```
int a {1};
int b {1};
int c {1};
int w1 {++a * --b + c++};
int w2 {++a * (--b + c++)};
```

Jakie wartości osiągną zmienne `w1` i `w2` po wykonaniu tego kodu?

UWAGA

Więcej informacji dotyczących priorytetów operatorów i kolejności wykonywania operacji można znaleźć w dokumentacji języka C++ — w artykule *C++ Operator Precedence* (https://en.cppreference.com/w/cpp/language/operator_precedence).

Operatory bitowe

Operatory bitowe (ang. *bitwise operators*) pozwalają na wykonywanie operacji na poszczególnych bitach liczb całkowitych. Podobnie jak operatory arytmetyczne, można je podzielić na jedno- i dwuargumentowe.

Wykaz operatów bitowych wraz z opisem przedstawiono w tabeli 3.4.

Tabela 3.4. Operatory bitowe

Nazwa	Symbol	Przykład	Opis
Operator bitowy AND	&	liczba1 & liczba2	Wykonanie iloczynu logicznego AND na odpowiadających sobie bitach dwóch liczb
Operator bitowy OR		liczba1 liczba2	Wykonanie sumy logicznej OR na odpowiadających sobie bitach dwóch liczb
Operator bitowy XOR	^	liczba1 ^ liczba2	Wykonanie sumy logicznej XOR na odpowiadających sobie bitach dwóch liczb
Operator bitowy NOT	~	~liczba	Wykonanie negacji logicznej NOT na każdym bicie liczby
Operator przesunięcia w lewo o <i>k</i> pozycji	<<	liczba << k	Wykonanie przesunięcia każdego bitu liczby o zadaną liczbę <i>k</i> pozycji w lewo
Operator przesunięcia w prawo o <i>k</i> pozycji	>>	liczba >> k	Wykonanie przesunięcia każdego bitu liczby o zadaną liczbę <i>k</i> pozycji w prawo

Przykład 3.12

```
int liczba = 7; // 0111
int wynik;

wynik = liczba & liczba; // 0111
wynik = liczba | liczba; // 0111
wynik = liczba ^ liczba; // 0000
wynik = ~liczba; // -1000
wynik = liczba << 1; // 1110
wynik = liczba >> 1; // 0011
```

Na początku zadeklarowano i zainicjowano wartością 7 zmienną `liczba`. W kodzie binarnym liczba 7 to 0111. Następnie zadeklarowano zmienną `wynik`, w której zapisywane będą kolejno rezultaty wykonania poszczególnych operacji bitowych. Bitowe AND przyjmuje na danej pozycji wartość 1 wtedy i tylko wtedy, gdy odpowiadające sobie bity mają wartości równe 1. Bitowe AND dla dwóch identycznych argumentów `liczba` daje w wyniku liczbę 7 w systemie dziesiętnym (binarnie 0111). Wartość bitowego OR to również 7 w systemie dziesiętnym. Wynika to z faktu, że bitowe OR przyjmuje na danej pozycji wartość 0 wtedy i tylko wtedy, gdy odpowiadające sobie bity mają wartości równe 0. W przeciwnym razie bitowe OR na tej pozycji osiąga wartość 1. Wartość bitowego XOR dla dwóch argumentów `liczba` wynosi 0, co wynika z faktu, że wynik XOR na danej pozycji jest równy 1 wtedy i tylko wtedy, gdy odpowiadające sobie bity w argumentach są różne. Wynik bitowego NOT to dziesięćne -8 (w systemie binarnym -1000). Operator jednoargumentowy bitowego NOT odwraca wszystkie bity w liczbie będącej jej argumentem. Wynik przesunięcia w lewo o jedną pozycję dla argumentu `liczba` daje w rezultacie liczbę 14 w systemie dziesiętnym (binarnie 1110). Przesunięcie w prawo o jedną pozycję daje wynik 3 dziesiętnie (0011 binarnie).

Ćwiczenie 3.12

Przeprowadź szczegółową analizę fragmentu kodu:

```
int a {1};
int b {1};
int w1 {a & b};
int w2 {a | b};
int w3 {~a >> 1};
int w4 {~b << 1};
```

Jakie wartości osiągną zmienne: w1, w2, w3 i w4 po wykonaniu tego kodu?

Operatory porównania

Operatory porównania (ang. *comparison operators*) są używane w celu porównania wartości dwóch wyrażeń. Do operatorów porównania zalicza się:

- **operatorzy równości** (ang. *equality operators*),
- **operatorzy relacyjne** (ang. *relational operators*).

Wykaz operatorów porównania zawarto w tabeli 3.5.

Tabela 3.5. Operatory porównania

Nazwa	Symbol	Przykład
Operator „równy”	<code>==</code>	<code>zmienna1 == zmienna2</code>
Operator „różny”	<code>!=</code>	<code>zmienna1 != zmienna2</code>
Operator „mniejszy”	<code><</code>	<code>zmienna1 < zmienna2</code>
Operator „większy”	<code>></code>	<code>zmienna1 > zmienna2</code>
Operator „mniejszy lub równy”	<code><=</code>	<code>zmienna1 <= zmienna2</code>
Operator „większy lub równy”	<code>>=</code>	<code>zmienna1 >= zmienna2</code>

Wynik wykonania operacji porównania wartości dwóch argumentów (operandów) należy do typu logicznego i wynosi albo `true`, albo `false`.

Przykład 3.13

```
int zmienna1 = 1;
int zmienna2 = 2;
bool wynik;
wynik = zmienna1 == zmienna2;
wynik = zmienna1 < zmienna2;
```

Po zadeklarowaniu i zainicjowaniu zmiennych całkowitych `zmienna1` i `zmienna2` następuje deklaracja zmiennej logicznej o nazwie `wynik`. Kolejna linia kodu zawiera wyrażenie `zmienna1 == zmienna2`, w którym za pomocą operatora równości `==` następuje sprawdzenie, czy zmienne `zmienna1` i `zmienna2` są równe. Wynik tego wyrażenia (`false`) jest zapisywany

w zmiennej wynik. W ostatniej linii kodu został wykorzystany operator < (mniejszy). Wynik wyrażenia zmienna1 < zmienna2 (true) jest podstawiany do zmiennej wynik.

Ćwiczenie 3.13

Przeprowadź analizę fragmentu kodu:

```
int a {1};
int b {10};
bool w1 {a > b};
bool w2 {a < b};
bool w3 {a == b};
bool w4 {a != b};
```

Jakie wartości osiągną zmienne: w1, w2, w3 i w4 po wykonaniu tego kodu?

Operatory logiczne

Operatory logiczne (ang. *logical operators*) operują na argumentach (operandach) typu logicznego `bool`, które mogą być stałymi i zmiennymi. W ogólności argumentami mogą być wyrażenia dające w wyniku albo wartość logiczną `true`, albo `false`. Wykaz operatorów logicznych zawarto w tabeli 3.6.

Tabela 3.6. Operatory logiczne

Nazwa	Symbol	Opis
Operator NOT (negacja)	!	Zwraca wartość logiczną <code>false</code> , jeśli argument jest prawdą (<code>true</code>), i odwrotnie
Operator AND (iloczyn logiczny)	&& and	Zwraca wartość <code>true</code> wtedy i tylko wtedy, gdy oba argumenty są prawdą (<code>true</code>). W przeciwnym razie zwraca wartość <code>false</code>
Operator OR (suma logiczna)	 or	Zwraca wartość <code>true</code> wtedy, gdy co najmniej jeden argument jest prawdą (<code>true</code>). W przeciwnym razie zwraca wartość <code>false</code>

Przykład 3.14

```
int zmienna1 = 1;
int zmienna2 = -1;
bool wynik;
wynik = !(zmienna1 < zmienna2);
wynik = (zmienna1 > 0) and (zmienna2 > 0);
wynik = (zmienna1 > 0) || (zmienna2 > 0);
```

Na początku zadeklarowano i zainicjowano zmienne `zmienna1` i `zmienna2`. Następną instrukcję stanowi deklaracja zmiennej logicznej `wynik`. W czwartej linii wykorzystano operator negacji `!`. Po tym wyznaczana jest wartość relacji `zmienna1 < zmienna2`. Jej wynik to `false`. Negacja `false` to `true`. I ta wartość jest podstawiana do zmiennej `wynik`. W kolejnej linii kodu

najpierw wyznaczane są wartości wyrażeń: `zmienna1 > 0` (`true`) i `zmienna2 > 0` (`false`). Taka kolejność wynika z zastosowania nawiasów (), które zmieniają kolejność wykonywania operacji. Następnie obliczany jest iloczyn logiczny z dwóch wartości wyrażeń obliczonych wcześniej. Jego wynik to `false`. W ostatniej linii następuje obliczenie sumy logicznej `||`. Jej wynik to `true`, ponieważ wartość jednego z wyrażeń wynosi `true`.

Ćwiczenie 3.14

Przeprowadź analizę fragmentu kodu:

```
int a (1);
int b (10);
int c (100);
bool w1 (a < b && b < c || a == b);
bool w2 ((a < b) && (b < c) || (a == b));
bool w3 (a < b and (b < c or a == b));
```

Jakie wartości osiągną zmienne: `w1`, `w2` i `w4` po wykonaniu tego kodu?

Operator sizeof

Operator `sizeof` zwraca rozmiar (w bajtach), jaki zajmuje w pamięci operacyjnej określony typ danych, stała, zmienna lub w ogólności wyrażenie. Ogólna postać użycia operatora `sizeof` jest następująca:

`sizeof (typ_danych)`

lub

`sizeof wyrażenie`

gdzie `typ_danych` oznacza typ danych (podstawowy lub zdefiniowany przez programistę), a `wyrażenie` jest dowolnym wyrażeniem w języku C++ zawierającym stałe, zmienne i operatory.

Przykład 3.15

```
const int stala = 1;
int zmienna = 1;
int rozmiar;

// Określenie rozmiaru typu int:
rozmiar = sizeof(int); // 4
// Określenie rozmiaru stałej stala:
rozmiar = sizeof stala; // 4
// Określenie rozmiaru zmiennej zmienna:
rozmiar = sizeof zmienna; // 4
// Określenie rozmiaru wyrażenia (stala + zmienna):
rozmiar = sizeof (stala + zmienna); // 4
```

Ćwiczenie 3.15

Z wykorzystaniem kodu przedstawionego w przykładzie 3.15 określ rozmiar (w bajtach) stałej o nazwie `PI` o wartości 3,14 oraz rozmiary danych (np. zmiennych) należących do typów danych `float` i `double`.

Operator przecinkowy

Operator przecinkowy (ang. *comma operator*) pozwala na wykonanie większej liczby instrukcji w sytuacji, gdy składnia języka wymaga tylko jednej.

Przykład 3.16

```
int bok1, bok2, pole;
pole = (bok1 = 1, bok2 = 2, bok1 * bok2);
```

We fragmencie kodu obliczane jest pole prostokąta. Boki prostokąta są reprezentowane przez zmienne `bok1` i `bok2`, a jego pole — przez zmienną `pole`. W pierwszej linii kodu deklarowane są potrzebne zmienne. Nazwy zmiennych są oddzielone od siebie przecinkami. W drugiej linii kodu najpierw wykonywana jest instrukcja `bok1 = 1`, następnie `bok2 = 2`, a potem obliczana jest wartość wyrażenia `bok1 * bok2`, określającego pole prostokąta. Na końcu obliczona wartość pola zostaje przypisana do zmiennej `pole`. Instrukcje składowe w drugiej linii kodu są oddzielone od siebie przecinkami.

Ćwiczenie 3.16

Z wykorzystaniem kodu źródłowego przedstawionego w przykładzie 3.16 zinterpretuj następujący fragment kodu:

```
float bok1, bok2, obwod;
obwod = (bok1 = 1, bok2 = 2, 2 * bok1 + 2 * bok2);
```

UWAGA

Do wbudowanych operatorów języka C++ należy również operator warunkowy (ang. *conditional operator*) oraz operator wywołania funkcji (ang. *function call operator*). Operator warunkowy przedstawiono w podrozdziale 3.2, poświęconym podejmowaniu decyzji w programie, a operator wywołania funkcji — w rozdziale 8.

Wyrażenia

O **wyrażeniach** (ang. *expressions*) była już mowa wcześniej w tym rozdziale. Dla przypomnienia: wyrażenie to sekwencja zmiennych, stałych i operatorów zgodna z regułami języka C++, określająca obliczenia (operacje), które dają w rezultacie wartość określonego typu. Wyrażenie składa się z operandów (argumentów) połączonych za pomocą operatorów. Na przykład wyrażeniem jest suma dwóch liczb całkowitych `a + b` lub warunek (`a > 0`) `&&` (`b > 0`).



UWAGA

W ogólności wyrażenie może zawierać również wywołanie funkcji (ang. *function call*). Tematyka ta została omówiona w dalszej części podręcznika — w rozdziale 8.

Wyrażenia można podzielić w zależności od typu ich wyniku (wyznaczonej wartości) na:

- **wyrażenia stałe** (ang. *constant expressions*), np. `1 + 10`,
- **wyrażenia całkowite** (ang. *integer expressions*), np. `10 * a`, gdzie `a` należy do typu `int`,
- **wyrażenia rzeczywiste** (ang. *float expressions*), np. `10 / a`, gdzie `a` należy do typu `float`,
- **wyrażenia bitowe** (ang. *bitwise expressions*), np. `a | b`, gdzie `a` i `b` należą do typu `int`,
- **wyrażenia porównania** (ang. *comparison expressions*), np. `a > 0`, gdzie `a` należy do typu `int`,
- **wyrażenia logiczne** (ang. *logical expressions*), np. `a && b`, gdzie `a` i `b` należą do typu `bool`.

UWAGA

Ważnym rodzajem wyrażeń są również wyrażenia wskaźnikowe (ang. *pointer expressions*), które zostały zaprezentowane w rozdziale 4.

W ogólności wyrażenie, jako sekwencja argumentów (operandów) połączonych za pomocą operatorów, może stanowić kombinację wyrażeń różnego (odmiennego) typu. Takie wyrażenia nazywa się **wyrażeniami złożonymi** (ang. *compound expressions*).

Konwersja typu

Konwersja typu (ang. *type conversion*) oznacza zmianę typu. W czasie konwersji tworzona jest nowa wartość pewnego typu z wartości innego typu. W ogólności konwersji typu mogą podlegać stałe, zmienne i wyrażenia. Konwersja typu może się odbywać w sposób niejawnym („po cichu”) lub jawnym. Z tego wynikają rodzaje konwersji:

- konwersja niejawną,
- konwersja jawną.

Konwersja niejawnia

Konwersja niejawnia (ang. *implicit conversion*) jest realizowana przez kompilator w sposób automatyczny — „po cichu”, bez użycia jakichkolwiek jawnych (widocznych w kodzie) poleceń jej wykonania. Dlatego ten rodzaj konwersji jest również nazywany **konwersją automatyczną** (ang. *automatic conversion*). Podczas konwersji automatycznej konwertowana wartość jest kopiowana w sposób niejawny do innego typu, który jest kompatybilny z typem początkowym. Na przykład w instrukcji `long zmiennaLong = 10;` najpierw w sposób automatyczny realizowana jest niejawnia konwersja wartości `10` z typu `int` na (kompatybilny z typem `int`) typ `long`, a dopiero później ta skonwertowana wartość jest przypisywana do zmiennej `zmiennaLong`.

Konwersja niejawną może być realizowana przez kompilator na typach danych podstawowych, np. `int`, `long`, `float`, `double`. Stąd ten rodzaj konwersji ma jeszcze jedną nazwę: **konwersja standardowa** (ang. *standard conversion*).

Konwersja standardowa zwykle zachodzi wtedy, gdy w danym wyrażeniu występuje więcej niż jeden typ danych. Jeżeli konwersja jest realizowana z typu „mniejszego” na typ „większy”, np. z typu `int` na typ `long`, to operacja ta nazywa się **promocją typu** (ang. *type promotion*). W promocji typu nie są tracone żadne informacje, np. precyzja. Jeżeli natomiast kompilator jest zmuszony wykonać konwersję z typu „większego” na typ „mniejszy”, np. z typu `double` na typ `float` lub typ `int`, niektóre informacje, np. część ułamkowa liczby, mogą zostać utracone.

Przykład 3.17

```
// Deklaracja zmiennej zmiennaLong typu long int:  
long zmiennaLong;  
  
// Definicja stałej stalaInt:  
const int stalaInt = 1;  
  
// Nadanie wartości zmiennej zmiennaLong:  
zmiennaLong = stalaInt; // 1  
/* UWAGA  
 * W wyrażeniu powyżej następuje niejawnia konwersja typu wartości stałej stalaInt z typu int na typ long —  
 * promocja typu.  
 */  
  
// Deklaracja i inicjalizacja zmiennej typu int:  
int zmiennaInt = 10;  
zmiennaLong = zmiennaInt; // 10  
/* UWAGA  
 * W wyrażeniu powyżej następuje niejawnia konwersja typu bieżcej wartości zmiennej liczbaInt z int na long.  
 */  
  
// Deklaracja i inicjalizacja zmiennej zmiennaDouble typu double:  
double zmiennaDouble = zmiennaInt + 1.5F; // 11.5  
/* UWAGA  
 * W wyrażeniu powyżej następuje niejawnia konwersja typu wartości zmiennej zmiennaInt z typu int na typ double.  
 * Oprócz tego zachodzi promocja typu wartości literalu zmiennoprzecinkowego 1.5F z typu float na typ double.  
 */  
zmiennaInt = zmiennaDouble; // 11  
/* UWAGA  
 * W wyrażeniu powyżej następuje niejawnia konwersja typu bieżcej wartości zmiennej zmiennaDouble  
 * z typu double na typ int.  
 * Konwersja z typu „większego” na typ „mniejszy” może się wiązać z utratą pewnych informacji, np. utratą precyzji.  
 */
```

W przykładzie zademonstrowano zarówno konwersję standardową z typu „mniejszego” na „większy”, tj. promocję typu (z typu `int` na typ `long`, z `int` na `double`, z `float` na `double`), jak i konwersję z typu „większego” na „mniejszy” (z typu `double` na typ `int`). Pokazano, że w tym ostatnim przypadku konwersja może prowadzić do utraty pewnych ważnych informacji, np. części ułamkowej liczby. Szczegółowe omówienie instrukcji zawartych w przykładzie przedstawiono w formie bieżących komentarzy w kodzie.

Ćwiczenie 3.17

Na podstawie kodu źródłowego zawartego w przykładzie 3.17 napisz przykład kodu, w którym wykorzystuje się konwersję niejawną z typu mniejszego na większy (czyli promocję typu) i odwrotnie.

Konwersja jawnia

Konwersja jawnia (ang. *explicit conversion*) jest definiowana przez programistę — użytkownika. Polega ona na wydaniu przez niego jawnego, bezpośredniego polecenia dla kompilatora, aby wykonał stosowną, zadaną konwersję typu określonej danej. Konwersja jawnia często jest nazywana **rzutowaniem typu** (ang. *type casting*). Składnia języka C++ dopuszcza trzy sposoby zapisu rzutowania typu:

- zapis odziedziczony po języku C,
- zapis funkcyjny,
- zapis w stylu C++11.

Wynikają z tego trzy postacie ogólne składni konwersji jawniej:

(nowy_typ) wyrażenie

lub

nowy_typ (wyrażenie)

lub

nowy_typ {wyrażenie}

gdzie `nowy_typ` oznacza nazwę typu docelowego, na który rzutuje się (faktyczny) typ wartości określonego `wyrażenia`.

UWAGA

W języku C++ istnieją jeszcze inne sposoby rzutowania typu, np. przy wykorzystaniu operatora `static_cast`. Więcej informacji na ten temat można znaleźć w dokumentacji języka C++ na stronie cppreference.com.

Przykład 3.18

```
// Deklaracja i inicjalizacja zmiennej zmiennaFloat typu float:  
float zmiennaFloat = 1.5F;  
  
// Deklaracja zmiennej zmiennaDouble typu double:  
double zmiennaDouble;  
zmiennaDouble = (double)zmiennaFloat; // rzutowanie w stylu języka C  
/* UWAGA  
 * W wyrażeniu powyżej następuje jawną konwersja typu wartości zmiennej zmiennaFloat z typu float na typ double,  
 * a więc konwersja z typu „mniejszego” na typ „większy”.  
 */  
  
// Deklaracja i inicjalizacja zmiennej zmiennaInt typu int:  
int zmiennaInt = int (zmiennaDouble); // notacja funkcyjna  
/* UWAGA  
 * W wyrażeniu powyżej następuje rzutowanie typu wartości zmiennej zmiennaDouble z typu double na typ int,  
 * a więc z typu „większego” na typ „mniejszy”.  
 */  
  
// Rzutowanie typu w stylu C++11:  
zmiennaFloat = float {zmiennaDouble + 1};
```

We fragmencie kodu wykorzystano rzutowanie typów, czyli jawną konwersję typów. Przedstawiono wszystkie trzy sposoby zapisu rzutowania. Zademonstrowano tutaj rzutowanie z typu „mniejszego” na „większy” (z typu `float` na typ `double`) oraz z typu „większego” na typ „mniejszy” (z typu `double` na typ `int` i z typu `double` na typ `float`).

Ćwiczenie 3.18

Na podstawie przykładu 3.18 napisz program, w którym realizowana jest konwersja z typu `float` na typ `double` oraz z typu `int` na `long int`. Napisz program w dwóch wariantach. W wariantie pierwszym zastosuj konwersję niejawną, a w drugim — jawną.

3.1.2. Przestrzenie nazw

Przestrzeń nazw (ang. *namespace*) to deklaratywny region (zakres), w którym można deklarować i definiować różne elementy (jednostki) programu (typy danych, zmienne, stałe, funkcje itp.) o określonych identyfikatorach. Przestrzeń nazw pozwala na grupowanie tych jednostek pod względem logicznym. Jeżeli dana jednostka (np. stała) została zdefiniowana w określonej przestrzeni nazw, to zasięg — zakres (ang. *scope*) — jej identyfikatora odnosi się wyłącznie do tej przestrzeni nazw.

Stosowanie przestrzeni nazw pozwala podzielić zakres globalny (ang. *global scope*) na mniejsze zakresy — przestrzenie nazw, z których każda ma własną nazwę.

W danej przestrzeni nazw określony identyfikator może reprezentować wyłącznie pojedynczą jednostkę, np. stałą. Tym samym w przestrzeni nazw nie mogą występować np. dwie stałe o identycznych nazwach. Z drugiej strony w różnych przestrzeniach nazw mogą się znajdować jednostki (stałe, zmienne itp.) o takich samych nazwach.

Ogólna postać definicji przestrzeni nazw jest następująca:

```
namespace nazwa {
    lista_deklaracji;
    lista_definicji;
}
```

gdzie `nazwa` oznacza nazwę tworzonej przestrzeni nazw, a `lista_deklaracji` i `lista_definicji` reprezentują zawarte w niej deklaracje i definicje.

Odwołanie się do jednostki zadeklarowanej lub zdefiniowanej w określonej przestrzeni nazw jest realizowane przy użyciu **operatora rozpoznawania zakresu** (ang. *scope resolution operator*), :::. Ogólna postać użycia operatora rozpoznawania zakresu jest następująca:

`nazwa_przestrzeni_nazw :: identyfikator_jednostki`

gdzie `nazwa_przestrzeni_nazw` oznacza identyfikator przestrzeni nazw, w której została zadeklarowana lub zdefiniowana jednostka (np. typ danych, stała, zmienna) o podanym `identyfikatorze_jednostki`.

Operator rozpoznawania zakresu czasem jest nazywany po prostu **operatorem zakresu** (ang. *scope operator*) lub **kwalifikatorem zakresu** (ang. *scope qualifier*). Konstrukcja `nazwa_przestrzeni_nazw :: identyfikator_jednostki` zaś jest nazywana **pełną nazwą** (ang. *full name*) lub **w pełni kwalifikowaną nazwą** (ang. *fully qualified name*) jednostki.

Przykład 3.19

```
// Definicja przestrzeni nazw o nazwie p1:
namespace p1 {
    // Definicja nowego typu danych o nazwie real:
    typedef float real;
    /* UWAGA
     * Identyfikator real stanowi faktycznie alias nazwy float.
     */
    // Definicja stałej PI:
    const float PI = 3.14159;
}

// Definicja przestrzeni nazw p2:
namespace p2 {
    // Definicja stałej PI:
    const double PI = 3.14159;
}
```

W przedstawionym kodzie źródłowym zdefiniowano dwie przestrzenie nazw. Identyfikator pierwszej z nich to `p1`, a drugiej — `p2`. W przestrzeni nazw `p1` zdefiniowano dwie jednostki (elementy): typ danych o nazwie `real` oraz stałą o nazwie `PI`. Identyfikator `real` to faktycznie alias nazwy predefiniowanego typu danych `float`. W przestrzeni nazw `p2` została natomiast zdefiniowana tylko jedna jednostka — stała `PI`. Należy zwrócić uwagę, że identyfikator stałej zdefiniowanej w przestrzeni nazw `p1`, `PI`, jest taki sam jak identyfikator stałej zdefiniowanej w przestrzeni nazw `p2`.

Odwołanie się w programie do typu danych `real`, zdefiniowanego w przestrzeni nazw `p1`, można zrealizować za pomocą konstrukcji `p1::real`, np. `p1::real promien = 1;`. Odwołanie się do stałej `PI` zdefiniowanej w `p1` zaś uzyskuje się przy użyciu konstrukcji `p1::PI`, np. `p1::real obwod = 2 * p1::PI * promien;`.

W tym samym programie można oczywiście korzystać ze stałej `PI` zdefiniowanej w przestrzeni nazw `p2`. Realizuje się to za pomocą konstrukcji `p2::PI`, np. `p1::real pole = p2::PI * promien * promien;`. W ostatniej instrukcji wykorzystano typ `real`, zdefiniowany w przestrzeni nazw `p1`, oraz stałą `PI` zdefiniowaną w przestrzeni nazw `p2`.

Ćwiczenie 3.19

Zdefiniuj przestrzeń nazw o nazwie `boolean_namespace`. W podanej przestrzeni zdefiniuj typ logiczny o nazwie `typ_logiczny` oraz stałe logiczne `prawda` i `falsz`.

Podawanie za każdym razem w kodzie programu nazwy przestrzeni nazw, w której została zadeklarowana lub zdefiniowana dana jednostka (np. stała), jest niepraktyczne i niewygodne. Staje się to widoczne zwłaszcza wtedy, gdy dana jednostka jest często używana w programie. Można temu zapobiec przez zadeklarowanie w programie użycia określonej przestrzeni nazw. Ogólna postać deklaracji użycia przestrzeni nazw jest następująca:

```
using namespace nazwa_przestrzeni_nazw;
```

gdzie `using` i `namespace` to słowa kluczowe, a `nazwa_przestrzeni_nazw` oznacza identyfikator deklarowanej do użycia przestrzeni nazw.

UWAGA

Deklaracja użycia przestrzeni nazw powinna się znajdować w programie przed kodem programu głównego. Pojęcie programu głównego zostało omówione w podrozdziale 3.1.4 w dalszej części rozdziału.

Bardzo ważną przestrzenią nazw jest **standardowa przestrzeń nazw** (ang. *standard namespace*) o nazwie `std`. Wynika to z faktu, że w standardowej przestrzeni nazw określone zostały identyfikatory wszystkich jednostek (typów danych, stałych, zmiennych itd.) zadeklarowanych lub zdefiniowanych w **standardowej bibliotece C++** (ang. *C++ Standard Library*).



UWAGA

Więcej informacji dotyczących standardowej biblioteki C++ można uzyskać w artykule [C++ Standard Library headers](#) zawartym w dokumentacji języka C++ na stronie [cppreference.com](#).

3.1.3. Standardowe wejście/wyjście

Określenie **standardowe wejście/wyjście** (ang. *standard input/output*) oznacza wprowadzanie danych wejściowych do programu i wyświetlanie informacji pochodzących z programu (np. komunikatów, wartości danych, wyników) na standardowych urządzeniach wejścia/wyjścia. Standardowym urządzeniem wejścia jest klawiatura, a standardowym urządzeniem wyjścia jest ekran monitora (konsola).

W języku C++ są dostępne dwa podejścia umożliwiające wykonywanie operacji związanych ze standardowym wejściem/wyjściem:

- wykorzystanie tzw. **strumieni wejścia/wyjścia** (ang. *input/output streams*),
- wykorzystanie standardowych funkcji wejścia/wyjścia odziedziczonych po języku C.



UWAGA

W podręczniku omówione zostały jedynie operacje wejścia/wyjścia oparte na użyciu strumieni, czyli tzw. podejście obiektowe. Z podejściem odziedziczonym po języku C, opartym na wykorzystaniu predefiniowanych funkcji wejścia/wyjścia, można się zapoznać w artykule [C-style file input/output](#) zawartym w dokumentacji języka C++ na stronie [cppreference.com](#).

Strumień danych stanowi sekwencję (ciąg) danych — bajtów. Jeżeli przepływ danych (ang. *data flow*) — ciągu bajtów — odbywa się w kierunku od urządzenia (np. klawiatury) do pamięci operacyjnej komputera, taki strumień danych nazywa się **strumieniem wejściowym** (ang. *input stream*). Jeżeli przepływ bajtów odbywa się w kierunku odwrotnym, tj. z pamięci głównej do urządzenia (np. do konsoli na ekranie monitora), to taki strumień nazywa się **strumieniem wyjściowym** (ang. *output stream*).

Standardowy strumień wejściowy (ang. *standard input stream*), czyli strumień wejściowy skojarzony ze standardowym urządzeniem wejścia (tj. klawiaturą), jest reprezentowany przez strumień o nazwie **cin**. **Standardowy strumień wyjściowy** (ang. *standard output stream*), skojarzony z konsolą na ekranie monitora, nazywa się **cout**.



UWAGA

Strumienie **cin** i **cout** są w rzeczywistości obiektami. Obiekt **cin** jest instancją klasy **istream**, a obiekt **cout** — instancją klasy **ostream**. Klasy i obiekty zostały omówione szczegółowo w dalszej części podręcznika — w rozdziale 11. Tutaj strumienie — obiekty **cin** i **cout** — zostaną przedstawione jedynie jako „czarne skrzynki”, które pozwalają na realizację podstawowych operacji wejścia/wyjścia.

Definicje strumieni `cin` i `cout` są zawarte w **pliku nagłówkowym** (ang. *header file*) o nazwie `iostream`, należącym do standardowej biblioteki wejścia/wyjścia. Dlatego też wykorzystanie w programie strumieni `cin` i `cout` wymaga wcześniejszego zadeklarowania użycia biblioteki `iostream` za pomocą **dyrektywy preprocessora** (ang. *preprocessor directive*) `#include <iostream>`.

UWAGA

Preprocesor i najważniejsze dyrektywy preprocessora, jak również pliki nagłówkowe zostały omówione w rozdziale 9. podręcznika.

W połączeniu ze standardowymi strumieniami wejścia/wyjścia używane są specjalne operatory pozwalające na kontrolowany, sekwencyjny przepływ danych (bajtów) w odpowiednim kierunku. W szczególności ze standardowym strumieniem wejściowym `cin` skojarzony jest **operator wyodrębnienia** (ang. *extraction operator*) `>>`, a ze strumieniem wyjściowym `cout` — **operator wstawiania** (ang. *insertion operator*) `<<`.

Przykład 3.20

```
int bok1, bok2;
// Wyświetlenie komunikatu, przejście do następnego wiersza i opróżnienie strumienia:
std::cout << "Obliczenie pola i obwodu prostokąta" << std::endl;
// Wstawienie pustego wiersza:
std::cout << std::endl;
// Wyświetlenie komunikatu pomocniczego:
std::cout << "Podaj długość pierwszego boku: ";
// Wprowadzenie z klawiatury długości pierwszego boku i zapamiętanie tej wartości w zmiennej bok1:
std::cin >> bok1;
// Wyświetlenie komunikatu pomocniczego:
std::cout << "Podaj długość drugiego boku: ";
// Wprowadzenie z klawiatury długości drugiego boku i zapamiętanie tej wartości w zmiennej bok2:
std::cin >> bok2;

// Obliczenie pola i obwodu prostokąta i przypisanie uzyskanych wyników do zmiennych pole i obwod:
int pole = bok1 * bok2;
int obwod = 2 * bok1 + 2 * bok2;

// Prezentacja wyników w konsoli na ekranie monitora:
std::cout << std::endl << "Wyniki:" << std::endl;
std::cout << "Pole wynosi " << pole << std::endl;
std::cout << "Obwód wynosi " << obwod << std::endl;
```

We fragmencie przedstawionego kodu obliczane są pole i obwód prostokąta. Dane wejściowe — długości boków prostokąta — są wprowadzane ze standardowego urządzenia wejściowego, tj. z klawiatury. Wyniki — obliczone wartości pola i obwodu — są wyświetlane w konsoli.

na ekranie monitora. Operacje wejścia/wyjścia są realizowane przy użyciu standardowych strumieni: wejściowego `cin` i wyjściowego `cout`. Nazwy wspomnianych strumieni zostały zdefiniowane w standardowej przestrzeni nazw `std`, stąd w programie konieczne jest podawanie ich pełnych, kwalifikowanych nazw: `std::cin` i `std::cout`. Wstawienie nowego wiersza na ekranie oraz opróżnienie strumienia uzyskuje się za pomocą obiektu o nazwie `endl`, którego identyfikator również został określony w przestrzeni nazw `std`. Jeśli przed programem głównym znajdowałaby się deklaracja użycia standardowej przestrzeni nazw `std::using namespace std;`, w programie głównym można byłoby używać tylko nazw samych strumieni, tj. `cin` i `cout`, bez potrzeby informowania kompilatora, gdzie — w jakiej przestrzeni nazw — zostały określone nazwy tych strumieni.

Ćwiczenie 3.20

Napisz fragment kodu programu zawierający następujące elementy:

- deklarację zmiennych `dlugosc`, `szerokosc` i `wysokosc` typu całkowitego `int`, reprezentujących parametry prostopadłościanu,
- instrukcje pozwalające wprowadzać wartości zmiennych — danych wejściowych — `dlugosc`, `szerokosc` i `wysokosc` z klawiatury,
- instrukcje pozwalające na obliczenie objętości, pola powierzchni bocznej oraz łącznej długości wszystkich krawędzi prostopadłościanu,
- instrukcje umożliwiające wyświetlenie obliczonych wartości objętości, pola i długości krawędzi na ekranie monitora.

3.1.4. Struktura programu

Strukturę programu komputerowego w języku C++ zilustrowano przez uzupełnienie kodu z przykładu 3.20 o niezbędne elementy. Kompletny program przedstawiono w przykładzie 3.21.

Przykład 3.21

```
// Dyrektywa preprocesora:
#include <iostream>

// Deklaracja użycia standardowej przestrzeni nazw:
using namespace std;

// Program główny:
int main() {
    // Instrukcje programu:
    int bok1, bok2;
    cout << "Obliczenie pola i obwodu prostokąta" << endl;
    cout << endl;
    cout << "Podaj długość pierwszego boku: ";
    cin >> bok1;
    cout << "Podaj długość drugiego boku: ";
```

```

    cin >> bok2;

    int pole = bok1 * bok2;
    int obwod = 2 * bok1 + 2 * bok2;

    cout << endl << "Wyniki:" << endl;
    cout << "Pole wynosi " << pole << endl;
    cout << "Obwód wynosi " << obwod << endl;

    // Przesłanie do systemu operacyjnego statusu wyjścia z programu:
    return 0;
}

```

Pierwsza instrukcja w programie to dyrektywa preprocesora. Zasadniczo dyrektyw preprocesora może być więcej. Są one interpretowane przez preprocesor jeszcze przed rozpoczęciem procesu kompilacji. Wykorzystana w programie dyrektywa `#include <iostream>` powoduje dołączenie do niego zasobów zbioru nagłówkowego o nazwie `iostream`, w którym znajdują się definicje standardowych strumieni wejścia/wyjścia.

Drugą instrukcją w programie jest deklaracja użycia identyfikatorów (nazw) zdefiniowanych w standardowej przestrzeni nazw `std`. Dzięki temu można używać nazw strumieni wejścia `cin` i wyjścia `cout` bez określania nazwy przestrzeni, `std`, z której te strumienie pochodzą.

Następnym blokiem jest **program główny** (ang. *main program*). W języku C++ program główny stanowi definicja funkcji `main()`. Jest ona wywoływana i wykonywana automatycznie podczas uruchamiania programu. W ogólności wszystkie programy w języku C++ rozpoczynają się od wykonania funkcji `main()`, niezależnie od jej położenia w kodzie źródłowym.

UWAGA

Budowa i praktyczne wykorzystanie funkcji w języku C++ zostały omówione w dalszej części podręcznika — w rozdziale 8.

W funkcji `main()` zawarty jest zestaw instrukcji składających się na program, ujętych w nawiasy klamrowe (ang. *braces*, `{}`). Instrukcje te są wykonywane sekwencyjnie jedna po drugiej, aż sterowanie natrafi na instrukcję `return`. Instrukcja `return` zawarta w funkcji `main()` jest wykonywana w programie jako ostatnia. Wartość wyrażenia po słowie kluczowym `return` (tutaj stała `0`, co jest równoważne statusowi `EXIT_SUCCESS`) jest zwracana do systemu operacyjnego. Jeżeli funkcja `main()` przekaże do systemu operacyjnego wartość `0` (jak tutaj), będzie to oznaczać, że wykonanie programu zakończyło się sukcesem.

Ćwiczenie 3.21

Zmodyfikuj kod źródłowy w przykładzie 3.21 — zamiast pola i obwodu prostokąta oblicz pole i obwód kwadratu.

Instrukcja złożona

W programowaniu często się zdarza, że trzeba wykonać ciąg (zestaw) instrukcji, ale składnia języka dopuszcza wykonanie w tych warunkach tylko pojedynczej instrukcji. W takim przypadku należy zastosować **instrukcję złożoną** (ang. *compound statement*).

Instrukcją złożoną nazywa się zestaw instrukcji ujętych w nawiasy klamrowe, {}. Ogólna postać tej instrukcji jest następująca:

```
{
    instrukcja_1;
    instrukcja_2;
    instrukcja_3;
    ...
}
```

gdzie `instrukcja_1`, `instrukcja_2` oraz `instrukcja_3` są dowolnymi instrukcjami języka C++.

Instrukcja złożona jest traktowana przez kompilator jako pojedyncza instrukcja, niezależnie od liczby instrukcji składowych, które są w niej zawarte. Instrukcja złożona często jest nazywana **instrukcją bloku** (ang. *block statement*) albo po prostu **blokiem kodu** (ang. *code block*).

Przykładem zastosowania instrukcji złożonej jest blok kodu zawarty w funkcji `main()` w przykładzie 3.21.

3.2. Podejmowanie decyzji w programie

W programie komputerowym często jest konieczne podejmowanie decyzji (ang. *decision making*). Na przykład, jeśli wprowadzona z klawiatury długość promienia koła jest dodatnia, należy obliczyć jego pole powierzchni i obwód oraz wyświetlić wyniki na ekranie monitora. W przeciwnym razie (długość promienia koła jest niedodatnia) należy wyświetlić komunikat o błędzie.

W ogólności proces podejmowania decyzji (ang. *decision making process*) w programie polega na ustaleniu kolejności (sekwencji) wykonywania poszczególnych instrukcji na podstawie określonych, zdefiniowanych warunków.

W języku C++ obsługa procesu podejmowania decyzji może zostać zrealizowana za pomocą:

- **instrukcji warunkowych** (ang. *conditional statements*) `if` oraz `if-else`,
- **instrukcji wyboru** (ang. *selection statement*) `switch`,
- **operatora warunkowego** (ang. *ternary operator, conditional operator*).

3.2.1. Instrukcja warunkowa if

Ogólna postać instrukcji warunkowej `if` jest następująca:

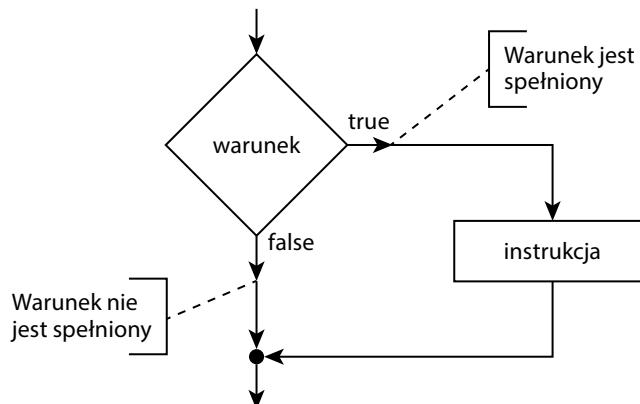
```
if ( warunek ) instrukcja;
```

gdzie:

- warunek jest wyrażeniem logicznym typu `bool`, którego wynik przyjmuje wartość logiczną albo `true`, albo `false`,
- instrukcja jest dowolną, ale pojedynczą instrukcją języka C++.

Jeśli w programie jest konieczne warunkowe wykonanie zestawu instrukcji, a nie pojedynczej instrukcji, należy zastosować instrukcję złożoną, o której była mowa wcześniej, w podrozdziale 3.1.

Działanie instrukcji warunkowej `if` zilustrowano na rysunku 3.1 za pomocą schematu blokowego.



Rysunek 3.1. Ilustracja działania instrukcji warunkowej if

Jeżeli warunek jest spełniony (`true`), wykonywana jest `instrukcja`. W przeciwnym razie (`false`) następuje przejście do kolejnej instrukcji w programie, bezpośrednio po instrukcji `if`.

Praktyczne wykorzystanie instrukcji warunkowej `if` zaprezentowano w przykładzie 3.21, w którym dla zadanej liczby całkowitej wyznaczana jest jej wartość bezwzględna.

Przykład 3.22

```
#include <iostream>
using namespace std;
int main() {
    int liczba = -1;
    cout << "Liczba = " << liczba << endl;
```

```

int wb;
// Zastosowanie instrukcji warunkowych:
if (liczba >= 0) wb = liczba;
if (liczba < 0) wb = -liczba;

cout << "Wartość bezwzględna: " << wb << endl;

return 0;
}

```

Zadaną liczbę reprezentuje w programie zmienna typu `int` o nazwie `liczba`. Jest ona inicjowana w programie wartością `-1`. Wartość bezwzględna `liczby` zaś jest reprezentowana przez zmienną `wb`.

W pierwszej instrukcji warunkowej `if` następuje sprawdzenie warunku `liczba >= 0`. Jeśli warunek ten jest spełniony (`true`), wykonywana jest pojedyncza instrukcja `wb = liczba`. Jeśli natomiast zadany warunek nie jest spełniony (`false`), to sterowanie przechodzi do następnej instrukcji w programie, którą jest następna instrukcja warunkowa `if`. Jeśli w drugiej instrukcji `if` spełniony jest warunek `liczba < 0`, wykonywana jest instrukcja `wb = -liczba`. Kolejną operacją wykonywaną w programie jest wyświetlenie wyznaczonej wartości bezwzględnej `wb` na ekranie monitora w konsoli.

Obie zastosowane w przykładzie instrukcje `if` są od siebie niezależne składniowo (syntaktycznie), ale zależne znaczeniowo (logicznie, semantycznie). Niezależność składniowa użytych instrukcji `if` wynika z tego, że usunięcie jednej z nich nie wpływa na wynik komilacji programu — komilacja ta nadal kończy się sukcesem. Zależność semantyczna omawianych instrukcji `if` zaś wynika ze znaczenia logicznego wykorzystywanych w nich warunków. Warunki te wzajemnie się uzupełniają — są względem siebie komplementarne.

Ćwiczenie 3.22

Napisz program pozwalający na obliczenie wartości funkcji danej wzorem:

$$y = f(x) = \begin{cases} -10 & \Leftrightarrow x < 0 \\ 100 & \Leftrightarrow x \geq 0 \end{cases}$$

Dane wejściowe (wartość argumentu `x`) mają być wprowadzane z klawiatury, a wynik (wartość `y`) niech będzie wyświetlany na ekranie monitora. Zastosuj dwie równoległe, niezależne od siebie instrukcje warunkowe `if`.

3.2.2. Instrukcja warunkowa `if-else`

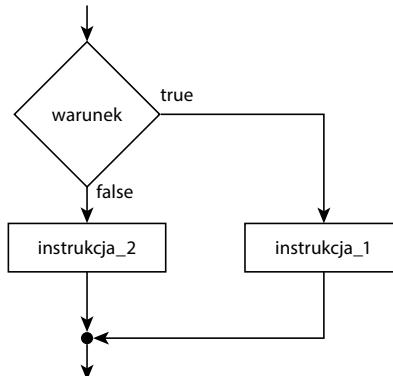
Ogólna postać instrukcji warunkowej `if-else` jest następująca:

```
if ( warunek ) instrukcja_1; else instrukcja_2;
```

gdzie:

- warunek to wyrażenie logiczne typu `bool`,
- `instrukcja_1`, `instrukcja_2` to dowolne, ale pojedyncze instrukcje języka C++, które w ogólności mogą być instrukcjami złożonymi.

Działanie instrukcji warunkowej `if-else` zilustrowano na rysunku 3.2 za pomocą schematu blokowego.



Rysunek 3.2. Ilustracja działania instrukcji warunkowej `if-else`

Jeżeli warunek jest spełniony (`true`), wykonywana jest `instrukcja_1`; w przeciwnym razie (`false`) wykonywana jest `instrukcja_2`.

Podobnie jak w przypadku instrukcji warunkowej `if`, zamiast pojedynczych instrukcji `instrukcja_1` oraz `instrukcja_2` można stosować instrukcje złożone (bloki kodu).

Przykład 3.23

```

#include <iostream>
using namespace std;

int main() {
    int liczba = -1;
    cout << "Liczba = " << liczba << endl;

    int wb;
    // Wykorzystanie instrukcji warunkowej if-else:
    if (liczba >= 0) {wb = liczba;}
    else {wb = -liczba;}

    cout << "Wartość bezwzględna: " << wb << endl;

    return 0;
}
  
```

Podobnie jak w przykładzie 3.22, również tutaj wyznaczana jest wartość bezwzględna `wb` z danej liczby całkowitej `liczba`. Zmienna `liczba` jest inicjowana w programie. Jeśli warunek `liczba >= 0` jest spełniony (`true`), wykonywana jest instrukcja `wb = liczba`. Jeśli warunek nie jest spełniony (`false`), wykonywana jest instrukcja `wb = -liczba`.

Należy zwrócić uwagę na to, że dopełnieniem logicznym zadanego, sformułowanego jawnie warunku `liczba >= 0` jest niejawny warunek `liczba < 0`. Instrukcja występująca po słowie kluczowym `else`: `wb = -liczba` jest wykonywana właśnie w przypadku, gdy jest spełnione dopełnienie jawnego warunku po słowie kluczowym `if`.

Ćwiczenie 3.23

Napisz program pozwalający na obliczenie wartości funkcji danej wzorem:

$$y = f(x) = \begin{cases} -10 & \Leftrightarrow x < 0 \\ 100 & \Leftrightarrow x \geq 0 \end{cases}$$

Dane wejściowe (wartość argumentu `x`) mają być wprowadzane z klawiatury, a wynik (wartość `y`) wyświetlane na ekranie monitora. Zastosuj instrukcję warunkową `if-else`.

3.2.3. Zagnieżdżone instrukcje warunkowe

W praktyce programowania często można się spotkać z sytuacją, w której określone instrukcje warunkowe są zawarte w innych instrukcjach warunkowych. W ten sposób uzyskuje się **zagnieżdżone instrukcje warunkowe** (ang. *nested conditional statements*).

Można zagnieździć zarówno instrukcje warunkowe `if`, jak i `if-else` — w zależności od potrzeb i ewentualnie dla wygody. Należy jednak pamiętać o jednej z podstawowych zasad programowania: preferować możliwie proste i czytelne oraz konwencjonalne sposoby kodowania.

UWAGA

Struktura zagnieżdżonych instrukcji warunkowych jest czasem nazywana drabinką instrukcji warunkowych.

Zagnieżdżone instrukcje warunkowe if

Ogólna postać zagnieżdżonych instrukcji warunkowych `if` jest następująca:

```
if ( warunek_1 ) {
    zestaw_instrukcji_1;
    if ( warunek_2 ) {
        zestaw_instrukcji_2;
        if ( warunek_3 ) {
```

```

        zestaw_instrukcji_3;
    }
}
}
}

```

gdzie:

- warunek_1, warunek_2, warunek_3 — wyrażenia logiczne typu bool,
- zestaw_instrukcji_1, zestaw_instrukcji_2, zestaw_instrukcji_3 — zestawy dowolnych instrukcji języka C++.

Zasadniczo liczba instrukcji składowych if w strukturze zagnieżdżonych instrukcji warunkowych if jest dowolna.

Działanie zagnieżdżonych instrukcji if można opisać za pomocą następującego algorytmu:

1. Jeżeli warunek_1 jest spełniony (true), to

- wykonywany jest zestaw_instrukcji_1 oraz
- jeżeli warunek_2 jest spełniony, to
 - » wykonywany jest zestaw_instrukcji_2 oraz
 - » jeżeli warunek_3 jest spełniony, to
 - wykonywany jest zestaw_instrukcji_3.

2. W przeciwnym razie, tj. jeśli warunek_1 nie jest spełniony (false), sterowanie przechodzi do następnej instrukcji w programie bezpośrednio po drabince instrukcji if.

Przykład 3.24

```

#include <iostream>
using namespace std;

int main() {
    int ocena = 5;

    if (ocena >= 2) {
        cout << "Uczeń zaliczył sprawdzian na ocenę pozytywną!" << endl;
        if (ocena >= 3) {
            cout << "Uczeń rozwiązuje zadania samodzielnie!" << endl;
            if (ocena >= 4) {
                cout << "Uczeń dobrze opanował materiał nauczania!" << endl;
                if (ocena >= 5) {
                    cout << "Uczeń jest prymusem!" << endl;
                }
            }
        }
    }

    return 0;
}

```

Zmienna całkowita ocena reprezentuje w programie ocenę uzyskaną przez ucznia ze sprawdzianu.

Jeżeli uczeń uzyskał ze sprawdzianu ocenę co najmniej 2 (czyli warunek `ocena >= 2` jest spełniony), na ekranie zostaje wyświetlony komunikat `Uczeń zaliczył sprawdzian na ocenę pozytywną!` oraz sprawdzony następny warunek, `ocena >= 3`. Jeśli również ten warunek jest spełniony, na ekranie pojawia się następny komunikat, `Uczeń rozwiązuje zadania samodzielnie!`, i sprawdzany jest kolejny warunek, `ocena >= 4`. Jeśli także ten warunek jest spełniony, wyświetlany jest komunikat `Uczeń dobrze opanował materiał nauczania!` oraz sprawdzany ostatni warunek, `ocena >= 5`. Jeśli ten ostatni warunek jest spełniony, na ekranie pojawia się komunikat `Uczeń jest prymusem!`.

Jeśli pierwszy (najogólniejszy) warunek (tj. `ocena >= 2`) nie jest spełniony, sterowanie przechodzi do instrukcji `return 0` i program kończy działanie.

UWAGA

Dobrą praktyką programowania jest unikanie zbyt mocnego zagnieżdżania instrukcji warunkowych, ponieważ kod źródłowy mógłby się wówczas stać nieczytelny.

Ćwiczenie 3.24

Napisz program pozwalający na obliczenie wartości funkcji $y = f(x)$ danej wzorem:

$$y = f(x) = \begin{cases} 1 & \Leftrightarrow x \in (0, 1) \\ 2 & \Leftrightarrow x \in (1, 2) \\ 3 & \Leftrightarrow x \in (2, \infty) \end{cases}$$

Wartości `x < 0` nie należą do dziedziny funkcji. Dane wejściowe (tj. wartość argumentu `x`) mają być wprowadzane z klawiatury, a wynik (wartość `y`) wyświetlany na ekranie monitora. Zastosuj zagnieżdżone instrukcje warunkowe `if`.

Zagnieżdżone instrukcje warunkowe if-else

Ogólna postać zagnieżdżonych instrukcji warunkowych `if-else` jest następująca:

```
if ( warunek_1 )
    instrukcja_1 / blok_kodu_1;
    else if ( warunek_2 )
        instrukcja_2 / blok_kodu_2;
        else instrukcja_3 / blok_kodu_3;
```

gdzie:

- warunek_1, warunek_2 — wyrażenia logiczne typu bool,
- instrukcja_1, instrukcja_2, instrukcja_3 — dowolne pojedyncze instrukcje C++, zamiast których można stosować bloki kodu (odpowiednio): blok_kodu_1, blok_kodu_2, blok_kodu_3.

Drabinka instrukcji if-else może się składać z dowolnej liczby zagnieżdżonych instrukcji if-else.

Działanie drabinki instrukcji if-else można opisać za pomocą algorytmu:

1. Jeżeli warunek_1 jest spełniony (true), wykonywana jest instrukcja_1 (lub blok_kodu_1).
2. W przeciwnym razie, tj. jeśli warunek_1 nie jest spełniony (false):
 - jeżeli warunek_2 jest spełniony (true), wykonywana jest instrukcja_2 (lub blok_kodu_2),
 - w przeciwnym razie, tj. jeśli warunek_2 nie jest spełniony (false), wykonywana jest instrukcja_3 (lub blok_kodu_3).

W praktyce zamiast drabinki zagnieżdżonych instrukcji if-else można stosować zestaw równoległych — niezależnych syntaktycznie (składniowo) — instrukcji if:

```
if ( warunek_1 ) instrukcja_1 / blok_kodu_1;
if ( warunek_2 ) instrukcja_2 / blok_kodu_2;
if ( warunek_3 ) instrukcja_3 / blok_kodu_3;
```

gdzie:

- warunek_1, warunek_2 są identyczne z tymi w drabinie instrukcji if-else,
- warunek_3 to dopełnienie do warunku_1 oraz warunku_2 razem wziętych (uwzględnionych wspólnie).

Poniżej za pomocą przykładu 3.25 zilustrowano różnice pomiędzy zastosowaniem drabinki (zagnieżdżonych) instrukcji if-else a użyciem zestawu równoległych instrukcji if w celu rozwiązania tego samego problemu — obliczenia wartości funkcji $y = f(x) = \text{signum}(x)$.

Przykład 3.25

Zastosowanie zagnieżdżonych instrukcji if-else:

```
if (x < 0) {y = -1;}
else if (x == 0) {y = 0;}
else {y = 1;}
```

Zastosowanie równoległych instrukcji if:

```
if (x < 0) {y = -1;}
if (x == 0) {y = 0;}
if (x > 0) {y = 1;}
```

Przedstawione przykłady kodu dobrze obrazują różnice pomiędzy użyciem zagnieżdżonych instrukcji `if-else` a wykorzystaniem równoległych — niezależnych składniowo — instrukcji `if`. W pierwszym przypadku (instrukcje `if-else`) zawsze sprawdzany jest co najmniej jeden warunek mniej niż w drugim. Ponadto w sytuacji, gdy jeden z warunków jest spełniony (`true`), kończy to dalsze przetwarzanie drabinki. Pozostałe warunki — czyli te, które są zawarte wewnętrznych instrukcjach `if-else` — nie są wtedy sprawdzane. To sprawia, że aplikacja jest lepiej zoptymalizowana, ale z drugiej strony kod jest mniej przejrzysty i czytelny. W drugim przypadku (instrukcje `if`) zawsze sprawdzane są wszystkie warunki (w każdej instrukcji `if`). Pogarsza to wydajność aplikacji, ale znacznie zwiększa przejrzystość i czytelność jej kodu źródłowego.

Bardziej złożony program ilustrujący praktyczne wykorzystanie zagnieżdżonych instrukcji warunkowych `if-else` zaprezentowano w przykładzie 3.26.

Ćwiczenie 3.25

Napisz program pozwalający na obliczenie wartości funkcji $y = f(x)$ danej wzorem:

$$y = f(x) = \begin{cases} 10 & \Leftrightarrow x \in (-\infty, 10) \\ 20 & \Leftrightarrow x \in (10, 20) \\ 30 & \Leftrightarrow x \in (20, \infty) \end{cases}$$

Wartość argumentu `x` ma być wprowadzana z klawiatury, a wynik $y = f(x)$ wyświetlany na ekranie monitora. Wykorzystaj zagnieżdżone instrukcje `if-else`.

Przykład 3.26

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

int main() {
    double a = 1;
    double b = 5;
    double c = 4;

    double delta = b * b - 4 * a * c;
    cout << "Delta wynosi: " << delta << endl;

    if (delta > 0) {
        double x1 = (-b - sqrt(delta))/(2 * a);
        double x2 = (-b + sqrt(delta))/(2 * a);
        cout << "Pierwszy pierwiastek = " << x1 << endl;
        cout << "Drugi pierwiastek = " << x2 << endl;
    }
}
```

```

    else if (delta == 0) {
        double x0 = -b /(2 * a);
        cout << "Pierwiastek = " << x0 << endl;
    }
    else {
        cout << "Równanie nie ma pierwiastków rzeczywistych" << endl;
    }

    return 0;
}

```

W programie wyznaczane są pierwiastki rzeczywiste (o ile istnieją) trójmianu kwadratowego $y = a \cdot x^2 + b \cdot x + c$. Wartości zmiennych a , b , c (które reprezentują współczynniki trójmianu) są inicjowane w programie. Zmienna `delta` odpowiada wyróżnikowi trójmianu Δ .

Jeśli $\Delta > 0$ (`delta > 0`), to trójmian ma dwa pierwiastki (miejsca zerowe). Wewnątrz bloku kodu obliczane są wartości pierwiastków (reprezentowanych przez zmienne $x1$ i $x2$), a następnie wartości te są wyświetlane na ekranie. Jeśli warunek `delta > 0` nie jest spełniony (`false`), następuje przejście do drugiej instrukcji warunkowej `if-else`, co wiąże się ze sprawdzeniem warunku `delta == 0`.

Jeśli warunek `delta == 0` jest spełniony, obliczana jest, a następnie wyświetlana wartość jedynczego pierwiastka $x0$. W przeciwnym razie — co stanowi spełnienie (`true`) niejawnego warunku dopełniającego do dwóch wcześniejszych warunków określonych jawnie: `delta > 0` oraz `delta == 0` — równanie nie ma pierwiastków rzeczywistych, o czym użytkownik programu zostaje poinformowany przez wyświetlenie odpowiedniego komunikatu na ekranie.

Ćwiczenie 3.26

Zmodyfikuj program zaprezentowany w przykładzie 3.26 — zamiast zagnieżdzonych instrukcji `if-else` wykorzystaj niezależne składniowo instrukcje `if`.

UWAGA

W przypadku ogólnym, zwłaszcza jeśli konieczne jest oprogramowanie wielu złożonych, wieloargumentowych warunków, z których część może „na siebie zachodzić”, a część może być „rozłączna”, jest bardzo prawdopodobne, że wykorzystanie zagnieżdzonych instrukcji `if-else` doprowadzi do niewidocznych i (lub) trudnych do wychwycenia błędów logicznych albo w ogóle będzie nieskuteczne. W takich skomplikowanych sytuacjach zaleca się stosowanie zestawu niezależnych od siebie, równoległych instrukcji `if`, pomimo pogorszenia wydajności aplikacji.

3.2.4. Instrukcja wyboru switch

Ogólna postać instrukcji wyboru switch jest następująca:

```
switch (wyrażenie_sterujące) {
    case etykieta_1: zestaw_instrukcji_1;
                      [break;]
    case etykieta_2: zestaw_instrukcji_2;
                      [break;]
    case etykieta_3: zestaw_instrukcji_3;
                      [break;]

    ...
    [default: zestaw_instrukcji_domyślnych;]
}
```

gdzie:

- wyrażenie_sterujące to wyrażenie (np. zmienna) typu całkowitego lub typu wyliczeniowego (ang. *enumerated type*), którego wynik zostanie porównany z wartościami etykiet występujących po słowach kluczowych `case`,
- etykieta_1, etykieta_2, etykieta_3 — wyrażenia stałe (ang. *constant expressions*), np. wartości określone za pomocą literalów, takie jak "1" oraz 1 (typ wyniku tych wyrażeń musi być zgodny z typem wyniku wyrażenia_sterującego),
- `zestaw_instrukcji_1, zestaw_instrukcji_2, zestaw_instrukcji_3` — zestawy dowolnych instrukcji języka C++,
- `break` — instrukcja opcjonalna, której wykonanie powoduje zakończenie działania instrukcji `switch` jako instrukcji nadzędnej względem instrukcji `break`,
- `default` — „etykieta domyślna” (opcjonalny składnik instrukcji `switch`),
- `zestaw_instrukcji_domyślnych` — zestaw dowolnych instrukcji C++.

Działanie instrukcji wyboru `switch` można opisać w sposób ogólny za pomocą algorytmu:

1. Najpierw obliczana jest wartość `wyrażenia_sterującego`. Następnie pośród wszystkich stałych wyboru, tj. etykiet występujących po słowach kluczowych `case`, wyszukiwana jest taka, której wartość jest równa wynikowi `wyrażenia_sterującego`:
 - jeżeli etykieta o wartości równej wynikowi `wyrażenia_sterującego` zostanie odnaleziona, to
 - » wykonywany jest `zestaw_instrukcji` odpowiadający tej etykiecie,
 - » następnie wykonywana jest instrukcja `break` (o ile istnieje), co kończy działanie instrukcji `switch`,
 - » w przypadku gdy w danej opcji instrukcji `switch` instrukcja `break` nie występuje, wykonywany jest następny `zestaw_instrukcji`, zapisany w kodzie źródłowym poniżej:

- jeżeli etykieta o wartości równej obliczonemu wynikowi `wyrażenia_sterującego` nie zostanie odnaleziona, wykonywany jest `zestaw_instrukcji_domyślnych`, występujący po słowie kluczowym `default` (etykiety domyślnej) — o ile istnieje,
- jeśli etykiety domyślnej nie określono, wykonywana jest kolejna instrukcja w programie zawarta w kodzie źródłowym poniżej instrukcji `switch`.

Przykład 3.27

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

int main() {
    int zmiennaSterujaca;

    zmiennaSterujaca = 3;
    cout << "Wartość zmiennej sterującej: " << zmiennaSterujaca << endl;

    switch(zmiennaSterujaca) {
        case 1: cout << "Wybrano opcję 1";
                  break;
        case 2: cout << "Wybrano opcję 2";
                  break;
        case 3:
        case 4: cout << "Wybrano opcje 3 i 4";
                  break;
        case 5: cout << "Wybrano opcję 5";
                  break;

        default: cout << "Nie wybrano żadnej opcji";
    }
    cout << endl;
    return 0;
}
```

W instrukcji `switch` przedstawionej w programie rolę wyrażenia sterującego odgrywa zmienna o nazwie `zmiennaSterujaca`. Etykiety wyboru przyjmują wartości liczb całkowitych z przedziału od 1 do 5. Jeżeli wartość zmiennej `zmiennaSterujaca` jest równa np. 3 (jak w programie), na ekranie zostaje wyświetlony stosowny komunikat: "Wybrano opcje 3 i 4". Następnie wykonywana jest instrukcja `break`, co kończy działanie instrukcji `switch`.

Jeśli wartość zmiennej `zmiennaSterujaca` jest różna od 1, 2, 3, 4 oraz 5, wykonywana jest instrukcja domyślna, która wyświetla komunikat: "Nie wybrano żadnej opcji".

Ćwiczenie 3.27

Zmodyfikuj program zawarty w przykładzie 3.27 — zamiast instrukcji wyboru `switch` wykorzystaj zestaw niezależnych (równoległych) instrukcji warunkowych `if`.

W praktyce instrukcja `switch` jest często stosowana np. w celu sterowania menu aplikacji. Dzięki użyciu tej instrukcji kod jest bardziej przejrzysty i czytelny, a jakość aplikacji lepsza od tej, jaką można osiągnąć przy wykorzystaniu równoległych (niezależnych) instrukcji `if` — biorąc pod uwagę kryterium optymalizacji kodu. Reasumując, można powiedzieć, że zastosowanie instrukcji `switch` łączy w sobie zalety użycia drabinki `if-else` (pod względem optymalizacji kodu) oraz równoległych instrukcji `if` (pod względem czytelności i przejrzystości kodu).

UWAGA

Występowanie warunku jako integralnej części instrukcji warunkowych `if`, `if-else` oraz instrukcji wyboru `switch` powoduje, że instrukcje te są zaliczane do struktur sterujących (ang. *control structures*) przebiegiem działania programu.

3.2.5. Operator warunkowy

W prostych sytuacjach zamiast instrukcji `if-else` można wykorzystać **operator warunkowy** (ang. *conditional operator, ternary operator*).

Ogólna postać operatora warunkowego jest następująca:

```
zmienna = ( warunek ) ? wyrażenie_1 : wyrażenie_2;
```

gdzie:

- warunek jest wyrażeniem logicznym typu `bool`,
- wyrażenie_1, wyrażenie_2 należą do tego samego typu, co `zmienna`, np. `double`, `int`.

Działanie operatora warunkowego można opisać następująco:

1. Jeżeli warunek jest spełniony, wykonywana jest instrukcja `zmienna = wyrażenie_1`.
2. W przeciwnym razie (`false`) wykonywana jest instrukcja `zmienna = wyrażenie_2`.

Przykład 3.28

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

int main() {
    int liczba = -1;
    cout << "Liczba = " << liczba << endl;
```

```

int wb = (liczba >= 0) ? liczba : -liczba;
cout << "Wartość bezwzględna: " << wb << endl;

return 0;
}

```

W programie wyznaczana jest wartość bezwzględna `wb` z zadanej (zainicjowanej w programie) zmiennej o nazwie `liczba`. Jeśli warunek `liczba >= 0` jest spełniony (`true`), wykonywana jest (niejawnie) instrukcja przypisania `wb = liczba`. W przeciwnym razie — jeśli warunek `liczba >= 0` nie jest spełniony — wykonywana jest niejawnie instrukcja `wb = -liczba`.

Ćwiczenie 3.28

Zmodyfikuj program z przykładu 3.28 — zamiast operatora warunkowego wykorzystaj instrukcję `if-else`.

3.3. Pętle programowe

Proces sterowania działaniem programu komputerowego może również polegać na powtarzaniu określonej operacji (lub grupy operacji) w zależności od tego, czy dany warunek jest spełniony, czy nie. Założymy, że przejście na następny poziom w grze komputerowej jest możliwe tylko po zdobyciu przez gracza określonej liczby punktów (np. 100 punktów) na bieżącym poziomie. Punkty zdobywane przez gracza na danym poziomie są sumowane i zapisywane np. w zmiennej `suma_punktow`. Tym samym gracz jest zmuszony kontynuować grę na bieżącym poziomie i zdobywać kolejne punkty dopóty, dopóki spełniony będzie warunek `suma_punktow < 100`. Jeśli zostanie spełniony warunek dopełniający, tj. `suma_punktow >= 100`, gracz przechodzi na następny poziom gry.

Inne uwarunkowania w programowaniu wynikają z sytuacji, w których należy powtórzyć określone czynności (operacje) konkretną, zadaną z góry liczbę razy. Przyjmijmy, że realizowanym zadaniem jest obliczenie sumy (a następnie średniej arytmetycznej) 10 liczb o wartościach wprowadzonych z klawiatury. Zatem należy wprowadzić dokładnie 10 liczb, czyli czynność „wprowadzenie liczby” należy powtórzyć 10 razy.

Język C++, podobnie jak inne języki programowania, dostarcza niezbędnych narzędzi — instrukcji pętli (ang. *loops*) programowych, które umożliwiają wielokrotne powtarzanie określonej instrukcji (lub zestawu instrukcji):

- dopóki zadany warunek jest spełniony (pierwszy przypadek omówiony wcześniej),
- określoną, zadaną liczbę razy (drugi przypadek omówiony wcześniej).

Integralną część instrukcji pętli omawianych w tym rozdziale stanowi warunek, czyli wyrażenie logiczne typu `bool`, które może dać w rezultacie albo wartość logiczną `true`, albo `false`. Występowanie warunku jako integralnej części instrukcji pętli powoduje, że rów-

nież pętle — podobnie jak instrukcje warunkowe — są zaliczane do struktur sterujących (ang. *control structures*) przebiegiem działania programu.

W języku C++ programista ma do dyspozycji trzy rodzaje pętli:

- `while`,
- `do-while`,
- `for`.

UWAGA

W wersji C++11 wprowadzono nowy rodzaj pętli `for`, mianowicie pętlę `for` „opartą na zakresie” (ang. *range-based for loop*), nazywaną potocznie pętlą `foreach`. Ta odmiana pętli `for` została omówiona w dalszej części podręcznika — w rozdziale 5., dotyczącym tablic i wektorów.

3.3.1. Pętla while

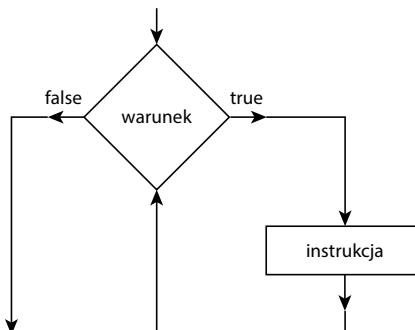
Pętla `while` stanowi pierwszą odmianę pętli w języku C++. Występuje również w innych językach programowania, np. Java, C#, a także JavaScript i PHP. Ogólna postać pętli `while` jest następująca:

```
while ( warunek ) instrukcja;
```

gdzie:

- warunek jest wyrażeniem logicznym typu `bool`, którego wynik przyjmuje wartość logiczną albo `true`, albo `false`,
- instrukcja to dowolna instrukcja języka C++, np. instrukcja złożona (ang. *compound statement*), zawierająca zestaw dowolnych instrukcji składowych.

Działanie pętli `while` zilustrowano na rysunku 3.3 za pomocą schematu blokowego.



Rysunek 3.3. Schemat blokowy działania pętli while

Jak wynika z przedstawionego schematu blokowego, instrukcja jest wykonywana, dopóki warunek jest spełniony (`true`). Przy tym nie jest z góry określone, ile razy operacja ta zostanie powtórzona. Jeśli warunek w pętli `while` nie jest spełniony (`false`), następuje przejście do kolejnej instrukcji w programie.

Pętla `while` jest nazywana **pętlą z warunkiem na początku**. Jeżeli już na samym początku (czyli podczas sprawdzenia wartości warunku po raz pierwszy) warunek nie jest spełniony (`false`), instrukcja zawarta wewnątrz pętli nie zostanie wykonana ani razu.

Przykład 3.29

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

int main() {
    int liczbaPocz = 1;
    int liczbaKonc = 3;

    int liczba = liczbaPocz, suma = 0;
    while (liczba <= liczbaKonc) {
        suma += liczba;
        liczba++;
    }

    cout << "Suma liczb wynosi: " << suma << endl;

    return 0;
}
```

W programie obliczana jest suma kolejnych liczb całkowitych, począwszy od zadanej liczby początkowej (zmienna `liczbaPocz`), a skończywszy na zadanej liczbie końcowej (zmienna `liczbaKonc`). Zmienne `liczbaPocz` i `liczbaKonc` są inicjowane w programie.

Działanie pętli jest kontrolowane za pomocą **zmiennej sterującej** (ang. *control variable*) o nazwie `liczba`. Dopóki spełniony jest warunek `liczba <= liczbaKonc`, wykonywana jest instrukcja `suma += liczba`, czyli do bieżącej wartości sumy dodawana jest bieżąca wartość `liczby`. Wynik tej operacji jest pamiętany w zmiennej `suma`. W bieżącej **iteracji** (ang. *iteration*) — czyli powtóżeniu instrukcji zawartych w pętli — oprócz instrukcji `suma += liczba` wykonywana jest instrukcja `liczba++`, czyli bieżąca wartość `liczby` (jako zmiennej sterującej pętli) jest zwiększana o 1. Na końcu wynik obliczeń jest prezentowany na ekranie.

Ćwiczenie 3.29

Z wykorzystaniem programu zawartego w przykładzie 3.29 napisz program umożliwiający obliczenie sumy liczb całkowitych z przedziału domkniętego $<1, 100>$. W obliczeniach uwzględnij wyłącznie liczby nieparzyste. Wynik należy zaprezentować na ekranie monitora. Do obliczeń wykorzystaj pętlę `while`.

3.3.2. Pętla do-while

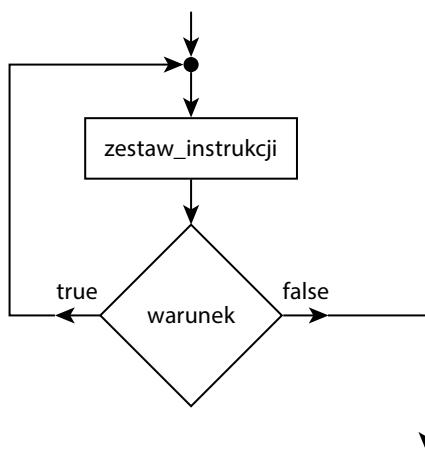
Podobnie jak pętla while, pętla do-while występuje w wielu językach programowania, np. Java, C#, JavaScript, PHP. Postać ogólna pętli do-while jest następująca:

```
do {
    zestaw_instrukcji;
}
while ( warunek );
```

gdzie:

- warunek jest wyrażeniem logicznym typu bool,
- zestaw_instrukcji to zestaw dowolnych instrukcji języka C++.

Działanie pętli do-while zilustrowano na rysunku 3.4 za pomocą schematu blokowego.



Rysunek 3.4. Schemat blokowy działania pętli do-while

Z przedstawionego schematu blokowego wynika, że zestaw_instrukcji jest wykonywany, dopóki warunek jest spełniony (true). Podobnie jak dla pętli while, nie jest z góry określone, ile razy zestaw_instrukcji zostanie wykonany.

W przeciwnym razie — jeśli warunek nie jest spełniony (false) — następuje przejście do instrukcji występującej w kodzie programu bezpośrednio po instrukcji pętli.

Pętla do-while jest nazywana **pętlą z warunkiem na końcu**. Oznacza to, że zestaw_instrukcji zostanie wykonany przynajmniej raz, nawet jeśli od początku warunek nie jest spełniony.

Przykład 3.30

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

int main() {
    int liczbaPocz = 1;
    int liczbaKonc = 3;

    int liczba = liczbaPocz, suma = 0;
    do {
        suma += liczba;
        liczba++;
    }
    while (liczba <= liczbaKonc);

    cout << "Suma liczb wynosi: " << suma << endl;

    return 0;
}
```

W programie — podobnie jak w przykładzie 3.29 — obliczana jest suma kolejnych liczb całkowitych, począwszy od zadanej liczby początkowej (`liczbaPocz`), a skończywszy na zadanej liczbie końcowej (`liczbaKonc`). Jednakże zamiast pętli `while` zastosowano tutaj pętlę `do-while`, czyli pętlę z warunkiem na końcu. Oznaczenia zmiennych w programie są identyczne z oznaczeniami w programie zawartym w przykładzie 3.29, a zmienne są tak samo interpretowane. Pętla `do-while` kończy swoje działanie, jeśli warunek `liczba <= liczbaKonc` nie jest spełniony (`false`).

Ćwiczenie 3.30

Z wykorzystaniem programu zawartego w przykładzie 3.30 napisz program umożliwiający obliczenie sumy liczb całkowitych z przedziału $<1, 100>$. W obliczeniach uwzględnij wyłącznie liczby całkowite parzyste. Wynik należy wyświetlić na ekranie monitora. Do obliczeń użyj pętli `do-while`.

3.3.3. Pętla for

Pętla `for` jest trzecią odmianą pętli w języku C++. Pętlę tę można również spotkać w kodach źródłowych napisanych w innych językach programowania, np. Java, C#, JavaScript, PHP. Postać ogólna pętli `for` jest następująca:

```
for ( wyrażenie_inicjujące;
      wyrażenie_warunkowe;
      wyrażenie_modyfikujące ) {
    zestaw_instrukcji;
}
```

gdzie:

- wyrażenie_inicjujące to wyrażenie wykonywane raz, na samym początku — przed wykonaniem zestawu_instrukcji — zwykle w celu zainicjowania zmiennej sterującej pętli (oraz ewentualnie innych zmiennych),
- wyrażenie_warunkowe to wyrażenie logiczne (warunek), które określa, czy zestaw_instrukcji zawartych wewnątrz pętli zostanie wykonany — jeżeli warunek jest spełniony (true), czy nie — jeżeli warunek nie jest spełniony (false),
- wyrażenie_modyfikujące to wyrażenie, które jest wykonywane każdorazowo po wykonaniu zestawu_instrukcji i ma na celu zwiększenie (lub zmniejszenie) wartości zmiennej sterującej pętli (najczęściej inkrementacja/dekrementacja),
- zestaw_instrukcji — zestaw dowolnych instrukcji języka C++.

Zasadniczo wszystkie wyrażenia w nagłówku pętli `for` są opcjonalne, czyli mogą być pominięte.

Działanie pętli `for` można opisać za pomocą algorytmu w postaci listy kroków:

Krok 1.

Wykonaj wyrażenie_inicjujące.

Krok 2.

Wykonaj wyrażenie_warunkowe, czyli sprawdź, czy warunek jest spełniony (true), czy nie (false).

Krok 3.

Jeśli warunek jest spełniony (wynik wyrażenia_warunkowego to true), wykonaj zestaw_instrukcji zawartych wewnątrz pętli. W przeciwnym razie (wynik wyrażenia_warunkowego to false) przejdź do następnej instrukcji w programie.

Krok 4.

Wykonaj wyrażenie_modyfikujące, a następnie przejdź do *kroku 2*.

Podobnie jak pętla `while`, `for` jest pętlą z warunkiem na początku — o ile warunek (wyrażenie_warunkowe) istnieje. Działanie obu tych pętli jest analogiczne, dlatego pętle `while` oraz `for` często można stosować wymiennie.

Wykorzystanie pętli `for` zademonstrowano w przykładzie 3.31 oraz przykładzie 3.32. W pierwszym z nich pokazano użycie pętli `for` jako pętli powtarzanej, dopóki zadany warunek jest spełniony. Drugi z wymienionych przykładów (przykład 3.32) dotyczy sytuacji, w której określone operacje są wykonywane konkretną, zadaną z góry liczbę razy.

Przykład 3.31

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

int main() {
    int liczbaPocz = 1;
    int liczbaKonc = 3;

    int liczba = liczbaPocz, suma = 0;
    for (; liczba <= liczbaKonc;) {
        suma += liczba;
        liczba++;
    }
    cout << "Suma liczb wynosi: " << suma << endl;

    return 0;
}
```

W programie — podobnie jak w przykładach 3.29 i 3.30 — obliczana jest suma kolejnych liczb całkowitych, począwszy od zadanej liczby początkowej (zmienna `liczbaPocz`), a skończywszy na zadanej liczbie końcowej (zmienna `liczbaKonc`). Należy zwrócić uwagę, że użyta pętla `for` nie zawiera ani `wyrażenia_inicjującego`, ani `wyrażenia_zwiększającego`. Rolę zmiennej sterującej działaniem pętli odgrywa zmienna `liczba`. Instrukcje składowe zawarte wewnątrz pętli są powtarzane, dopóki warunek `liczba <= liczbaKonc` jest spełniony (`true`).

Omawiany program odpowiada funkcjonalnie i logicznie programowi zawartemu w przykładzie 3.29, w którym zastosowano pętlę `while`.

Ćwiczenie 3.31

Z wykorzystaniem programu zawartego w przykładzie 3.31 napisz program umożliwiający obliczenie sumy liczb całkowitych z przedziału $<1, 100>$. W obliczeniach uwzględnij osobno liczby parzyste (oblicz sumę liczb parzystych) i nieparzystych (oblicz sumę liczb nieparzystych). Wyniki należy wyświetlić na ekranie monitora. Do obliczeń użyj pętli `for`. Wspomniana pętla nie powinna zawierać ani `wyrażenia_inicjującego`, ani `wyrażenia_zwiększającego` — jak w programie w przykładzie 3.31.

Przykład 3.32

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

int main() {
    int liczbaPocz = 1;
    int iloscLiczb = 3;
```

```

int liczba = liczbaPocz, suma = 0;
for (int i = 1; i <= iloscLiczb; i++) {
    suma += liczba;
    liczba++;
}

cout << "Suma liczb wynosi " << suma << endl;

return 0;
}

```

W programie obliczana jest suma kolejnych liczb całkowitych, począwszy od zadanej liczby początkowej reprezentowanej przez zmienną `liczbaPocz`. Ilość sumowanych liczb wynosi `iloscliczb`.

Tutaj rolę zmiennej sterującej działaniem pętli `for` odgrywa zmienna `i`. Zmienna ta została zadeklarowana w pętli. Jej wartość zmienia się od wartości początkowej równej 1 do wartości równej `iloscliczb`. Zatem instrukcje składowe zawarte wewnątrz pętli są powtarzane określona, zadaną z góry liczbę razy równą `iloscliczb`.

Ćwiczenie 3.32

Zmodyfikuj program zawarty w przykładzie 3.32 — zamiast pętli `for` wykorzystaj pętlę `while` (wariant pierwszy programu) oraz pętlę `do-while` (wariant drugi).

UWAGA

Wszystkie pętle programowe można zagnieźdzać — podobnie jak instrukcje warunkowe.

3.3.4. Instrukcje `break` i `continue`

Wewnątrz instrukcji pętli programowych można stosować instrukcje `break` i `continue`. Stanowią one wówczas instrukcje składowe bloku kodu powtarzanego w danej pętli, a sama instrukcja pętli jest dla nich instrukcją nadziedną.

Instrukcje `break` i `continue` mają wpływ na proces iteracyjny, zachodzący jako skutek wykonywania pętli.

Instrukcja `break`

Zastosowanie instrukcji `break` jako jednej z instrukcji składowych zestawu_instrukcji powoduje zaprzestanie wykonywania instrukcji nadziednej, czyli instrukcji pętli, i przejście do następnej instrukcji w programie, występującej w kodzie źródłowym po instrukcji pętli. Innymi słowy, wykonanie instrukcji `break` spowoduje natychmiastowe wyjście na zewnątrz pętli.

Instrukcja continue

Zastosowanie instrukcji `continue` wewnątrz pętli (jako jednej z instrukcji zestawu_instrukcji) powoduje przerwanie bieżącej iteracji (bieżącego powtórzenia pętli) i przejście do następnej iteracji — jeśli oczywiście jest to możliwe, czyli warunek wykonania następnej iteracji jest spełniony.

Przerwanie wykonywania bieżącej iteracji (jako skutek wykonania instrukcji `continue`) spowoduje, że instrukcje składowe zawarte w zestawie_instrukcji po instrukcji `continue` nie zostaną wykonane.

Przykład 3.33

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

int main() {
    int liczbaPocz = 2;
    int liczbaKonc = 6;

    int liczba = liczbaPocz-1, suma = 0;
    for (;;) {
        // Inkrementacja liczby, która jest brana pod uwagę w obliczeniach:
        liczba++;

        // Sprawdzenie, czy bieżąca wartość liczby jest większa od zadanej wartości końcowej:
        if (liczba > liczbaKonc) break;
        /* UWAGA
         * Jeśli bieżąca wartość liczby jest większa od zadanej wartości końcowej, pętla kończy działanie.
         */

        // Sprawdzenie, czy bieżąca wartość liczby jest nieparzysta:
        if ((liczba % 2) != 0) continue;
        /* UWAGA
         * Jeśli bieżąca wartość liczby jest nieparzysta, bieżąca iteracja (bieżące powtórzenie pętli) się kończy.
         * Instrukcje poniżej nie zostaną wykonane.
         */

        // Dodanie liczby do bieżącej wartości sumy liczb:
        suma += liczba;
    }
    cout << "Suma liczb parzystych wynosi: " << suma << endl;

    return 0;
}
```

W programie obliczana jest suma kolejnych liczb całkowitych parzystych (zmienna `suma`), począwszy od zadanej liczby początkowej reprezentowanej przez zmienną `liczbaPocz`, a skończywszy na zadanej liczbie końcowej `liczbaKonc`. Wykorzystana pętla `for` nie zawiera ani `wyrażenia_inicjującego`, ani `wyrażenia_warunkowego`, ani `wyrażenia_zwiększającego`. Stegowanie przebiegiem działania pętli jest realizowane za pomocą instrukcji `break` i `continue`.

Ćwiczenie 3.33

Zmodyfikuj program zawarty w przykładzie 3.33 — zamiast pętli `for` wykorzystaj pętlę `while` (wariant pierwszy programu) oraz pętlę `do-while` (wariant drugi).

UWAGA

Instrukcje `break` oraz `continue` mogą występować w połączeniu z wymuszeniem za pomocą instrukcji `goto` skoku do określonego wyrażenia (instrukcji) opatrzonego etykietą. W takim przypadku następuje zaprzestanie wykonywania albo całej pętli (`break`), albo bieżącej iteracji (`continue`), a później skok do instrukcji poprzedzonej wskazaną etykietą. Jednakże zgodnie z zasadami programowania strukturalnego należy unikać stosowania instrukcji skoku `goto`, ponieważ niszczy to strukturę programu. Zasady programowania strukturalnego zostały omówione szczegółowo w podrozdziale 11.1.

3.3.5. Zagnieżdżanie pętli programowych

Pętle programowe też można zagnieżdzać, podobnie jak instrukcje warunkowe. Zagnieżdżanie pętli polega na wykonaniu jednej instrukcji pętli w innej pętli. Przy tym nie ma reguły, że w strukturze zagnieżdżonych pętli muszą występować pętle tego samego typu, np. wyłącznie pętle `while`. W ogólności na dowolnym poziomie w strukturze zagnieżdżenia pętli mogą występować dowolne pętle — jak to zademonstrowano w przykładzie 3.34.

Przykład 3.34

W programie są wprowadzane z klawiatury długości trzech boków prostopadłościanu. Długość każdego z nich powinna być określona za pomocą liczby dodatniej.

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

int main() {
    double bok {0}; // inicjalizacja zmiennej w stylu C++11

    cout << "Podaj długości boków prostopadłościanu" << endl;
    for (int i = 1; i <= 3; i++) {
        do {
            cout << "bok " << i << " = ";
            cin >> bok;
        } while (bok <= 0);
        cout << endl;
    }
}
```

```

        cin >> bok;
        if (bok < 0) cout << "Błąd! Wprowadź ponownie!" << endl;
    }
    while (bok < 0);

    cout << "Wprowadzona długość boku: " << bok << endl;
}

return 0;
}

```

Mamy tutaj do czynienia z zagnieżdzaniem pętli programowych. Pętla `do-while` stanowi instrukcję składową zawartą w pętli `for`. Pętla `while` jest zatem pętlą wewnętrzną, a `for` — zewnętrzną.

Instrukcje składowe w pętli `do-while` są powtarzane, dopóki spełniony jest warunek `bok < 0`, czyli wprowadzona długość boku jest błędna. Instrukcje zawarte w pętli `for` zaś są powtarzane określona, zadana z góry liczbę razy — trzykrotnie.

Ćwiczenie 3.34

Zmodyfikuj program zawarty w przykładzie 3.34 — zamiast pętli `for` zastosuj pętlę `while` (wariant pierwszy) oraz pętlę `do-while` (wariant drugi).



3.4. Typ wyliczeniowy

3.4.1. Definicja typu wyliczeniowego

Typ wyliczeniowy (ang. *enumerated data type*) należy do typów danych definiowanych przez użytkownika (ang. *user-defined data types*). Postać ogólna definicji typu wyliczeniowego jest następująca:

```
enum nazwa {lista_identyfikatorów};
```

gdzie:

- `enum` — słowo kluczowe,
- `nazwa` — identyfikator definiowanego typu danych,
- `lista_identyfikatorów` — zestaw wartości oddzielonych od siebie przecinkami.

Wartości wyszczególnione na liście_identyfikatorów to stałe symboliczne, które są nazywane **enumeratorami**. Każdemu enumeratorowi odpowiada unikatowy numer porządkowy — liczba całkowita (o dowolnym znaku). Domyślnie enumeratory są numerowane kolejnymi liczbami całkowitymi: 0, 1, 2 itd. — zgodnie z kolejnością ich występowania na liście_identyfikatorów.

Na przykład lista identyfikatorów (enumeratorów) reprezentujących stanowiska pracy w danej firmie może mieć postać: dyrektor, kierownik, murarz, malarz, tynkarz, stolarz, a kompletna definicja typu wyliczeniowego o nazwie stanowiska określonego na podstawie tej listy:

```
enum Stanowiska {dyrektor, kierownik, murarz, malarz, tynkarz, stolarz, portier};
```

Wartości (enumeratorowi) dyrektor odpowiada liczba (numer) 0, kierownikowi — 1 itd.

Poszczególnym lub wybranym enumeratorom na liście_identyfikatorów można również nadawać wartości w sposób jawnny — bezpośredni. W tym celu wykorzystuje się postać ogólną:

identyfikator = wartość

gdzie identyfikator oznacza określony enumerator na liście_identyfikatorów, a wartość — przypisaną mu liczbę całkowitą.

3.4.2. Zmienne wyliczeniowe

Zmienna należąca do typu wyliczeniowego — **zmienna wyliczeniowa** (ang. *enumeration variable*) — może przyjmować wartość (całkowitą) wyłącznie pojedynczego, wybranego enumeratora.

Deklarację takiej zmiennej można zrealizować w dwojakim sposobie:

- w wyrażeniu zawierającym definicję typu wyliczeniowego,
- w klasyczny sposób, podobnie jak deklaracje zmiennych należących do typów podstawowych.

Pierwszy z wymienionych powyżej sposobów zilustrowano w przykładzie 3.35, a drugi — w przykładzie 3.36.

Przykład 3.35

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

int main() {
    // Deklaracja zmiennej o nazwie stanowisko połączona z definicją typu wyliczeniowego Stanowisko:
    enum Stanowisko { // definicja typu wyliczeniowego Stanowisko
        dyrektor, // enumerator o wartości 0
        kierownik, // enumerator o wartości 1
        sekretarka, // enumerator o wartości 2
        portier // enumerator o wartości 3
    } stanowisko; // deklaracja zmiennej stanowisko

    // Prezentacja wartości zmiennej stanowisko:
    cout << stanowisko << endl; // 0
    /* UWAGA
     * Zmienna stanowisko została w czasie deklaracji zainicjowana wartością domyślną 0.
     */
```

```

// Przypisanie zadanej wartości enumeratora zmiennej stanowisko:
stanowisko = sekretarka;
// Prezentacja wartości zmiennej stanowisko:
cout << stanowisko << endl; //2

return 0;
}

```

W programie zdefiniowano typ wyliczeniowy o nazwie `Stanowisko` zawierający cztery enumeratory. Enumeratorom zostały niejawnie przypisane wartości całkowite: 0, 1, 2 i 3 — zgodnie z porządkiem ich wyliczenia w definicji typu `Stanowisko`, co odpowiada domyльнemu zachowaniu kompilatora.

Zmienna `stanowisko` należąca do typu `Stanowisko` została podczas deklaracji zainicjowana wartością domyślną zerową: 0. Przypisanie zmiennej `stanowisko` wartości innego enumeratora (`sekretarka`) skutkuje zmianą wartości zmiennej `stanowisko` na 2.

Ćwiczenie 3.35

Zmodyfikuj kod z przykładu 3.35 tak, żeby wartość zmiennej `stanowisko` była wprowadzana z klawiatury i żeby wartość zapamiętanego w niej enumeratora była wyświetdana kontrolnie na ekranie monitora.

Przykład 3.36

```

#include <iostream>
using namespace std;

int main() {
    // Definicja typu wyliczeniowego o nazwie Ocena:
    enum Ocena {
        niedostateczny = 1,
        dopuszczajacy = 2,
        dostateczny = 3,
        dobry = 4,
        bardzo_dobry = 5,
        celujacy = 6
    };

    // Deklaracja i inicjalizacja zmiennej ocena należącej do typu Ocena:
    Ocena ocena = dostateczny;

    // Prezentacja wartości oceny w formie słownej i liczbowej:
    cout << "Ocena słowna: ";
    switch (ocena) {
        case niedostateczny:

```

```

        cout << "niedostateczny" << endl;
        break;
    case dopuszczajacy:
        cout << "dopuszczający" << endl;
        break;
    case dostateczny:
        cout << "dostateczny" << endl;
        break;
    case dobry:
        cout << "dobry" << endl;
        break;
    case bardzo_dobry:
        cout << "bardzo dobry" << endl;
        break;
    case celujacy:
        cout << "celujący" << endl;
        break;

    default: cout << "nieokreślona" << endl;
}
cout << "Ocena liczbowa: " << ocena << endl;

return 0;
}

```

W programie wykorzystano typ wyliczeniowy o nazwie Ocena. Enumeratorom wchodząącym w jego skład nadano wartości w sposób jawnego. Nadana wartość zmiennej ocena typu Ocena jest prezentowana na ekranie monitora w formie słownej (odpowiadającej identyfikatorowi właściwego enumeratora) i w formie liczbowej.

Ćwiczenie 3.36

Napisz program pozwalający wyświetlić w formie słownej ustaloną długość okresu kształcenia w szkole podstawowej, liceum i technikum. Wykorzystaj zdefiniowany samodzielnie typ wyliczeniowy.



3.5. Pytania i zadania kontrolne

3.5.1. Pytania

1. Co to jest zmienna i jaka jest rola zmiennej zadeklarowanej w programie?
2. Na czym polega inicjalizacja zmiennej?
3. Co oznacza sformułowanie: „zmienna o nazwie promien należy do typu danych float”?

4. Wymień i krótko opisz predefiniowane, podstawowe typy danych w języku C++.
5. Co to jest literal? Jaka jest różnica pomiędzy literałem a stałą nazwaną? Czym się różni zmienna od stałej nazwanej?
6. Co oznacza pojęcie operator, a co pojęcie operand?
7. Omów działanie operatora przypisania prostego.
8. Co to jest wyrażenie? Czym się różni wyrażenie od instrukcji?
9. Na czym polega konwersja typu danej?
10. W jakim celu definiuje się różne przestrzenie nazw?
11. Co oznacza termin *wejściowy strumień danych*, a co *strumień wyjściowy*?
12. Jaką rolę w programie odgrywa funkcja specjalna `main()`?
13. Wyjaśnij pojęcie *wyrażenie logiczne*.
14. Na czym polega różnica w działaniu instrukcji warunkowej `if` względem instrukcji `if-else`?
15. Opisz działanie instrukcji wyboru `switch`. Narysuj blokowy schemat działania tej instrukcji.
16. Na czym polega zagnieżdżanie instrukcji warunkowych `if-else`?
17. Dlaczego instrukcje warunkowe są zaliczane do struktur sterujących?
18. Z jakiej przyczyny instrukcje `if`, `if-else` oraz `switch` są również nazywane instrukcjami wyboru?
19. Wymień różnice w budowie i działaniu pętli `while` względem pętli `do-while`.
20. Wymień różnice w działaniu pętli `for` względem pętli `while`.
21. Wyjaśnij znaczenie terminu *zmienna sterująca pętli*.
22. Wyjaśnij znaczenie terminu *iteracja*.
23. Na czym polega różnica w działaniu instrukcji `break` względem instrukcji `continue`, jeśli obie są wykonywane wewnątrz pętli?
24. Czy pętle programowe można zagnieździć jedna w drugiej?
25. Czy wartości enumeratorów w definicji typów wyliczeniowych mogą być ujemne?
26. Czy wartości enumeratorów mogą być wykorzystywane w połączeniu z pętlami programowymi? Uzasadnij odpowiedź.

3.5.2. Zadania

1. Napisz program umożliwiający obliczenie pola i obwodu trójkąta równoramiennego. Dane wejściowe — długość podstawy i wysokość — należy zainicjować w programie, a wyniki obliczeń zaprezentować w konsoli na ekranie monitora.

2. Napisz program zawierający definicję stałej nazwanej *PI*. Wykorzystaj tę stałą do wyznaczenia objętości i pola powierzchni kuli. Dane wejściowe, tj. promień kuli, mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora.
3. Napisz program pozwalający na przeliczenie prędkości zadanej w kilometrach na godzinę na prędkość wyrażoną w węzłach. Dane wejściowe mają być wprowadzane z klawiatury, a wynik wyświetlany na ekranie monitora.
4. Napisz program umożliwiający obliczenie wartości wyrażenia: $w = \frac{a+b}{c+d} + \frac{a}{c} + \frac{b}{d}$. Dane wejściowe (wartości parametrów *a*, *b*, *c*, *d*) mają być wprowadzane z klawiatury, a wartość wyniku *w* ma być wyświetlana na ekranie monitora.
5. Napisz program pozwalający na wyznaczenie miejsca zerowego funkcji danej wzorem: $y = f(x) = 2 \cdot x - 4$. Wynik wyświetl na ekranie monitora.
6. Napisz program pozwalający na obliczenie średniej arytmetycznej z trzech ocen semestralnych uzyskanych z następujących przedmiotów: język polski, język angielski, matematyka. Dane wejściowe mają być wprowadzane z klawiatury, a wynik niech będzie wyświetlany w konsoli na ekranie monitora.
7. Napisz program umożliwiający obliczenie pola i obwodu koła dla promienia o długości wprowadzonej z klawiatury. Zabezpiecz program przed wykonywaniem obliczeń dla promienia o długości niedodatniej. W razie wprowadzenia z klawiatury błędnej (np. ujemnej) wartości promienia na ekranie powinien zostać wyświetlony odpowiedni komunikat.
8. Napisz program umożliwiający obliczenie pola i obwodu prostokąta dla boków o długościach wprowadzonych z klawiatury. Zabezpiecz program przed wykonywaniem obliczeń dla boków o długości niedodatniej. Zaprojektuj i zastosuj system komunikatów dla użytkownika informujący go o błędnych (niepoprawnych) wartościach danych wejściowych.
9. Napisz program pozwalający na obliczenie wartości wyrażenia *w*, danego wzorem:

$$w = \frac{a}{b} + \frac{c}{a+b} - \frac{a+c}{a+b+a}$$

Wartości argumentów: *a*, *b*, *c* i *d* mają być wprowadzane z klawiatury. Zabezpiecz program przed próbą (probami) dzielenia przez zero. Wykonaj program w dwóch wariantach:

- z użyciem instrukcji *if* (wariant pierwszy),
- z użyciem instrukcji *if-else* (wariant drugi).

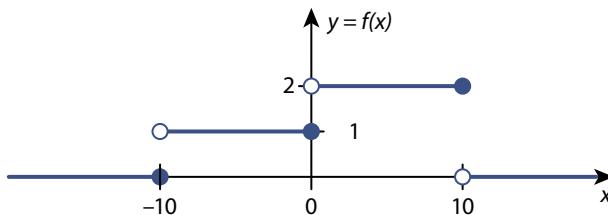
10. Napisz program pozwalający na obliczenie średniej geometrycznej *s* z czterech liczb: *a*, *b*, *c* i *d*, na podstawie zależności: $s = \sqrt[4]{a \cdot b \cdot c \cdot d}$, gdzie $a \geq 0$, $b \geq 0$, $c \geq 0$ oraz $d \geq 0$. Dane wejściowe (liczby *a*, *b*, *c* i *d*) mają być wprowadzane z klawiatury.

11. Napisz aplikację pozwalającą na obliczenie wartości funkcji $y = f(x)$, danej wzorem:

$$y = f(x) = \sqrt{\frac{\sin(x)}{x-1}}.$$

Wartość argumentu x ma być wprowadzana z klawiatury. Zabezpiecz aplikację przed wykonaniem niedozwolonych operacji, tj. przed próbą wyznaczenia pierwiastka z liczby ujemnej oraz próbą dzielenia przez zero.

12. Napisz program umożliwiający obliczenie wartości funkcji $y = f(x)$ danej za pomocą wykresu przedstawionego na rysunku 3.5.



Rysunek 3.5. Wykres funkcji $y = f(x)$

Wartość argumentu x ma być wprowadzana z klawiatury. Wykonaj aplikację w dwóch wariantach:

- przy wykorzystaniu zagnieżdżonych instrukcji warunkowych `if` (wariant pierwszy),
- przy wykorzystaniu zagnieżdżonych instrukcji `if-else` (wariant drugi).

13. Napisz program umożliwiający obliczenie sumy kolejnych liczb całkowitych, począwszy od zadanej liczby początkowej $L1$, a skończywszy na zadanej liczbie końcowej $L2$. Liczby $L1$ i $L2$ mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora w konsoli. W obliczeniach uwzględnij osobno liczby parzyste i liczby nieparzyste. Wykonaj program w trzech wariantach odpowiadających różnym rodzajom pętli (`while`, `do-while` i `for`).

14. Napisz program umożliwiający wyznaczenie wartości największej (maksimum) i najmniejszej (minimum) z n zadanych liczb rzeczywistych przy założeniu, że wartość n nie przekracza 10. Dane wejściowe (ilość liczb n oraz wartości n liczb rzeczywistych) należy wprowadzić z klawiatury. Wyniki powinny być wyświetlane w konsoli na ekranie monitora. Wykonaj program w trzech wariantach odpowiadających różnym rodzajom pętli: `while`, `do-while` oraz `for`.

15. Napisz program umożliwiający wyznaczenie sumy oraz wartości największej (maksimum) i najmniejszej (minimum) z 10 liczb całkowitych wygenerowanych losowo. Niech wyniki będą wyświetlane w konsoli na ekranie monitora. Wykonaj program w trzech wariantach odpowiadających różnym rodzajom pętli: `while`, `do-while` i `for`.

16. Napisz program pozwalający na sprawdzenie, które z liczb całkowitych z przedziału $<1, 100>$ są podzielne przez liczbę 9. Wypisz te liczby na ekranie monitora. Wykonaj program w trzech wariantach odpowiadających różnym rodzajom pętli: `while`, `do-while` i `for`.

- 17.** Napisz program pozwalający obliczyć pole i obwód prostokąta. Długości boków prostokąta mają być wprowadzane z klawiatury. Uwzględnij możliwość wielokrotnego wprowadzenia danych wejściowych (długości boków prostokąta) w razie podania błędnych wartości. Załącz, że maksymalna liczba prób wprowadzenia każdej z danych wejściowych wynosi 10. Wyniki powinny być wyświetlane na ekranie monitora.
- 18.** Napisz program pozwalający wyznaczyć średnią arytmetyczną i średnią geometryczną z 5 liczb rzeczywistych dodatnich o wartościach wprowadzonych z klawiatury. Uwzględnij możliwość wielokrotnego wprowadzenia danych wejściowych w razie podania błędnych wartości. Wyniki powinny być wyświetlane na ekranie monitora.
- 19.** Napisz program umożliwiający obliczenie średniej arytmetycznej z ocen semestralnych ucznia uzyskanych z następujących przedmiotów: język polski, język angielski, matematyka, informatyka. Wykorzystaj zdefiniowany samodzielnie typ wyliczeniowy pozwalający przedstawić średnią ocenę w postaci słownej. Dane wejściowe mają być wprowadzane z klawiatury, a wynik powinien być wyświetlany na ekranie monitora.
- 20.** Napisz program umożliwiający określenie rodzaju szkoły na podstawie znajomości długości okresu kształcenia. Dane wejściowe (długość okresu kształcenia) mają być wprowadzane z klawiatury, a wynik (rodzaj szkoły) niech będzie wyświetlany na ekranie monitora. Wykorzystaj zdefiniowany samodzielnie typ wyliczeniowy pozwalający skojarzyć rodzaj szkoły z długością okresu kształcenia.

4

Programowanie z użyciem wskaźników

Programowanie w języku C++ jest ściśle związane z wykorzystaniem **wskaźników** (ang. *pointers*). W pełni zasadne jest stwierdzenie, że nie można skutecznie i wydajnie programować w języku C++ bez użycia wskaźników.

W tym rozdziale można się zapoznać jedynie z podstawowymi pojęciami dotyczącymi wskaźników. W szczególności są w nim omawiane wskaźniki do danych podstawowych, takich jak `int` czy `double`. Wskaźniki do innych typów danych, takich jak tablice, struktury, klasy, zostały przedstawione w następnych rozdziałach podręcznika. To samo dotyczy wykorzystania wskaźników jako parametrów i argumentów funkcji. Dlatego też dopiero po wnioskowym zapoznaniu się z materiałem zawartym w dalszych rozdziałach wiedza i umiejętności dotyczące zasad i technik wykorzystania wskaźników w programowaniu w języku C++ będą satysfakcjonujące.

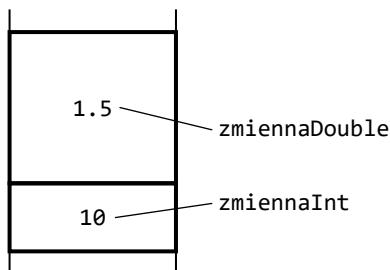
4.1. Operator adresu

Z treści zaprezentowanych w rozdziale 3. jasno wynika, że każdą zmienną zadeklarowaną w programie charakteryzuje:

- jej nazwa — identyfikator,
- typ danych, które można w niej zapamiętać,
- wartość, która faktycznie jest w niej przechowywana.

Zmienne są pamiętane w pamięci operacyjnej komputera w określonych obszarach — blokach. Położenie wspomnianych bloków, czyli ich adresy początkowe, jest zależne od systemu operacyjnego. Rozmiary bloków są zaś determinowane przez typy zmiennych zadeklarowanych w programie.

Na przykład po jawnym zadeklarowaniu i zainicjowaniu w programie zmiennych: `int zmiennaInt = 10;` oraz `double zmiennaDouble = 1.5;` fragment pamięci operacyjnej odpowiadający tym zmiennym można przedstawić poglądowo w sposób pokazany na rysunku 4.1.



Rysunek 4.1. Interpretacja bloków w pamięci operacyjnej, w których są przechowywane zmienne

Identyfikatory zmiennych (tj. `zmiennaInt` i `zmiennaDouble`) można potraktować jak nazwy określonych obszarów (bloków) w pamięci operacyjnej. Zawartość tych bloków (10 i 1.5) jest zaś związana z wartościami, które są przechowywane w zmiennych. Rozmiar bloków pamięci skojarzonych z zadeklarowanymi zmiennymi odpowiada typom tych zmiennych. Mianowicie zmienna `zmiennaInt` typu `int` zajmuje w pamięci operacyjnej 4 bajty, a zmienna `zmiennaDouble` typu `double` 8 bajtów.

Zmienne należące do predefiniowanych typów danych podstawowych (np. `char`, `bool`, `int`, `double`), które w kodzie programu zostały zadeklarowane w sposób jawny (ang. *explicit declaration*), mają pamięć operacyjną przydzieloną — zaalokowaną w sposób statyczny (ang. *static memory allocation*). Zapotrzebowanie na pamięć wymaganą przez te zmienne jest znane już na etapie komplikacji programu i nie zmienia się w trakcie jego wykonywania.

Należące do typów podstawowych zmienne, które zostały zadeklarowane w sposób jawny, są przechowywane w pamięci operacyjnej komputera w regionie (segmencie) nazywanym **stosem** (ang. *stack*). Zapotrzebowanie programu na pamięć operacyjną na stosie jest ustalane przez kompilator w czasie komplikacji. Proces ten jest nazywany **alokacją pamięci w czasie komplikacji** (ang. *compile-time memory allocation*). Podczas działania programu rozmiar pamięci przydzielonej mu na stosie się nie zmienia.

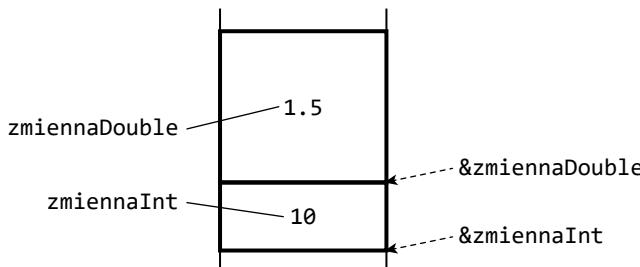
UWAGA

Więcej informacji na temat zmiennych przechowywanych na stosie (np. zmiennych należących do typów danych zdefiniowanych przez programistę), z uwzględnieniem podziału zmiennych na globalne i lokalne, można znaleźć w rozdziale 8. podręcznika, dotyczącym funkcji.

**UWAGA**

W programowaniu pojęcie stosu oznacza — oprócz segmentu (regionu) w pamięci operacyjnej — dynamiczną strukturę danych.

Każdy z bloków pamięci operacyjnej komputera, w których przechowywane są zadeklarowane zmienne, można zidentyfikować jednoznacznie za pomocą adresu tego bloku. W języku C++ adres bloku pamięci operacyjnej, w którym zapisana jest zmienna o określonej nazwie, można uzyskać za pomocą **operatora adresu** (ang. *address-of operator, address operator*): & (ang. *ampersand sign*), np. `&zmiennaInt`, `&zmiennaDouble`. Zilustrowano to na rysunku 4.2.



Rysunek 4.2. Interpretacja operatora adresu zmiennej

Wyrażenie `&zmiennaInt` określa adres komórki pamięci operacyjnej stanowiącej początek bloku, w którym przechowywana jest zmienna `zmiennaInt`. Wyrażenie `&zmiennaDouble` określa zaś adres początku bloku pamięci, w którym przechowywana jest zmienna `zmiennaDouble`.

Przykład 4.1

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

int main() {
    // Deklaracja i inicjalizacja zmiennej o nazwie zmiennaInt należącej do typu całkowitego int:
    int zmiennaInt = 10;

    // Prezentacja informacji dotyczących zmiennej zmiennaInt na ekranie monitora:
    cout << "Informacje dotyczące zmiennej zmiennaInt:" << endl;
    // Wyświetlenie wartości przechowywanej w zmiennej zmiennaInt:
    cout << "wartość: " << zmiennaInt << endl;
    // Prezentacja rozmiaru zmiennej zmiennaInt:
    cout << "rozmiar: " << sizeof(zmiennaInt) << endl;
    // Wykorzystanie operatora adresu w celu określenia adresu zmiennej zmiennaInt:
    cout << "adres: " << &zmiennaInt << endl;

    // Deklaracja i inicjalizacja zmiennej o nazwie zmiennaDouble należącej do typu rzeczywistego double:
    double zmiennaDouble = 1.5;
```

```
// Prezentacja informacji dotyczącej zmiennej zmiennośćDouble w konsoli na ekranie monitora:
cout << "Informacje dotyczące zmiennej zmiennośćDouble:" << endl;
cout << "wartość: " << zmiennośćDouble << endl;
cout << "rozmiar: " << sizeof(zmiennośćDouble) << endl;
cout << "adres: " << &zmiennośćDouble << endl;

return 0;
}
```

W przedstawionym programie wykorzystano dwie zmienne należące do typów podstawowych: `zmiennośćInt` oraz `zmiennośćDouble`. Rozmiary tych zmiennych są wyświetlane na ekranie monitora w bajtach, a ich adresy — w systemie szesnastkowym (heksadecymalnym).

Ćwiczenie 4.1

Zmodyfikuj program zaprezentowany w przykładzie 4.1. Zamiast zmiennych typu `int` (`zmiennośćInt`) i `double` (`zmiennośćDouble`) użyj zmiennych typu `long int` i `float`. Zinterpretuj uzyskane rezultaty.

4.2. Wskaźniki

Symboliczną reprezentację adresów zmiennych stanowią **zmienne wskaźnikowe** (ang. *pointer variables*), nazywane krótko **wskaźnikami** (ang. *pointers*). Wskaźniki to zwykłe zmienne nazwane (tj. mające identyfikatory), w których można przechowywać adresy innych zmiennych określonego typu.

Zmienna określonego typu (np. `int`, `double`), której adres został zapamiętany we wskaźniku, jest nazywana zmienną wskazywaną przez ten wskaźnik, a typ danych, do którego należy zmienna wskazywana — **typem bazowym** wskaźnika (ang. *pointer base type*).

4.2.1. Deklaracja zmiennej wskaźnikowej

Każdy wskaźnik używany w programie powinien zostać wcześniej zadeklarowany. Postać ogólna deklaracji wskaźnika jest następująca:

typ_bazowy *wskaźnik;

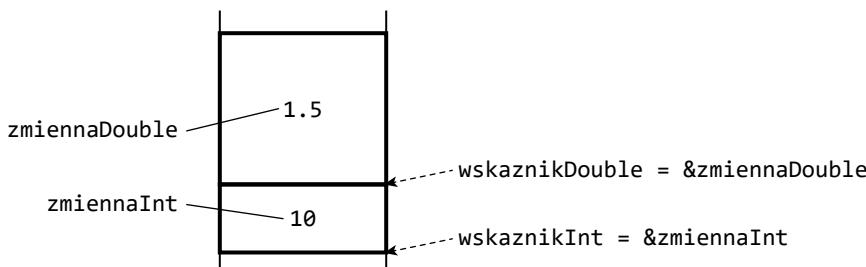
gdzie `typ_bazowy` oznacza określony typ zmiennej (np. `int`, `double`), która może być wskazywana przez wskaźnik.

UWAGA

Położenie symbolu `*` w deklaracji wskaźnika jest dowolne. Poprawną składnią deklaracji wskaźnika jest zarówno notacja `typ_bazowy *wskaźnik;`, jak i `typ_bazowy* wskaźnik;` oraz `typ_bazowy * wskaźnik;`. Jednakże zalecaną formą deklaracji wskaźnika jest ta, którą zastosowano w postaci ogólnej powyżej.

Wskaźnik jako zmienna zadeklarowana w programie — co wiąże się z nadaniem jej unikatowej nazwy — należy do określonego **typu wskaźnikowego** (ang. *pointer type*). Typ wskaźnikowy jest określony jednoznacznie za pomocą typu bazowego, o którym była mowa wcześniej. W ogólności typy wskaźnikowe należą do **typów pochodnych** (ang. *derived data types*), ponieważ są definiowane na podstawie typów bazowych.

Deklaracja wskaźnika w programie może być połączona z jego inicjalizacją — tak samo jak w przypadku innych zmiennych. Na przykład deklaracja zmiennej wskaźnikowej o nazwie `wskaznikInt`, w której można zapamiętać adres dowolnej zmiennej typu `int`, jest następująca: `int *wskaznikInt;`. Przypisanie wskaźnikowi `wskaznikInt` adresu zmiennej `zmiennaInt` może zaś mieć postać: `wskaznikInt = &zmiennaInt;`. Analogicznie deklaracja połączona z inicjalizacją wskaźnika `wskaznikDouble`, który wskazuje na zmienną `zmiennaDouble` typu `double`, może mieć postać: `double *wskaznikDouble = &zmiennaDouble;`. Zilustrowano to na rysunku 4.3.



Rysunek 4.3. Interpretacja zmiennej wskaźnikowej (wskaźnika)

Zmienna `zmiennaInt` jest zmienną wskazywaną przez wskaźnik `wskaznikInt`. Typem bazowym wskaźnika `wskaznikInt` jest typ całkowity `int`. Typ wskaźnikowy, do którego należy wskaźnik `wskaznikInt`, można określić (poglądowo) jako `int*` — co oznacza, że dopuszczalnym zbiorem wartości, jakie może przyjmować zmiana `wskaznikInt`, są adresy zmiennych typu całkowitego `int`, który jest typem bazowym dla tego wskaźnika. Zmienną wskazywaną przez wskaźnik `wskaznikDouble` jest zaś `zmiennaDouble`, a typem bazowym tego wskaźnika — typ `double`.

W operacjach na wskaźnikach często wykorzystuje się literał wskaźnikowy (ang. *pointer literal*) o nazwie `nullptr`, który określa wskaźnik pusty (ang. *null-pointer*). Zamiast literatu `nullptr` można również używać stałej o nazwie `NULL`, predefiniowanej za pomocą makrodefinicji (ang. *macro-definition*).

UWAGA

Tematyka definiowania stałych za pomocą makrodefinicji została omówiona w podrozdziale 9.2.1 podręcznika.

Stała `NULL` przyjmuje wartość całkowitego 0 (lub `0L`), ale może być konwertowana na dowolny typ wskaźnikowy. Na przykład poprawną instrukcją jest `int *wskaźnikInt = NULL;`, jak również `double *wskaźnikDouble = NULL;.` Przypisanie wskaźnikowi wartości `NULL` oznacza, że nie wskazuje on na żadną zmienną.

Przykład 4.2

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

int main() {
    int jPolski, matematyka;

    // Deklaracja i inicjalizacja wskaźnika w_jPolski:
    int *w_jPolski {}; // wskaźnik w_jPolski zostanie zainicjowany adresem o wartości 0
    // Nadanie wskaźnikowi w_jPolski wartości równej adresowi zmiennej jPolski:
    w_jPolski = &jPolski;

    // Deklaracja wskaźnika w_matematyka i nadanie mu wartości początkowej równej adresowi zmiennej
    // matematyka:
    int *w_matematyka {&matematyka};

    cout << "Podaj oceny semestralne ucznia " << endl;
    cout << "ocena z języka polskiego = "; cin >> jPolski;
    cout << "ocena z matematyki = "; cin >> matematyka;

    cout << endl << "Wprowadzono następujące dane:" << endl;
    cout << "wartość zmiennej jPolski = " << jPolski << endl;
    cout << "adres zmiennej jPolski = " << &jPolski << endl;
    cout << "adres przechowywany we wskaźniku w_jPolski = " << w_jPolski
        << endl;

    cout << "wartość zmiennej matematyka = " << matematyka << endl;
    cout << "adres zmiennej matematyka = " << &matematyka << endl;
    cout << "adres przechowywany we wskaźniku w_matematyka = " << w_matematyka
        << endl;

    return 0;
}
```

W zaprezentowanym programie przetwarzane są dwie dane: oceny semestralne uzyskane przez ucznia z języka polskiego i z matematyki. Oceny te są reprezentowane przez zmienne `jPolski` i `matematyka`. Wskaźnikom do tych zmiennych nadano nazwy, odpowiednio, `w_jPolski` i `w_matematyka`. Inicjalizacja wskaźników została przeprowadzona w notacji C++11. Wyjście programu polega na wyświetleniu różnych informacji: wartości zmiennych

`jPolski` i `matematyka`, adresów tych zmiennych oraz, dla porównania, wartości przechowywanych we wskaźnikach `w_jPolski` i `w_matematyka`.

Ćwiczenie 4.2

Zmodyfikuj program zawarty w przykładzie 4.2. Uwzględnij oceny semestralne uzyskane przez ucznia z czterech przedmiotów: języka polskiego, matematyki, języka angielskiego i informatyki. Oblicz średnią arytmetyczną tych ocen i wyświetl wynik na ekranie monitora.

4.2.2. Operator dereferencji

Odwołanie się do zawartości bloku pamięci, w którym przechowywana jest zmienna wskażywana przez wskaźnik, można uzyskać za pomocą **operatora dereferencji** (ang. *dereference operator*), `*`.

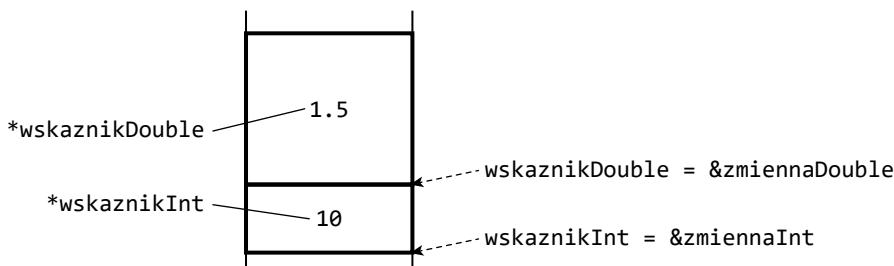
Jeśli spojrzeć z perspektywy składni języka C++, operatora dereferencji używa się w powiązaniu ze wskaźnikiem:

***wskaźnik**

gdzie **wskaźnik** stanowi formalnie zmienną wskaźnikową zadeklarowaną wcześniej, czyli należącą do określonego typu wskaźnikowego.

Odwołanie się do zawartości bloku pamięci wskazywanego przez wskaźnik ma sens praktyczny jedynie wtedy, gdy wskaźnikowi została wcześniej przypisany adres określonej zmiennej typu bazowego. Wówczas konstrukcja językowa `*wskaźnik` będzie reprezentować faktyczną wartość zapisaną w zmiennej wskazywanej przez ten wskaźnik.

Na przykład odwołanie się do zawartości bloku pamięci, w której pamiętała jest zmienna o nazwie `zmiennaInt` wskazywana przez wskaźnik `wskaznikInt`, ma postać: `*wskaznikInt`. Analogicznie odwołanie się do wartości zmiennej `zmiennaDouble` wskazywanej przez wskaźnik `wskaznikDouble` jest realizowane przez użycie `*wskaznikDouble`. Zilustrowano to na rysunku 4.4.



Rysunek 4.4. Interpretacja operatora dereferencji `*`

Jak wynika z rysunku 4.4, wyrażenie `*wskaznikInt` stanowi pewnego rodzaju umowny alias nazwy `zmiennaInt`, a wyrażenie `*wskaznikDouble` — alias nazwy `zmiennaDouble`.

Przykład 4.3

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

int main() {
    // Deklaracja i inicjalizacja zmiennej bok1 typu double:
    double bok1 = 1;
    // Deklaracja zmiennej wskaźnikowej (wskaźnika) w_bok1 na zmienną typu double:
    double *w_bok1;
    // Przypisanie wskaźnikowi w_bok1 adresu zmiennej bok1:
    w_bok1 = &bok1;

    // Deklaracja i inicjalizacja zmiennej bok2 typu double:
    double bok2 = 2;
    // Deklaracja zmiennej wskaźnikowej (wskaźnika) w_bok2 na zmienną typu double:
    double *w_bok2;
    // Przypisanie wskaźnikowi w_bok2 adresu zmiennej bok2:
    w_bok2 = &bok2;

    // Deklaracja zmiennej pole typu double:
    double pole;
    // Deklaracja i inicjalizacja wskaźnika w_pole:
    double *w_pole = &pole;
    /* UWAGA
     * We wskaźniku w_pole zostaje zapamiętany adres zmiennej pole typu double.
     */
    // Obliczenie pola prostokąta — wykorzystanie wskaźników i operatora dereferencji:
    *w_pole = (*w_bok1) * (*w_bok2); // równoważne instrukcje: pole = bok1 * bok2;

    // Deklaracja zmiennej obwod typu double, deklaracja i inicjalizacja wskaźnika w_obwod:
    double obwod;
    double *w_obwod = &obwod;
    /* UWAGA
     * We wskaźniku w_obwod pamiętany jest adres zmiennej obwod typu double.
     */
    // Obliczenie obwodu prostokąta — wykorzystanie wskaźników i operatora dereferencji:
    *w_obwod = 2 * (*w_bok1) + 2 * (*w_bok2);

    // Prezentacja wyników — wykorzystanie wskaźników:
    cout << "Wyniki:" << endl;
    cout << "Pole wynosi " << *w_pole << endl;
    cout << "Obwód wynosi " << *w_obwod << endl;

    return 0;
}
```

W przykładzie obliczane są pole i obwód prostokąta dla danych wejściowych (długości boków) inicjowanych w programie. Przetwarzanie danych jest realizowane przy użyciu wskaźników. Wyrażenia `*w_bok1`, `*w_bok2` reprezentują dane wejściowe — stanowią aliasy nazw zmiennych, odpowiednio, `bok1` i `bok2`. Wyrażenia `*w_pole` i `*w_obwod` odpowiadają zaś, odpowiednio, polu prostokąta — zmiennej `pole`, i obwodowi — zmiennej `obwod`.

Ćwiczenie 4.3

Zmodyfikuj program z przykładu 4.3 — oprócz pola i obwodu prostokąta wyznacz dodatkowo długość jego przekątnej. Ponadto:

- dane wejściowe do programu niech będą wprowadzane z klawiatury,
- nadanie wartości początkowych wskaźników przeprowadź za pomocą inicjalizacji jednolitej.

UWAGA

Szczególnym rodzajem wskaźników są **wskaźniki ogólne** (ang. *generic pointers*, *general-purpose pointers*), tzw. wskaźniki typu `void` (ang. *void pointers*). Wskaźniki typu `void` pozwalają na zapamiętanie i przechowanie adresu zmiennej dowolnego typu (np. adresu zmiennej typu `int` lub `double`), ale zarazem nie są skojarzone z żadnym konkretnym, bazowym typem danych. Tym samym dereferencja wskaźników `void` nie jest możliwa. Na przykład po deklaracji `double zmieniona`; można przypisać adres tej zmiennej (`&zmieniona`) wskaźnikowi typu `void`: `void *wskaźnik = &zmieniona;`. Jednocześnie nie jest możliwe odwołanie do wartości przechowywanej w zmiennej `zmieniona` za pomocą operatora dereferencji `*wskaźnik`.

4.2.3. Wskaźniki i słowo kluczowe `const`

W programowaniu w języku C++ często można się spotkać z deklaracjami wskaźników połączonych ze słowem kluczowym `const`. Mogą wówczas zajść dwa przypadki zależne od położenia słowa kluczowego `const` w deklaracji wskaźnika. Pierwszy z nich zilustrowano w przykładzie 4.4, a drugi — w przykładzie 4.5.

Przykład 4.4

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

int main() {
    // Deklaracja i inicjalizacja zmiennej zmieniona1 typu int:
    int zmieniona1 = 1;
    // Deklaracja wskaźnika o nazwie wskaźnik z modyfikatorem const:
    const int *wskaźnik;
    // Przypisanie wskaźnikowi wskaźnik adresu zmiennej o nazwie zmieniona1:
```

```
wskaznik = &zmienna1;
// Odczyt wartości zmiennej wskazywanej przez wskaźnik wskaznik:
cout << "Wartość zmiennej zmienna1: " << *wskaznik << endl;

// Instrukcja zawarta w komentarzu poniżej jest błędna!
// *wskaznik = 10;

// Deklaracja i inicjalizacja zmiennej zmienna2 typu int:
int zmienna2 = 2;
// Przypisanie wskaźnikowi wskaznik adresu zmiennej zmienna2:
wskaznik = &zmienna2;
// Odczyt wartości zmiennej wskazywanej przez wskaźnik wskaznik:
cout << "Wartość zmiennej zmienna2: " << *wskaznik << endl;

return 0;
}
```

W programie zadeklarowano wskaźnik o nazwie `wskaznik` za pomocą instrukcji `const int *wskaznik;`. W ogólności `wskaznik` może wskazywać na dowolną zmienną typu `int`, ponieważ typ `int` jest jego typem bazowym. Zmiennej `wskaznik` najpierw przypisano adres zmiennej `zmienna1`, a następnie adres zmiennej `zmienna2`.

Modyfikator `const` w deklaracji wskaźnika `wskaznik` powoduje, że wskaźnik ten traktuje zmienną wskazywaną tak, jak gdyby zmienna ta była stałą `const` — pomimo tego, że stałą nie jest. Tym samym za jego pomocą można jedynie odczytać wartość przechowywaną we wskazywanym przez niego bloku pamięci (tutaj: najpierw wartość zmiennej `zmienna1`, a potem `zmienna2`), ale nie jest możliwa modyfikacja (zmiana) tej wartości.

Wskaźnik zadeklarowany w programie zgodnie z postacią ogólną:

```
const typ_bazowy *wskaźnik;
```

jest nazywany **wskaźnikiem na stałą** (ang. *pointer to constant*).

Jak to zademonstrowano w przykładzie 4.4, za pomocą wskaźnika na stałą można wyłącznie odczytywać (ang. *read-only*) wartość zmiennej wskazywanej przez taki wskaźnik. Zapis i/lub modyfikacja wartości zmiennej wskazywanej nie są możliwe. Z drugiej strony we wskaźniku na stałą można zapisywać bez ograniczeń adresy innych zmiennych zgodnych z zadeklarowanym typem bazowym. Zatem taki wskaźnik może wskazywać w programie kolejno różne zmienne.

Ćwiczenie 4.4

Na podstawie przykładu 4.4 napisz program pozwalający obliczyć pole i obwód prostokąta. Długości boków prostokąta mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Wykorzystaj wskaźniki na stałe `const`.

W przykładzie 4.5 zilustrowano z kolei drugi przypadek, jaki może wystąpić w deklaracji wskaźnika z użyciem słowa kluczowego `const`.

Przykład 4.5

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

int main() {
    int zmienna1 = 1;
    int zmienna2 = 2;

    // Deklaracja i inicjalizacja wskaźnika o nazwie wskaznik z wykorzystaniem słowa kluczowego const:
    int *const wskaznik = &zmienna1;
    /* UWAGA
     * Wskaźnik zadeklarowany z użyciem słowa kluczowego const jak powyżej musi zostać zainicjowany
     * podczas deklaracji. Tutaj: wskaźnikowi o nazwie wskaznik przypisano adres zmiennej zmienna1.
     */
    cout << "Wartość zmiennej zmienna1: " << *wskaznik << endl;

    // Modyfikacja wartości zmiennej wskazywanej przez wskaźnik stały:
    (*wskaznik)++;
    cout << "Zmieniona wartość zmiennej zmienna1: " << *wskaznik << endl;

    // Instrukcja w komentarzu poniżej jest błędna!
    // wskaznik = &zmienna2;

    return 0;
}
```

W programie zadeklarowano i jednocześnie zainicjowano zmienną wskaźnikową o nazwie `wskaznik`. W tym celu wykorzystano instrukcję: `int *const wskaznik = &zmienna1;`. Modyfikator `const` sprawia, że zmienna `wskaznik` nosi znamiona stałej. Oznacza to, że zmienna `wskaznik` nie może w programie zmieniać adresu, na który wskazuje. Jeśli przypisano jej wartość początkową adresu określonej zmiennej (tutaj zmiennej `zmienna1`), adres ten nie może się zmienić w dalszej części programu. Tym samym taki wskaźnik nie może wskazywać na inną zmienną. Z drugiej strony przy użyciu wskaźnika stałego można zarówno odczytywać wartość zmiennej wskazywanej, jak i zapisywać (modyfikować) tę wartość.

W ogólności wskaźnik zadeklarowany w programie zgodnie z postacią ogólną:

`typ_bazowy *const wskaźnik;`

jest nazywany **wskaźnikiem stałym** (ang. *constant pointer*).

Wskaźnik stały ma cechę „tylko do odczytu” (ang. *read-only*) w tym znaczeniu, że zmiana jego wartości na inną (tj. przypisanie adresu innej zmiennej) nie jest możliwa. Z drugiej strony wskaźnik stały pozwala zarówno na odczyt, jak i na zmianę (modyfikację) wartości wskazywanej przez niego zmiennej.

UWAGA

W praktyce możliwa jest również kombinacja (połączenie) obu omówionych wcześniej sposobów deklarowania wskaźników wraz ze słowem kluczowym `const`, tj. zadeklarowanie **stałego wskaźnika na stałą** (ang. *constant pointer to constant value*). Na przykład deklaracja: `const int *const wskaznik = &zmienna;` oznacza, że zmienna wskaźnikowa `wskaznik` wskazuje na zmienną, której wartość można wyłącznie odczytywać, i dodatkowo — że wskaźnik ten nie może wskazać w programie na inną zmienną, ponieważ nie można mu przypisać adresu innej zmiennej.

Ćwiczenie 4.5

Na podstawie przykładu 4.5 napisz program pozwalający obliczyć pole i obwód prostokąta. Długości boków prostokąta mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Wykorzystaj wskaźniki stałe.

4.3. Dynamiczna alokacja pamięci

4.3.1. Przydzielanie pamięci operacyjnej dla zmiennych

Zmienne zadeklarowane w programie bez użycia wskaźników (np. `int zmienna;`) mają statycznie przydzieloną pamięć operacyjną na stosie. Wspomniana alokacja pamięci jest realizowana podczas komplikacji programu (ang. *compile-time memory allocation*). W trakcie działania programu wielkość stosu się nie zmienia. Pozostała część pamięci operacyjnej dostępna w czasie działania programu — po przydzielaniu niezbędnej pamięci przeznaczonej na stos — jest nazywana **stertą** (ang. *heap*).

UWAGA

W dziedzinie programowania termin *sterta*, analogicznie do *stosu*, oprócz segmentu (obszaru) w pamięci operacyjnej oznacza dynamiczną strukturę danych.

Sterta, jako część pamięci operacyjnej komputera, może być również wykorzystywana przez zmienne programowe. Przydzielanie (alokacja) pamięci dla zmiennych na stercie odbywa się w sposób dynamiczny w czasie wykonywania programu. Na przykład w trakcie działania programu można — w zależności od potrzeby — utworzyć nową zmienną określonego typu,

wykonać zadane operacje na tej zmiennej, a w chwili gdy zmienna ta nie jest już potrzebna, zwolnić pamięć przydzieloną dla niej w sposób dynamiczny.

Omawiany proces alokacji pamięci operacyjnej komputera dla zmiennych programowych jest nazywany **alokacją pamięci w czasie wykonywania programu** (ang. *runtime memory allocation*) — w odróżnieniu od alokacji pamięci w czasie komplikacji.

Ten sposób przydzielania pamięci w czasie wykonywania programu ze względu na jego charakterystyczną cechę — tworzenie i niszczenie (usuwanie) zmiennych w dowolnej chwili, w zależności od potrzeby — jest nazywany **dynamiczną alokacją pamięci** (ang. *dynamic memory allocation*). Mechanizm dynamicznego przydzielania/zwalniania pamięci na stercie dla zmiennej (zmiennych) jest kontrolowany przez programistę. Jest to przeciwieństwo statycznego przydzielania pamięci (ang. *static memory allocation*), które jest sterowane przez kompilator w czasie komplikacji programu.

Proces tworzenia, przetwarzania oraz zwalniania pamięci dla zmiennych zaalokowanych w pamięci operacyjnej (na stercie) w sposób dynamiczny jest prowadzony przy wykorzystaniu wskaźników.

4.3.2. Operatory new i delete

Operator new służy do utworzenia nowej zmiennej określonego typu. Pamięć operacyjna potrzebna na przechowanie tej zmiennej jest przydzielana (alokowana) dynamicznie na stercie. Jej wielkość (rozmiar) zależy od typu tworzonej zmiennej. Po zaalokowaniu dla zmiennej odpowiedniego bloku (obszaru) pamięci operator new zwraca adres tego bloku.

Ogólna postać wyrażenia umożliwiającego utworzenie nowej zmiennej na stercie jest następująca:

```
wskaźnik = new typ_zmiennej;
```

lub

```
wskaźnik = new typ_zmiennej(wartość);
```

gdzie:

- **wskaźnik** — może wskazywać na dane (zmienne) typu **typ_zmiennej**,
- **typ_zmiennej** — typ nowo tworzonej zmiennej. Typ **typ_zmiennej** jest typem bazowym wskaźnika **wskaźnik**,
- **wartość** — wartość początkowa zmiennej. Jeśli zmiennej nie zostanie przypisana wartość początkowa, zostanie ona zainicjowana wartością przypadkową.

Na przykład utworzenie na stercie zmiennej typu **int** wskazywanej przez wskaźnik o nazwie **wsk** można zrealizować za pomocą instrukcji: **int *wsk; wsk = new int;.**

Zwolnienie pamięci przydzielonej dynamicznie dla zmiennej na stercie jest realizowane za pomocą operatora `delete`. Postać ogólna użycia tego operatora jest następująca:

`delete wskaźnik;`

gdzie `wskaźnik` wskazuje na blok pamięci, w którym została zaalokowana usuwana zmienna.

Na przykład zwolnienie pamięci zaalokowanej dynamicznie dla zmiennej wskazywanej przez wskaźnik `wsk` można zrealizować za pomocą instrukcji: `delete wsk;`. Zwolnienie pamięci operacyjnej przydzielonej dynamicznie dla zmiennej na stercie jest tożsame z usunięciem tej zmiennej (z pamięci). W przypadku gdy pamięć przydzielona dla zmiennej na stercie nie zostaje zwolniona przy użyciu operatora `delete`, zmienna ta blokuje zaalokowany obszar pamięci do czasu, aż działanie programu się zakończy.

Zmienne alokowane dynamicznie na stercie mają pewne szczególne cechy, które odróżniają je od zmiennych deklarowanych w programie bez użycia wskaźników (a które są przechowywane na stosie). Po pierwsze, zmienne przechowywane na stercie nie mają nazw (identyfikatorów). Tym samym można te zmienne przetwarzać wyłącznie przy użyciu wskaźników i operatora dereferencji (`*`). Po drugie, czas życia zmiennych na stercie jest kontrolowany bezpośrednio przez programistę: są tworzone, jeśli jest taka potrzeba — przy użyciu operatora `new`, i usuwane, jeśli nie są już potrzebne — za pomocą operatora `delete`. Im krótszy jest czas życia zmiennych na stercie, tym gospodarowanie pamięcią operacyjną komputera jest skuteczniejsze i wydajniejsze. Trzecią cechą odróżniającą zmienne na stosie od zmiennych na stercie jest ich wartość początkowa. Zmienne na stosie, które zostały zadeklarowane, ale nie zainicjowane, przyjmują wartość początkową domyślną — zerową, np. `0`, `""`. Niezainicjowane zmienne utworzone na stercie przyjmują kolej wartość początkową przypadkową.

UWAGA

W co najmniej dwóch sytuacjach dobrym nawykem programistycznym jest przypisanie wskaźnikowi wartości `NULL`: jeśli wskaźnik został zadeklarowany, ale nie przypisano mu żadnego adresu początkowego (np. `double *wskaznik = NULL;`), i po zwolnieniu pamięci operacyjnej zaalokowanej dla zmiennej na stercie, czyli po zastosowaniu operatora `delete` (np. `delete wskaznik; wskaznik = NULL;`).

Przykład 4.6

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

int main() {
    // Deklaracja zmiennej wskaźnikowej (wskaźnika) w_ocena:
    int *w_ocena;
    // Utworzenie zmiennej typu int wskazywanej przez wskaźnik w_ocena:
    w_ocena = new int;
```

```

// Przypisanie zmiennej wskazywanej przez wskaźnik w_ocena wartości 4:
*w_ocena = 4;
cout << "Ocena: " << *w_ocena << endl;

// Usunięcie zmiennej wskazywanej przez wskaźnik w_ocena:
delete w_ocena;

return 0;
}

```

Zadeklarowany wskaźnik `w_ocena` może wskazywać na dowolną zmienną typu `int`. Jednakże w programie przypisano mu adres konkretnej zmiennej typu `int`, która została utworzona za pomocą operatora `new`. Pamięć operacyjna dla tej zmiennej została zaalokowana dynamicznie na stercie. W operacji przypisania wartości utworzonej zmiennej wykorzystano operator dereferencji: `*w_ocena = 4;`. Po wyświetleniu wartości zmiennej pamięć przydzielona dla niej dynamicznie jest zwalniana za pomocą operatora `delete`.

Ćwiczenie 4.6

Na podstawie przykładu 4.6 napisz program, w którym zamiast pojedynczej oceny rozpatrywane są cztery oceny semestralne ucznia z następujących przedmiotów: język polski, matematyka, język angielski, informatyka. Dane wejściowe do programu — wartości poszczególnych ocen — należy wprowadzić z klawiatury. Oblicz średnią arytmetyczną tych ocen i zaprezentuj wynik na ekranie. W celu zapamiętania ocen ucznia wykorzystaj zmienne zaalokowane w pamięci operacyjnej w sposób dynamiczny.

Przykład 4.7

```

#include <iostream>
using namespace std;

int main() {
    // Deklaracja zmiennej wskaźnikowej (wskaźnika) na zmienną typu double i przypisanie jej wartości NULL:
    double *w_bok1 = NULL;
    // Utworzenie zmiennej (dynamicznej) typu double wskazywanej przez wskaźnik w_bok1:
    w_bok1 = new double(1);
    /* UWAGA
     * Zmienna na stercie, dla której pamięć została zaalokowana dynamicznie, została zainicjowana wartością 1.
     */

    double *w_bok2 = new double(2);
    /* UWAGA
     * W wyrażeniu powyżej następuje deklaracja wskaźnika w_bok2 oraz jego inicjalizacja.
     * Wspomniana inicjalizacja wskaźnika jest realizowana po zaalokowaniu na stercie obszaru pamięci operacyjnej
     * niezbędnego do przechowania danych typu double i zwróceniu (przez operator new) adresu początku tego obszaru.
    */
}

```

* Zwrócony adres zostaje następnie przypisany do wskaźnika `w_bok2`.
 * Zmienna na stercie, dla której pamięć została zaalokowana dynamicznie, została zainicjowana wartością 2.
 */

```
// Utworzenie zmiennej dynamicznej wskazywanej przez wskaźnik w_pole:  

double *w_pole = new double;  

// Obliczenie pola prostokąta — wykorzystanie wskaźników i operatora dereferencji:  

*w_pole = (*w_bok1) * (*w_bok2);  
  

// Utworzenie zmiennej dynamicznej wskazywanej przez wskaźnik w_obwod:  

double *w_obwod = new double;  

// Obliczenie obwodu prostokąta — wykorzystanie wskaźników i operatora dereferencji:  

*w_obwod = 2 * (*w_bok1) + 2 * (*w_bok2);  
  

// Zwolnienie pamięci zaalokowanej dla zmiennych wskazywanych przez wskaźniki w_bok1 i w_bok2:  

delete w_bok1;  

delete w_bok2;  
  

// Prezentacja wyników — wykorzystanie wskaźników i operatora dereferencji:  

cout << "Wyniki:" << endl;  

cout << "Pole wynosi " << *w_pole << endl;  

cout << "Obwód wynosi " << *w_obwod << endl;  
  

// Zwolnienie pamięci zaalokowanej dla zmiennych wskazywanych przez wskaźniki w_pole i w_obwod:  

delete w_pole;  

delete w_obwod;  
  

return 0;  
}
```

W przykładzie 4.7 obliczane są pole i obwód prostokąta. Dane wejściowe (długości boków) są przechowywane w zmiennych przechowywanych na stercie, dla których pamięć operacyjna została zaalokowana dynamicznie. Dostęp do tych zmiennych uzyskuje się za pomocą wskaźników, odpowiednio, `w_bok1` i `w_bok2`. Długości boków zostały zainicjowane w programie wartościami, odpowiednio, 1 i 2. Pole i obwód prostokąta również są w programie reprezentowane przez zmienne utworzone na stercie, dla których pamięć została przydzielona dynamicznie. Zmienne te są wskazywane przez wskaźniki `w_pole` (pole prostokąta) i `w_obwod` (obwód prostokąta). Przetwarzanie tych zmiennych odbywa się przy wykorzystaniu wyrażeń z operatorem dereferencji `*`, tj. `*w_pole` oraz `*w_obwod`.

Ćwiczenie 4.7

Zmodyfikuj program zawarty w przykładzie 4.7 — zamiast pola i obwodu prostokąta oblicz pole i obwód kwadratu.

**UWAGA**

Ze wskaźnikami związany jest mechanizm tzw. **arytmetyki wskaźników** (ang. *pointer arithmetics*). Pomimo że dotyczy on (m.in.) danych typów podstawowych, takich jak `int` czy `float`, szczególną rolę odgrywa w przetwarzaniu tablic. Dlatego ten temat został omówiony w dalszej części podręcznika — w rozdziale 5., dotyczącym tablic.

**UWAGA**

Wskaźniki często są parametrami i argumentami funkcji. Ta tematyka została szczegółowo omówiona w rozdziale 8., dotyczącym funkcji.



4.4. Pytania i zadania kontrolne

4.4.1. Pytania

- 1.** Czy adres określonej zmiennej w pamięci operacyjnej oznacza to samo, co wskaźnik na tę zmienną?
- 2.** Podaj definicję wskaźnika.
- 3.** Czym różni się alokacja pamięci operacyjnej dla zmiennych na stosie od dynamicznej alokacji pamięci dla zmiennych na stercie?
- 4.** Czy zmienne, dla których pamięć jest przydzielana dynamicznie, mają nazwy?
- 5.** W jaki sposób można odwoływać się do zadeklarowanych zmiennych przechowywanych na stosie?
- 6.** Czy słowo kluczowe `const` może występować w deklaracji wskaźnika? Jeśli tak, to jaki to ma wpływ na cechy tego wskaźnika?

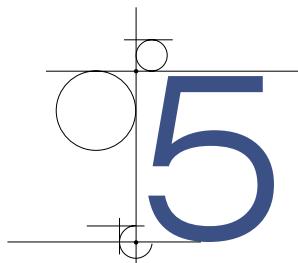
4.4.2. Zadania

- 1.** Napisz program umożliwiający obliczenie pola powierzchni i obwodu koła. Dane wejściowe mają być wprowadzane z klawiatury. Wyniki zaprezentuj na ekranie monitora w konsoli.

Dane wejściowe (promień koła) oraz wyniki (pole i obwód koła) program powinien zapamiętywać w zmiennych zadeklarowanych w sposób jawnny — przechowywanych w pamięci operacyjnej na stosie. Przetwarzanie danych (obliczenia) zrealizuj przy wykorzystaniu wskaźników i operatora dereferencji.

- 2.** Zrób tak samo jak w zadaniu 1., z tym że promień oraz pole i obwód koła powinny być zapamiętywane w zmiennych zaalokowanych w pamięci operacyjnej w sposób dynamiczny (na stercie).

3. Napisz program umożliwiający obliczenie objętości, pola powierzchni bocznej oraz długości wszystkich krawędzi prostopadłościanu. Dane wejściowe zainicjuj w programie. Wyniki niech będą wyświetlane w konsoli. Przetwarzanie danych wykonaj przy użyciu wskaźników do jawnie zadeklarowanych zmiennych przechowywanych w pamięci operacyjnej na stosie oraz operatora dereferencji.
4. Zrób tak samo jak w zadaniu 3., z tym że do przechowania danych wejściowych i wyników obliczeń wykorzystaj zmienne, dla których pamięć operacyjna została zaalokowana dynamicznie na stercie.
5. Napisz program pozwalający sprawdzić, czy liczba całkowita o wartości wprowadzonej z klawiatury jest parzysta, czy nieparzysta. Wykorzystaj zmienne, dla których pamięć operacyjna jest przydzielana dynamicznie.
6. Napisz program — prosty kalkulator logiczny — pozwalający obliczyć iloczyn i sumę logiczną dla dwóch danych logicznych o wartościach wprowadzonych z klawiatury. Przyjmij, że wartość danej wejściowej 1 reprezentuje wartość logiczną true, natomiast 0 — false. Wykorzystaj:
 - wskaźniki do stałych zadeklarowane jako const int* (wariant pierwszy),
 - wskaźniki stałe zadeklarowane jako int *const (wariant drugi).



Tablice i wektory

5.1. Tablice statyczne

W programowaniu często jest konieczne przechowywanie i przetwarzanie wielu danych należących do tego samego typu, np. liczb całkowitych lub rzeczywistych. Na przykład obliczenie średniej pensji miesięcznej pracownika w dużym zakładzie pracy (w którym jest zatrudnionych np. tysiąc osób) będzie wymagało m.in. dodania do siebie pensji wszystkich pracowników. Innym przykładem może być potrzeba przechowywania (a następnie ewentualnego przetwarzania) wartości temperatury w serwerowni w określonym miesiącu przy założeniu, że próbki temperatury są pobierane i zapisywane co 30 minut.

Problemy tego rodzaju łatwo rozwiązać przy wykorzystaniu **tablic** (ang. *arrays*). Tablica stanowi skończony, uporządkowany ciąg elementów należących do tego samego typu. Ze względu na budowę wewnętrzną tablice są zaliczane do **typów danych złożonych** (ang. *compound data types*).

Każda tablica zajmuje ciągły obszar pamięci operacyjnej komputera, niezbędny do zapamiętania jej wszystkich elementów składowych. Jest to bardzo ważna właściwość tablic, ponieważ pozwala na ich łatwe i wydajne przetwarzanie z użyciem pętli programowych i wskaźników.

UWAGA

Tablice mają swoje odpowiedniki w matematyce (dokładnie: w algebrze liniowej), tzw. macierze (ang. *matrices*).

Tablice można podzielić w zależności od liczby wymiarów na:

- **tablice jednowymiarowe** (ang. *one-dimensional arrays*),
- **tablice wielowymiarowe** (ang. *multidimensional arrays*).

W praktyce najczęściej wykorzystywane są tablice jednowymiarowe i dwuwymiarowe.

Tablice są zaliczane do tzw. **typów danych agregacyjnych** (ang. *aggregate data types*), ponieważ pozwalają na zgrupowanie wielu oddzielnych danych w jednym kontenerze. Wspomniane dane są reprezentowane w tablicy przez jej elementy składowe należące do tego samego typu. Dane w tablicy mogą być od siebie w pełni niezależne. Na przykład tablica może zawierać wykaz ocen semestralnych ucznia np. z języka polskiego, języka angielskiego, matematyki i informatyki, które są autonomiczne — niezależne jedna od drugiej.

5.1.1. Deklarowanie tablic

Ogólna postać deklaracji tablicy jednowymiarowej (jako zmiennej) jest następująca:

```
typ_składowy nazwa_tablicy[liczba_elementów];
```

gdzie:

- **typ_składowy** oznacza typ danych, do którego należą elementy składowe tablicy,
- **nazwa_tablicy** to nazwa (identyfikator) deklarowanej zmiennej tablicowej,
- **liczba_elementów** to liczba elementów składowych tablicy.

Na przykład deklaracja tablicy o nazwie **tablica** składającej się z 10 elementów składowych należących do typu całkowitego ma postać: **int tablica[10];**.

Wszystkie elementy składowe tablicy muszą należeć do tego samego typu. Przy czym może to być zarówno typ podstawowy (ang. *fundamental data type*) — np. **float**, **int**, jak i typ pochodny (ang. *derived data type*) — np. typ wskaźnikowy.

Liczba elementów tablicy musi być stałą lub wyrażeniem dającym stałą. Liczba elementów tablicy określa jej **rozmiar** (ang. *size*). W ogólności rozmiar tablicy jest dowolny — ograniczony jedynie pojemnością dostępnego ciągłego obszaru pamięci.

UWAGA

Niektórzy programiści i teoretycy programowania zaliczają tablice do typów danych pochodzących, ponieważ buduje się je na podstawie zadanego (bazowego) typu elementów składowych. Inna grupa programistów traktuje tablice jako typ danych definiowany przez użytkownika, co wynika z konieczności samodzielnego wyboru zarówno typu elementów składowych tablicy, jak i jej rozmiaru. Biorąc pod uwagę powyższe, argumentacja obu grup jest racjonalna i uzasadniona.

5.1.2. Odwołanie się do elementu składowego tablicy

Każdy element składowy tablicy ma swój unikatowy numer porządkowy nazywany **indeksem** (ang. *index*), który określa jednoznacznie jego położenie w tablicy. Indeksy są reprezentowane przez kolejne liczby całkowite począwszy od zera: indeks pierwszego elementu tablicy wynosi 0, drugiego — 1, trzeciego — 2 itd. Indeks elementu składowego tablicy pozwala na

indywidualne odwołanie się do tego elementu w programie. Stanowi zatem jego jednoznaczny identyfikator.

Ogólna postać odwołania do elementu składowego tablicy jednowymiarowej jest następująca:

nazwa_tablicy[indeks_elementu]

gdzie:

- nazwa_tablicy to identyfikator zmiennej tablicowej,
- indeks_elementu jest wyrażeniem całkowitym określającym indeks tego elementu.

Operator [], zapewniający dostęp do zadanego elementu tablicy, jest nazywany **operatorem indeksowym** (lub **operatorem indeksowania**) (ang. *index operator*).

Przykład 5.1

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

int main() {
    const int n = 5; // rozmiar tablicy

    // Deklaracja zmiennej tablicowej o nazwie tablica:
    float tablica[n];

    cout << "Podaj wartości elementów składowych tablicy:" << endl;
    // Pobranie wartości poszczególnych elementów składowych tablicy z klawiatury:
    for (int i = 0; i < n; i++) {
        cout << "tablica[" << i << "] = ";
        cin >> tablica[i];
    }

    cout << "Wprowadzone wartości elementów tablicy:" << endl;
    // Wyświetlenie w konsoli na ekranie monitora wartości poszczególnych elementów tablicy:
    for (int i = 0; i < n; i++) {
        cout << "tablica[" << i << "] = " << tablica[i] << endl;
    }

    return 0;
}
```

W programie zadeklarowano zmienną tablicową — tablicę o nazwie tablica. Tablica ta zawiera 5 elementów składowych należących do typu rzeczywistego float. Wartości elementów składowych tablicy tablica wprowadzono z klawiatury przy użyciu pętli for i odwołania o postaci tablica[i], gdzie i, jako zmienna sterująca pętlą, zmienia się w poszczególnych iteracjach od wartości 0 do wartości 4 (n-1). Należy zwrócić uwagę, że wyrażenie tablica[i]

należy do typu `float`. Wyświetlenie na ekranie monitora wprowadzonych wcześniej wartości elementów tablicy zrealizowano w analogiczny sposób.

Ćwiczenie 5.1

Zmodyfikuj program z przykładu 5.1 — dopisz kod pozwalający obliczyć sumę elementów zapisanych w tablicy `tablica`.

Tablice, jako zmienne zadeklarowane w programie w sposób jawnym, są przechowywane w pamięci operacyjnej w obszarze nazywanym stosem (ang. *stack*) — o którym była mowa w rozdziale 4.

Wielkość pamięci zarezerwowanej dla tablicy na stosie jest ściśle określona — ustalona przez kompilator już na etapie komplikacji programu. Ten proces jest nazywany alokacją pamięci w czasie komplikacji (ang. *compile-time memory allocation*). Wielkość tej pamięci nie zmienia się w czasie wykonywania programu. Dlatego też tablice, o których mowa w tym podrozdziale, są nazywane **tablicami deklarowanymi statycznie** (ang. *static declared arrays*) albo po prostu **tablicami statycznymi** (ang. *static arrays*). W odniesieniu do tablic statycznych używane jest jeszcze jedno określenie: **tablice stałe** (ang. *fixed arrays*).

Z punktu widzenia ogólnych zasad programowania alokacja — przydział pamięci w czasie komplikacji — jest również nazywana **statyczną alokacją pamięci** (ang. *static memory allocation*).

Z tego, że dla tablic zadeklarowanych jawnie (czyli tablic statycznych) pamięć operacyjna jest alokowana już w czasie komplikacji programu, wynikają ich dwie bardzo ważne właściwości. Po pierwsze, rozmiar tablicy statycznej (tablicy stałej) nie może się zmieniać w trakcie wykonywania programu. Tym samym liczba elementów składowych takiej tablicy również nie może się zmieniać w czasie działania programu. Po drugie, ustalenie rozmiaru tablicy statycznej w jej deklaracji nie może się opierać na wartościach zmiennych programowych (ang. *program variables*), lecz jedynie na stałych (dokładnie: wyrażeniach dających w wyniku stałe). Dlatego też rozmiar tablicy statycznej nie może być ustalony np. na podstawie wartości jakiekolwiek danej wejściowej, która z założenia musi być zmienną.

Jeśli spojrzeć z perspektywy ogólnych zasad programowania, niezależnych od języka (w tym języka C++), tablice, jako zmienne, dla których pamięć jest alokowana statycznie, nazywa się **zmiennymi statycznymi** (ang. *static variables*).

UWAGA

Więcej informacji o zmiennych statycznych można znaleźć w podrozdziale 8.4.1, dotyczącym funkcji, oraz w podrozdziale 11.5, w którym są omawiane pola i metody statyczne definiowane w klasach.



UWAGA

Niepoprawne odwołanie się do elementu składowego tablicy, polegające na użyciu nieistniejącego indeksu (indeksu o wartości spoza dopuszczalnego zakresu indeksów), nie jest kontrolowane przez kompilator. Tym samym może to spowodować nieprzewidziane i niepożądane skutki, np. uzyskanie błędnych wyników obliczeń, nadpisanie innej zmiennej przechowywanej w pamięci czy zatrzymanie programu. Zatem programista powinien starannie kontrolować zakres używanych indeksów, zwłaszcza w połączeniu z wykorzystaniem indeksów tablic w pętlach programowych (np. jako zmiennych sterujących pętli).

5.1.3. Inicjowanie tablic

Podczas deklarowania tablicy w programie głównym (tj. wewnętrz — w ciele funkcji `main()`) kompilator domyślnie nie inicjuje wartości jej elementów składowych. Elementy składowe tablicy pozostają nieokreślone. Inaczej jest, jeśli tablica jest deklarowana na zewnątrz funkcji `main()` — bezpośrednio w przestrzeni nazw. W tym przypadku, nawet jeśli zadeklarowana tablica nie została zainicjowana w sposób jawny, jej elementy składowe zostaną zainicjowane automatycznie (po „cichu”) wartościami domyślnymi — „zerowymi”, np. `0`, `""`.



UWAGA

Skutki deklaracji tablic wewnętrz i na zewnątrz funkcji `main()` związane z inicjalizacją ich elementów składowych obowiązują również dla innych funkcji. Ta tematyka została omówiona w rozdziale 8.

Analogicznie jak w przypadku zmiennych należących do typów podstawowych, deklarację tablicy można połączyć z jej inicjalizacją. Można to zrealizować przy wykorzystaniu jednej z następujących postaci ogólnych:

`typ_składowy nazwa_tablicy[liczba_elementów] = {lista_wyrażeń};` (1)

lub

`typ_składowy nazwa_tablicy[liczba_elementów] {lista_wyrażeń};` (2)

lub

`typ_składowy nazwa_tablicy[] = {lista_wyrażeń};` (3)

lub

`typ_składowy nazwa_tablicy[] = {lista_wyrażeń};` (4)

gdzie:

- `typ_składowy` i `liczba_elementów` mają takie samo znaczenie jak w deklaracji tablicy,

- lista_wyrażeń to ciąg wyrażeń oddzielonych od siebie przecinkami, które dają w wyniku wartości stałe należące do typu typ_składowy, określające wartości początkowe elementów składowych tablicy.

Jeżeli w deklaracji tablicy połączonej z jej inicjalizacją w postaci (1) lub (2):

- liczba zadeklarowanych elementów tablicy jest większa od liczby podanych wartości początkowych (np. `int tablica[10] = {1, 2, 3, 4, 5};`), to jej pozostałe (nieuwzględnione) elementy składowe zostaną wypełnione zerami (lub wartościami pustymi),
- jeżeli lista_wyrażeń jest pusta (np. `int tablica[5] = {};`), to wszystkie elementy składowe tablicy zostaną wypełnione wartościami domyślnymi — „zerowymi”, np. 0, "".

Jeżeli w deklaracji tablicy nie została podana liczba jej elementów składowych — jak w (3) i (4), to rozmiar tej tablicy jest ustalany przez kompilator automatycznie na podstawie liczby wyrażeń określających wartości początkowe jej elementów składowych.

Przykład 5.2

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

// Deklaracja stałej o nazwie stalaPom:
const float stalaPom = 100;

// Deklaracja i inicjalizacja zmiennej globalnej zmiennaPom:
float zmiennaPom = 100;

// Deklaracja tablicy o nazwie tablica na zewnątrz funkcji main():
float tablica[5];
/* UWAGA
* Domyślnie wszystkie elementy składowe tablicy tablica zostaną zainicjowane w sposób niejawny
* wartościami zerowymi.
*/

int main() {
    // Wyświetlenie zawartości tablicy tablica zadeklarowanej na zewnątrz funkcji main():
    for (int i = 0; i < 5; i++) {
        cout << "tablica[" << i << "] = " << tablica[i] << endl;
    }
    cout << endl;

    float tablica1[5] = {1, 2, 3, 4, 5 + zmiennaPom - stalaPom};
    /* UWAGA
* Pierwsze cztery elementy składowe tablicy tablica1 zostały zainicjowane za pomocą literałów.
* Z kolei ostatni, piąty element został zainicjowany wartością wyrażenia zawierającego stałą i zmienną.
```

```

*/
for (int i = 0; i < 5; i++) {
    cout << "tablica1[" << i << "] = " << tablica1[i] << endl;
}
cout << endl;

float tablica2[10] = {1, 2, 3, 4, 5};
/* UWAGA
 * Liczba zadeklarowanych elementów składowych tablicy tablica2 jest większa od liczby wartości początkowych
 * jej elementów składowych zainicjowanych w sposób jawny.
 */
for (int i = 0; i < 10; i++) {
    cout << "tablica2[" << i << "] = " << tablica2[i] << endl;
}
cout << endl;

float tablica3[] = {1, 2, 3, 4, 5};
/* UWAGA
 * W deklaracji tablicy tablica3 nie podano liczby jej elementów. Dlatego też rozmiar tej tablicy zostanie ustalony
 * automatycznie na podstawie liczby wartości początkowych jej elementów składowych zainicjowanych
 * w sposób jawny.
 */
for (int i = 0; i < 5; i++) {
    cout << "tablica3[" << i << "] = " << tablica3[i] << endl;
}
cout << endl;

float tablica4[] {1, 2, 3, 4, 5}; // C++11
for (int i = 0; i < 5; i++) {
    cout << "tablica4[" << i << "] = " << tablica4[i] << endl;
}
cout << endl;

float tablica0a[5] = {};
/* UWAGA
 * Wszystkie elementy składowe tablicy tablica0a zostały zainicjowane wartościami 0.
 */
for (int i = 0; i < 5; i++) {
    cout << "tablica0a[" << i << "] = " << tablica0a[i] << endl;
}
cout << endl;

```

```

float tablica0b[5] {}; // C++11
for (int i = 0; i < 5; i++) {
    cout << "tablica0b[" << i << "] = " << tablica0b[i] << endl;
}

return 0;
}

```

W programie dla celów porównawczych wykorzystano wszystkie omówione sposoby deklarowania tablic połączone z inicjalizacją ich elementów składowych. Uwzględniono zarówno sposoby charakterystyczne dla wersji C++98, jak i te, które wprowadzono w wersji C++11.

Ćwiczenie 5.2

Na podstawie kodu zawartego w przykładzie 5.2 napisz program pozwalający obliczyć sumę elementów składowych jednowymiarowej tablicy rzeczywistej. Załącz, że rozmiar tablicy wynosi 10. Elementy składowe tablicy należy zainicjować w kodzie źródłowym programu. Wynik wyświetli na ekranie monitora.

Wykonaj program w dwóch wariantach, w zależności od sposobu inicjalizacji elementów tablicy:

- w stylu C++98 (wariant pierwszy),
- w stylu C++11 (wariant drugi).

5.1.4. Tablice wielowymiarowe

Idea konstrukcji tablicy wielowymiarowej (ang. *multidimensional array*) jest oparta na budowie tablicy jednowymiarowej.

Elementy składowe tablicy jednowymiarowej należą do dowolnego typu danych, np. `int` czy `float`. Zasadniczo może to być dowolny typ danych — w tym typ pochodny, np. inna tablica. Po przyjęciu założenia, że elementami składowymi tablicy jednowymiarowej są inne tablice jednowymiarowe, w rezultacie powstanie tablica dwuwymiarowa. Idąc dalej tym tokiem rozumowania, jeżeli elementami składowymi tablicy jednowymiarowej będą uzyskane wcześniej tablice dwuwymiarowe, w efekcie otrzymamy tablice trójwymiarowe.

W ogólności liczba wymiarów tablicy jest dowolna. Jednakże należy pamiętać o tym, że pojemność pamięci operacyjnej potrzebnej do przechowania dużej tablicy wielowymiarowej może być znaczna. Ograniczeniem jest również wymaganie, że musi to być ciągły obszar pamięci, w dodatku zaalokowanej (na stosie) już na etapie kompilacji programu.

Deklaracja tablicy

Ogólna postać deklaracji tablicy dwuwymiarowej jest następująca:

```
typ_składowy nazwa_tablicy[liczba_wierszy][liczba_kolumn];
```

gdzie:

- **typ_składowy** i **nazwa_tablicy** mają takie samo znaczenie jak w deklaracji tablicy prostej — jednowymiarowej,
- **liczba_wierszy** oznacza liczbę wierszy w tablicy, a **liczba_kolumn** — liczbę kolumn. W ogólności są to wyrażenia całkowite dające w rezultacie wartości stałe.

Na przykład deklaracja tablicy 2-wymiarowej o nazwie **tablica** złożonej z 3 wierszy i 2 kolumn i zawierającej elementy składowe należące do typu **float** ma postać: **float tablica[3][2];**.



UWAGA

To, że pierwszy wymiar w deklaracji tablicy dwuwymiarowej przyjmuje się za reprezentację wierszy, a drugiego — za reprezentację kolumn, jest w pełni umowne. Można byłoby przyjąć odwrotne założenie i wówczas pierwszy wymiar odpowiadałby kolumnom, a drugi wierszom.

Deklaracja tablicy trójwymiarowej powinna zawierać określenie rozmiaru w każdym z trzech wymiarów, np. **float tablica[3][2][4];**. W ogólności w deklaracji tablicy wielowymiarowej należy określić jej rozmiary z uwzględnieniem każdego wymiaru z osobna.

Odwołanie się do elementu tablicy

Odwołanie do elementu składowego tablicy dwuwymiarowej ma postać ogólną:

```
nazwa_tablicy[numer_wiersza][numer_kolumny];
```

gdzie **numer_wiersza** i **numer_kolumny** oznaczają, odpowiednio, indeksy elementów w pierwszym i drugim wymiarze, przy czym te indeksy to kolejne liczby całkowite począwszy od liczby 0.

Na przykład odwołanie do elementu tablicy **tablica** o indeksie „wierszowym” 2 i „kolumnowym” 0 ma postać: **tablica[2][0]**. Element ten jest położony na „przecięciu” trzeciego wiersza i pierwszej kolumny.

Przykład 5.3

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

int main() {
    const int m = 3; // liczba wierszy w tablicy
    const int n = 2; // liczba kolumn w tablicy
```

```

// Deklaracja tablicy 2-wymiarowej o nazwie tablica:
int tablica [m][n];

// Nadanie wartości początkowych elementom składowym tablicy:
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
        tablica[i][j] = i + j;
    }

// Wyświetlenie wartości poszczególnych elementów tablicy:
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
        cout << "Element[" << i << ", " << j << "] = " << tablica[i][j] << endl;
    }

// Wyznaczenie najmniejszej i największej wartości zapisanej w tablicy:
int minimum, maksimum;
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
        if ((i == 0) && (j == 0)) {
            minimum = tablica[i][j];
            maksimum = tablica[i][j];
        } else {
            if (tablica[i][j] > maksimum) maksimum = tablica[i][j];
            if (tablica[i][j] < minimum) minimum = tablica[i][j];
        }
    }

// Prezentacja wyników na ekranie monitora:
cout << "Najmniejsza wartość zapisana w tablicy: " << minimum << endl;
cout << "Największa wartość zapisana w tablicy: " << maksimum << endl;

return 0;
}

```

W programie wykorzystano tablicę 2-wymiarową o nazwie `tablica`, która ma 3 wiersze i 2 kolumny. Elementy składowe tablicy `tablica` należą do typu `int`. Deklaracja tablicy nie zapewnia inicjalizacji jej elementów składowych. Stąd kolejne linie kodu zawierają instrukcje pozwalające nadać wartości początkowe poszczególnym elementom składowym tablicy. W tym celu wykorzystywane są dwie pętle `for` — jedna zagnieżdzona w drugiej. Zewnętrzna pętla `for` umożliwia poruszanie się po wierszach tablicy, a wewnętrzna — po kolumnach. Indeksy tablicy są reprezentowane przez zmienne sterujące pętlą: `i` oraz `j`. Zmienna sterująca `i` odpowiada wierszom tablicy (zewnętrzna pętla `for`), `j` zaś — kolumnom (pętla wewnętrzna).

Po inicjalizacji elementów składowych tablicy są one prezentowane w konsoli na ekranie monitora.

Kolejne instrukcje kodu odpowiadają za wyznaczenie wartości największej (maksimum) i najmniejszej (minimum) z elementów zapisanych w tablicy. Analogicznie jak przy wprowadzaniu wartości elementów z klawiatury, wykorzystywane są zagnieżdżone pętle `for`. Wartości minimum i maksimum są ustalane przy użyciu instrukcji warunkowych `if`.

Ćwiczenie 5.3

Zmodyfikuj program zawarty w przykładzie 5.3 — oblicz dodatkowo sumę elementów zapisanych w tablicy `tablica` oraz ich średnią arytmetyczną.

Inicjalizacja tablicy

Inicjowanie tablic wielowymiarowych jest realizowane tymi samymi sposobami — i zgodnie z tymi samymi zasadami — które zostały przedstawione wcześniej w odniesieniu do tablic jednowymiarowych. Przy tym wykorzystywane są reguły konstruowania tablic wielowymiarowych jako „złożenia” innych tablic.

Przykład 5.4

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

int main() {
    const int m {3}; // liczba wierszy
    const int n {2}; // liczba kolumn

    // Deklaracja tablicy 2-wymiarowej tablica1 połączona z inicjalizacją jej elementów składowych:
    int tablica1 [m][n] {{1, 2}, {3, 4}, {5, 6}}; // C++11

    // Wyświetlenie wartości elementów tablicy tablica1:
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++) {
            cout << "tablica1 [" << i << ", " << j << "] = "
                << tablica1[i][j] << endl;
    }
    cout << endl;

    // Deklaracja i inicjalizacja elementów składowych tablicy 2-wymiarowej tablica2:
    int tablica2 [m][n] {1, 2, 3, 4, 5, 6}; // C++11
    /* UWAGA
     * Z formalnego punktu widzenia tablicę wielowymiarową można również zainicjować w sposób
     * przedstawiony powyżej. Jednakże jest to sposób wykorzystywany raczej w przypadku tablic
     * jednowymiarowych — niezalecany dla tablic wielowymiarowych.
     */
}
```

```
// Wyświetlenie wartości elementów tablicy tablica2:
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
        cout << "tablica2 [" << i << ", " << j << "] = "
            << tablica2[i][j] << endl;
    }

    return 0;
}
```

W programie wykorzystano dwie tablice dwuwymiarowe: `tablica1` i `tablica2`. Pierwsza z nich została zainicjowana w sposób standardowy, zalecany dla tablic wielowymiarowych. Forma zapisu wartości początkowych tablicy `tablica1` wynika bezpośrednio z jej budowy, jako tablicy 1-wymiarowej 3-elementowej, której elementy składowe stanowią tablice 1-wymiarowe 2-elementowe.

Z kolei druga tablica, `tablica2`, jest inicjowana w sposób charakterystyczny dla tablic jednowymiarowych. Jest to sposób niezalecany ze względu na brak przejrzystości i duże prawdopodobieństwo popełnienia błędu w razie konieczności przetwarzania tablicy o złożonej budowie.

Ćwiczenie 5.4

Zmodyfikuj program z przykładu 5.4 — inicjalizację elementów składowych tablic `tablica1` i `tablica2` przeprowadź w sposób zgodny ze standardem C++98, zamiast użyć formy wprowadzonej w wersji C++11.

5.2. Tablice i wskaźniki

Definicja tablicy w języku C++ informuje, że zawiera ona uporządkowany, skończony ciąg elementów należących do tego samego typu. Tym samym każdy element składowy tablicy zajmuje w pamięci operacyjnej blok o identycznej pojemności (rozmiarze). To bardzo ważna właściwość tablic w C++. Drugą istotną cechą tablic w C++ jest to, że stanowią one zmienne, które w pamięci operacyjnej zajmują obszar ciągły, pozbawiony jakichkolwiek luk (niespójności). To oznacza, że pomiędzy kolejnymi elementami składowymi tablicy nie są przechowywane żadne inne zmienne, w tym zmienne należące do tego samego typu, co elementy składowe tablicy.

Wymienione powyżej właściwości tablic w języku C++ pozwalają na ich łatwe, a zarazem skuteczne i wydajne przetwarzanie przy wykorzystaniu wskaźników.

W języku C++ obowiązuje zasada, że nazwa zmiennej tablicowej jest wskaźnikiem na jej pierwszy element składowy. Powodem takiego stanu rzeczy jest to, że kompilator „po cichu” (niejawnie) konwertuje tablicę na wskaźnik do jej elementu składowego o indeksie 0. Na przykład, jeżeli w programie zadeklarowano tablicę o nazwie `tablica: float tablica[10];`,

to w tym przypadku tablica `tablica` typu `float[10]` została przekonwertowana na wskaźnik `tablica` należący do typu `float*`. Zakładając, że dalej w programie zadeklarowano wskaźnik `wsk: float *wsk;`, przypisanie `wsk = tablica` oznacza skopiowanie do wskaźnika `wsk` wartości zapisanej we wskaźniku `tablica` — co jest równoważne przypisaniu do wskaźnika `wsk` adresu elementu tablicy `tablica` o indeksie `0`, czyli `wsk = &tablica[0]`.

Jeżeli wskaźnik `wsk` wskazuje na element tablicy o indeksie `i`, czyli `wsk == &tablica[i]`, to dodanie do wskaźnika liczby `1` powoduje, że wyrażenie `wsk + 1` wskazuje na następny, tj. $(i + 1)$ element tablicy. Ogólnie dodanie do wskaźnika liczby `k` ($k > 0$) sprawia, że wyrażenie `wsk + k` wskazuje na element tablicy o indeksie `i + k`. Następuje przy tym **automatyczne skalowanie** (ang. *automatic scaling*), uwzględniające w przypadku ogólnym rozmiar zmiennej wskazywanej przez wskaźnik, a w przypadku tablic — rozmiar elementu składowego tablicy. Wspomniane skalowanie polega w szczególności na każdorazowym wykonywaniu przez kompilator operacji mnożenia rozmiaru elementu tablicy przez liczbę, która jest dodawana do wskaźnika. Na przykład dodanie liczby `2` do wskaźnika `wsk: wsk + 2` spowoduje automatyczne skalowanie polegające na pomnożeniu rozmiaru elementu tablicy `tablica` należącego do typu `float` (4 bajty) przez `2`, co daje w rezultacie liczbę `8` (bajtów). Tym samym wskaźnik `wsk + 2` będzie wskazywał na element tablicy `tablica` przesunięty w kierunku jej końca o dwa elementy (pozycje).

W analizie powyżej wskaźnik `wsk` pozostaje „nieruchomy”, ponieważ zawsze wskazuje na element tablicy o indeksie `i`. Można to jednak zmienić i przeprowadzić analogiczną — równoważną — analizę, w której wskaźnik „porusza się” po tablicy i wskazuje na jej wybrane elementy składowe. Mianowicie, zamiast dodawać do wskaźnika liczbę `1` (tj. `wsk + 1`), można przecież zmienić jego wartość o `1` za pomocą operatora inkrementacji `wsk++` — co jest równoważne przypisaniu `wsk = wsk + 1`. Po wykonaniu takiej instrukcji wskaźnik `wsk` wskazuje na następny element w tablicy, czyli element o indeksie `i + 1`. Ogólnie wykonanie instrukcji przypisania `wsk += k` (czyli `wsk = wsk + k`) spowoduje przesunięcie wskaźnika `wsk` w tablicy o `k` elementów „do przodu”, tj. w kierunku końca tablicy. Tym samym wskaźnik `wsk` będzie wskazywał na element tablicy o indeksie `i + k`.

W ogólności oprócz dodawania do wskaźników liczb całkowitych można od nich odejmować liczby całkowite, jak też porównywać wartości przechowywane we wskaźnikach wskazujących na określone elementy składowe tablicy. Operacje arytmetyczne na wskaźnikach, takie jak te, nazywa się **arytmetyką wskaźnika** (ang. *pointer arithmetic*).

Jeśli uwzględni się przedstawione reguły, widać, że przetwarzanie tablic można realizować nie tylko za pomocą indeksowania (tj. przy wykorzystaniu operatora indeksowego `[]`), ale również z użyciem reguł arytmetyki wskaźnika i operatora dereferencji `*`.

W tabeli 5.1 zaprezentowano zestaw najważniejszych praktycznych zasad umożliwiających wykonywanie operacji na tablicach z zastosowaniem arytmetyki wskaźnika.

Tabela 5.1. Reguły arytmetyki wskaźnika dotyczące tablic jednowymiarowych

Założenia	Wyrażenie	Wyrażenie równoważne
float tablica[10]; float *wsk; wsk = tablica;	&tablica[i]	wsk + i
	tablica[i]	*(wsk + i)
	tablica[i]	wsk[i]

Przykład 5.5

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

int main() {
    const int n = 5; // rozmiar tablicy

    // Deklaracja zmiennej tablicowej o nazwie tablica zawierającej elementy typu double o rozmiarze n (=5).
    double tablica[n];
    /* UWAGA
     * Zmienna tablica jest przechowywana na stosie. Alokacja pamięci na stosie odbywa się automatycznie,
     * ponieważ kompilator już w czasie kompilacji określa, jaką pamięć należy zarezerwować dla każdej zmiennej
     * zadeklarowanej w programie.
     */

    // Deklaracja wskaźnika na dane typu double:
    double *wsk;

    // Przypisanie wskaźnikowi adresu pierwszego elementu tablicy:
    wsk = tablica;
    /* UWAGA
     * W nazwie zmiennej tablicowej pamiętany jest adres pierwszego elementu, czyli elementu o indeksie 0.
     */

    cout << "Podaj wartości " << n << " elementów tablicy:" << endl;
    for (int i = 0; i < n; i++) {
        cout << " tablica[" << i << "] = ";
        cin >> *(wsk + i); // równoważne instrukcji: cin >> tablica[i];
    }

    // Deklaracja i inicjalizacja zmiennej suma reprezentującej sumę wartości elementów tablicy:
    double suma = 0;
    // Obliczenie sumy elementów zapisanych w tablicy:
    for (int i = 0; i < n; i++) {
        suma += wsk[i]; // równoważne instrukcji: suma += tablica[i];
    }
}
```

```

cout << "Suma elementów tablicy wynosi: " << suma << endl;

return 0;
}

```

W programie obliczana jest suma wartości elementów zawartych w tablicy o nazwie `tablica`. Tablica ta ma rozmiar 5 i zawiera elementy składowe należące do typu `double`, co wynika z deklaracji: `double tablica[n];`, gdzie `n` to stała nazwana równa 5.

Wartości elementów tablicy są wprowadzane z klawiatury przy wykorzystaniu pętli `for`. Przy czym podczas tej operacji odniesienie się do poszczególnych elementów tablicy `tablica` jest realizowane za pomocą wyrażenia `*(wsk + i)`, gdzie `*` to operator dereferencji, `wsk` to wskaźnik, do którego została skopiowana wartość wskaźnika na pierwszy element tablicy (`wsk = tablica;`), `i` zaś stanowi zmienną sterującą pętli `for`.

Obliczenie sumy elementów zapisanych w tablicy `tablica` jest wykonywane również za pomocą pętli `for`. W tym przypadku odniesienie do poszczególnych elementów składowych tablicy jest realizowane w inny sposób niż w trakcie wprowadzania wartości elementów tablicy z klawiatury. Tutaj wykorzystano wyrażenie `wsk[i]`, czyli wskaźnik `wsk` (w którym przechowywana jest kopia wskaźnika `tablica`) oraz operator indeksowy `[]`. Wyrażenie `wsk[i]` jest równoważne wyrażeniu `tablica[i]`, w którym używany jest operator indeksowy `[]`, oraz wyrażeniu `*(wsk + i)`, w którym stosuje się operator dereferencji `*`.

Należy zwrócić uwagę na to, że wskaźnik `wsk` podczas wykonywania programu nie zmienia swojej wartości.

Ćwiczenie 5.5

Zmodyfikuj program z przykładu 5.5 — odwołania do poszczególnych elementów tablicy `tablica` zrealizuj przy użyciu wskaźnika `wsk` oraz:

- operatora dereferencji `*` (wariant pierwszy),
- operatora indeksowego `[]` (wariant drugi).

Załącz, że wskaźnik `wsk` w trakcie wykonywania programu nie zmienia swojej wartości (czyli tak, jak w programie zawartym w przykładzie 5.5).

W przykładzie 5.6 zilustrowano drugi przypadek omówiony na początku tego podrozdziału. Tutaj wskaźnik `wsk` kieruje bezpośrednio na kolejne elementy składowe tablicy, czyli podczas wykonywania programu zmienia swoją wartość — „porusza się” pomiędzy kolejnymi elementami składowymi tablicy.

Przykład 5.6

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

int main() {
    const int n = 5; // rozmiar tablicy

    // Deklaracja zmiennej tablicowej:
    double tablica[n];
    // Deklaracja wskaźnika wsk połączona z jego inicjalizacją:
    double *wsk = tablica;

    // Wprowadzenie wartości elementów tablicy z klawiatury:
    cout << "Podaj wartości " << n << " elementów tablicy:" << endl;
    for (int i = 0; i < n; i++) {
        cout << " tablica[" << i << "] = ";
        cin >> *wsk; // równoważne wyrażeniu: cin >> tablica[i];

        // Inkrementacja wskaźnika wsk:
        wsk++;
        /* UWAGA
         * Inkrementacja wskaźnika wsk spowoduje jego przesunięcie przed następny element tablicy.
         */
    }
    // Przesunięcie wskaźnika wsk przed pierwszy element tablicy:
    wsk = tablica; // równoważne instrukcji wsk = &tablica[0];

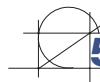
    double suma = 0;
    // Obliczenie sumy elementów zapisanych w tablicy:
    for (int i = 0; i < n; i++) {
        suma += *wsk++; // równoważne instrukcji: suma += tablica[i];
    }

    cout << "Suma elementów tablicy wynosi: " << suma << endl;
    return 0;
}
```

Analogicznie jak w przykładzie 5.5, również tutaj obliczana jest suma elementów zapisanych w tablicy o nazwie `tablica`. Odniesienie do poszczególnych elementów składowych tablicy jest realizowane przy użyciu wskaźnika `wsk` oraz operatora dereferencji `*: *wsk`. Jednakże tutaj wskaźnik `wsk` zmienia swoją wartość w trakcie wykonywania programu — w każdej iteracji w obu wykorzystanych pętlach `for` wskaźnik ten wskazuje bezpośrednio na kolejny element tablicy. „Reset” wskaźnika `wsk` przed rozpoczęciem wykonywania drugiej pętli `for` jest konieczny ze względu na to, że po ostatniej iteracji w pierwszej pętli `for` wskaźnik ten wskazuje na ostatni element tablicy.

Ćwiczenie 5.6

Zmodyfikuj program z przykładu 5.6 — oprócz obliczenia sumy elementów wyznacz wartość największą (maksimum) i najmniejszą (minimum) z elementów składowych tablicy.



5.3. Tablice dynamiczne i wektory

5.3.1. Tablice dynamiczne

W poprzednim podrozdziale omówione zostały tablice statyczne (ang. *static arrays*), inaczej tablice stałe (ang. *fixed arrays*). Cechą charakterystyczną takich tablic jest to, że pamięć operacyjna niezbędna do ich przechowania jest alokowana (przydzielana) przez kompilator na stosie w sposób statyczny w czasie komplikacji programu. Podczas wykonywania programu wielkość tej pamięci się nie zmienia — tym samym liczba elementów składowych tablicy statycznej również nie może się zmieniać w trakcie działania programu. Drugą ważną cechą tablic statycznych jest to, że ich rozmiar (liczba elementów składowych) musi być znany już na etapie komplikacji programu. Stąd ustalenie rozmiaru takiej tablicy w jej deklaracji opiera się na wykorzystaniu stałych (dokładnie: wyrażeń dających w wyniku stałe), a nie na użyciu zmiennych.

Tablice dynamiczne (ang. *dynamic arrays*) są natomiast tablicami, dla których pamięć operacyjna jest alokowana dynamicznie (ang. *dynamic memory allocation*) w czasie wykonywania programu (ang. *runtime allocation*), a nie na etapie komplikacji (ang. *compile-time allocation*). Konsekwencją tego jest istotna cecha tablic dynamicznych polegająca na tym, że ich rozmiar — i tym samym liczba elementów składowych — może być ustalany na podstawie wartości zmiennych programowych, a nie stałych.

Wadą tablic dynamicznych jest to, że nie mogą one zmieniać rozmiaru po utworzeniu. Tym samym w trakcie wykonywania programu nie jest możliwe ani dodawanie nowych elementów składowych takich tablic, ani usuwanie istniejących.

Tablice dynamiczne są przechowywane w pamięci operacyjnej na **stercie** (ang. *heap*).



UWAGA

Dynamiczna alokacja pamięci dla zmiennych typów podstawowych została omówiona szczegółowo w podrozdziale 4.3 podręcznika.

W celu dynamicznego przydzielenia pamięci dla tablicy należy użyć operatora `new`. Zwolnienie pamięci przeznaczonej dla tablicy dynamicznej jest realizowane za pomocą operatora `delete[]`. Ogólna postać instrukcji umożliwiającej utworzenie tablicy dynamicznej jest następująca:

```
wskaźnik = new nazwa_tablicy[liczba_elementów];
```

gdzie:

- **wskaźnik** stanowi zmienną wskaźnikową należącą do typu **typ_składowy***, a **typ_składowy** określa typ, do którego należą elementy składowe tablicy,
- **nazwa_tablicy** jest identyfikatorem zmiennej tablicowej,
- **liczba_elementów** to liczba elementów składowych tablicy, która w przypadku ogólnym może być wyrażeniem dającym w wyniku zmienną należącą do typu całkowitego, np. typu **int**.

Po utworzeniu tablicy i zaalokowaniu dla niej niezbędnej pamięci na stercie adres pierwszego elementu tablicy (tj. elementu o indeksie 0) zostaje skopiowany do wskaźnika **wskaźnik**.

Na przykład utworzenie tablicy dynamicznej **tablica** o rozmiarze pobranym ze zmiennej **n** i zawierającej elementy składowe typu **float** ma postać: **int n = 10; float *wsk = new float[n];**, gdzie **wsk** to wskaźnik do pierwszego elementu utworzonej tablicy.

Zwolnienie pamięci zaalokowanej dla tablicy dynamicznej na stercie ma postać ogólną:

```
delete[] wskaźnik;
```

gdzie **wskaźnik** jest zmienną wskaźnikową wskazującą na tablicę (dokładnie: pierwszy element tablicy).

Na przykład zwolnienie pamięci zaalokowanej na stercie dla tablicy **tablica** wskazywanej przez wskaźnik o nazwie **wsk** ma postać: **delete[] wsk;**.

Tablice dynamiczne nie mają identyfikatorów (nazw). Odwołanie się do ich elementów składowych jest realizowane przy wykorzystaniu operacji na wskaźnikach — z zastosowaniem reguł arytmetyki wskaźników, które zostały omówione wcześniej.

Przykład 5.7

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

int main() {
    int n; // rozmiar tablicy
    cout << "Podaj liczbę elementów tablicy: " << endl;
    cout << "n = ";
    cin >> n;

    // Deklaracja wskaźnika wsk należącego do typu double*:
    double *wsk;
    // Dynamiczny przydział pamięci na stercie dla tablicy:
    wsk = new double[n];
```

```

/* UWAGA
 * Adres pierwszego elementu tablicy dynamicznej zostanie zapisany we wskaźniku wsk.
 */

// Wprowadzenie wartości elementów składowych tablicy z klawiatury:
cout << "Podaj wartości " << n << " elementów tablicy:" << endl;
for (int i = 0; i < n; i++) {
    cout << " tablica[" << i << "] = ";
    cin >> *(wsk + i); // równoważne zapisowi: cin >> wsk[i];
}

double suma {0};
// Obliczenie sumy elementów zapisanych w tablicy:
for (int i = 0; i < n; i++) {
    suma += *(wsk + i); // równoważne zapisowi: suma += wsk[i];
}

// Zwolnienie pamięci przydzielonej dynamicznie dla tablicy wskazywanej przez wskaźnik wsk:
delete[] wsk;

cout << "Suma elementów tablicy wynosi: " << suma << endl;
return 0;
}

```

W programie obliczana jest suma elementów przechowywanych w tablicy, dla której pamięć operacyjna została zaalokowana dynamicznie — na stercie, w trakcie wykonywania programu. Tablica ta została utworzona przy wykorzystaniu operatora `new`. Liczba elementów składowych tablicy jest podawana z klawiatury jako dane wejściowe. Tym samym rozmiar tablicy nie jest ustalany na etapie komplikacji programu, lecz w czasie jego wykonywania. Tablica dynamiczna nie ma nazwy. Odwołanie się do jej elementów składowych jest realizowane z zastosowaniem arytmetyki wskaźnika `wsk`. Zwolnienie pamięci na stercie przydzielonej dla tablicy uzyskano za pomocą operatora `delete[]`.

Ćwiczenie 5.7

Zmodyfikuj program z przykładu 5.7 — operacje na tablicy dynamicznej zrealizuj przy użyciu wskaźnika `wsk`, który w kolejnych iteracjach w wykorzystanych pętlach `for` wskazuje na kolejne elementy składowe tablicy `tablica` — czyli jest „ruchomy” (zmienia swoją wartość w czasie wykonywania programu).

5.3.2. Wektory

Wektory (ang. *vectors*) to **kontenery** (ang. *containers*), w których można przechowywać listę (kolekcję) określonych danych. Dane te można przetwarzać przy wykorzystaniu funkcji predefiniowanych w zbiorze nagłówkowym o nazwie `vector`.

Analogicznie jak tablice, wektory są przechowywane w ciągłych obszarach pamięci operacyjnej. Pamięć operacyjna dla wektorów jest alokowana dynamicznie.

Wektory są tablicami dynamicznymi. Jednakże charakteryzuje je kilka ważnych właściwości (cech), które odróżniają je od zwykłych, klasycznych tablic dynamicznych (omówionych w podrozdziale 5.3.1):

1. Wektory po utworzeniu mogą zmieniać swój rozmiar. Tym samym możliwe jest dodawanie nowych lub usuwanie istniejących elementów składowych wektorów w czasie wykonywania programu. Dlatego też wektory są zaliczane do **struktur danych skalowalnych** (ang. *resizable data structures*).
2. Pamięć operacyjna przydzielona dla wektora w sposób dynamiczny jest zwalniana automatycznie. Nie ma potrzeby jawnego usuwania wektorów z pamięci przez użycie operatora `delete[]`.
3. Istnieje możliwość określenia rozmiaru (liczby elementów składowych) wektora w trakcie wykonywania programu.
4. Wektory nie mogą być kopowane i przypisywane bezpośrednio do zmiennych programowych, natomiast zwykłe tablice dynamiczne mogą.
5. Wektory mogą mieć nazwy (identyfikatory). Dzięki temu można je przetwarzać w tradycyjny sposób, przy użyciu operatora indeksowego `[]`.

UWAGA

Więcej informacji dotyczących wektorów oraz predefiniowanych operacji, które można wykonywać na elementach składowych wektorów, znajduje się w dokumentacji języka C++ na stronie cppreference.com, w artykule *Vector*.

Wykorzystanie wektorów wiąże się z koniecznością dołączenia do programu zbioru nagłówkowego o nazwie `vector` za pomocą dyrektywy preprocessora: `#include <vector>`.

UWAGA

Dyrektyny preprocessora zostały omówione w rozdziale 9. podręcznika.

Przykład 5.8

```
#include <iostream>
// Dołączenie do programu zasobów zbioru nagłówkowego vector:
#include <vector>
using namespace std;

int main() {
    // Deklaracja i inicjalizacja (w stylu C++11) wektora o nazwie wektor:
```

```

vector <int> wektor = {10, 20, 30, 40, 50};
/* UWAGA
 * Wektor należący do typu vector zawiera elementy składowe typu int.
 * Początkowa liczba elementów (rozmiar) wektora wynosi 5.
 */
// Wyświetlenie zawartości wektora:
for (int i = 0; i < wektor.size(); i++) {
    cout << "wektor[" << i << "] = " << wektor[i] << endl;
}

// Dopuszczenie nowego elementu z zadaną wartością na końcu wektora:
wektor.push_back(60);
cout << "Ostatni element (dopisany): " << wektor[wektor.size() - 1] << endl;

// Wstawienie nowego elementu o wartości 0 przed pierwszym elementem:
wektor.insert(wektor.begin(), 0);
cout << "Pierwszy element (dopisany): " << wektor[0] << endl;

// Wyświetlenie bieżącej zawartości wektora:
for (int i = 0; i < wektor.size(); i++) {
    cout << "wektor[" << i << "] = " << wektor[i] << endl;
}

return 0;
}

```

Wektor o nazwie `wektor` należący do typu `vector` reprezentuje w programie tablicę dynamiczną zawierającą elementy składowe typu `int`. Początkowy rozmiar wektora, ustalony na podstawie liczby wartości początkowych jego elementów składowych, wynosi 5.

Rozmiar (liczba elementów) wektora `wektor` zmienia się w trakcie wykonywania programu. Najpierw (za pomocą metody `push_back()`) dopisywany jest element o wartości 60 na jego końcu — jako nowy, dodatkowy element. Później dopisywany jest (przy wykorzystaniu metody `insert()`) kolejny element na jego początku — przed pierwszym istniejącym elementem. Bieżący rozmiar wektora uzyskuje się za pomocą metody `size()`.

Ćwiczenie 5.8

Zmodyfikuj program z przykładu 5.8 — zamiast inicjować elementy składowe wektora `wektor` w kodzie źródłowym programu, zrób tak, żeby wartości tych elementów były wprowadzane z klawiatury.



5.4. Pętla foreach

W standardzie C++11 wprowadzono nowy rodzaj pętli `for` — **pętlę `for` opartą na zakresie** (ang. *range-based for loop*). Pętla ta stanowi odpowiednik zwykłej pętli `for` przeznaczony do wykorzystania w przypadkach, w których iteracje obejmują wszystkie elementy kontenera zawierającego dane należące do tego samego typu. Wspomnianym kontenerem może być np. tablica czy też wektor. Tym samym „zakresem” omawianej tutaj pętli `for` jest zbiór wszystkich elementów składowych tablicy lub wektora.

Pętlę `for` opartą na zakresie nazywa się często **pętlą `foreach`**, co wynika z jej charakterystycznych właściwości, które przedstawiono powyżej. Jest to dobre podejście nazewnictwo — bardzo uniwersalne, ponieważ np. język programowania C# udostępnia pętlę o formalnej nazwie `foreach`, której właściwości i działanie są analogiczne do pętli `for` opartej na zakresie w języku C++.



UWAGA

Nie należy mylić omawianej tutaj pętli `foreach` z natywną pętlą języka C++ o formalnej nazwie `for_each`, która pozwala na wykonanie określonej funkcji na wszystkich elementach w zadanym zakresie. Informacje dotyczące pętli `for_each` można znaleźć na stronie pod adresem cppreference.com, w artykule `for_each`.

Przykład 5.9

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

int main() {
    const int n {5}; // rozmiar tablicy

    // Deklaracja zmiennej tablicowej o nazwie tablica:
    double tablica[n];

    // Wprowadzenie z klawiatury wartości wszystkich elementów tablicy:
    cout << "Podaj wartości poszczególnych elementów tablicy: " << endl;
    for (double& element : tablica) {
        cout << "element = ";
        cin >> element;
    }
    /* UWAGA
     * W pętli for powyżej zmienna element stanowi referencję do elementu składowego tablicy.
     * Tym samym wartości elementów tablicy można zmieniać (nadawać wartość, modyfikować).
     */
}
```

```

// Obliczenie sumy wszystkich elementów zapisanych w tablicy:
double suma {0};
for (double element : tablica) {
    suma += element;
}
/* UWAGA
* W pętli for powyżej zmienna element oznacza kopię elementu tablicy. Oznacza to, że operacje są wykonywane
* na kopiiach elementów, a nie na oryginałach. Tym samym zmiana wartości elementów tablicy jest niemożliwa.
*/
// Wyświetlenie wyniku — sumy elementów tablicy:
cout << "Suma elementów tablicy wynosi: " << suma << endl;

return 0;
}

```

W programie przetwarzana jest tablica statyczna o nazwie `tablica`. Wartości elementów składowych tablicy są wprowadzane z klawiatury przy wykorzystaniu pętli `foreach`. Zmienna `element` reprezentuje pojedynczy element tablicy `tablica`, a dokładniej: stanowi referencję do danego elementu tablicy. Dzięki temu możliwa jest modyfikacja (zmiana) wartości elementów składowych tablicy. Rozmiar (liczba elementów) tablicy `tablica` określa zakres zastosowania pętli — liczbę wykonywanych iteracji. Obliczenie sumy elementów składowych zapisanych w tablicy również jest realizowane przy użyciu pętli `foreach`. W tym przypadku zmienna `element` reprezentuje kopię elementu tablicy. Tym samym wartości elementów tablicy można jedynie odczytywać, natomiast ich modyfikacja nie jest możliwa.

Szczególnie istotne w działaniu przedstawionych pętli `for` jest to, że iteracje są wykonywane na wszystkich (bez wyjątku) elementach składowych tablicy.

Ćwiczenie 5.9

Zmodyfikuj program z przykładu 5.9 — zamiast tablicy zastosuj wektor należący do typu `vector`.

UWAGA

Tablice i wektory mogą odgrywać rolę parametrów/argumentów funkcji. Ta tematyka została omówiona w podrozdziale 8.3.4 podręcznika.



5.5. Pytania i zadania kontrolne

5.5.1. Pytania

1. Czy elementy składowe tablic mogą należeć do różnych typów? Dlaczego tablice zalicza się do agregacyjnych typów danych?
2. Na czym polega różnica w alokacji pamięci operacyjnej pomiędzy tablicami statycznymi a tablicami dynamicznymi?
3. Na czym polega arytmetyka wskaźnika w odniesieniu do tablic?
4. Czy w trakcie wykonywania programu można dodawać do tablic dynamicznych (niebędących wektorami typu `vector`) nowe elementy i usuwać istniejące?
5. Czy w trakcie wykonywania programu można dodawać do wektorów nowe elementy i usuwać istniejące?
6. Czym różni się zwykła pętla `for` od pętli `foreach` (tj. pętli `for` opartej na zakresie)?

5.5.2. Zadania

1. Napisz program pozwalający wyznaczyć wartość największą (maksimum) i najmniejszą (minimum) z elementów tablicy rzeczywistej. Liczba elementów tablicy wynosi 10. Wykonaj program w trzech wariantach, w zależności od rodzaju wykorzystanej tablicy/wektora. Użyj:
 - tablicy statycznej (wariant pierwszy),
 - tablicy dynamicznej (wariant drugi),
 - wektora (wariant trzeci).
 Dane wejściowe (wartości elementów składowych tablicy/wektora) mają być wprowadzane z klawiatury. Wyniki powinny być wyświetlane w konsoli na ekranie monitora.
2. Napisz program umożliwiający posortowanie w porządku rosnącym elementów tablicy jednowymiarowej o wartościach należących do typu całkowitego za pomocą **metody bąbelkowej** (ang. *bubble sort*). Wykorzystaj tablicę statyczną. Wykonaj program w dwóch wariantach — odwołanie do elementów składowych tablicy zrealizuj przy wykorzystaniu:
 - identyfikatora tablicy i operatora indeksowego (wariant pierwszy),
 - arytmetyki wskaźników i operatora dereferencji (wariant drugi).
3. Napisz program pozwalający obliczyć sumę i różnicę dwóch 1-wymiarowych 5-elementowych tablic rzeczywistych. Dane wejściowe (wartości elementów składowych tablic) mają być wprowadzane z klawiatury. Wyniki niech będą wyświetlane w konsoli na ekranie monitora.
4. Napisz program pozwalający wyznaczyć największą i najmniejszą wartość przechowywaną w tablicy 2-wymiarowej składającej się z 4 wierszy i 3 kolumn. Wykonaj program w trzech wariantach — z użyciem:

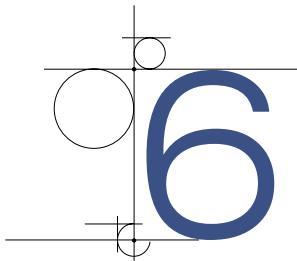
- pętli `for` (wariant pierwszy),
- pętli `while` (wariant drugi),
- pętli `do-while` (wariant trzeci).

5. Napisz program pozwalający obliczyć sumę elementów składowych wektora oraz ich średnią arytmetyczną. Dane wejściowe (wartości elementów składowych wektora) mają być wprowadzane z klawiatury. Wykorzystaj:

- pętlę `for` (wariant pierwszy),
- pętlę `foreach` (wariant drugi).

6. Napisz program umożliwiający sprawdzenie, czy tablica złożona ze znaków (`char`) zawiera zadany znak wprowadzony z klawiatury. Jeśli tak, to niech program określi liczbę wystąpień tego znaku. Wykorzystaj:

- pętlę `for` (wariant pierwszy),
- pętlę `foreach` (wariant drugi).



Łańcuchy znaków

Łańcuch znaków (ang. *string*) stanowi kolekcję (sekwencję) znaków, inaczej: ciąg znaków.

Ciąg znaków ujętych w podwójne apostrofy "" reprezentuje **literał łańcuchowy** (ang. *string literal*), np. "Język C++", "łańcuch znaków". Literał łańcuchowy może zawierać również znaki specjalne. Na przykład literał "\nWieloliniowy\nłańcuch\nznaków\n" zawiera znak specjalny \n, który oznacza wstawienie nowej linii. Była o tym mowa wcześniej — w rozdziale 3.

UWAGA

Więcej informacji na temat literałów łańcuchowych w języku C++ można znaleźć w dokumentacji języka na stronie cppreference.com, w artykule *String literal*.

Język C++ udostępnia dwa podejścia umożliwiające przetwarzanie łańcuchów znaków rozumiane jako wykonywanie operacji na łańcuchach, np. deklarowanie, inicjowanie, kopiowanie i porównywanie, mianowicie:

- wykorzystanie tzw. C-łańcuchów, które w Polsce przyjęło się nazywać raczej **C-napisami** (ang. *C-strings*),
- wykorzystanie typu danych o nazwie `string`.

Podejście związane z użyciem C-napisów C++ odziedziczył po języku C. C++ nadal zapewnia pełną obsługę C-napisów.

Natomiast drugie podejście zostało wprowadzone dopiero w C++. Polega ono na wykorzystaniu typu danych o nazwie `string`, który z formalnego punktu widzenia jest klasą zdefiniowaną w bibliotece (zbiorze nagłówkowym) o nazwie *string*. Biblioteka ta należy do biblioteki standardowej C++.

UWAGA

Klasy i obiekty zostały omówione w rozdziale 11. podręcznika.

**UWAGA**

Dla uproszczenia w dalszej części tego rozdziału w odniesieniu do identyfikatora (nazwy) `string` zamiast terminu klasa używane będzie określenie typ danych.

**6.1. C-napisy**

Łańcuch znaków w języku C (C-napis, C-łańcuch) stanowi jednowymiarową tablicę zawierającą pojedyncze znaki (należące do typu `char`), z których ostatni jest znakiem specjalnym '`\0`' (NUL) — znakiem pustym (ang. *null character*).

**UWAGA**

Nie należy mylić znaku specjalnego NUL z wartością `NULL`, która reprezentuje wskaźnik pusty.

Rozmiar tablicy, w której są przechowywane znaki C-łańcucha, jest przydzielany w sposób statyczny na etapie komplikacji programu i nie zmienia się w trakcie jego wykonywania. Tym samym podczas wykonywania programu nie może się zmienić długość C-łańcucha — tj. analogicznie jak w przypadku klasycznych tablic.

**UWAGA**

W dalszej części podręcznika łańcuchy znaków znane z języka C, a więc tablice znaków zakończone znakiem specjalnym '`\0`', będą nazywane *C-napisami* lub *zmiennymi napisowymi (C-napisowymi)*.

6.1.1. Deklarowanie C-napisów

W deklaracji C-napisu jest używana w sposób bezpośredni jego reprezentacja w postaci tablicy znaków. Można również wykorzystać szczególną właściwość tablicy w językach C i C++, mianowicie to, że nazwa zmiennej tablicowej odpowiada wskaźnikowi do jej pierwszego elementu składowego.

**UWAGA**

Tematyka przetwarzania tablic przy wykorzystaniu wskaźników (arytmetyki wskaźników) została omówiona szczegółowo w podrozdziale 5.2.

Ogólna postać deklaracji C-napisu (zmiennej napisowej) jest następująca:

```
char nazwa[długość_napisu + 1];
```

lub

```
char *nazwa;
```

gdzie:

- char to nazwa predefiniowanego typu danych znakowych,
- nazwa to identyfikator zmiennej napisowej,
- długość_napisu to maksymalna liczba znaków w C-napisie.

Na przykład `char język[15];` oznacza deklarację C-napisu (zmiennej C-napisowej) o nazwie język i maksymalnej długości 14 znaków (nie wliczając znaku '\0'), a `char *wersja` — deklarację wskaźnika wersja do C-napisu.

6.1.2. Inicjowanie C-napisów

C-napisy można inicjować w dwojakim sposobie:

- przy użyciu literałów łańcuchowych, np. `char język[15] = "C++";`,
- jako sekwencję pojedynczych znaków tablicy zakończonych znakiem specjalnym '\0' (NUL, czyli ASCII 0), np. `char język[15] = {'C', '+', '+', '\0'};;`.

Inicjalizację C-napisów za pomocą literałów łańcuchowych realizuje się w analogiczny sposób jak inicjalizację zmiennych należących do typów podstawowych (podrozdział 3.1.1).

Przykład 6.1

```
#include <iostream>
// Dołączenie do programu zasobów biblioteki (zbioru nagłówkowego) o nazwie cstring:
#include <cstring>
/* UWAGA
 * Dołączenie zasobów biblioteki cstring jest konieczne ze względu na wykorzystanie w programie funkcji strcpy()
 * zdefiniowanej w tej bibliotece. Zamiast cstring można dołączyć równoważny zbiór nagłówkowy string.h.
 */
using namespace std;

int main() {
    // Deklaracja i inicjalizacja C-napisu język1:
    char język1[15] = "C++"; // inicjalizacja kopującą
    /* UWAGA
     * C-napis język1 można również zainicjować na inne sposoby:
     * char język1[15] ("C++"); // inicjalizacja bezpośrednia
     * lub
     * char język1[15] = {"C++"}; // inicjalizacja jednolita kopującą
```

```

* lub
* char jezyk1[15] {"C++"}; // inicjalizacja jednolita bezpośrednia
*/
cout << jezyk1 << endl;

// Skopiowanie literatuła łańcuchowego "Java" do zmiennej jezyk1:
strcpy(jezyk1, "Java");
/* UWAGA
* Funkcja strcpy() pozwala na skopiowanie wartości stałej napisowej (typu const char*), jako jej drugiego
* argumentu, do zmiennej typu char* — pierwszego argumentu. Funkcja ta została zdefiniowana
* w bibliotece cstring.
*/
cout << jezyk1 << endl;

// Deklaracja i inicjalizacja C-napisu jezyk2:
char jezyk2[] = "C#";
/* UWAGA
* Rozmiar tablicy zostanie ustalony na podstawie rozmiaru wartości początkowej.
*/
cout << jezyk2 << endl;
// Skopiowanie napisu "Java" do zmiennej jezyk2:
strcpy(jezyk2, "Java");
cout << jezyk2 << endl;

// Deklaracja i inicjalizacja C-napisu jezyk3:
char *jezyk3 = "C";
cout << jezyk3 << endl;
/* UWAGA
* Modyfikacja napisu jezyk3 za pomocą instrukcji przypisania, w której po prawej stronie będzie inny literał
* łańcuchowy, jest możliwa. Wynika to z faktu, że konwersja stałej łańcuchowej do typu char* jest wykonalna,
* chociaż jest to sposób zdeprecatedowany.
*/
jezyk3 = "Java";
cout << jezyk3 << endl;

return 0;
}

```

W programie wykorzystano zmienne napisowe (C-napisy): jezyk1, jezyk2 i jezyk3. Deklarcje wszystkich trzech C-napisów są połączone z ich inicjalizacjami.

Zmienna jezyk1 jest tablicą znaków o rozmiarze 15. Tym samym można w niej przechować C-napis o maksymalnej długości 14 znaków. Tę zmienną zainicjowano za pomocą literatułu łańcuchowego "C++". Wykorzystano inicjalizację kopującą. Na przykładzie zmiennej

jezyk1 przedstawiono w komentarzach wewnątrz kodu programu inne metody inicjalizacji C-napisów.

Ze względu na to, że C-napis jezyk1 został zainicjowany literałem napisowym, nie będzie możliwa jego późniejsza modyfikacja za pomocą instrukcji przypisania, w której po prawej stronie będzie się znajdował inny literal łańcuchowy. Przykładowo instrukcja jezyk1 = "Java"; nie zostanie wykonana ze względu na niezgodność typów z lewej i prawej strony operatora przypisania. Typ z lewej strony operatora = to `char[15]`, a z prawej — `const char[5]`.

Jeśli zmienna jezyk1 zostałaby zadeklarowana, ale nie zainicjowana, tj. `char jezyk1[15];`, to instrukcja `jezyk1 = "Java";` również nie zostałaby wykonana. Powód jest taki sam jak wcześniej.

Zmienną (tablicę) jezyk2 zainicjowano wartością "C#" przy użyciu metody bezpośredniej. Rozmiar tej tablicy, 3, został ustalony na etapie kompilacji na podstawie liczby znaków literalu "C#", która wynosi 2, z uwzględnieniem znaku specjalnego '\0', automatycznie dodawanego na końcu łańcucha.

Napis jezyk2 został zainicjowany literałem łańcuchowym. Dlatego też nie będzie możliwa jego modyfikacja za pomocą instrukcji przypisania, w której po prawej stronie będzie inny literal łańcuchowy. Na przykład instrukcja `jezyk2 = "Java";` jest błędna, ponieważ próbuje przypisać wartość typu `const char[5]` do zmiennej typu `char[3]`.

Zmienna jezyk3 została z kolei zadeklarowana jako wskaźnik do znaku (`char*`). W tym przypadku także zastosowano inicjalizację kopującą.

Komentarze zawarte w programie zawierają również ważne informacje dotyczące możliwości (lub braku możliwości) modyfikowania wartości wykorzystanych zmiennych napisowych. W przypadku zmiennych jezyk1 i jezyk2 zmiana ich wartości za pomocą operatora przypisania nie jest możliwa ze względu na niezgodność typów po lewej i prawej stronie operatora. Do zmiany wartości tych zmiennych należy zastosować predefiniowaną funkcję `strcpy()` z biblioteki `cstring`. Zmiana wartości zmiennej jezyk3 przy użyciu operatora przypisania jest formalnie wykonalna, jednakże jest to metoda zdeprecatedjowanej.

Ćwiczenie 6.1

Zmodyfikuj program z przykładu 6.1 — zainicjuj wszystkie zmienne napisowe za pomocą metody jednolitej bezpośredniej.

Drugi sposób inicjalizacji C-napisów, na podstawie sekwencji pojedynczych znaków zapisanych w tablicy zakończonej znakiem specjalnym `NUL`, jest charakterystyczny dla tablic (podrozdział 5.1). Takie podejście jest możliwe z uwzględnieniem definicji C-napisu, który jest tablicą znaków zakońzoną znakiem specjalnym '\0', np. `{'J', 'a', 'v', 'a', '\0'}`.

6.1.3. Operacje wejścia/wyjścia

Operacje wejścia/wyjścia w odniesieniu do C-napisów mogą być realizowane przy użyciu standardowych strumieni wejścia i wyjścia, odpowiednio: `cin` i `cout`.

UWAGA

Standardowe wejście/wyjście zostało omówione w podrozdziale 3.1.3 podręcznika.

Wartości C-napisów można pobierać ze standardowego urządzenia wejściowego (klawiatury) przy wykorzystaniu standardowego strumienia wejściowego `cin` wraz z operatorem wyodrębniania (ang. *extraction operator*), `>>`. Jednakże w tym przypadku wprowadzenie znaku białej spacji (ang. *whitespace*) — np. znaku spacji, tabulatora, znaku wstawiania nowej linii — kończy wprowadzanie danej wejściowej. Tym samym użycie strumienia `cin` i operatora `>>` w celu wprowadzenia z klawiatury C-napisu wielozłonowego, np. "Język C++", jest nieskuteczne. Rozwiązaniem tego problemu jest użycie metody `get()` należącej do strumienia (obiektu) `cin`: `cin.get()`, która pozwala na odczytanie C-napisu wraz ze znakami białych spacji.

Wyjście, czyli wyświetlanie C-napisów w konsoli na ekranie monitora, można zaś realizować za pomocą standardowego strumienia wyjściowego `cout` z operatorem wstawiania (ang. *insertion operator*), `<<`.

Przykład 6.2

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

int main() {
    // Deklaracja zmiennej (C-napisu) wersja1 połączona z jej inicjalizacją:
    char wersja1[15] {"C++98"};
    // Prezentacja wartości zmiennej wersja1 na ekranie:
    cout << wersja1 << endl;

    cout << "Podaj inną wersję, np. C++17: ";
    // Wprowadzenie wartości zmiennej wersja1 z klawiatury:
    cin >> wersja1;
    // Prezentacja wartości zmiennej wersja1 na ekranie:
    cout << wersja1 << endl;
    cout << endl;

    // Deklaracja C-napisu wersja2:
    char wersja2[15];
    cout << "Podaj wersję C#, np. 8.0: ";
    // Wprowadzenie wartości zmiennej wersja2 z klawiatury:
```

```

    cin >> wersja2;
    // Wyświetlenie wartości zmiennej wersja2 na ekranie:
    cout << wersja2 << endl;

    // Deklaracja C-napisu wersja3:
    char wersja3[15];
    cout << "Podaj alternatywną nazwę Visual C++ 2015 (Visual C++ 14.0): ";

    // Opróżnienie strumienia wejściowego:
    cin.ignore(numeric_limits<std::streamsize>::max(), '\n');

    // Wprowadzenie wartości zmiennej wersja3 z klawiatury:
    cin.get(wersja3, 15);
    // Wyświetlenie wartości zmiennej wersja3 na ekranie:
    cout << wersja3 << endl;

    return 0;
}

```

W programie wykorzystano trzy C-napisy: `wersja1`, `wersja2` i `wersja3`. Deklarację zmiennej `wersja1` połączono z jej inicjalizacją literałem łańcuchowym "C++98". Po wyświetleniu wartości `wersja1` na standardowym urządzeniu wyjściowym (na ekranie monitora) nowa wartość tej zmiennej jest pobierana ze standardowego urządzenia wejściowego (klawiatury). Operacje wejścia zrealizowano z zastosowaniem standardowego strumienia wejściowego `cin` oraz operatora wyodrębniania `>>`, a operacje wyjścia — przy użyciu strumienia wyjściowego `cout` wraz z operatorem wstawiania `<<`.

Deklaracji C-napisu `wersja2` nie towarzyszy jej inicjalizacja przeprowadzona w sposób jawnny. Zmienna ta zostaje zainicjowana niejawnie wartością ASCII 0 (NUL). Nowa wartość tej zmiennej została pobrana z klawiatury za pomocą strumienia wejścia `cin` i operatora `>>`. Analogicznie jak w przypadku zmiennej `wersja1`, wartość zmiennej `wersja2` jest prezentowana na ekranie przy użyciu strumienia `cout`.

Zmienna `wersja3` z kolei została wprowadzona z klawiatury z wykorzystaniem strumienia `cin` za pomocą funkcji `getline()`. Jest to konieczne dlatego, że sugerowane (w programie) dane wejściowe to C-napis wieloczłonowy.

Należy zwrócić uwagę, że nie dołączono do programu zbioru nagłówkowego `cstring`. Powodem jest to, że w programie nie użyto żadnego zasobu (np. funkcji) zdefiniowanego w tej bibliotece.

Ćwiczenie 6.2

Napisz program pozwalający wprowadzić z klawiatury markę samochodu. Załóż, że łańcuch znaków odpowiadający wprowadzanej marce może być wieloczłonowy.

6.1.4. Operacje na C-napisach, funkcje napisowe

Na C-napisach (zmiennych napisowych) można wykonywać różne operacje, np. dodawanie (konkatenacja), porównywanie, kopiowanie. Można do tego celu wykorzystać funkcje zdefiniowane w bibliotece (zbiorze nagłówkowym) *cstring* języka C++. Zamiast biblioteki *cstring* można również zastosować odpowiadający jej zbiór nagłówkowy języka C o nazwie *string.h*. Uproszczony opis wybranych funkcji umożliwiających wykonywanie operacji na C-napisach zawarto w tabeli 6.1.

Tabela 6.1. Opis wybranych funkcji łańcuchowych z biblioteki *cstring*

Nazwa	Opis
strcpy(argument_1, argument_2)	Skopiowanie C-napisu źródłowego argument_2 należącego do typu <code>const char*</code> do zmiennej docelowej (tablicy znaków) argument_1 typu <code>char*</code>
strlen(argument)	Funkcja zwraca liczbę znaków w C-napisie argument typu <code>const char*</code> bez uwzględnienia znaku specjalnego '\0'
strcmp(argument_1, argument_2)	Pozwala na porównanie dwóch C-napisów typu <code>const char*</code> : argument_1 i argument_2. Funkcja zwraca: <ul style="list-style-type: none"> wartość 0, jeśli napisy są leksykograficznie identyczne, wartość ujemną, jeśli napis argument_1 jest „mniejszy” od napisu argument_2, wartość dodatnią, jeśli napis argument_1 jest „większy” od napisu argument_2
strcat(argument_1, argument_2)	Umożliwia dodanie napisu argument_2 typu <code>const char*</code> na końcu zmiennej argument_1 typu <code>char*</code>
strstr(argument_1, argument_2)	Pozwala sprawdzić, czy zadany napis argument_2 typu <code>const char*</code> jest podłańcuchem napisu argument_1 typu <code>char*</code> lub <code>const char*</code> Funkcja zwraca wskaźnik do pozycji pierwszego wystąpienia napisu argument_2 w napisie argument_1 lub wskaźnik <code>NULL</code> , jeśli argument_2 nie jest podłańcuchem napisu argument_1
strchr(argument_1, argument_2)	Pozwala sprawdzić, czy zadany znak (należący do typu <code>char</code>) jako argument_2 jest zawarty w napisie argument_1 typu <code>char*</code> lub <code>const char*</code>



UWAGA

Więcej informacji dotyczących funkcji napisowych zdefiniowanych w bibliotece *cstring* można znaleźć w dokumentacji języka C++ na stronie cppreference.com.

Przykład 6.3

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

int main() {
    // Deklaracja i inicjalizacja C-napisów napis1 i napis2:
    char napis1[20] = "Community";
    char napis2[20] = "community";

    // Określenie długości napisów:
    cout << "Długość pierwszego napisu (w znakach): " << strlen(napis1) << endl;
    cout << "Długość drugiego napisu: " << strlen(napis2) << endl;
    /* UWAGA
     * Funkcja strlen() zwraca liczbę znaków w łańcuchu, nie uwzględniając znaku specjalnego '\0' na końcu łańcucha.
     */

    // Określenie rozmiaru napisów:
    cout << "Rozmiar pierwszego napisu (w bajtach): " << sizeof(napis1) << endl;
    cout << "Rozmiar drugiego napisu (w bajtach): " << sizeof(napis2) << endl;

    // Porównanie C-napisu napis1 z C-napisem napis2:
    cout << "Czy napisy są identyczne?: " << strcmp(napis1, napis2) << endl;

    // Skopiowanie literalów łańcuchowych do zmiennych napis1 i napis2:
    strcpy(napis1, "Visual Studio");
    cout << "napis1: " << napis1 << endl;
    strcpy(napis2, "Community");
    cout << "napis2: " << napis2 << endl;

    // Dodanie literatu "", a następnie C-napisu napis2 na końcu C-napisu napis1:
    strcat(napis1, " ");
    strcat(napis1, napis2);
    cout << "napis1: " << napis1 << endl;

    // Sprawdzenie, czy literal "Studio" jest podłańcuchem napisu napis1:
    cout << "Napis \"Studio\"" << " stanowi podłańcuch łańcucha " << napis1
        << " od pozycji " << strstr(napis1, "Studio") << endl;
}
```

```
// Sprawdzenie, czy znak 'm' jest zawarty w napisie napis1:
cout << "Znak 'm' jest zawarty w łańcuchu " << napis1
    << " od pozycji " << strchr(napis1, 'm') << endl;

return 0;
}
```

W programie wykorzystano kilka predefiniowanych funkcji pozwalających na wykonywanie różnych operacji na C-napisach. W szczególności użyto wszystkich, które zostały wymienione w tabeli 6.1.

Na szczególną uwagę zasługują wyniki zastosowania funkcji `strlen()` i `sizeof()`. Mianowicie wyniki uzyskane za pomocą tych funkcji w odniesieniu do tego samego C-napisu są różne. Pierwsza z wymienionych funkcji, `strlen()`, zwraca rozmiar łańcucha reprezentowany przez liczbę zawartych w nim znaków, ale bez uwzględnienia znaku specjalnego '\0' na końcu C-napisu. Funkcja `sizeof()` zwraca zaś rozmiar napisu mierzony w bajtach, przy założeniu, że jeden bajt odpowiada pojedynczemu znakowi. Znak '\0' również jest uwzględniany.

Funkcje `strstr()` oraz `strchr()` zwracają wskaźnik do znaku należący do typu `char*` lub `const char*`. W praktyce oznacza to, że wynikiem działania obu wymienionych funkcji jest część łańcucha źródłowego (pierwszego argumentu funkcji) od zadanego napisu (funkcja `strstr()`) lub od zadanego znaku (funkcja `strchr()`) aż do końca łańcucha źródłowego.

Ćwiczenie 6.3

Na podstawie kodu zawartego w przykładzie 6.3 napisz program pozwalający na wykonanie zadanych operacji na C-łańcuchach: "Zespół Szkół" oraz "Zespół Szkół Ponadpodstawowych". W szczególności należy:

- określić długości (mierzono w znakach) i rozmiary (w bajtach),
- porównać napisy leksykograficznie,
- na końcu każdego z łańcuchów dodać napis " nr 1",
- sprawdzić, czy napis "Szkół" jest zawarty w napisach źródłowych,
- sprawdzić, czy znak 'm' jest zawarty w każdym z napisów.

Odwołanie się do określonych — zadanych — znaków w C-napisie można zrealizować z wykorzystaniem jego budowy wewnętrznej, jako tablicę złożoną ze znaków należących do typu `char`. Ogólna postać odwołania do pojedynczego znaku w C-napisie ma postać:

nazwa[indeks_znaku]

gdzie `nazwa` oznacza identyfikator C-napisu, a `indeks_znaku` to numer porządkowy znaku w napisie począwszy od liczby 0.

Przykład 6.4

```
#include <iostream>
#include <cstring>

using namespace std;
int main() {
    // Deklaracja i inicjalizacja C-napisu:
    char napis[] = "Visual Studio Community";

    // Wyświetlenie całego napisu:
    for (int i = 0; i <= strlen(napis) - 1; i++) {
        cout << napis[i];
    }
    cout << endl;

    return 0;
}
```

Ćwiczenie 6.4

Na podstawie kodu z przykładu 6.4 napisz program wypisujący na ekranie monitora w kolejnych liniach wszystkie znaki zawarte w C-napisie "Apache NetBeans". Zastosuj pętlę *foreach* (pętlę *for* opartą na zakresie) wprowadzoną w C++11 (podrozdział 5.4).

UWAGA

Tematyka związana z przekazywaniem C-napisów jako parametrów/argumentów funkcji została omówiona w podrozdziale 8.3.5 podręcznika.

6.2. Łańcuchy typu string

Łańcuch znaków należący do typu `string` stanowi ciąg (sekwencję) znaków. Z formalnego punktu widzenia łańcuch ten jest obiektem należącym do klasy `string`. Klasa ta została zdefiniowana w bibliotece (zbiorze nagłówkowym) o nazwie `string` w standardowej przestrzeni nazw `std`. Stąd wykorzystanie danych (zmiennych) typu `string` wiąże się z koniecznością dołączenia do programu zasobów tej biblioteki za pomocą dyrektywy: `#include <string>`.

Typ danych `string` nie zalicza się do wbudowanych typów danych (ang. *build-in data types*) w języku C++.

Pamięć operacyjna przeznaczona dla łańcuchów typu `string` jest alokowana dynamicznie — podobnie jak w przypadku wektorów należących do typu `vector`, które zostały przedstawione w podrozdziale 5.3.2. Tym samym rozmiar łańcucha typu `string` może się zmieniać dynamicznie w trakcie działania programu. Jest to znacząca różnica — zaleta — w porównaniu

z C-napisami, których długość (rozmiar) podczas wykonywania programu nie może się zmienić.

UWAGA

W dalszej części podręcznika łańcuchy znaków typu `string` (charakterystyczne dla języka C++) będą nazywane krótko *łańcuchami znaków* lub po prostu *łańcuchami* — w odróżnieniu od *C-napisów*, będących reprezentacją łańcuchów znaków w języku C.

6.2.1. Deklarowanie łańcuchów

Ogólna postać deklaracji zmiennej łańcuchowej jest następująca:

```
std::string nazwa;
```

gdzie:

- `std` oznacza standardową przestrzeń nazw,
- `::` to operator zakresu,
- `string` to nazwa typu danych — klasy zdefiniowanej w zbiorze nagłówkowym o nazwie `string`,
- nazwa jest identyfikatorem zmiennej.

Jeżeli w programie zadeklarowano użycie standardowej przestrzeni nazw `std` za pomocą polecenia: `using namespace std;`, operator zakresu `::` można pominąć. Na przykład deklaracja zmiennej łańcuchowej o nazwie `lancuch` ma postać: `string lancuch;`.

6.2.2. Inicjowanie łańcuchów

UWAGA

Z formalnego punktu widzenia inicjowanie łańcuchów (zmiennych łańcuchowych) realizuje się zgodnie z zasadami inicjalizacji obiektów należących do danej klasy. Zagadnienie to jest omawiane szczegółowo w rozdziale 12. podręcznika. Tutaj został przedstawiony jedynie opis uproszczony — zgodny z bazą pojęciową, jaką masz na obecnym etapie nauki programowania.

Żeby połączyć deklarację łańcuchów znaków z ich inicjalizacją, trzeba użyć inicjalizacji konstruktorowej. Przy tym możliwa jest zarówno inicjalizacja bezpośrednia (jawna), jak i pośrednia (niejawna), z uwzględnieniem inicjalizacji jednolitej, wprowadzonej w C++11.

Ogólna postać inicjalizacji łańcucha typu `string` jest następująca:

```
string nazwa = wyrażenie;
```

lub

```
string nazwa (wyrażenie);
```

lub

```
string nazwa = (wyrażenie);
```

lub

```
string nazwa {wyrażenie};
```

lub

```
string nazwa = {wyrażenie};
```

gdzie:

- `string` jest identyfikatorem typu danych (klasy),
- `nazwa` jest identyfikatorem zmiennej,
- `wyrażenie` to wyrażenie dające w wyniku literał łańcuchowy.

Przykład 6.5

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

int main() {
    // Deklaracja łańcucha (zmiennej łańcuchowej) o nazwie lancuch1:
    string lancuch1;
    cout << lancuch1 << endl;
    /* UWAGA
     * Łańcuch lancuch1 został zainicjowany wartością domyślną zerową, czyli "".
     */
    // Przypisanie zadanej wartości literała łańcuchowego do łańcucha lancuch1:
    lancuch1 = "C++";
    cout << lancuch1 << endl;

    string lancuch2 = "C++98";
    /* UWAGA
     * Łańcuch lancuch2 został zainicjowany przy wykorzystaniu inicjalizacji kopująccej.
     */
    cout << lancuch2 << endl;

    string lancuch3 ("C++03");
    /* UWAGA
     * Łańcuch lancuch3 został zainicjowany przy wykorzystaniu inicjalizacji konstruktorowej bezpośrednio.
     */
}
```

```
cout << lancuch3 << endl;

string lancuch4 = ("C++11");
/* UWAGA
 * Łańcuch został zainicjowany przy wykorzystaniu inicjalizacji konstruktorowej pośredniej.
 */
cout << lancuch4 << endl;

string lancuch5 {"C++14"};
/* UWAGA
 * Łańcuch został zainicjowany przy wykorzystaniu inicjalizacji jednolitej bezpośredniej.
 */
cout << lancuch5 << endl;

string lancuch6 = {"C++17"};
/* UWAGA
 * Łańcuch został zainicjowany przy wykorzystaniu inicjalizacji jednolitej pośredniej.
 */
cout << lancuch6 << endl;

return 0;
}
```

W programie zaprezentowano różne sposoby inicjalizacji łańcuchów znaków. Z formalnego punktu widzenia każdy z tych sposobów wymaga wywołania (ang. *call*) tzw. konstruktora (ang. *constructor*), którego zadaniem jest utworzenie i inicjalizacja obiektu należącego do danej klasy. Jest to istotne, ponieważ formalnie każdy łańcuch typu `string` jest obiektem. Tym samym każdy ze sposobów inicjalizacji wykorzystanych łańcuchów stanowi inicjalizację konstruktorową.



UWAGA

Konstruktory zostały omówione w rozdziale 12. podręcznika.

Jeśli zaś potraktuje się łańcuchy jako zmienne należące do typu `string`, widać, że przedstawione sposoby inicjalizacji łańcuchów odpowiadają sposobom stosowanym przy inicjalizacji zmiennych należących do typów podstawowych (ang. *fundamental data types*). Można zatem wyróżnić np. inicjalizację kopującą, bezpośrednią, jednolitą oraz kombinacje wymienionych sposobów, np. inicjalizację jednolitą bezpośrednią, która została wykorzystana przy inicjalizacji zmiennej `lancuch5`.



UWAGA

Sposoby inicializacji zmiennych należących do wbudowanych typów podstawowych zostały zaprezentowane w podrozdziale 3.1.1.

Ćwiczenie 6.5

Zmodyfikuj program z przykładu 6.5 — zmienne łańcuchowe: `lancuch3`, `lancuch4`, `lancuch5` i `lancuch6` zainicjuj przy użyciu wartości zmiennych `lancuch1` i `lancuch2` oraz wyrażeń zawierających te zmienne zamiast za pomocą literałów łańcuchowych.

6.2.3. Standardowe wejście/wyjście

Operacje wejścia/wyjścia mogą być realizowane za pomocą standardowych strumieni wejścia (`cin`) oraz wyjścia (`cout`) przy wykorzystaniu operatorów: wyodrębniania strumienia `>>` i wstawiania strumienia `<<`.

Jednakże należy zwrócić uwagę na to, że wprowadzenie znaku białej spacji (ang. *white-space*) — np. spacji, tabulatora — kończy wprowadzanie danej wejściowej. Tym samym użycie strumienia wejściowego `cin` wraz z operatorem wyodrębniania `>>` w celu wprowadzenia z klawiatury wartości danych wielozłonowych, np. "Visual Studio Community", zakończy się niepowodzeniem. W takich przypadkach można się posłużyć funkcją o nazwie `getline`, która pozwala wprowadzać z klawiatury całą linię tekstu z zastosowaniem strumienia `cin` do napotkania znaku nowej linii (`\r\n` lub `\n`). Funkcja `getline()`, należąca do biblioteki standardowej C++, została zdefiniowana w bibliotece `string`.

Przykład 6.6

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

int main() {
    // Deklaracja zmiennej łańcuchowej lancuch:
    string lancuch;
    cout << "Podaj nazwę jednoczlonową języka programowania: ";
    // Wprowadzenie wartości zmiennej lancuch z klawiatury:
    cin >> lancuch;
    // Wyświetlenie na ekranie bieżącej wartości zmiennej lancuch:
    cout << lancuch << endl;

    cout << "Podaj inną, dwuczłonową nazwę języka progr. (np. Visual C++): ";
    cin >> lancuch;
    cout << lancuch << endl;
```

```

/* UWAGA
 * Wprowadzanie danych wejściowych z klawiatury przy użyciu strumienia cin wraz z operatorem
 * wyodrębniania >> wiąże się z tym, że biała spacja (np. spacja, tabulator) kończy wprowadzanie danych.
 */

// Opróżnienie strumienia wejściowego:
cin.ignore(numeric_limits<std::streamsiz>::max(), '\n');

/* UWAGA
 * Działanie metody ignore() polega na odrzuceniu (ang. discard) wszystkich znaków w strumieniu wejściowym
 * aż do napotkania znaku odgrywającego rolę ogranicznika, a następnie wyodrębnieniu pozostały części łańcucha.
 * Pierwszy argument metody ignore() stanowi liczba znaków do wyodrębnienia, a drugi — znak ograniczający.
 * Tutaj pierwszy argument oznacza maksymalną (teoretyczną) liczbę znaków, którą może obsługiwać
 * bufor strumienia. Praktycznie reprezentuje on nieograniczoną liczbę znaków. Drugi argument odpowiada
 * naciśnięciu klawisza Enter.
 */

cout << "Podaj ponownie tę samą dwuczłonową nazwę języka programowania: ";
// Odczyt łańcucha znaków ze strumienia wejściowego:
getline(cin, lancuch);
/* UWAGA
 * Wywołanie funkcji getline() powoduje wczytanie linii tekstu ze strumienia wejściowego (tutaj: cin) i dołączenie go
 * * do zmiennej łańcuchowej typu string (tutaj: zmiennej lancuch).
 * Funkcja getline() czyta cały wiersz, w tym wszystkie początkowe i końcowe białe znaki, aż do naciśnięcia
 * klawisza Enter (Return).
 */
cout << lancuch << endl;

return 0;
}

```

W celu realizacji operacji wejścia/wyjścia w programie wykorzystano strumienie: wejścia `cin` oraz wyjścia `cout`. Zademonstrowano wprowadzanie z klawiatury wartości łańcucha `lancuch` o budowie jednoczłonowej oraz dwuczłonowej. W pierwszym przypadku wykorzystano strumień `cin` wraz z operatorem wyodrębniania `>>`, w drugim zaś — strumień `cin` i funkcję `getline()`.

Ćwiczenie 6.6

Zmodyfikuj program z przykładu 6.6 — wprowadź z klawiatury jednoczłonową nazwę miasta w Polsce, a następnie dwuczłonową nazwę innego miasta.

6.2.4. Funkcje łańcuchowe

Biblioteka o nazwie `string`, dołączana do programu za pomocą dyrektywy preprocesora `#include <string>`, dostarcza wielu użytecznych funkcji, które operują na łańcuchach znaków. Wykaz wybranych funkcji łańcuchowych wraz z krótkim, uproszczonym opisem zawarto w tabeli 6.2.

Tabela 6.2. Uproszczony opis wybranych funkcji biblioteki *string*

Nazwa	Opis
<code>argument_1.append(argument_2)</code>	Dodanie podłańcucha <code>argument_2</code> na końcu łańcucha <code>argument_1</code> Wynik zostanie zapamiętany w łańcuchu <code>argument_1</code>
<code>argument_1.insert(poczatek, argument_2)</code>	Wstawienie podłańcucha <code>argument_2</code> do łańcucha <code>argument_1</code> począwszy od pozycji <code>poczatek</code>
<code>argument_1.compare(argument_2)</code>	Porównanie łańcucha <code>argument_2</code> z łańcuchem <code>argument_1</code> Jeśli łańcuchy są równe, funkcja zwraca wartość całkowitą 0, w przeciwnym razie zwraca wartość różną od zera
<code>argument_1.find(argument_2)</code>	Wyszukanie podłańcucha <code>argument_2</code> w łańcuchu źródłowym <code>argument_1</code> Funkcja zwraca pozycję szukanego podłańcucha w łańcuchu źródłowym
<code>argument.substr(poczatek, dlugosc)</code>	„Wyciągnięcie” (wyodrębnienie) podłańcucha o określonej długości <code>dlugosc</code> z łańcucha źródłowego <code>argument</code> począwszy od pozycji początkowej <code>poczatek</code>
<code>argument_1.replace(poczatek, dlugosc, argument_2)</code>	Zastąpienie podłańcucha o długości <code>dlugosc</code> w łańcuchu źródłowym <code>argument_1</code> podłańcuchem <code>argument_2</code> począwszy od zadanej pozycji początkowej <code>poczatek</code>
<code>argument_1.assign(argument_2)</code>	Nadanie nowej wartości określonej za pomocą łańcucha <code>argument_2</code> łańcuchowi źródłowemu <code>argument_1</code>
<code>argument.length()</code>	Zwraca długość (liczbę znaków) łańcucha <code>argument</code>

**UWAGA**

Więcej informacji dotyczących funkcji łańcuchowych zawartych w bibliotece *string* można znaleźć w dokumentacji języka C++ na stronie cppreference.com.

Jedną z najczęściej wykonywanych operacji na łańcuchach jest ich dodawanie. Ta operacja jest nazywana **konkatenacją** (ang. *concatenation*). Konkatenację łańcuchów można realizować przy wykorzystaniu operatorów dodawania + i +=, jak również za pomocą funkcji zdefiniowanych w bibliotece *string*, np. `append()` czy też `insert()`. Zilustrowano to w przykładzie 6.7.

Przykład 6.7

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

int main() {
    // Deklaracje zmiennych łańcuchowych: s1, s2 i s3, połączone z ich inicjalizacjami:
    string s1 ("Język programowania ");
    string s2 ("C++");
    string s3 ("11");

    // Konkatenacja (dodanie) łańcuchów s1 i s2:
    string s4 = s1 + s2;
    /* UWAGA
     * Wynik dodawania łańcuchów s1 i s2 zostaje przypisany do zmiennej łańcuchowej s4.
     */
    cout << s4 << endl;

    // Skopiowanie zawartości zmiennej (łańcucha) s1 do zmiennej s5;
    string s5 = s1;
    // Konkatenacja łańcuchów — dodanie łańcucha s2 na końcu łańcucha s5:
    s5 += s2;
    cout << s5 << endl;
    /* UWAGA
     * Wynik dodawania łańcuchów zostaje przypisany do zmiennej s5.
     */

    // Dodanie łańcucha s3 na końcu łańcucha s5:
    s5.append(s3);
    /* UWAGA
     * Wynik dodawania łańcuchów (łańcuch wynikowy) zostaje zapamiętany w zmiennej s5.
     */
    cout << s5 << endl;

    // Skopiowanie zmiennej (łańcucha) s2 do zmiennej s6;
    string s6 = s2;
    // Wstawienie podłańcucha s1 na początku łańcucha s6:
    s6.insert(0, s1);
```

```

/* UWAGA
 * Łańcuch wynikowy zostaje zapisany w zmiennej s6. Liczba 0 oznacza pozycję początkową
 * podłańcucha s1 w łańcuchu s6.
 */
cout << s6 << endl;

return 0;
}

```

W programie zaprezentowano następujące operacje na łańcuchach:

- kopiowanie zawartości jednej zmiennej łańcuchowej do innej za pomocą operatora przypisania =,
- konkatenację łańcuchów przy użyciu operatora +, której wynik (łańcuch wynikowy) zostaje zapamiętany w innej zmiennej,
- dodawanie podłańcucha na końcu łańcucha jako rezultat wykorzystania operatora += oraz funkcji append(),
- wstawianie podłańcucha na początku łańcucha — zastosowanie funkcji insert().

Ćwiczenie 6.7

Zmodyfikuj program z przykładu 6.7 — z łańcuchów źródłowych "Język programowania ", "C# " i "8.0" zbuduj łańcuchy wynikowe:

- "Język programowania C# ",
- "C# 8.0",
- "Język programowania C# 8.0".

Wykorzystaj operatory +, += oraz funkcje append() i insert().

W kolejnym przykładzie (przykład 6.8) zaprezentowano wykonywanie innych, bardzo przydatnych w praktyce operacji na łańcuchach, przeprowadzanych przy wykorzystaniu funkcji, które wchodzą w skład biblioteki *string*. W szczególności dotyczy to operacji porównania dwóch łańcuchów — funkcja compare(), „wyciągania” i zastępowania zadanego podłańcucha w łańcuchu źródłowym — funkcje substr() i replace(), oraz określania długości i rozmiaru łańcucha — funkcje length() i size().

Przykład 6.8

```

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

int main() {
    // Deklaracje łańcuchów połączone z ich inicjalizacjami:
    string s1 = "C++11";
    string s2 = "11";
    string s3 = "14";
}

```

```

// Porównanie dwóch łańcuchów:
cout << s3.compare(s2) << endl; // 1
/* UWAGA
 * Funkcja compare() pozwala na porównanie dwóch łańcuchów — tutaj: łańcucha s2 z łańcuchem s3.
 * Jeżeli funkcja compare() zwraca wartość 0, to oznacza, że łańcuchy są równe. W przeciwnym razie łańcuchy
 * są różne.
 *
 * Jeżeli zwrocona wartość jest dodatnia, to oznacza, że:
 *   a) albo łańcuch s3 ma tą samą długość co s2, ale jest od niego leksykograficznie „większy”,
 *   b) albo s3 jest dłuższy od s2.
 * Jeżeli zwrocona wartość jest ujemna, to oznacza, że:
 *   a) albo łańcuch s3 ma tą samą długość co s2, ale jest od niego „mniejszy”,
 *   b) albo s3 jest krótszy od s2.
*/

```

// Wyszukanie pozycji zadanego podłańcucha w łańcuchu źródłowym:

```

cout << s1.find(s2) << endl; // 3
/* UWAGA
 * łańcuchem źródłowym jest tutaj łańcuch s1. Zadanym podłańcuchem, który jest wyszukiwany w łańcuchu
 * źródłowym, jest s2.
 * Funkcja find() zwraca pozycję pierwszego znaku w pierwszym „dopasowaniu”, czyli wystąpieniu podłańcucha s2
 * w łańcuchu s1 — począwszy od pierwszego znaku w s1 (czyli od pozycji 0).
 * Jeżeli dopasowania nie znaleziono, funkcja zwraca maksymalną możliwą wartość typu całkowitego bez znaku
 * reprezentującego rozmiar łańcucha — npos, która oznacza „aż do końca łańcucha”.
*/

```

```

int poczatek = 0; // pozycja początkowa w łańcuchu źródłowym
int dlugosc = 3; // długość podłańcucha
// „Wyciągnięcie” podłańcucha z zadanego łańcucha źródłowego:
string s4 = s1.substr(poczatek, dlugosc);
/* UWAGA
 * Funkcja substr() zwraca podłańcuch wynikowy (tutaj: s4) o określonej długości (tutaj: dlugosc) „wyciągnięty”
 * — wyodrębniony z zadanego łańcucha źródłowego (tutaj: s1) począwszy od pozycji początkowej
 * (tutaj: poczatek).
*/
cout << s4 << endl; // C++

```

// Określenie długości łańcucha:

```

cout << s4.length() << endl; // 3
/* UWAGA
 * Długość łańcucha znaków określona za pomocą funkcji length() zwraca liczbę znaków w łańcuchu — tutaj:
 * w łańcuchu s4.
*/

```

```

// Określenie rozmiaru łańcucha:
cout << s4.size() << endl; //3
/* UWAGA
 * Rozmiar łańcucha znaków (tutaj: łańcucha s4) określony za pomocą funkcji size() jest mierzony w bajtach.
 */

cout << s3 << endl; //14
poczatek = 1; // od drugiego znaku
dlugosc = 1; // jeden znak
// Zastąpienie zadanego podłańcucha w łańcuchu innym podłańcuchem:
cout << s3.replace(poczatek,dlugosc,"7") << endl; //17
/* UWAGA
 * Funkcja replace() pozwala zastąpić podłańcuch o określonej długości (tutaj: dlugosc) w łańcuchu źródłowym
 * (tutaj: s3) począwszy od zadanej pozycji początkowej (tutaj: poczatek).
 */

cout << s3 << endl; //17
// Nadanie wartości łańcuchowi s3:
cout << s3.assign("14") << endl; //14
/* UWAGA
 * Funkcja assign() pozwala nadać nową wartość (tutaj: wartość literalu "14") łańcuchowi źródłowemu (tutaj: s3).
 * Funkcja zwraca nową wartość łańcucha źródłowego.
 */

return 0;
}

```

Opis składni funkcji wykorzystywanych w programie jest zawarty w jego kodzie źródłowym. Działanie tych funkcji zostało zilustrowane przez ich użycie w odniesieniu do konkretnych danych — łańcuchów typu `string`: `s1`, `s2` oraz `s3`.

Ćwiczenie 6.8

Wykonaj na danych łańcuchowych: "c# 8.0", "5.0" i "6.0" analogiczne operacje jak w programie z przykładu 6.8. Wykorzystaj funkcje: `compare()`, `substr()`, `replace()` oraz `length()` i `size()`.

UWAGA

Zagadnienie przekazywania łańcuchów typu `string` jako parametrów/argumentów funkcji zostało omówione w podrozdziale 8.3.5 podręcznika.



6.3. Pytania i zadania kontrolne

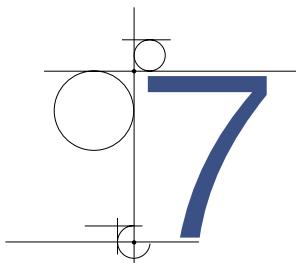
6.3.1. Pytania

1. Podaj definicję C-napisu.
2. Wymień co najmniej dwa sposoby deklarowania C-napisów.
3. W jaki sposób inicjuje się C-napisy?
4. Czy długość C-napisu może się zmienić w trakcie wykonywania programu? Uzasadnij odpowiedź.
5. Czy długość łańcucha znaków typu `string` może się zmienić w trakcie wykonywania programu? Uzasadnij odpowiedź.
6. Opisz składnię i działanie funkcji `strcpy()` zdefiniowanej w bibliotece `cstring`.
7. Opisz składnię i działanie funkcji `compare()` zdefiniowanej w bibliotece `string`.
8. W jaki sposób odczytać z klawiatury wielozłonowy łańcuch znaków (np. "Język C++")?

6.3.2. Zadania

1. Napisz program pozwalający na zapamiętanie w zmiennych łańcuchowych następujących danych samochodu wprowadzonych z klawiatury: marka, model, rok produkcji. Wprowadzone dane mają być wyświetlane kontrolnie na ekranie monitora. Wykorzystaj:
 - C-napisy (wariant pierwszy),
 - łańcuchy typu `string` (wariant drugi).
2. Napisz program pozwalający dodać do siebie trzy łańcuchy znaków zadane w postaci literalów łańcuchowych, w kolejności: "Programowanie ", "zorientowane ", "obiektywowe". Wykorzystaj:
 - C-napisy (wariant pierwszy),
 - łańcuchy typu `string` (wariant drugi).
3. Napisz program pozwalający porównać dwa łańcuchy: "Java" i "JavaScript". Wykorzystaj:
 - C-napisy (wariant pierwszy),
 - łańcuchy typu `string` (wariant drugi).
4. Napisz program umożliwiający sprawdzenie, czy łańcuch znaków "Runtime" jest podłańcuchem łańcucha "Java Runtime Environment". Wykorzystaj:
 - C-napisy (wariant pierwszy),
 - łańcuchy typu `string` (wariant drugi).
5. Napisz program pozwalający wprowadzać z klawiatury dwuzłonowe nazwy miejscowości, np. Sucha Beskidzka. Wykorzystaj:
 - C-napisy (wariant pierwszy),
 - łańcuchy typu `string` (wariant drugi).

- 6.** Napisz program umożliwiający zastąpienie podłańcucha "JavaScript" łańcuchem "Java" w łańcuchu znaków "C++, JavaScript, C#". Wykorzystaj:
- C-napisy (wariant pierwszy),
 - łańcuchy typu `string` (wariant drugi).



C-struktury i C-unie

7.1. C-struktury

Struktura (ang. *struct*) znana z języka C to zbiór danych należących do dowolnych typów. Przy tym dowolne są zarówno typy elementów składowych struktury, jak i ich liczba. Dlatego struktura stanowi złożony typ danych — podobnie jak tablica czy C-napis. Ponadto, patrząc przez pryzmat różnych kryteriów przedstawionych w podrozdziale 3.1.1, struktury należą do typów danych zdefiniowanych przez użytkownika (ang. *user-defined data types*), a także do typów pochodnych (ang. *derived data types*).

Na przykład poprawną strukturę danych stanowią wybrane dane personalne ucznia, takie jak: imię, nazwisko, rok urodzenia, klasa, grupa. Innym przykładem mogą być określone dane samochodu: marka, model, rok produkcji, cena, numer rejestracyjny.

W kodzie programu struktury określa się za pomocą słowa kluczowego `struct`. Elementy składowe struktury są nazywane **elementami członkowskimi** (ang. *members*) albo **polami** (ang. *fields*), albo **zmiennymi członkowskimi** (ang. *member variables*).

W języku C znaczenie pojęcia „struktura” jest odmienne niż w języku C++. Ujmując to najprościej, w języku C struktura oznacza — jak wspomniano wcześniej — „zbiór danych należących do różnych typów”, a w języku C++ struktura to „zbiór danych różnych typów wraz z operacjami, które na tych danych można zrealizować”.

UWAGA

W tym rozdziale omawiane są wyłącznie struktury charakterystyczne dla języka C — a nie C++. W dalszej części podręcznika są one nazywane **C-strukturami** (ang. *C-structures*). Z kolei struktury (i klasy) jako elementy języka C++ zostały zaprezentowane w rozdziale 11. podręcznika.

7.1.1. Deklarowanie i definiowanie struktur

C-struktura to złożony typ danych definiowany przez użytkownika (programistę) — o czym była mowa wcześniej. Dlatego żeby w programie można było korzystać z C-struktury, należy ją wcześniej zadeklarować i zdefiniować. Deklaracja struktury może być połączona z jej definicją lub nie.

Deklaracja struktury

Deklaracja **typu struktury** (ang. *structure type*) zawiera słowo kluczowe `struct` oraz nazwę (identyfikator) typu bez opisu (deklaracji) elementów członkowskich struktury:

`struct nazwa_struktury;`

Deklaracja C-struktury oznacza jedynie zapowiedź (anons) jej późniejszego użycia. Z kolei wykorzystanie struktury w programie w celu przetwarzania konkretnych danych wiąże się z koniecznością jej zdefiniowania, a następnie zadeklarowania zmiennej/zmiennych należących do typu struktury, w których wymagane dane są zapamiętywane.

Definicja struktury

Definicja C-struktury (typu struktury) powinna zawierać deklaracje jej elementów składowych, czyli elementów członkowskich (pół). Pola struktury deklaruje się w analogiczny sposób jak zwykle zmienne należące do typów podstawowych.

Ogólna postać definicji typu struktury (C-struktury) jest następująca:

```
struct nazwa_struktury {
    typ_pola_1 nazwa_pola_1;
    typ_pola_2 nazwa_pola_2;
    ...
};
```

gdzie:

- `struct` to słowo kluczowe oznaczające definicję struktury,
- `nazwa_struktury` jest identyfikatorem typu struktury,
- `typ_pola_1, typ_pola_2` oznaczają typy pól (zmiennych członkowskich) wchodzących w skład struktury,
- `nazwa_pola_1, nazwa_pola_2` to identyfikatory pól struktury.

Liczba pól C-struktury jest dowolna — w zależności od potrzeb. Typy pól wchodzących w skład struktury również są dowolne. Tym samym w skład struktur mogą wchodzić zarówno dane typów podstawowych, np. `int`, `float`, jak i typów złożonych, np. tablice oraz inne C-struktury.

C-struktury — podobnie jak tablice — są zaliczane do agregacyjnych typów danych (ang. *aggregative data types*), ponieważ pozwalają na zgrupowanie wielu indywidualnych danych w jednym kontenerze. Wspomniane dane są reprezentowane w strukturze przez jej pola (zmienne członkowskie).

Przykład 7.1

```
// Definicja typu struktury o nazwie Pracownik:
struct Pracownik {
    // Deklaracje pól (zmiennych członkowskich) struktury:
    char imie[20];
    char nazwisko[20];
    char stanowisko[30];
};
```

W przedstawionym fragmencie kodu zdefiniowano typ struktury (C-strukturę) o nazwie `Pracownik`. Struktura ta zawiera trzy pól (zmienne członkowskie): `imie`, `nazwisko` i `stanowisko`. Każde z tych pól stanowi C-napis, czyli tablicę znaków zakończoną znakiem specjalnym `NUL`.

Ćwiczenie 7.1

Zmodyfikuj kod źródłowy zawarty w przykładzie 7.1 — zdefiniuj C-strukturę pozwalającą na zapamiętanie w jej elementach członkowskich następujących danych ucznia: imienia, nazwiska, klasy, grupy i numeru w dzienniku.

C-struktury można zagnieźdzać. Oznacza to, że w skład struktury mogą wchodzić inne struktury. Zilustrowano to we fragmencie kodu pokazanym w przykładzie 7.2.

Przykład 7.2

```
// Definicja typu struktury Data:
struct Data {
    // Deklaracje pól:
    unsigned short int dd, mm, rr; // dzień, miesiąc, rok
};

// Definicja typu struktury Pracownik:
struct Pracownik {
    // Deklaracje pól stanowiących C-napisy:
    char imie[20];
    char nazwisko[20];
    char stanowisko[30];
    // Deklaracja pola należącego do typu struktury Data:
    Data data_zatrudnienia;
};
```

W kodzie zdefiniowano dwie struktury: Data oraz Pracownik. W polach struktury Data można zapamiętać określoną datę: dzień (pole dd), miesiąc (pole mm) i rok (rr). Struktura Pracownik zawiera deklaracje czterech pól, z których jedno, o nazwie data_zatrudnienia, należy do typu strukturowego Data. Tym samym mamy tutaj do czynienia z zagnieżdżaniem C-struktur: struktura Data jest elementem członkowskim (polem) struktury Pracownik.

Ćwiczenie 7.2

Zmodyfikuj kod źródłowy zawarty w przykładzie 7.2 — zdefiniuj C-strukturę pozwalającą na zapamiętanie w jej polach następujących danych ucznia: imienia, nazwiska, klasy, grupy oraz daty i miejsca urodzenia.

7.1.2. Zmienne strukturowe

Deklarowanie zmiennych strukturowych

Zmienne należące do typów strukturowych można deklarować na dwa sposoby. Pierwszy z nich polega na tym, że listę wymaganych zmiennych umieszcza się bezpośrednio po definicji struktury — w taki sam sposób jak w przypadku zmiennych należących do typów wyliczeniowych (podrozdział 3.4):

```
struct nazwa_struktury {
    typ_pola_1 nazwa_pola_1;
    typ_pola_2 nazwa_pola_2;
    ...
} zmienna_1, zmienna_2, ...;
```

gdzie zmienna_1 i zmienna_2 oznaczają identyfikatory zmiennych strukturowych, a pozostałe oznaczenia są takie same jak w definicji typu strukturowego, którą omówiono wcześniej w tym rozdziale.

Drugi sposób również przypomina deklarowanie zmiennych wyliczeniowych. Ponadto jest on identyczny ze sposobem deklarowania zmiennych należących do typów podstawowych:

```
nazwa_struktury zmienna_1, zmienna_2, ...;
```

Odwołania do pól struktury

Odwołania do pól (elementów członkowskich) struktury można realizować przy użyciu **operatora wyboru elementu członkowskiego** (ang. *member selection operator*) struktury: . (ang. *dot*). Postać ogólna takiego odwołania jest następująca:

```
nazwa_zmiennej_strukturowej.nazwa_pola
```

gdzie nazwa_zmiennej_strukturowej stanowi identyfikator zmiennej strukturowej, a nazwa_pola — identyfikator określonego pola (zmiennej członkowskiej) C-struktury.



UWAGA

Operator wyboru elementu członkowskiego . często jest nazywany także **operatorem dostępu do elementu członkowskiego** (ang. *member access operator*).

Przykład 7.3

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

// DEKLARACJA typu struktury o nazwie Pracownik:
struct Pracownik;
// Pracownik p; // BŁĘDNA DEKLARACJA!!!
/* UWAGA
 * Deklaracja zmiennej typu Pracownik jest błędna, ponieważ ten typ nie został zdefiniowany.
 */

int main() {
    // DEFINICJA typu struktury Pracownik:
    struct Pracownik {
        // Deklaracje pól (zmiennych członkowskich):
        char imie[20];
        char nazwisko[20];
        char stanowisko[30];
    };

    // Deklaracja zmiennej (struktury) pracownik należącej do typu struktury Pracownik:
    Pracownik pracownik;

    // Skopiowanie wartości literalów lańcuchowych do poszczególnych pól struktury pracownik:
    strcpy(pracownik.imie, "Adam");
    strcpy(pracownik.nazwisko, "Kowalski");
    strcpy(pracownik.stanowisko, "dyrektor");

    // Prezentacja wartości przechowywanych w polach struktury pracownik:
    cout << pracownik.imie << endl;
    cout << pracownik.nazwisko << endl;
    cout << pracownik.stanowisko << endl;

    return 0;
}
```

Typ struktury o nazwie `Pracownik` został zadeklarowany na zewnątrz programu głównego — tj. funkcji `main()`. Zatem deklaracja ta ma zasięg (zakres) globalny. Definicja tej C-struktury jest zawarta wewnątrz funkcji `main()`. To oznacza, że nie można definiować zmiennych struktury typu `Pracownik` o zasięgu globalnym. Jednakże jest możliwe deklarowanie zmiennych tego typu, ale o zakresie lokalnym — ograniczonym do wnętrza funkcji `main()`, np. zmiennej `pracownik`.

Odwołania do poszczególnych pól struktury `pracownik` są realizowane przy użyciu operatora wyboru `.:pracownik.imie, pracownik.nazwisko i pracownik.stanowisko`.

Ćwiczenie 7.3

Zmodyfikuj program z przykładu 7.3 — zdefiniuj typ struktury `Uczeń` pozwalający na zapamiętanie następujących danych ucznia: imienia, nazwiska, klasy i grupy. Zadeklaruj w programie zmienną struktury `Uczeń` typu `Uczeń`, a następnie przypisz jej polom (zmiennym członkowskim) konkretne, wybrane przez siebie wartości. Na końcu wyświetl kontrolnie na ekranie wartości pól struktury `Uczeń`.

7.1.3. Inicjowanie struktur

Zmienne należące do typów struktury typu struktury można inicjować w czasie deklaracji — analogicznie jak inne zmienne należące do typów podstawowych i złożonych. W przypadku C-struktury inicjalizacja polega na nadaniu wartości początkowych jej elementom członkowskim (polom).

Można zainicjować albo wszystkie pola struktury, albo tylko wybrane. W tym drugim przypadku polom, dla których nie zostały określone wartości początkowe, zostaną nadane wartości domyślne zerowe (np. polom należącym do typów liczbowych — wartość 0).

Ogólna postać deklaracji zmiennej struktury połączona z jej inicjalizacją jest następująca:

`nazwa_typu_struktury nazwa_zmiennej = {lista_wyrażeń};`

lub

`nazwa_typu_struktury nazwa_zmiennej {lista_wyrażeń};`

gdzie:

- `nazwa_typu_struktury` oznacza identyfikator zdefiniowanego wcześniej typu struktury,
- `nazwa_zmiennej` jest identyfikatorem deklarowanej i inicjowanej zmiennej struktury,
- `lista_wyrażeń` to lista wyrażeń oddzielonych od siebie przecinkami. Wyrażenia te dają w wyniku wartości (stałe) zgodne z określonymi typami pól struktury. Lista wyrażeń może obejmować wszystkie pola albo tylko wybrane.

Druga z przedstawionych powyżej postaci inicjalizacji struktury została wprowadzona w standardzie C++11.

Inicjalizację struktur zilustrowano za pomocą dwóch przykładów. W przykładzie 7.4 zaprezentowano inicjalizację struktury zawierającej pola należące do typu `string` oraz do typu `unsigned int`. Z kolei w przykładzie 7.5 pokazano inicjalizację C-struktury, w której jedno z pól stanowi inną C-strukturę, a inne — tablicę liczbową.

Przykład 7.4

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

// Definicja typu struktury Pracownik:
struct Pracownik {
    // Deklaracje pól (zmiennych członkowskich):
    string imie;
    string nazwisko;
    unsigned int nr_ewid;
};

int main() {
    // Deklaracja i inicjalizacja struktury (zmiennej struktury) pracownik1:
    Pracownik pracownik1 = {"Jan", "Kowalski", 1234};
    // Prezentacja danych zapisanych w strukturze pracownik1 na ekranie monitora:
    cout << pracownik1.imie << endl;
    cout << pracownik1.nazwisko << endl;
    cout << pracownik1.nr_ewid << endl;

    // Deklaracja i inicjalizacja zerowa struktury pracownik2:
    Pracownik pracownik2 {};
    cout << pracownik2.imie << endl;
    cout << pracownik2.nazwisko << endl;
    cout << pracownik2.nr_ewid << endl;

    // Deklaracja i inicjalizacja struktury pracownik3:
    Pracownik pracownik3 {"Jan", "Nowak", 2345}; // C++11
    cout << pracownik3.imie << endl;
    cout << pracownik3.nazwisko << endl;
    cout << pracownik3.nr_ewid << endl;

    // Deklaracja i inicjalizacja zerowa struktury pracownik4:
    Pracownik pracownik4 {}; // C++11
    cout << pracownik4.imie << endl;
    cout << pracownik4.nazwisko << endl;
    cout << pracownik4.nr_ewid << endl;

    return 0;
}
```

Typ struktury Pracownik jest typem globalnym, ponieważ został zdefiniowany na zewnątrz funkcji main(). W programie głównym (wewnętrz funkcji main()) zadeklarowano cztery zmienne struktury należące do typu Pracownik. Każda z nich została zainicjowana podczas deklaracji, przy czym zmienne pracownik3 i pracownik4 zainicjowano wartościami domyślnymi (inicjalizacja zerowa). W przypadku pól (zmennych członkowskich) imię i nazwisko należących do typu string jest to wartość łańcucha pustego "", a w przypadku pola nr_ewid — całkowite 0.

Ćwiczenie 7.4

Zmodyfikuj program z przykładu 7.4 — zamiast danych pracownika zapisanych w strukturze pracownik typu Pracownik zainicjuj w C-strukturze uczen typu struktury Uczen dane ucznia: imię, nazwisko, numer w dzienniku.

Przykład 7.5

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

// Definicja typu struktury (C-struktury) Data:
struct Data {
    // Deklaracje pól struktury Data:
    unsigned int dd, mm, rr;
};

// Definicja typu struktury (C-struktury) Uczen:
struct Uczen {
    // Deklaracje pól struktury Uczen:
    string imie;
    string nazwisko;
    unsigned int nr_ewid;
    Data data_urodzenia;
    unsigned short oceny[3];
};

int main() {
    // Deklaracja i inicjalizacja struktury uczen należącej do typu struktury Uczen:
    Uczen uczen = {"Jan", "Kowalski", 1234, {30, 10, 2000}, {3, 4, 5}};

    // Prezentacja danych zapisanych w strukturze uczen na ekranie monitora:
    cout << uczen.imie << endl;
    cout << uczen.nazwisko << endl;
    cout << uczen.nr_ewid << endl;
}
```

```

// Wyświetlenie danych zapisanych w strukturze wewnętrznej data_urodzenia należącej do typu Data:
cout << uczen.data_urodzenia.dd << endl;
cout << uczen.data_urodzenia.mm << endl;
cout << uczen.data_urodzenia.rr << endl;

// Wyświetlenie danych zapisanych w tablicy oceny:
cout << uczen.oceny[0] << endl;
cout << uczen.oceny[1] << endl;
cout << uczen.oceny[2] << endl;

return 0;
}

```

W programie zademonstrowano wykorzystanie C-struktury, która zawiera pola należące do typów złożonych. Mianowicie zdefiniowano dwa typy struktury: `Data` i `Uczen`. Typ struktury `Data` jest typem pola (zmiennej członkowskiej) o nazwie `data_urodzenia` w C-strukturze `Uczen`. Mamy więc tutaj do czynienia ze strukturami zagnieżdzonymi. Ponadto pole o nazwie `oceny` stanowi tablicę liczbową składającą się z trzech elementów składowych.

Wszystkie pola zmiennej struktury `uczen` zostały zainicjowane podczas jej deklarowania. Prezentacja wartości zainicjowanych pól na ekranie monitora ma na celu zilustrowanie sposobu odwoływania się do pól struktury będących innymi strukturami lub tablicami.

Ćwiczenie 7.5

Zmodyfikuj program z przykładu 7.5 — zamiast danych ucznia zapisanych w strukturze `uczen` typu `Uczen` należy zapamiętać w strukturze `pracownik` typu `Pracownik` dane pracownika: imię, nazwisko, datę zatrudnienia (C-struktura) oraz wykaz kwot premii kwartalnych w poprzednim roku (tablica liczbową).

7.1.4. C-struktury i wskaźniki

Dane przechowywane w polach (elementach członkowskich) C-struktur można przetwarzać przy użyciu wskaźników — analogicznie jak inne zmienne, np. zmienne należące do typów podstawowych. To samo dotyczy alokacji pamięci operacyjnej dla zmiennych struktury, która to pamięć może być przydzielana statycznie podczas komplikacji programu albo dynamicznie w trakcie jego wykonywania.

Wskaźniki do struktur

Jeżeli zmienna należąca do typu struktury została w programie zadeklarowana w sposób jawnny, pamięć operacyjna dla tej zmiennej jest przydzielana statycznie w trakcie komplikacji programu. Pamięć ta zostaje zaalokowana na stosie i jej wielkość nie zmienia się w czasie wykonywania programu.

Adres początku bloku (segmentu) pamięci zaalokowanej dla C-struktury (zmiennej strukturowej) można uzyskać za pomocą operatora adresu &, o którym była mowa w rozdziale 4. Adres ten można skopiować (przypisać) do wskaźnika należącego do typu wskaźnikowego określonego za pomocą nazwy typu bazowego, który stanowi w tym przypadku typ struktury.

Na przykład przy uwzględnieniu deklaracji zmiennej struktury pracownik należącej do typu struktury Pracownik: Pracownik pracownik; adres zmiennej pracownik można uzyskać za pomocą wyrażenia &pracownik. Po zadeklarowaniu wskaźnika wsk: Pracownik *wsk;, można mu przypisać określony wcześniej adres zmiennej pracownik: wsk = &pracownik;;

Dostęp do elementów członkowskich, czyli pól struktury, można uzyskać za pomocą wskaźnika przy użyciu **operatora wyboru elementu członkowskiego** (ang. *member selection operator*), ->, oraz nazwy pola struktury. Ogólna postać takiego odwołania jest następująca:

wskaznik->nazwa_pola

gdzie:

- **wskaznik** to zmienna wskaźnikowa, w której pamiętany jest adres zmiennej struktury należącej do określonego typu struktury,
- **nazwa_pola** to identyfikator wybranego pola (zmiennej członkowskiej) struktury.

UWAGA

Operator -> jest przez wielu programistów nazywany **operatorem strzałkowym** (ang. *arrow operator*).

Przykład 7.6

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

// Definicja typu struktury (C-struktury) Pracownik:
struct Pracownik {
    char imie[20];
    char nazwisko[20];
    char stanowisko[30];
};

int main() {
    // Deklaracja i inicjalizacja zmiennej pracownik należącej do typu struktury Pracownik:
    Pracownik pracownik = {"Jan", "Kowalski", "dyrektor"};

    // Deklaracja wskaźnika, który może wskazywać na zmienną typu Pracownik:
    Pracownik *wsk;
```

```

// Przypisanie wskaźnikowi wsk adresu zmiennej struktury pracownik:
wsk = &pracownik;

// Prezentacja danych zapisanych w strukturze pracownikI:
cout << "Dane pracownika " << endl;
cout << "Imię: " << wsk->imie << endl;
cout << "Nazwisko: " << wsk->nazwisko << endl;
cout << "Stanowisko: " << wsk->stanowisko << endl;

return 0;
}

```

W programie wykorzystano C-strukturę o zasięgu globalnym o nazwie `Pracownik`. Deklaracja zmiennej struktury `pracownik` należącej do typu struktury `Pracownik` została zainicjowana w kodzie programu głównego.

Wskaźnikowi `wsk` należącemu do typu wskaźnikowego `Pracownik*` przypisano wartość adresu zmiennej `pracownik`: `wsk = &pracownik;`. Dzięki temu odwołania do poszczególnych pól struktury `pracownik` można zrealizować przy wykorzystaniu operatora strzałkowego `->`, np. `wsk->imie`, `wsk->nazwisko`.

Ćwiczenie 7.6

Na podstawie programu z przykładu 7.6 napisz nowy program, pozwalający na zapamiętanie w polach C-struktury danych ucznia: imienia, nazwiska, płci i numeru w dzienniku. Dostęp do poszczególnych pól struktury zrealizuj za pomocą operatora strzałkowego `->`.

Dostęp do elementów członkowskich zmiennej struktury można uzyskać również za pomocą wskaźnika przy użyciu **operatora dereferencji** (ang. *dereference operator*), `*`, oraz nazwy pola struktury. W tym przypadku ogólna postać odwołania do pola struktury jest następująca:

`(*wskaźnik).nazwa_pola`

Zilustrowano to w przykładzie 7.7 za pomocą programu o identycznej funkcjonalności jak w przykładzie 7.6, ale przy wykorzystaniu operatora dereferencji `*`.

Przykład 7.7

```

#include <iostream>
using namespace std;

// Definicja typu struktury (C-struktury) Pracownik:
struct Pracownik {
    char imie[20];
    char nazwisko[20];
    char stanowisko[30];
};


```

```

int main() {
    // Deklaracja i inicjalizacja zmiennej struktury pracownik typu Pracownik:
    Pracownik pracownik = {"Jan", "Kowalski", "dyrektor"};
    // Deklaracja i inicjalizacja wskaźnika wsk:
    Pracownik *wsk = &pracownik;
    /* UWAGA
     * Po wykonaniu instrukcji powyżej wskaźnik wsk wskazuje na strukturę pracownik należącą do typu Pracownik.
     */

    cout << "Dane pracownika " << endl;
    cout << "Imię: " << (*wsk).imie << endl;
    cout << "Nazwisko: " << (*wsk).nazwisko << endl;
    cout << "Stanowisko: " << (*wsk).stanowisko << endl;
    /* UWAGA
     * Odwołanie się do poszczególnych pól struktury pracownik jest realizowane za pomocą nazwy tego pola (np. imie,
     * nazwisko) oraz operatora dereferencji i wskaźnika: *wsk.
     */
    return 0;
}

```

Ćwiczenie 7.7

Na podstawie programu z przykładu 7.7 napisz nowy program, pozwalający na zapamiętanie w polach struktury danych ucznia: imienia, nazwiska, płci i numeru w dzienniku. Dostęp do poszczególnych pól struktury zrealizuj za pomocą wskaźnika oraz operatora dereferencji `*`.

Struktury dynamiczne

C-struktura, podobnie jak inne zmienne należące do typów podstawowych (np. `int`, `float`) oraz złożonych (takich jak tablice), może być alokowana w pamięci operacyjnej (na stercie) w sposób dynamiczny. Proces ten przebiega w czasie wykonywania programu (ang. *runtime memory allocation*).

Wielkość pamięci zarezerwowanej dla C-struktury zaalokowanej dynamicznie na stercie nie zmienia się po jej utworzeniu. Tym samym rozmiar C-struktury również nie może się zmienić. Z tego wynika ważna uwaga: w trakcie działania programu nie można ani dodawać, ani usuwać żadnych pól (zmiennych członkowskich) C-struktury.

W celu utworzenia na stercie C-struktury wykorzystuje się operator `new`. Zwolnienie pamięci operacyjnej zarezerwowanej dla struktury uzyskuje się zaś za pomocą operatora `delete`. Dostęp do elementów członkowskich struktury, dla której pamięć została przydzielona w sposób dynamiczny, jest realizowany przy użyciu wskaźnika należącego do typu wskaźnikowego, którego konstrukcja opiera się na bazowym typie struktury.



UWAGA

Tematyka dynamicznego przydziału pamięci operacyjnej dla zmiennych należących do typów podstawowych została omówiona wcześniej — w podrozdziale 4.3. To samo dotyczy operatorów `new` oraz `delete`.

Przykład 7.8

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

// Definicja typu struktury (C-struktury) Pracownik:
struct Pracownik {
    string imie, nazwisko, stanowisko;
};

int main() {
    // Deklaracja wskaźnika wsk, który może wskazywać na zmienne typu struktury Pracownik:
    Pracownik *wsk;
    // Utworzenie na stercie struktury typu Pracownik wskazywanej przez wskaźnik wsk:
    wsk = new Pracownik;
    /* UWAGA
     * W pamięci operacyjnej po wykonaniu powyższej instrukcji zostanie utworzona zmienność typu Pracownik.
     * Pamięć dla tej zmiennej została zaalokowana w sposób dynamiczny na stercie.
     * We wskaźniku wsk pamiętały jest adres początku bloku pamięci, w którym przechowywana
     * jest nowo utworzona zmienność.
     */

    // Wprowadzenie wartości pól struktury z klawiatury:
    cout << "Podaj dane pracownika:" << endl;
    cout << "Imię = ";
    // Odwołanie się do pola (zmiennej członkowskiej) struktury o nazwie imie:
    cin >> wsk->imie;
    /* UWAGA
     * Odwołanie do pól struktury następuje przez wykorzystanie wskaźnika na strukturę, operatora strzałkowego ->
     * i nazwy pola struktury: wskaźnik->nazwa_pola.
     */
    cout << "Nazwisko = ";
    cin >> wsk->nazwisko;
    cout << "Stanowisko = ";
    cin >> wsk->stanowisko;
```

```

// Prezentacja danych zapisanych w strukturze:
cout << "Dane pracownika " << endl;
cout << "Imię: " << wsk->imie << endl;
cout << "Nazwisko: " << wsk->nazwisko << endl;
cout << "Stanowisko: " << wsk->stanowisko << endl;

// Zwolnienie pamięci zarezerwowanej na stercie dla struktury typu Pracownik, wskazywanej przez wskaźnik wsk:
delete wsk;

return 0;
}

```

W programie wykorzystano C-strukturę (zmienną) należącą do typu struktury Pracownik, dla której pamięć operacyjna została zaalokowana dynamicznie na stercie. Zmienna typu Pracownik jest tworzona przy wykorzystaniu operatora new w trakcie wykonywania programu.

Wartości poszczególnych elementów członkowskich struktury są wprowadzane z klawiatury, a następnie wyświetlane kontrolnie na ekranie monitora. Odwołanie się do poszczególnych pól struktury uzyskuje się za pomocą wskaźnika wsk, operatora strzałkowego -> i nazwy pola (imie, nazwisko i stanowisko), tj. wsk->imie, wsk->nazwisko i wsk->stanowisko.

Po zakończeniu przetworzenia danych pamięć operacyjna (na stercie) zarezerwowana dla struktury jest zwalniana za pomocą operatora delete.

Ćwiczenie 7.8

Zmodyfikuj program z przykładu 7.8 — dostęp do poszczególnych pól C-struktury zrealizuj za pomocą wskaźnika wsk w połączeniu z operatorem dereferencji * oraz identyfikatorów pól struktury. Dane wejściowe mają być wprowadzane przy wykorzystaniu strumienia wejściowego cin za pomocą funkcji getline().

UWAGA

C-struktury jako parametry funkcji zostały omówione w podrozdziale 8.3.3 podręcznika.

7.2. C-unie

W języku C **unia** (ang. *union*) stanowi złożony typ danych, który można określić jako „C-strukturę z wariantami”. Unie, podobnie jak struktury, należą do typów danych definiowanych przez użytkownika.



UWAGA

W niniejszym podrozdziale omawiane są wyłącznie unie charakterystyczne dla języka C, a nie języka C++. W dalszej części podręcznika będą one nazywane **C-uniami** (ang. *C-unions*).

C-unię można porównać do ciężarówki — wywrotki, która może transportować różne ładunki (materiały), np. piasek, węgiel, żwir, ale należące do określonego, skończonego zbioru materiałów. Przy tym w danym momencie może ona przewozić tylko jeden określony materiał, np. tylko piasek. Nie jest możliwe, aby jednocześnie transportować tą ciężarówką np. zarówno piasek, jak i węgiel. W dziedzinie programowania ciężarówka reprezentuje C-unię, a materiały (należące do skończonego, określonego zbioru materiałów) — jej **elementy członkowskie** (ang. *members*).

Unie są przechowywane w pamięci operacyjnej w bloku o pojemności równej rozmiarowi największego z jej elementów członkowskich. Jest to pierwsza z dwóch kluczowych właściwości unii. Na przykład, jeżeli unia zawiera tylko dwa elementy, z których jeden należy do typu `short`, a drugi do typu `float`, cała unia zajmuje w pamięci obszar o pojemności 4 bajtów. Wynika to z faktu, że dane typu `float` wymagają 4 bajtów pamięci, a dane typu `short` — 2 bajtów. Druga ważna cecha unii polega na tym, że w danej chwili w pamięci operacyjnej może być przechowywana wyłącznie wartość pojedynczego elementu członkowskiego unii. W ogólności wartość każdego elementu członkowskiego unii jest pamiętaana w tym samym obszarze (bloku) pamięci — bloku odpowiadającym jej największemu elementowi członkowskiemu. Pamięć dla pozostałych (mniejszych) elementów składowych unii jest przydzielana w części (fragmencie) tego bloku. To trzecia ważna cecha unii.

Wymienione powyżej cechy C-unii odróżniają ją znaczco od C-struktury: po pierwsze, ilość pamięci, jakiej wymaga C-struktura, wynika z sumy rozmiarów jej elementów członkowskich, a nie, jak w przypadku unii, z rozmiaru największego elementu składowego; po drugie, w strukturze w danej chwili przechowywane są wartości wszystkich jej elementów członkowskich (a nie jednego, jak w unii); po trzecie, każdemu z pól struktury odpowiada inny, niezależny segment pamięci operacyjnej (a nie ten sam, jak w unii).

7.2.1. Definiowanie unii

Ażeby można było w programie korzystać z zasobów C-unii, należy ją wcześniej zdefiniować. Definicja unii może być połączona z jej deklaracją — analogicznie jak w przypadku C-struktur. Ogólna postać definicji C-unii jest następująca:

```
union nazwa {
    typ_1 nazwa_1;
    typ_2 nazwa_2;
    ...
}
```

gdzie:

- `union` to słowo kluczowe oznaczające definicję (deklarację) unii,
- `typ_1, typ_2` to typy danych, do których należą elementy członkowskie unii o identyfikatorach, odpowiednio, `nazwa_1` i `nazwa_2`.

W ogólności typy elementów członkowskich unii są dowolne. Mogą to być zarówno typy podstawowe (np. `int`, `float`, `char`), jak i typy złożone — w tym typy danych zdefiniowane przez użytkownika: tablice, C-struktury itp. Liczba elementów członkowskich unii również jest dowolna — w zależności od potrzeb.

7.2.2. Zmienne typu unijnego

Deklaracja zmiennej

Zmienne należące do typu unijnego deklaruje się w analogiczny sposób jak zmienne innych typów, np. zmienne typu strukturywego. Na przykład deklaracja zmiennej unijnej o nazwie `ocena` należącej do zdefiniowanego typu unijnego o nazwie `ocena` ma postać: `ocena ocena;`

Inicjalizacja zmiennej

Sposób inicjalizacji zmiennych typu unijnego wynika z jednej z podstawowych cech C-unii, a mianowicie z tego, że w danej chwili w takiej zmiennej może być przechowywana wartość wyłącznie jednego z jej elementów członkowskich. W związku z tym w procesie inicjalizacji zmiennej typu unijnego należy nadać wartość wyłącznie pojedynczemu elementowi unii. Ogólna postać deklaracji zmiennej unijnej połączona z jej inicjalizacją jest następująca:

`nazwa_typu nazwa_zmiennej = {.nazwa_członka = wyrażenie};`

lub

`nazwa_typu nazwa_zmiennej {.nazwa_członka = wyrażenie};`

gdzie:

- `nazwa_typu` to identyfikator typu unijnego (C-unii),
- `nazwa_zmiennej` to identyfikator zmiennej,
- `.` to operator wyboru elementu członkowskiego,
- `nazwa_członka` to identyfikator elementu członkowskiego unii,
- `wyrażenie` to wyrażenie, które w wyniku daje wartość stałą należącą do typu zgodnego z typem wartości elementu członkowskiego `nazwa_członka`.

7.2.3. Odwołanie się do elementu członkowskiego unii

Odwołanie się do elementu członkowskiego unii można zrealizować przy wykorzystaniu nazwy zmiennej unijnej, operatora wyboru . oraz nazwy elementu członkowskiego unii. Ogólna postać takiego odwołania jest następująca:

nazwa_zmiennej.nazwa_członka

Interpretacja identyfikatorów nazwa_zmiennej i nazwa_członka jest analogiczna jak w opisie inicjalizacji unii.

Przykład 7.9

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

// Definicja C-unii o nazwie Ocena:
union Ocena {
    short ocena_c; // ocena całkowita, np. 3
    float ocena_r; // ocena z połówkami, np. 3.5
};

int main() {
    // Deklaracja zmiennej ocena połączona z inicjalizacją wartością 4 jej elementu członkowskiego ocena_c:
    Ocena ocena = {.ocena_c = 4};

    // Rozmiar unii:
    cout << sizeof(ocena) << endl; // 4
    // Rozmiar elementu członkowskiego ocena_c:
    cout << sizeof(ocena.ocena_c) << endl; //2

    // Wyświetlenie wartości elementu członkowskiego ocena_c:
    cout << ocena.ocena_c << endl; // 4

    // Nadanie wartości elementowi członkowskemu ocena_r:
    ocena.ocena_r = 3.5;

    cout << sizeof(ocena) << endl; // 4
    cout << sizeof(ocena.ocena_r) << endl; // 4

    // Wyświetlenie wartości elementu członkowskiego ocena_r:
    cout << ocena.ocena_r << endl; // 3.5

    return 0;
}
```

W programie zdefiniowano C-unię o zasięgu globalnym i nazwie `ocena`. Unia ta zawiera dwa elementy członkowskie: `ocena_c` i `ocena_r`. Pierwszy z nich należy do typu `short`, a drugi — do typu `float`. Elementy te reprezentują oceny studenta, odpowiednio, całkowite, np. 3, 4, oraz „połówkowe”, np. 3.5, 4.5.

W programie głównym zadeklarowano i zainicjowano zmienną unijną o nazwie `ocena`. Inicjalizacja tej zmiennej polega tutaj na nadaniu wartości początkowej równej 4 jej elementowi członkowskiemu o nazwie `ocena_c`.

Odwołanie do tego elementu członkowskiego w celu wyświetlenia jego wartości na ekranie monitora ma postać: `ocena.ocena_c`. Następnie elementowi członkowskiemu o nazwie `ocena_r` nadano wartość równą 3.5. Odwołanie do tego elementu unii ma postać: `ocena.ocena_r`.

Wyświetlenie rozmiarów zmiennej `ocena` oraz elementów członkowskich `ocena_c` i `ocena_r` ma na celu zilustrowanie budowy wewnętrznej unii.

Ćwiczenie 7.9

Zmodyfikuj program z przykładu 7.9 — zdefiniuj C-unię pozwalającą na zapamiętanie oceny jako liczby całkowitej (np. 3) albo w postaci łańcucha znaków (np. „dostateczny”). Użytkownik programu ma wprowadzić z klawiatury — sekwencyjnie, jedna po drugiej — wartość jednej oceny w obu formach (liczbowej i słownej) jako elementy członkowskie unii. Spróbuj wyświetlić w pojedynczej instrukcji wartość tej oceny w obu formach. Zinterpretuj uzyskany wynik.

Elementy członkowskie unii mogą należeć również do typów złożonych, np. tablicowych, strukturowych. Była o tym mowa wcześniej — w podrozdziale 7.2.1. W przykładzie 7.10 zilustrowano przypadek, w którym elementami członkowskimi C-unii są C-struktury.

Przykład 7.10

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

// Definicja C-struktury pomocniczej Plane:
struct Plane {
    float x, y;
};

// Definicja C-struktury pomocniczej Space
struct Space {
    float x, y, z;
};

// Definicja C-unii Coordinates:
union Coordinates {
    Plane w2; // współrzędne punktu na płaszczyźnie
}
```

```

Space w3; // współrzędne punktu w przestrzeni 3-wymiarowej
};

int main() {
    // Deklaracja zmiennej unijnej o nazwie w, należącej do typu Coordinates:
    Coordinates w;

    // Wprowadzenie z klawiatury wartości pól x i y elementu członkowskiego w2:
    cout << "Podaj wartości współrzędnych punktu na płaszczyźnie: " << endl;
    cout << "x = ";
    cin >> w.w2.x;
    cout << "y = ";
    cin >> w.w2.y;
    // Wyświetlenie na ekranie wartości pól elementu członkowskiego w2:
    cout << "Wprowadzone współrzędne punktu: " << endl;
    cout << "x: " << w.w2.x << endl;
    cout << "y: " << w.w2.y << endl;

    // Wprowadzenie z klawiatury wartości pól w, y i z elementu członkowskiego w3:
    cout << "Podaj wartości współrzędnych punktu w przestrzeni: " << endl;
    cout << "x = ";
    cin >> w.w3.x;
    cout << "y = ";
    cin >> w.w3.y;
    cout << "z = ";
    cin >> w.w3.z;
    // Wyświetlenie na ekranie wartości pól elementu członkowskiego w3:
    cout << "Wprowadzone współrzędne punktu: " << endl;
    cout << "x: " << w.w3.x << endl;
    cout << "y: " << w.w3.y << endl;
    cout << "z: " << w.w3.z << endl;

    return 0;
}

```

W programie zdefiniowano dwie struktury o zasięgu globalnym: `Plane` oraz `Space`. Pierwsza z nich pozwala na zapamiętanie współrzędnych punktu na płaszczyźnie, a druga — współrzędnych punktu w przestrzeni trójwymiarowej. Wspomniane struktury zostały wykorzystane w definicji unii `Coordinates` jako typy jej elementów członkowskich.

W programie głównym zadeklarowano zmienną unijną o nazwie `w`, należącą do typu `Coordinates`. Można w niej przechować albo współrzędne punktu na płaszczyźnie, albo współrzędne punktu w przestrzeni. Pierwszy przypadek jest reprezentowany za pomocą elementu członkowskiego `w.w2`, a drugi — `w.w3`. Wartości współrzędnych w obu przypadkach są wprowadzane z klawiatury, a następnie kontrolnie wyświetlane na ekranie monitora.

Ćwiczenie 7.10

Zmodyfikuj program z przykładu 7.10 — zdefiniuj C-unie pozwalającą na zapamiętanie współrzędnych zadanego punktu na płaszczyźnie: albo w układzie współrzędnych kartezjańskich (jak w przykładzie 7.10), albo w układzie współrzędnych biegunowych (polarnych). Przetestuj poprawność merytoryczną zdefiniowanej w programie C-unii.



7.3. Pytania i zadania kontrolne

7.3.1. Pytania

- 1.** Czy elementy członkowskie (pola) C-struktury mogą należeć do różnych typów? Co oznacza stwierdzenie, że struktury są zaliczane do agregacyjnych typów danych?
- 2.** Czy C-struktury można alokować w pamięci operacyjnej w sposób dynamiczny? Czy liczba elementów członkowskich C-struktur może się zmienić w trakcie wykonywania programu?
- 3.** W jaki sposób można obliczyć wielkość pamięci operacyjnej przeznaczonej do przechowania C-struktury?
- 4.** Czy dostęp do elementów członkowskich (pół) C-struktur można uzyskać za pomocą wskaźników?
- 5.** Jakie są podstawowe różnice pomiędzy C-strukturami a C-uniami?
- 6.** Czy C-unie można zaliczyć do typów danych agregacyjnych? Uzasadnij odpowiedź.

7.3.2. Zadania

- 1.** Napisz program pozwalający przechować w polach C-struktury następujące dane o smartfonie: markę, model, cenę, przekątną ekranu. Dane te należy zainicjować w programie, po czym wyświetlić je kontrolnie na ekranie monitora.
- 2.** Napisz program umożliwiający zapisanie w C-strukturze następujących danych o samochodzie: marki, modelu, roku produkcji, daty pierwszej rejestracji. Wykorzystaj struktury zagnieżdżone. Dane te mają być wprowadzane z klawiatury i wyświetlane kontrolnie na ekranie monitora.
- 3.** Napisz program pozwalający obliczyć pole i obwód prostokąta. Parametry prostokąta powinny być przechowywane w C-strukturze jako jej elementy członkowskie (pola). Dostęp do pól struktury zrealizuj za pomocą wskaźników. Dane wejściowe (długość i szerokość prostokąta) mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora.
- 4.** Napisz program pozwalający obliczyć średnią arytmetyczną z ocen semestralnych uzyskanych przez ucznia z następujących przedmiotów: języka polskiego, języka angielskiego, matematyki, informatyki. Dane personalne ucznia — imię i nazwisko, należy przechować w C-strukturze jako jej elementy członkowskie. To samo dotyczy

ocen semestralnych ucznia, które powinny być zagregowane (zgrupowane) w tablicy będącej trzecim polem struktury. Wykorzystaj zmienną należącą do zdefiniowanego typu struktury zaalokowaną w pamięci operacyjnej w sposób dynamiczny. Dane wejściowe powinny być wprowadzane z klawiatury, a potem wyświetlane kontrolnie na ekranie monitora.

- 5.** Napisz program pozwalający obliczyć pole powierzchni i obwód albo kwadratu, albo prostokąta — w zależności od decyzji użytkownika. W celu zapamiętania danych wybranej figury wykorzystaj C-unię. Dane wejściowe (parametry figury) mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora.
- 6.** Napisz program pozwalający obliczyć długość odcinka zdefiniowanego za pomocą współrzędnych kartezjańskich jego wierzchołków. Przy tym odcinek ten jest położony albo na płaszczyźnie, albo w przestrzeni trójwymiarowej — w zależności od decyzji użytkownika. Wykorzystaj C-unię, której elementami członkowskimi są dwie C-struktury umożliwiające przechowanie współrzędnych określonego punktu na płaszczyźnie lub w przestrzeni trójwymiarowej. Dane wejściowe (współrzędne punktów) mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora.



Funkcje

Za rozwiązywanie postawionego problemu (zadania) programistycznego w całości odpowiada program — aplikacja (ang. *application*). Za rozwiązywanie określonego podproblemu składowego tego problemu zaś odpowiada podprogram (ang. *subroutine*). W programowaniu bardzo często występuje sytuacja, w której jest możliwe podzielenie podproblemów (zadań) składowych na jeszcze mniejsze jednostki — podproblemy składowe podproblemów nadrzędnych (ogólniejszych).

Podążając dalej tym tokiem myślenia — nasuwają się dwa sposoby rozwiązywania postawionego problemu programistycznego. Sposób pierwszy, który można nazwać podejściem *od ogólnego do szczegółu*, polega na tym, że w pierwszej kolejności określa się rozwiązanie ogólne zadania. Następnie rozwiązuje się zadania składowe mniej ogólnie, a potem zadania o jeszcze mniejszym stopniu ogólności. Na końcu rozwiązuje się zadania najbardziej uszczegółowione — elementarne, podstawowe. Natomiast korzystając z drugiego sposobu, należy postępować w odwrotnej kolejności. W tym przypadku najpierw rozwiązuje się podproblemy (zadania) elementarne — najbardziej uszczegółowione, w dalszej kolejności zadania coraz bardziej ogólne (mniej szczegółowe) itd. Ostatecznie takie działania powinny doprowadzić do rozwiązania całości postawionego problemu. Dlatego też sposób ten można określić mianem podejścia *od szczegółu do ogólnego*.

UWAGA

Przedstawione powyżej sposoby rozwiązywania problemów są związane tematycznie z szerszymi zagadnieniami, takimi jak programowanie proceduralne (ang. *procedural programming*) czy też programowanie strukturalne (ang. *structural programming*). Te zagadnienia zostały omówione w podrozdziale 11.1.

W języku C++ podprogramy nazywają się **funkcjami** (ang. *functions*). Zadaniem funkcji jest wykonanie (realizacja) określonego zadania, takiego jak wyświetlenie komunikatu na ekranie monitora, wprowadzenie danych wejściowych z klawiatury czy obliczenie pola figury geometrycznej. Funkcję stanowi zatem zestaw (blok) instrukcji o określonej funkcjonalności, który jest opatrzony nazwą (identyfikatorem). Funkcje mogą komunikować się ze swoim otoczeniem — z programem głównym (funkcją `main()`) czy też z inną funkcją.

Ażeby z określonej funkcji można było korzystać w programie, należy ją wcześniej zdefiniować. Definicje wielu przydatnych funkcji są zawarte w **bibliotece standardowej C++** (ang. *Standard C++ Library*). Dlatego też funkcje te są nazywane **funkcjami standardowymi** (ang. *standard functions*). Funkcje standardowe są **funkcjami predefiniowanymi** (ang. *predefined functions*), co oznacza, że zostały one zdefiniowane wcześniej i są gotowe do użycia. Do funkcji standardowych należą np. funkcje predefiniowane operujące na C-napisach, jak również funkcje łańcuchowe, o których była mowa w rozdziale 6. W celu wykorzystania w programie określonej funkcji standardowej, np. funkcji `length()`, należy do niego dołączyć bibliotekę (zbiór nagłówkowy), w której funkcja ta została zadeklarowana, np. `#include <string>`.

Drugą grupę funkcji wykorzystywanych w programach tworzą **funkcje zdefiniowane przez użytkownika** (ang. *user-defined functions*). Użytkownik jest w tym przypadku rozumiany jako programista.

Zdefiniowaną funkcję — niezależnie od tego, czy jest to funkcja standardowa, czy funkcja zdefiniowana przez użytkownika — można wykonać w funkcji `main()` lub innej. Działanie to nazywa się **wywoływaniem funkcji** (ang. *function call*). Wywołanie funkcji powoduje wykonanie zestawu (bloku) instrukcji zawartych w jej definicji.

8.1. Deklarowanie i definiowanie funkcji

8.1.1. Deklaracja funkcji

Deklaracja funkcji (ang. *function declaration*) ma za zadanie poinformowanie kompilatora, że, po pierwsze, funkcja o określonej nazwie (identyfikatorze) istnieje oraz, po drugie, w jaki sposób funkcja ta komunikuje się ze swoim otoczeniem, np. z programem głównym. Funkcja może się komunikować z otoczeniem na dwa sposoby:

- za pośrednictwem **wartości zwracanej przez funkcję** (ang. *value returned by function*),
- za pomocą **parametrów funkcji** (ang. *function parameters*).

W pierwszym przypadku określona pojedyncza wartość jest przekazywana w kierunku od funkcji do jej otoczenia. Może zatem odgrywać jedynie rolę wyjścia funkcji. Jest to komunikacja jednokierunkowa.

W drugim przypadku komunikacja pomiędzy funkcją a jej otoczeniem może być dwukierunkowa. Przy użyciu parametrów można zarówno dostarczać do funkcji wymagane dane (wejściowe), jak i przekazywać na zewnątrz funkcji przetworzone za jej pomocą dane (wyjściowe). Stąd parametry funkcji można podzielić na:

- **parametry wejściowe** (ang. *input parameters* lub *in parameters*),
- **parametry wyjściowe** (ang. *output parameters* lub *out parameters*).

Parametry wejściowe funkcji stanowią jej „wejście”, a wyjściowe — „wyjście”.

Ogólna postać deklaracji funkcji jest następująca:

typ_zwracany nazwa (lista_parametrów);

gdzie:

- typ_zwracany to typ wartości zwracanej przez funkcję na zewnątrz, np. `int`, `float`, `void`,
- nazwa to identyfikator funkcji,
- lista_parametrów to lista parametrów oddzielonych od siebie przecinkami.

Każdy z parametrów na liście_parametrów deklaruje się w analogiczny sposób jak zwykłe zmienne należące do typów podstawowych, tj.: typ_parametru nazwa_parametru. Na przykład deklaracja `float bok` oznacza, że parametr o nazwie `bok` należy do typu `float`. Parametry funkcji wymienione w jej deklaracji często nazywa się **parametrami formalnymi** (ang. *formal parameters*).

W przypadku ogólnym liczba parametrów funkcji jest dowolna. Tym samym można również deklarować funkcje bezparametrowe, czyli takie, które nie mają żadnych parametrów. Dowolne są także typy parametrów funkcji.

Jeśli w deklaracji funkcji jako typ wyniku zwracanego podano słowo kluczowe `void`, to oznacza, że funkcja nie zwraca na zewnątrz żadnej wartości. Takie funkcje można nazwać **bezrezultatowymi**. W przypadku ogólnym typ_wyniku zwracanego przez funkcje często nazywa się **typem funkcji**.

Deklaracje funkcji często są również nazywane **nagłówkami funkcji** (ang. *function headers*) lub **prototypami funkcji** (ang. *function prototypes*).

Przykład 8.1

Deklaracja:

`float polePr(float promien);`

informuje kompilator, że:

- `polePr` jest nazwą funkcji,
- funkcja zwraca na zewnątrz wartość typu `float`,
- funkcja ma jeden parametr (formalny) o nazwie `bok` należący do typu `float`.

Ćwiczenie 8.1

Napisz deklarację funkcji o nazwie `obwodPr`, która będzie miała dwa parametry formalne należące do typu `double` o nazwach `bok1` i `bok2` oraz zwracała na zewnątrz wartość typu `double`.

8.1.2. Definicja funkcji

Definicja funkcji wiąże deklarację (nagłówek) funkcji z jej **ciałem** (ang. *body*). Ciało funkcji określa jej **implementację** (ang. *implementation*) — czyli blok kodu, który zawiera zestaw instrukcji realizujących określone zadania (operacje). Definicja funkcji (czyli jej deklaracja wraz z ciałem) opisuje wyczerpująco funkcjonalność funkcji z uwzględnieniem sposobu, w jaki funkcja komunikuje się ze swoim otoczeniem.

Ogólna postać definicji funkcji jest następująca:

```
nagłówek_funkcji {
    zestaw_instrukcji;
}
```

gdzie **nagłówek_funkcji** stanowi deklarację funkcji, a **zestaw_instrukcji** to implementacja zadań (operacji) realizowanych przez funkcję.

Jeśli funkcja zwraca na zewnątrz określoną wartość, w ciele funkcji musi wystąpić co najmniej raz instrukcja:

```
return wyrażenie;
```

gdzie **wyrażenie** daje w wyniku wartość typu zgodnego z zadeklarowanym typem wartości zwracanej przez funkcję.

Przykład 8.2

Definicja funkcji o nazwie **polePr**, która zwraca na zewnątrz wartość typu **float** i ma dwa parametry należące do typu **float**: **bok1** i **bok2**, może mieć postać:

```
float polePr(float bok1, float bok2) {
    return bok1 * bok2;
}
```

Ćwiczenie 8.2

Napisz definicję funkcji o nazwie **obwodPr** pozwalającej obliczyć obwód prostokąta. Funkcja powinna mieć dwa parametry wejściowe, odpowiadające długościom boków prostokąta. Obliczoną wartość obwodu powinna przekazywać na zewnątrz.

8.2. Wywołanie funkcji

Zdefiniowane funkcje można wielokrotnie — w zależności od potrzeb — **wywoływać** (ang. *call*). W ogólności wywołanie funkcji to wyrażenie, które powoduje przeniesienie sterowania przebiegiem wykonywania programu do instrukcji zawartych w ciele funkcji. Instrukcje te są wykonywane sekwencyjnie jedna po drugiej. Zakończenie wykonywania funkcji następuje albo po wykonaniu instrukcji **return**, jeśli ta znajduje się w kodzie, albo po wykonaniu ostatniej z instrukcji w jej ciele. Pierwszy z podanych przypadków dotyczy

funkcji, które zwracają na zewnątrz określona wartość, a drugi — funkcji typu `void`. Po wykonaniu instrukcji zawartych w funkcji sterowanie jest przenoszone z powrotem do miejsca jej wywołania — do następnej instrukcji występującej w kodzie źródłowym programu po wyrażeniu, które zawiera wywołanie funkcji.

Ogólna postać wywołania funkcji jest następująca:

```
nazwa(lista_argumentów);
```

gdzie `nazwa` oznacza identyfikator funkcji, a `lista_argumentów` to zestaw oddzielonych przecinkami argumentów, które reprezentują kolejne parametry formalne funkcji wyszczególnione w jej deklaracji (definicji). Przy tym zarówno liczba argumentów, jak i ich typy powinny być zgodne z liczbą wspomnianych parametrów formalnych i odpowiadających im typów.

Przykład 8.3

Przy założeniu, że w programie zdefiniowano funkcję o nazwie `polePr`:

```
float polePr(float bok1, float bok2) {
    return bok1 * bok2;
}
```

oraz trzy zmienne:

```
float b1, b2, p;
```

wywołanie funkcji `polePr()` może mieć postać:

```
p = polePr(b1, b2);
```

W tym przypadku argument wywołania (zmienna) `b1` odpowiada parametrowi formalnemu `bok1`, a `b2` — parametrowi `bok2`. Wartość zwracana przez funkcję `polePr()` zostaje przypisana do zmiennej `p`.

UWAGA

W dalszej części podręcznika zapis `funkcja()` będzie oznaczać funkcję o nazwie `funkcja` — w odróżnieniu od identyfikatorów innych elementów programu, takich jak stałe i zmienne, które są zapisywane bez użycia nawiasów okrągłych.

Ćwiczenie 8.3

Na podstawie przykładu 8.3 zdefiniuj, a następnie wywołaj funkcję o nazwie `obwod`, której zadaniem jest obliczenie pola i obwodu prostokąta.

W przypadku ogólnym argumentem funkcji może być stała, zmienna, wyrażenie, a nawet inna funkcja — w zależności od rodzaju odpowiadającego mu parametru formalnego. Tematyka ta została omówiona w następnym podrozdziale (podrozdział 8.3), dotyczącym parametrów

funkcji. W praktyce argumenty funkcji często są nazywane jej **parametrami aktualnymi** — bieżącymi (ang. *actual parameters*).

Funkcja (nadrzędna), w której kodzie zawarte jest wywołanie innej funkcji, jest nazywana **funkcją wywołującą** (ang. *caller*). W praktyce rolę funkcji wywołującej może odgrywać funkcja `main()`, jak również dowolna inna.

8.3. Parametry funkcji

Parametry funkcji odgrywają rolę jej interfejsu, umożliwiającego jej dwukierunkowy kontakt z jej otoczeniem, np. funkcją `main()` lub innymi. W przypadku ogólnym parametry funkcji mogą odgrywać rolę zarówno jej wejścia, jak i wyjścia.

8.3.1. Parametry wejściowe

Parametry przekazywane przez wartość

Parametry (argumenty) wejściowe funkcji pozwalają dostarczać do niej wymagane dane (informacje). Dane te są następnie przetwarzane w funkcji w celu osiągnięcia założonego rezultatu (rezultatów). W przypadku gdy do funkcji należy dostarczyć dane, które konsumują niewiele pamięci operacyjnej (np. dane należące do typów podstawowych), parametry wejściowe są zazwyczaj **przekazywane przez wartość** (ang. *pass by value*). Ogólna postać takiego parametru w deklaracji (i definicji) funkcji jest następująca:

typ_parametru nazwa_parametru

gdzie `typ_parametru` określa typ danych, do którego należy parametr (np. `int`, `float`), a `nazwa_parametru` stanowi jego identyfikator.

Jeżeli dany parametr formalny w deklaracji (i definicji) funkcji jest przekazywany przez wartość, w chwili wywołania funkcji wartość odpowiadającego mu argumentu (np. zmiennej) jest do niego kopiowana. Operacje wewnętrz (w ciele) funkcji wykonywane przy użyciu tego parametru traktowanego jako operand (np. zmiana jego wartości) nie mają żadnego wpływu na wartość argumentu przekazanego do funkcji na poziomie funkcji wywołującej. Tym samym wartość ta się nie zmienia. Reasumując, jeśli argument dostarczony do funkcji odpowiada jej parametrowi przekazywanemu przez wartość, może on odgrywać rolę wyłącznie parametru wejściowego (wejścia) funkcji. Zmiana wartości takiego parametru w ciele funkcji nie skutkuje zmianą argumentu przekazywanego do funkcji z jej otoczenia (funkcji wywołującej).

W przypadku ogólnym parametrami aktualnymi funkcji (czyli argumentami) przekazywanymi przez wartość mogą być stałe, zmienne lub wyrażenia.

Przykład 8.4

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

// Definicja funkcji polePr():
double polePr(double b1, double b2) { // parametry przekazywane przez wartość
    // Operacje w ciele funkcji są realizowane przy użyciu jej parametrów:
    return b1 * b2; // funkcja zwraca na zewnątrz obliczoną wartość wyrażenia b1 * b2
}

/* UWAGA
* Funkcja polePr() ma dwa parametry wejściowe przekazywane przez wartość. Oba parametry należą do typu double.
* Funkcja polePr() zwraca na zewnątrz wartość typu double.
*/

// Deklaracja — prototyp funkcji obwodPr():
double obwodPr(double, double);
/* UWAGA
* Funkcja obwodPr() ma dwa parametry wejściowe przekazywane przez wartość. Oba parametry są typu double.
* Funkcja ta zwraca na zewnątrz wartość typu double.
*
* Definicja funkcji obwodPr() jest zamieszczona w dalszej części programu — poniżej programu głównego.
*/

int main() {
    // Deklaracja i inicjalizacja zmiennych pomocniczych:
    double bok1 = 1; // pierwszy bok prostokąta
    double bok2 = 2; // drugi bok prostokąta

    double pole; // pole prostokąta
    // Wywołanie funkcji polePr() jako części składowej wyrażenia:
    pole = polePr(bok1, bok2);
/* UWAGA
* W wyrażeniu powyżej wynik wykonania funkcji polePr() jest podstawiany do zmiennej pole.
* Argumentami wywołania funkcji polePr() są zmienne pomocnicze bok1 i bok2 zdefiniowane wcześniej.
*/

    double obwod = obwodPr(bok1, bok2); // wywołanie funkcji obwodPr()
/* UWAGA
* Wywołanie funkcji obwodPr() jest realizowane z dwoma argumentami: bok1 i bok2.
* Wynik wykonania funkcji zostaje przypisany do zmiennej obwod.
*/
    cout << "POLE I OBWÓD PROSTOKĄTA:" << endl;
    cout << "Pole = " << pole << endl;
    cout << "Obwód = " << obwod << endl;

    return 0;
}
```

```
// Definicja funkcji obwodPr();
double obwodPr(double b1, double b2) {
    return 2 * b1 + 2 * b2; //funkcja zwraca na zewnatrz wartosc wyrazenia 2 * b1 + 2 * b2
}
```

W programie obliczane są pole i obwód prostokąta. Wykorzystuje się do tego dwie zdefiniowane funkcje: `polePr()` i `obwodPr()`. Obie przekazują obliczone wartości na zewnątrz. Obie też mają po dwa parametry formalne przekazywane przez wartość: `b1` i `b2`. Tym samym parametry `b1` i `b2` odgrywają rolę wejścia funkcji.

Funkcja `obwodPr()` została najpierw zadeklarowana, a dopiero w dalszej części programu — poniżej funkcji `main()` — zdefiniowana.

Argumentami wywołania funkcji `polePr()` i `obwodPr()` są zmienne `bok1` i `bok2`, zadeklarowane i zainicjowane w funkcji `main()`. Argument `bok1` odpowiada parametrowi `b1`, a `bok2` — parametrowi `b2`.

Ćwiczenie 8.4

Zmodyfikuj program z przykładu 8.4 — zamiast pola i obwodu prostokąta oblicz pole i obwód koła.

Jeśli określone parametry funkcji są przekazywane do funkcji przez wartość, to w chwili wywołania funkcji tworzone są ich kopie. Kopie te są traktowane jako zmienne lokalne funkcji i składowane w pamięci operacyjnej na stosie. Funkcja przetwarza właśnie te kopie, a nie oryginały zmiennych. Tym samym instrukcje zawarte w ciele funkcji mogą zmienić jedynie wartości kopii, a nie oryginałów zmiennych będących argumentami funkcji.



UWAGA

Zmienne lokalne i globalne zostały omówione szczegółowo w podrozdziale 8.4.

Parametry jako referencje do stałych `const`

Jeżeli istnieje potrzeba dostarczenia do funkcji danej, która pochłania dużą ilość pamięci operacyjnej (np. rozbudowanej C-struktury), to zamiast przekazywać taką daną przez stałą, należy przekazać ją jako referencję do stałej `const`. Ogólna postać takiego parametru jest następująca:

`const typ_parametru& nazwa_parametru`

gdzie & jest **operatorem referencji** (ang. *reference operator*), a interpretacja pozostałych składników jest identyczna jak w przypadku parametrów przekazywanych przez stałą.

Ten sposób przekazywania danych wejściowych do funkcji gwarantuje, że funkcja nie ma możliwości jakiejkolwiek modyfikacji argumentu jej wywołania, który odpowiada parametrowi

przekazywanemu jako referencja do stałej `const`. Wynika to z faktu, że parametr/argument będący referencją do stałej zapewnia mu status „tylko do odczytu”.

W pamięci operacyjnej w chwili wywołania funkcji z argumentem będącym referencją do stałej `const`, która należy do określonego typu (np. typu struktury), nie jest oczywiście tworzona kopia tego argumentu. Do parametru funkcji podstawiana jest jedynie referencja (adres) faktycznego argumentu, a nie jego kopia. Zapewnia to bardziej efektywne gospodarowanie pamięcią operacyjną niż w przypadku przekazywania dużej porcji danych (np. C-struktury) przez stałą.

Przykład 8.5

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

// Prototypy funkcji:
float polePr(const float&, const float&); // parametry wejściowe jako referencje do stałych const
                           // typu float
float obwodPr(const float&, const float&); // parametry wejściowe jako referencje do stałych const
                           // typu float

int main() {
    float bok1 = 1; // pierwszy bok prostokąta
    float bok2 = 2; // drugi bok prostokąta

    // Wywołanie funkcji polePr():
    float pole = polePr(bok1, bok2);
    cout << "Pole wynosi " << pole << endl;

    // Wywołanie funkcji obwodPr():
    float obwod = obwodPr(bok1, bok2);
    cout << "Obwód wynosi " << obwod << endl;

    return 0;
}

// Definicje funkcji:
float polePr(const float& b1, const float& b2) {
    // Modyfikacja wartości parametru b1 nie jest możliwa, ponieważ ma on status „read only”:
    // b1 = 3;

    return b1 * b2;
}
float obwodPr(const float& b1, const float& b2) {
    return 2 * b1 + 2 * b2;
}
```

Funkcjonalność programu jest analogiczna do funkcjonalności programu zawartego w przykładzie 8.4 — również tutaj obliczane są pole i obwód prostokąta. Do wykonania niezbędnych obliczeń wykorzystywane są funkcje `polePr()` i `obwodPr()`. Jednakże tutaj funkcje te mają parametry wejściowe przekazywane przez referencje do stałych `const` typu `float`, a nie przez wartości, jak w przykładzie 8.4.

UWAGA

Wykorzystywanie parametrów wejściowych przekazywanych przez referencje do stałych `const` w odniesieniu do danych typów podstawowych (jak w przykładzie 8.5) nie jest zalecane. W praktyce parametry tego typu zazwyczaj są stosowane do przekazywania dużych porcji danych, np. struktur (C-struktur) — co zostało zaprezentowane w podrozdziale 8.3.3.

Ćwiczenie 8.5

Zmodyfikuj program z przykładu 8.5 — zamiast pola i obwodu prostokąta oblicz pole i obwód koła. Wykorzystaj parametry wejściowe przekazywane przez referencje do stałych `const`.

Parametry jako wskaźniki do stałych `const`

Tematyka wskaźników była omawiana wcześniej — w rozdziale 4. podręcznika. Tutaj zostanie przedstawione zagadnienie wykorzystania wskaźników jako parametrów wejściowych funkcji.

W przypadku parametrów wejściowych istotne jest to, aby podczas wykonywania funkcji modyfikacja wartości argumentów jej wywołania w ciele (wnętrzu) funkcji nie miała wpływu na wartości tych argumentów obowiązujące w otoczeniu funkcji — tj. w funkcji wywołującej, np. w funkcji `main()`. Cel ten można osiągnąć pośrednio, przez zadeklarowanie parametru formalnego funkcji jako wskaźnika do stałej `const`.

Ogólna postać parametru formalnego przekazywanego jako wskaźnik do stałej `const` jest następująca:

`const typ_parametru *nazwa_parametru`

Interpretacja poszczególnych składników w przedstawionej postaci ogólnej jest identyczna jak w przypadku parametrów przekazywanych przez stałą.

W ogólności po zastosowaniu w danej funkcji parametru (argumentu) jako wskaźnika do stałej `const` wszelkie operacje w jej ciele realizowane na tym argumencie są ograniczone klauzulą „tylko do odczytu”.

W pamięci operacyjnej w chwili wywołania funkcji z argumentem jako wskaźnikiem do stałej `const` wykonywana jest kopia tego wskaźnika (czyli adresu) — ale nie kopia wskazywanej przez niego zmiennej. Innymi słowy, wskaźnik (adres) do stałej `const` jest kopowany do parametru formalnego funkcji i w ciele funkcji operacje są realizowane przy użyciu tego parametru. Modyfikacja wartości parametru formalnego (kopii argumentu wskaźnikowego) nie ma żadnego wpływu na wartość argumentu wywołania funkcji (oryginału).

Ze względu na to, że w chwili wywołania funkcji z parametrem jako wskaźnikiem do stałej `const` nie jest wykonywana kopia zmiennej wskazywanej przez ten wskaźnik, omawiany rodzaj parametrów wejściowych jest szczególnie przydatny w sytuacji, gdy do funkcji przekazywana jest duża porcja danych, np. C-struktura, klasa.

UWAGA

W praktyce parametry wejściowe funkcji jako wskaźniki do stałych `const` zazwyczaj są wykorzystywane w celu dostarczenia do funkcji danych wejściowych tablicowych, co zostało omówione w podrozdziale 8.3.4. C-struktury należy przekazywać do funkcji z wykorzystaniem parametrów (argumentów) jako referencji do stałych `const`, o czym będzie mowa w podrozdziale 8.3.3. Z kolei zagadnienie przekazywania obiektów jako parametrów funkcji zostało przedstawione w podrozdziale 11.8.

Przykład 8.6

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

// Prototypy funkcji:
float polePr(const float*, const float*); // parametry wejściowe jako wskaźniki do stałych const
                           // typu float
float obwodPr(const float*, const float*); // parametry wejściowe jako wskaźniki do stałych const
                           // typu float

int main() {
    float bok1 = 1; // pierwszy bok prostokąta

    // Deklaracja i inicjalizacja wskaźnika w_bok1 wskazującego na zmienną bok1:
    float *w_bok1 = &bok1;

    float bok2 = 2; // drugi bok prostokąta
    // Deklaracja i inicjalizacja wskaźnika w_bok2 wskazującego na zmienną bok2:
    float *w_bok2 = &bok2;

    // Wywołanie funkcji polePr() z parametrami aktualnymi (argumentami) w_bok1 i w_bok2:
    float pole = polePr(w_bok1, w_bok2);
    cout << "Pole wynosi " << pole << endl;

    // Wywołanie funkcji obwodPr() z parametrami aktualnymi (argumentami) w_bok1 i w_bok2:
    float obwod = obwodPr(w_bok1, w_bok2);
    cout << "Obwód wynosi " << obwod << endl;

    return 0;
}
```

```
// Definicje funkcji:
float polePr(const float *b1, const float *b2) {
    // Modyfikacja wartości parametru *b1 nie jest możliwa, ponieważ ma on status „read only”:
    // *b1 = 3;

    return *b1 * *b2;
}

float obwodPr(const float *b1, const float *b2) {
    return 2 * *b1 + 2 * *b2;
}
```

W celach ilustracyjnych zaprezentowany program ma identyczną funkcjonalność jak w dwóch poprzednich przykładach: 8.4 i 8.5. Różnice występują w implementacjach i wywołaniach funkcji `polePr()` i `obwodPr()`. Tutaj funkcje te mają parametry będące wskaźnikami do stałych `const` typu `float`.

Przedstawiona w programie metoda nie jest zalecana do danych należących do typów podstawowych. Zwykle stosuje się ją do przekazywania tablic, rzadziej do struktur i klas.

Ćwiczenie 8.6

Zmodyfikuj program z przykładu 8.6 — zamiast pola i obwodu prostokąta oblicz pole i obwód koła. Wykorzystaj parametry wejściowe jako wskaźniki do stałych `const`.

8.3.2. Parametry wyjściowe

Parametry przekazywane przez referencję

Parametry (argumenty) funkcji **przekazywane przez referencję** (ang. *pass by reference*) mogą stanowić zarówno jej wyjście, jak i wejście.

Jeżeli parametr funkcji jest przekazywany przez referencję, w chwili jej wywołania do tego parametru kopiwana jest referencja do zmiennej będącej argumentem funkcji. Tym samym w ciele funkcji przetwarzana jest nie kopia argumentu funkcji (jak w przypadku argumentów przekazywanych przez wartość), lecz jej oryginał. Ta kluczowa właściwość omawianego sposobu komunikowania się funkcji z jej otoczeniem pozwala na przekazywanie na zewnątrz funkcji wyników wykonywanych w niej operacji za pomocą argumentów. Innymi słowy, parametry/argumenty funkcji mogą odgrywać rolę nie tylko wejścia funkcji, ale także jej wyjścia.

Należy zwrócić uwagę na to, że w funkcji przetwarzany jest oryginał argumentu (zmiennej), a nie kopia — o czym wspomniano powyżej. W związku z tym modyfikacja wartości argumentu w ciele funkcji (traktowanego jako jej wejście) będzie skutkować tym, że zmiana ta będzie również obowiązywać w otoczeniu funkcji — tj. na poziomie funkcji wywołującej.

Ogólna postać parametru przekazywanego przez referencję jest następująca:

typ_parametru& nazwa_parametru

gdzie typ_parametru określa typ danych, do którego należy parametr, a nazwa_parametru to jego identyfikator.

Przykład 8.7

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

// Deklaracje — prototypy funkcji:
void polePr(double, double, double&);
void obwodPr(double, double, double&);

/* UWAGA
 * Funkcje polePr() i obwodPr() mają po trzy parametry. Dwa pierwsze są przekazywane przez wartość, a trzeci —
 * przez referencję. Słowo kluczowe void oznacza, że funkcja nie zwraca na zewnątrz żadnej wartości.
 */

int main() {
    double bok1 = 1; // pierwszy bok prostokąta
    double bok2 = 2; // drugi bok prostokąta
    double pole; // pole prostokąta

    // Wywołanie funkcji polePr():
    polePr(bok1, bok2, pole);
    /* UWAGA
     * Wywołanie funkcji polePr() typu void jest niezależną (samodzielnią) instrukcją.
     * Dwa pierwsze argumenty wywołania są przekazywane do funkcji przez wartość, a trzeci argument —
     * przez referencję, jak to określono w deklaracji funkcji. Oznacza to, że na stosie tworzone są kopie
     * zmennych bok1 i bok2. Są to zmienne lokalne o nazwach b1 i b2, odpowiadające parametrom funkcji.
     * Innymi słowy, parametrom b1 i b2 w definicji funkcji przypisane zostają wartości kopii argumentów
     * jej wywołania — bok1 i bok2. Dla argumentu referencyjnego o nazwie pole jej kopia nie jest wykonywana,
     * ponieważ nie jest potrzebna. Wynika to z faktu, że w chwili wywołania funkcji referencja do zmiennej pole,
     * będącej jej argumentem, jest kopiowana do parametru poleP. Wszystkie operacje odbywają się na oryginale
     * zmiennej pole, pod nazwą aliasową poleP.
    */

    double obwod; // obwód prostokąta

    // Wywołanie funkcji obwodPr():
    obwodPr(bok1, bok2, obwod);
    /* UWAGA
     * Argumentami wywołania funkcji obwodPr() są zmienne: bok1, bok2 i obwod.
    */
}
```

* Argumenty `bok1` i `bok2` stanowią wejście funkcji — są przekazywane przez wartość.
 * Argument `obwod` odgrywa rolę wyjścia funkcji — jest on przekazywany przez referencję.
 */

```

cout << "Wyniki:" << endl;
cout << "Pole wynosi " << pole << endl;
cout << "Obwód wynosi " << obwod << endl;

return 0;
}

// Definicje funkcji polePr() i obwodPr():
void polePr(double b1, double b2, double &poleP) {
    poleP = b1 * b2;
}
void obwodPr(double b1, double b2, double &obwodP) {
    obwodP = 2 * b1 + 2 * b2;
}
  
```

Funkcjonalność przedstawionego programu jest analogiczna do funkcjonalności programów zawartych w przykładach: 8.4, 8.5 i 8.6. Również tutaj obliczane są pole i obwód prostokąta.

Jednakże w prezentowanym programie funkcje `polePr()` i `obwodPr()` dostarczają obliczone wartości pola i obwodu do programu głównego (czyli funkcji wywołującej) za pomocą argumentów przekazywanych przez referencję, a nie, jak w wymienionych powyżej przykładach, za pośrednictwem wartości zwracanych przez funkcje.

Deklaracje wspomnianych funkcji są zawarte w kodzie programu powyżej programu głównego, natomiast ich definicje — w końcowej części kodu.

Ćwiczenie 8.7

Zmodyfikuj program z przykładu 8.7 — zamiast pola i obwodu prostokąta oblicz pole i obwód koła.

W następnym programie, zawartym w przykładzie 8.8, zilustrowano sytuację, w której rolę wyjścia funkcji odgrywają zarówno wartości, które funkcja zwraca, jak i parametry/argumenty przekazywane przez referencję.

Przykład 8.8

```

#include <iostream>
using namespace std;

// Deklaracje funkcji:
double polePr(double, double, double&);

  
```

```

double obwodPr(double, double, double&);
/* UWAGA
 * Funkcje polePr() i obwodPr() mają po trzy parametry. Dwa pierwsze są przekazywane do tych funkcji przez wartość,
 * a trzeci — przez referencję. Funkcje przekazują na zewnątrz obliczone wartości za pośrednictwem wartości
 * zwracanych oraz za pomocą parametru referencyjnego.
 */

int main() {
    double bok1 = 1;
    double bok2 = 2;

    double pole = polePr(bok1, bok2, pole); // wywołanie funkcji polePr()
    /* UWAGA
     * Wynik wykonania funkcji polePr(), przekazywany do programu głównego za pośrednictwem wartości zwracanej,
     * został przypisany do zmiennej pole.
    */

    double obwod;
    // Wywołanie funkcji obwodPr():
    obwodPr(bok1, bok2, obwod); // wywołanie funkcji obwodPr()
    /* UWAGA
     * Wynik wykonania funkcji obwodPr() został przekazany przez referencję do argumentu stanowiącego
     * zmenną obwod.
    */

    cout << "Wyniki:" << endl;
    cout << "Pole wynosi " << pole << endl;
    cout << "Obwód wynosi " << obwod << endl;

    return 0;
}

// Definicje funkcji:
double polePr(double b1, double b2, double &poleP) {
    poleP = b1 * b2; // ustalenie wartości parametru wyjściowego poleP

    // Funkcja zwraca na zewnątrz wartość parametru wyjściowego poleP:
    return poleP;
}
double obwodPr(double b1, double b2, double &obwodP) {
    obwodP = 2 * b1 + 2 * b2;
    return obwodP;
}

```

W programie wykorzystano dwie funkcje: `polePr()` oraz `obwodPr()`. Zadaniem pierwszej z nich jest obliczenie pola prostokąta, a drugiej — obwodu. Wymienione funkcje mają po dwa parametry wejściowe, reprezentujące długości boków prostokąta. Wyjściami funkcji `polePr()` i `obwodPr()` są zarówno wartości zwarcane, jak i parametry (argumenty) wyjściowe przekazywane przez referencję.

Wywołanie funkcji `polePr()` jest częścią składową wyrażenia, w którym do zmiennej `pole` przypisany zostaje zwrócony przez nią wynik. W drugim przypadku wywołanie funkcji `obwodPr()` jest niezależną instrukcją prostą, w której wynik działania funkcji zostaje podstawiony do zmiennej `obwod` za pośrednictwem argumentu przekazanego przez referencję.

Ćwiczenie 8.8

Zmodyfikuj program z przykładu 8.8 — zamiast pola i obwodu prostokąta oblicz pole i obwód koła.

Wskaźniki jako parametry funkcji

W podrozdziale 8.3.1 została omówiona tematyka wykorzystania wskaźników do stałych `const` jako parametrów wejściowych funkcji. Jednakże wskaźniki mogą odgrywać rolę nie tylko wejścia funkcji, ale również jej wyjścia.

Rozważając ten drugi przypadek, tj. wyjście funkcji, jeśli argumentem funkcji jest wskaźnik do zmiennej przechowywanej w pamięci operacyjnej, w chwili wywołania takiej funkcji adres tej zmiennej jest kopiowany do parametru formalnego funkcji. Operacje w ciele funkcji wykonywane przy wykorzystaniu tego parametru skutkują zmianą wartości zmiennej przechowywanej przez argument funkcji obowiązującej w jej otoczeniu, czyli w funkcji wywołującej. Wynika to z faktu, że zmienna wskazywana przez wskaźnik będący parametrem formalnym funkcji to ta sama zmienna, która jest wskazywana przez argument funkcji.

Ogólna postać parametru przekazywanego przez wskaźnik jest następująca:

`typ_parametru *nazwa_parametru`

gdzie `typ_parametru` określa typ danych, do którego należy parametr, a `nazwa_parametru` to jego identyfikator.

Jeśli wskaźnik do określonej zmiennej stanowi parametr wejściowy funkcji, to zmiana wartości zmiennej wskazywanej przez ten wskaźnik w ciele funkcji oznacza, że zmiana ta będzie obowiązywać również w funkcji wywołującej. Dlatego też w celu zabezpieczenia się przed taką sytuacją należy używać parametru jako wskaźnika do stałej `const` — o czym była mowa wcześniej, w podrozdziale 8.3.1.

Analogicznie do parametru/argumentu referencyjnego wskaźnik do zmiennej może zostać przekazany na zewnątrz funkcji w dwojakim sposobie. Pierwszy z nich polega na użyciu wskaźnika jako wartości zwarcanej przez funkcję, a drugi — na wykorzystaniu tego wskaźnika jako jej parametru (argumentu).

Przykład 8.9

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

// Deklaracje funkcji:
void polePr(double, double, double*);  

/* UWAGA  

* Funkcja typu void polePr() ma dwa parametry wejściowe przekazywane przez wartość i jeden parametr wyjściowy  

* przekazywany przez wskaźnik.  

*/

double* obwodPr(const double*, const double*, double*);  

/* UWAGA  

* Funkcja obwodPr() ma trzy parametry formalne. Pierwsze dwa z nich są parametrami wejściowymi  

* będącymi wskaźnikami do stałych const, a trzeci to parametr wyjściowy przekazywany przez wskaźnik.  

* Funkcja zwraca na zewnątrz wskaźnik równoważny parametrowi wyjściowemu funkcji.  

*/

int main() {
    double bok1 = 1; // pierwszy bok prostokąta
    double bok2 = 2; // drugi bok

    double pole; // pole prostokąta
    // Deklaracja i inicjalizacja wskaźnika do zmiennej pole:
    double *w_pole = &pole;

    // Wywołanie funkcji polePr():
    polePr(bok1, bok2, w_pole);
    /* UWAGA  

* Wywołanie funkcji polePr() stanowi instrukcję prostą.  

* Argument wskaźnikowy w_pole odgrywa rolę wyjścia funkcji.  

*/

    // Deklaracja i inicjalizacja wskaźników na zmienne bok1 i bok2:
    double *w_bok1 = &bok1;
    double *w_bok2 = &bok2;

    double obwod; // obwód prostokąta
    // Deklaracja i inicjalizacja wskaźnika na zmienną obwod:
    double *w_obwod = &obwod;

    // Wywołanie funkcji obwodPr():
    w_obwod = obwodPr(w_bok1, w_bok2, w_obwod);
```

```

/* UWAGA
 * Wszystkie argumenty wywołania funkcji obwodPr() są wskaźnikami.
 * Argumenty w_bok1 i w_bok2 stanowią wejście funkcji, ponieważ są przekazywane jako wskaźniki
 * do stałych const typu double.
 * Wynika to z definicji funkcji. Parametr w_obwod reprezentuje jedno z dwóch wyjść funkci.
 * Drugi wygliem jest wartość zwracana przez funkcję.
 */

cout << "Wyniki:" << endl;
cout << "Pole wynosi " << pole << endl;
cout << "Obwód wynosi " << obwod << endl;

return 0;
}

// Definicje funkcji polePr() i obwodPr():
void polePr(double b1, double b2, double *w_poleP) {
    *w_poleP = b1 * b2;
}

/* UWAGA
 * Parametry formalne b1 i b2 funkcji polePr() stanowią wejście funkcji, ponieważ są przekazywane przez wartość.
 * Parametr wskaźnikowy w_pole z kolei reprezentuje wyjście funkcji.
 */
double* obwodPr(const double *w_b1, const double *w_b2, double *w_obwodP) {
    *w_obwodP = 2 * *w_b1 + 2 * *w_b2;

    // Funkcja przekazuje na zewnątrz wartość parametru wyjściowego w_obwod:
    return w_obwodP;
}

/* UWAGA
 * Parametry w_b1 i w_b2 stanowią wejście funkcji obwodPr(), ponieważ są przekazywane jako wskaźniki
 * do stałych const typu double. Wyjście funkcji jest reprezentowane przez parametr wskaźnikowy w_obwod
 * oraz zwracaną przez nią wartość.
 */

```

W programie zdefiniowano dwie funkcje: `polePr()` i `obwodPr()`. Pierwsza z nich zwraca na zewnątrz obliczoną wartość za pośrednictwem parametru/argumentu wskaźnikowego. Wejściem tej funkcji są parametry przekazywane przez wartość. Z kolei funkcja `obwodPr()` ma dwa wyjścia. Pierwsze z nich jest reprezentowane przez parametr wskaźnikowy, a drugie — wartość zwracaną. Funkcja ta ma dwa parametry wejściowe będące wskaźnikami do stałych `const`.

Należy pamiętać o tym, że jeżeli do funkcji trzeba dostarczyć jedną lub więcej wartości należących do typów podstawowych (jak tutaj), zalecane jest stosowanie parametrów wejściowych przekazywanych przez wartość. Wykorzystanie w funkcji `obwodPr()` parametrów wejściowych będących wskaźnikami do stałych `const` ma cel wyłącznie ilustracyjny.

Ćwiczenie 8.9

Zmodyfikuj program z przykładu 8.9 — zamiast pola i obwodu prostokąta oblicz pole i obwód koła. Jako parametry wyjściowe funkcji wykorzystaj wskaźniki.

8.3.3. C-struktury jako parametry funkcji

C-struktury zostały omówione w podrozdziale 7.1. Należą one do złożonych typów danych odziedziczonych przez C++ po języku C. W zależności od potrzeb C-struktura może zawierać nawet dużą liczbę zmiennych członkowskich (pół), które w sumie mogą zajmować sporo miejsca w pamięci operacyjnej.

Przesyłanie rozbudowanej (pamięciożernej) C-struktury do funkcji z wykorzystaniem parametru przekazywanego przez wartość wiążałoby się z tworzeniem kopii tej struktury w pamięci operacyjnej — co nie jest rozwiązaniem efektywnym. Dlatego też, jeśli w programie konieczne jest przekazanie C-struktury do funkcji jako jej parametru wejściowego, należy to zrobić za pomocą parametru referencyjnego do stałej `const` typu struktury. Takie rozwiązanie jest skuteczne i zapewnia efektywne gospodarowanie pamięcią operacyjną komputera.

Najbardziej zaś wydajnym (i zalecanym) sposobem przekazania C-struktury do otoczenia funkcji jest użycie parametru referencyjnego, tj. referencji do C-struktury, albo przekazanie jej po prostu jako wartości zwracanej przez funkcję.

Przykład 8.10

```
#include <iostream>

// Definicja typu struktury Pracownik:
struct Pracownik {
    int id;
    char imie[20];
    char nazwisko[30];
};

// Prototypy funkcji:
void pobierzDane(Pracownik&); // parametr formalny funkcji przekazywany przez referencję
void wyswietlDane(const Pracownik&); // parametr przekazywany przez referencję do stałej const typu
                           // Pracownik

using namespace std;
int main() {
    // Deklaracja zmiennej pracownik należącej do typu struktury Pracownik:
    Pracownik pracownik;

    cout << "Wprowadź dane wejściowe:" << endl;
    // Wywołanie funkcji pobierzDane():
}
```

```

pobierzDane(pracownik);
/* UWAGA
 * Argumentem wywołania funkcji pobierzDane() jest referencja do zmiennej struktury o nazwie pracownik
 * należącej do typu struktury Pracownik.
 */

cout << "\n\nWprowadzone dane: " << endl;
// Wywołanie funkcji wyswietlDane():
wyswietlDane(pracownik);
/* UWAGA
 * Argumentem wywołania funkcji wyswietlDane() jest referencja do zmiennej pracownik typu Pracownik,
 * traktowanej jak stała const.
 */

return 0;
}

// Definicje funkcji:
void pobierzDane(Pracownik& p) {
    cout << "numer identyfikacyjny = ";
    cin >> p.id;
    cout << "imię = ";
    cin >> p.imie;
    cout << "nazwisko = ";
    cin >> p.nazwisko;
}
/* UWAGA
 * Funkcja pobierzDane() ma jeden parametr formalny, który odgrywa rolę jej wyjścia.
 * Parametr ten jest przekazywany przez referencję do zmiennej należącej do typu struktury Pracownik.
 */
void wyswietlDane(const Pracownik& p) {
    cout << "Nr identyfikacyjny: " << p.id << endl;
    cout << "Imię: " << p.imie << endl;
    cout << "Nazwisko: " << p.nazwisko << endl;
}
/* UWAGA
 * Funkcja wyswietlDane() ma jeden parametr formalny, reprezentujący jej wejście.
 * Parametr ten jest przekazywany jako referencja do stałej const należącej do typu struktury Pracownik.
 */

```

W programie zadeklarowano, a następnie zdefiniowano dwie funkcje: `pobierzDane()` i `wyswietlDane()`. Zadaniem funkcji `pobierzDane()` jest pobranie wartości zmiennych członkowskich (pół) struktury typu `Pracownik` z klawiatury, a funkcji `wyswietlDane()` — wyświetlenie na ekranie monitora danych zapisanych w polach struktury.

Obie z wymienionych funkcji mają po jednym parametrze referencyjnym. W szczególności funkcja `pobierzDane()` ma jeden parametr, który odgrywa rolę jej wyjścia. Parametr ten jest przekazywany przez referencję. Druga funkcja, `wyswietlDane()`, ma jeden parametr wejściowy, będący referencją do stałej `const` typu `Pracownik`.

Jako podsumowanie przykładu można sformułować dwie zasady — zalecenia:

- C-struktura jako parametr wejściowy funkcji powinna być do niej przekazywana jako referencja do stałej `const` typu struktury,
- C-strukturę jako parametr wyjściowy funkcji należy przekazywać do jej otoczenia (funkcji wywołującej) przez referencję do zmiennej należącej do typu struktury.

Ćwiczenie 8.10

Zmodyfikuj program z przykładu 8.10 — dane wejściowe (zmienne członkowskie) C-struktury `pracownik` mają być wprowadzane z klawiatury za pomocą funkcji `pobierzDane()`, która przekazuje wspomnianą C-strukturę do otoczenia jako wartość zwracaną.

UWAGA

Zagadnienie przekazywania obiektów jako instancji klas i struktur języka C++ jako parametrów (argumentów) funkcji zaprezentowano w dalszej części podręcznika — w rozdziale 11.

8.3.4. Tablice jako parametry funkcji

Przetwarzanie tablic w języku C++ jest realizowane przy wykorzystaniu wskaźników. W szczególności dotyczy to operacji nazywanych arytmetyką wskaźnika, które zostały omówione w podrozdziale 5.2.

W praktyce programistycznej bardzo często jest konieczne dostarczenie tablic do funkcji jako ich parametrów (danych) wejściowych, jak również przekazanie tablic z funkcji do ich otoczenia. Można to zrealizować z zastosowaniem charakterystycznych właściwości tablic w C++, a głównie tej, że w nazwie zmiennej tablicowej pamiętany jest adres jej pierwszego elementu. Stąd nawet gdyby jako parametru/argumentu funkcji użyć nazwy tablicy, to i tak niezmiennie stanowiłyby ona parametr/argument wskaźnikowy. Tym samym nie jest możliwe przekazanie całej tablicy ani do funkcji, ani z funkcji do jej otoczenia. Można jedynie przekazać wskaźnik (adres) jej pierwszego elementu z wykorzystaniem innej ważnej właściwości tablic w C++ — tego, że zajmuje ona ciągły obszar pamięci operacyjnej.

Zgodnie z zasadami arytmetyki wskaźników można przetwarzać tablice przy użyciu notacji tradycyjnej oraz notacji wskaźnikowej. Była o tym mowa w podrozdziale 5.2. To samo dotyczy sposobów zapisu parametrów (argumentów) funkcji.

Tak więc do przekazywania tablic jako parametrów/argumentów funkcji można stosować:

- **notację tablicową** (ang. *array notation*),
- **notację funkcyjną** (ang. *pointer notation*).

Oba wymienione powyżej sposoby są równoważne, ponieważ gdy kompilator natrafia w procesie komplikacji programu na parametry tablicowe funkcji zapisane w notacji tradycyjnej, zamienia je na parametry zapisane w notacji wskaźnikowej.

Tablicę traktowaną jako parametr wejściowy funkcji można do niej przekazać za pomocą parametru/argumentu wskaźnikowego należącego do typu zgodnego z typem składowym elementów tablicy. Należy jednak pamiętać, że jakakolwiek zmiana (modyfikacja) wartości elementów składowych tablicy wewnętrz funkcji będzie wówczas automatycznie obowiązywać również w jej otoczeniu — co często bywa niepożądane i może być niebezpieczne. W pełni bezpiecznym sposobem przekazywania tablicy jako parametru/argumentu funkcji jest wykorzystanie wskaźnika do stałej `const` należącej do typu zgodnego z typem składowym elementów tablicy. Jest to sposób zalecany ze względu na jego efektywność.

Z drugiej strony tablice traktowane jako wyjście funkcji można przekazać do jej otoczenia za pomocą parametru (argumentu) wskaźnikowego należącego do typu zgodnego z typem składowym elementów tablicy.

Przekazywanie tablic jako parametrów/argumentów funkcji zilustrowano za pomocą programów zawartych w kolejnych przykładach. W programie w przykładzie 8.11 zastosowano tradycyjną notację tablicową, a w programie w przykładzie 8.12 — notację wskaźnikową.

Przykład 8.11

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

// Deklaracja stałej globalnej:
const int n = 5;

// Deklaracje — prototypy funkcji:
void daneWejsciowe(double[]);
/* UWAGA
 * W funkcji daneWejsciowe() w przekazywaniu tablic wykorzystano tradycyjny zapis tablicowy: typ_składowy[] —
 * tutaj: double[]. Zapis double[] jest zamieniany przez kompilator podczas komplikacji na zapis równoważny:
 * double*. Oznacza to, że do/z funkcji nie jest przekazywana cała tablica, lecz jedynie wskaźnik do jej pierwszego
 * elementu składowego. Tym samym tak określony parametr może odgrywać rolę wejścia lub wyjścia funkcji,
 * w zależności od kontekstu w jej wywołaniu.
 */
void sumaElementow(const double[], double&);
/* UWAGA
 * Funkcja sumaElementow() ma dwa parametry. Pierwszy z nich jest wskaźnikiem do stałej const typu double,
 * który odpowiada typowi elementów składowych tablicy. Parametr ten odgrywa rolę wejścia funkcji.
```

* Drugi parametr stanowi referencję do zmiennej należącej do typu double.

*/

```

int main() {
    // Deklaracja tablicy rzeczywistej:
    double tablica[n];
    double suma; // zmienna pomocnicza reprezentująca sumę elementów tablicy

    cout << "Dane wejściowe:" << endl;
    // Wywołanie funkcji daneWejsciowe() z argumentem tablica:
    daneWejsciowe(tablica);
    /* UWAGA
     * Argument tablica to faktycznie wskaźnik do jej pierwszego elementu składowego.
     */
    // Wywołanie funkcji sumaElementow() z argumentami tablica i suma:
    sumaElementow(tablica, suma);
    /* UWAGA
     * Argument wskaźnikowy tablica odgrywa rolę wejścia funkcji sumaElementow(), ponieważ jest do niej
     * przekazywany jako wskaźnik do stałej typu double, co wynika z deklaracji (i definicji) tej funkcji.
     * Argument suma stanowi wyjście funkcji sumaElementow().
     */
    cout << "Suma elementów tablicy: " << suma << endl;

    return 0;
}

// Definicje funkcji, w których wykorzystano notację tradycyjną — tablicową:
void daneWejsciowe(double t[]) { // t[] stanowi parametr wyjściowy
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = ";
        cin >> t[i];
    }
}
void sumaElementow(const double t[], double& suma) { // t[] stanowi parametr wejściowy
    // suma — parametr wyjściowy
    suma = 0;
    for (int i = 0; i < n; i++) {
        suma += t[i];
    }
}
/* UWAGA
 * W definicjach funkcji powyżej parametr formalny t oznacza wskaźnik na typ double: double *t, który odpowiada
 * typowi elementów składowych tablicy.
 * Jeżeli w programie głównym zadeklarowano tablicę double tablica[n]; (jak tutaj) oraz w wywołaniu każdej

```

- * z powyższych funkcji jej argumentem jest zmienna (wskaźnik) tablica, to wskaźnik ten jest kopowany
 - * do parametru formalnego t. Zgodnie z zasadami arytmetyki wskaźników zapis tablica[i] jest równoważny zapisowi t[i]
 - * i dalej — zapisowi *(t + i).
- */

W programie obliczana jest suma elementów zapisanych w jednowymiarowej tablicy rzeczywistej o nazwie `tablica`. Dane wejściowe (wartości elementów składowych tablicy) są wprowadzane z klawiatury przy wykorzystaniu funkcji `daneWejsciowe()`. Funkcja ta jest wywoływana z argumentem — wskaźnikiem `tablica`, przekazywanym do otoczenia funkcji `daneWejsciowe()`, tj. do funkcji `main()`. Argument `tablica` odpowiada wskaźnikowi `t` typu `double*`.

Suma elementów tablicy `tablica` jest wyznaczana przy użyciu funkcji `sumaElementow()`, która zwraca do funkcji `main()` obliczoną wartość sumy za pomocą argumentu referencyjnego `suma`. Argumentem wejściowym w wywołaniu funkcji `sumaElementow()` jest wskaźnik do stałej `const` typu `double` — `tablica`, który odpowiada typowi elementów składowych tablicy.

Parametry formalne funkcji `daneWejsciowe()` oraz `sumaElementow()`, jak również operacje realizowane w ciałach tych funkcji są zapisywane z zastosowaniem tradycyjnej notacji tablicowej.

Ćwiczenie 8.11

Zmodyfikuj program z przykładu 8.11 — zamiast sumy elementów składowych zapisanych w tablicy wyznacz wartość najmniejszą (minimum) i największą (maksimum) z tych elementów. W deklaracjach i definicjach wykorzystanych funkcji zastosuj tradycyjną notację tablicową. Uwzględnij to założenie zarówno w odniesieniu do parametrów formalnych funkcji, jak i do operacji przetwarzania elementów składowych tablicy w ich ciałach.

Przykład 8.12

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

// Deklaracja stałej:

```
const int n = 5;
```

// Deklaracje — prototypy funkcji:

```
void daneWejsciowe(double*);  
void sumaElementow(const double*, double&);
```

/* UWAGA

- * W zadeklarowanych powyżej funkcjach w przekazywaniu tablic do/z funkcji wykorzystano zapis wskaźnikowy: `double*`. Oznacza to, że do/z funkcji nie jest przekazywana cała tablica, lecz jedynie wskaźnik do tablicy (adres jej pierwszego elementu). Tak określony parametr może odgrywać rolę wejścia lub wyjścia funkcji, w zależności od kontekstu jej wywołania.
- */

```

int main() {
    // Deklaracja tablicy:
    double tablica[n];
    double suma;

    cout << "Dane wejściowe:" << endl;
    // Wywołanie funkcji daneWejsciowe() z argumentem tablica:
    daneWejsciowe(tablica);
    // Wywołanie funkcji sumaElementow() z argumentami tablica i suma:
    sumaElementow(tablica, suma);

    cout << "Suma elementów tablicy: " << suma << endl;

    return 0;
}

// Definicje funkcji:
void daneWejsciowe(double *t) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = ";
        cin >> *(t + i);
    }
}
void sumaElementow(const double *t, double &suma) {
    suma = 0;
    for (int i = 0; i < n; i++) {
        suma += *t;
        t++;
    }
}

/* UWAGA
 * Jeżeli w programie głównym zadeklarowano tablicę: double tablica[n]; to w chwili wywołania zdefiniowanych
 * powyżej funkcji wskaźnik tablica jest kopowany do parametru formalnego t, będącego wskaźnikiem na zmienną
 * typu double, który stanowi typ elementów składowych tablicy.
 * Dzięki arytmetyce wskaźnika zapisy:
 *   a) *(t + i) w funkcji daneWejsciowe() oraz
 *   b) *t w funkcji sumaElementow(), gdzie t zmienia się w każdej iteracji wraz z inkrementacją i,
 * są równoważne zapisowi t[i] i w rezultacie zapisowi tablica[i].
 */

```

Funkcjonalność przedstawionego programu jest analogiczna do funkcjonalności programu zawartego w przykładzie 8.11. Programy te różnią się szczegółami implementacji. Mianowicie w deklaracjach oraz definicjach funkcji `daneWejsciowe()` i `sumaElementow()` wykorzystano notację wskaźnikową, a w przykładzie 8.11 — tradycyjną notację tablicową.

Tutaj parametry formalne umożliwiające przekazywanie tablicy do funkcji i z funkcji zostały zdefiniowane z zastosowaniem notacji wskaźnikowej. To samo dotyczy operacji wykonywanych w ciałach funkcji.

Ćwiczenie 8.12

Zmodyfikuj program z przykładu 8.12 — zamiast sumy elementów składowych zapisanych w tablicy wyznacz wartość najmniejszą (minimum) i największą (maksimum) z tych elementów. W deklaracjach i definicjach użytych funkcji wykorzystaj notację wskaźnikową. Przy czym dotyczy to zarówno parametrów formalnych funkcji, jak i operacji przetwarzania elementów składowych tablicy w ich ciałach.

8.3.5. Łańcuchy znaków jako parametry funkcji

Przekazywanie C-napisów do funkcji i z funkcji

Sposób przekazywania C-napisów do funkcji i z funkcji do jej otoczenia jest uwarunkowany ich budową, jako jednowymiarowych tablic znaków zakończonych znakiem specjalnym '`\0`' (NUL).

UWAGA

C-napisy zostały omówione szczegółowo w podrozdziale 6.1.

Przy uwzględnieniu zasad konstrukcji C-napisów odpowiadające im parametry/argumenty funkcji powinny być definiowane w analogiczny sposób jak w przypadku tablic. Przy czym dotyczy to zarówno parametrów wejściowych funkcji, jak i wyjściowych.

UWAGA

Tematyka przekazywania tablic do funkcji i z funkcji została omówiona w podrozdziale 8.3.4.

Dlatego też C-napisy należy przekazywać do funkcji jako parametry wskaźnikowe do stałych `const` należących do typu `char`, a z funkcji do jej otoczenia — za pośrednictwem wskaźników typu `char*`. Można oczywiście stosować zarówno tradycyjną notację tablicową, jak i notację wskaźnikową.

Przykład 8.13

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

// Definicje funkcji:
void pobierzNapis(char napis[]) {
    cin >> napis;
}

/* UWAGA
* Zadaniem funkcji pobierzNapis() jest pobranie C-napisu z klawiatury przy wykorzystaniu standardowego strumienia
* wejściowego cin. Funkcja ma jeden parametr wyjściowy o nazwie napis typu wskaźnikowego char*.
*/

void wyswietlNapis(const char napis[]) {
    cout << napis << std::endl;
}

/* UWAGA
* Zadaniem funkcji wyswietlNapis() jest wyświetlenie C-napisu na ekranie monitora przy wykorzystaniu standardowego
* strumienia wyjściowego cout. Funkcja ma jeden parametr wejściowy o nazwie napis, przekazywany do funkcji
* jako wskaźnik do stałej const typu char.
*/

void sumaNapisow(const char napis1[], const char napis2[], char napis3[]) {
    char temp[10] = "";

    strcat(temp, napis1);
    strcat(temp, " ");
    strcat(temp, napis2);

    strcat(napis3, temp);
}

/* UWAGA
* Zadaniem funkcji sumaNapisow() jest dodanie (konkatenacja) C-napisów: napis2, „ ” i napis2.
* Wynik zostaje podstawiony do C-lancucha napis3.
* Funkcja ma dwa parametry wejściowe: napis1 i napis2, przekazywane do funkcji jako wskaźniki do stałych const
* typu char. Parametr wyjściowy funkcji napis3 jest przekazywany do jej otoczenia jako wskaźnik char*.
*/

int main() {
    char nazwaJezyska[10] = "", wersjaJezyska[10] = "";
    char jezyk[10] = "";
```

```
// Pobranie danych wejściowych:  
cout << "Podaj nazwę języka programowania: ";  
pobierzNapis(nazwaJęzyka);  
  
cout << "Podaj wersję: ";  
pobierzNapis(wersjaJęzyka);  
  
// Przetwarzanie danych:  
sumaNapisow(nazwaJęzyka, wersjaJęzyka, język);  
  
// Prezentacja wyniku:  
cout << "Język programowania: ";  
wyswietlNapis(język);  
  
return 0;  
}
```

Zadaniem programu jest pobranie z klawiatury nazwy i wersji języka programowania w postaci C-napisów. Następnie dane te są przetwarzane — wyznaczana jest suma tych C-napisów przedzielona znakiem spacji. Wynik konkatenacji C-napisów jest prezentowany w konsoli na ekranie monitora.

W programie wykorzystano trzy funkcje: `pobierzNapis()`, `wyswietlNapis()` i `sumaNapisow()`. Funkcja `pobierzNapis()` ma jeden parametr wyjściowy, przekazywany za pośrednictwem wskaźnika należącego do typu `char*`. Funkcja `wyswietlNapis()` ma jeden parametr wejściowy, przekazywany do funkcji jako wskaźnik do stałej `const` typu `char`. Funkcja `sumaNapisow()` zaś ma dwa parametry wejściowe i jeden wyjściowy.

Ćwiczenie 8.13

Zmodyfikuj program zawarty w przykładzie 8.13 — zamiast nazwy i wersji języka programowania wprowadź wieloczłonową nazwę środowiska programistycznego, np. Visual Studio Community. Każdy człon nazwy wybranego środowiska należy oczywiście wprowadzić z klawiatury oddzielnie i zapamiętać w programie w zmiennych napisowych.

Przekazywanie łańcuchów typu `string` do funkcji i z funkcji

Łańcuchy znaków, które należą do typu `string` i stanowią wejście/wejścia oraz wyjście/wyjścia funkcji, traktuje się podobnie jak parametry/argumenty należące do typów podstawowych. Zostało to zilustrowane w zawartym w przykładzie 8.14 programie, który ma taką samą funkcjonalność jak program z przykładu 8.13.

Przykład 8.14

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

void pobierzLancuch(string& lancuch) {
    cin >> lancuch;
}

void sumaLancuchow(string lancuch1, string lancuch2, string& lancuch3) {
    lancuch3 = lancuch1 + " " + lancuch2;
}

void wyswietlLancuch(string lancuch) {
    cout << lancuch << endl;
}

int main() {
    string nazwaJazyka = "", wersjaJazyka = "";
    string jazyk = "";
    // Wprowadzenie danych wejściowych w postaciłańcuchów znaków należących do typu string:
    cout << "Podaj nazwę języka programowania: ";
    pobierzLancuch(nazwaJazyka);

    cout << "Podaj wersję: ";
    pobierzLancuch(wersjaJazyka);

    // Przetwarzanie danych typu string:
    sumaLancuchow(nazwaJazyka, wersjaJazyka, jazyk);

    // Wyświetlenie wyniku —łańcucha znaków należącego do typu string:
    cout << "Język programowania: ";
    wyswietlLancuch(jazyk);

    return 0;
}
```

Ćwiczenie 8.14

Zmodyfikuj program zawarty w przykładzie 8.14 — zamiast nazwy i wersji języka programowania wprowadź wieloczlonową nazwę środowiska programistycznego, np. IDEA IntelliJ Community. Każdy człon nazwy wybranego środowiska powinien być wprowadzany z klawiatury oddzielnie i zapamiętywany w programie w zmiennychłańcuchowych należących do typu string.

8.3.6. Parametry domyślne

Parametrom formalnym wyszczególnionym w definicji funkcji można podczas ich deklarowania nadać wartości domyślne. W szczególności dotyczy to parametrów wejściowych przekazywanych do funkcji przez wartość albo przez referencję do stałej `const`.

Parametry domyślne (ang. *default parameters*) są również nazywane **parametrami opcjonalnymi funkcji** (ang. *function optional parameters*), ponieważ nie wymagają podania odpowiadających im argumentów w ich wywołaniach.

Jeżeli w kodzie źródłowym programu następuje wywołanie funkcji, dla której w jej definicji zostały określone parametry domyślne, to mogą zajść dwa przypadki:

- argument odpowiadający parametrowi domyльнemu nie został podany,
- argument odpowiadający parametrowi domyльнemu został określony.

W pierwszym przypadku kompilator już na etapie komplikacji programu zastąpi nieistniejący argument wywołania funkcji jego domylnym odpowiednikiem w postaci wartości parametru domyślnego.

W drugim przypadku argument (parametr) domyślny zostanie zastąpiony (nadpisany) argumentem wyszczególnionym w wywołaniu funkcji w sposób jawnny.

Przykład 8.15

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

// Definicje funkcji z parametrami domyślnymi:
void wyswietl_pi(float pi = 3.14159) {
    cout << "Stała Pi = " << " " << pi << endl;
}
/* UWAGA
 * Parametr formalny pi jest przekazywany do funkcji wyswietl_pi() przez wartość.
 */

void wyswietl_e(const float& e = 2.71828) {
    cout << "Stała e = " << " " << e << endl;
}
/* UWAGA
 * Parametr formalny e jest przekazywany do funkcji wyswietl_pi() przez referencję do stałej const typu float.
 */

int main() {
    // Wywołanie funkcji wyswietl_pi():
    wyswietl_pi();
```

```

/* UWAGA
 * Ze względu na to, że funkcja została wywołana bez żadnego argumentu, argumentami domyślnymi (tutaj:
 * jeden argument)
 * stają się automatycznie wartości jej domyślnych parametrów formalnych (tutaj: jeden parametr domyślny).
 */

const float Pi = 3.14;
// Ponowne wywołanie funkcji wyswietl_pi():
wyswietl_pi(Pi);
/* UWAGA
 * Wywołanie funkcji wyswietl_pi() z argumentem podanym w sposób jawnym. W związku z tym wszystkie
 * argumenty domyślne (tutaj: jeden) zostają nadpisane przez argumenty zadane.
 */

// Wywołanie funkcji wyswietl_e() bez żadnych argumentów:
wyswietl_e();

const float E = 2.72;
// Wywołanie funkcji wyswietl_e() z argumentem określonym w sposób jawnym:
wyswietl_e(E);

return 0;
}

```

W programie zdefiniowano dwie funkcje: `wyswietl_pi()` oraz `wyswietl_e()`. Pierwsza z wymienionych funkcji ma zadanie wyświetlić w konsoli na ekranie wartość stałej `pi` (stałej Archimedesa), a druga — wyświetlić stałą `e` (liczbę Eulera). Dla obu funkcji zostały określone parametry domyślne (opcjonalne).

W programie głównym wywołania funkcji `wyswietl_pi()` i `wyswietl_e()` występują zarówno z argumentami domyślnymi, jak i z argumentami określonymi w ich wywołaniach w sposób jawnym.

Ćwiczenie 8.15

Na podstawie kodu pokazanego w przykładzie 8.15 napisz program pozwalający obliczyć pole i obwód koła. Wykorzystaj zdefiniowane samodzielnie funkcje `pole()` i `obwod()`. Każda z wymienionych funkcji powinna mieć po dwa parametry wejściowe: jeden odpowiadający stałej `pi`, a drugi — długości promienia koła. Parametr formalny reprezentujący stałą `pi` należy zdefiniować jako domyślny.



8.4. Zmienne globalne i lokalne

8.4.1. Zmienne globalne

W ogólności zmienne mogą być deklarowane w programie:

- albo na zewnątrz każdej z funkcji,
- albo w ciele (we wnętrzu) funkcji.

Dotyczy to także programu głównego, którym w języku C++ jest funkcja `main()`. W ogólności zmienne mogą być również deklarowane w blokach kodu — to trzeci z możliwych przypadków umiejscowienia deklaracji zmiennej.

Zmienne, które zostały zadeklarowane na zewnątrz jakiegokolwiek funkcji (włączając w to funkcję `main()`), nazywa się **zmiennymi globalnymi** (ang. *global variables*). Zmienne globalne charakteryzują się **zakresem (zasięgiem) globalnym** (ang. *global scope*), co oznacza, że są one **widoczne** (ang. *visible*) począwszy od miejsca deklaracji aż do końca pliku, w którym zostały zadeklarowane. Zmienne globalne są także dostępne (widoczne) we wszystkich funkcjach zdefiniowanych w tym pliku.

Dlatego niektórzy programiści zasięg zmiennych globalnych nazywają **zasięgiem plikowym** (ang. *file scope*). Z kolei, jeśli wziąć pod uwagę to, że deklaracje zmiennych globalnych są umiejscowione na zewnątrz każdej z funkcji zdefiniowanych w pliku, można je uznać za część **globalnej przestrzeni nazw** (ang. *global namespace*).

Czas życia (ang. *lifetime*) zmiennych globalnych rozpoczyna się z chwilą uruchomienia programu, w którym zostały zadeklarowane, i kończy w chwili zakończenia jego działania. Ze względu na to, że zmiennych globalnych nie można wcześniej „zniszczyć” (ang. *destroy*), charakteryzują się one **statyczną długością życia** (ang. *static life duration*). Dlatego też wielu programistów zalicza zmienne globalne do **zmiennych statycznych** (ang. *static variables*).

Zmienne globalne można deklarować bez nadawania im wartości początkowych (np. `double g_waga;`), jak również ich deklaracja może być połączona z inicjalizacją (np. `double g_waga = 0.5;`). Jeśli zmienna globalna zostanie w sposób jawnny jedynie zadeklarowana, to i tak kompilator nada jej „po cichu” (niejawnie) „zerową” wartość początkową, np. wartość 0 w przypadku zmiennej należącej do któregoś z typów liczbowych.



UWAGA

Zgodnie z zalecanymi konwencjami kodowania w języku C++ deklaracje zmiennych globalnych powinny być umieszczone na samym początku pliku — zaraz po dyrektywach preprocesora `#include`, ale przed rozpoczęciem kodu programu. Nazwy (identyfikatory) zmiennych globalnych poprzedza się zwykle przedrostkiem `g_` lub `g` (np. `g_waga`) dla podkreślenia ich globalnego zasięgu.

Zmienne globalne można deklarować również jako stałe. Wówczas ich deklaracja musi być poprzedzona słowem kluczowym `const`. Stałe globalne `const` muszą zostać zainicjowane w sposób jawny, np. `const double g_waga = 0.5;`.

Przykład 8.16

```
#include <iostream>
#include <cmath>

// Deklaracja i inicjalizacja globalnej stałej const o nazwie g_waga_standard należącej do typu double:
const double g_waga_standard = 0.5;

// Deklaracja zmiennej globalnej:
double g_waga;
/* UWAGA
* Zmienna globalna to taka zmienna, która została zadeklarowana na zewnątrz jakiejkolwiek funkcji.
* Zasięg zmiennej globalnej rozpoczyna się od miejsca deklaracji i obejmuje wszystkie funkcje zdefiniowane w tym pliku,
* aż do jego końca.
*/

// Deklaracja — prototyp funkcji:
int ocenaKoncowa(float, float);

using namespace std;

// Program główny:
int main() {
    // Testowa prezentacja wartości stałej globalnej g_waga_standard:
    cout << "Waga: " << g_waga_standard << endl;

    // Testowa prezentacja wartości zmiennej globalnej g_waga:
    cout << "Waga: " << g_waga << endl;
    // Nadanie (modyfikacja) wartości zmiennej globalnej g_waga:
    g_waga = 0.75;
    cout << "Waga: " << g_waga << endl;

    double ocenaJPolski = 3;
    cout << "Ocena końcowa z języka polskiego = "
        << ocenaKoncowa(g_waga_standard, ocenaJPolski) << endl;

    double ocenaJNiemiecki = 4;
    cout << "Ocena końcowa z języka niemieckiego = "
        << ocenaKoncowa(g_waga, ocenaJNiemiecki) << endl;
}
```

```

        return 0;
}

// Definicja funkcji:
int ocenaKoncowa(float waga, float ocena) {
    return round(waga * ocena);
}

```

W programie zadeklarowano dwie zmienne globalne o identyfikatorach: `g_waga_standard` i `g_waga`. Zmienna `g_waga_standard`, jako stała `const`, została zainicjowana w sposób jawnny. Z kolei zmiennej `g_waga` podczas deklaracji została nadana w sposób niejawnym wartość zerowa (`0`), która później (w ciele funkcji `main()`) została zmodyfikowana.

Zdefiniowane zmienne wykorzystano w funkcji `main()` również jako argumenty funkcji `ocenaKoncowa()`.

Ćwiczenie 8.16

Na podstawie przykładu 8.16 napisz program pozwalający obliczyć pole i obwód koła. Wykorzystaj zmienną globalną o nazwie `g_pi` o wartości równej `3.14159`.

8.4.2. Zmienne lokalne

Zmiennymi lokalnymi (ang. *local variables*) nazywane są zmienne, które deklaruje się:

- w ciele funkcji (ang. *function body*),
- w bloku kodu (ang. *code block*), ograniczonym przez nawiasy klamrowe `{}`.

Ponadto do zmiennych lokalnych zaliczane są zmienne stanowiące parametry formalne funkcji w ich definicjach.

Zmienne lokalne definiowane w ciele funkcji

Każda zmienna zadeklarowana w ciele funkcji jest zmienną lokalną tej funkcji. Zakres widoczności takiej zmiennej jest ograniczony do ciała funkcji, w której zmienna ta została zadeklarowana — począwszy od miejsca deklaracji aż do końca bloku ciała funkcji. Tym samym nie jest możliwe uzyskanie dostępu do zmiennej lokalnej w kodzie źródłowym przed jej deklaracją. Zmienna lokalna, która została zdefiniowana w ciele funkcji, nie jest dostępna w jej otoczeniu, czyli na zewnątrz tej funkcji. Dlatego też zakres widoczności zmiennej lokalnej nazywa się czasem **zakresem blokowym** (ang. *block scope*).

Określenie, że zmienna lokalna o danej nazwie (identyfikatorze) znajduje się w zakresie (ang. *in scope*), oznacza, że dostęp do tej zmiennej jest możliwy. W przeciwnym razie, tj. jeśli zmienna znajduje się poza zakresem (ang. *out of scope*), jej przetwarzanie nie jest możliwe, ponieważ nie jest dostępna.

Czas życia zmiennej lokalnej rozpoczyna się z chwilą jej zdefiniowania w ciele funkcji, a kończy wraz z zakończeniem wykonywania tej funkcji. Wówczas zmienność ta jest niszczona w sposób automatyczny. Stąd czas życia zmiennej lokalnej często jest określany jako automatyczny (ang. *automatic lifetime*).

Przykład 8.17

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

// Definicja funkcji polePr():
double polePr(double b1, double b2) {
    // Definicja (deklaracja i inicjalizacja) zmiennej lokalnej o nazwie temp:
    double temp = b1 * b2; // początek życia i zakresu widoczności zmiennej lokalnej temp
    /* UWAGA
     * Zakres widoczności zmiennej temp obejmuje wyłącznie ciało funkcji polePr().
     * Na zewnątrz (w otoczeniu) funkcji polePr() zmienność ta nie jest widoczna, czyli nie jest dostępna.
     */
    return temp; // koniec życia zmiennej temp, określony przez koniec działania funkcji
} // koniec zakresu widoczności zmiennej lokalnej temp

// Definicja funkcji obwodPr():
double obwodPr(double b1, double b2) {
    // Definicja zmiennej lokalnej o nazwie temp:
    double temp = 2 * b1 + 2 * b2;

    return temp;
}

int main() {
    // Definicja (deklaracja i inicjalizacja) zmiennej lokalnej bok1:
    double bok1 = 1; // początek życia i zakresu widoczności zmiennej bok1

    // Definicja zmiennej lokalnej bok2:
    double bok2 = 2; // początek życia i zakresu widoczności zmiennej bok2
    /* UWAGA
     * Zmienne lokalne bok1 i bok2 są dostępne bezpośrednio w ciele funkcji main() począwszy od miejsca ich
     * zdefiniowania oraz we wszystkich funkcjach, które są w niej wywoływanie.
     */

    cout << "Pole wynosi: " << polePr(bok1, bok2) << endl;
    cout << "Obwód wynosi: " << obwodPr(bok1, bok2) << endl;

    return 0; // koniec życia zmennych lokalnych bok1 i bok2 określony przez koniec działania funkcji main()
} // koniec zakresu widoczności zmennych lokalnych bok1 i bok2
```

W programie obliczane są pole i obwód prostokąta dla długości jego boków zainicjowanych w programie głównym. W tym celu zdefiniowano dwie funkcje globalne: `polePr()` i `obwodPr()`. W ciele każdej z tych funkcji zdefiniowano zmienne lokalne o jednakowej nazwie `temp`. Zmienna `temp` zdefiniowana w funkcji `polePr()` jest zupełnie niezależna od zmiennej o tej samej nazwie zdefiniowanej w funkcji `obwodPr()`. Zmienne te są widoczne wyłącznie wewnętrz (w ciałach) funkcji, w których zostały zdefiniowane, począwszy od miejsca ich zdefiniowania aż do końca bloku ciała każdej z funkcji.

Ponadto w funkcji `main()` zostały zdefiniowane zmienne `bok1` i `bok2`, które reprezentują długości boków prostokąta. Zmienne te są lokalne względem funkcji `main()`. Zmienne `bok1` i `bok2` są przekazywane do funkcji `polePr()` i `obwodPr()` jako ich argumenty wejściowe przekazywane przez wartość. Zakres widoczności zmiennych lokalnych `bok1` i `bok2` obejmuje blok (ciało) funkcji `main()` począwszy od miejsc, w których zostały zdefiniowane, aż do końca tego bloku. Czas życia tych zmiennych jest zaś określony przez ich definicje w kodzie źródłowym i koniec działania programu (funkcji `main()`).

Ćwiczenie 8.17

Zmodyfikuj program z przykładu 8.17 — zamiast pola i obwodu prostokąta oblicz pole i obwód koła. Wykorzystaj funkcje, w których zdefiniowano zmienne lokalne.

Zmienne lokalne definiowane w bloku kodu

Do zmiennych lokalnych zalicza się również zmienne, które zostały zadeklarowane w bloku kodu, czyli w zestawie instrukcji ograniczonym za pomocą nawiasów klamrowych {}.

Właściwości takich zmiennych lokalnych są analogiczne do właściwości zmiennych lokalnych zdefiniowanych w ciałach funkcji. W szczególności dotyczy to ich zakresu widoczności oraz czasu życia.

Zakres widoczności zmiennej lokalnej zdefiniowanej w bloku kodu jest ograniczony przez początek i koniec tego bloku. Czas życia takiej zmiennej obejmuje zaś czas wykonywania instrukcji zawartych w tym bloku kodu.

Przykład 8.18

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

int main() {
    // początek bloku kodu

    // Definicje — deklaracje połączone z inicjalizacją — zmiennych lokalnych bok1 i bok2:
    double bok1 = 1; // początek życia i widoczności zmiennej lokalnej bok1
    double bok2 = 2; // początek życia i widoczności zmiennej lokalnej bok2
```

```

/* UWAGA
 * Zmienne bok1 i bok2 są zmiennymi lokalnymi względem bloku kodu, w którym zostały zdefiniowane.
 * Na zewnątrz tego bloku zmienne te nie są dostępne.
 */

cout << "Pole wynosi " << bok1 * bok2 << endl;
cout << "Obwód wynosi " << 2 * bok1 + 2 * bok2 << endl;

} // koniec życia i widoczności zmiennych lokalnych bok1 i bok2

// cout << "Pole wynosi " << bok1 * bok2 << endl;
// cout << "Obwód wynosi " << 2 * bok1 + 2 * bok2 << endl;
/* UWAGA
 * Instrukcje powyżej (oznaczone jako komentarz) są błędne, ponieważ zmienne lokalne bok1 i bok2 znajdują się
 * poza zakresem, tj. nie zostały zadeklarowane w zakresie bloku ciała funkcji main().
 */

return 0;
}

```

W programie zdefiniowano dwie zmienne: bok1 i bok2, które reprezentują długości boków prostokąta. Na podstawie tych zmiennych obliczane są pole i obwód prostokąta.

Zmienne bok1 i bok2 są zmiennymi lokalnymi względem bloku kodu, w którym zostały zdefiniowane. Zakres widoczności i czas życia rozpatrywanych zmiennych rozpoczynają się z chwilą (w miejscu) ich zdefiniowania. Zmienne te są niszczone na końcu wspomnianego powyżej bloku kodu, co wiąże się z końcem ich życia (i widoczności).

Zmienne bok1 i bok2 nie są dostępne w ciele funkcji `main()` poza blokiem, w którym zostały zdefiniowane. Tym samym linie kodu oznaczone w programie jako komentarz są błędne.

Ćwiczenie 8.18

Zmodyfikuj program z przykładu 8.18 — zamiast pola i obwodu prostokąta oblicz pole i obwód koła. Wykorzystaj zmienne lokalne zdefiniowane w niezależnym bloku kodu zawartym w funkcji `main()`.

Parametry formalne funkcji jako zmienne lokalne

Parametry formalne zawarte na liście parametrów w definicji funkcji również są zmiennymi lokalnymi. Przy czym „lokalność” tych zmiennych odnosi się wyłącznie do funkcji, w której zostały określone. W szczególności dotyczy to parametrów wejściowych przekazywanych do funkcji przez wartość.

Jeśli dany parametr funkcji zostanie w definicji funkcji zadeklarowany jako przekazywany przez wartość, to w chwili wywołania tej funkcji jest do niego kopowana odpowiadająca mu wartość argumentu. Kopia tego argumentu jest przechowywana w pamięci operacyjnej na stosie jako zmienna lokalna funkcji o identyfikatorze odpowiadającym nazwie parametru formalnego funkcji. Wszelkie operacje są wykonywane w ciele funkcji z użyciem tej właśnie zmiennej lokalnej. Na zewnątrz (w otoczeniu) funkcji zmienna ta nie jest dostępna — jest niewidoczna. Po zakończeniu działania funkcji kopia argumentu funkcji (jako jej zmienna lokalna) jest niszczona. Tym samym kończy się czas życia zmiennej lokalnej. To samo dotyczy zakresu widoczności zmiennej lokalnej będącej kopią argumentu przekazywanego do funkcji przez wartość — jest on określony przez ciało funkcji.

Przykład 8.19

```
#include <iostream>
#include <cmath>

// Definicja funkcji:
double przekatnaPr(double b1, double b2) {
    return sqrt(pow(b1, 2) + pow(b2, 2));
}

/* UWAGA
* Parametry formalne funkcji (b1 i b2) są przekazywane do funkcji przez wartości. Odgrywają one rolę wejścia funkcji.
*/

using namespace std;
int main() {
    double bok1 = 1;
    double bok2 = 1;

// Wywołanie funkcji przekatnaPr() z argumentami bok1 i bok2:
cout << "Długość przekątnej wynosi " << przekatnaPr(bok1, bok2) << endl;
/* UWAGA
* W chwili wywołania funkcji przekatnaPr() z argumentami bok1 i bok2 wartości te są kopowane do zmiennych
* lokalnych. Te zmienne to parametry formalne w definicji funkcji, które odpowiadają wymienionym argumentom:
* bok1, bok2. Innymi słowy, parametry formalne funkcji, jako jej zmienne lokalne, zostają zainicjowane
* wartościami argumentów przekazanych z funkcji wywołującej (funkcji main()) do funkcji przekatnaPr()
* przez wartość.
*/

    return 0;
}
```

W programie obliczana jest długość przekątnej prostokąta. Operacja ta jest realizowana przez wywołanie funkcji `przekatnaPr()` z argumentami `bok1` i `bok2` przekazanymi do funkcji przez wartości.

W pamięci operacyjnej komputera w chwili wywołania wspomnianej funkcji tworzone są kopie wartości argumentów, zapamiętywane następnie w zmiennych lokalnych odpowiadających parametrom formalnym funkcji: `b1` i `b2`.

Zakres widoczności zmiennych lokalnych `b1` i `b2` obejmuje ciało funkcji `przekatnaPr()`, a ich czas życia jest równy czasowi działania tej funkcji począwszy od chwili jej wywołania.

Ćwiczenie 8.19

Zmodyfikuj program z przykładu 8.19 — zamiast przekątnej prostokąta oblicz średnicę koła. Wykorzystaj funkcję z parametrem przekazywanym przez wartość. Zinterpretuj operację wywołania tej funkcji z uwzględnieniem jej zmiennych lokalnych.

8.4.3. Przesłanianie zmiennych

Wykorzystanie zmiennych globalnych i lokalnych w programach komputerowych może się wiązać ze zjawiskiem tzw. **przesłonięcia** (ang. *override*) nazwy (identyfikatora) jednej zmiennej przez inną.

Na przykład nazwa zadeklarowanej zmiennej globalnej może zostać przesłonięta w bloku (ciele) funkcji przez nazwę zadeklarowanej tam zmiennej lokalnej. Wówczas odwołanie się w ciele funkcji do nazwy zmiennej odnosi się nie do zmiennej globalnej, lecz do zmiennej lokalnej. Tym samym zakres widoczności zmiennej globalnej został ograniczony, mianowicie została z niego wyłączony blok funkcji, w którym zadeklarowano zmienną lokalną o tej samej nazwie. Z drugiej strony efekt przesłaniania nie ma żadnego wpływu na czas życia przesłoniętych zmiennych.

W sytuacji gdy nazwa zmiennej globalnej została przesłonięta nazwą zmiennej lokalnej, odwołać się do przesłoniętej zmiennej globalnej można w każdej chwili — wystarczy poprzedzić jej nazwę operatorem zakresu (ang. *scope operator*), `::`.

W przypadku ogólnym zjawisko przesłaniania obejmuje nie tylko zmienne globalne. Przesłaniane mogą być również zmienne lokalne jednej funkcji przez zmienne lokalne innej funkcji. Dotyczy to także parametrów funkcji. Inny przypadek jest związany z przesłanianiem zmiennych lokalnych przez inne zmienne lokalne zdefiniowane w bloku kodu.

Przykład 8.20

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

// Definicje funkcji:
double poleKw(double bok) {
    // Definicja zmiennej lokalnej pole:
    double pole = bok * bok;
    return pole;
}
```

```

double obwodKw(double bok) {
    // Definicja zmiennej lokalnej obwod:
    double obwod = 4 * bok;
    return obwod;
}

// Definicja zmiennej globalnej bok:
double bok = 1;

int main() {
    // Wywołanie funkcji poleKw():
    double pole = poleKw(bok);
    /* UWAGA
     * Zmienna globalna o nazwie bok została w bloku (ciele) funkcji poleKw() przesłonięta
     * przez zmienną lokalną bok, która jest parametrem formalnym tej funkcji.
     */

    cout << "Pole wynosi " << pole << endl;

    { // początek bloku kodu
        double bok = 2;
        /* UWAGA
         * Zmienna globalna bok została przesłonięta przez zmienną bok zdefiniowaną
         * w bloku kodu zawartym w funkcji main().
         */

        double obwod = obwodKw(bok);
        cout << "Obwód wynosi " << obwod << endl;
    } // koniec bloku

    cout << "Długość przekątnej wynosi " << sqrt(2 * pow(bok, 2)) << endl;
    /* UWAGA
     * W ciele funkcji main() obowiązuje zmienna globalna bok — chyba że zostanie ona
     * przesłonięta przez inną zmienną (lokalną) o tej samej nazwie.
     */

    return 0;
}

```

W programie obliczane są pole i obwód oraz długość przekątnej kwadratu. Pole i obwód są obliczane przy wykorzystaniu funkcji `poleKw()` i `obwodKw()`. W funkcjach tych określono parametr formalny o nazwie `bok`.

Oprócz tego zdefiniowano tutaj zmienną globalną `bok` oraz zmienną lokalną o tej samej nazwie. Definicja wspomnianej zmiennej lokalnej `bok` jest zawarta w bloku kodu w funkcji `main()`.

Najpierw zmienna globalna `bok` zostaje przesłonięta przez zmienną lokalną o tej samej nazwie, będącą parametrem funkcji `poleKw()`. Następnie ta sama zmienna zostaje przesłonięta przez zmienną lokalną `bok` zdefiniowaną w bloku kodu zawartym w funkcji `main()`.

Ćwiczenie 8.20

Na podstawie programu z przykładu 8.20 omów zjawisko przesłaniania zmiennych związanych z wywołaniem w programie głównym funkcji `obwodKw()`.

8.5. Funkcje przeciążone

Przeciążanie funkcji (ang. *function overloading*) polega na tym, że w tym samym zakresie (np. w zakresie globalnym, tj. na zewnątrz funkcji `main()`) definiuje się funkcje, które mają taką samą nazwę, ale różnią się od siebie liczbą i/lub typem parametrów.

Różne typy i/lub liczba parametrów funkcji przeciążonych zwykle wiążą się z różną implementacją tych funkcji. Ponadto funkcje przeciążone mogą się od siebie różnić funkcjonalnością, i to znacznie.

Jeśli w programie zdefiniowano wiele funkcji przeciążonych, kompilator, gdy natrafia w kodzie na wywołanie którejkolwiek z nich, sprawdza w sposób niejawny („po cichu”) liczbę i typy argumentów w tym wywołaniu. Po zakończeniu tego procesu kompilator wywołuje właściwą funkcję — funkcję, w której wywołaniu liczba i typy argumentów odpowiadają liczbie i typom parametrów formalnych w jej definicji. Proces ten odbywa się w całości na etapie komplikacji programu.

UWAGA

Przeciążanie funkcji jest ściśle powiązane z jedną z cech programowania zorientowanego obiektowo — **polimorfizmem statycznym** (ang. *static polymorphism*), który oznacza wielopostaciowość (tutaj: wielopostaciowość funkcji). Polimorfizm został omówiony w dalszej części podręcznika — w rozdziale 15.

Przykład 8.21

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

// Definicje funkcji przeciążonych:
double srednia(double l1, double l2) {
    return (l1 + l2) / 2;
}

double srednia(int l1, int l2) {
    return (double(l1 + l2) / 2);
}
```

```

double srednia(double l1, double l2, double l3) {
    return (l1 + l2 + l3) / 3;
}

double srednia(int l1, int l2, int l3) {
    return (double(l1 + l2 + l3) / 3);
}

/* UWAGA
 * Funkcje przeciążone zdefiniowane powyżej różnią się od siebie zarówno liczbą, jak i typem parametrów.
 */

int main() {
    // Wywołania funkcji srednia():
    cout << "Średnia z dwóch liczb całkowitych: " << srednia(1, 2) << endl;
    cout << "Średnia z dwóch liczb rzeczywistych: " << srednia(1.0, 2.0)
        << endl;

    cout << "Średnia z trzech liczb całkowitych: " << srednia(1, 2, 3) << endl;
    cout << "Średnia z trzech liczb rzeczywistych: " << srednia(1.0, 2.0, 3.0)
        << endl;
}

/* UWAGA
 * Kompilator na etapie komplikacji programu sprawdza liczbę i typy argumentów przekazanych do/z funkcji
 * w ich wywołaniach. Na podstawie tych informacji podejmuje decyzję, którą ze zdefiniowanych funkcji
 * przeciążonych należy wywołać.
 */

return 0;
}

```

W programie zdefiniowano cztery funkcje o nazwie `srednia`. Każda z nich różni się od reszty liczbą i/lub typem parametrów formalnych. Tym samym stanowią one funkcje przeciążone.

Zadaniem każdej ze zdefiniowanych funkcji `srednia()` jest obliczenie średniej arytmetycznej z dwóch lub trzech liczb należących albo do typu rzeczywistego `double`, albo do typu całkowitego `int`.

W kodzie programu głównego wywołano kilka razy funkcję `srednia()`. Wywołania te różnią się od siebie liczbą i/lub typem argumentów. Na podstawie argumentów wywołania kompilator wybiera odpowiednią funkcję i ją wywołuje. Odbywa się to podczas procesu komplikacji programu.

Ćwiczenie 8.21

Zmodyfikuj program z przykładu 8.21 — zamiast obliczać średnią arytmetyczną z zestawu zadanych liczb (całkowitych i rzeczywistych), niech wyznacza ich średnią geometryczną. Wykorzystaj zestaw zdefiniowanych funkcji przeciążonych.



8.6. Funkcje inline

Funkcje `inline` to w języku C++ funkcje szczególnego rodzaju. Ich charakterystyczna cecha polega na tym, że ich wywołania są „rozwijane”. Określenie „rozwijane” oznacza, że wywołania funkcji `inline` są zastępowane w kodzie źródłowym programu kompletnym, „rozwinietym” kodem ich ciał.

Wspomniane powyżej zastąpienie wywołań funkcji `inline` ich kodami źródłowymi odbywa się na etapie komplikacji programu (ang. *compile time*). To wyraźnie odróżnia omawiany tutaj rodzaj funkcji od zwykłych funkcji C++, ponieważ zastąpienie wywołania zwykłej funkcji jej kodem źródłowym odbywa się w czasie wykonywania programu (ang. *runtime*).

Głównym celem stosowania funkcji `inline` jest zwiększenie wydajności programu, ponieważ ich wywołania „rozwinięte w kodzie” powodują przyspieszenie działania programu. Oprócz tego użycie funkcji `inline` może znaczco polepszyć gospodarowanie pamięcią operacyjną komputera dzięki zminimalizowaniu „skoków” zapotrzebowania na tę pamięć na stosie, występujących w chwilach wywołania zwykłych funkcji — zwłaszcza tych z parametrami/argumentami przekazywanymi przez wartość. Najważniejszą wadą funkcji `inline` jest zwiększenie rozmiaru kodu (pliku wykonywalnego) programu.

UWAGA

Więcej informacji na temat funkcji `inline` można znaleźć w dokumentacji języka C++ na stronie cppreference.com, w artykule *Inline function specifier*.

Przykład 8.22

```
#include <iostream>

// Definicje funkcji inline:
inline double inline_max(double x, double y) {
    return (x > y) ? x : y;
}
inline double inline_min(double x, double y) {
    return (x < y) ? x : y;
}

using namespace std;
int main() {
    double a = 1, b = 2;

    cout << "Dane wejściowe: " << a << ", " << b << endl;

    // Wywołanie funkcji inline_max():
}
```

```

cout << "Większa z zadanych liczb = " << inline_max(a, b) << endl;
/* UWAGA
 * Kompilator już na etapie kompilacji programu zastępuje wywołanie funkcji inline_max() jej treścią —
 * kodem źródłowym zawartym w jej definicji.
 */

return 0;
}

```

W programie wyznaczana jest większa z dwóch zadanych liczb. Operacja ta jest realizowana przy wykorzystaniu funkcji `inline` o nazwie `inline_max`. Podczas kompilacji programu wywołanie tej funkcji jest zastępowane jej treścią — kodem źródłowym zawartym w definicji.

Ćwiczenie 8.22

Zmodyfikuj program z przykładu 8.22 — zamiast większej z dwóch liczb wyznacz największą wartość z trzech zadanych liczb. Operacje zrealizuj za pomocą funkcji `inline`. Zinterpretuj wywołanie zdefiniowanej funkcji `inline` w programie głównym.

8.7. Funkcje rekurencyjne

Z tematyką programowania z użyciem funkcji wiąże się co najmniej kilka ważnych pojęć, np. deklaracja funkcji, definicja funkcji, wywołanie funkcji. Była o tym mowa wcześniej w tym rozdziale.

W języku C++ wywołanie funkcji jest realizowane w innej funkcji, nazywanej **funkcją wywołującą** (ang. *caller*). Bardzo często rolę funkcji wywołującej odgrywa np. funkcja `main()`, czyli program główny. Jednakże niekiedy jest konieczne, aby rolę funkcji wywołującej danej funkcji odgrywała ona sama.

Proces, w którym dana funkcja wywołuje samą siebie, nazywa się **rekurencją** (ang. *recursion*), a funkcję, która wywołuje samą siebie — **funkcję rekurencyjną** (ang. *recursion function*). Przy czym w przypadku ogólnym funkcja może wywoływać samą siebie w sposób bezpośredni albo pośredni.

W pierwszym z wymienionych powyżej przypadków, tj. jeśli funkcję wywołującą dla danej funkcji jest ona sama, mówimy o **rekurencji bezpośredniej** (ang. *direct recursion*). Tym samym rekurencję bezpośrednią najprościej można rozpoznać po tym, że bezpośrednio w ciele danej funkcji wywoływana jest ona sama.

W drugim przypadku funkcję wywołującą dla danej funkcji rekurencyjnej (np. funkcji `r()`) stanowi inna, pośrednia funkcja (np. `p()`), która sama jest wywoływana we wspomnianej funkcji rekurencyjnej `r()`. Ten rodzaj rekurencji jest określany terminem **rekurencja pośrednia** (ang. *indirect recursion*).

W praktyce rekurencja pośrednia jest stosowana o wiele rzadziej od rekurencji bezpośrednią.

Przykład 8.23

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

// Definicja funkcji rekurencyjnej:
int nwd(int pLiczba1, int pLiczba2) {
    int temp;
    if (pLiczba2 == 0){
        temp = pLiczba1;
    } else {
        // Warunkowe wywołanie funkcji nwd w „samej siebie”:
        temp = nwd(pLiczba2, pLiczba1 % pLiczba2);
    }

    return temp;
}

/* UWAGA
* Funkcja nwd() zdefiniowana powyżej realizuje algorytm Euklidesa w celu znalezienia największego
* wspólnego dzielnika (NWD) dwóch liczb całkowitych.
*/

int main() {
    // Pierwsza liczba:
    int liczba1 = 12;
    cout << "Pierwsza liczba = " << liczba1 << endl;

    // Druga liczba:
    int liczba2 = 18;
    cout << "Druga liczba = " << liczba2 << endl;

    // Wywołanie funkcji rekurencyjnej:
    int dzielnik = nwd(liczba1, liczba2);
    cout << "NWD (największy wspólny dzielnik): " << dzielnik << endl;

    return 0;
}
```

W programie przy wykorzystaniu algorytmu Euklidesa wyznaczany jest największy wspólny podzielnik dwóch liczb całkowitych. W tym celu zdefiniowano funkcję rekurencyjną `nwd()`, która wywołuje w swoim ciele samą siebie. Mamy zatem do czynienia z rekurencją bezpośrednią.

Ćwiczenie 8.23

Zmodyfikuj program zawarty w przykładzie 8.23 — zamiast rekurencji bezpośredniej zastosuj rekurencję pośrednią.



8.8. Pytania i zadania kontrolne

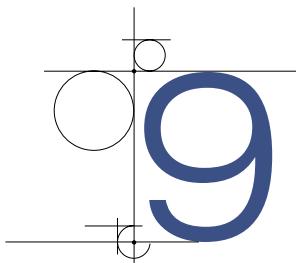
8.8.1. Pytania

- 1.** Czym się różni deklaracja funkcji od definicji funkcji?
- 2.** Jak się ma liczba i typy argumentów wywołania funkcji względem liczby i typów odpowiadających im parametrów formalnych w definicji funkcji?
- 3.** W jaki sposób funkcja komunikuje się ze swoim otoczeniem (funkcją wywołującą)?
- 4.** Co odróżnia parametr/argument funkcji przekazywany przez wartość od parametru/argumentu przekazywanego przez referencję?
- 5.** Czy wskaźniki mogą odgrywać rolę parametrów/argumentów funkcji?
- 6.** Omów zagadnienie przekazywania tablic jako parametrów/argumentów funkcji.
- 7.** W jaki sposób przekazać C-strukturę do funkcji lub z funkcji?
- 8.** Co oznacza określenie: funkcje przeciążone?
- 9.** Wymień najważniejsze cechy zmiennych globalnych i zmiennych lokalnych.
- 10.** Dlaczego parametry/argumenty funkcji przekazywane przez wartość są jej zmiennymi lokalnymi?
- 11.** Wyjaśnij zagadnienie przesłaniania nazw zmiennych.
- 12.** Wymień i omów najważniejsze zasady programowania strukturalnego.

8.8.2. Zadania

- 1.** Napisz program pozwalający obliczyć pole i obwód trójkąta dla zadanych długości jego podstawy oraz wysokości. Wspomniane dane wejściowe mają być wprowadzane z klawiatury. Do obliczeń wykorzystaj funkcje zdefiniowane samodzielnie: jedną do obliczenia pola, a drugą — obwodu. Dane wejściowe powinny być dostarczane do funkcji za pomocą parametrów/argumentów przekazywanych przez wartość. Wynik (pole i obwód trójkąta) należy przekazać do otoczenia funkcji za pośrednictwem ich nazw.
- 2.** Napisz program pozwalający obliczyć objętość, pole powierzchni oraz łączną długość wszystkich krawędzi prostopadłościanu. Dane wejściowe mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Do obliczeń oraz operacji wejścia/wyjścia wykorzystaj funkcje. Dane wejściowe należy dostarczyć do funkcji za pomocą parametrów (argumentów) przekazywanych przez wartość. Obliczone wartości funkcji powinny być przekazywane do otoczenia zarówno za pośrednictwem nazw funkcji, jak i przy użyciu parametrów/argumentów przekazywanych przez referencję.
- 3.** Napisz program pozwalający obliczyć objętość i pole powierzchni bocznej kuli. Długość promienia koła zainicjuj w programie, a wyniki wyświetl na ekranie monitora. Do obliczeń wykorzystaj funkcje. Wykonaj program w wariantach odpowiadających wejściu/wyjściu funkcji zdefiniowanemu w następujący sposób:

- *wariant I*: wejście — parametry/argumenty przekazywane przez wartość, wyjście — nazwa funkcji,
 - *wariant II*: wejście — parametry/argumenty przekazywane przez wartość, wyjście — nazwa funkcji i zarazem parametry/argumenty przekazywane przez referencję,
 - *wariant III*: wejście — parametry/argumenty jako referencje do stałych `const`, wyjście — parametry/argumenty przekazywane przez referencję,
 - *wariant IV*: wejście — parametry/argumenty jako wskaźniki do stałych `const`, wyjście — parametry/argumenty przekazywane za pośrednictwem wskaźników.
4. Napisz program pozwalający zapamiętać w polach C-struktury dane personalne pracownika firmy: imię, nazwisko, płeć i numer legitymacji służbowej. Dane te mają być wprowadzane z klawiatury, a następnie wyświetlane kontrolnie na ekranie monitora. Wykorzystaj funkcje. Pierwsza ze zdefiniowanych funkcji powinna umożliwiać wprowadzenie danych pracownika z klawiatury, a druga — wyświetlenie tych danych na ekranie monitora.
5. Napisz program pozwalający obliczyć sumę i średnią arytmetyczną oraz wyznaczyć wartość największą (maksimum) i najmniejszą (minimum) z elementów jednowymiarowej tablicy rzeczywistej o długości 10. Elementy składowe tablicy należy zainicjować w programie. Do obliczeń i przetwarzania danych wykorzystaj funkcje.
6. Napisz program pozwalający posortować elementy jednowymiarowej tablicy liczbowej (rzeczywistej) w porządku rosnącym za pomocą metody bąbelkowej. Operacje wejścia/wyjścia oraz przetwarzanie danych należy zrealizować za pomocą funkcji zdefiniowanych samodzielnie.



Preprocesor

Przed rozpoczęciem komplikacji kod źródłowy programu jest przetwarzany w wieloetapowym (wielofazowym) procesie nazywanym **translacią** (ang. *translation*). Jedna z najważniejszych faz translacji jest związana z wykorzystaniem tzw. **preprocesora** (ang. *preprocessor*). W uproszczeniu preprocesor można potraktować jako odrębny program, który przetwarza plik źródłowy programu przed rozpoczęciem jego komplikacji. To przetwarzanie polega na wyszukiwaniu w kodzie źródłowym specjalnych instrukcji (poleceń), które rozpoczynają się od znaku # (ang. *hash*), a następnie ich wykonywaniu. Instrukcje te są nazywane **dyrektywami preprocesora** (ang. *preprocessor directives*). Dyrektywy to polecenia wykonania przez preprocesor określonych działań (operacji), np. dołączenia do programu zasobów innego pliku zawierającego kod źródłowy języka C++.

Dyrektyny preprocesora nie są elementami języka C++ (ani C). Ich składnia jest całkowicie odmienna od składni języka C++ (oraz C). Ponadto preprocesor nie sprawdza poprawności składni elementów programu ani zgodności typów danych. Końcowe efekty „pracy” preprocesora są przekazywane do przetworzenia w dalszych fazach procesu translacji, a następnie są kompilowane.

Typów dyrektyw preprocesora jest wiele. Najczęściej używane to:

- dyrektywy pozwalające na dołączenie do programu plików nagłówkowych i/lub plików zdefiniowanych przez użytkownika,
- makrodefinicje,
- dyrektywy komplikacji warunkowej (ang. *conditional compilation*).

9.1. Dyrektywa #include

Dyrektyna preprocesora `#include` jest używana w celu dołączenia do programu głównego lub do innego pliku źródłowego (ang. *source file*) kodu zawartego w odrębnym (wskazanym) pliku. Skutkiem wykonania tej dyrektywy jest zastąpienie jej treści w kodzie źródłowym programu (lub pliku źródłowego) zawartością dołączanego pliku.

Dyrektywą `#include` wykorzystuje się w dwóch przypadkach. W pierwszym do programu dołącza się (importuje) zasoby określonego pliku nagłówkowego (ang. *header file*) albo innego pliku ze standardowej biblioteki C++. Ogólna postać takiej dyrektywy jest następująca:

```
#include <nazwa_pliku_nagłówkowego>
```

gdzie:

- `<>` — nawiasy kątowe (ang. *angle brackets*),
- `nazwa_pliku_nagłówkowego` — nazwa pliku nagłówkowego wchodzącego w skład standardowej biblioteki C++.

Zbiory nagłówkowe, których nazwy zostały podane w nawiasach `<>`, są poszukiwane w standardowym katalogu systemowym środowiska programistycznego (komplikatora) przeznaczonym dla zasobów (bibliotek) języka, np. katalogu o nazwie *include*. Przykładowo dyrektywa pozwalająca na dołączenie (zimportowanie) do programu zasobów standardowej biblioteki *iostream* ma postać: `#include <iostream>`.

Przykład 9.1

```
// Dołączenie do programu plików nagłówkowych z biblioteki standardowej:  
#include <iostream>  
#include <cmath>  
  
using namespace std;  
int main() {  
    // Deklaracja i inicjalizacja zmiennej liczbaInt:  
    int liczbaInt = 4;  
  
    // Wykorzystanie strumienia wyjściowego cout zdefiniowanego w bibliotece iostream oraz predefiniowanej  
    // funkcji sqrt() z biblioteki cmath:  
    cout << sqrt(liczbaInt) << endl;  
  
    return 0;  
}
```

W programie zademonstrowano wykorzystanie dyrektywy `#include` w celu dołączenia (zimportowania) zasobów bibliotek standardowych *iostream* oraz *cmath*. Pierwszą z wymienionych bibliotek trzeba dołączyć ze względu na użycie w programie strumienia wyjściowego `cout`, a drugą — ze względu na zastosowanie predefiniowanej funkcji `sqrt()`, która została tam zdefiniowana.

Ćwiczenie 9.1

Zmodyfikuj program z przykładu 9.1 — wykorzystaj w programie wybrane zasoby (funkcje) zawarte w bibliotekach *string* i *cstring*.

Drugi przypadek użycia dyrektywy `#include` jest związany z potrzebą dołączenia do programu głównego (lub do innego pliku źródłowego) pliku zdefiniowanego przez użytkownika (ang. *user-defined file*). Ogólna postać takiej dyrektywy jest następująca:

```
#include "nazwa_pliku"
```

gdzie:

- "" — podwójne cudzysłowy (ang. *double quotes*),
- nazwa_pliku — nazwa pliku nagłówkowego lub innego pliku zawierającego kod źródłowy zdefiniowany przez użytkownika.

Plik `nazwa_pliku` jest poszukiwany w tym samym katalogu, w którym jest zapisany program główny, tj. funkcja `main()`. Zazwyczaj jest to katalog bieżący. Jeśli jednak wskazany plik nie zostanie tam znaleziony, to w następnym kroku jest on szukany w katalogu systemowym przeznaczonym dla plików nagłówkowych.

Na przykład dołączenie do programu pliku nagłówkowego o nazwie `functions.h`, który został zdefiniowany wcześniej przez użytkownika, ma postać: `#include "functions.h"`.

Przykład 9.2

Zawartość programu głównego zapisanego w pliku `main.cpp`:

```
// Dołączenie do aplikacji systemowych plików nagłówkowych:
#include <iostream>
#include <string>
/* UWAGA
 * W miejscu wywołania każdej z dyrektyw #include kompilator dołącza zawartość podanych plików nagłówkowych.
 * Kompilator będzie poszukiwał tych zasobów na liście standardowych katalogów systemowych.
 */
// Dołączenie do aplikacji „własnego” (zdefiniowanego) pliku nagłówkowego:
#include "operacje.h"
/* UWAGA
 * Kompilator będzie poszukiwał wskazanego pliku w katalogu bieżącym, tj. tym, w którym jest zapisany plik main.cpp.
 */
using namespace std;
int main() {
    double bok = 1;
    cout << "Pole kwadratu wynosi: " << poleKw(bok) << endl;
    cout << "Obwód kwadratu wynosi: " << obwodKw(bok) << endl;

    return 0;
}
// Koniec funkcji main()
```

Zawartość pliku *operacje.h*:

```
// PLIK NAGŁÓWKOWY
// Deklaracje (prototypy) funkcji zdefiniowanych przez użytkownika

// Deklaracje funkcji umożliwiających obliczenie pola i obwodu kwadratu:
double poleKw(double);
double obwodKw(double);

// Deklaracje funkcji umożliwiających obliczenie pola i obwodu prostokąta:
double PolePr(double, double);
double ObwodPr(double, double);
// koniec pliku operacje.h
```

Zawartość pliku *operacje.cpp*:

```
// PLIK ŹRÓDŁOWY
// Długość zawartości pliku nagłówkowego operacje.h:
#include "operacje.h"

// Definicje funkcji umożliwiających obliczenie pola i obwodu kwadratu:
double poleKw(double bok) {
    return bok * bok;
}
double obwodKw(double bok) {
    return 4 * bok;
}

// Definicje funkcji umożliwiających obliczenie pola i obwodu prostokąta:
double PolePr(double bok1, double bok2) {
    return bok1 * bok2;
}
double ObwodPr(double bok1, double bok2) {
    return 2 * bok1 + 2 * bok2;
}
// koniec pliku operacje.cpp
```

Program składa się z trzech plików zdefiniowanych przez użytkownika — programistę: *main.cpp*, *operacje.h* oraz *operacje.cpp*. Plik *main.cpp* to program główny, do którego — za pomocą dyrektywy `#include "operacje.h"` — dołączono plik nagłówkowy *operacje.h*. Plik ten zawiera deklaracje (prototypy) funkcji umożliwiających obliczenie pól i obwodów figur geometrycznych: kwadratu i prostokąta. Definicje tych funkcji zaś są zawarte w pliku źródłowym o nazwie *operacje.cpp*. Do pliku *operacje.cpp* trzeba dołączyć plik nagłówkowy *operacje.h*.

Schemat pokazany w przykładzie można uogólnić na inne programy wykorzystujące zarówno biblioteki standardowe, jak i pliki nagłówkowe i źródłowe zdefiniowane przez użytkownika.

Ćwiczenie 9.2

Z wykorzystaniem zastosowanego w przykładzie 9.2 schematu użycia w programie plików nagłówkowych i źródłowych zdefiniowanych przez użytkownika napisz program pozwalający przeliczyć jednostki długości w systemie SI (np. centymetr, metr, kilometr) na jednostki w systemie anglosaskim (np. cal, stopę, jard, milę) i odwrotnie.

9.2. Dyrektywa #define

Dyrektyna preprocessora `#define` jest używana do definiowania określonych elementów (bloków) programu — „zamienników” (ang. *replacement*), do których zalicza się:

- stałe,
- typy danych,
- makrofunkcje.

Ogólna postać definicji każdego z wymienionych powyżej elementów programu jest następująca:

`#define NAZWA zamiennik`

gdzie:

- NAZWA — identyfikator zamiennika, który zgodnie z konwencją składa się z dużych liter oraz ewentualnie kreski podkreślenia,
- zamiennik — kod określający (opisujący) stałą, typ danych lub makrofunkcję.

Kiedy preprocessor w czasie trwania procesu translacji natrafi na dyrektywę `#define`, której postać ogólną przedstawiono powyżej, zastępuje wszelkie jej wystąpienia w pozostałojej części kodu programu zdefiniowanym zamiennikiem. Przy tym należy zwrócić uwagę, że preprocesser nie sprawdza poprawności składniowej określenia zamiennika, w tym zgodności typów danych.

UWAGA

Prezentowaną tutaj definicję zamiennika (np. stałej, makrofunkcji) wielu programistów określa wspólnym terminem **makrodefinicja** (ang. *macro-definition*) lub krótko **makro** (ang. *macro*).

W celu anulowania definicji zamiennika o określonej NAZWIĘ należy wykonać dyrektywę `#undef` o postaci:

`#undef NAZWA`

gdzie NAZWA oznacza identyfikator zamiennika.

9.2.1. Definiowanie stałych

Jeśli uwzględnimy przedstawioną wcześniej składnię użycia dyrektywy preprocesora `#define`, ogólna postać definicji stałej jest następująca:

```
#define NAZWA wyrażenie
```

gdzie `NAZWA` oznacza identyfikator stałej, a `wyrażenie` jest wyrażeniem dającym w wyniku pożądaną wartość.

Na przykład stałą o nazwie `PI` można zdefiniować za pomocą dyrektywy `#define PI 3.14159`. W wyniku zastosowania takiej definicji każde wystąpienie identyfikatora `PI` w kodzie programu jest zastępowane przez preprocesor określona wartością `3.14159`.

UWAGA

Makrodefinicje pozwalające definiować stałe są zaliczane do grupy **makr obiektowych** (ang. *object-like macros*), ponieważ przypominają swoim wyglądem obiekt zawierający dane.

Przykład 9.3

```
#include <iostream>

// Makrodefinicje stałych:
#define TRUE 1
#define FALSE 0

using namespace std;
int main() {
    int liczba {100};
    // Wykorzystanie stałych TRUE i FALSE:
    int wynik = (liczba > 0) ? TRUE : FALSE;
    cout << wynik << endl; // 1

    return 0;
}
```

W programie zdefiniowano dwie stałe: `TRUE` oraz `FALSE`. Wspomniane stałe wykorzystano w instrukcji zawierającej operator warunkowy. Przed rozpoczęciem właściwej komplikacji programu preprocesor spowoduje zastąpienie w jego kodzie źródłowym stałej `TRUE` wartością `1`, a `FALSE` — wartością `0`.

Ćwiczenie 9.3

Zmodyfikuj program z przykładu 9.3 — w definicji stałych `TRUE` i `FALSE` zamiast wartości (literałów) `1` i `0` użyj literałów `true` i `false`.

9.2.2. Definiowanie typów danych

Ogólną postać makrodefinicji typu danych można zapisać tak:

```
#define NAZWA opis_typu
```

gdzie NAZWA oznacza identyfikator stałej, a opis_typu to określenie nowego typu danych.

Na przykład typ danych reprezentujący typ całkowity bez znaku (`unsigned int`) można zdefiniować za pomocą dyrektywy: `#define UNSIGNED_INT unsigned int`. Jeśli preprocesor natrafi w programie na taką definicję, to każde wystąpienie identyfikatora `UNSIGNED_INT` zastąpi w kodzie zamiennikiem `unsigned int`.

Przykład 9.4

```
#include <iostream>

// Definicje stałych:
#define TRUE 1
#define FALSE 0

// Definicja typu danych o nazwie UNSIGNED_INT:
#define UNSIGNED_INT unsigned int
// Definicja typu danych o nazwie BOOLEAN:
#define BOOLEAN UNSIGNED_INT

using namespace std;
int main() {
    int liczba {100};

    // Wykorzystanie typu danych BOOLEAN oraz stałych TRUE i FALSE:
    BOOLEAN wynik = (liczba > 0) ? TRUE : FALSE;
    cout << wynik << endl; // 1

    return 0;
}
```

W programie za pomocą dyrektywy preprocesora `#define` zdefiniowano dwa typy danych: `UNSIGNED_INT` oraz `BOOLEAN`. Pierwszy z wymienionych typów jest aliasem nazwy typu podstawowego `unsigned int`, a drugi — `UNSIGNED_INT`. Ponadto zdefiniowano dwie stałe: `TRUE` oraz `FALSE`, które reprezentują literały całkowite, odpowiednio, `1` i `0`.

Przed rozpoczęciem komplikacji programu preprocesor zastąpi wszystkie wystąpienia wymienionych powyżej nazw (identyfikatorów) typów danych i stałych (tj. `UNSIGNED_INT`, `BOOLEAN`, `TRUE` i `FALSE`) ich zamiennikami w odpowiadających im makrodefinicjach.

Ćwiczenie 9.4

Na podstawie przykładu 9.4 napisz program pozwalający sprawdzić, czy wartość zmiennej wprowadzonej z klawiatury należy do przedziału otwartego (0, 100). Wykorzystaj typ danych odpowiadający typowi podstawowemu `bool`, zdefiniowany za pomocą dyrektywy `#include`.

9.2.3. Definiowanie makrofunkcji

Za pomocą dyrektywy `#include` można definiować również makrofunkcje. Makrofunkcje są zaliczane do grupy **makr funkcyjnych** (ang. *function-like macros*).

UWAGA

Wielu programistów makrofunkcje nazywa **makrami funkcyjnymi** (ang. *function-like macro*), ponieważ ich użycie w kodzie programu przypomina wywołanie zwykłej funkcji.

W dużym uproszczeniu makrofunkcje można porównać do tradycyjnych funkcji C++ typu `inline`. Podobnie jak zwykłe funkcje `inline`, makrofunkcje mają określona funkcjonalność. Podobieństwo pomiędzy nimi występuje także w ich wywołaniu. Mianowicie wywołania zarówno funkcji `inline`, jak i makrofunkcji są „rozwijane”, co oznacza, że w kodzie źródłowym programu są one zastępowane kompletnym, „rozwiniętym” kodem, który odpowiada za ich funkcjonalność. Biorąc pod uwagę wyłącznie to kryterium, różnica pomiędzy funkcją `inline` a makrofunkcją polega na tym, że „rozwinięcie” funkcji `inline` odbywa się na etapie komplikacji programu, a za rozwinięcie makrofunkcji odpowiedzialny jest preprocesor — przed rozpoczęciem procesu komplikacji.

Inna ważna różnica pomiędzy funkcją `inline` a makrofunkcją wynika z tego, że w przypadku funkcji `inline` sprawdzana jest zarówno zgodność pomiędzy liczbą jej parametrów — w definicji — a liczbą argumentów — w wywołaniu, jak i zgodność typów pomiędzy odpowiadającymi sobie parametrami i argumentami. W przypadku makrofunkcji taka kontrola zgodności nie występuje, co może prowadzić do niespodziewanych błędów w programie.

Makrofunkcje (makra funkcyjne) można podzielić na dwie zasadnicze grupy:

- zwracające na zewnątrz wartość „za pośrednictwem swojej nazwy”;
- zwracające na zewnątrz wartości za pośrednictwem parametrów/argumentów.

Makrofunkcje należące do pierwszej grupy odpowiadają zwykłym funkcjom (`inline`), które w treści (ciele) zawierają słowo kluczowe `return`.

Makrofunkcje drugiego rodzaju nawiązują do tzw. procedur (ang. *procedures*), znanych z takich języków programowania jak Pascal czy Delphi. Dlatego też ten rodzaj makrofunkcji można nazwać **makroprocedurami** (ang. *macroprocedures*). Makroprocedury komunikują się ze swoim otoczeniem za pośrednictwem parametrów/argumentów. Przy tym możliwe jest wykorzystanie zarówno parametrów wejściowych, jak i wyjściowych.

Postać ogólna definicji makrofunkcji jest następująca:

```
#define NAZWA (lista_parametrów) opis_makrofunkcji
```

gdzie:

- NAZWA jest identyfikatorem makrofunkcji,
- lista_parametrów to lista parametrów wejściowych i/lub wyjściowych,
- opis_makrofunkcji to kod opisujący funkcjonalność makrofunkcji z wykorzystaniem parametrów należących do lista_parametrów.

W ogólnym przypadku opis_makrofunkcji może się składać z pojedynczej instrukcji albo zestawu instrukcji oddzielonych od siebie przecinkami. Ponadto opis ten może obejmować jedną linię kodu, jak również kilka linii. Jeżeli opis_makrofunkcji rozciąga się na kilka linii, należy go umieścić w bloku kodu { ... }, przy czym każda linia wchodząca w skład tego bloku (oprócz ostatniej) powinna być zakończona znakiem \ (ang. *backslash*)

Przykład 9.5

```
#include <iostream>

// Definicje makrofunkcji:
#define IS_LETTER(c) (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')
#define IS_DIGIT(c) (c >= '0' && c <= '9')
#define IS_UPPER(c) c >= 'A' && c <= 'Z'
#define IS_LOWER(c) c >= 'a' && c <= 'z'

// Definicja makroprocedury READ_VARIABLE_1:
#define READ_VARIABLE_1(m, v) std::cout << m; std::cin >> v

// Definicja wieloliniowej makroprocedury READ_VARIABLE_2:
#define READ_VARIABLE_2(m, v) { \
    std::cout << m; \
    std::cin >> v; \
}

using namespace std;
int main() {
    int liczba;
    // Wywołanie procedury READ_VARIABLE_1:
    READ_VARIABLE_1("Podaj wartość liczby całkowitej: ", liczba);
    cout << liczba << endl;

    char znak[1];
    // Wywołanie procedury READ_VARIABLE_2:
    READ_VARIABLE_2("Podaj znak: ", znak[0]);
```

```

if (IS LETTER(znak[0])) { // wywołanie makrofunkcji IS LETTER
    cout << "Wprowadzono literę " << znak[0] << endl;
    if (IS LOWER(znak[0])) // wywołanie makrofunkcji IS LOWER
        cout << "Litera jest mała" << endl;
    else cout << "Litera jest duża" << endl;
}
else if (IS DIGIT(znak[0])) // wywołanie makrofunkcji IS DIGIT
    cout << "Wprowadzono cyfrę " << znak[0] << endl;
else
    cout << "Naciśnięto klawisz różny od litery i cyfry: " << znak[0] << endl;

return 0;
}

```

W programie zdefiniowano makrofunkcje: `IS LETTER`, `IS DIGIT`, `IS LOWER`, `IS UPPER`, które mają po jednym parametrze wejściowym nieokreślonego typu. Wymienione makrofunkcje wykorzystano do sprawdzenia, czy wprowadzony z klawiatury znak jest literą, cyfrą lub innym znakiem. Jeśli znak jest literą, następuje sprawdzenie, czy jest to mała litera, czy duża. Wymienione makrofunkcje zwracają na zewnątrz wartości logiczne za pośrednictwem swoich nazw (identyfikatorów).

Oprócz tego zostały zdefiniowane dwie makropredykcje: `READ_VARIABLE_1` oraz `READ_VARIABLE_2`, których funkcjonalność jest identyczna. Obie umożliwiają wyświetlenie komunikatu informacyjnego na ekranie oraz wprowadzenie z klawiatury wartości określonej zmiennej. Wspomniane makropredykcje mają po dwa parametry. Parametr `m` pełni funkcję parametru wejściowego, a `v` — parametru wyjściowego. Oba wspomniane parametry nie mają określonego typu. Definicja makropredykcji `READ_VARIABLE_2` jest zawarta w bloku kodu rozciągającym się na kilka linii, z których każda jest zakończona znakiem \.

Ćwiczenie 9.5

Na podstawie przykładu 9.5 napisz program pozwalający obliczyć pole i obwód koła. Funkcjonalność programu — możliwość wprowadzenia z klawiatury danych wejściowych (długości boków), obliczenia i prezentację wyników na ekranie — zrealizuj przy użyciu zdefiniowanych makrofunkcji.

UWAGA

W praktyce używanie makrofunkcji nie jest zalecane. Zamiast makrofunkcji należy stosować zwykłe funkcje, w tym — w razie potrzeby — funkcje `inline`.

9.3. Dyrektywy kompilacji warunkowej

Dyrektyny kompilacji warunkowej (ang. *conditional compilation directives*) pozwalają na włączenie lub odrzucenie fragmentu kodu źródłowego programu w zależności od tego, czy zadany warunek jest spełniony, czy nie. Oznacza to, że „włączony” fragment kodu programu jest poddawany kompilacji, natomiast „odrzucony” — nie.

UWAGA

Dyrektyny kompilacji warunkowej są również nazywane **dyrektywami włączeń warunkowych** (ang. *conditional inclusions directives*).

Do grupy dyrektyw kompilacji warunkowej zalicza się dyrektywy:

- `#ifdef`,
- `#ifndef`,
- `#endif`,
- `#if`,
- `#else`,
- `#elif`.

Ogólna postać użycia dyrektywy `#ifdef` jest następująca:

```
#ifdef NAZWA
    fragment_kodu_programu
#endif
```

gdzie **NAZWA** oznacza identyfikator zamiennika, a **fragment_kodu_programu** stanowi część składową (segment) programu.

Użycie dyrektywy `#ifdef` powoduje sprawdzenie przez preprocesor, czy zamiennik (np. typ danych) został zdefiniowany (ang. *if defined*), czy nie. Jeśli tak, **fragment_kodu_programu** zakończony dyrektywą `#endif` zostaje włączony do programu i poddany kompilacji. W przeciwnym razie wspomniany segment programu zostaje odrzucony i jest traktowany przez kompilator jako komentarz.

Ogólna postać użycia dyrektywy `#ifndef` jest identyczna z opisaną wcześniej ogólną postacią dyrektywy `#ifdef`. Różnica pomiędzy tymi dyrektywami tkwi w ich działaniu. Mianowicie działanie dyrektywy `#ifndef` jest całkowicie odwrotne do działania `#ifdef`. Oznacza to, że **fragment_kodu_programu** zawarty pomiędzy dyrektywami `#ifndef` a `#endif` jest kompilowany wyłącznie wtedy, gdy zamiennik (stała, typ danych, makrofunkcja) nie został zdefiniowany (ang. *if not defined*).

Przykład 9.6

```
#include <iostream>

// Definicja stałej PI:
#define PI 3.14159
// Definicja makroprocedury READ_VARIABLE:
#define READ_VARIABLE(m, v) std::cout << m; std::cin >> v

using namespace std;
int main() {
    float promien;

#ifndef READ_VARIABLE // jeśli zdefiniowano zmiennik (makroprocedurę) o nazwie READ_VARIABLE
    // Wywołanie makroprocedury READ_VARIABLE:
    READ_VARIABLE("Podaj promień koła: ", promien);
#endif

#ifndef READ_VARIABLE // jeśli nie zdefiniowano makroprocedury READ_VARIABLE
    cout << "Podaj promień koła: ";
    cin >> promien;
#endif

    cout << promien << endl;
    float pole, obwod;

#ifndef PI // jeśli zdefiniowano zmiennik (stałą) o nazwie PI
    pole = PI * promien * promien; // wykorzystanie stałej PI
    obwod = 2 * PI * promien; // wykorzystanie stałej PI
#endif

#ifndef PI // jeśli nie zdefiniowano stałej PI
    pole = 3.14 * promien * promien;
    obwod = 2 * 3.14 * promien;
#endif

    cout << pole << endl;
    cout << obwod << endl;

    return 0;
}
```

Program ma za zadanie obliczenie pola i obwodu koła dla promienia wprowadzonego z klawiatury. Wyniki (pole i obwód) są — po obliczeniu — wyświetlane w konsoli na ekranie monitora.

W programie zdefiniowano dwa zamienniki: stałą o nazwie PI i makroprocedurę READ_VARIABLE. W programie głównym występują zarówno bloki `#ifdef — #endif`, jak i `#ifndef — #endif`.

Jeśli makroprocedura READ_VARIABLE została zdefiniowana (`#ifdef READ_VARIABLE`), promień koła jest wprowadzany za jej pomocą. W przeciwnym razie (`#ifndef READ_VARIABLE`) promień jest wprowadzany z zastosowaniem standardowego strumienia wejścia `cin`.

Analogiczna sytuacja występuje przy obliczaniu pola i obwodu koła. Jeśli stała PI została zdefiniowana (`#ifdef PI`), obliczenia są wykonywane z jej użyciem. W przeciwnym razie (`#ifndef PI`) obliczenia są realizowane z wykorzystaniem literała `3.14`, a nie stałej PI.

Ćwiczenie 9.6

Zmodyfikuj program zawarty w przykładzie 9.6 — zamiast pola i obwodu koła oblicz objętość i pole powierzchni bocznej kuli.

UWAGA

Więcej informacji dotyczących makrodefinicji można znaleźć w dokumentacji języka C++ na stronie cppreference.com, w artykule *Replacing text macros*.

9.4. Pytania i zadania kontrolne

9.4.1. Pytania

1. Co to jest dyrektywa preprocesora? Czy dyrektywy preprocesora są wykonywane przed rozpoczęciem komplikacji programu, czy w trakcie komplikacji?
2. Jaka jest różnica pomiędzy dyrektywą `#include <plik>` a dyrektywą `#include "plik"`?
3. Czym jest makrofunkcja, a czym makroprocedura?
4. Wymień, a następnie omów podobieństwa i różnice pomiędzy makrofunkcjami a „zwyczajnymi” funkcjami w języku C++.
5. Dlaczego użycie makrofunkcji nie jest zalecane?
6. Na czym polega warunkowa komplikacja fragmentu kodu programu? W jaki sposób można „wymusić” komplikację warunkową?

9.4.2. Zadania

1. Napisz program pozwalający obliczyć zadaną potęgę z liczby o wartości wprowadzonej z klawiatury. Wykorzystaj predefiniowaną funkcję `pow()`, zdefiniowaną w bibliotece `cmath`.
2. Napisz program pozwalający przeliczać wybrane jednostki wagi w systemie SI (np. gram, kilogram, tonę) na jednostki w systemie anglosaskim (np. funt, uncję, tonę angielską).

Deklaracje (prototypy) funkcji, których zadaniem jest przeliczenie wagi podanej w danej jednostce na inną, należy umieścić w pliku nagłówkowym [jednostki.h](#), a ich definicje — w pliku źródłowym [jednostki.cpp](#).

3. Napisz program umożliwiający obliczenie długości promienia koła przy założeniu, że jego pole powierzchni podane w hektarach jest znane. Wykorzystaj stałą o nazwie PI o wartości równej 3.14159, zdefiniowaną za pomocą makrodefinicji.
4. Napisz program umożliwiający wyznaczenie za pomocą zdefiniowanych makrofunkcji większej i mniejszej z dwóch liczb o wartościach wprowadzonych z klawiatury.
5. Napisz program pozwalający wyświetlić na ekranie monitora dowolny jednoliniowy komunikat. Przy tym po wyświetleniu komunikatu kurSOR powinien znajdować się na początku następnej linii. Wykorzystaj zdefiniowaną makroprocedurę o nazwie WRITE_MESSAGE.
6. Napisz program pozwalający obliczyć pole i obwód prostokąta z użyciem zdefiniowanych makrofunkcji POLE i OBWOD. Wymuś w programie warunkową komplikację fragmentu kodu, w którym zastosowano wymienione makrofunkcje. Jeśli makrofunkcje POLE i OBWOD nie zostały zdefiniowane, obliczenia należy zrealizować z wykorzystaniem „zwykłych” funkcji. Dane wejściowe mają być wprowadzane z klawiatury.

10

Funkcje biblioteczne

Język C++, podobnie jak inne języki programowania, jest wyposażony w szereg przydatnych w praktyce **funkcji predefiniowanych** (ang. *predefined functions*). Funkcje te są zgrupowane w **standardowej bibliotece C++** (ang. *C++ Standard Library*). Dlatego też nazywa się je **funkcjami standardowymi** (ang. *standard functions*).

Biblioteka standardowa C++ składa się z dwóch części:

- **biblioteki funkcji standardowych** (ang. *Standard Function Library*),
- **zorientowanej obiektywnej biblioteki klas** (ang. *Object Oriented Class Library*).

Tym samym funkcje standardowe można podzielić na dwie grupy:

- funkcje odziedziczone po języku C, zgrupowane w bibliotece funkcji standardowych,
- funkcje członkowskie klas wchodzących w skład biblioteki klas.

UWAGA

Klasy jako elementy języka C++ zostały omówione szczegółowo w rozdziale 11. podręcznika.

Funkcje odziedziczone po języku C są funkcjami autonomicznymi — nie są zależne ani jedna od drugiej, ani też od żadnej klasy. Funkcje te są zgrupowane w różnych „bibliotekach”, które z punktu widzenia programisty są **zbiorami nagłówkowymi** (ang. *header files*). Tym samym, aby można było skorzystać z określonej funkcji, należy dołączyć do programu — za pomocą dyrektywy `#include` — bibliotekę, w której ta funkcja została zadeklarowana.

Natomiast funkcje standardowe należące do biblioteki klas to funkcje członkowskie określonej klasy, która została tam zdefiniowana. Dostęp do nich uzyskuje się w analogiczny sposób jak do funkcji odziedziczonych po języku C — przez dołączenie za pomocą dyrektywy `#include` zbioru nagłówkowego zawierającego deklaracje niezbędnych zasobów.

Tematyka wykorzystania funkcji standardowych była już poruszana w podręczniku. W szczególności dotyczy to funkcji napisowych i łańcuchowych, pozwalających wykonywać różne operacje na, odpowiednio, C-napisach oraz łańcuchach znaków typu `string`. Funkcje na-

pisowe należą do funkcji standardowych odziedziczonych po języku C i zostały opisane w podrozdziale 6.1.4. Biblioteka funkcji napisowych nazywa się *cstring*. Funkcje łańcuchowe operujące na łańcuchach znaków typu *string* zaś są funkcjami członkowskimi klasy *string* i zostały szczegółowo omówione w podrozdziale 6.2.4.



10.1. Funkcje matematyczne

Funkcje matematyczne należą w C++ do najczęściej wykorzystywanych funkcji standardowych. Są zgrupowane w bibliotece *cmath*. W języku C odpowiednikiem biblioteki *cmath* jest zbiór nagłówkowy o nazwie *math.h*.

Opis wybranych bibliotecznych funkcji matematycznych jest zawarty w tabeli 10.1.

Tabela 10.1. Opis wybranych funkcji matematycznych

Nazwa	Opis
<code>abs(argument)</code>	Zwraca wartość bezwzględną z argumentu
<code>sqrt(argument)</code>	Zwraca pierwiastek kwadratowy argumentu
<code>pow(base, exponent)</code>	Zwraca wartość równą podstawie podniesionej do potęgi wykładnik
<code>log(argument)</code>	Zwraca wartość logarytmu naturalnego z argumentu
<code>exp(argument)</code>	Zwraca wartość funkcji wykładniczej: e podniesione do potęgi argument
<code>log10(argument)</code>	Zwraca wartość logarytmu dziesiętnego z argumentu
<code>sin(argument)</code>	Zwraca wartość funkcji <i>sinus</i> z argumentu mierzonego w radianach
<code>cos(argument)</code>	Zwraca wartość funkcji <i>cosinus</i> z argumentu mierzonego w radianach
<code>tan(argument)</code>	Zwraca wartość funkcji <i>tangens</i> z argumentu mierzonego w radianach
<code>ceil(argument)</code>	Zwraca najmniejszą wartość całkowitą nie mniejszą od wartości argumentu — zaokrąglenie w górę
<code>floor(argument)</code>	Zwraca największą wartość całkowitą, która jest nie większa od wartości argumentu — zaokrąglenie w dół
<code>trunc(argument)</code>	Zwraca najbliższą wartość całkowitą, która jest nie większa od wartości argumentu — obcięcie części ułamkowej
<code>round(argument)</code>	Zwraca wartość całkowitą, której wartość jest najbliższa wartości argumentu — zaokrąglenie



UWAGA

Więcej informacji dotyczących funkcji bibliotecznych zawartych w zbiorze nagłówkowym *cmath* można znaleźć w dokumentacji języka C++ na stronie cppreference.com, w artykule *Standard library header <cmath>*.

Przykład 10.1

W przedstawionym programie wykorzystano stałą o nazwie `M_PI`, która reprezentuje wartość liczby *pi*. Ponadto zademonstrowano użycie wybranych funkcji standardowych: `abs()`, `sqrt()`, `pow()`, `round()`, `ceil()`, `floor()` i `sin()`, a także niektórych funkcji z biblioteki *cmath*: `fmin()`, `fmax()` i `modf()`.

```
#include <iostream>
#define _USE_MATH_DEFINES // w celu użycia stałej M_PI, która nie jest zdefiniowana w standardowym C/C++
// Dобавление библиотеки cmath в программу:
#include <cmath>
using namespace std;

int main() {
    cout << "Wartość stałej PI: " << M_PI << endl;

    cout << "Wartość bezwzględna liczby -1 = " << abs(-1) << endl;
    cout << "Kwadrat liczby 3 = " << pow(3, 2) << endl;
    cout << "Pierwiastek kwadratowy liczby 4 = " << sqrt(4) << endl;

    cout << "Mniejsza z dwóch liczb: 1 i 10 = " << fmin(1, 10) << endl;
    cout << "Większa z dwóch liczb: 1 i 10 = " << fmax(1, 10) << endl;

    cout << "Zaokrąglenie do liczby całkowitej liczby 1.55 = " << round(1.55)
        << endl;

    double czesc_calkowita;
    double czesc_ulamkowa = modf(1.55, &czesc_calkowita);
    cout << "Część całkowita liczby 1.55 = " << czesc_calkowita << endl;
    cout << "Część ułamkowa liczby 1.55 = " << czesc_ulamkowa << endl;
    cout << "Zaokrąglenie w górę liczby 1.55 = " << ceil(1.55) << endl;
    cout << "Zaokrąglenie w dół liczby 1.55 = " << floor(1.55) << endl;

    cout << "Sinus liczby PI/2 = " << sin(M_PI/2) << endl;

    return 0;
}
```

Ćwiczenie 10.1.

Z wykorzystaniem kodu programu z przykładu 10.1 omów składnię i działanie funkcji standardowych: `fmin()`, `fmax()` i `modf()`.

10.2. Funkcje znakowe

Standardowe funkcje znakowe pozwalają wykonywać różne operacje na znakach, tj. danych należących do typu `char`. Na przykład można sprawdzić, czy dany znak należy do grupy znaków alfanumerycznych, czy jest literą, czy cyfrą itd.

Nagłówki predefiniowanych funkcji znakowych są zawarte w pliku nagłówkowym (bibliotece) `cctype`. W języku C biblioteka ta nazywa się `cctype.h`.

Wykaz wybranych funkcji znakowych wraz z krótkim opisem przedstawiono w tabeli 10.2.

Tabela 10.2. Opis wybranych funkcji znakowych z biblioteki `cctype`

Nazwa	Opis
<code>isalnum(argument)</code>	Sprawdza, czy argument należy do grupy znaków alfanumerycznych. Zwraca wartość <code>true</code> , jeśli znak jest literą (małą lub dużą) albo cyfrą. W przeciwnym razie zwraca <code>false</code>
<code>isalpha(argument)</code>	Sprawdza, czy argument jest literą alfabetu, z uwzględnieniem ustawień narodowych. Jeśli tak, zwraca wartość <code>true</code> , jeśli nie — <code>false</code>
<code>islower(argument)</code>	Sprawdza, czy argument jest małą literą. Jeśli tak, zwraca wartość <code>true</code> , jeśli nie — <code>false</code>
<code>isupper(argument)</code>	Sprawdza, czy argument jest dużą literą. Jeśli tak, zwraca wartość <code>true</code> , jeśli nie — <code>false</code>
<code>isdigit(argument)</code>	Sprawdza, czy argument jest cyfrą dziesiętną. Jeśli tak, zwraca wartość <code>true</code> , jeśli nie — <code>false</code>
<code>isxdigit(argument)</code>	Sprawdza, czy argument jest cyfrą szesnastkową. Jeśli tak, zwraca wartość <code>true</code> , jeśli nie — <code>false</code>
<code>isprint(argument)</code>	Sprawdza, czy argument jest znakiem drukowanym (ang. <i>printable character</i>). Jeśli tak, zwraca wartość <code>true</code> , jeśli nie — <code>false</code>
<code>ispunct(argument)</code>	Sprawdza, czy argument jest znakiem interpunkcyjnym (ang. <i>punctuation character</i>). Jeśli tak, zwraca wartość <code>true</code> , jeśli nie — <code>false</code>
<code>iscntrl(argument)</code>	Sprawdza, czy argument jest znakiem sterującym (ang. <i>control character</i>). Jeśli tak, zwraca wartość <code>true</code> , jeśli nie — <code>false</code> . Działanie odwrotne do funkcji <code>isprint()</code>
<code>tolower(argument)</code>	Konwertuje argument (dużą literę) na małą literę
<code>toupper(argument)</code>	Konwertuje argument (małą literę) na dużą literę



UWAGA

Więcej informacji dotyczących funkcji zawartych w bibliotece `cctype` można znaleźć w dokumentacji języka C++ na stronie cppreference.com, w artykule *Standard library header <cctype>*.

Przykład 10.2

```
#include <iostream>
// Długość programu zasobów biblioteki cctype:
#include <cctype>

using namespace std;
int main() {
    char znak = 'c';

    // Sprawdzenie, czy znak należy do grupy znaków alfanumerycznych:
    cout << isalnum(znak) << endl;

    // Sprawdzenie, czy znak jest literą:
    cout << isalpha(znak) << endl;

    // Sprawdzenie, czy znak jest małą literą:
    cout << islower(znak) << endl;

    // Sprawdzenie, czy znak jest dużą literą:
    cout << isupper(znak) << endl;

    // Sprawdzenie, czy znak jest cyfrą dziesiętną:
    cout << isdigit(znak) << endl;

    // Sprawdzenie, czy znak należy do grupy znaków drukowalnych:
    cout << isprint(znak) << endl;

    // Sprawdzenie, czy znak należy do grupy znaków interpunkcyjnych:
    cout << ispunct(znak) << endl;

    // Sprawdzenie, czy znak należy do grupy znaków sterujących:
    cout << iscntrl(znak) << endl;

    return 0;
}
```

Ćwiczenie 10.2

Z wykorzystaniem opisu funkcji `isprint()` i `iscntrl()` w dokumentacji języka C++ na stronie cppreference.com omów grupę znaków drukowalnych i grupę znaków sterujących.



10.3. Konwersja typu danych

W programie często trzeba wykonywać różnego rodzaju operacje pomocnicze, takie jak konwersja łańcucha znaków na liczbę, sortowanie tablicy czy wygenerowanie liczby losowej.

Funkcje ogólnego przeznaczenia umożliwiające realizację tych i innych operacji są zgrupowane w bibliotece *cstdlib* (w języku C — *stdlib.h*).

Wykaz wybranych funkcji zawartych w bibliotece *cstdlib* przedstawiono w tabeli 10.3.

Tabela 10.3. Opis wybranych funkcji standardowych biblioteki *cstdlib*

Nazwa	Opis
<code>atof(argument)</code>	Konwersja C-napisu argument typu <code>const char*</code> na liczbę zmiennoprzecinkową należącą do typu <code>double</code> . Funkcje zwracają wartość liczby typu <code>double</code>
<code>strtod(argument, NULL)</code>	
<code>strtof(argument, NULL)</code>	Konwersja C-napisu argument typu <code>const char*</code> na liczbę zmiennoprzecinkową należącą do typu <code>float</code> . Funkcja zwraca wartość liczby typu <code>float</code>
<code>atoi(argument)</code>	Konwersja C-napisu argument typu <code>const char*</code> na liczbę należącą do typu <code>int</code> . Funkcja zwraca wartość liczby typu <code>int</code>
<code>atol(argument)</code>	Konwersja C-napisu argument typu <code>const char*</code> na liczbę należącą do typu <code>long int</code> . Funkcje zwracają wartość liczby typu <code>long int</code>
<code>strtol(argument, NULL, 0)</code>	

UWAGA

Więcej informacji dotyczących funkcji zawartych w bibliotece *cstdlib* można znaleźć w dokumentacji języka C++ na stronie cppreference.com, w artykule *Standard library header <cstdlib>*.

Przykład 10.3

```
#include <iostream>

// Dołączenie do programu zasobów biblioteki cstdlib:
#include <cstdlib>

using namespace std;
int main() {
    const char *napisI = "3";
    const char *napisF = "3.14159";

    // Konwersja C-napisu napisI na liczbę całkowitą typu long int:
    cout << atol(napisI) << endl;
    cout << strtol(napisI, NULL, 0) << endl;

    // Konwersja C-napisu napisI na liczbę rzeczywistą typu double:
    cout << atof(napisI) << endl;
}
```

```

cout << atof(napisF) << endl;
cout << strtod(napisF, NULL) << endl;

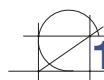
// Konwersja C-napisu napisI na liczbę rzeczywistą typu float:
cout << strtof(napisF, NULL) << endl;

return 0;
}

```

Ćwiczenie 10.3

Napisz program pozwalający skonwertować C-napis "2.71828182845904523536" na liczbę rzeczywistą odpowiadającą wartości stałej Eulera. Uzyskana w wyniku konwersji liczba powinna być zaokrąglona do czterech miejsc po przecinku. Wynik zaprezentuj na ekranie monitora.



10.4. Funkcje wejścia/wyjścia

W języku C++ operacje wejścia/wyjścia można realizować przy użyciu standardowych strumieni wejścia `cin` i wyjścia `cout`. (Była o tym mowa w podrozdziale 3.1.3). Jednakże nie jest to jedyny sposób. Można bowiem wykorzystać predefiniowane funkcje `scanf()` i `printf()`, które język C++ odziedziczył po C. Funkcje te są dostępne po dołączeniu do programu zbioru nagłówkowego o nazwie `cstdio` (w języku C: `stdio.h`) za pomocą dyrektywy preprocessora `#include <cstdio>`.

Funkcja `scanf()` jest stosowana do wprowadzania danych wejściowych z klawiatury, a `printf()` — do wyświetlania danych i informacji na ekranie monitora. Funkcje te należą do grupy funkcji standardowych pozwalających formatować wejście/wyjście, np. ustalić szerokość pola wyjściowego przeznaczonego do wyświetlenia danych.



UWAGA

Szczegółowe informacje na temat składni funkcji `printf()` i `scanf()` można znaleźć w dokumentacji języka C++ na stronie cppreference.com. To samo dotyczy innych funkcji standardowych zawartych w bibliotece `cstdio`, umożliwiających wykonywanie operacji wejścia/wyjścia (np. `fgets()`, `fputs()`), operacji na plikach (np. `fopen()`, `fread()`, `fwrite()`, `fseek()`, `fclose()`) i innych.

Przykład 10.4

```

#include <iostream>
// Dołączenie zasobów biblioteki cstdio do programu:
#include <cstdio>

using namespace std;

```

```
int main() {
    float bok1, bok2, pole, obwod;

    // Wyświetlenie komunikatu na ekranie monitora:
    printf("Podaj długości boków prostokąta: \n");
    // Wyświetlenie komunikatu:
    printf("Pierwszy bok = ");

    // Wprowadzenie wartości zmiennej bok1 z klawiatury:
    scanf("%f",&bok1);

    printf("Drugi bok = ");
    scanf("%f",&bok2);

    pole = bok1 * bok2;

    // Wyświetlenie komunikatu pomocniczego i wartości zmiennej pole na ekranie monitora:
    printf("Pole wynosi: %.2f\n",pole);

    obwod = 2 * bok1 + 2 * bok2;
    // Wyświetlenie komunikatu i wartości zmiennej obwod na ekranie monitora:
    printf("Obwód wynosi: %.2f\n",obwod);

    return 0;
}
```

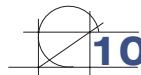
W programie obliczane są pole i obwód prostokąta dla długości boków wprowadzonych z klawiatury. Wyniki są wyświetlane w konsoli na ekranie monitora. Dane wejściowe są wprowadzane za pomocą predefiniowanej funkcji `scanf()`, a wyniki — prezentowane na ekranie przy użyciu funkcji `printf()`. Prototypy obu wymienionych funkcji są zawarte w zbiorze nagłówkowym o nazwie `cstdio`, dołączonym do programu głównego za pomocą dyrektywy preprocessora `#include <cstdio>`.

Funkcja `scanf()` została użyta wraz ze specyfikatorem `%f`, który informuje kompilator, że wprowadzana wartość należy do typu `float`. W innym przypadku, np. gdyby dane wejściowe należały do typu `int`, trzeba byłoby użyć specyfikatora `%d`, dane typu `char` wymagałyby specyfikatora `%c`, a C-napis — specyfikatora `%s`. Te same reguły dotyczą wykorzystania funkcji `printf()`.

Użycie funkcji `printf()` umożliwia również formatowanie wyjścia. Tutaj formatowanie wyjścia zastosowano w celu wyświetlenia obliczonych wartości pola i obwodu. Mianowicie wykorzystano specyfikator `6.2f`, który informuje kompilator, że łączna szerokość pola wyjściowego w konsoli wynosi 6 pól, a liczba miejsc po przecinku w prezentowanych wynikach to 2.

Ćwiczenie 10.4

Zmodyfikuj program z przykładu 10.4 — niech oblicza pole i obwód prostokąta dla danych wejściowych należących do typu `int`. Dane wejściowe mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora.



10.5. Przykłady innych funkcji bibliotecznych

W programowaniu czasem jest konieczne wygenerowanie liczby pseudolosowej z zadanego przedziału. W tym celu można wykorzystać funkcje `srand()` i `rand()`, które wchodzą w skład biblioteki standardowej `cstdlib` (w języku C — `stdlib.h`).

Opis tych funkcji przedstawiono w tabeli 10.4.

Tabela 10.4. Opis funkcji `srand()` i `rand()` z biblioteki `cstdlib`

<code>srand(time(NULL));</code>	Inicjalizacja generatora liczb pseudolosowych. Użycie funkcji <code>time()</code> wymaga dołączenia do programu biblioteki standardowej <code>ctime</code>
<code>rand()</code>	Zwraca liczbę całkowitą pseudolosową z przedziału od zera do wartości makra <code>RAND_MAX</code> , która jest zależna od implementacji. Wartość <code>RAND_MAX</code> wynosi zazwyczaj 32767

Przykład 10.5

```
#include <iostream>
#include <ctime>

// Dołączenie do programu zasobów biblioteki cstdlib:
#include <cstdlib>

using namespace std;
int main() {
    // Inicjalizacja generatora liczb losowych:
    srand(time(NULL));
    /* UWAGA
     * Wykorzystanie funkcji time() wymaga dołączenia do programu biblioteki ctime.
     */

    // Prezentacja wartości makra RAND_MAX:
    cout << RAND_MAX << endl;

    // Wygenerowanie liczby pseudolosowej całkowitej z przedziału od 1 do 100:
    cout << (rand() % 100 + 1) << endl;

    return 0;
}
```

W programie zaprezentowano sposób generowania liczby pseudolosowej zawierającej się w zadanym przedziale. Tutaj wybrano przedział domknięty od 1 do 100.

W celu wygenerowania liczby pseudolosowej należy najpierw „uruchomić” generator liczb pseudolosowych: `srand(time(NULL));`, a następnie wywołać funkcję standardową `rand()` z uwzględnieniem wartości brzegowych zadanego przedziału (tutaj: `rand() % 100 + 1`).

Ćwiczenie 10.5

Napisz program pozwalający wygenerować liczbę pseudolosową z przedziału od 100 do 1000.



10.6. Pytania i zadania kontrolne

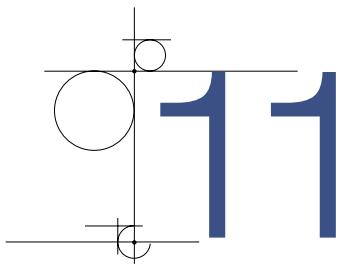
10.6.1. Pytania

1. Wyjaśnij znaczenie terminów: funkcja standardowa, funkcja predefiniowana, funkcja wbudowana.
2. Co to jest zbiór nagłówkowy? Dlaczego zbiory nagłówkowe często nazywa się bibliotekami?
3. W jaki sposób dołącza się do programu zbiory nagłówkowe należące do biblioteki standardowej C++?
4. Wymień i krótko opisz kilka (np. pięć) funkcji standardowych należących do biblioteki `cmath`.
5. W jaki sposób można skonwertować C-napis na liczbę należącą do typu `double`? A na dane typu `int`?
6. Omów składnię i zastosowanie predefiniowanych funkcji `printf()` i `scanf()`.

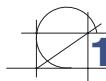
10.6.2. Zadania

1. Napisz program pozwalający podnieść zadaną liczbę całkowitą do potęgi, która również należy do zbioru liczb całkowitych. Dane wejściowe mają być wprowadzane z klawiatury, a wynik wyświetlany na ekranie monitora. Operacje wejścia/wyjścia zrealizuj za pomocą funkcji standardowych `printf()` i `scanf()`.
2. Napisz program umożliwiający obliczenie funkcji *sinus*, *cosinus* oraz *tangens* kąta o wartości wyrażonej w stopniach, a nie w radianach. Dane wejściowe mają być wprowadzane z klawiatury, a wynik wyświetlany na ekranie monitora. Wykorzystaj funkcje `printf()` i `scanf()`.
3. Napisz program pozwalający zapamiętać w C-strukturze dane pracownika: imię, nazwisko i płeć. Płeć pracownika jest reprezentowana przez znak należący do typu `char`: kobieta — 'k' lub 'K', mężczyzna — 'm' lub 'M'. Operacje wejścia/wyjścia należy zrealizować przy użyciu funkcji `printf()` i `scanf()`.

4. Napisz program umożliwiający zaokrąglenie liczby rzeczywistej z przedziału od 1 do 10 do najbliższej liczby całkowitej. Zadana liczba rzeczywista powinna być wylosowana przez generator liczb pseudolosowych.
5. Napisz program pozwalający obliczyć pole trójkąta na podstawie wzoru Herona. Niech dane wejściowe — długości boków trójkąta — będą wprowadzane z klawiatury za pomocą predefiniowanej funkcji `scanf()` jako wartości będące C-napisami.
6. Napisz program pozwalający zapamiętać w tablicy nazwy czterech przedmiotów szkolnych, np. język polski, matematyka. Dane wejściowe mają być wprowadzane z klawiatury. Niech zawartość tablicy będzie kontrolnie wyświetlna na ekranie monitora. Operacje wejścia/wyjścia zrealizuj za pomocą funkcji `printf()` i `scanf()`.



Klasy i obiekty



11.1. Wprowadzenie do programowania obiektowego

11.1.1. Programowanie strukturalne

Pojęcie **programowania strukturalnego** (ang. *structured programming*) jest ściśle powiązane tematycznie z innymi ważnymi pojęciami, takimi jak **programowanie imperatywne** (ang. *imperative programming*) i **programowanie proceduralne** (ang. *procedural programming*). Wymienione pojęcia to jedne z najważniejszych **paradygmatów programowania** (ang. *programming paradigms*).

Paradygmat programowania można określić jako podejście do rozwiązymania określonego problemu — metodę rozwiązyania problemu za pomocą języka programowania, włączając w to zarówno dostępne narzędzia, jak i techniki programowania. Innymi słowy, paradygmat programowania określa styl programowania, sposób myślenia — analizy dotyczącej konstrukcji (budowy) oprogramowania. Nie odnosi się on do konkretnego języka programowania, ale do sposobu — metodyki programowania.

Programowanie imperatywne

Programowanie imperatywne należy do najstarszych paradygmatów programowania. Polega ono na rozwiązyaniu postawionego problemu za pomocą ściśle określonej sekwencji działań — przez wykonanie ich kolejno krok po kroku (ang. *step-by-step*). Wspomniane kroki są reprezentowane w programie przez ciąg następujących po sobie instrukcji, które operują na danych.

Najważniejszymi instrukcjami w programie imperatywnym są instrukcje przypisania. Instrukcje te mogą być oczywiście zawarte w innych instrukcjach (złożonych), np. pętlach programowych, lub stanowić instrukcje składowe innych konstrukcji sterujących przebiegiem działania programu, np. instrukcji warunkowych (wyboru). Dane zaś są reprezentowane w programie

imperatywnym przez zmienne. Konkretne wartości przechowywane w zmiennych określają w danej chwili tzw. **stan programu** (ang. *program state*) w pamięci operacyjnej komputera. Kolejne instrukcje przypisania operujące na zmiennych — wykonywane sekwencyjnie — skutkują zmianą tego stanu, aż do osiągnięcia założonego celu: rezultatu lub rezultatów.

W programach imperatywnych mogą być z powodzeniem stosowane również podprogramy, które odpowiadają za rozwiązywanie podproblemów składowych zadanego problemu (w całości). To samo dotyczy bloków kodu, jako bloków funkcjonalnych konstrukcji sterujących przebiegiem działania programu.

Charakterystyczną cechą programów imperatywnych jest stosowanie instrukcji skoku, np. instrukcji `goto`. Instrukcje skoku to jedne z najważniejszych konstrukcji sterujących przebiegiem wykonywania programu imperatywnego. To duża wada programów imperatywnych, ponieważ niszczy naturalną sekwencję (kolejność) wykonywania instrukcji. Ponadto stosowanie instrukcji skoku powoduje częste problemy (błędy) logiczne, których znalezienie i usunięcie może być bardzo kłopotliwe.

Zmienne określające stan programu są bezpośrednio powiązane z instrukcjami, które na nich operują. Dlatego też modyfikacja programu imperatywnego może być dla programisty niemałym wyzwaniem.

Program napisany w stylu imperatywnym skupia się na *jak* — na tym, *w jaki sposób* rozwiązać postawiony problem, ze szczególnym uwzględnieniem procesu implementacji, czyli kodowania. To przeciwieństwo paradygmatu **programowania deklaratywnego** (ang. *declarative programming*), w którym sposób rozwiązania problemu nie jest scisłe określony. Program deklaratywny skupia się na tym, *co* należy rozwiązać, a nie *w jaki sposób*.

Podsumowując, kod źródłowy napisany w imperatywnym języku programowania stanowi sekwencję instrukcji wykonywanych krok po kroku, które określają: *kiedy* i *w jaki sposób* należy wykonać zadanie, aby osiągnąć cel (rezultat). Stan programu jest definiowany przez stan zmiennych reprezentujących dane. Stan ten można zmienić za pomocą instrukcji przypisania. Podstawową instrukcją sterującą w programie imperatywnym jest instrukcja skoku `goto`. Dlatego taki program łatwo rozpoznać.

Do imperatywnych języków programowania — obok np. Fortranu, C — należą takie nowoczesne języki jak C#, Java oraz oczywiście C++.

Programowanie proceduralne

Termin *programowanie proceduralne* wywodzi się ze stosowania — w celu rozwiązywania podproblemu składowego danego problemu — podprogramów nazywanych **procedurami** (ang. *procedures*). Z procedur można było korzystać w takich językach programowania jak Pascal i Delphi. Procedura w Pascalu i Delphi odpowiada w języku C++ funkcji, która nie zwraca wartości (czyli funkcji typu `void`). Procedury komunikują się ze swoim otoczeniem za pomocą parametrów/argumentów reprezentujących zarówno ich wejście, jak i wyjście. W wymienionych powyżej językach programowania można również definiować inny rodzaj

podprogramów — **funkcje** (ang. *functions*). Funkcje znane z Pascalą i Delphi odpowiadają w języku C++ funkcjom, które zwracają wartości. Tym samym funkcje w Pascalu i Delphi mogą się komunikować ze swoim otoczeniem także za pośrednictwem parametrów (argumentów) wejściowych i wyjściowych. Jednakże zalecane jest, aby funkcje nie modyfikowały wartości argumentów.

UWAGA

Terminu *programowanie proceduralne* nie należy mylić z pojęciem **programowania funkcyjnego** (ang. *functional programming*). Programowanie funkcyjne stanowi bowiem odrębny paradymat programowania, związany z deklaratywnym modelem programowania, a nie imperatywnym.

Funkcja odpowiada w programie za rozwiązywanie określonego zadania cząstkowego. W programie w języku C++ można stosować zarówno **funkcje predefiniowane** (ang. *predefined functions*), jak i **funkcje zdefiniowane samodzielnie przez programistę — użytkownika** (ang. *user-defined functions*). Funkcje predefiniowane zwykle są funkcjami wbudowanymi, zgrupowanymi w bibliotekach. Wybraną bibliotekę można w razie potrzeby dołączyć (zimportować) do programu za pomocą odpowiedniego polecenia. Kod źródłowy funkcji zdefiniowanych samodzielnie przez programistę może być zawarty w tym samym pliku co program główny, jak również w bibliotekach zewnętrznych.

Programowanie proceduralne jest podziobrem programowania imperatywnego. Dlatego też program proceduralny składa się z zestawu kroków (instrukcji) wykonywanych sekwencyjnie jeden po drugim aż do osiągnięcia założonego rezultatu. Kluczowymi instrukcjami w programie proceduralnym są te, które zawierają wywołania funkcji albo same w sobie stanowią wywołania funkcji. W języku C++ ten ostatni przypadek dotyczy oczywiście wywołań funkcji `void`. Nie mniej istotne jest to, że jedna funkcja, wewnętrzna, może być wywoływana w innej — zewnętrznej. Ta druga jest powszechnie nazywana funkcją wywołującą. Ponadto funkcje mogą być parametrami/argumentami wywołania innych funkcji. To samo dotyczy wartości zwracanych przez funkcje na zewnątrz — do ich otoczenia.

Program proceduralny skupia się na funkcjach — ich indywidualnej funkcjonalności, implementacji oraz wzajemnej wymianie informacji pomiędzy nimi za pomocą odpowiednio zdefiniowanych interfejsów. Danym w postaci zmiennych programowych przyznaje się w programie proceduralnym niższy priorytet.

Charakterystyczną cechą programów proceduralnych jest występowanie w nich zarówno zmiennych globalnych, jak i zmiennych lokalnych, co ma ściśły związek ze sposobem gospodarowania pamięcią operacyjną komputera. Inną charakterystyczną cechą programów proceduralnych są skoki w pamięci operacyjnej do definicji funkcji wynikające z ich wywołań w określonych miejscach kodu źródłowego.

Programy proceduralne najczęściej są pisane z zastosowaniem podejścia „od ogółu do szczegółu”, nazywanego również **podejściem „z góry na dół”** (ang. *top-down approach*). Metoda ta polega na sformułowaniu w pierwszej kolejności ogólnego rozwiązania postawionego problemu. Następnie komponowane (definiowane) są mniej ogólne problemy składowe i dalej „w dół” — coraz bardziej szczegółowe podproblemy. Proces ten, nazywany **dekompozycją** (ang. *decomposition*), kończy się wtedy, gdy problem składowy jest na tyle prosty funkcjonalnie, że można go rozwiązać za pomocą podprogramu (funkcji) o określonej implementacji i określonym interfejsie.

W podejściu „z góry na dół” program główny — funkcja `main()` — stanowi rozwiązanie ogólnego problemu w całości. Jest ona konstruowana jako pierwsza. Funkcje mające za zadanie rozwiązanie podproblemów składowych problemu ogólnego są definiowane (lub importowane z biblioteki) w dalszej kolejności na podstawie konkretnych wymagań na danym poziomie analizy problemu.

UWAGA

Niektórzy programiści preferują jednak inny sposób programowania — alternatywny wobec „z góry na dół”. To **podejście „z dołu do góry”** (ang. *bottom-up approach*). Jest ono nierozerwalnie związane z programowaniem zorientowanym obiektowo.

Zasady programowania strukturalnego

Programowanie strukturalne wiąże się z zastosowaniem **struktur sterujących** (ang. *control structures*) przebiegiem działania programu, do których zalicza się:

- struktury sekwencji,
- struktury wyboru,
- struktury iteracyjne.

Struktury sekwencji (ang. *sequence structures*) należą do struktur wbudowanych do niemal każdego współczesnego języka programowania. Zapewniają one automatyczne wykonywanie kolejnych instrukcji w kodzie źródłowym programu w sposób sekwencyjny — jedna po drugiej (krok po kroku). Jeżeli w jednej linii programu znajduje się większa liczba instrukcji, są one wykonywane od lewej strony do prawej.

Struktury wyboru (ang. *selection structures*) to nie mniej ważny element składowy programu strukturalnego, ponieważ umożliwiają **sterowanie przepływem** (ang. *flow control*) w programie w zależności od spełnienia lub niespełnienia określonego warunku (warunków). Przy czym sterowanie przepływem należy rozumieć jako porządek (kolejność) wykonywania instrukcji w czasie wykonywania programu. Do struktur wyboru zaliczane są instrukcje warunkowe, np. `if`, `if-else`, oraz instrukcje wyboru, np. `switch`. Wspomniane instrukcje warunkowe pozwalają organizować i implementować w programie **kontrolowane rozgałęzienia**.

zienia (ang. *controlled branching*). Na przykład określony blok kodu jest wykonywany tylko wtedy, gdy określony — zadany — warunek jest spełniony. W przeciwnym razie sterowanie przechodzi do następnej instrukcji w programie (po instrukcji `if`) albo do innego bloku kodu, jeśli użyto instrukcji `if-else`.

Do **struktur iteracyjnych** (ang. *iteration structures*) zaliczane są pętle programowe, np. `while`, `do-while` oraz `for`. Stosowanie pętli pozwala na powtarzanie bloku kodu (czyli zestawu instrukcji) albo określona — zadana z góry — liczbę razy, albo dopóki określony warunek jest spełniony. Po wykonaniu zadanej liczby powtórzeń lub jeśli warunek nie jest spełniony, sterowanie przechodzi do instrukcji w programie występującej w kodzie źródłowym bezpośrednio po instrukcji pętli.

Następną ważną zasadą programowania strukturalnego jest to, że struktury sterujące można zagnieździć. Na przykład instrukcją składową pętli programowej może być instrukcja warunkowa lub inna pętla.

Program strukturalny stanowi uporządkowaną strukturę, w której poszczególne instrukcje są wykonywane sekwencyjnie jedna po drugiej oraz sterowanie przepływem jest realizowane za pomocą struktur sterujących omówionych powyżej. Program strukturalny nie obejmuje obsługi skoków z jednej instrukcji do drugiej, które powodują zaburzenie kolejności wykonywania instrukcji i tym samym niszczą jego zorganizowaną i uporządkowaną strukturę. Nie jest więc zalecane stosowanie w programie strukturalnym instrukcji `break` i `continue` jako instrukcji składowych pętli, ponieważ ich użycie powoduje „skoki” w jego kodzie, zaburzające kontrolę — sterowanie przepływem. To samo dotyczy wykorzystania instrukcji skoku `goto`, i to nie tylko wewnątrz pętli, ale w całym w programie.

W programie strukturalnym wykorzystywane są bloki funkcjonalne (np. funkcje), które mają jednoznacznie zdefiniowane interfejsy. Interfejsy te nie są bezpośrednio powiązane z konkretnymi danymi. Jednakże są one określone za pomocą liczby i typów danych. Jest to pewnym ograniczeniem w programowaniu strukturalnym.

Programy strukturalne czasem są nazywane modułowymi, co wynika z ich „modułowości”. Przy czym moduł jest rozumiany tutaj jako blok funkcjonalny, a nie jako biblioteka funkcji (jak np. w języku Pascal czy Delphi). Konstruowanie i organizowanie funkcjonalnych bloków kodu w programie strukturalnym opiera się na założeniu, że taki blok ma wyłącznie jedno wejście i jedno wyjście. Takie podejście zapewnia łatwe sterowanie przepływem we fragmencie programu zawierającym taki blok. Ponadto, im bardziej bloki — moduły programu strukturalnego — są od siebie niezależne, tym łatwiej jest programistom uniknąć błędów logicznych w procesie jego kodowania — implementacji. Dążenie do uniezależnienia od siebie bloków funkcjonalnych programu strukturalnego zwiększa także przejrzystość i czytelność programu. Ma również duży wpływ na łatwość jego testowania, debugowania oraz ewentualnej modyfikacji. W programowaniu strukturalnym przyjmuje się zasadę, że dowolna struktura sterująca (nawet zagnieżdziona), która ma jedno wejście i jedno wyjście, jest równoważna (funkcjonalnemu) blokowi kodu.

Każdy program proceduralny powinien być zgodny z zasadami programowania strukturalnego. Innymi słowy, każdy program proceduralny jest programem strukturalnym. Jednakże nie każdy program strukturalny jest programem proceduralnym. Na przykład można napisać dobry program strukturalny z wykorzystaniem w poprawny sposób struktur sterujących (konstrukcji warunkowych, pętli) — ale bez użycia jakichkolwiek podprogramów (funkcji).

Odniesienie programowania imperatywnego do programowania strukturalnego nasuwa w rezultacie analogiczny wniosek. Mianowicie każdy program strukturalny powinien spełniać założenia programowania imperatywnego. Jednakże nie każdy program imperatywny jest programem strukturalnym. Jako przykład można podać program imperatywny, w którym sterowanie jest realizowane za pomocą instrukcji `goto`. Taki program nie jest programem strukturalnym.

11.1.2. Programowanie zorientowane obiektowo

Idea programowania proceduralnego polega na definiowaniu podprogramów (np. funkcji), których zadaniem jest przetwarzanie danych. W programowaniu proceduralnym konkretne dane są niezależne od funkcji. W definicjach funkcji określa się jedynie liczbę i typ danych, które są przez nie przetwarzane. Faktyczne powiązanie funkcji z konkretnymi danymi następuje na etapie wywoływanego funkcji, na którym dane są argumentami tych wywołań. Zasady programowania strukturalnego zaś zapewniają poprawność struktury programu, która uwzględnia umiejętne stosowanie konstrukcji sterujących jego przebiegiem (działaniem).

Koncepcja **programowania zorientowanego obiektowo** (ang. *object-oriented programming*) to następny etap rozwoju metod i technik programowania, po programowaniu imperatywnym, proceduralnym i strukturalnym.

Głównym celem programowania obiektowego jest ścisłe powiązanie ze sobą funkcji i danych, inaczej niż w programowaniu proceduralnym i strukturalnym. Jeśli spojrzeć z perspektywy programowania obiektowego, wspomniane funkcje i dane powinny mieć szereg ważnych cech (właściwości) pozwalających na wzajemną komunikację pomiędzy nimi, regulowaną przez określone zasady. Na przykład wybrane funkcje i dane można hermetyzować (izolować) w określonych konstrukcjach programistycznych odgrywających rolę kontenerów. Wówczas te funkcje i dane są dostępne wyłącznie w zadanym kontenerze. Wspomniane kontenery pełnią wtedy funkcję pojemników (pułapek) ze skuteczną izolacją. Innym przykładem jest możliwość dziedziczenia w określonym kontenerze (kontenerach) wybranych funkcji i/lub danych zdefiniowanych w innym kontenerze lub kontenerach. Przez zastosowanie mechanizmu dziedziczenia funkcji i/lub danych pomiędzy różnymi kontenerami można uzyskać rozbudowaną strukturę kontenerów, które komunikują się ze sobą w sposób bezpośredni lub pośredni.

Programowanie obiektowe opiera się na kilku ważnych pojęciach, takich jak klasa i obiekt, oraz na założeniach opisujących zasady programowania zorientowanego obiektowo.

Klasa (ang. *class*) to podstawowy element konstrukcyjny w programowaniu obiektowym. Jest to złożony typ danych definiowany przez użytkownika (ang. *user-defined compound data type*). Klasa definiuje zawartość i właściwości kontenera, o którym była mowa wcześniej. W szczególności klasa zawiera definicje **elementów członkowskich** (ang. *members*). Ujmując to inaczej, klasa jednoznacznie definiuje zawartość kontenera, tj. jego elementy składowe (członkowskie) oraz wzajemne zależności pomiędzy nimi.

Do najważniejszych elementów członkowskich klasy należą **zmienne członkowskie** (ang. *member variables*), reprezentujące dane, oraz **metody** (ang. *methods*), odpowiadające funkcjom. Podstawowa idea konstrukcji klasy opiera się na założeniu, że metody (funkcje członkowskie) operują na zmiennych członkowskich klasy. Tym samym zapewnione jest ściśle określone powiązanie pomiędzy funkcjami — metodami a danymi — zmiennymi członkowskimi klasy.

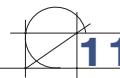
Obiekt (ang. *object*) z kolei jest **instancją** (ang. *instance*) danej klasy, czyli jej członkiem. Obiekt jest bytem rzeczywistym — aby móc z niego korzystać, należy go wcześniej utworzyć. Utworzony obiekt zajmuje w pamięci operacyjnej określony obszar, do którego można się odwołać za pośrednictwem adresu. Adres ten jest przechowywany w zmiennej referencyjnej, która zapewnia dostęp do jego wszystkich elementów członkowskich. Właściwości i zachowanie obiektu są określone zarówno przez zmienne członkowskie i metody zdefiniowane w klasie, do której należy ten obiekt, jak i przez właściwości samej klasy, np. to, czy jest ona powiązana z inną klasą.

Do najważniejszych założeń (cech) programowania obiektowego należą:

- hermetyzacja, inaczej **enkapsulacja** (ang. *encapsulation*),
- **dziedziczenie** (ang. *inheritance*),
- **polimorfizm** (ang. *polymorphism*),
- **abstrakcja** (ang. *abstraction*).

UWAGA

Wymienione powyżej cechy programowania zorientowanego obiektowo zostały omówione w dalszych rozdziałach podręcznika.



11.2. Definiowanie klas

Klasa stanowi złożony typ danych definiowany przez użytkownika — programistę. Definicja klasy nieco przypomina definicję C-struktury, która została omówiona wcześniej, w podrozdziale 7.1. Jednakże oprócz definicji zmiennych członkowskich definicja klasy zawiera definicje funkcji członkowskich, czyli metod.



UWAGA

Definicja klasy może obejmować także definicje innych elementów członkowskich, np. konstruktorów. Poszczególne składniki klasy zostały omówione w dalszej części podręcznika.

Ogólna postać definicji klasy jest następująca:

```
class nazwa_klasy {
    specyfikator_dostępu_1:
        definicje_zmiennych_członkowskich_1;
        deklaracje_funkcji_członkowskich_1;
        definicje_funkcji_członkowskich_1;
    specyfikator_dostępu_2:
        definicje_zmiennych_członkowskich_2;
        deklaracje_funkcji_członkowskich_2;
        definicje_funkcji_członkowskich_2;
    ...
};
```

gdzie:

- nazwa_klasy oznacza identyfikator klasy,
- specyfikator_dostępu_1, specyfikator_dostępu_2 to specyfikatory dostępu do elementów członkowskich klasy wymienionych poniżej danego specyfikatora,
- definicje_zmiennych_członkowskich_1, definicje_zmiennych_członkowskich_2 to definicje zmiennych członkowskich klasy o dostępności ustalonej za pomocą, odpowiednio, specyfikatora_dostępu_1 i specyfikatora_dostępu_2,
- deklaracje_funkcji_członkowskich_1, deklaracje_funkcji_członkowskich_2 to deklaracje funkcji członkowskich klasy o dostępności określonej za pomocą, odpowiednio, specyfikatora_dostępu_1 i specyfikatora_dostępu_2,
- definicje_funkcji_członkowskich_1, definicje_funkcji_członkowskich_2 to definicje funkcji członkowskich klasy o dostępności określonej za pomocą, odpowiednio, specyfikatora_dostępu_1 i specyfikatora_dostępu_2.

Specyfikator dostępu (ang. *access specifier*) pozwala ustalić dostępność do elementów członkowskich klasy, które zostały wymienione (zdefiniowane) w klasie poniżej niego. W języku C++ można wyróżnić trzy specyfikatory dostępu:

- public,
- private,
- protected.

Elementy członkowskie określone za pomocą specyfikatora `public` są dostępne wszędzie — zarówno w obrębie klasy, w której zostały zdefiniowane, jak i w otoczeniu (na zewnątrz) tej klasy — wszędzie tam, gdzie jest widoczny obiekt będący jej instancją. Składniki klasy, którym nadano status `public`, są nazywane **elementami członkowskimi publicznymi** (ang. *public members*).

Elementy członkowskie o dostępności ustalonej za pomocą specyfikatora `private` są dostępne wyłącznie w obrębie (we wnętrzu) klasy, w której zostały zdefiniowane. Tym samym są one widoczne tylko dla innych elementów członkowskich zdefiniowanych w tej samej klasie. Takie składniki klasy są nazywane **elementami członkowskimi prywatnymi** (ang. *private members*).

UWAGA

Specyfikator `protected` został omówiony w dalszej części podręcznika — w rozdziale 14., dotyczącym mechanizmu dziedziczenia.

Specyfikatory dostępu określone w definiowanej klasie mają wpływ na dostępność wszystkich elementów członkowskich tej klasy wymienionych (zadeklarowanych lub zdefiniowanych) poniżej danego specyfikatora. Zatem dotyczy to zarówno zmiennych, jak i funkcji członkowskich klasy. W języku C++ domyślnym specyfikatorem dostępu jest `private`. Oznacza to, że jeśli w odniesieniu do określonych elementów członkowskich w definicji klasy nie wyszczególniono w sposób jawnego żadnego specyfikatora dostępu, kompilator domyślnie przyjmie, że te elementy mają status „prywatny”.

UWAGA

Dostęp do prywatnych elementów członkowskich klasy mogą uzyskać również ich „przyjaciele” (ang. *friends*). Tematyka funkcji i klas zaprzyjaźnionych została zaprezentowana w dalszej części podręcznika — w rozdziale 17.

Zmienne członkowskie klasy (ang. *class member variables*) reprezentują dane, które są przechowywane w obiektach będących instancjami tej klasy. Zmienne członkowskie klasy definiuje się w analogiczny sposób jak w C-strukturach (podrozdział 7.1). Na przykład zmienne członkowskie o nazwach `bok1` i `bok2` należące do typu `float` można zdefiniować następująco: `float bok1, bok2;`.

UWAGA

Zmienne członkowskie klasy są również nazywane **danymi członkowskimi** (ang. *data members*) lub **polami** (ang. *fields*) — jak w C-strukturach.

W ogólności liczba zmiennych członkowskich jest dowolna — zależnie od potrzeb. Ich typy także są dowolne. Tym samym w skład klasy mogą wchodzić zarówno dane typów podstawowych, np. `int`, `float`, jak i dane należące do typów złożonych, np. tablice i C-struktury.

UWAGA

Klasy — podobnie jak tablice i C-struktury — są zaliczane do **typów danych agregacyjnych** (ang. *aggregate data types*), ponieważ umożliwiają grupowanie wielu indywidualnych danych w jednym kontenerze (pojemniku).

Funkcje członkowskie klasy (ang. *class member functions*) odpowiadają za realizację określonych działań (operacji) na danych przechowywanych w klasie, które są reprezentowane przez jej zmienne członkowskie. Ponadto mogą one pełnić funkcję wspomagającą w celu realizacji innych działań pomocniczych.

W klasie może być zawarta kompletna definicja funkcji członkowskiej albo tylko jej deklaracja — prototyp. W tym drugim przypadku definicja funkcji członkowskiej znajduje się na zewnątrz klasy. W definicji nazwa funkcji powinna być poprzedzona nazwą klasy, której ta funkcja jest członkiem — w połączeniu z operatorem zakresu (ang. *scope operator*), `::`. Na przykład nagłówek definicji funkcji członkowskiej o nazwie `pole`, która została zadeklarowana w klasie `Prostokat`, może mieć postać: `Prostokat::pole()`.

Przykład 11.1

```
class Pracownik {
public: // specyfikator dostępu
    // Deklaracje zmiennych członkowskich:
    string imie;
    string nazwisko;
    // Definicja funkcji członkowskiej:
    void wyswietlDane() {
        cout << "Dane pracownika: " << imie << " " << nazwisko << endl;
    }
};
```

W przedstawionym fragmencie kodu zawarto definicję klasy `Pracownik`. Klasa ta ma dwie zmienne członkowskie, `imie` i `nazwisko`, należące do typu łańcuchowego `string`. Ponadto zdefiniowano tam funkcję członkowską `wyswietlDane()`. Wszystkie składniki klasy `Pracownik` to jej publiczne elementy członkowskie.

Ćwiczenie 11.1

Zmodyfikuj kod źródłowy przedstawiony w przykładzie 11.1 — zdefiniuj klasę o nazwie `Uczeń`, która będzie miała następujące elementy członkowskie publiczne:

- zmienne członkowskie: `imie`, `nazwisko`, `klasa`, należące do typu łańcuchowego,

- funkcje członkowskie: `wyswietlDane()`, `wyswietlPersonalia()` i `wyswietlKlase()`.

Przy czym zadaniem funkcji `wyswietlDane()` jest wyświetlenie na ekranie monitora wartości przechowywanych we wszystkich zmiennych członkowskich klasy, `wyswietlPersonalia()` — wyświetlenie wyłącznie imienia i nazwiska ucznia, a `wyswietlKlase()` — wyświetlenie nazwy klasy.

Przykład 11.2

```
class Pracownik {
public:
    // Deklaracje zmiennych członkowskich:
    string imie, nazwisko;
    // Deklaracje funkcji członkowskich o nazwach ustawImie i ustawNazwisko:
    void ustawImie(string);
    void ustawNazwisko(string);
    // Definicja funkcji członkowskiej o nazwie wyswietlDane:
    void wyswietlDane() {
        cout << "Dane pracownika: " << imie << " " << nazwisko << endl;
    }
};

// Definicje funkcji członkowskich ustawImie() i ustawNazwisko() zadeklarowanych w klasie Pracownik:
void Pracownik::ustawImie(string pImie) {
    imie = pImie;
}
void Pracownik::ustawNazwisko(string pNazwisko) {
    nazwisko = pNazwisko;
}
```

W zaprezentowanym fragmencie kodu zawarto definicję klasy o nazwie `Pracownik`. Klasa ta ma dwie zmienne członkowskie o nazwach `imie` i `nazwisko` należące do typu łańcuchowego `string`.

Ponadto definicja klasy zawiera deklaracje (prototypy) dwóch funkcji członkowskich (metod): `ustawImie()` i `ustawNazwisko()`. Metody te zostały zdefiniowane na zewnątrz klasy. Nazwy funkcji w tych definicjach są poprzedzone nazwą klasy, do której funkcje te należą (`Pracownik`), wraz z operatorem zakresu `::`.

Ostatnim elementem członkowskim klasy `Pracownik` jest funkcja członkowska `wyswietlDane()`.

Wszystkie składniki klasy `Pracownik` mają status publiczny, ponieważ zostały zdefiniowane/zadeklarowane poniżej specyfikatora `public`.

Ćwiczenie 11.2

Zmodyfikuj kod źródłowy z przykładu 11.2 — zamiast klasy `Pracownik` zdefiniuj klasę `Uczeń`, która będzie miała następujące elementy członkowskie publiczne:

- zmienne członkowskie: `imie`, `nazwisko`, `klasa`, należące do typu łańcuchowego,
- funkcje członkowskie: `ustawImie()`, `ustawNazwisko()`, `ustawKlase()`, `wyswietlDane()`, `wyswietlPersonalia()` i `wyswietlKlase()`.

Funkcje: `ustawImie()`, `ustawNazwisko()` i `ustawKlase()` powinny umożliwiać nadanie/zmianę wartości zmiennej członkowskiej, odpowiednio, `imie`, `nazwisko`, `klasa`.

Zadaniem funkcji `wyswietlDane()` jest wyświetlenie na ekranie monitora wartości przechowywanych we wszystkich zmiennych członkowskich klasy, `wyswietlPersonalia()` — wyświetlenie imienia i nazwiska ucznia, a `wyswietlKlase()` — wyświetlenie nazwy klasy.

11.3. Deklarowanie zmiennych obiektowych

Jak już wspomiano, aby w programie można było korzystać z obiektu jako instancji określonej klasy, należy go wcześniej utworzyć.

UWAGA

W tym podrozdziale omawiany jest wyłącznie najprostszy sposób umożliwiający utworzenie obiektu — przez zadeklarowanie zmiennej obiektowej. W ogólności z procesem tworzenia obiektu wiąże się wywołanie tzw. **konstruktora** (ang. *constructor*). To zagadnienie zostało przedstawione w dalszej części podręcznika — w rozdziale 12.

Zmienne obiektowe (ang. *object variables*) można deklarować na dwa sposoby. Pierwszy z nich polega na tym, że listę wymaganych zmiennych umieszcza się bezpośrednio po definicji klasy — czyli tak samo jak w przypadku zmiennych strukturywych (podrozdział 7.1):

```
class nazwa_klasy {
    ...
} zmienna_1, zmienna_2, ... ;
```

gdzie `nazwa_klasy` oznacza identyfikator klasy, a zmienne `zmienna_1` i `zmienna_2` — identyfikatory zmiennych obiektowych.

Drugi sposób jest analogiczny do deklarowania zmiennych należących do typów podstawowych (ale również do deklarowania zmiennych strukturywych):

```
nazwa_klasy zmienna_1, zmienna_2, ... ;
```

Na przykład utworzenie obiektu `pracownik` należącego do klasy `Pracownik` można zrealizować za pomocą wyrażenia `Pracownik pracownik;`.

11.4. Odwołania do elementów członkowskich obiektów

Odwołania do zmiennych członkowskich obiektu można realizować przy użyciu **operatora wyboru elementu członkowskiego** (ang. *member selection operator*) . (ang. *dot*). Postać ogólna takiego odwołania jest następująca:

zmienna_obejktowa.zmienna_członkowska

gdzie **zmienna_obejktowa** jest identyfikatorem zmiennej obiektowej, a **zmienna_członkowska** — identyfikatorem określonej zmiennej członkowskiej obiektu.

Na przykład odwołania do zmiennych członkowskich **imie** i **nazwisko** należących do obiektu **pracownik** mają postać, odpowiednio: **pracownik.imie** oraz **pracownik.nazwisko**.

Odwołania do funkcji członkowskich obiektu realizuje się w podobny sposób jak odwołania do zmiennych członkowskich, z uwzględnieniem zasad wywoływania zwykłych funkcji (omówionych w podrozdziale 8.2). Na przykład odwołanie się do bezparametrowej funkcji członkowskiej **wyswietlDane()** obiektu **pracownik** ma postać: **pracownik.wyswietlDane()**.

UWAGA

Operator **.** jest również nazywany **operatorem dostępu do elementu członkowskiego** (ang. *member access operator*).

Przykład 11.3

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Deklaracje zmiennych członkowskich:
    string imie, nazwisko;
    // Deklaracje funkcji członkowskich:
    void ustawImie(string);
    void ustawNazwisko(string);
    // Definicja funkcji członkowskiej:
    void wyswietlDane() {
        cout << "Dane pracownika: " << imie << " " << nazwisko << endl;
    }
};
```

```
// Definicje funkcji członkowskich ustawImie() i ustawNazwisko() należących do klasy Pracownik:  
void Pracownik::ustawImie(string pImie) {  
    imie = pImie;  
}  
void Pracownik::ustawNazwisko(string pNazwisko) {  
    nazwisko = pNazwisko;  
}  
  
int main() {  
    // Utworzenie obiektu pracownik jako instancji klasy Pracownik:  
    Pracownik pracownik;  
    // Odwołanie się do zmiennych członkowskich obiektu pracownik:  
    pracownik.imie = "Jan";  
    pracownik.nazwisko = "Kowalski";  
    // Wywołanie funkcji członkowskiej (metody) należącej do obiektu pracownik:  
    pracownik.wyswietlDane();  
    // Wywołanie funkcji członkowskich ustawImie() i ustawNazwisko():  
    pracownik.ustawImie("Adam");  
    pracownik.ustawNazwisko("Nowak");  
    // Ponowne wywołanie funkcji członkowskiej (metody) należącej do obiektu pracownik:  
    pracownik.wyswietlDane();  
  
    return 0;  
}
```

W programie pokazano kompletną definicję klasy Pracownik, zawierającą zarówno zmienne, jak i funkcje członkowskie — metody. Przy czym metody ustawImie() i ustawNazwisko() zdefiniowano na zewnątrz definicji klasy.

W programie głównym utworzono zmienną obiektową — obiekt pracownik, będący instancją klasy Pracownik.

Bezpośrednie odwołania do zmiennych członkowskich imie i nazwisko obiektu pracownik o postaciach pracownik.imie i pracownik.nazwisko są realizowane w celu nadania im wartości, odpowiednio, "Jan" i "Kowalski". Wartości te są następnie wyświetlane kontrolnie na ekranie jako skutek wywołania funkcji członkowskiej wyswietlDane().

W kolejnych instrukcjach wartości zmiennych członkowskich imie i nazwisko są modyfikowane. Operacje te są wykonywane (w sposób pośredni) poprzez wywołania metod ustawImie() i ustawNazwisko() z argumentami, odpowiednio, "Adam" oraz "Nowak". Na końcu zmodyfikowane wartości zmiennych imię i nazwisko są ponownie prezentowane na ekranie monitora.

Ćwiczenie 11.3

Na podstawie kodu zawartego w przykładzie 11.3 napisz program pozwalający na przetwarzanie danych ucznia: imienia, nazwiska, roku urodzenia, klasy, grupy. Wymienione dane ucznia

należy zapamiętać w zmiennych członkowskich obiektu `uczen` należącego do zdefiniowanej samodzielnie klasy `Uczen`. W klasie `Uczen` należy zdefiniować ponadto metody umożliwiające pobranie danych ucznia z klawiatury oraz wyświetlenie wszystkich (jak i wybranych) danych ucznia na ekranie monitora.

Przykład 11.4

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

// Definicja klasy Prostokat:
class Prostokat {
public: // Wszystkie elementy członkowskie są publiczne.
    // Deklaracje zmiennych członkowskich:
    float bok1, bok2;
    // Deklaracje (prototypy) funkcji członkowskich:
    float pole();
    float obwod();
};

// Definicje funkcji członkowskich zadeklarowanych w klasie Prostokat:
float Prostokat::pole() {
    return bok1 * bok2;
}
float Prostokat::obwod() {
    return 2 * bok1 + 2 * bok2;
}

int main() {
    // Utworzenie obiektu prostokat jako instancji klasy Prostokat:
    Prostokat prostokat;

    // Przypisanie zadanego wartości do zmiennych członkowskich obiektu prostokat:
    prostokat.bok1 = 1;
    prostokat.bok2 = 2;
    // Kontrolne wyświetlenie długości boków prostokąta:
    cout << "Pierwszy bok = " << prostokat.bok1 << endl;
    cout << "Drugi bok = " << prostokat.bok2 << endl;

    // Obliczenie pola i obwodu prostokąta i wyświetlenie ich wartości na ekranie monitora:
    cout << "Pole wynosi: " << prostokat.pole() << endl; // odwołanie się do metody pole()
    cout << "Obwód wynosi: " << prostokat.obwod() << endl; // odwołanie się do metody
                                                // obwod()

    return 0;
}
```

W przedstawionym programie obliczane są pole i obwód prostokąta dla zadanych długości jego boków. Wyniki obliczeń są prezentowane na ekranie monitora.

Klasa `Prostokat` zawiera dwie zmienne członkowskie, `bok1` i `bok2`, które reprezentują boki prostokąta. Zadaniem bezparametrowych funkcji członkowskich (metod) `pole()` i `obwod()` jest obliczenie, odpowiednio, pola i obwodu prostokąta. Wszystkie elementy członkowskie klasy `Prostokat` są publiczne. Oznacza to, że można z nich korzystać:

- w obrębie klasy, w której zostały zdefiniowane/zadeklarowane,
- wszędzie tam, gdzie jest widoczny obiekt `prostokat` jako instancja klasy `Prostokat`.

W pierwszym z wymienionych powyżej przypadków funkcje członkowskie `pole()` i `obwod()` wykorzystują obie zmienne członkowskie, `bok1` i `bok2`, w obrębie definicji klasy `Prostokat`. W drugim zaś odwołania do zmiennych i funkcji członkowskich klasy `Prostokat` są realizowane na zewnątrz tej klasy — w programie głównym.

Program w przejrzysty sposób ilustruje ścisłe powiązanie funkcji z danymi, co stanowi jedną z najważniejszych cech programowania obiektowego. Dotyczy to w szczególności funkcji członkowskich `pole()` i `obwod()` oraz danych przechowywanych w zmiennych członkowskich `bok1` i `bok2`.

Ćwiczenie 11.4

Na podstawie kodu źródłowego zawartego w przykładzie 11.4 napisz program pozwalający obliczyć pole i obwód kwadratu. Wykorzystaj obiekt `kwadrat` jakoinstancję zdefiniowanej samodzielnie klasy `Kwadrat`. Klasa ta powinna zawierać definicje elementów członkowskich umożliwiających przechowanie długości boku kwadratu oraz obliczenie jego pola i obwodu.

Przykład 11.5

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

// Definicja C-struktury Data:
struct Data {
    // Deklaracja pól struktury:
    int dd, mm, rr;
};

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Deklaracja zmiennych członkowskich połączona z ich inicjalizacją zerową:
    int id {};
    string imie {}, nazwisko {};
    Data data_urodzenia {};// Zmienna członkowska data_urodzenia należy do typu struktury Data.
};
```

```

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy Pracownik:
    Pracownik pracownik;

    // Wyświetlenie kontrolne bieżących wartości zmiennych członkowskich obiektu pracownik:
    cout << "Numer identyfikacyjny " << pracownik.id << endl;
    cout << "Imię: " << pracownik.imie << endl;
    cout << "Nazwisko: " << pracownik.nazwisko << endl;
    cout << "Data urodzenia:" << endl;
    cout << "dzień = " << pracownik.data_urodzenia.dd << endl;
    cout << "miesiąc = " << pracownik.data_urodzenia.mm << endl;
    cout << "rok = " << pracownik.data_urodzenia.rr << endl;

    return 0;
}

```

W programie zawartym w przykładzie zdefiniowano klasę `Pracownik`, która zawiera zmienne członkowskie należące do typów predefiniowanych (`int`, `string`), jak też typów złożonych (`Data`). C-struktura `Data` jest typem zdefiniowanym przez użytkownika.

W przedstawionym programie pokazano również, że definicja zmiennych członkowskich klasy może być połączona z ich inicjalizacją. Tutaj wykorzystano inicjalizację zerową. W ogólności inicjalizacja zmiennych członkowskich klasy połączona z ich deklarowaniem w obrębie definicji klasy jest prowadzona według tych samych zasad, zgodnie z którymi inicjuje się zwykłe zmienne.

UWAGA

Zasadniczo za inicjalizację zmiennych członkowskich obiektów odpowiadają konstruktory, które zostały szczegółowo omówione w rozdziale 12. podręcznika.

Ćwiczenie 11.5

Zmodyfikuj program zawarty w przykładzie 11.5 — zdefiniuj klasę `Uczeń`, która pozwala przechować wybrane dane ucznia: numer w dzienniku, imię, nazwisko, płeć, klasę, grupę oraz datę i miejsce urodzenia.

11.5. Statyczne elementy członkowskie klas

Oprócz zwykłych zmiennych i funkcji członkowskich klasa może zawierać **statyczne elementy członkowskie** (ang. *static members*). Do statycznych elementów członkowskich należą:

- zmienne członkowskie (pola) statyczne,
- funkcje członkowskie (metody) statyczne.

11.5.1. Statyczne zmienne członkowskie

Statyczne zmienne członkowskie (ang. *static member variables*) zadeklarowane w określonej klasie są wspólne dla wszystkich obiektów należących do tej klasy. Z drugiej strony nie są one powiązane z żadnym obiektem należącym do tej klasy. Oznacza to, że nawet jeśli żaden obiekt jako instancja określonej klasy nie zostanie utworzony, statyczne zmienne członkowskie będą istnieć i będzie można z nich korzystać. Dlatego też często nazywa się je **zmiennymi klasowymi** (ang. *class variables*) — w odróżnieniu od zwykłych zmiennych członkowskich, które są nazywane **zmiennymi instancyjnymi** (ang. *instance variables*).

Najważniejsza różnica pomiędzy zmiennymi klasowymi a zmiennymi instancyjnymi polega na tym, że zmiennych klasowych można używać nawet wtedy, gdy żaden obiekt będący instancją klasy nie został utworzony. Z drugiej strony zastosowanie zmiennych instancyjnych wymaga istnienia, czyli wcześniejszego utworzenia, obiektu.

Deklaracja zmiennej klasowej — w odróżnieniu od deklaracji zwykłej zmiennej członkowskiej — jest poprzedzona słowem kluczowym `static`, np. `static string s_stanowisko;`, gdzie `s_stanowisko` jest identyfikatorem zmiennej klasowej. Deklaracja zmiennej klasowej jest realizowana w obrębie definicji klasy. Natomiast jej inicjalizacja musi być przeprowadzona na zewnątrz definicji tej klasy — w zakresie globalnym (ang. *global scope*). Wyjątkiem jest inicjalizacja statycznych „stałych” `const` należących do któregoś z typów porządkowych (ang. *integral types*), tj. całkowitego, znakowego, logicznego i wyliczeniowego, którą można zrealizować wewnątrz definicji klasy, np. `static const int s_id {100};`.

Do zawartości zmiennej klasowej można się odwołać w dwojaki sposób:

- za pośrednictwem klasy,
- poprzez obiekty będąceinstancjami klasy.

Jednakże ze względu na to, że zmienne klasowe istnieją niezależnie od obiektów, lepszym rozwiązaniem jest odwoływanie się do nich jako członków klas, a nie obiektów. W praktyce robi się to przy użyciu operatora zakresu (ang. *scope operator*) `::`, np. `Pracownik::s_stanowisko`, gdzie `Pracownik` jest identyfikatorem klasy, a `s_stanowisko` to identyfikator zmiennej klasowej.

Przykład 11.6

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

// Definicja klasy Pracownik:
class Pracownik {
public: // Wszystkie elementy członkowskie klasy Pracownik są publiczne.
    // Deklaracje statycznych zmiennych członkowskich (zmiennych klasowych):
    static string s_szkola;
    static string s_stanowisko;
```

```

// Deklaracje zmiennych członkowskich instancyjnych:
string imie;
string nazwisko;
// Prototyp funkcji członkowskiej:
void wyswietlDane();
};

// Inicjalizacja zmiennych klasowych s_szkola i s_stanowisko:
string Pracownik::s_szkola = "Technikum Informatyczne";
string Pracownik::s_stanowisko = "nauczyciel";
// Definicja funkcji członkowskiej wyswietlDane() zadeklarowanej w klasie Pracownik:
void Pracownik::wyswietlDane() {
    cout << "Dane pracownika: " << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
    cout << "Szkoła: " << s_szkola << endl;
    cout << "Stanowisko: " << s_stanowisko << endl;
}

// PROGRAM GŁÓWNY:
int main() {
    // Utworzenie obiektu pracownik1:
    Pracownik pracownik1;
    // Przypisanie zadanego wartości zmiennym instancyjnym obiektu pracownik1:
    pracownik1.imie = "Jan";
    pracownik1.nazwisko = "Kowalski";
    // Wywołanie metody instancyjnej wyswietlDane():
    pracownik1.wyswietlDane();
    // Zmiana wartości zmiennej klasowej s_stanowisko:
    Pracownik::s_stanowisko = "portier";
    /* UWAGA
     * Odwołanie się do zmiennej klasowej s_stanowisko jest tutaj realizowane przy użyciu nazwy klasy,
     * w której zmienna ta została zadeklarowana, oraz operatora zakresu ::.
     * Nowa (zmieniona) wartość zmiennej klasowej s_stanowisko obowiązuje dla wszystkich obiektów klasy.
     *
     * Możliwość odwołania się do zmiennej klasowej s_stanowisko w programie głównym, a więc na zewnątrz klasy
     * Pracownik, w której zmienna ta została zadeklarowana, wynika z jej publicznego statusu.
     */
}

// Utworzenie obiektu pracownik2:
Pracownik pracownik2;
// Przypisanie wartości zmiennym instancyjnym obiektu pracownik2:
pracownik2.imie = "Jan";
pracownik2.nazwisko = "Nowak";
// Wywołanie metody wyswietlDane():
pracownik2.wyswietlDane();

return 0;
}

```

W programie zdefiniowano klasę `Pracownik`, która zawiera dwie zmienne klasowe, `s_szkoła` i `s_stanowisko`. W zmiennej `s_szkoła` przechowywana jest nazwa szkoły, w której jest zatrudniony pracownik, a w zmiennej `s_stanowisko` — stanowisko służbowe tego pracownika.

Zmienne klasowe `s_szkoła` i `s_stanowisko` to publiczne elementy członkowskie klasy `Pracownik`. Tym samym na zewnątrz klasy `Pracownik` dostęp do nich można uzyskać w sposób bezpośredni. Zilustrowano to za pomocą instrukcji `Pracownik::s_stanowisko = "dyrektor";`, w której następuje modyfikacja wartości przechowywanej w zmiennej klasowej `s_stanowisko`. Odwołanie do tej zmiennej zrealizowano za pośrednictwem klasy (`Pracownik`) przy użyciu operatora zakresu `::`.

Ćwiczenie 11.6

Zmodyfikuj program z przykładu 11.6 — zamiast klasy `Pracownik` zdefiniuj klasę `Uczeń` zawierającą dwie zmienne klasowe: `s_klasa` i `s_zawod`. W zmiennej `s_klasa` powinna być przechowywana nazwa klasy, do której uczeń uczęszcza (np. 2a), a w zmiennej `s_zawod` — zawód, w którym się kształci (np. technik programista). Wykorzystaj zdefiniowane zmienne klasowe w programie: wyświetl na ekranie monitora ich bieżące wartości, zmodyfikuj je, a następnie ponownie odczytaj i wyświetl.

11.5.2. Statyczne funkcje członkowskie

Statyczne funkcje członkowskie (ang. *static member functions*), inaczej **metody statyczne** (ang. *static methods*), są oznaczone w definicji klasy słowem kluczowym `static`. Metody statyczne są wspólne dla wszystkich obiektów stanowiących instancje tej klasy. Żeby móc korzystać z metod statycznych, nie trzeba tworzyć obiektów będących instancjami klasy — metody statyczne są od nich zupełnie niezależne.

Definicje metod statycznych mogą się znajdować zarówno wewnętrz, jak i na zewnątrz klasy — jest to zupełnie obojętne.

W ogólności metody statyczne mogą korzystać ze statycznych zmiennych członkowskich klasy (zmiennych klasowych), innych metod statycznych i dowolnych funkcji na zewnątrz klasy.

Metodę statyczną, podobnie jak zmienną klasową, można wywołać na dwa sposoby:

- za pośrednictwem klasy,
- za pośrednictwem obiektów — instancji klasy.

Jednakże ze względu na to, że metody statyczne są niezależne od obiektów, lepszym rozwiązaniem jest wywoływanie metod statycznych jako członków klas, a nie obiektów.

Jednym z najczęstszych zastosowań metod statycznych jest rola akcesorów, czyli publicznych metod dostępowych do prywatnych zmiennych klasowych.

Przykład 11.7

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

// Definicja klasy Pracownik:
class Pracownik {
    // Deklaracja prywatnej zmiennej członkowskiej statycznej o nazwie s_szkola:
    static string s_szkola;
public:
    // Deklaracje zmiennych członkowskich instancjnych:
    string imie;
    string nazwisko;
    // Prototyp metody statycznej s_zwrocSzkola():
    static string s_zwrocSzkola();
    // Definicja metody statycznej s_ustawSzkola():
    static void s_ustawSzkola(string pSzkoła) {
        s_szkola = pSzkoła;
    }
    // Definicja metody instancjnej wyswietlDane():
    void wyswietlDane() {
        cout << "Imię: " << imie << endl;
        cout << "Nazwisko: " << nazwisko << endl;
        cout << "Szkoła :" << s_szkola << endl;
    }
};

// Inicjalizacja zmiennej klasowej s_szkola:
string Pracownik::s_szkola = "Technikum Informatyczne";
// Definicja metody statycznej s_pobierzSzkoła():
string Pracownik::s_zwrocSzkola() {
    return s_szkola;
}

// PROGRAM GŁÓWNY
int main() {
    // Utworzenie obiektu pracownik1:
    Pracownik pracownik1;
    // Nadanie wartości zmiennym członkowskim instancjnym obiektu pracownik1:
    pracownik1.imie = "Jan";
    pracownik1.nazwisko = "Kowalski";
    cout << "Dane pracownika: " << endl;
    // Odwołanie się do zmiennych instancjnych obiektu pracownik1:
    cout << "Imię: " << pracownik1.imie << endl;
    cout << "Nazwisko: " << pracownik1.nazwisko << endl;
}
```

```

// Odwołanie się do zmiennej klasowej s_szkola za pośrednictwem metody statycznej s_pobierzSzkola():
cout << "Szkoła: " << Pracownik::s_zwrocSzkola() << endl;

// Utworzenie obiektu pracownik2:
Pracownik pracownik2;
// Nadanie wartości zmiennym instancyjnym obiektu pracownik2:
pracownik2.imie = "Adam";
pracownik2.nazwisko = "Nowak";
// Zmiana wartości zmiennej klasowej s_szkola za pośrednictwem metody statycznej s_ustawSzkola():
Pracownik::s_ustawSzkola("Technikum Elektroniczne");
cout << "Dane pracownika: " << endl;
cout << "Imię: " << pracownik2.imie << endl;
cout << "Nazwisko: " << pracownik2.nazwisko << endl;
cout << "Szkoła: " << Pracownik::s_zwrocSzkola() << endl;

return 0;
}

```

W programie zdefiniowano klasę `Pracownik`, która zawiera prywatną zmienną klasową o nazwie `s_szkola`. Dlatego też dostęp do tej zmiennej na zewnątrz klasy można uzyskać wyłącznie za pomocą publicznych metod dostępowych. W programie zdefiniowano dwie takie metody: `s_ustawSzkola()` i `s_zwrocSzkola()`. Są to metody statyczne. Definicja metody `s_ustawSzkola()` jest umiejscowiona wewnątrz klasy, a metody `s_zwrocSzkola()` — na zewnątrz. Zadaniem metody `s_ustawSzkola()` jest nadanie/zmiana wartości zmiennej klasowej `s_szkola`. Metoda `s_zwrocSzkola` zaś zwraca na zewnątrz wartość tej zmiennej.

Odwołanie się w programie głównym do zmiennej klasowej `s_szkola` jest realizowane w sposób pośredni. Funkcję „pośredników” pełnią metody statyczne `s_ustawSzkola()` i `s_zwrocSzkola()`.

Ćwiczenie 11.7

Zmodyfikuj program zawarty w przykładzie 11.7 — zdefiniuj jako prywatną dodatkową zmienną klasową o nazwie `s_stanowisko` oraz dwie metody statyczne: `s_ustawStanowisko()` i `s_zwrocStanowisko()`, z których pierwsza będzie miała za zadanie nadanie/zmianę wartości zmiennej klasowej `s_stanowisko`, a druga — odczytanie wartości tej zmiennej. Wykorzystaj w programie zdefiniowane funkcje statyczne.

11.6. Funkcje członkowskie typu inline

Podobnie jak zwykłe funkcje, również funkcje członkowskie klas mogą być funkcjami wbudowanymi typu `inline`. Funkcja członkowska `inline` to funkcja, której kod jest wstawiany w linii zawierającej jej wywołanie. Działanie to jest realizowane na etapie komplikacji programu.

**UWAGA**

„Zwykłe” funkcje inline zostały omówione w podrozdziale 8.6.

W ogólności funkcje członkowskie `inline` nie powinny zawierać żadnych pętli, zagnieżdżonych instrukcji warunkowych, instrukcji wyboru, instrukcji skoku itd. Ponadto takie funkcje nie mogą korzystać ze zmiennych statycznych.

Każda funkcja `inline` musi być zdefiniowana przed jej wywołaniem.

Przykład 11.8

```
#include <iostream>
#include <cmath>
using namespace std;
// Definicja klasy Pokoj:
class Pokoj {
public:
    float dlugosc;
    float szerokosc;
    // Definicje funkcji członkowskich inline:
    inline int dlugoscCalk() {
        return round(dlugosc);
    }
    inline int szerokoscCalk() {
        return round(szerokosc);
    }
};

int main() {
    Pokoj pokoj;
    pokoj.dlugosc = 1.51;
    pokoj.szerokosc = 2.51;

    // Wywołania funkcji członkowskich inline:
    cout << "Długość pokoju (zaokrąglona): " << pokoj.dlugoscCalk() << endl;
    cout << "Szerokość pokoju (zaokrąglona): " << pokoj.szerokoscCalk()
        << endl;

    return 0;
}
```

W klasie `Pokoj` zdefiniowano dwie funkcje członkowskie `inline`: `dlugoscCalk` i `szerokoscCalk`. Zadaniem tych funkcji jest zaokrąglenie wartości przechowywanych w zmiennych członkowskich `dlugosc` i `szerokosc`.

Wywołania funkcji `dlugoscCalk()` i `szerokoscCalk` w programie głównym są zastępowane ich „rozwiniętym” kodem.

Ćwiczenie 11.8

Zmodyfikuj program zawarty w przykładzie 11.8 — w klasie `Pokoj` zdefiniuj, a następnie wywołaj w programie głównym funkcje `inline`: `dlugoscGora()` i `szerokoscGora()` oraz `dlugoscDol()` i `szerokoscDol()`. Zadaniem wymienionych funkcji jest zaokrąglenie wartości zmiennych członkowskich `dlugosc` i `szerokosc` do najbliższej liczby całkowitej, odpowiednio, w górę i w dół. Wywołaj zdefiniowane funkcje w programie głównym.



11.7. Wskaźniki do obiektów

Na temat wskaźników do obiektów można spojrzeć w analogiczny sposób jak na zagadnienie wskaźników do C-struktur. Dlatego też zostaną tutaj przedstawione jedynie dwa przykłady ilustrujące ten temat.

UWAGA

Wykorzystanie wskaźników do C-struktur jest omówione w podrozdziale 7.1.4.

W pierwszym z przedstawionych przykładów (przykład 11.9) wykorzystano obiekt, dla którego pamięć operacyjna została przydzielona na stosie. Drugi przykład (przykład 11.10) pokazuje, jak utworzyć obiekt, dla którego pamięć została zaalokowana w sposób dynamiczny na stercie.

Przykład 11.9

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    string imie;
    string nazwisko;
    void wyswietlDane() {
        cout << "Imię: " << imie << endl;
        cout << "Nazwisko: " << nazwisko << endl;
    }
};

int main() {
    // Utworzenie obiektu pracownik należącego do klasy Pracownik:
    Pracownik pracownik;
```

```

// Deklaracja i inicjalizacja zmiennej wskaźnikowej (wskaźnika) w_pracownik na dowolny obiekt należący
// do typu Pracownik:
Pracownik *w_pracownik = nullptr;
// Przypisanie wskaźnikowi w_pracownik adresu obiektu pracownik:
w_pracownik = &pracownik;
// Nadanie wartości zmiennym członkowskim obiektu pracownik:
w_pracownik->imie = "Jan";
w_pracownik->nazwisko = "Kowalski";
// Prezentacja danych zapisanych w zmiennych członkowskich obiektu pracownik na ekranie monitora:
w_pracownik->wyswietlDane();

return 0;
}

```

W programie utworzono obiekt `pracownik` jako instancję zdefiniowanej samodzielnie klasy `Pracownik`. Pamięć operacyjna dla obiektu `pracownik` została zaalokowana w sposób statyczny (tj. podczas komplikacji programu) na stosie.

Ponadto zadeklarowano tutaj i zainicjowano wartością `nullptr` wskaźnik `w_pracownik`, który może (z definicji) wskazywać na dowolny obiekt należący do klasy `Pracownik`. Wskaźnikowi `w_pracownik` przypisano następnie adres obiektu `pracownik`: `w_pracownik = &pracownik;`. Tym samym po wykonaniu podanej instrukcji wskaźnik ten wskazuje na obiekt `pracownik`.

Odwołania do poszczególnych elementów członkowskich obiektu `pracownik` (tj. `w_pracownik->imie` oraz `w_pracownik->nazwisko`) są realizowane przy użyciu wskaźnika `w_pracownik` oraz operatora wyboru elementu członkowskiego `->`, nazywanego także operatorem strzałkowym.

Ćwiczenie 11.9

Zmodyfikuj program zawarty w przykładzie 11.9 — w klasie `Pracownik` zdefiniuj dodatkową zmienną członkowską `data_zatrudnienia`, będącą C-strukturą. Wspomniana zmienna powinna umożliwiać zapamiętanie daty zatrudnienia pracownika. Utwórz i zainicjuj jako instancję klasy `Pracownik` obiekt `pracownik`, dla którego pamięć operacyjna została przydzielona statycznie na stosie. Odwołania do elementów członkowskich obiektu `pracownik` zrealizuj przy użyciu wskaźnika.

Przykład 11.10

```

#include <iostream>
using namespace std;

// Definicja klasy Pracownik:
class Pracownik {
public:
    string imie;
    string nazwisko;

```

```

void wyswietlDane() {
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
}
};

int main() {
    // Deklaracja zmiennej wskaźnikowej (wskaźnika) w_pracownik na dowolny obiekt należący do typu Pracownik:
    Pracownik *w_pracownik;
    // Utworzenie obiektu typu Pracownik (instancji klasy Pracownik) wskazywanego przez wskaźnik w_pracownik:
    w_pracownik = new Pracownik();
    // Nadanie wartości zmiennym członkowskim utworzonego obiektu:
    w_pracownik->imie = "Jan";
    w_pracownik->nazwisko = "Kowalski";
    // Prezentacja danych zapisanych w zmiennych członkowskich obiektu na ekranie monitora:
    w_pracownik->wyswietlDane();
    // Usunięcie (zniszczenie) obiektu wskazywanego przez wskaźnik w_pracownik:
    delete w_pracownik;

    return 0;
}

```

W programie utworzono obiekt będący instancją klasy `Pracownik`, dla którego pamięć operacyjna została zaalokowana w sposób dynamiczny na stercie. Obiekt ten jest wskazywany przez wskaźnik `w_pracownik`.

Odwołania do poszczególnych elementów członkowskich obiektu `pracownik` są realizowane przy użyciu wskaźnika `w_pracownik` oraz operatora strzałkowego `->`.

Ćwiczenie 11.10

Zmodyfikuj program zawarty w przykładzie 11.10 — zamiast klasy `Pracownik` zdefiniuj klasę `Uczeń`, pozwalającą zapamiętać w jej zmiennych członkowskich następujące dane ucznia: imię, nazwisko, klasę, grupę, numer w dzienniku oraz datę urodzenia. Zmienna członkowska `data_urodzenia` powinna umożliwiać zapamiętanie w C-strukturze daty urodzenia ucznia. Utwórz obiekt będący instancją klasy `Pracownik`, dla którego pamięć operacyjna została przydzielona w sposób dynamiczny na stercie. Odwołania do elementów członkowskich tego obiektu zrealizuj przy użyciu wskaźnika.

11.8. Przekazywanie obiektów jako parametrów funkcji

Obiekty, jako instancje klas, mogą mieć bardzo rozbudowaną i złożoną strukturę wewnętrzną. Zapotrzebowanie na pamięć operacyjną potrzebną do zapamiętania takiego obiektu może być

znaczne. Tym samym przekazywanie obiektów do funkcji za pośrednictwem wartości może w przypadku ogólnym prowadzić do problemów dwojakiego rodzaju. Pierwszym z nich jest znaczne zwiększenie popytu programu na pamięć operacyjną, gdyż przy przekazywaniu argumentu przez wartość tworzona jest na stosie jego kopia. Drugim problemem jest spowolnienie działania programu, spowodowane długim czasem przeznaczonym na kopiowanie obiektu.

Biorąc pod uwagę powyższe, w przypadku dużych obiektów odpowiadające im parametry funkcji należy definiować jako:

- przekazywane przez referencje albo
- przekazywane przez wskaźniki.

Parametry wejściowe funkcji (ang. *function input parameters*) najlepiej jest definiować jako referencje albo wskaźniki do obiektów traktowanych jako stałe `const`. W wyniku tego w pamięci operacyjnej nie będzie wykonywana kopia argumentu (obiektu), lecz jedynie — w przypadku wskaźnika — kopia tego wskaźnika, która zajmuje 4 bajty lub 8 bajtów pamięci. Ponadto zdefiniowanie parametrów wejściowych w taki sposób zapewnia, że obiekty, jako argumenty funkcji, są traktowane w jej ciele jako mające status „read-only”, czyli zmiana wartości ich zmiennych członkowskich nie jest możliwa.

Parametry wyjściowe funkcji (ang. *function output parameters*) zaleca się definiować jako referencje albo wskaźniki do obiektów. To samo dotyczy **danych zwracanych przez funkcje** (ang. *data returned by functions*).

Przekazywanie obiektów jako parametrów/argumentów funkcji zilustrowano za pomocą dwóch przykładów. Pierwszy z nich (przykład 11.11) dotyczy przekazywania referencji do obiektów jako parametrów/argumentów funkcji. W drugim przykładzie (przykład 11.12) pokazano, jak przekazywać wskaźniki do obiektów do funkcji i z funkcji.

Przykład 11.11

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    string imie;
    string nazwisko;
    void wyswietlDane() {
        cout << "Imię: " << imie << endl;
        cout << "Nazwisko: " << nazwisko << endl;
    }
};

// Prototypy funkcji zewnętrznych:
Pracownik& pobierzDane(Pracownik&);
void wyswietlDane(const Pracownik&);
```

```
// PROGRAM GŁÓWNY
int main() {
    // Utworzenie obiektu pracownika jako instancji klasy Pracownik:
    Pracownik pracownik;
    // Pobranie danych pracownika z klawiatury:
    pracownik = pobierzDane(pracownik);
    // Prezentacja danych pracownika na ekranie monitora:
    wyswietlDane(pracownik);

    return 0;
}

// Definicje funkcji zewnętrznych:
Pracownik& pobierzDane(Pracownik& p) {
    cout << "Imię = "; cin >> p.imie;
    cout << "Nazwisko = "; cin >> p.nazwisko;

    return p;
}
void wyswietlDane(const Pracownik& p) {
    cout << "Imię: " << p.imie << endl;
    cout << "Nazwisko: " << p.nazwisko << endl;
}
```

W programie zdefiniowano klasę `Pracownik` oraz dwie niezależne funkcje zewnętrzne: `pobierzDane()` i `wyswietlDane()`. Zadaniem funkcji `pobierzDane()` jest pobranie danych pracownika (tj. imienia i nazwiska) z klawiatury, a funkcji `wyswietlDane()` — wyświetlenie tych danych na ekranie monitora.

Funkcja `pobierzDane()` przekazuje do swojego otoczenia obiekt klasy `Pracownik` przez referencję. Przy tym przekazanie jest realizowane zarówno za pośrednictwem parametru/argumentu tej funkcji, jak i przez wartość, którą funkcja zwraca. Funkcja `pobierzDane()` ma zatem dwa wyjścia, określone jako referencje do obiektu klasy `Pracownik`.

Z drugiej strony referencja do obiektu klasy `Pracownik`, traktowanego jako stała `const`, jest parametrem (argumentem) funkcji `wyswietlDane()`. Parametr ten odgrywa rolę wejścia funkcji.

Ćwiczenie 11.11

Zmodyfikuj program z przykładu 11.11 — wyjście funkcji `pobierzDane()` zdefiniuj jako obiekt klasy `Pracownik` zwracany do jej otoczenia. Wejście funkcji `wyswietlDane()` zaś określ za pomocą parametru — obiektu klasy `Pracownik`, przekazywanego do niej przez wartość.

Przykład 11.12

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    string imie;
    string nazwisko;
    void wyswietlDane() {
        cout << "Imię: " << imie << endl;
        cout << "Nazwisko: " << nazwisko << endl;
    }
};

// Prototypy funkcji zewnętrznych:
Pracownik *pobierzDane(Pracownik*);
void wyswietlDane(const Pracownik*);

// PROGRAM GŁÓWNY
int main() {
    // Utworzenie obiektu wskazywanego przez wskaźnik w_pracownik jako instancji klasy Pracownik.
    Pracownik *w_pracownik = new Pracownik();
    // Pobranie danych pracownika z klawiatury:
    w_pracownik = pobierzDane(w_pracownik);
    // Prezentacja danych pracownika na ekranie monitora:
    wyswietlDane(w_pracownik);
    // Usunięcie obiektu wskazywanego przez wskaźnik w_pracownik:
    delete w_pracownik;

    return 0;
}

// Definicje funkcji zewnętrznych:
Pracownik* pobierzDane(Pracownik *w_p) {
    cout << "Imię = "; cin >> w_p->imie;
    cout << "Nazwisko = "; cin >> w_p->nazwisko;
    return w_p;
}
void wyswietlDane(const Pracownik *w_p) {
    cout << "Imię: " << w_p->imie << endl;
    cout << "Nazwisko: " << w_p->nazwisko << endl;
}
```

Funkcjonalność programu jest identyczna z funkcjonalnością programu z przykładu 11.11. Różnice tkwią w implementacji funkcji pobierzDane() i wyswietlDane(), a co za tym idzie — w ich wywołaniach w programie głównym.

Mianowicie funkcja `pobierzDane()` przekazuje do swojego otoczenia wskaźnik do obiektu klasy `Pracownik`. Przy tym przekazanie to jest realizowane za pośrednictwem zarówno parametru/argumentu tej funkcji, jak i wartości zwracanej. Funkcja `pobierzDane()` ma zatem dwa wyjścia, określone jako wskaźniki do obiektu klasy `Pracownik`.

Z kolei funkcja `wyswietlDane()` komunikuje się ze swoim otoczeniem wyłącznie za pomocą parametru. Jest nim wskaźnik do obiektu klasy `Pracownik`, traktowanego jako stała `const`. Parametr ten odgrywa rolę wejścia funkcji.

Ćwiczenie 11.12

Zmodyfikuj program z przykładu 11.12 — zamiast obiektu klasy `Pracownik`, dla którego pamięć została zaalokowana dynamicznie na stercie, wykorzystaj obiekt `pracownik` należący do klasy `Pracownik`, dla którego pamięć została przydzielona w sposób statyczny na stosie.

11.9. Struktury w języku C++

W rozdziale 7. podręcznika zostały omówione tzw. C-struktury, które są charakterystyczne dla języka C. W języku C++ **struktura** (ang. *struct*) to klasa „specjalna”, mająca pewne charakterystyczne cechy (właściwości), które odróżniają ją od „zwykłej” klas. Nie zmienia to jednak faktu, że każda instancja struktury jest obiektem — podobnie jak instancja zwykłej klasy.

Najważniejsza różnica pomiędzy zwykłą klasą a strukturą tkwi w **domyślnym dostępie** (ang. *default access*) do ich elementów członkowskich. Mianowicie o ile w klasach elementy członkowskie (np. zmienne członkowskie) są domyślnie prywatne (ang. *private*), o tyle w strukturach wszystkie elementy członkowskie są domyślnie publiczne (ang. *public*).

Dostęp do elementów członkowskich struktury w jej otoczeniu można zrealizować z zastosowaniem **operatorów wyboru elementu członkowskiego** (ang. *member selection operators*) . lub `->`.

UWAGA

Istotną różnicą pomiędzy zwykłymi klasami a strukturami jest również domyślny typ (rodzaj) dziedziczenia (ang. *inheritance*). Mianowicie domyślnym typem dziedziczenia klas w C++ jest **dziedziczenie prywatne** (ang. *private inheritance*), a struktur — **dziedziczenie publiczne** (ang. *public inheritance*). Mechanizm dziedziczenia został omówiony w rozdziale 14.

W praktyce programistycznej struktury zazwyczaj są wykorzystywane do przechowywania niewielkiej „porcji surowych danych”, bez definiowania żadnych metod (funkcji członkowskich) i innych elementów składowych. Są to najczęściej „zwyczajne” dane, których postać przypomina dane zapisane w pojedynczym wierszu — **rekordzie** (ang. *record*) **tabeli relacyjnej bazy danych** (ang. *relational database table*). Dane tego typu często określa się angielskim terminem *plain old data* (w skrócie *POD*).

Dane przechowywane w strukturze nie powinny się zaliczać do grupy danych wrażliwych, dla których ochrona i bezpieczeństwo mają kluczowe znaczenie. W tym przypadku priorytetem jest prostota, przejrzystość oraz łatwość dostępu do zdefiniowanej struktury danych i jej elementów składowych. Dobrym przykładem danych typu POD, które nadają się do przechowania w strukturze, są dane samochodu osobowego w komisie: marka, model, rok produkcji, cena, data pierwszej rejestracji itp.

We współczesnym programowaniu obiektowym zaleca się, aby struktur w języku C++ używać do takich samych zastosowań, w jakich wykorzystuje się C-struktury w języku C. Innymi słowy, funkcjonalność struktur w C++ powinna odpowiadać funkcjonalności C-struktur w C. Tym samym wszelkie reguły stosowania C-struktur — np. definiowanie, inicjalizacja, przekazywanie C-struktur jako parametrów funkcji, wskaźniki do C-struktur — w pełni odnoszą się do struktur w języku C++.

Z idei przedstawionej powyżej wynika kolejna ważna praktyczna zasada dotycząca korzystania ze struktur w języku C++. Mianowicie pomimo że zmienne struktury w języku C++ są faktycznie obiektami, dobrym zwyczajem jest niestosowanie do nich bezpośrednio zasad wynikających z założeń (cech) programowania obiektowego, tj. mechanizmu dziedziczenia, integralnej hermetyzacji danych realizowanej za pomocą funkcji dostępowych (akcesorów) wbudowanych w strukturę, wielopostaciowości (polimorfizmu) metod itp.

UWAGA

Poszczególne założenia (cechy) programowania obiektowego zostały dokładnie omówione w rozdziale 13. i następnych.

Przykład 11.13

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

// Definicja struktury Data:
struct Data {
    // Deklaracje pól — zmiennych członkowskich:
    int dd, mm, rr;
};

// Definicja struktury Pracownik:
struct Samochod {
    // Deklaracje pól — zmiennych członkowskich:
    string marka, model;
    int rok_produkcji;
    int cena;
    Data data_rejestracji;
};
```

```
/* UWAGA
 * Elementami członkowskimi struktur Data i Pracownik są wyłącznie zmienne członkowskie.
 * Struktury te nie zawierają żadnych metod ani innych składników.
 */
```

// PROGRAM GŁÓWNY:

```
int main() {
    // Deklaracja i utworzenie struktur (obiektów) samochod1 i samochod2:
    Samochod samochod1, samochod2;
    // Wprowadzenie wartości pól struktury samochod1 z klawiatury:
    cout << "Podaj dane samochodu:" << endl;
    cout << "Marka = "; cin >> samochod1.marka;
    cout << "Model = "; cin >> samochod1.model;
    cout << "Rok produkcji = "; cin >> samochod1.rok_produkcji;
    cout << "Cena = "; cin >> samochod1.cena;
    cout << "Data pierwszej rejestracji: ";
    cout << "dzień = "; cin >> samochod1.data_rejestracji.dd;
    cout << "miesiąc = "; cin >> samochod1.data_rejestracji.mm;
    cout << "rok = "; cin >> samochod1.data_rejestracji.rr;
    cout << endl;
    // Skopiowanie zawartości wszystkich zmiennych członkowskich struktury samochod1 do struktury samochod2:
    samochod2 = samochod1;
    // Kontrolne wyświetlenie na ekranie monitora danych przechowywanych w strukturze samochod2:
    cout << "Dane samochodu przechowywane w strukturze danych: " << endl;
    cout << "Marka: " << samochod2.marka << endl;
    cout << "Model: " << samochod2.model << endl;
    cout << "Rok produkcji: " << samochod2.rok_produkcji << endl;
    cout << "Cena: " << samochod2.cena << endl;
    cout << "Data pierwszej rejestracji: ";
    cout << "dzień = " << samochod2.data_rejestracji.dd << endl;
    cout << "miesiąc = " << samochod2.data_rejestracji.mm << endl;
    cout << "rok = " << samochod2.data_rejestracji.rr << endl;

    return 0;
}
```

W programie zdefiniowano dwie struktury (typy strukturowe): Data i Samochod, których elementami członkowskimi są wyłącznie pola (ang. *fields*) — zmienne członkowskie. Struktury te nie zawierają metod (funkcji członkowskich) ani żadnych innych składników. Są więc zbudowane zgodnie z zaleceniami dotyczącymi funkcjonalności struktur w programowaniu obiektowym.

Dostęp do zawartości (pól) zmiennych struktury (obiektów) samochod1 i samochod2 uzyskano w klasyczny sposób, tj. za pomocą ich identyfikatorów, operatora wyboru elementu członkowskiego . oraz nazw pól, np. samochod2.marka.

Ćwiczenie 11.13

Zmodyfikuj program zawarty w przykładzie 11.13 — operacje wejścia/wyjścia zrealizuj za pomocą funkcji, które będą niezależne od struktur `Data` i `Samochod`, tj. zdefiniowane na zewnątrz tych struktur.

Przykład 11.14

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

// Definicja struktury Data:
struct Data {
    int dd, mm, rr;
};

// Definicja struktury Pracownik:
struct Samochod {
    string marka, model;
    int rok_produkcji;
    int cena;
    Data data_rejestracji;
};

int main() {
    // Deklaracja i utworzenie struktury (obiektu) samochod1:
    Samochod samochod1;
    // Deklaracja i inicjalizacja wskaźnika wSamochod1:
    Samochod *wSamochod1 = &samochod1;
    /* UWAGA
     * Wskaźnikowi wSamochod1 przypisano wartość początkową równą adresowi struktury (obiektu) samochod1.
     * Tym samym wskaźnik wSamochod1 wskazuje na tę strukturę.
     */
    // Wprowadzenie wartości pól struktury (obiektu) samochod1 z klawiatury:
    cout << "Podaj dane samochodu:" << endl;
    cout << "Marka = "; cin >> wSamochod1->marka;
    cout << "Model = "; cin >> wSamochod1->model;
    cout << "Rok produkcji = "; cin >> wSamochod1->rok_produkcji;
    cout << "Cena = "; cin >> wSamochod1->cena;
    cout << "Data pierwszej rejestracji: ";
    cout << "dzień = "; cin >> wSamochod1->data_rejestracji.dd;
    cout << "miesiąc = "; cin >> wSamochod1->data_rejestracji.mm;
    cout << "rok = "; cin >> wSamochod1->data_rejestracji.rr;
    cout << endl;
    /* UWAGA
     * Dostęp do zmiennych członkowskich (pól) struktury (obiektu) samochod1 został zrealizowany przy użyciu
     * wskaźnika wSamochod1 oraz operatora strzałkowego ->.
     */
}
```

```

// Utworzenie struktury typu Samochod, dla której pamięć operacyjna została zaalokowana w sposób dynamiczny:
Samochod *wSamochod2 = new Samochod;
/* UWAGA
 * Wskaźnik wSamochod2 wskazuje na strukturę typu Samochod, dla której pamięć została zaalokowana
 * dynamicznie. Struktura wskazywana przez wskaźnik wSamochod2 jest obiektem.
 */
// Skopiowanie wartości przechowywanych w polach struktury (obiektu) samochod1 do struktury — obiektu
// wskazywanego przez wskaźnik wSamochod2:
*wSamochod2 = *wSamochod1;
/* UWAGA
 * Dostęp do obu struktur (obiektów) zrealizowano za pomocą operatora dereferencji * oraz wskaźników:
 * wSamochod1 i wSamochod2.
 */
// Wyświetlenie na ekranie monitora danych przechowywanych w strukturze wskazywanej przez
// wskaźnik wSamochod2:
cout << "Dane samochodu przechowywane w strukturze danych: " << endl;
cout << "Marka: " << wSamochod2->marka << endl;
cout << "Model: " << wSamochod2->model << endl;
cout << "Rok produkcji: " << wSamochod2->rok_produkcji << endl;
cout << "Cena: " << wSamochod2->cena << endl;
cout << "Data pierwszej rejestracji: ";
cout << "dzień = " << wSamochod2->data_rejestracji.dd << endl;
cout << "miesiąc = " << wSamochod2->data_rejestracji.mm << endl;
cout << "rok = " << wSamochod2->data_rejestracji.rr << endl;

return 0;
}

```

W programie — analogicznie jak w przykładzie 11.13 — zdefiniowano dwa typy strukturowe: Data i Samochod.

Struktura (zmienna strukturalna) samochod1, należąca do typu Samochod, została zaalokowana w pamięci operacyjnej w sposób statyczny — na stosie. Struktura ta jest obiektem będącym instancją struktury Samochod. Dostęp do zmiennych członkowskich (pół) struktury samochod1 uzyskuje się za pomocą wskaźnika wSamochod1 (w którym przechowywany jest jej adres) oraz operatora strzałkowego ->.

Oprócz tego w programie utworzono strukturę, dla której pamięć została przydzielona dynamicznie na stercie. Wspomniana struktura również jest obiektem będącym instancją struktury Samochod. Dostęp do tej struktury uzyskuje się za pomocą wskaźnika wSamochod2.

Ćwiczenie 11.14

Zmodyfikuj program zawarty w przykładzie 11.14 — operacje wejścia/wyjścia zrealizuj za pomocą funkcji, które będą niezależne od struktur Data i Samochod, czyli będą zdefiniowane na zewnątrz tych struktur. Dostęp do elementów członkowskich (pół) zmiennych strukturalnych — obiektów — zrealizuj za pomocą wskaźników.



11.10. Pytania i zadania kontrolne

11.10.1. Pytania

1. Wymień najważniejsze różnice pomiędzy programowaniem proceduralnym a programowaniem obiektowym.
2. Czym jest klasa, a czym struktura? Co odróżnia klasę od struktury?
3. Opisz znaczenie specyfikatorów dostępu do elementów członkowskich klasy typu `private` i `public`.
4. Czy funkcje członkowskie klasy można definiować na zewnątrz klasy?
5. Czym się różnią zmienne członkowskie instancjne od zmiennych klasowych?
6. W jakim celu wykorzystuje się statyczne funkcje członkowskie klasy?
7. W jaki sposób można przekazać obiekty do funkcji i z funkcji? Czy na wybór sposobu przekazania obiektu do funkcji i z funkcji ma wpływ jego wielkość (rozmiar)?
8. Jakie są najważniejsze różnice pomiędzy klasami a strukturami w języku C++?

11.10.2. Zadania

1. Napisz program pozwalający obliczyć objętość, pole powierzchni całkowitej oraz długość wszystkich krawędzi prostopadłościanu. Wykorzystaj klasę zawierającą definicje zmiennych i funkcji członkowskich umożliwiających zapamiętanie parametrów prostopadłościanu i wykonanie zadanych obliczeń. Dane do programu mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora.
2. Napisz program pozwalający przeliczyć długość zmierzoną w wybranej jednostce systemu SI (np. w metrach) na długość wyrażoną w wybranych jednostkach anglosaskich (np. w milach angielskich). Wykorzystaj zmienne członkowskie i funkcje samodzielnie zdefiniowanej klasy.
3. Zdefiniuj klasę, która będzie zawierać zmienną klasową reprezentującą przelicznik mocy samochodu wyrażonej w kilowatach na konie mechaniczne oraz dwie metody statyczne typu `inline`. Pierwsza z nich powinna zapewniać przeliczenie mocy zmierzonej w kilowatach na konie mechaniczne, a druga — odwrotnie. Przeprowadź testy poprawności działania zdefiniowanych metod w programie.
4. Napisz program pozwalający zapamiętać wybrane dane samochodu osobowego: markę, model, rok produkcji, cenę, numer rejestracyjny. Zapewnij możliwość wprowadzania danych samochodu z klawiatury oraz ich modyfikowania, a ponadto zrób tak, żeby były wyświetlane na ekranie monitora. Wykorzystaj obiekt należący do zdefiniowanej samodzielnie klasy, dla którego pamięć operacyjna została zaalokowana:
 - w sposób statyczny na stosie (wariant I),
 - w sposób dynamiczny na stercie (wariant II).

Dostęp do elementów członkowskich klasy zrealizuj za pośrednictwem wskaźników.

5. Napisz program pozwalający zapamiętać w zmiennych członkowskich obiektu uczeń należącego do klasy Uczeń wybrane dane ucznia: imię, nazwisko, rok urodzenia, klasę, grupę. Pobierz dane ucznia z klawiatury, a następnie wyświetl je kontrolnie na ekranie monitora. To wszystko zrealizuj za pomocą zdefiniowanych samodzielnie funkcji zewnętrznych, niezależnych od klasy Uczeń.
6. Napisz program pozwalający obliczyć odległość pomiędzy dwoma punktami na płaszczyźnie przy założeniu, że położenie wspomnianych punktów jest określone za pomocą współrzędnych kartezjańskich. Wykorzystaj zdefiniowaną samodzielnie klasę zawierającą dwie zmienne członkowskie i jedną metodę. Każda ze zmiennych członkowskich powinna należeć do typu strukturywego, w którym zdefiniowano dwa pola, odpowiadające współrzędnym punktu, a funkcja członkowska ma za zadanie obliczenie odległości pomiędzy dwoma zadanymi punktami. Przeprowadź testy poprawności definicji klasy w przykładowym programie.

12

Tworzenie i inicjowanie obiektów

12.1. Konstruktory

Konstruktor (ang. *constructor*) jest specjalną funkcją członkowską klasy (ang. *class member function*). Konstruktor jest wywoływanym automatycznie w sytuacji, w której jest tworzony nowy obiekt należący do tej klasy — czyli obiekt będący jej instancją. Oprócz tego konstruktory są wykorzystywane do inicjalizacji (nadania wartości początkowych) zmiennym członkowskim (ang. *member variables*) tworzonego obiektu, a czasem również w celu wykonania innych działań pomocniczych.

UWAGA

W języku C++ konstruktory tak naprawdę nie tworzą nowych obiektów. To kompilator odpowiada za utworzenie obiektu, ponieważ to on rezerwuje (allokuje) pamięć operacyjną niezbędną do zapamiętania obiektu — przed faktycznym wywołaniem konstruktora. Obiekt należący do danej klasy zostanie utworzony tylko wtedy, gdy kompilator: (1) albo znajdzie definicję konstruktora pasującego do wyrażenia zawierającego wywołanie konstruktora (w sposób jawnym lub niejawnym), albo (2) sam utworzy niejawnego konstruktora domyślnego.

W obrębie (we wnętrzu) klasy może być zawarta zarówno pełna definicja konstruktora, jak i wyłącznie jego prototyp (deklaracja). W tym drugim przypadku definicję konstruktora zamieszczoną na zewnątrz klasy — podobnie jak definicję zwykłej funkcji członkowskiej (metody) — należy poprzedzić nazwą klasy oraz operatorem zakresu `...`. Klasa może mieć wiele konstrktorów — w zależności od potrzeb.

Konstruktory mają pewne szczególne właściwości, które odróżniają je od zwykłych funkcji członkowskich klasy, mianowicie:

- nazwa konstruktora jest taka sama jak nazwa klasy,
- konstruktor nie zwraca do swojego otoczenia żadnej/żadnych wartości,
- każdy konstruktor jest metodą publiczną.

Biorąc pod uwagę pierwszą z wymienionych powyżej właściwości konstruktorów, identyfikator konstruktora odpowiada jednoznacznie identyfikatorowi klasy — są one identyczne. Na przykład, jeśli `Pracownik` jest nazwą klasy, to identyfikatorem konstruktora również jest `Pracownik`. Dotyczy to wszystkich konstruktorów zdefiniowanych w klasie.

Konstruktor, jako integralna funkcja członkowska klasy, ma jednoznacznie określony interfejs, który pozwala mu komunikować się z otoczeniem. Z drugiej z wymienionych wcześniej charakterystycznych właściwości konstruktorów wynika, że dla konstruktora można określić jedynie wejście. Tym samym konstruktor nie zwraca na zewnątrz żadnej wartości. Stąd jego definicja nie zawiera typu zwracanej wartości. Konstruktor nie ma też żadnych parametrów/argumentów wyjściowych.

Wejście konstruktora stanowią parametry wejściowe, które odpowiadają wszystkim lub wybranym zmiennym członkowskim klasy. Lista parametrów konstruktora może być również pusta. Typy parametrów konstruktora powinny być zgodne z typami zmiennych członkowskich klasy reprezentowanych w jego definicji/deklaracji.

Trzecia z wymienionych wcześniej właściwości konstruktorów oznacza, że są dostępne wszędzie — w całym otoczeniu klasy, w której zostały zdefiniowane, oczywiście z uwzględnieniem zakresu widoczności tej klasy.

12.1.1. Konstruktory domyślne

W ogólności w języku C++ obowiązuje zasada, że **konstruktor domyślny** (ang. *default constructor*) jest:

- albo konstruktorem bezparametrowym (ang. *unparametrized/non parametrized constructor*),
- albo konstruktorem, w którym wszystkie parametry mają przypisane wartości domyślne.

Konstruktor niejawny

Klasa zdefiniowana przez programistę może zawierać jawną definicję konstruktora lub nie. Jeśli w danej klasie żaden konstruktor nie został zdefiniowany w sposób jawnym, kompilator automatycznie utworzy „po cichu” (niejawnie) konstruktor domyślny.

Konstruktor domyślny utworzony „po cichu” przez kompilator (a nie zdefiniowany jawnie przez programistę) często jest nazywany **konstruktorem niejawnym** (ang. *implicit*

constructor). Taki konstruktor nie ma żadnych parametrów. Jeżeli programista zdefiniuje w klasie jakikolwiek „własny” konstruktor, kompilator nie utworzy konstruktora niejawnego. Konstruktor niejawny (utworzony automatycznie przez kompilator) może być wywoływany w programie również w sposób niejawnym. W praktyce można to zrealizować na dwa sposoby:

1. Bez inicjalizacji zmiennych członkowskich klasy.
2. Z inicjalizacją zmiennych członkowskich wartościami domyślnymi.

Oba wymienione powyżej sposoby zilustrowano, odpowiednio, w przykładzie 12.1 oraz przykładzie 12.2.

Przykład 12.1

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

// Definicja struktury Data:
struct Data {
    // Deklaracje pól struktury:
    int dd, mm, rr;
};

// Definicja klasy Pracownik:
class Pracownik {
public:
    int id;
    string imie, nazwisko;
    Data data_urodzenia;
};

/* UWAGA
* Należy zwrócić uwagę na to, że wewnątrz klasy Pracownik nie jest zawarta definicja żadnego konstruktora.
* Ponadto żadna ze zmiennych członkowskich klasy Pracownik nie została zainicjowana.
*/

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy Pracownik:
    Pracownik pracownik; // wywołanie niejawnego konstruktora klasy Pracownik w sposób niejawy
/* UWAGA
* Pomimo że w klasie Pracownik nie zdefiniowano żadnego konstruktora, to po utworzeniu obiektu pracownik —
* czyli po zaalokowaniu dla niego niezbędnej pamięci operacyjnej przez kompilator —
* następuje automatyczne wywołanie konstruktora domyślnego bezparametrowego,
* czyli konstruktora niejawnego. Zmienne członkowskie obiektu nie zostały w tym przypadku zainicjowane!
*/

    // Wyświetlenie wartości zmiennych członkowskich obiektu pracownik:
```

```

cout << "Numer identyfikacyjny " << pracownik.id << endl;
cout << "Imię: " << pracownik.imie << endl;
cout << "Nazwisko: " << pracownik.nazwisko << endl;
cout << "Data urodzenia:" << endl;
cout << "dzień = " << pracownik.data_urodzenia.dd << endl;
cout << "miesiąc = " << pracownik.data_urodzenia.mm << endl;
cout << "rok = " << pracownik.data_urodzenia.rr << endl;

return 0;
}

```

W programie zdefiniowano klasę `Pracownik` zawierającą zmienne członkowskie: `id`, `imie`, `nazwisko` i `data_urodzenia`. Żadna z tych zmiennych nie została zainicjowana w obrębie definicji klasy.

Ponadto definicja klasy `Pracownik` nie zawiera definicji żadnego konstruktora. Tym samym kompilator utworzy „po cichu” (niejawnie) konstruktor domyślny bezparametryowy, czyli konstruktor niejawny. Po zaalokowaniu pamięci operacyjnej dla obiektu `pracownik` wspomniany konstruktor niejawny zostaje automatycznie wywołany również w sposób niejawnny: `Pracownik pracownik;`.

Żadna ze zmiennych członkowskich klasy nie została zainicjowana. Wartości zmiennych członkowskich wyświetcone kontrolnie na ekranie monitora są przypadkowe.

Ćwiczenie 12.1

Zmodyfikuj program zawarty w przykładzie 12.1 — każdą ze zmiennych członkowskich klasy `Pracownik` zainicjuj w obrębie jej definicji, np. `int id = 1; string imie = "imie domyślne".` Zinterpretuj uzyskane wyniki.

Przykład 12.2

```

#include <iostream>
using namespace std;

// Definicja struktury Data:
struct Data {
    int dd, mm, rr;
};

// Definicja klasy Pracownik:
class Pracownik {
public:
    int id {-1};
    string imie {"Imię domyślne"}, nazwisko {"Nazwisko domyślne"};
    Data data_urodzenia {31, 12, 1899};
};

```

```

/* UWAGA
 * Wewnątrz definicji klasy Pracownik nie jest zawarta definicja żadnego konstruktora.
 * Każda ze zmiennych członkowskich klasy Pracownik została zainicjowana ustaloną wartością domyślną.
 */

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy Pracownik:
    Pracownik pracownik; // wywołanie konstruktora niejawnego również w sposób niejawnny

    /* UWAGA
     * Po zarezerwowaniu przez kompilator pamięci operacyjnej dla zmiennej pracownik, czyli po
     * utworzeniu obiektu pracownik, następuje automatyczne wywołanie konstruktora niejawnego.
     * Zmienne członkowskie obiektu zostały zainicjowane wartościami domyślnymi określonymi w definicji klasy.
     */

    // Wyświetlenie wartości zmiennych członkowskich obiektu pracownik:
    cout << "Numer identyfikacyjny " << pracownik.id << endl;
    cout << "Imię: " << pracownik.imie << endl;
    cout << "Nazwisko: " << pracownik.nazwisko << endl;
    cout << "Data urodzenia:" << endl;
    cout << "dzień = " << pracownik.data_urodzenia.dd << endl;
    cout << "miesiąc = " << pracownik.data_urodzenia.mm << endl;
    cout << "rok = " << pracownik.data_urodzenia.rr << endl;

    return 0;
}

```

W programie zdefiniowano klasę `Pracownik` zawierającą zmienne członkowskie: `id`, `imie`, `nazwisko` i `data_urodzenia`, które w obrębie tej definicji zostały zainicjowane ustalonymi wartościami domyślnymi.

Definicja klasy `Pracownik` nie zawiera definicji żadnego konstruktora. Dlatego też kompilator utworzy w sposób automatyczny konstruktor niejawnny (czyli konstruktor domyślny bezparametryowy).

Po utworzeniu obiektu `pracownik`, czyli po zaalokowaniu pamięci operacyjnej niezbędnej do zapamiętania zmiennej `pracownik`, zostaje automatycznie wywołany konstruktor niejawny: `Pracownik pracownik;`. Wywołanie konstruktora niejawnego również w sposób niejawnny skutkuje zainicjowaniem wszystkich jego zmiennych członkowskich wartościami domyślnymi określonymi w definicji klasy.

Ćwiczenie 12.2

Zmodyfikuj program zawarty w przykładzie 12.2 — zainicjuj zmienne członkowskie klasy `Pracownik` przy użyciu innego sposobu inicjalizacji, np. inicjalizacji kopująccej.

Konstruktor domyślny zdefiniowany przez programistę

Konstruktor domyślny może być również zdefiniowany samodzielnie przez programistę. Należy jednak pamiętać o tym, że konstruktor domyślny jest z założenia konstruktorem bezparametrowym.

Jeżeli w klasie zdefiniowano co najmniej jeden konstruktor, kompilator nie utworzy konstruktora niejawnego. Innymi słowy, kompilator nie utworzy konstruktora niejawnego, jeśli programista zdefiniował w klasie chociaż jeden własny konstruktor — np. konstruktor domyślny.

Przykład 12.3

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

// Definicja struktury Data:
struct Data {
    int dd, mm, rr;
};

// Definicja klasy Pracownik:
class Pracownik {
public:
    int id;
    string imie, nazwisko;
    Data data_urodzenia;
    // Definicja konstruktora domyślnego:
    Pracownik() {
        id = -1;
        imie = "Imię domyślne", nazwisko = "Nazwisko domyślne";
        data_urodzenia = {31, 12, 1899};
    }
    /* UWAGA
     * W konstruktorze domyślnym zdefiniowanym samodzielnie przez programistę zainicjowano zadanymi wartościami
     * domyślnymi wszystkie zmienne członkowskie klasy Pracownik. Nie jest to jednak wymagane. W ogólności
     * ciało konstruktora domyślnego może być puste.
    */
};

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy Pracownik:
    Pracownik pracownik;
    /* UWAGA
     * Po utworzeniu obiektu pracownik przez kompilator następuje automatyczne wywołanie konstruktora
     * domyślnego zdefiniowanego samodzielnie przez programistę. Wszystkie zmienne członkowskie obiektu
```

```

* pracownik zostały zainicjowane wartościami domyślnymi określonymi w definicji tego konstruktora.
*/
// Wyświetlenie wartości zmiennych członkowskich obiektu pracownik:
cout << "Numer identyfikacyjny " << pracownik.id << endl;
cout << "Imię: " << pracownik.imie << endl;
cout << "Nazwisko: " << pracownik.nazwisko << endl;
cout << "Data urodzenia:" << endl;
cout << "dzień = " << pracownik.data_urodzenia.dd << endl;
cout << "miesiąc = " << pracownik.data_urodzenia.mm << endl;
cout << "rok = " << pracownik.data_urodzenia.rr << endl;

return 0;
}

```

W programie zdefiniowano klasę Pracownik ze zmiennymi członkowskimi: id, imie, nazwisko i data_urodzenia. Definicja klasy Pracownik zdefiniowanej tutaj — w odróżnieniu od klasy Pracownik z przykładu 12.1 oraz przykładu 12.2 — obejmuje definicję konstruktora domyślnego. W ciele wspomnianego konstruktora zainicjowano wszystkie zmienne członkowskie klasy zadanyymi wartościami domyślnymi.

Po zaalokowaniu przez kompilator pamięci operacyjnej dla obiektu pracownik następuje niejawne wywołanie konstruktora domyślnego zdefiniowanego samodzielnie przez programistę: Pracownik pracownik;. Skutkuje to zainicjowaniem wszystkich zmiennych członkowskich obiektu pracownik wartościami domyślnymi określonymi w definicji tego konstruktora.

Ćwiczenie 12.3

Zmodyfikuj kod źródłowy programu zawartego w przykładzie 12.3 — zainicjuj zmienne członkowskie klasy w definicji konstruktora domyślnego w inny sposób, niż zaproponowano w podanym przykładzie.

12.1.2. Konstruktory z parametrami

Konstruktory z parametrami, inaczej **konstruktory parametryczne** (ang. *parametrized constructors*), to konstruktory, które mają parametry. Konstruktory parametryczne są definiowane samodzielnie przez programistę. Parametry te odpowiadają zmiennym członkowskim klasy: wszystkim lub wybranym. Tym samym za pomocą argumentów wywołania konstruktora parametrycznego można zainicjować określonymi wartościami wszystkie lub wybrane zmienne członkowskie klasy. Wspomniane wartości inicjujące są zadane (ustalone) na poziomie funkcji wywołującej konstruktor, np. funkcji main().

Jeżeli liczba parametrów konstruktora parametrycznego nie obejmuje wszystkich zmiennych członkowskich klasy, wywołanie tego konstruktora z zadanyymi argumentami powoduje, że pozostałe zmienne członkowskie (czyli te, które nie zostały zainicjowane za pośrednictwem argumentów):

- albo nie znajdują się zainicjowane,
- albo znajdują się zainicjowane wartościami domyślnymi.

Pierwszy z wymienionych powyżej przypadków dotyczy sytuacji, w której rozpatrywanym zmiennym członkowskim — tym, których nie obejmują argumenty wywołania konstruktora — nie przypisano wartości domyślnych w obrębie definicji klasy.

Drugi przypadek obejmuje sytuację przeciwną, czyli zmiennym członkowskim zostały w definicji klasy przypisane wartości domyślne.

Rozpatrując równocześnie obydwa omówione przypadki, należy uwzględnić to, że kompilator, o czym już wspomniano, nie utworzy konstruktora domyślnego po samodzielny zdefiniowaniu konstruktora przez programistę.

Przykład 12.4

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

// Definicja struktury Data:
struct Data {
    // Deklaracje pól struktury Data:
    int dd, mm, rr;
};

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Deklaracje zmiennych członkowskich:
    int id;
    string imie, nazwisko;
    Data data_urodzenia;

    // Prototyp konstruktora parametrycznego:
    Pracownik(string, string, Data);
};

// Definicja konstruktora parametrycznego:
Pracownik::Pracownik(string pImie, string pNazwisko, Data pDataUrodzenia) {
    imie = pImie;
    nazwisko = pNazwisko;
    data_urodzenia = pDataUrodzenia;
}

// PROGRAM GŁÓWNY
int main() {
    // Niejawne wywołanie konstruktora parametrycznego z zadanimi argumentami:
    Pracownik pracownik("Jan", "Kowalski", {31, 12, 2000});
}
```

```

/* UWAGA
 * Po zaalokowaniu pamięci operacyjnej dla obiektu pracownik będącego instancją klasy Pracownik,
 * jego zmienne członkowskie imię, nazwisko i data_urodzenia zostają zainicjowane wartościami argumentów
 * podanych w wywołaniu konstruktora parametrycznego.
 * Jednocześnie zmienna członkowska id nie została zainicjowana!
 */

// Wyświetlenie wartości przechowywanych w zmiennych członkowskich obiektu pracownik:
cout << "Numer identyfikacyjny: " << pracownik.id << endl;
cout << "Imię: " << pracownik.imie << endl;
cout << "Nazwisko: " << pracownik.nazwisko << endl;
cout << "Data urodzenia:" << endl;
cout << "dzień = " << pracownik.data_urodzenia.dd << endl;
cout << "miesiąc = " << pracownik.data_urodzenia.mm << endl;
cout << "rok = " << pracownik.data_urodzenia.rr << endl;

return 0;
}

```

W programie zdefiniowano klasę `Pracownik` ze zmiennymi członkowskimi: `id`, `imie`, `nazwisko` i `data_urodzenia`. Definicja klasy `Pracownik` zdefiniowanej tutaj obejmuje definicję konstruktora parametrycznego — w odróżnieniu od klasy `Pracownik` z przykładu 12.3, w której zdefiniowano konstruktor domyślny.

Zdefiniowany konstruktor parametryczny ma dwa zadania. Po pierwsze kompilator, natrafiając w kodzie źródłowym programu na wyrażenie `Pracownik pracownik("Jan", "Kowalski", {31, 12, 2000})`; będzie poszukiwał w klasie `Pracownik` pasującej do niego definicji konstruktora parametrycznego. Po znalezieniu definicji odpowiedniego konstruktora obiekt `pracownik` zostaje utworzony, a następnie wspomniany konstruktor jest wywoływany.

Drugim zadaniem omawianego konstruktora jest inicjalizacja zmiennych członkowskich `imie`, `nazwisko` i `data_urodzenia` obiektu `pracownik` wartościami odpowiadającymi im argumentów wywołania konstruktora. Należy zwrócić uwagę, że zdefiniowany w klasie `Pracownik` (i wywołyany w programie głównym) konstruktor parametryczny nie pozwala na inicjalizację zmiennej członkowskiej `id`. Wartość tej zmiennej wyświetlana na ekranie monitora jest przypadkowa.

Konstruktor parametryczny jest wywoływany w programie głównym w sposób niejawny. Wyrażenie, w którym konstruktor ten jest wywoływany jawnie, może mieć postać: `Pracownik pracownik = Pracownik("Jan", "Kowalski", {31, 12, 2000});`.

UWAGA

Dobrą praktyką programistyczną jest to, aby podczas tworzenia obiektu inicjować jego wszystkie zmienne członkowskie. Tym samym konstruktor parametryczny powinien pozwalać na inicjalizację wszystkich zmiennych członkowskich klasy.

Ćwiczenie 12.4

Zmodyfikuj program zawarty w przykładzie 12.4 — zainicjuj zmienną członkowską `id` w definicji klasy `Pracownik` wartością 1. Zinterpretuj uzyskany wynik. W kolejnym teście zmodyfikuj definicję (i prototyp) konstruktora parametrycznego w taki sposób, aby jego wywołanie pozwalało na inicjalizację wszystkich zmiennych członkowskich klasy `Pracownik`. Wywołaj ten konstruktor, a wartość argumentu odpowiadającego zmiennej członkowskiej `id` ustal na 10. Zinterpretuj uzyskany wynik.

12.1.3. Przeciążanie konstruktorów

Konstruktory są funkcjami specjalnymi, które można przeciążać tak samo jak inne funkcje. Dlatego też w definicji klasy mogą być zawarte definicje wielu konstruktorów, które różnią się od siebie liczbą i/lub typem parametrów.

Przeciążanie konstruktorów (ang. *overloading constructors*) jest realizowane w praktyce według tych samych zasad, co przeciążanie zwykłych funkcji.

UWAGA

Przeciążanie funkcji (ang. *function overloading*) zostało szczegółowo omówione w podrozdziale 8.5.

Dla przypomnienia — przeciążanie funkcji polega na tym, że w tym samym zakresie (ang. *scope*) definiowane są funkcje, które mają taką samą nazwę, ale różnią się od siebie liczbą i/lub typem parametrów. W przypadku przeciążania konstruktorów wspomniany zakres jest określony przez **zakres klasy** (ang. *class scope*), który obejmuje jej ciało (ang. *class body*).

Kompilator, natrafiwszy w kodzie źródłowym programu na wyrażenie zawierające wywołanie konstruktora, automatycznie wybiera — na podstawie wyniku analizy liczby i typów argumentów tego wywołania — jego odpowiednią wersję zdefiniowaną w klasie.

Przykład 12.5

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

// Definicja struktury Data:
struct Data {
    int dd, mm, rr;
};

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Deklaracje zmiennych członkowskich:
```

```

int id;
string imie, nazwisko;
// Deklaracja zmiennej członkowskiej data_urodzenia połączona z inicjalizacją jej elementów składowych
// wartościami domyślnymi:
Data data_urodzenia {31, 12, 1899};

// Prototyp konstruktora domyślnego:
Pracownik();
// Prototypy konstruktorów parametrycznych:
Pracownik(int, string, string);
Pracownik(int, string, string, Data);
// Prototyp publicznej metody dostępowej:
void wyswietlDane();

};

// Definicja konstruktora domyślnego:
Pracownik::Pracownik() {
    id = -1;
    imie = "Imię domyślne";
    nazwisko = "Nazwisko domyślne";
    data_urodzenia = {1, 1, 1900};
}

// Definicje konstruktorów parametrycznych:
Pracownik::Pracownik(int pId, string pImie, string pNazwisko) {
    id = pId;
    imie = pImie;
    nazwisko = pNazwisko;
}

Pracownik::Pracownik(int pId, string pImie, string pNazwisko,
                     Data pDataUrodzenia) {
    id = pId;
    imie = pImie;
    nazwisko = pNazwisko;
    data_urodzenia = pDataUrodzenia;
}

// Definicja metody wyswietlDane()
void Pracownik::wyswietlDane() {
    cout << "Numer identyfikacyjny: " << id << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
    cout << "Data urodzenia:" << endl;
    cout << "dzień = " << data_urodzenia.dd << endl;
    cout << "miesiąc = " << data_urodzenia.mm << endl;
    cout << "rok = " << data_urodzenia.rr << endl;
}

```

```

// PROGRAM GŁÓWNY
int main() {
    // Utworzenie obiektu pracownik1 będącego instancją klasy Pracownik:
    Pracownik pracownik1;
    /* UWAGA
     * Po zaalokowaniu pamięci dla obiektu pracownik1 został wywołany w sposób niejawnny konstruktor domyślny
     * zdefiniowany przez programistę. Zmienne członkowskie tego obiektu zostały zainicjowane wartościami
     * domyślnymi określonymi w definicji tego konstruktora.
     */
    // Wyświetlenie wartości przechowywanych w zmiennych członkowskich obiektu pracownik1:
    pracownik1.wyswietlDane();
    cout << endl;

    // Utworzenie obiektu pracownik2 jako instancji klasy Pracownik:
    Pracownik pracownik2(1, "Jan", "Kowalski");
    /* UWAGA
     * Po zarezerwowaniu pamięci operacyjnej dla obiektu pracownik2 został wywołany w sposób niejawnny
     * konstruktor parametryczny zdefiniowany przez programistę. Zmienne członkowskie id, imie i nazwisko obiektu
     * zostały zainicjowane zadanymi wartościami argumentów wywołania tego konstruktora.
     * Zmienna członkowska data_urodzenia została zainicjowana wartościami domyślnymi jej składowych, które
     * zostały ustalone w obrębie definicji klasy.
     */
    pracownik2.wyswietlDane();
    cout << endl;

    // Utworzenie obiektu pracownik3 jako instancji klasy Pracownik:
    Pracownik pracownik3(10, "Adam", "Nowak", {10, 10, 2000});
    /* UWAGA
     * Konstruktor parametryczny został wywołany w sposób niejawnym. Wszystkie zmienne członkowskie obiektu
     * pracownik3 zostały zainicjowane zadanymi wartościami argumentów wywołania tego konstruktora.
     */
    pracownik3.wyswietlDane();

    return 0;
}

```

W klasie `Pracownik` zdefiniowano trzy konstruktory. Pierwszy z nich jest konstruktorem domyślnym, ponieważ nie ma żadnych parametrów. Dwa pozostałe to konstruktory parametryczne. Pierwszy z nich pozwala inicjować wybrane zmienne członkowskie obiektu klasy `Pracownik`, a drugi — wybrane.

Wszystkie z wymienionych konstruktorów zostały zdefiniowane w tym samym zakresie — zakresie klasy `Pracownik`. Konstruktory te mają oczywiście tę samą nazwę, ale jednocześnie różnią się od siebie liczbą i typem parametrów. Mamy więc tutaj do czynienia z mechanizmem przeciążania konstruktorów.

Konstruktory są wywoływanie w programie głównym. Kompilator każdorazowo sprawdza liczbę i typy argumentów tych wywołań i dopasowuje automatycznie odpowiedni — zdefiniowany w klasie — konstruktor w celu jego wywołania.

Ćwiczenie 12.5

Zmodyfikuj program z przykładu 12.5 — przydziel wartości domyślne wszystkim zmiennym członkowskim klasy `Pracownik` w obrębie jej definicji. Wykonaj testy zmodyfikowanego programu. Zinterpretuj uzyskane rezultaty.

12.2. Inicjalizacja obiektów

Inicjalizacja obiektu (ang. *object initialization*) polega na nadaniu wartości początkowych jego zmiennym członkowskim. Różne obiekty były inicjowane w wielu przykładowych programach zaprezentowanych wcześniej w tym rozdziale. Jednakże zwracaliśmy tam uwagę na aspekty praktyczne stosowania konstruktorów, nie skupiając się na zagadnieniu ich inicjalizacji. W niniejszym podrozdziale tematyka inicjalizacji obiektów została usystematyzowana.

W przypadku ogólnym obiekty można inicjować wartościami domyślnymi lub wartościami zadanymi.

12.2.1. Inicjalizacja obiektów wartościami domyślnymi

Obiekty można inicjować wartościami domyślnymi na trzy sposoby:

- z użyciem **inicjalizatorów wewnętrzklasowych** (ang. *in-class initializers*) bezpośrednio w definicji (wewnętrz) klasy,
- za pomocą zdefiniowanego konstruktora domyślnego (ang. *default constructor*) zawierającego w swoim ciele inicjalizatory zmiennych członkowskich,
- z zastosowaniem konstruktora parametrycznego (ang. *parametrized constructor*) z parametrami domyślnymi — jeśli nie zdefiniowano konstruktora domyślnego.

Należy zaznaczyć, że inicjalizacja z użyciem konstruktora (domyślnego i parametrycznego) ma wyższy priorytet niż inicjalizacja za pomocą inicjalizatorów wewnętrzklasowych.

Przykład 12.6

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Definicje zmiennych członkowskich:
    int id = -1; // inicjalizator wewnętrzklasowy
```

```

string imie = "Imię domyślne"; // inicjalizator wewnętrzklasowy
string nazwisko = "Nazwisko domyślne"; // inicjalizator wewnętrzklasowy
// Prototyp konstruktora domyślnego:
Pracownik();
// Prototyp konstruktora parametrycznego:
Pracownik(int, string, string);
// Prototyp funkcji członkowskiej:
void wyswietlDane();

};

// Definicje konstruktorów:
Pracownik::Pracownik() {
    cout << "Komunikat kontrolny: wywołanie konstruktora domyślnego ..."
        << endl;
}

Pracownik::Pracownik(int pId, string pImie, string pNazwisko) {
    id = pId;
    imie = pImie;
    nazwisko = pNazwisko;
}

// Definicja funkcji członkowskiej wyswietlDane():
void Pracownik::wyswietlDane() {
    cout << "Numer identyfikacyjny: " << id << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
}

int main() {
    // Utworzenie obiektu pracownik:
    Pracownik pracownik;
    /* Uwaga
     * Konstruktor domyślny został wywołany w sposób niejawny po przydzieleniu pamięci operacyjnej dla obiektu
     * pracownik. Zmienne członkowskie obiektu pracownik zostały zainicjowane wartościami domyślnymi
     * określonymi w definicji klasy, czyli za pomocą inicjalizatorów wewnętrzklasowych.
     */
    pracownik.wyswietlDane();

    return 0;
}

```

W programie zdefiniowano klasę `Pracownik`, w której zawarto wewnętrzklasowe inicjalizatory wszystkich zmiennych członkowskich.

Klasa `Pracownik` ma dwa konstruktory — domyślny i parametryczny. Konstruktor domyślny zdefiniowany przez programistę nie zawiera inicjalizatorów zmiennych członkowskich.

Konstruktor domyślny został wywołany w programie w sposób niejawny: Pracownik pracownik;. Zmienne członkowskie id, imie i nazwisko zostały zainicjowane za pomocą inicjalizatorów wewnętrzklasowych zawartych w definicji klasy Pracownik.

Ćwiczenie 12.6

Zmodyfikuj program zawarty w przykładzie 12.6 — w klasie Pracownik zdefiniuj dodatkową zmienną członkowską o nazwie data_zatrudnienia należącą do typu strukturywego Data. Struktura Data powinna zawierać trzy pola umożliwiające zapamiętanie zadanej daty — dnia, miesiąca i roku. Zainicjuj obiekt pracownik ustalonymi wartościami domyślnymi z użyciem inicjalizatorów wewnętrzklasowych.

Przykład 12.7

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Definicje zmiennych członkowskich:
    int id = -1; // wewnętrzklasowy inicjalizator zmiennej członkowskiej
    string imie = "Imię domyślne"; // wewnętrzklasowy inicjalizator zmiennej członkowskiej
    string nazwisko = "Nazwisko domyślne"; // wewnętrzklasowy inicjalizator zmiennej
                                         // członkowskiej

    // Prototyp konstruktora domyślnego:
    Pracownik();
    // Prototyp konstruktora parametrycznego:
    Pracownik(int, string, string);
    // Prototyp funkcji członkowskiej:
    void wyswietlDane();

};

// Definicje konstruktorów:
Pracownik::Pracownik() {
    id = 0; // inicjalizacja zmiennej członkowskiej wartością domyślną
    imie = "I m i e"; // inicjalizacja zmiennej członkowskiej wartością domyślną
    nazwisko = "N a z w i s k o"; // inicjalizacja zmiennej członkowskiej wartością domyślną

    cout << "Komunikat kontrolny: wywołanie konstruktora domyślnego ..."
         << endl;
}

Pracownik::Pracownik(int pId, string pImie, string pNazwisko) {
    id = pId;
    imie = pImie;
    nazwisko = pNazwisko;
}
```

```

// Definicja funkcji członkowskiej:
void Pracownik::wyswietlDane() {
    cout << "Numer identyfikacyjny: " << id << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
}

int main() {
    // Utworzenie obiektu pracownik1:
    Pracownik pracownik1;
    /* Uwaga
     * Konstruktor domyślny został wywołany w sposób niejawny. Zmienne członkowskie obiektu pracownik1
     * zostały zainicjowane wartościami domyślnymi ustalonymi w definicji tego konstruktora.
     */
    pracownik1.wyswietlDane();

    // Utworzenie obiektu pracownik2:
    Pracownik pracownik2 = Pracownik();
    /* Uwaga
     * Konstruktor domyślny został wywołany w sposób jawnym. Zmienne członkowskie obiektu pracownik2
     * zostały zainicjowane wartościami domyślnymi określonymi w definicji tego konstruktora.
     */
    pracownik2.wyswietlDane();

    return 0;
}

```

W programie zdefiniowano klasę `Pracownik`, w której wszystkie zmienne członkowskie zainicjowano wartościami domyślnymi z wykorzystaniem inicjalizatorów wewnętrzklasowych.

Należy zwrócić uwagę, że wartości domyślne zmiennych członkowskich klasy zostały również określone w ciele konstruktora domyślnego. Jednakże wartości te są odmienne od wartości domyślnych ustalonych za pomocą inicjalizatorów wewnętrzklasowych.

Obiekt `pracownik1` został utworzony przez niejawne wywołanie konstruktora domyślnego, a obiekt `pracownik2` — jawnie wywołanie tego samego konstruktora. W obu przypadkach zmienne członkowskie `id`, `imie` i `nazwisko` zostały zainicjowane z zastosowaniem inicjalizatorów zawartych w definicji (ciele) konstruktora domyślnego, który ma wyższy priorytet od inicjalizatorów wewnętrzklasowych.

Ćwiczenie 12.7

Zmodyfikuj program zawarty w przykładzie 12.7 — w klasie `Pracownik` zdefiniuj dodatkową zmienną członkowską o nazwie `data_zatrudnienia` należącą do typu strukturowego `Data`. Struktura `Data` powinna zawierać trzy pola umożliwiające zapamiętanie zadanej daty — dnia, miesiąca i roku. Zainicjuj obiekt `pracownik` należący do klasy `Pracownik` wartościami domyślnymi określonymi w definicji konstruktora domyślnego.

Przykład 12.8

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Deklaracje zmiennych członkowskich:
    int id = -1;
    string imie = "Imię domyślne";
    string nazwisko = "Nazwisko domyślne";
    // Prototyp konstruktora parametrycznego:
    Pracownik(int pId = 0, string pImie = "I m i e",
              string pNazwisko = "N a z w i s k o");
        // dla wszystkich parametrów powyżej określone zostały wartości domyślne

    // Prototyp funkcji członkowskiej:
    void wyswietlDane();
};

/* UWAGA
* W klasie Pracownik nie zdefiniowano konstruktora domyślnego.
*/

// Definicja konstruktora parametrycznego:
Pracownik::Pracownik(int pId, string pImie, string pNazwisko) {
    id = pId;
    imie = pImie;
    nazwisko = pNazwisko;
}

// Definicja funkcji członkowskiej:
void Pracownik::wyswietlDane() {
    cout << "Numer identyfikacyjny: " << id << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
}

int main() {
    Pracownik pracownik = Pracownik();
    /* Uwaga
* Konstruktor parametryczny został wywołany w sposób jawnym — ale bez żadnych argumentów. Tym samym
* zmienne członkowskie obiektu pracownik zostały zainicjowane wartościami domyślnymi jego parametrów.
*/
    pracownik.wyswietlDane();

    return 0;
}
```

W programie zdefiniowano klasę `Pracownik`, w której wszystkie zmienne członkowskie zainicjowano wartościami domyślnymi z użyciem inicjalizatorów wewnętrznych. Wartości domyślne zmiennych członkowskich zostały również ustalone dla parametrów zdefiniowanego konstruktora parametrycznego.

Obiekt `pracownik` został utworzony przez jawne wywołanie konstruktora parametrycznego. Zmienne członkowskie `id`, `imie` i `nazwisko` zostały zainicjowane za pomocą wartości domyślnych stanowiących parametry domyślne konstruktora, które mają wyższy priorytet od inicjalizatorów wewnętrznych.

Ćwiczenie 12.8

Zmodyfikuj program zawarty w przykładzie 12.8 — w klasie `Pracownik` zdefiniuj dodatkową zmienną członkowską o nazwie `data_zatrudnienia` należącą do typu strukturywego `Data`. Struktura `Data` powinna zawierać trzy pola umożliwiające zapamiętanie zadanej daty — dnia, miesiąca i roku. Zainicjuj obiekt `pracownik` należący do klasy `Pracownik` wartościami domyślnymi określonymi jako parametry domyślne zdefiniowanego samodzielnie konstruktora parametrycznego.

12.2.2. Inicjalizacja obiektów wartościami zadanimi

W ogólności inicjalizacja obiektu wartościami zadanimi może być zrealizowana:

- za pomocą tzw. listy inicjalizacyjnej,
- z zastosowaniem inicjalizacji konstruktorowej.

Lista inicjalizacyjna

Ogólna postać **listy inicjalizacyjnej** (ang. *initializer list, initialization list*) jest następująca:

`{wyrażenie_1, wyrażenie_2, ...}`

gdzie `wyrażenie_1`, `wyrażenie_2` itd. stanowią wyrażenia określające zadane wartości początkowe zmiennych członkowskich obiektu.

UWAGA

Wykorzystanie listy inicjalizacyjnej jest możliwe począwszy od standardu C++11.

Ogólna postać wyrażenia pozwalającego na inicjalizację zmiennych członkowskich obiektu przy użyciu listy inicjalizacyjnej jest następująca:

`nazwa_klasy nazwa_obiektu = {wyrażenie_1, wyrażenie_2, ...};`

lub

`nazwa_klasy nazwa_obiektu {wyrażenie_1, wyrażenie_2, ...};`

gdzie:

- nazwa_klasy jest identyfikatorem klasy,
- nazwa_obiektu jest identyfikatorem obiektu (zmiennej obiektowej),
- {wyrażenie_1, wyrażenie_2, ...} to lista inicjalizacyjna.

Pierwsza z zaprezentowanych powyżej postaci stanowi **inicjalizację kopiującą** (ang. *copy initialization*), a druga — **inicjalizację bezpośrednią** (ang. *direct initialization*).



UWAGA

Sposoby inicjalizacji zmiennych należących do typów podstawowych zostały omówione szczegółowo w podrozdziale 3.1 podręcznika.

Wymienione sposoby inicjalizacji zmiennych członkowskich obiektów można opisać również z wykorzystaniem pojęcia listy inicjalizacyjnej. Stąd pierwsza z przedstawionych powyżej postaci jest **nazywana inicjalizacją listową kopiującą** (ang. *copy-list-initialization*), a druga — **inicjalizacją listową bezpośrednią** (ang. *direct-list-initialization*). Inicjalizacja bezpośrednia jest w tym przypadku **inicjalizacją jednolitą** (ang. *uniform initialization*), która jest zalecanym sposobem inicjalizacji obiektów zadanymi wartościami.



UWAGA

Inicjalizacja jednolita jest czasem nazywana **inicjalizacją klamrową** (ang. *brace initialization*).

Inicjalizacja agregacyjna

W najprostszym przypadku, jeśli wszystkie zmienne członkowskie klasy są publiczne i zarazem w klasie nie zdefiniowano żadnego konstruktora, w celu ich inicjalizacji można zastosować tzw. **inicjalizację agregacyjną** (ang. *aggregate initialization*).

Przykład 12.9

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Deklaracje zmiennych członkowskich:
    int id;
    string imie;
    string nazwisko;
```

```

// Prototyp funkcji członkowskiej:
void wyswietlDane();

};

/* UWAGA
* Klasa Pracownik nie zawiera żadnego konstruktora zdefiniowanego przez programistę.
*/

// Definicja funkcji członkowskiej wyswietlDane():
void Pracownik::wyswietlDane() {
    cout << "Numer identyfikacyjny: " << id << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
}

int main() {
    // Utworzenie i inicjalizacja agregacyjna obiektu pracownik1:
    Pracownik pracownik1 = {1, "Jan", "Kowalski"};
    /* Uwaga
     * Wszystkie zmienne członkowskie obiektu pracownik1 zainicjowano przy wykorzystaniu inicjalizacji kopiącej
     * z listą inicjalizacyjną.
     */
    pracownik1.wyswietlDane();

    // Utworzenie i inicjalizacja agregacyjna obiektu pracownik2:
    Pracownik pracownik2 {2, "Adam", "Nowak"};
    /* Uwaga
     * Zmienne członkowskie obiektu pracownik1 zainicjowano z zastosowaniem inicjalizacji bezpośredniej
     * z listą inicjalizacyjną.
     */
    pracownik2.wyswietlDane();

    return 0;
}

```

W programie zainicjowano dwa obiekty należące do klasy `Pracownik`. Zastosowano inicjalizację agregacyjną, ponieważ wszystkie zmienne członkowskie klasy `Pracownik` są publiczne oraz nie zawiera ona definicji żadnego konstruktora. Należy zauważyć, że inicjalizacja bezpośrednią z użyciem listy inicjalizacyjnej jest inicjalizacją jednolitą.

Ćwiczenie 12.9

Zmodyfikuj program zawarty w przykładzie 12.9 — w klasie `Pracownik` zdefiniuj dodatkową zmienną członkowską o nazwie `data_zatrudnienia` należącą do typu strukturowego `Data`. Struktura `Data` powinna zawierać trzy pola umożliwiające zapamiętanie zadanej daty — dnia, miesiąca i roku. Zainicjuj obiekt `pracownik` należący do klasy `Pracownik` wybranymi przez siebie wartościami z wykorzystaniem inicjalizacji agregacyjnej.

Inicjalizacja obiektów za pomocą zdefiniowanego konstruktora

Jeśli zmienne członkowskie klasy są prywatne albo programista zdefiniował w klasie własne konstruktory, inicjalizacja agregacyjna nie jest możliwa. W takiej sytuacji w celu inicjalizacji obiektu należy zastosować konstruktor zdefiniowany samodzielnie.

W ogólności wywołanie konstruktora może być zrealizowane w sposób niejawny lub jawny — co zostało już zaprezentowane we wcześniejszych przykładach w tym rozdziale.

Przykład 12.10

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

// Definicja klasy Pracownik:
class Pracownik {
    // Deklaracje prywatnych zmiennych członkowskich:
    int id;
    string imie, nazwisko;
public:
    // Prototyp konstruktora domyślnego:
    Pracownik();
    // Prototyp konstruktora parametrycznego:
    Pracownik(int, string, string);
    // Prototyp funkcji członkowskiej:
    void wyswietlDane();
};

// Definicje konstruktorów:
Pracownik::Pracownik() {
    id = -1;
    imie = "Imię domyślne";
    nazwisko = "Nazwisko domyślne";
}
Pracownik::Pracownik(int pId, string pImie, string pNazwisko) {
    id = pId;
    imie = pImie;
    nazwisko = pNazwisko;

    cout << "Komunikat kontrolny: wywołano konstruktor parametryczny ..."
        << endl;
}

// Definicja funkcji członkowskiej wyswietlDane():
void Pracownik::wyswietlDane() {
    cout << "Numer identyfikacyjny: " << id << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
}
```

```

int main() {
    // Utworzenie obiektu pracownik1 jako instancji klasy Pracownik:
    Pracownik pracownik1 = {1, "Jan", "Kowalski"}; // inicjalizacja listowa kopującą
    pracownik1.wyswietlDane();
    cout << endl;

    // Utworzenie obiektu pracownik2 jako instancji klasy Pracownik:
    Pracownik pracownik2 {2, "Adam", "Nowak"}; // inicjalizacja jednolita (klamrowa)
    pracownik2.wyswietlDane();
    cout << endl;

    return 0;
}

```

W programie zostały utworzone dwa obiekty należące do klasy Pracownik: pracownik1 i pracownik2. Oba zostały utworzone i zainicjowane przez niejawne wywołanie konstruktora parametrycznego.

W celu inicjalizacji obiektu pracownik1 zastosowano inicjalizację listową kopującą: Pracownik pracownik1 = {1, "Jan", "Kowalski"};.

Z kolei obiekt pracownik2 został zainicjowany przy użyciu inicjalizacji listowej bezpośredniej, czyli inicjalizacji jednolitej (klamrowej): Pracownik pracownik2 {2, "Adam", "Nowak"};.

Zalecanym sposobem inicjalizacji jest w tym przypadku inicjalizacja jednolita.

Ćwiczenie 12.10

Zmodyfikuj program z przykładu 12.10 — utwórz i zainicjuj obiekty pracownik1 i pracownik2 przez jawne wywołanie konstruktora parametrycznego.

Inicjalizacja konstruktorowa

Ogólna postać wyrażenia pozwalającego na inicjalizację zmiennych członkowskich obiektu z wykorzystaniem **inicjalizacji konstruktorowej** (ang. *constructor initialization*) jest następująca:

```
nazwa_klasy nazwa_obiektu (wyrażenie_1, wyrażenie_2, ...);
```

gdzie:

- nazwa_klasy jest identyfikatorem klasy,
- nazwa_obiektu jest identyfikatorem obiektu (zmiennej obiektowej),
- wyrażenie_1, wyrażenie_2, ... to wyrażenia określające zadane wartości dla kolejnych parametrów konstruktora.

Przykład 12.11

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

// Definicja klasy Pracownik:
class Pracownik {
    // Deklaracje prywatnych zmiennych członkowskich:
    int id;
    string imie, nazwisko;
public:
    // Prototypy konstruktorów:
    Pracownik();
    Pracownik(int, string, string);
    // Prototyp funkcji członkowskiej:
    void wyswietlDane();
};

// Definicje konstruktorów:
Pracownik::Pracownik() {
    id = -1;
    imie = "Imię domyślne";
    nazwisko = "Nazwisko domyślne";
}
Pracownik::Pracownik(int pId, string pImie, string pNazwisko) {
    id = pId;
    imie = pImie;
    nazwisko = pNazwisko;
}

// Definicja funkcji członkowskiej wyswietlDane():
void Pracownik::wyswietlDane() {
    cout << "Numer identyfikacyjny: " << id << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
}

int main() {
    // Utworzenie obiektu pracownik1 jako instancji klasy Pracownik:
    Pracownik pracownik (1, "Jan", "Kowalski"); // inicjalizacja konstruktorowa
    pracownik.wyswietlDane();

    return 0;
}
```

Klasa Pracownik zawiera m.in. prywatne zmienne członkowskie `id`, `imie` i `nazwisko` oraz definicje konstruktorów, domyślnego i parametrycznego.

Obiekt `pracownik` został zainicjowany zadanymi wartościami przy użyciu inicjalizacji konstruktorowej.

Ćwiczenie 12.11

Zmodyfikuj program z przykładu 12.11 — utwórz i zainicjuj obiekt `pracownik` przez jawne wywołanie konstruktora parametrycznego. Zmienne członkowskie obiektu `pracownik` zainicjuj z wykorzystaniem inicjalizacji konstruktorowej.

Lista inicjalizacyjna zmiennych członkowskich

W niektórych sytuacjach nie można przypisać zadanych wartości elementom członkowskim klasy w treści konstruktora, np. w przypadku gdy są one zadeklarowane jako stałe `const`. Problem ten można rozwiązać z zastosowaniem listy inicjalizacyjnej zmiennych członkowskich.

Lista inicjalizacyjna zmiennych członkowskich klasy (ang. *class member initializer list, class member initialization list*) wchodzi w skład nagłówka konstruktora:

`nazwa_konstruktora(lista_parametrów) : lista_inicjalizatorów;`

gdzie:

- nazwa_konstruktora jest nazwą (identyfikator) klasy,
- lista_parametrów to lista parametrów formalnych konstruktora odpowiadających zmiennym członkowskim klasy,
- lista_inicjalizatorów to lista inicjalizatorów wybranych/wszystkich zmiennych członkowskich, z których każdy składa się z nazwy zmiennej członkowskiej wraz z przypisaną wartością początkową.

Przykład 12.12

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

// Definicja klasy Pracownik:
class Pracownik {
    const int id = 0; // inicjalizacja stałej const wartością domyślną
    string imie, nazwisko;
public:
    // Prototypy konstruktorów:
    Pracownik();
    Pracownik(int, string, string);
    // Prototyp funkcji członkowskiej:
    void wyswietlDane();
};

// Definicja konstruktora domyślnego z listą inicjalizatorów zmiennych członkowskich:
Pracownik::Pracownik():
    id {-1},
    imie {"Imię domyślne"},
    nazwisko {"Nazwisko domyślne"} {
```

```

    cout << "Komunikat kontrolny: wywołano konstruktor domyślny ..." << endl;
}

// Definicja konstruktora parametrycznego z listą inicjalizatorów zmiennych członkowskich:
Pracownik::Pracownik(int pId, string pImie, string pNazwisko):
    id {pId},
    imie {pImie},
    nazwisko {pNazwisko} {
    cout << "Komunikat kontrolny: wywołano konstruktor parametryczny ..."
        << endl;
}

// Definicja funkcji członkowskiej:
void Pracownik::wyswietlDane() {
    cout << "Numer identyfikacyjny: " << id << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
}

int main() {
    // Utworzenie obiektu pracownik1 jako instancji klasy Pracownik:
    Pracownik pracownik1;
    /* UWAGA
     * Konstruktor domyślny zdefiniowany przez programistę został wywołany w sposób niejawny.
     * Konstruktor ten można oczywiście wywołać w sposób jawny:
     *   Pracownik pracownik1 = Pracownik();
     * Obiekt pracownik1 został utworzony i zainicjowany wartościami domyślnymi.
     */
    // Wyświetlenie wartości przechowywanych w zmiennych członkowskich obiektu pracownik1:
    pracownik1.wyswietlDane();
    cout << endl;

    // Utworzenie obiektu pracownik2 jako instancji klasy Pracownik:
    Pracownik pracownik2(1, "Jan", "Kowalski");
    /* UWAGA
     * Konstruktor parametryczny zdefiniowany przez programistę został wywołany w sposób niejawny.
     * Obiekt pracownik2 został utworzony i zainicjowany wartościami argumentów podanych w wywołaniu
     * tego konstruktora.
     * Konstruktor parametryczny można oczywiście wywołać w sposób jawny:
     *   Pracownik pracownik2 = Pracownik(1, "Jan", "Kowalski");
     */
    // Wyświetlenie wartości przechowywanych w zmiennych członkowskich obiektu pracownik2:
    pracownik2.wyswietlDane();

    return 0;
}

```

W programie zdefiniowano klasę `Pracownik`. W klasie tej zadeklarowano i zainicjowano wartością domyślną 0 zmienną członkowską o nazwie `id` jako stałą `const`. Zmienne członkowskie `imie` i `nazwisko` nie zostały zainicjowane przy użyciu inicjatorów wewnętrznych. W klasie `Pracownik` zdefiniowano ponadto dwa konstruktory (domyślny i parametryczny) z listami inicjalizatorów wszystkich zmiennych członkowskich.

Obiekt `pracownik1` został utworzony i zainicjowany wartościami domyślnymi przy wykorzystaniu konstruktora domyślnego z listą inicjalizatorów wszystkich zmiennych członkowskich. Inicjalizacja zmiennych członkowskich (tutaj zmiennej `id`) zrealizowana na poziomie konstruktora ma wyższy priorytet od inicjalizacji bezpośrednio w definicji klasy za pomocą inicjalizatorów wewnętrznych.

Obiekt `pracownik2` utworzono i zainicjowano zadanymi wartościami z zastosowaniem konstruktora parametrycznego, który również zawiera listę inicjalizatorów zmiennych członkowskich.

Ćwiczenie 12.12

Zmodyfikuj program z przykładu 12.12 — zmodyfikuj definicje konstruktorów domyślnego i parametrycznego w taki sposób, aby nadać, odpowiednio, wartość domyślną i wartość zadaną zmiennej członkowskiej `id` typu `const` w ciałach tych konstraktorów, a nie na liście inicjalizatorów zmiennych członkowskich. Czy komplikacja programu zakończy się sukcesem? Uzasadnij, dlaczego rezultaty komplikacji są takie, a nie inne.

12.3. Konstruktor kopiący

Konstruktor kopiący (ang. *copy constructor*) to konstruktor przeciążony (ang. *overloaded constructor*), którego zadaniem jest utworzenie i inicjalizacja nowego obiektu należącego do określonej klasy na podstawie innego obiektu będącego instancją tej klasy — obiektu wzorcowego. Nowy obiekt jest identyczny z obiektem wzorcowym (oryginałem) i jednocześnie jest od niego całkowicie niezależny.

Ogólna postać prototypu (deklaracji) konstruktora kopiącego jest następująca:

`nazwa_klasy (const nazwa_klasy &wzorzec);`

gdzie `nazwa_klasy` jest identyfikatorem klasy, której instancją jest nowy obiekt, a `wzorzec` to obiekt wzorcowy, na podstawie którego zostanie utworzony nowy obiekt.

UWAGA

Jeśli programista nie zdefiniuje własnego konstruktora kopiącego, kompilator automatycznie utworzy domyślny konstruktor kopiący.

Konstruktor kopiujący może być wywołany w sposób jawnny lub niejawnny. Pierwszy przypadek dotyczy sytuacji, w której nowy obiekt jest tworzony na podstawie wywołania konstruktora z argumentem referencyjnym do obiektu wzorcowego traktowanego jako stała const. Niejawne wywołanie konstruktora kopiującego zaś jest stosowane, jeśli:

- nowy obiekt jest tworzony na podstawie innego obiektu przy użyciu operatora =,
- istniejący obiekt jest parametrem wejściowym dowolnej funkcji przekazywanym do niej przez wartość,
- istniejący obiekt jest zwracany do otoczenia (np. do programu głównego) przez inną (dowolną) funkcję.

Przykład 12.13

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

// Definicja klasy Prostokat:
class Prostokat {
private:
    double bok1;
    double bok2;
public:
    // Prototypy konstruktorów:
    Prostokat(); // konstruktor domyślny
    Prostokat(double, double); // konstruktor z parametrami
    // Prototyp konstruktora kopiującego:
    Prostokat(const Prostokat &); // konstruktor kopiujący
    // Prototypy metod publicznych:
    void pobierzBoki(double&, double&);
    void ustawBoki(double, double);
    double pole();
    double obwod();
};

// Definicje „zwykłych” konstruktorów:
Prostokat::Prostokat() {
    cout << "Nastąpiło wywołanie konstruktora domyślnego ..." << endl;
}
Prostokat::Prostokat(double pBok1, double pBok2) {
    cout << "Nastąpiło wywołanie konstruktora parametrycznego ..." << endl;
    bok1 = pBok1;
    bok2 = pBok2;
}
// Definicja konstruktora kopiuującego:
Prostokat::Prostokat(const Prostokat& wzorzec) {
```

```

cout << "Nastąpiło wywołanie konstruktora kopiącego ..." << endl;
bok1 = wzorzec.bok1;
bok2 = wzorzec.bok2;
}

// Definicje metod publicznych należących do klasy Prostokat:

void Prostokat::pobierzBoki(double &pBok1, double &pBok2) {
    pBok1 = bok1;
    pBok2 = bok2;
}
void Prostokat::ustawBoki(double pBok1, double pBok2) {
    bok1 = pBok1;
    bok2 = pBok2;
}
double Prostokat::pole() {
    return bok1 * bok2;
}
double Prostokat::obwod() {
    return 2 * bok1 + 2 * bok2;
}

// Definicja funkcji zewnętrznej, niezależnej od klasy Prostokat:

Prostokat kopiujProstokat(Prostokat prostokat) {
    return prostokat;
}

/* UWAGA
* Parametrem formalnym funkcji kopiujProstokat() jest obiekt klasy Prostokat, który jest przekazywany do funkcji
* przez wartość. Dlatego też na stosie podczas wywołania tej funkcji zostanie wykonana kopia obiektu jako argumentu
* tego wywołania.
* Funkcja zwraca na zewnątrz obiekt typu Prostokat. Tym samym na stosie w trakcie wywołania tej funkcji ponownie
* zostanie wykonana kopia obiektu.
*/

int main() {
    double b1, b2; // zmienne pomocnicze

    cout << "PIERWSZY PROSTOKĄT" << endl;
// Utworzenie obiektu wzorcowego:
    Prostokat p1(1, 2); // wywołanie konstruktora parametrycznego
// Wyświetlenie wartości zmiennych członkowskich obiektu p1:
    p1.pobierzBoki(b1, b2);
    cout << "bok1 = " << b1 << endl;
    cout << "bok2 = " << b2 << endl;
    cout << endl;

    cout << "DRUGI PROSTOKĄT" << endl;

```

```

// Utworzenie obiektu p2 na podstawie obiektu p1:
Prostokat p2 = p1;
/* UWAGA
 * Na stoisie utworzony zostaje nowy obiekt p2 jako kopia obiektu wzorcowego p1.
 * Konstruktor kopiujący jest wywoływany w sposób niejawny.
 */
p2.pobierzBoki(b1, b2);
cout << "bok1 = " << b1 << endl;
cout << "bok2 = " << b2 << endl;
cout << endl;

cout << "TRZECI PROSTOKĄT" << endl;
// Utworzenie obiektu p3 na podstawie obiektu p1:
Prostokat p3(p1);
/* UWAGA
 * Na stoisie utworzony zostaje nowy obiekt p3, należący do klasy Prostokat, jako kopia obiektu wzorcowego p1.
 * Konstruktor kopiujący jest wywoływany w sposób jawny — bezpośredni.
 * Argumentem wywołania konstruktora kopiującego jest referencja do obiektu p1, traktowanego jako stała const.
 */
p3.pobierzBoki(b1, b2);
cout << "bok1 = " << b1 << endl;
cout << "bok2 = " << b2 << endl;
cout << endl;

cout << "CZWARTY PROSTOKĄT" << endl;
// Utworzenie obiektu p4 przez użycie konstruktora domyślnego:
Prostokat p4;
// Skopiowanie obiektu p1 do obiektu p4:
p4 = kopiujProstokat(p1);
/* UWAGA
 * Konstruktor kopiujący został w tym przypadku wywołany dwukrotnie: pierwszy raz — podczas kopowania
 * argumentu do funkcji kopiujProstokat(), a drugi — podczas zwracania wartości przez tę funkcję.
 * Obydwa wywołania konstruktora kopiującego są niejawne.
 */
p4.pobierzBoki(b1, b2);
cout << "bok1 = " << b1 << endl;
cout << "bok2 = " << b2 << endl;
cout << endl;

return 0;
}

```

W przykładzie zdefiniowano konstruktor kopiujący należący do klasy `Prostokat`. Konstruktor ten podczas wykonywania programu jest wywoływany kilkakrotnie — zarówno w sposób jawny, jak i niejawny.

Jawne wywołanie konstruktora kopiącego `Prostokat p3(p1)`; powoduje utworzenie nowego obiektu `p3` jako instancji klasy `Prostokat` — na podstawie obiektu wzorcowego `p1`. Zmienne członkowskie obiektu `p3` zostają zainicjowane wartościami skopiowanymi z elementów członkowskich obiektu `p1`.

Natomiast niejawne wywołanie konstruktora kopiącego następuje przy tworzeniu i inicjalizacji obiektu `p2`: `Prostokat p2 = p1;`. Operator `=` oznacza w tym przypadku wywołanie konstruktora kopiącego, a nie przypisanie.

Oprócz tego dwukrotne niejawne wywołanie konstruktora kopiącego następuje przy wywołaniu niezależnej funkcji `kopijProstokat()`: `p4 = kopijProstokat(p1);`. Obiekt `p1` jest przekazywany do tej funkcji `kopijProstokat()` przez wartość, a rezultat wywołania jest zwracany na zewnątrz omawianej funkcji — również przez wartość. Tym samym kopia obiektu `p1` jest wykonywana w pamięci operacyjnej na stosie dwukrotnie.

Ćwiczenie 12.13

Zmodyfikuj program zawarty w przykładzie 12.13 — utwórz i zainicjuj obiekty `p2` oraz `p4` przez jawne, bezpośrednie, wywołanie konstruktora kopiącego. Jako wzorzec wykorzystaj obiekt `p1`.

12.4. Delegowanie konstruktorów

W procesie definiowania konstruktatorów często się zdarza, że każdy z nich realizuje dokładnie te same czynności (operacje), np. inicjuje te same zmienne członkowskie. Skutkuje to koniecznością powielenia tego samego kodu w każdym konstraktorze — co jest zjawiskiem niepożądanym.

Powielania tego samego kodu źródłowego w wielu konstruktorkach w praktyce można uniknąć na dwa sposoby. Pierwszy z nich polega na zdefiniowaniu dodatkowej funkcji, która zawiera kod źródłowy wspólny dla wielu konstruktorek. Następnie w każdym z konstruktorek zdefiniowaną funkcję należy po prostu wywołać. Drugi sposób opiera się na wykorzystaniu mechanizmu delegowania konstruktora.

Delegowanie konstruktora (ang. *constructor delegation, delegating constructor*) polega na wywołaniu jednego konstruktora — tego, który zawiera wspólny kod — przez inny konstruktor należący do tej samej klasy. Delegowany konstruktor może być wywoływany przez wiele innych konstruktorek. W celu wywołania konstruktora zawierającego wspólny kod należy to wywołanie umieścić na liście inicjalizacyjnej innego konstruktora.

Składnia delegacji konstruktora do innego konstruktora jest zgodna z zasadami inicjalizacji zmiennych członkowskich klasy z zastosowaniem listy inicjalizacyjnej.

**UWAGA**

Delegowanie konstruktorów jest możliwe począwszy od standardu C++11.

Poniżej zilustrowano za pomocą przykładów oba wymienione wcześniej sposoby uniknięcia powielania tego samego kodu w konstruktorach. Przy czym w przykładzie 12.14 wykorzystano sposób polegający na zdefiniowaniu dodatkowej funkcji, która zawiera kod źródłowy wspólny dla wielu konstruktorów. W przykładzie 12.15 z kolei zaimplementowano mechanizm delegowania konstruktora.

Przykład 12.14

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

// Definicja klasy Prostokat:
class Prostokat {
private:
    // Deklaracja prywatnych zmiennych członkowskich:
    double bok1;
    double bok2;
    string kolor;
public:
    // Prototyp konstruktora domyślnego:
    Prostokat();
    // Prototyp konstruktora z parametrami:
    Prostokat(double, double);
    // Prototypy metod publicznych:
    void ustawKolor(string);
    void wyswietlDane();
};

// Definicja konstruktora domyślnego:
Prostokat::Prostokat() {
    // Wyświetlenie komunikatu kontrolnego:
    cout << "Komunikat kontrolny: został wywołany konstruktor domyślny ..."
        << endl;
    // Wywołanie metody ustawKolor():
    ustawKolor("biały"); // Ustalenie wartości domyślnej zmiennej członkowskiej kolor.
}

// Definicja konstruktora parametrycznego:
Prostokat::Prostokat(double pBok1, double pBok2) {
    // Wyświetlenie komunikatu kontrolnego:
    cout << "Komunikat kontrolny: został wywołany konstruktor parametryczny ..."
        << endl;
}
```

```
// Inicjalizacja zmiennych członkowskich bok1 i bok2:  
bok1 = pBok1;  
bok2 = pBok2;  
// Wywołanie metody ustawKolor():  
ustawKolor("biały"); // Ustalenie wartości domyślnej zmiennej członkowskiej kolor.  
}  
/* UWAGA  
* W celu uniknięcia powielania kodu w zdefiniowanych konstruktorach w każdym z nich wywoływana jest  
* ta sama funkcja: ustawKolor(), z argumentem „biały”.  
* Wartość początkowa „biały” jest traktowana jako wartość domyślna zmiennej członkowskiej kolor.  
*/  
// Definicje metod publicznych należących do klasy Prostokat:  
void Prostokat::ustawKolor(string pKolor) {  
    kolor = pKolor;  
}  
void Prostokat::wyswietlDane() {  
    cout << "Pierwszy bok: " << bok1 << endl;  
    cout << "Drugi bok: " << bok2 << endl;  
    cout << "Kolor: " << kolor << endl;  
}  
  
int main() {  
    // Utworzenie obiektu p1:  
    Prostokat p1;  
    // Wyświetlenie parametrów prostokąta p1:  
    p1.wyswietlDane();  
    cout << endl;  
  
    // Utworzenie obiektu p2:  
    Prostokat p2(1, 2);  
    // Wyświetlenie parametrów prostokąta p2:  
    p2.wyswietlDane();  
  
    // Ustawienie (modyfikacja) wartości zmiennej członkowskiej kolor obiektu p2:  
    p2.ustawKolor("czarny");  
    /* UWAGA  
     * Modyfikacja wartości prywatnej zmiennej członkowskiej kolor została wykonana przez wywołanie publicznej  
     * metody dostępowej ustawKolor().  
     */  
    // Ponowne wyświetlenie parametrów prostokąta p2:  
    p2.wyswietlDane();  
  
    return 0;  
}
```

W klasie Prostokat zdefiniowano metodę publiczną ustawKolor(), która pozwala na ustanowienie/zmianę wartości zmiennej członkowskiej kolor. Wywołanie tej metody w każdym ze zdefiniowanych konstruktorów stanowi ich wspólny kod.

Wywołanie konstruktora domyślnego (Prostokat p1;) powoduje zainicjowanie zmiennej członkowskiej kolor prostokąta p1. Pozostałe zmienne członkowskie, tj. bok1 i bok2, nie zostają zainicjowane. Wywołanie konstruktora parametrycznego (Prostokat p2(1, 2);) skutkuje natomiast inicjalizacją wszystkich zmiennych członkowskich obiektu p2.

Po wywołaniu metody ustawKolor() z argumentem "czarny" następuje zmiana wartości zmiennej członkowskiej kolor prostokąta p2 z wartości "biały" na "czarny".

Ćwiczenie 12.14

Zmodyfikuj program z przykładu 12.14 — zdefiniuj dodatkowo prywatną metodę bezparametrową o nazwie ustawKolorPoczątkowy(), która będzie zawierać w swoim ciele jedną instrukcję: kolor = "biały";. Wywołaj tę metodę w obu konstruktorach zamiast metody publicznej ustawKolor(). Zinterpretuj uzyskane rezultaty.

Przykład 12.15

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

// Definicja klasy Prostokat:
class Prostokat {
private:
    // Deklaracja prywatnych zmiennych członkowskich:
    double bok1;
    double bok2;
    string kolor;
public:
    // Prototyp konstruktora domyślnego:
    Prostokat();
    // Prototyp konstruktora z parametrami:
    Prostokat(double, double);
    // Prototypy metod publicznych:
    void ustawKolor(string);
    void wyswietlDane();
};

// Definicja konstruktora domyślnego:
Prostokat::Prostokat() {
    // Wyświetlenie komunikatu kontrolnego:
    cout << "Komunikat kontrolny: został wywołany konstruktor domyślny ..."
        << endl;
    // Inicjalizacja zmiennej członkowskiej kolor:
    kolor = "biały";
}
```

```

// Definicja konstruktora z parametrami z delegacją konstruktora domyślnego:
Prostokat::Prostokat(double pBok1, double pBok2) : Prostokat::Prostokat() {
    // Wyświetlenie komunikatu kontrolnego:
    cout << "Komunikat kontrolny: został wywołany konstruktor parametryczny ..."
        << endl;
    // Inicjalizacja zmiennych członkowskich bok1 i bok2:
    bok1 = pBok1;
    bok2 = pBok2;
}

/* UWAGA
* W celu uniknięcia powielania kodu, który jest zawarty w konstruktorze domyślnym,
* wywołanie konstruktora domyślnego umieszczone na liście inicjalizacyjnej konstruktora parametrycznego.
*/

// Definicje metod publicznych należących do klasy Prostokat:
void Prostokat::ustawKolor(string pKolor) {
    kolor = pKolor;
}
void Prostokat::wyswietlDane() {
    cout << "Pierwszy bok: " << bok1 << endl;
    cout << "Drugi bok: " << bok2 << endl;
    cout << "Kolor: " << kolor << endl;
}

int main() {
    // Utworzenie obiektu p1:
    Prostokat p1;
    // Wyświetlenie parametrów prostokąta p1:
    p1.wyswietlDane();
    cout << endl;

    // Utworzenie obiektu p2:
    Prostokat p2(1, 2);
    // Wyświetlenie parametrów prostokąta p2:
    p2.wyswietlDane();

    // Ustawienie (modyfikacja) wartości zmiennej członkowskiej kolor dla prostokąta p2:
    p2.ustawKolor("czarny");
/* UWAGA
* Modyfikacja wartości prywatnej zmiennej członkowskiej kolor została wykonana przez wywołanie
* publicznej metody dostępowej ustawKolor().
*/
    // Ponowne wyświetlenie parametrów prostokąta p2:
    p2.wyswietlDane();

    return 0;
}

```

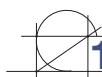
W klasie `Prostokat` zdefiniowano dwa konstruktory: domyślny i parametryczny. Konstruktor domyślny pozwala na inicjalizację wyłącznie jednej zmiennej członkowskiej — `kolor`. Wartość domyślna tej zmiennej została określona jako "biały".

Wywołanie w programie głównym konstruktora domyślnego, `Prostokat p1;`, powoduje zainicjowanie zmiennej członkowskiej `kolor` prostokąta `p1`. Pozostałe zmienne członkowskie, tj. `bok1` i `bok2`, nie zostają zainicjowane.

Wywołanie konstruktora parametrycznego z dwoma argumentami: `1` oraz `2`, skutkuje natomiast inicjalizacją wszystkich zmiennych członkowskich obiektu `p2`: `bok1`, `bok2` i `kolor`. Dzieje się tak dlatego, że przed wywołaniem konstruktora parametrycznego zostaje wywołany „delegowany” konstruktor domyślny, który zawiera inicjalizację zmiennej `kolor`.

Ćwiczenie 12.15

Zmodyfikuj program z przykładu 12.15 — zainicjuj w konstruktorze domyślnym wszystkie zmienne członkowskie klasy `Prostokat`. Program główny pozostaw bez zmian. Zinterpretuj uzyskane rezultaty.



12.5. Destruktory

Destruktor (ang. *destructor*) jest specjalną funkcją członkowską klasy, która jest wykonywana automatycznie podczas niszczenia lub usuwania obiektu. Zadaniem destruktora jest zwolnienie zasobów pamięci operacyjnej przydzielonej obiekowi podczas jego tworzenia i inicjalizacji za pomocą konstruktora.

Ogólna postać definicji destruktora jest następująca:

```
~nazwa_klasy() {
    zestaw_instrukcji_pomocniczych;
}
```

gdzie:

- `nazwa_klasy` jest identyfikatorem klasy, która zawiera destruktur,
- `zestaw_instrukcji_pomocniczych` to ciąg instrukcji wykonujących zadania pomocnicze, np. informacyjne.

Definicja destruktora w klasie jest opcjonalna. W przypadku klas, które nie operują na strukturach dynamicznych, plikach, bazach danych, definiowanie destruktora nie jest konieczne. C++ automatycznie zwolni (wyczyści) pamięć zaalokowaną dla wykorzystywanych obiektów.

Destruktory mają pewne szczególne właściwości (cechy), które odróżniają je od zwykłych metod członkowskich klasy, np.:

- w klasie można zdefiniować tylko jeden destruktur,
- destruktur nie ma żadnych parametrów (argumentów),

- destruktor nie zwraca do swojego otoczenia żadnej wartości,
- destruktor nie może być zadeklarowany jako static albo const.

Destruktory są wywoływanie automatycznie w sposób niejawny w sytuacji, gdy obiekt jest niszczony lub usuwany, tj.:

- w chwili zakończenia programu (z wyjątkiem awaryjnego zabicia aplikacji),
- w chwili zakończenia wykonywania funkcji, w której obiekt został zdefiniowany (w przypadku inicjalizacji na stosie),
- na końcu bloku kodu, w którym obiekt został zdefiniowany (w przypadku inicjalizacji na stosie),
- w chwili wywołania operatora delete.

Przykład 12.16

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    string imie;
    string nazwisko;
// Definicje konstruktörów:
    Pracownik() {
        cout << "Komunikat kontrolny: wywołanie konstruktora domyślnego ..."
            << endl;
        imie = "Imię domyślne";
        nazwisko = "Nazwisko domyślne";
    };
    Pracownik(string pImie, string pNazwisko) {
        cout << "Komunikat kontrolny: wywołanie konstruktora parametrycznego ..."
            << endl;
        imie = pImie;
        nazwisko = pNazwisko;
    };
// Definicja destruktora:
    ~Pracownik() {
        cout << "Komunikat kontrolny: wywołanie destruktora obiektu ..."
            << endl;
    };
// Definicja funkcji członkowskiej:
    void wyswietlDane() {
        cout << "Imię: " << imie << endl;
        cout << "Nazwisko: " << nazwisko << endl;
    }
};
```

```

// Definicja zewnętrznej funkcji pomocniczej:
void funkcjaPomocnicza() {
    cout << "Komunikat kontrolny: wywołanie funkcji pomocniczej ..." << endl;
    Pracownik pracownik;
    pracownik.wyswietlDane();
}

int main() {
    { // początek bloku kodu
        // Utworzenie obiektu pracownik1 należącego do klasy Pracownik:
        Pracownik pracownik1;
        pracownik1.wyswietlDane();
    } // koniec bloku
    /* UWAGA
     * Koniec bloku oznacza koniec życia obiektu pracownik1,
     * co wiąże się z wywołaniem destruktora i zniszczeniem obiektu.
     */
    cout << endl;

    // Utworzenie obiektu typu Pracownik wskazywanego przez wskaźnik w_pracownik2:
    Pracownik *w_pracownik2 = new Pracownik("Jan", "Kowalski");
    w_pracownik2->wyswietlDane();
    // Usunięcie obiektu wskazywanego przez wskaźnik w_pracownik2:
    delete w_pracownik2;
    /* UWAGA
     * Użycie operatora delete spowoduje wywołanie destruktora obiektu.
     */
    cout << endl;

    // Wywołanie zewnętrznej funkcji pomocniczej:
    funkcjaPomocnicza();
    cout << endl;
    cout << "Wykonanie pozostałych instrukcji w programie ...";

    return 0;
}

```

Klasa `Pracownik` zawiera — oprócz innych elementów członkowskich — definicję dwóch konstruktorów i destruktora. Destruktor jest wywoływany w programie trzykrotnie. Każde z tych wywołań jest niejawne.

Pierwsze z wywołań destruktora następuje w chwili wyjścia obiektu `pracownik1` poza zakres bloku kodu, w którym obiekt ten został zdefiniowany. Powoduje to zniszczenie obiektu i zwolnienie zasobów w pamięci operacyjnej przydzielonych w celu jego przechowania.

Drugie wywołanie destruktora następuje po użyciu operatora `delete`, powodującego usunięcie obiektu, dla którego pamięć operacyjna została zaalokowana dynamicznie.

Trzecie, ostatnie wywołanie destruktora następuje w chwili zakończenia działania funkcji zewnętrznej `funkcjaPomocnicza()`, w której został zdefiniowany obiekt `pracownik`.

Ćwiczenie 12.6

Zmodyfikuj program z przykładu 12.16 — wywołaj konstruktory obiektów w sposób jawnny. Sprawdź, czy ma to wpływ na wywołania destruktora. Uzasadnij uzyskane rezultaty.



12.6. Pytania i zadania kontrolne

12.6.1. Pytania

- 1.** Co to jest konstruktor? Jakie zadania realizuje?
- 2.** Co to jest konstruktor domyślny? Czym się różni konstruktor domyślny od konstruktora z parametrami?
- 3.** Jaka jest rola konstruktora kopiącego?
- 4.** Na czym polega delegowanie konstruktorów? W jakim celu deleguje się konstruktory?
- 5.** Omów proces inicjowania obiektów z zastosowaniem listy inicjalizacyjnej.
- 6.** Co to jest destruktor? Jakie zadania realizuje?

12.6.2. Zadania

- 1.** Napisz program pozwalający zapamiętać w zmiennych członkowskich obiektu uczen dane ucznia: imię, nazwisko, klasę, grupę, numer w dzienniku. Wykorzystaj klasę `Uczen` zawierającą definicje niezbędnych zmiennych członkowskich, konstruktorów — domyślnego i parametrycznego — oraz metody `wyswietlDane()`. Konstruktor domyślny powinien mieć możliwość inicjalizacji wszystkich zmiennych członkowskich wartościami domyślnymi, a konstruktor parametryczny — wartościami zadanymi. Metoda `wyswietlDane()` ma za zadanie wyświetlenie na ekranie monitora wartości przechowywanych w zmiennych członkowskich. Dane wejściowe mają być wprowadzane z klawiatury. Dane po wprowadzeniu i zapisaniu w obiekcie `uczen` powinny być wyświetlane kontrolnie na ekranie monitora. Załóż, że wszystkie elementy członkowskie klasy `Uczen` są publiczne.
- 2.** Napisz program umożliwiający obliczenie objętości, pola powierzchni bocznej oraz długości wszystkich krawędzi prostopadłościanu. Parametry prostopadłościanu mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Wykorzystaj klasę `Prostopadloscian` zawierającą definicje niezbędnych zmiennych członkowskich, konstruktorów — domyślnego i parametrycznego — oraz funkcji członkowskich. Konstruktor domyślny powinien mieć możliwość inicjalizacji wszystkich zmiennych członkowskich wartościami domyślnymi, a konstruktor parametryczny — wartościami zadanymi.
- 3.** Napisz program pozwalający obliczyć pola powierzchni i obwody pomieszczeń w mieszkaniu. Załóż, że każde z pomieszczeń ma kształt prostokąta, a ich liczba wynosi 4. Wykorzystaj zdefiniowaną samodzielnie klasę `Prostokat`. Obiekty odpowiadające po-

szczególnym pomieszczeniom w mieszkaniu utwórz i zainicjuj wartościami domyślnymi z użyciem konstruktora kopiującego.

4. Napisz program pozwalający przeliczyć długość wyrażoną w wybranej jednostce angielskiej na długość mierzoną w metrach. Zastosuj zdefiniowaną samodzielnie klasę o nazwie `Przeliczenie_Dlugosci` zawierającą niezbędne definicje, w tym definicje konstruktora domyślnego i parametrycznego. Zadaniem konstruktora domyślnego jest m.in. wyświetlenie komunikatu informującego użytkownika, że docelową jednostką miary jest metr. Definicja konstruktora parametrycznego powinna zawierać delegację (wywołanie) konstruktora domyślnego. Wykorzystaj obiekt utworzony i zainicjowany przy użyciu konstruktora parametrycznego.
5. Napisz program pozwalający zapamiętać w obiekcie `samochod` dane samochodu: markę, model, rok produkcji, cenę, numer rejestracyjny, datę pierwszej rejestracji. Wykorzystaj zdefiniowaną samodzielnie klasę `Samochod` zawierającą definicje niezbędnych elementów członkowskich, w tym definicje konstruktora domyślnego i parametrycznego oraz destruktora. Zmienna członkowska reprezentująca datę pierwszej rejestracji samochodu powinna należeć do typu strukturowego. Zainicjuj obiekt `samochod` będący instancją klasy `Samochod` przy użyciu konstruktora parametrycznego z listą inicjalizacyjną. Dane wejściowe mają być wprowadzane z klawiatury. Dane po wprowadzeniu i zapisaniu w obiekcie `samochod` powinny być wyświetlane kontrolnie na ekranie monitora. Załącz, że wszystkie zmienne członkowskie klasy `Samochod` są prywatne.
6. Napisz program pozwalający obliczyć odległość pomiędzy dwoma punktami na płaszczyźnie. Dane wejściowe (tj. współrzędne kartezjańskie dwóch punktów) mają być wprowadzane z klawiatury, a wynik wyświetlany na ekranie monitora. Wykorzystaj obiekt `odleglosc` będący instancją klasy `Odleglosc`, która zawiera definicje: zmiennych członkowskich typu strukturowego odpowiadających współrzędnym punktów, konstruktörów — domyślnego i parametrycznego — destruktora oraz metody umożliwiającej obliczenie odległości pomiędzy zadanymi punktami. Zainicjuj obiekt `odleglosc` zadanymi wartościami za pomocą konstruktora parametrycznego z użyciem listy inicjalizacyjnej (wariant pierwszy) i metody konstruktorowej (wariant drugi). Załącz, że wszystkie zmienne członkowskie klasy `Odleglosc` są prywatne.

13

Hermetyzacja i ukrywanie danych

13.1. Hermetyzacja danych

Hermetyzacja danych (ang. *data encapsulation*) to jedno z najważniejszych założeń para-dygmatu obiektowego.

UWAGA

Mechanizm hermetyzacji danych jest również nazywany **enkapsulacją** lub **kapsułkowaniem**.

W ogólności enkapsulacja polega na powiązaniu ze sobą danych z operacjami wykonywanymi na tych danych. Wspomniane powiązanie jest realizowane w ramach określonego kontenera (kapsuły).

Język C++ pozwala na obsługę mechanizmu enkapsulacji dzięki możliwości samodzielnego definiowania klas, które odgrywają rolę „inteligentnego” kontenera (kapsuły). W ramach klas programista określa (definiuje) zmienne członkowskie, reprezentujące dane, oraz funkcje członkowskie (metody), które stanowią implementację operacji wykonywanych na danych.

Niezwykle istotne jest również to, że programista w ramach realizowanej koncepcji klasy ma możliwość określenia „wiązań” pomiędzy danymi a funkcjami oraz może ustalić indywidualnie poziom dostępu do każdego z elementów członkowskich klasy za pomocą specyfikatorów dostępu, np. `public`, `private`.

Przykład 13.1

```
#include <iostream>
using namespace std;
// Definicja klasy Prostokat:
class Prostokat {
public:
    // Definicje zmiennych członkowskich:
    double bok1 {0};
    double bok2 {0};
    // Prototypy konstruktorów:
    Prostokat();
    Prostokat(double, double);
    // Prototyp metody dostępowej:
    void wyswietlBoki();
};

// Definicje konstruktorów:
Prostokat::Prostokat() {}
Prostokat::Prostokat(double pBok1, double pBok2) {
    bok1 = pBok1;
    bok2 = pBok2;
}
// Definicja metody wyswietlBoki():
void Prostokat::wyswietlBoki() {
    cout << "Komunikat kontrolny: wywołanie metody wyswietlBoki()" << endl;
    cout << "Pierwszy bok = " << bok1 << endl;
    cout << "Drugi bok = " << bok2 << endl;
}
/* UWAGA
 * Powiązanie zmiennych członkowskich (danych) bok1 i bok2 z funkcją członkowską wyswietlBoki() następuje już
 * na poziomie definicji tej metody.
 */
// Definicja funkcji zewnętrznej:
void wyswietlBoki(double b1, double b2) {
    cout << "Komunikat kontrolny: wywołanie funkcji zewnętrznej wyswietlBoki()"
        << endl;
    cout << "Pierwszy bok = " << b1 << endl;
    cout << "Drugi bok = " << b2 << endl;
}
/* UWAGA
 * Funkcja wyswietlBoki() została zdefiniowana w przestrzeni globalnej — na zewnątrz klasy Prostokat.
 * Jest ona zatem w pełni niezależna od klasy Prostokat.
 */
```

```

int main() {
    // Deklaracja i inicjalizacja zmiennych reprezentujących boki prostokąta:
    double dlugosc = 1; // Pierwszy bok
    double szerokosc = 2; // Drugi bok
    // Wywołanie funkcji zewnętrznej wyswietlBoki():
    wyswietlBoki(dlugosc, szerokosc);
    cout << endl;

    // Utworzenie i inicjalizacja obiektu prostokąt:
    Prostokat prostokat(1, 2);
    // Wywołanie funkcji członkowskiej wyswietlBoki():
    prostokat.wyswietlBoki();

    return 0;
}

```

W programie zilustrowano mechanizm hermetyzacji danych. W tym celu zdefiniowano klasę `Prostokat`, zawierającą m.in. zmienne członkowskie `bok1` i `bok2` oraz funkcję członkowską `wyswietlBoki()`.

Zmienne `bok1` i `bok2` reprezentują dane — długości boków prostokąta. Funkcja członkowska `wyswietlBoki()` zaś odpowiada za wykonanie operacji na tych danych — wyświetlenie ich wartości na ekranie monitora.

W ciele metody `wyswietlBoki()` powiązano dane (`bok1` i `bok2`) ze wspomnianymi operacjami na tych danych. Należy podkreślić, że „wiązanie” to zostało zrealizowane już na poziomie definicji klasy `Prostokat`. Mamy więc tutaj do czynienia z mechanizmem hermetyzacji danych `bok1` i `bok2` powiązanych z metodą `wyswietlBoki()` w ramach klasy `Prostokat`.

Ćwiczenie 13.1

Szczegółowo przeanalizuj kod programu zawartego w przykładzie 13.1. Na podstawie przeprowadzonej analizy odpowiedz na pytania:

- Czy funkcja zewnętrzna `wyswietlBoki()` zdefiniowana w przestrzeni globalnej jest powiązana funkcjonalnie na poziomie jej definicji z danymi reprezentowanymi w programie przez zmienne `dlugosc` i `szerokosc`?
- Czy wywołanie funkcji zewnętrznej `wyswietlBoki()` z argumentami — danymi `dlugosc` i `szerokosc` — wiąże te dane ze wspomnianą funkcją?



13.2. Ukrywanie danych

Mechanizm hermetyzacji danych może prowadzić do **ukrywania danych** (ang. *data hiding*) przed „światem zewnętrznym” — rozumianym jako otoczenie klasy, w której te dane zostały określone (zdefiniowane). W tym celu wykorzystuje się specyfikator dostępu (ang. *access specifier*) `private` w odniesieniu do zmiennych członkowskich klasy reprezentujących wspomniane dane.



UWAGA

Dane można ukrywać również z zastosowaniem specyfikatora dostępu o nazwie `protected`. Specyfikator `protected` jest wykorzystywany w implementacji mechanizmu dziedziczenia, omówionego w rozdziale 14. podręcznika.

Użycie specyfikatora dostępu `private` względem zmiennych członkowskich klasy zapewnia, że dane, które są przechowywane w tych zmiennych, są dostępne wyłącznie w obrębie definicji tej klasy. Patrząc na to z drugiej strony — dane pamiętane w prywatnych zmiennych członkowskich klasy są ukryte przed światem zewnętrznym.

W praktyce mechanizm enkapsulacji pozwala na **ukrywanie danych wrażliwych** (ang. *sensitive data hiding*) przed nieautoryzowanym lub nieprawidłowym dostępem, czyli zapewnia zwiększenie bezpieczeństwa tych danych.

Dostęp do prywatnych zmiennych członkowskich na poziomie otoczenia (na zewnątrz) klasy, w której zostały zdefiniowane, można uzyskać za pomocą metod publicznych nazywanych **akcesorami** (ang. *accessors*) lub **metodami dostępowymi** (ang. *access methods*). Akcesory, jako publiczne metody dostępowe do prywatnych zmiennych członkowskich klasy, można podzielić na dwie zasadnicze grupy: settery i gettery.

Setter (ang. *setter*) to publiczna metoda dostępową, która umożliwia nadanie oraz zmianę wartości prywatnej zmiennej członkowskiej klasy.

Getter (ang. *getter*) to metoda publiczna, która umożliwia odczyt — pobranie wartości zmiennej członkowskiej klasy.

Status publiczny (`public`) setterów i getterów oznacza, że można z nich korzystać w dowolnym otoczeniu klasy, w której zostały zdefiniowane — oczywiście z zachowaniem ograniczeń związanych z zakresem widoczności klasy.

Podsumowując, mechanizm ukrywania danych polega na definiowaniu klas, które zawierają prywatne zmienne członkowskie reprezentujące dane. „Prywatność” tych zmiennych — danych — oznacza, że nie są one dostępne na zewnątrz klasy w sposób bezpośredni. Dostęp do prywatnych zmiennych członkowskich w otoczeniu klasy, w której zostały zdefiniowane, zapewniają publiczne metody dostępowe tej klasy — settery i gettery.

Przykład 13.2

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

// Definicja klasy Prostokat:
class Prostokat {
private:
    // Definicje prywatnych zmiennych członkowskich:
```

```

    double bok1 {0};
    double bok2 {0};
public:
    // Prototypy konstruktorów:
    Prostokat();
    Prostokat(double, double);

    // Prototypy publicznych metod dostępowych — akcesorów:
    void setBok1(double); // setter zmiennej członkowskiej bok1
    double getBok1(); // getter zmiennej członkowskiej bok1
    void setBok2(double); // setter zmiennej członkowskiej bok2
    double getBok2(); // getter zmiennej członkowskiej bok2
};

// Definicje konstruktorów:
Prostokat::Prostokat() {}
Prostokat::Prostokat(double pBok1, double pBok2) {
    bok1 = pBok1;
    bok2 = pBok2;
}

// Definicje akcesorów — setterów i getterów:
void Prostokat::setBok1(double pBok1) {
    bok1 = pBok1;
}
double Prostokat::getBok1() {
    return bok1;
}
void Prostokat::setBok2(double pBok2) {
    bok2 = pBok2;
}
double Prostokat::getBok2() {
    return bok2;
}

int main() {
    // Deklaracje zmiennych reprezentujących boki prostokąta:
    double dlugosc = 1; // pierwszy bok
    double szerokosc = 2; // drugi bok

    // Utworzenie obiektu prostokąt:
    Prostokat prostokat;
    /* UWAGA
     * Wywołanie zdefiniowanego konstruktora domyślnego nie skutkuje inicjalizacją zmiennych
     * członkowskich bok1 i bok2.
     */
}

```

```

// Nadanie wartości początkowych zmiennym członkowskim bok1 i bok2:
prostokat.setBok1(dlugosc); // wywołanie settera setBok1() z argumentem dlugosc
prostokat.setBok2(szerokosc); // wywołanie settera setBok2() z argumentem szerokosc
// Prezentacja danych prostokąta na ekranie monitora:
cout << "Długości boków prostokąta: " << endl;
cout << "Pierwszy bok = " << prostokat.getBok1() << endl; // wywołanie gettera
// getBok1()
cout << "Drugi bok = " << prostokat.getBok2() << endl; // wywołanie gettera getBok2()

return 0;
}

```

W programie zdefiniowano klasę `Prostokat` zawierającą definicje prywatnych zmiennych członkowskich `bok1` i `bok2`. Oprócz tego w klasie `Prostokat` zdefiniowano publiczne metody dostępowe — akcesory do wymienionych zmiennych.

Akcesory `setBok1()` i `setBok2()` pełnią funkcję setterów dla zmiennych członkowskich `bok1` i `bok2`. Akcesory `getBok1()` i `getBok2()` to gettery.

Zmienne członkowskie `bok1` i `bok2`, reprezentujące dane — długości boków prostokąta — są ukryte przed „światem zewnętrznym” w ramach definicji klasy `Prostokat`. Dostęp do wspomnianych danych z poziomu otoczenia klasy `Prostokat` można uzyskać wyłącznie za pomocą wymienionych wcześniej publicznych metod dostępowych — setterów i getterów, które zostały zdefiniowane dla każdej zmiennej członkowskiej w tej klasie.

Konstruktor parametryczny pełni funkcję settera specjalnego, ale w odniesieniu do obu zmiennych członkowskich jednocześnie.

Ćwiczenie 13.2

Na podstawie przykładu 13.2 napisz program o takiej samej funkcjonalności, ale bez użycia żadnych obiektów. W tym celu zapisz dane prostokąta w strukturze traktowanej jako C-struktura. Zdefiniuj niezależne od siebie funkcje pozwalające na dostęp do danych prostokąta w trybie „zapisz/modyfikuj” i w trybie „odczytaj”.

13.3. Pytania i zadania kontrolne

13.3.1. Pytania

1. Na czym polega mechanizm hermetyzacji danych, a na czym mechanizm ukrywania danych?
2. Co to jest akcesor? Jakie są rodzaje akcesorów?

13.3.2. Zadania

1. Napisz program pozwalający obliczyć objętość, pole powierzchni całkowitej oraz długość wszystkich krawędzi prostopadłościanu. Dane wejściowe — wartości parametrów prostopadłościanu — mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Wykorzystaj zdefiniowaną samodzielnie klasę `Prostopadloscian`, która będzie zawierać zmienne członkowskie odpowiadające parametrom prostopadłościanu. Zastosuj do tych danych mechanizm hermetyzacji — ukrywania ich przed „światem zewnętrznym”.
2. Napisz program pozwalający przechować w zmiennych członkowskich klasę `Uczen` następujące dane ucznia: imię, nazwisko, datę urodzenia, klasę, grupę. Do wymienionych danych zastosuj mechanizm hermetyzacji — ukrywania. Dane ucznia powinny zostać wprowadzone z klawiatury, a następnie wyświetcone kontrolnie na ekranie monitora.

14

Mechanizm dziedziczenia

Dziedziczenie (ang. *inheritance*) to jedna z najważniejszych cech paradygmatu obiektowego. Polega na tym, że dana klasa może zostać zdefiniowana na podstawie innej, już istniejącej klasy i w ten sposób „oddziedziczyć” po niej określone elementy członkowskie (ang. *members*).

Dziedziczone mogą być zarówno zmienne członkowskie (ang. *member variables*), jak i funkcje członkowskie (ang. *member functions*), inaczej metody (ang. *methods*). Przy tym, aby wspomniane zmienne i funkcje członkowskie mogły być dziedziczone do innych klas, powinny mieć status albo `public`, albo `protected`.

Specyfikator dostępu (ang. *access specifier*) `public` wyszczególniony w definicji klasy przed deklaracjami jej elementów członkowskich, tj.:

public:

`deklaracje_elementów_członkowskich;`

oznacza, że elementy członkowskie klasy wymienione w deklaracjach_elementów_członkowskich są dostępne w sposób bezpośredni zarówno wewnątrz tej klasy, jak i na zewnątrz, tj. w jej otoczeniu. Była o tym mowa w podrozdziale 11.2.

Użycie w odniesieniu do elementów członkowskich klasy specyfikatora dostępu `protected`, tj.:

protected:

`deklaracje_elementów_członkowskich;`

oznacza zaś, że składniki klasy wymienione w deklaracjach_elementów_członkowskich nie są dostępne w sposób bezpośredni w otoczeniu — na zewnątrz tej klasy (czyli tak samo jak prywatne elementy członkowskie), ale zarazem mogą być dziedziczone do innych klas.

Patrząc z drugiej strony, dziedziczeniu nie podlegają następujące składniki klasy:

- konstruktory i destruktor,
- prywatne zmienne członkowskie, czyli takie, które zostały określone za pomocą specyfikatora dostępu `private`.

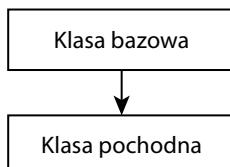
To samo dotyczy funkcji zaprzyjaźnionych z daną klasą (ang. *friend functions*).

UWAGA

Tematyka przyjaciół klas (ang. *class friends*) została omówiona w dalszej części podręcznika — w rozdziale 17. Obejmuje ona m.in. funkcje zaprzyjaźnione.

14.1. Definicja relacji dziedziczenia

Klasa istniejąca, na podstawie której tworzona jest nowa klasa, jest nazywana **klasą bazową** (ang. *base class*). Inne spotykane nazwy klasy bazowej to np. **klasa nadrzędna** (ang. *parent class*) i **superklasa** (ang. *superclass*). Nowo tworzona klasa, która dziedziczy określone pola i metody po klasie bazowej, jest zaś nazywana **klasą pochodną** (ang. *derived class*), a także **klassą dzieckiem** (ang. *child class*) lub **podklassą** (ang. *subclass*). Zilustrowano to na rysunku 14.1.



Rysunek 14.1. Interpretacja klasy podstawowej i klasy pochodnej

Relacja dziedziczenia jest deklarowana na poziomie definicji klasy pochodnej. Ogólna postać definicji relacji dziedziczenia jest następująca:

```

class nazwa_klasy_pochodnej : specyfikator_dostępu nazwa_klasy_bazowej {
    deklaracje_i_definicje_elementów_członkowskich;
};
  
```

gdzie:

- `nazwa_klasy_pochodnej` oznacza identyfikator klasy pochodnej,
- `specyfikator_dostępu` to specyfikator dostępu (ang. *access specifier*): `public`, `protected` lub `private`, określający tryb (rodzaj) dziedziczenia,
- `nazwa_klasy_bazowej` jest identyfikatorem klasy bazowej,
- `deklaracje_i_definicje_elementów_członkowskich` to deklaracje i definicje zmiennych członkowskich, metod i innych elementów składowych klasy pochodnej.

Jak wspomniano powyżej, **typ dziedziczenia** (ang. *type of inheritance*) jest określony za pomocą specyfikatora dostępu: `public`, `protected` lub `private`, wyszczególnionego w nagłówku definicji klasy pochodnej pomiędzy dwukropkiem : a identyfikatorem klasy bazowej `nazwa_klasy_bazowej`. Specyfikator ten definiuje **tryb dostępu** (ang. *access mode*) do elementów członkowskich klasy pochodnej, które zostały odziedziczone (lub nie) po klasie bazowej.

Typ (rodzaj) dziedziczenia wynikający z trybu dostępu do elementów członkowskich klasy pochodnej odziedziczonych po klasie bazowej jest nazywany **trybem dziedziczenia** (ang. *inheritance mode*).

Biorąc pod uwagę powyższe, w języku C++ można wyróżnić następujące tryby dziedziczenia:

- publiczne — jeśli w nagłówku definicji klasy pochodnej wyszczególniono specyfikator dostępu określony za pomocą słowa kluczowego `public`,
- chronione — jeśli w nagłówku definicji klasy pochodnej podano specyfikator `protected`,
- prywatne — jeśli w nagłówku definicji klasy pochodnej podano specyfikator `private`.

Domyślnym typem dziedziczenia w języku C++ jest dziedziczenie prywatne.



UWAGA

W przypadku struktur (ang. *structs*) domyślnym rodzajem dziedziczenia w C++ jest dziedziczenie publiczne.

Dziedziczenie publiczne (ang. *public inheritance*) oznacza, że:

- publiczne elementy członkowskie klasy bazowej, czyli te, które są określone za pomocą specyfikatora dostępu `public`, są dostępne również jako publiczne w klasie pochodnej i stanowią publiczne elementy składowe obiektów klasy pochodnej,
- chronione elementy składowe klasy bazowej, określone za pomocą specyfikatora dostępu `protected`, są dostępne również jako chronione w klasie pochodnej,
- prywatne elementy klasy bazowej — zdefiniowane tam jako `private` — nie są dostępne bezpośrednio w klasie pochodnej, przy czym można uzyskać do nich dostęp w sposób pośredni, przez wykorzystanie publicznych i chronionych metod należących do klasy bazowej.

Dziedziczenie chronione (ang. *protected inheritance*) ma następujące cechy:

- publiczne elementy składowe klasy bazowej są dostępne jako chronione w klasie pochodnej,
- chronione elementy składowe klasy bazowej są dostępne jako chronione w klasie pochodnej,
- prywatne elementy klasy bazowej nie są dostępne bezpośrednio w klasie pochodnej.

Dziedziczenie prywatne (ang. *private inheritance*) zaś wiąże się z tym, że:

- publiczne elementy składowe klasy bazowej są dostępne jako prywatne w klasie pochodnej,
- chronione elementy składowe klasy bazowej są dostępne jako prywatne w klasie pochodnej,
- prywatne elementy klasy bazowej nie są dostępne bezpośrednio w klasie pochodnej.

Mechanizm dziedziczenia zilustrowano na podstawie dziedziczenia publicznego za pomocą dwóch przykładów. W przykładzie 14.1 zmienne członkowskie hermetyzowane w klasach bazowej i pochodnej charakteryzują się statusem `public`, a w przykładzie 14.2 przedstawiono zastosowanie zarówno mechanizmu dziedziczenia, jak i ukrywania danych.

UWAGA

Podział dziedziczenia uwzględniający strukturę klas, pomiędzy którymi zachodzi relacja dziedziczenia, został omówiony w następnym podrozdziale.

Przykład 14.1

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

// Definicja klasy bazowej Pracownik:
class Pracownik {
public: // wszystkie elementy członkowskie klasy są publiczne
    // Definicje zmiennych członkowskich:
    string imie {" "};
    string nazwisko {" "};
    // Definicja funkcji członkowskiej:
    string zwrocDane() {
        return imie + " " + nazwisko;
    }
};

// Definicja klasy pochodnej Nauczyciel:
class Nauczyciel: public Pracownik { // dziedziczenie publiczne
public:
    // Definicja zmiennej członkowskiej:
    string przedmiot {" "};
    // Definicja funkcji członkowskiej:
    string zwrocDane() {
        return imie + " " + nazwisko + ", przedmiot: " + przedmiot;
    }
};
```

```

int main() {
    // Utworzenie obiektu nauczyciel jako instancji klasy pochodnej Nauczyciel:
    Nauczyciel nauczyciel;
    // Nadanie wartości zmiennym członkowskim obiektu nauczyciel:
    nauczyciel.imie = "Jan"; // zmienna odziedziczona po klasie bazowej
    nauczyciel.nazwisko = "Kowalski"; // zmienna odziedziczona
    nauczyciel.przedmiot = "elektronika"; // zmienna natywna — zdefiniowana w klasie pochodnej
    /* UWAGA
     * Ze względu na to, że dziedziczenie jest publiczne, w klasie pochodnej dostępne są wszystkie publiczne zmienne
     * członkowskie odziedziczone po klasie bazowej. W klasie pochodnej zmienne te (imie i nazwisko)
     * również są publiczne.
     */
    // Wywołanie metody zwrocDane() zdefiniowanej w klasie pochodnej:
    cout << "Dane nauczyciela: " << nauczyciel.zwrocDane() << endl;

    return 0;
}

```

W programie zdefiniowano dwie klasy: `Pracownik` i `Nauczyciel`, przy czym klasa `Nauczyciel` została zdefiniowana na podstawie klasy `Pracownik` z wykorzystaniem mechanizmu dziedziczenia. Klasa `Pracownik` jest klasą bazową, a klasa `Nauczyciel` — pochodną.

Dziedziczenie określono jako publiczne: `class Nauczyciel: public Pracownik`. Tym samym publiczne elementy członkowskie klasy bazowej są dostępne również jako publiczne w klasie pochodnej.

Funkcja członkowska `zwrocDane()` należąca do klasy bazowej `Pracownik` została przesłonięta (ang. *overriding*) przez funkcję o tej samej nazwie zdefiniowaną w klasie pochodnej `Nauczyciel`.

Ćwiczenie 14.1

Zmodyfikuj program z przykładu 14.1 — zdefiniuj klasę `Wychowawca`, która będzie klasą pochodną klasy `Nauczyciel`. W klasie `Wychowawca` zdefiniuj natywną zmienną członkowską `klasa` oraz metodę `zwrocDane()` o funkcjonalności analogicznej do funkcjonalności metod o tej samej nazwie zdefiniowanych w klasach `Pracownik` i `Nauczyciel`. Zastosuj dziedziczenie publiczne. Utwórz obiekt `wychowawca` jako instancję klasy `Wychowawca`. Przypisz zadane wartości jego zmiennym członkowskim, a następnie wyświetl je kontrolnie na ekranie monitora za pomocą funkcji `zwrocDane()`.

Przykład 14.2

```

#include <iostream>
using namespace std;

// Definicja klasy bazowej Pracownik:

```

```
class Pracownik {  
private:  
    // Definicje prywatnych zmiennych członkowskich:  
    string imie {" "};  
    string nazwisko {" "};  
public:  
    // Definicje publicznych setterów:  
    void setImie(string pImie) {  
        imie = pImie;  
    }  
    void setNazwisko(string pNazwisko) {  
        nazwisko = pNazwisko;  
    }  
    // Definicje publicznych getterów:  
    string getImie() {  
        return imie;  
    }  
    string getNazwisko() {  
        return nazwisko;  
    }  
    // Definicja publicznej funkcji członkowskiej:  
    string zwrocDane() {  
        return imie + " " + nazwisko;  
    }  
};  
// Definicja klasy pochodnej Nauczyciel:  
class Nauczyciel: public Pracownik { // dziedziczenie publiczne  
private:  
    // Definicja zmiennej członkowskiej:  
    string przedmiot {" "};  
public:  
    // Definicja settera:  
    void setPrzedmiot(string pPrzedmiot) {  
        przedmiot = pPrzedmiot;  
    }  
    // Definicja gettera:  
    string getPrzedmiot() {  
        return przedmiot;  
    }  
    // Definicja funkcji członkowskiej:  
    string zwrocDane() {  
        return getImie() + " " + getNazwisko() + ", przedmiot: " + przedmiot;  
    }  
}
```

```

/* UWAGA
 * Dostęp do prywatnych zmiennych członkowskich imie i nazwisko należących do klasy bazowej
 * uzyskano za pomocą publicznych metod dostępowych — setterów i getterów,
 * które również zostały zdefiniowane w tej klasie.
 */
};

int main() {
    // Utworzenie obiektu nauczyciel jako instancji klasy pochodnej Nauczyciel:
    Nauczyciel nauczyciel;
    // Nadanie wartości zmiennym członkowskim obiektu nauczyciel:
    nauczyciel.setImie("Jan");
    nauczyciel.setNazwisko("Kowalski");
    nauczyciel.setPrzedmiot("elektronika");
    /* UWAGA
     * Prywatne zmienne członkowskie imie i nazwisko należące do klasy bazowej Pracownik nie są dziedziczone
     * do klasy pochodnej Nauczyciel. Dostęp do tych zmiennych członkowskich z poziomu klasy Nauczyciel
     * uzyskano przez wykorzystanie publicznych metod dostępowych (setterów) zdefiniowanych w klasie Pracownik,
     * które podlegają dziedziczeniu. Dostęp do prywatnej zmiennej członkowskiej przedmiot zdefiniowanej
     * w klasie Nauczyciel uzyskano w analogiczny sposób.
     */
    // Wywołanie metody zwrocDane() zdefiniowanej w klasie pochodnej:
    cout << "Dane nauczyciela: " << nauczyciel.zwrocDane() << endl;

    return 0;
}

```

Funkcjonalność programu zawartego w przykładzie 14.2 jest analogiczna do funkcjonalności programu z przykładu 14.1, jednakże klasy `Pracownik` i `Nauczyciel` zaimplementowano w odmienny sposób. Tutaj zmienne członkowskie `imie` i `nazwisko` należące do klasy `Pracownik` oraz `przedmiot` z klasy `Nauczyciel` są prywatne. Dostęp do tych zmiennych jest realizowany za pomocą publicznych metod dostępowych — setterów i getterów, zdefiniowanych w obu wymienionych klasach. Mamy więc do czynienia z hermetyzacją — ukrywaniem danych.

Pomimo że zmienne członkowskie `imie` i `nazwisko` zostały zdefiniowane w klasie bazowej `Pracownik` jako prywatne — i tym samym nie są dziedziczone do klasy pochodnej `Nauczyciel` w sposób bezpośredni — można do nich uzyskać dostęp w klasie pochodnej `Nauczyciel` za pośrednictwem publicznych setterów i getterów zdefiniowanych w klasie `Pracownik`, które podlegają dziedziczeniu.

Ćwiczenie 14.2

Zmodyfikuj program z przykładu 14.2 — zdefiniuj klasę `Wychowawca`, która będzie klasą pochodną klasy `Nauczyciel`. W klasie `Wychowawca` zdefiniuj prywatną zmienną członkowską `klasa` oraz publiczne metody dostępowe do tej zmiennej, setter i getter, a także metodę

`zwrocDane()` o funkcjonalności analogicznej do funkcjonalności metody o tej samej nazwie w klasach `Pracownik` i `Nauczyciel`. Wykorzystaj dziedziczenie publiczne. Utwórz obiekt `wychowawca` jako instancję klasy `Wychowawca`, a następnie przypisz zadane wartości jego zmiennym członkowskim. Na końcu wyświetl kontrolnie wartości wszystkich zmiennych członkowskich obiektu `wychowawca` na ekranie monitora za pomocą funkcji `zwrocDane()` zdefiniowanej w klasie `Wychowawca`.

14.2. Rodzaje dziedziczenia

W poprzednim podrozdziale zostały omówione typy (rodzaje) dziedziczenia wynikające bezpośrednio z postaci definicji klasy pochodnej. Uwzględniają one tryb dostępu do elementów członkowskich klasy pochodnej wynikający z wykorzystanego specyfikatora dostępu (`public`, `private` i `protected`) w jej definicji.

Dziedziczenie można podzielić również w zależności od budowy (struktury wewnętrznej) tzw. **łańcucha dziedziczenia** (ang. *inheritance chain*), który jest zestawem klas powiązanych ze sobą relacją dziedziczenia. Uwzględniając to kryterium, można wyróżnić:

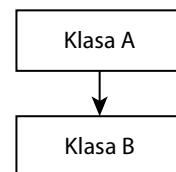
- dziedziczenie pojedyncze,
- dziedziczenie wielokrotne,
- dziedziczenie hierarchiczne,
- dziedziczenie wielopoziomowe,
- dziedziczenie mieszane.

Dziedziczenie pojedyncze (ang. *single inheritance*) zachodzi wtedy, gdy klasa pochodna dziedziczy wyłącznie po jednej klasie bazowej (rysunek 14.2).

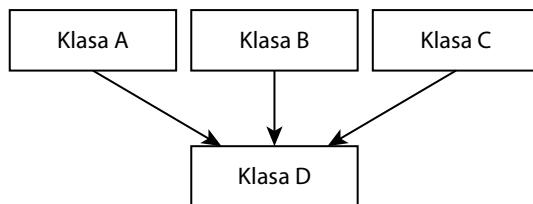
Łańcuch dziedziczenia na rysunku 14.2 składa się z dwóch klas. Klasa pochodna B dziedziczy po jednej klasie bazowej — klasie A. Zachodzi zatem dziedziczenie pojedyncze.

Dziedziczenie wielokrotne (ang. *multiple inheritance*) zachodzi wtedy, gdy dana klasa pochodna dziedziczy po większej liczbie klas bazowych (rysunek 14.3).

Relacja dziedziczenia na rysunku 14.3 została określona pomiędzy czterema klasami. Łańcuch dziedziczenia jest zatem złożony z czterech klas. Klasa pochodna D dziedziczy po trzech klasach bazowych — A, B i C. Zachodzi dziedziczenie wielokrotne.

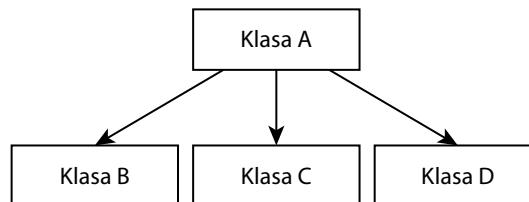


Rysunek 14.2.
Interpretacja dziedziczenia pojedynczego



Rysunek 14.3. Interpretacja dziedziczenia wielokrotnego

Dziedziczenie hierarchiczne (ang. *hierarchical inheritance*) to takie, w którym dana klasa bazowa ma wiele klas pochodnych (rysunek 14.4).



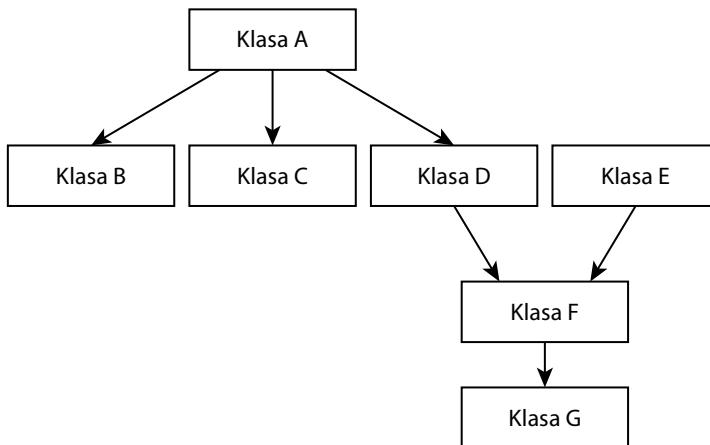
Rysunek 14.4. Interpretacja dziedziczenia hierarchicznego

Klasa bazowa A na rysunku 14.4 ma trzy klasy pochodne — B, C i D. Zachodzi dziedziczenie hierarchiczne.

Dziedziczenie wielopoziomowe (ang. *multilevel inheritance*) zachodzi wówczas, gdy dana klasa pochodna dziedziczy po klasie bazowej, która sama jest już klasą pochodną innej klasy (rysunek 14.5).

Klasa pochodna C na rysunku 14.5 dziedziczy po klasie bazowej B, która sama jest klasą pochodną względem klasy bazowej A — zachodzi dziedziczenie wielopoziomowe.

Dziedziczenie mieszane (hybrydowe) (ang. *hybrid inheritance*) to dowolna kombinacja powyższych rodzajów dziedziczenia (rysunek 14.6).



Rysunek 14.6. Interpretacja dziedziczenia mieszanego

Relacje dziedziczenia zachodzące pomiędzy klasami, które należą do łańcucha dziedziczenia na rysunku 14.6, określają różne rodzaje dziedziczenia, np. dziedziczenie wielokrotne, hierarchiczne i wielopoziomowe. Mamy więc tutaj do czynienia z dziedziczeniem hybrydowym.



UWAGA

W praktyce struktury klas połączone ze sobą relacją dziedziczenia zazwyczaj są organizowane z uwzględnieniem różnych rodzajów dziedziczenia.

Przykład 14.3

```
#include <iostream>
using namespace std;
// Definicja klasy bazowej Osoba:
class Osoba {
public:
    string imie;
    string nazwisko;
    string zwrocDane() {
        return imie + " " + nazwisko;
    }
};

// Definicja klasy bazowej Przedmiot:
class Przedmiot {
public:
    string przedmiot;
    string zwrocDane() {
        return przedmiot;
    }
};

// Definicja klasy bazowej Klasa:
class Klasa {
public:
    string klasa;
    string zwrocDane() {
        return klasa;
    }
};

// Definicja klasy pochodnej Pracownik klasy bazowej Osoba:
class Pracownik: public Osoba {
public:
    string id;
    string zwrocDane() {
        return imie + " " + nazwisko + ", numer id: " + id;
    }
};

// Definicja klasy pochodnej Nauczyciel klas bazowych Pracownik, Przedmiot:
class Nauczyciel: public Pracownik, public Przedmiot {
public:
    string zwrocDane() {
        return imie + " " + nazwisko + ", przedmiot " + przedmiot;
    }
};
```

```

// Definicja klasy pochodnej Wychowawca klas bazowych Nauczyciel i Klasa:
class Wychowawca: public Nauczyciel, public Klasa {
public:
    string zwrocDane() {
        return imie + " " + nazwisko + ", przedmiot " + przedmiot + ", klasa "
               + klasa;
    }
};

// Definicja klasy pochodnej Uczen klas bazowych Osoba i Klasa:
class Uczen: public Osoba, public Klasa {
public:
    string zwrocDane() {
        return imie + " " + nazwisko + ", klasa " + klasa;
    }
};

int main() {
    // Utworzenie obiektu wychowawca klasy Wychowawca:
    Wychowawca wychowawca;
    // Nadanie wartości zmiennym członkowskim obiektu wychowawca:
    wychowawca.imie = "Jan"; // zmienna członkowska odziedziczona po klasie Osoba
    wychowawca.nazwisko = "Kowalski"; // zmienna odziedziczona po klasie Osoba
    wychowawca.przedmiot = "informatyka"; // zmienna odziedziczona po klasie Przedmiot
    wychowawca.klasa = "4A"; // zmienna odziedziczona po klasie Klasa
    // Wywołanie metody zwrocDane() zdefiniowanej w klasie Wychowawca:
    cout << "Dane nauczyciela wychowawcy: " << wychowawca.zwrocDane() << endl;
    /* UWAGA
     * Funkcja zwrocDane() wywoływaną powyżej została zdefiniowana w klasie Wychowawca.
     * Metoda ta przesłoniła metody o tej samej nazwie zdefiniowane w klasach,
     * po których dziedziczy klasa Wychowawca().
     */
}

// Utworzenie obiektu uczen jako instancji klasy Uczen:
Uczen uczen;
// Nadanie wartości zmiennym członkowskim obiektu uczen:
uczen.imie = "Adam";
uczen.nazwisko = "Nowak";
uczen.klasa = "3B";
// Prezentacja danych ucznia:
cout << "Dane ucznia: " << uczen.zwrocDane() << endl;

return 0;
}

```

W programie zademonstrowano różne typy dziedziczenia, które zachodzą pomiędzy klasami wchodząymi w skład łańcucha dziedziczenia.

Na przykład klasa bazowa `Osoba` ma dwie klasy pochodne: `Pracownik` i `Uczeń`. Tym samym pomiędzy tymi klasami zachodzi relacja dziedziczenia hierarchicznego, które cechuje się tym, że dana klasa bazowa ma wiele klas pochodnych.

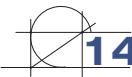
Klasy `Nauczyciel`, `Wychowawca` i `Uczeń` dziedziczą po większej liczbie klas bazowych. Klasa `Nauczyciel` dziedziczy po klasach `Pracownik` i `Przedmiot`, klasa `Wychowawca` — po klasach `Nauczyciel` i `Klasa`, a klasa `Uczeń` — po klasach `Osoba` i `Klasa`. W tym przypadku mamy zatem do czynienia z dziedziczeniem wielokrotnym, którego cechą charakterystyczną jest to, że dana klasa pochodna może dziedziczyć po większej (niż jedna) liczbie klas bazowych.

Klasa `Wychowawca` jest „prawnukiem” klasy `Osoba`, klasa `Nauczyciel` — „wnukiem”, a klasa `Pracownik` — „synem”. Wynika z tego, że relacje dziedziczenia, które łączą wymienione klasy, definiują łańcuch dziedziczenia określający dziedziczenie wielopoziomowe.

Uwzględniając to, że w łańcuchu dziedziczenia obejmującym wszystkie klasy zdefiniowane w programie występują opisane powyżej różne relacje dziedziczenia, możemy powiedzieć, że mamy tutaj do czynienia z dziedziczeniem mieszanym.

Ćwiczenie 14.3

Zmodyfikuj program z przykładu 14.3 — uwzględnij hermetyzację i ukrywanie danych we wszystkich klasach wchodzących w skład łańcucha dziedziczenia. Dostęp do ukrytych zmiennych członkowskich zrealizuj za pomocą zdefiniowanych samodzielnie metod dostępowych (setterów i getterów).



14.3. Dziedziczenie a konstruktory

Pomimo że klasa pochodna nie dziedziczy konstruktorów po klasie bazowej, przy tworzeniu obiektu klasy pochodnej — czyli przy wywołaniu konstruktora tej klasy — wywoływany jest automatycznie konstruktor jej klasy bazowej. Przy tym, jeśli w definicji konstruktora klasy pochodnej nie określono inaczej, zostanie wywołany konstruktor domyślny klasy bazowej. To samo dotyczy wywołania destruktora klasy pochodnej, które skutkuje wywołaniem konstruktora klasy bazowej.

Przykład 14.4

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

// Definicja klasy bazowej Pracownik:
class Pracownik {
public:
    string imie;
```

```

string nazwisko;
// Definicje konstruktorów:
Pracownik() {}; // konstruktor domyślny
Pracownik(string pImie, string pNazwisko) { // konstruktor z parametrami
    imie = pImie;
    nazwisko = pNazwisko;
}

string zwrocDane() {
    return imie + " " + nazwisko;
}
};

// Definicja klasy pochodnej Nauczyciel:
class Nauczyciel: public Pracownik {
public:
    string przedmiot;
// DEFINICJE KONSTRUKTORÓW:
Nauczyciel() {}; // konstruktor domyślny
/* UWAGA
* Przy wywołaniu konstruktora domyślnego zdefiniowanego powyżej zostanie wywołany konstruktor domyślny
* klasy bazowej, ponieważ nie zdecydowano inaczej.
*/
Nauczyciel(string pImie, string pNazwisko, string pPrzedmiot) { // konstruktor
// z parametrami
    imie = pImie;
    nazwisko = pNazwisko;
    przedmiot = pPrzedmiot;
}
/* UWAGA
* Przy wywołaniu konstruktora parametrycznego zdefiniowanego powyżej zostanie wywołany konstruktor
* domyślny klasy bazowej, ponieważ nie zdecydowano inaczej.
*/
// Definicja konstruktora parametrycznego z określeniem konstruktora klasy bazowej:
Nauczyciel(string pImie, string pNazwisko) : Pracownik(pImie, pNazwisko) {}
// konstruktor z parametrami
/* UWAGA
* Przy wywołaniu konstruktora parametrycznego zdefiniowanego powyżej zostanie wywołany określony —
* zdefiniowany konstruktor klasy bazowej.
*/
string zwrocDane() {
    return imie + " " + nazwisko + ", przedmiot: " + przedmiot;
}
};

};

```

```

int main() {
    // Utworzenie obiektu nauczyciel jako instancji klasy Nauczyciel:
    Nauczyciel nauczyciel1;
    /* UWAGA
     * Przy tworzeniu obiektu nauczyciel1 z wykorzystaniem konstruktora domyślnego zostaje wywołany
     * automatycznie konstruktor domyślny (bezparametrowy) klasy bazowej Pracownik.
     */
    // Nadanie wartości polom obiektu nauczyciel1:
    nauczyciel1.imie = "Jan";
    nauczyciel1.nazwisko = "Kowalski";
    nauczyciel1.przedmiot = "język angielski";
    // Wywołanie metody zwrocDane() zdefiniowanej w klasie pochodnej Nauczyciel:
    cout << "Dane nauczyciela: " << nauczyciel1.zwrocDane() << endl;

    // Utworzenie obiektu nauczyciel2 klasy Nauczyciel przez użycie konstruktora z parametrami:
    Nauczyciel nauczyciel2("Adam", "Nowak", "język niemiecki");
    /* UWAGA
     * Przy tworzeniu obiektu nauczyciel2 z użyciem konstruktora z parametrami zostanie wywołany automatycznie
     * konstruktor domyślny (bezparametrowy) klasy bazowej Pracownik.
     */
    cout << "Dane nauczyciela: " << nauczyciel2.zwrocDane() << endl;

    // Utworzenie obiektu nauczyciel3 klasy Nauczyciel:
    Nauczyciel nauczyciel3 = Nauczyciel("Jan", "Nowak");
    /* UWAGA
     * Przy tworzeniu obiektu nauczyciel3 z użyciem konstruktora z parametrami zostanie wywołany zdefiniowany
     * konstruktor parametryczny klasy bazowej Pracownik.
     */
    nauczyciel3.przedmiot = "język francuski";
    cout << "Dane nauczyciela: " << nauczyciel3.zwrocDane() << endl;

    return 0;
}

```

Klasa `Nauczyciel` została zdefiniowana na podstawie klasy `Pracownik` z zastosowaniem relacji dziedziczenia. Klasa `Pracownik` jest klasą bazową, a klasa `Nauczyciel` — pochodną. W klasie bazowej `Pracownik` zdefiniowano dwa konstruktory: domyślny i parametryczny. Klasa pochodna `Nauczyciel` z kolei zawiera definicje trzech konstruktorów: domyślnego i dwóch parametrycznych.

Utworzenie obiektu `nauczyciel1` jest związane z niejawnym wywołaniem konstruktora domyślnego klasy pochodnej, co skutkuje automatycznym wywołaniem konstruktora domyślnego klasy bazowej. Wynika to z definicji konstruktora domyślnego klasy pochodnej, w której nie określono żadnego konstruktora klasy bazowej.

Obiekt `nauczyciel2` zaś został utworzony przez niejawne wywołanie konstruktora parametrycznego klasy pochodnej, co również powoduje automatyczne wywołanie konstruktora domyślnego klasy bazowej.

W przypadku obiektu `nauczyciel3`, utworzonego przez wywołanie konstruktora parametrycznego klasy pochodnej w sposób jawnym, wywołany zostanie zdefiniowany konstruktor parametryczny klasy bazowej. Wynika to z definicji tego konstruktora w klasie pochodnej, w której ustalono, jaki konstruktor klasy bazowej ma być wywołany podczas tworzenia obiektu będącego instancją klasy pochodnej.

Ćwiczenie 14.4

Zmodyfikuj program zawarty w przykładzie 14.4 — uzupełnij ciała konstruktorów domyślnych klas `Pracownik` i `Nauczyciel` o instrukcje pozwalające na inicjalizację ich zmiennych członkowskich. Zaimplementuj hermetyzację — ukrycie danych przechowywanych w zmiennych członkowskich obu klas.



14.4. Pytania i zadania kontrolne

14.4.1. Pytania

- 1.** Na czym polega mechanizm dziedziczenia? Jakie są korzyści wynikające z zastosowania dziedziczenia?
- 2.** Wyjaśnij znaczenie pojęć: klasa bazowa, klasa pochodna.
- 3.** Co to jest łańcuch dziedziczenia?
- 4.** Wymień i scharakteryzuj tryby (rodzaje) dziedziczenia uwzględniające tryb dostępu do elementów członkowskich klasy pochodnej.
- 5.** Na czym polega dziedziczenie hierarchiczne, a na czym dziedziczenie wielokrotne?
- 6.** Czy klasa pochodna dziedziczy konstruktory po klasie bazowej? Czy wywołanie konstruktora klasy pochodnej może powodować wywołanie konstruktora klasy bazowej?

14.4.2. Zadania

- 1.** Napisz program pozwalający przetwarzać dane komputerów. Uwzględnij komputery stacjonarne PC oraz laptopy. Przyjmij następujące założenia:
 - cechy wspólne każdego komputera to marka i model,
 - cecha charakterystyczna komputera PC to rodzaj obudowy, np. slim,
 - cechy charakterystyczne laptopa to długość przekątnej ekranu i kolor obudowy.

Wykonaj program zgodnie z zasadami programowania obiektowego — wykorzystaj zdefiniowane samodzielnie klasy powiązane ze sobą relacjami dziedziczenia. Zastosuj dziedziczenie hierarchiczne w trybie publicznym. Niech w celach testowych będzie

możliwe wprowadzenie danych określonego laptopa z klawiatury, zapamiętanie ich w zmiennych członkowskich obiektu, a następnie wyświetlenie kontrolnie na ekranie monitora.

2. Zrób tak jak w zadaniu 1., z tym że dodatkowo uwzględnij hermetyzację — ukrywanie danych komputerów. Dostęp do ukrytych danych zrealizuj za pomocą publicznych metod dostępowych (setterów i getterów).
3. Napisz program pozwalający przetwarzać dane pracowników w szkole. Uwzględnij następujące stanowiska: dyrektor, nauczyciel, wychowawca, sekretarka. Wykonaj program zgodnie z zasadami programowania obiektowego — wykorzystaj zdefiniowane samodzielnie klasy powiązane ze sobą relacjami dziedziczenia. Niech w celach testowych będzie możliwe wprowadzenie danych określonego wychowawcy z klawiatury, zapamiętanie ich w zmiennych członkowskich obiektu, a następnie wyświetlenie kontrolnie na ekranie monitora.
4. Zrób tak jak w zadaniu 3., z tym że dodatkowo uwzględnij hermetyzację — ukrywanie danych pracowników. Dostęp do ukrytych danych zrealizuj za pomocą publicznych setterów i getterów.
5. Napisz program pozwalający przetwarzać dane samochodów w przedsiębiorstwie transportowym. Załóż, że firma ma na stanie samochody ciężarowe i autobusy. Każdy pojazd można opisać za pomocą cech wspólnych: marki, modelu, roku produkcji. Oprócz wymienionych danych samochody ciężarowe charakteryzują się unikatowym przeznaczeniem — wywrotka, chłodnia i cysterna, a autobusy — liczbą miejsc stojących i siedzących. Wykorzystaj mechanizm dziedziczenia. W każdej klasie wchodzącej w skład łańcucha dziedziczenia zdefiniuj konstruktor domyślny i parametryczny. Niech w celach testowych będzie możliwe wprowadzenie danych określonego autobusu z klawiatury, zapamiętanie ich w zmiennych członkowskich obiektu, a następnie wyświetlenie ich kontrolnie na ekranie monitora. Obiekt odpowiadający przetwarzanemu autobusowi utwórz przez wywołanie konstruktora parametrycznego zdefiniowanego w klasie pochodnej, który będzie wywoływał konstruktor parametryczny zdefiniowany w klasie bazowej.
6. Zrób tak jak w zadaniu 5., z tym że dodatkowo uwzględnij hermetyzację — ukrywanie danych samochodów. Dostęp do ukrytych danych zrealizuj za pomocą publicznych setterów i getterów.

15

Polimorfizm

Polimorfizm (ang. *polymorphism*) oznacza wielopostaciowość. W szczególności dotyczy ona funkcji członkowskich (metod) definiowanych w klasach. W języku C++ można implementować dwa rodzaje polimorfizmu:

- polimorfizm statyczny,
- polimorfizm dynamiczny.

15.1. Polimorfizm statyczny

Polimorfizm statyczny (ang. *static polymorphism*) zachodzi w czasie komplikacji programu. Dlatego też często nazywa się go **polimorfizmem w czasie komplikacji** (ang. *compile-time polymorphism*).

Stosowanie polimorfizmu statycznego może być związane:

- z przeciążaniem metod,
- z przesłanianiem metod.

15.1.1. Przeciążanie metod

Polimorfizm statyczny (ang. *static polymorphism*) może wynikać z wykorzystania mechanizmu **przeciążania metod** (ang. *methods overloading*) zdefiniowanych w obrębie danej klasy. Mianowicie jeśli w danej klasie zostanie zdefiniowanych wiele metod o tej samej nazwie, ale różniących się od siebie liczbą i/lub typem parametrów (ewentualnie typem zwracanej wartości), to mamy do czynienia z przeciążeniem tych metod.

Wywołanie przeciążonej metody powoduje, że kompilator już na etapie komplikacji programu porównuje to wywołanie z definicjami wszystkich przeciążonych metod zawartych w klasie. Mówiąc dokładniej, kompilator porównuje liczbę i typy argumentów wywołania funkcji z liczbą i typami parametrów w poszczególnych definicjach funkcji przeciążonych. To samo dotyczy typu wartości zwracanej przez metody. Na tej podstawie kompilator wybiera najbardziej odpowiednią funkcję do wywołania.

Ze względu na to, że działania kompilatora w opisanej sytuacji są realizowane na etapie kompilacji programu, polimorfizm statyczny często jest nazywany **polimorfizmem w czasie kompilacji** (ang. *compile-time polymorphism*).

Wielu programistów polimorfizm statyczny utożsamia z tzw. **wczesnym wiązaniem** (ang. *early binding*). Przy czym wspomniane **wiązanie** (ang. *binding*) należy rozumieć jako powiązanie wywołania metody z treścią (definicją) metody. Ujmując to dokładniej, wiązanie oznacza skojarzenie nazwy (identyfikatora) wywoływanej metody z adresem bloku pamięci operacyjnej, w którym ta metoda jest przechowywana. Technicznie rzecz biorąc, skutkuje to zastąpieniem wywołania metody instrukcją języka maszynowego wywołującą skok w pamięci do adresu odpowiedniej metody.

UWAGA

Wczesne wiązanie często jest nazywane również **wiązaniem statycznym** (ang. *static binding*) lub **wiązaniem w czasie kompilacji** (ang. *compile-time binding*).

Przykład 15.1

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Deklaracje zmiennych członkowskich:
    string imie, nazwisko, stanowisko;
    // Deklaracje (prototypy) przeciążonych metod zwrocDane():
    string zwrocDane();
    void zwrocDane(string&, string&, string&);
    /* UWAGA
     * Funkcje członkowskie zadeklarowane powyżej mają taką samą nazwę, ale różnią się parametrami
     * (liczbą i/lub typem) i/lub typem zwracanej wartości.
     */
};

// Definicje funkcji członkowskich zwrocDane() z klasy Pracownik:
string Pracownik::zwrocDane() {
    return imie + " " + nazwisko + ", stanowisko: " + stanowisko;
}
void Pracownik::zwrocDane(string &pImie, string &pNazwisko,
                           string &pStanowisko) {
    pImie = imie;
    pNazwisko = nazwisko;
    pStanowisko = stanowisko;
}
```

```

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy Pracownik:
    Pracownik pracownik;
    // Nadanie wartości zmiennym członkowskim obiektu pracownik:
    pracownik.imie = "Jan";
    pracownik.nazwisko = "Kowalski";
    pracownik.stanowisko = "nauczyciel";

    // Wywołanie przeciążonej funkcji członkowskiej (metody) zwrocDane():
    cout << "Dane pracownika: " << pracownik.zwrocDane() << endl;
    /* UWAGA
     * Kompilator już na etapie komplikacji zdecyduje, która z funkcji zwrocDane() zdefiniowanych w klasie Pracownik
     * zostanie w tym miejscu wywołana.
     */
    string imie, nazwisko, stanowisko; // zmienne pomocnicze
    // Wywołanie przeciążonej funkcji członkowskiej zwrocDane():
    pracownik.zwrocDane(imie, nazwisko, stanowisko);
    /* UWAGA
     * Kompilator już na etapie komplikacji zdecyduje, która ze zdefiniowanych funkcji zwrocDane() zostanie
     * w tym miejscu wywołana.
     */
    cout << "Dane pracownika: ";
    cout << imie + " " + nazwisko + ", stanowisko: " + stanowisko << endl;

    return 0;
}

```

W programie zdefiniowano klasę `Pracownik`, która zawiera m.in. definicje przeciążonych metod o nazwie `zwrocDane`. Pomimo że metody te mają identyczną nazwę, różnią się zarówno liczbą, jak i typem parametrów, a ponadto typami wartości zwracanych.

W programie głównym metoda `zwrocDane()` została wywołana dwukrotnie. Kompilator już na etapie komplikacji programu porównuje postacie tych wywołań z definicjami metod, a w szczególności argumenty tych wywołań z parametrami formalnymi, jak też typy zwracanych przez nie wartości. Po przeprowadzeniu analizy komplikator wybiera tę postać metody `zwrocDane()`, która najbardziej pasuje do danego wywołania.

Mamy tutaj do czynienia z wielopostaciowością metody `zwrocDane()`. Wielopostaciowość ta ma miejsce w czasie komplikacji programu.

Podsumowując — w zaprezentowanym programie zachodzi polimorfizm statyczny.

Ćwiczenie 15.1

Zmodyfikuj program z przykładu 15.1 — zaimplementuj w klasie `Pracownik` hermetyzację (ukrycie) danych.

15.1.2. Przesłanianie metod

Stosowanie polimorfizmu statycznego w programowaniu obiektowym może być również związane z mechanizmem **przesłaniania metod** (ang. *method overriding*) w ramach łańcucha dziedziczenia klas (ang. *class inheritance chain*).

Najprostszy sposób implementacji efektu przesłaniania metod polega na tym, że zarówno w klasie bazowej, jak i w jej klasie pochodnej definiuje się funkcje członkowskie (metody) o tych samych nazwach. Także liczba i typy parametrów formalnych oraz typy wartości zwracanych przez te metody są takie same.

Jeśli założyć, że wywołanie jednej z metod o opisanych powyżej cechach jest skojarzone w sposób bezpośredni z obiektem będącym instancją klasy pochodnej, tj. przy wykorzystaniu identyfikatora tego obiektu oraz operatora dostępu `..`, metoda zdefiniowana w klasie pochodnej przeszła (ang. *overrides*) metodę o tej samej nazwie zdefiniowaną w jej klasie bazowej. Patrząc na to samo zjawisko z drugiej strony — metoda zdefiniowana w klasie bazowej zostaje przesłonięta (ang. *overridden*) przez metodę o tej samej nazwie zdefiniowaną w jej klasie pochodnej.

Dopasowanie odpowiedniej definicji metody do postaci jej wywołania (a dokładniej: do wyrażenia zawierającego jej wywołanie) polega w opisywanej sytuacji na podjęciu decyzji, która metoda ma zostać wykonana — czy ta zdefiniowana w klasie bazowej, czy też metoda o tej samej nazwie zdefiniowana w klasie pochodnej. Proces ten jest realizowany w całości przed wykonaniem programu — w czasie jego komplikacji. Tym samym mamy tu do czynienia z **wczesnym wiązaniem** (ang. *early binding*) wywołania metody z jej treścią (definicja). To zaś oznacza, że zachodzi polimorfizm statyczny, tj. polimorfizm w czasie komplikacji (ang. *compile-time polymorphism*).

UWAGA

Efekt przesłaniania metod występuje również w polimorfizmie dynamicznym, związanym z wykorzystaniem metod wirtualnych. Tematyka ta została omówiona szczegółowo w podrozdziale 15.2.2.

Przykład 15.2

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

// Definicja klasy bazowej Pracownik:
class Pracownik {
public:
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};


```

```

// Definicja klasy pochodnej Nauczyciel:
class Nauczyciel: public Pracownik {
public:
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};

// Definicja klasy pochodnej Wychowawca:
class Wychowawca: public Nauczyciel {
public:
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};

/* UWAGA
 * W każdej z klas wchodzących w skład łańcucha dziedziczenia zadeklarowano metodę publiczną o takiej samej
 * nazwie oraz identycznych parametrach (liczbie i typie) i typach wartości zwracanych.
 */

// Definicje metod:
void Pracownik::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Pracownik"
        << endl;
}

void Nauczyciel::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Nauczyciel"
        << endl;
}

void Wychowawca::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Wychowawca"
        << endl;
}

/* UWAGA
 * Implementacja i funkcjonalność każdej ze zdefiniowanych metod zwrocDane() jest w ogólności dowolna.
 */

int main() {
    // Utworzenie obiektu pracownik1 jako instancji klasy Pracownik:
    Pracownik pracownik1;
    // Wywołanie metody zwrocDane():
    pracownik1.zwrocDane();

    /* UWAGA
     * Powyższe wywołanie dotyczy metody zwrocDane() zdefiniowanej w klasie bazowej Pracownik.
     */

    // Utworzenie obiektu pracownik2 jako instancji klasy Nauczyciel:
    Nauczyciel pracownik2;
    // Wywołanie metody zwrocDane():
}

```

```

pracownik2.zwrocDane();
/* UWAGA
 * Powyższe wywołanie dotyczy metody zwrocDane() zdefiniowanej w klasie pochodnej Nauczyciel.
 */

// Utworzenie obiektu pracownik3 jako instancji klasy Wychowawca:
Wychowawca pracownik3;
// Wywołanie metody zwrocDane():
pracownik3.zwrocDane(); // zostanie wywołana metoda zdefiniowana w klasie pochodnej Wychowawca

return 0;
}

```

W programie zdefiniowano trzy klasy: Pracownik, Nauczyciel i Wychowawca, pomiędzy którymi określono relacje dziedziczenia. W każdej z wymienionych klas zdefiniowano metodę o taka samej nazwie — zwrocDane. Zarówno liczba, jak i typy parametrów metod zwrocDane() zdefiniowanych w każdej z wymienionych powyżej klas są takie same — tj. brak parametrów. To samo dotyczy wartości zwracanych przez te metody na zewnątrz — brak wartości zwracanych (void).

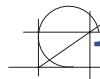
Metoda zwrocDane() została wywołana w programie trzykrotnie. Pierwsze ze wspomnianych wywołań dotyczy obiektu pracownik1 jako instancji klasy bazowej Pracownik. Drugie wywołanie jest związane z obiektem pracownik2 należącym do klasy pochodnej Nauczyciel, a trzecie — z obiektem pracownik3 klasy Wychowawca. W każdym z wymienionych przypadków wywoływana jest metoda zdefiniowana w klasie, której instancję stanowi powiązany z nią obiekt. Zatem wywołanie pracownik1.zwrocDane() powoduje wykonanie metody zdefiniowanej w klasie bazowej Pracownik, wywołanie pracownik2.zwrocDane() — wykonanie metody z klasy pochodnej Nauczyciel, a wywołanie pracownik3.zwrocDane() — wykonanie metody z klasy pochodnej Wychowawca.

Mamy tutaj do czynienia z efektem przesłanania metod w łańcuchu dziedziczenia. Metoda zwrocDane() zdefiniowana w klasie Wychowawca przesyła metodę o tej samej nazwie zdefiniowaną w jej klasie bazowej, tj. klasie Nauczyciel. Z kolei metoda zwrocDane() zdefiniowana w klasie Nauczyciel przesyła metodę zwrocDane() z jej klasie bazowej, czyli z klasy Pracownik.

Można w tym przypadku mówić o wczesnym wiązaniu (ang. *early binding*) wywołań metody zwrocDane() z jej definicjami w poszczególnych klasach wchodzących w skład łańcucha dziedziczenia. Proces ten zachodzi w czasie komplikacji programu. To z kolei oznacza, że w odniesieniu do wymienionej metody zachodzi polimorfizm statyczny, tj. polimorfizm w czasie komplikacji (ang. *compile-time polymorphism*). Metoda zwrocDane() jest wielopostaciowa (na etapie komplikacji programu).

Ćwiczenie 15.2

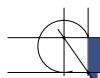
Zmodyfikuj program z przykładu 15.2 — uzupełnij klasy `Pracownik`, `Nauczyciel` i `Wychowawca` o odpowiednie zmienne członkowskie (`dane`) i konstruktory. Metody `zwrocDane()` zdefiniowane w każdej z klas łańcucha dziedziczenia uzupełnij o kod pozwalający na wyświetlenie na ekranie monitora danych obiektu należącego do danej klasy.



15.2. Polimorfizm dynamiczny

Polimorfizm dynamiczny (ang. *dynamic polymorphism*) to polimorfizm, który zachodzi w trakcie wykonywania programu — już po zakończeniu procesu jego komplikacji. Dlatego też często jest on nazywany **polimorfizmem w czasie wykonywania programu** (ang. *runtime polymorphism*).

Jak wspomniano w poprzednim podrozdziale, zjawisko polimorfizmu w programowaniu obiektowym jest utożsamiane ze sposobem wiązania (ang. *binding*) wywołania metody z jej definicją (treścią). Biorąc pod uwagę czas, w którym ten proces jest realizowany, termin **późne wiązanie** (ang. *late binding*) oznacza, że zachodzi on w czasie wykonywania programu — w odróżnieniu od wiązania wcześniego, które następuje na etapie komplikacji programu.



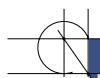
UWAGA

Późne wiązanie wywołania metody z jej definicją jest również nazywane **wiązaniem w czasie wykonywania programu** (ang. *runtime binding*) lub **wiązaniem dynamicznym** (ang. *dynamic binding*).

Stosowanie polimorfizmu dynamicznego wiąże się z wykorzystaniem obiektów należących do różnych klas wchodzących w skład łańcucha dziedziczenia, do których dostęp został zrealizowany za pomocą wskaźników.

15.2.1. Wskaźniki w mechanizmie dziedziczenia

Dostęp do obiektów, które są instancjami klas wchodzących w skład łańcucha dziedziczenia, można zrealizować za pomocą wskaźników. Dotyczy to zarówno obiektów należących do klasy bazowej (lub klas bazowych), jak i obiektów klas pochodnych.



UWAGA

Tematyka wskaźników do obiektów — bez uwzględnienia mechanizmu dziedziczenia — została omówiona w podrozdziale 11.7.

Interesująca sytuacja występuje wówczas, gdy w celu uzyskania dostępu do obiektów, które wchodzą w skład łańcucha dziedziczenia, wykorzystuje się wskaźnik należący do typu

statycznego określonego przez klasę bazową. **Typ statyczny wskaźnika** (ang. *static type of pointer, pointer static type*) określa się w jego deklaracji. Na przykład deklaracja `Pracownik *w_pracownik;` oznacza, że typem statycznym wskaźnika `w_pracownik` jest klasa `Pracownik`.

Za pośrednictwem takiego wskaźnika można również uzyskać dostęp nie tylko do obiektów należących do klasy bazowej, ale też do obiektów będących instancjami jej klas pochodnych. Wynika to z jednej z kluczowych cech programowania obiektowego w języku C++, mianowicie z tego, że typ wskaźnika na obiekt klasy bazowej jest kompatybilny (czyli zgodny pod względem typu) z typem wskaźnika na obiekt klasy pochodnej.

Przykład 15.3

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

// Definicja klasy bazowej Pracownik:
class Pracownik {
public:
    string imie, nazwisko;
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};

// Definicja klasy pochodnej Nauczyciel:
class Nauczyciel: public Pracownik {
public:
    string przedmiot;
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};

// Definicja klasy pochodnej Wychowawca:
class Wychowawca: public Nauczyciel {
public:
    string klasa;
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};

/* UWAGA
 * W każdej z klas wchodzących w skład łańcucha dziedziczenia zadeklarowano metodę zwrocDane() —
 * o takiej samej nazwie oraz identycznych parametrach (liczbie i typie) i typie zwracanej wartości.
 */

// Definicje metod instancjnych zwrocDane():
void Pracownik::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Pracownik"
        << endl;
    cout << "Imię: " << imie << endl;
```

```

    cout << "Nazwisko: " << nazwisko << endl;
}
void Nauczyciel::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Nauczyciel"
        << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
    cout << "Przedmiot: " << przedmiot << endl;
}
void Wychowawca::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Wychowawca"
        << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
    cout << "Przedmiot: " << przedmiot << endl;
    cout << "Klasa:" << klasa << endl;
}
/* UWAGA
 * Implementacja (i co za tym idzie — funkcjonalność) każdej ze zdefiniowanych metod zwrocDane() jest dowolna.
 */
int main() {
    // Deklaracja wskaźnika w_pracownik typu statycznego Pracownik:
    Pracownik *w_pracownik;
    /* UWAGA
     * Zmienna w_pracownik jest zmienną statyczną, która może wskazywać (z definicji) na obiekty typu bazowego
     * Pracownik oraz obiekty klas pochodnych klasy Pracownik, czyli obiekty klas Nauczyciel i Wychowawca.
     */
    // Utworzenie obiektu pracownik1 jako instancji klasy Pracownik:
    Pracownik pracownik1;
    // Przypisanie wskaźnikowi w_pracownik adresu obiektu pracownik1:
    w_pracownik = &pracownik1;
    // Nadanie wartości zmiennym członkowskim obiektu pracownik1:
    w_pracownik->imie = "Jan";
    w_pracownik->nazwisko = "Kowalski";
    // Wywołanie metody zwrocDane():
    w_pracownik->zwrocDane(); // zostaje wywołana metoda z klasy bazowej Pracownik
    cout << endl;

    // Utworzenie obiektu pracownik2 jako instancji klasy Nauczyciel:
    Nauczyciel pracownik2;
    w_pracownik = &pracownik2;
    w_pracownik->imie = "Adam";
    w_pracownik->nazwisko = "Nowak";
}

```

```

/* UWAGA
 * Użycie dereferencji w_pracownik-> pozwala jedynie uzyskać dostęp do elementów członkowskich klasy
 * Nauczyciel odziedziczonych po klasie bazowej Pracownik — czyli do zmiennych imie i nazwisko.
 * Dostęp do elementów członkowskich zdefiniowanych bezpośrednio w klasie pochodnej Nauczyciel nie jest
 * możliwy. Tym samym próba wykonania instrukcji przypisania: w_pracownik->przedmiot = „Informatyka”;
 * zakończy się komunikatem o błędzie i informacją, że klasa Pracownik nie zawiera elementu członkowskiego
 * przedmiot.
 */
// Wywołanie metody zwrocDane():
w_pracownik->zwrocDane(); // zostaje wywołana metoda odziedziczona po klasie bazowej Pracownik
cout << endl;

// Utworzenie obiektu pracownik3 jako instancji klasy Wychowawca:
Wychowawca pracownik3;
w_pracownik = &pracownik3;
w_pracownik->imie = "Jan";
w_pracownik->nazwisko = "Polski";
// w_pracownik->przedmiot = „Informatyka”; INSTRUKCJA BŁĘDNA!
// w_pracownik->klasa = „3A”; INSTRUKCJA BŁĘDNA!

// Wywołanie metody zwrocDane():
w_pracownik->zwrocDane(); // zostaje wywołana metoda odziedziczona po klasie bazowej Pracownik

return 0;
}

```

Program zawarty w rozpatrywanym przykładzie to modyfikacja programu z przykładu 15.2. Modyfikacja polega na tym, że dostęp do obiektów: pracownik1, pracownik2 i pracownik3 uzyskano za pomocą jednego wskaźnika, w_pracownik, oraz operatora dostępu ->.

W szczególności dostęp do obiektów: pracownik1, pracownik2 i pracownik3, będących instancjami klas, odpowiednio, Pracownik, Nauczyciel, Wychowawca, które wchodzą w skład łańcucha dziedziczenia, uzyskano za pomocą jednego wskaźnika, w_pracownik. Wskaźnik w_pracownik został zdefiniowany jako Pracownik *w_pracownik. Tym samym typem statycznym wskaźnika w_pracownik jest klasa bazowa Pracownik. Wskaźnik w_pracownik może wskazywać (z definicji) zarówno na obiekty należące do klasy bazowej Pracownik, jak i na obiekty jej klas pochodnych, czyli Nauczyciel i Wychowawca.

Metoda zwrocDane(), zdefiniowana w każdej z klas wchodzących w skład łańcucha dziedziczenia, została wywołana w programie trzykrotnie. Pierwsze ze wspomnianych wywołań dotyczy obiektu pracownik1, będącego instancją klasy bazowej Pracownik. Drugie wywołanie jest związane z obiektem pracownik2, należącym do klasy pochodnej Nauczyciel, a trzecie — z obiektem pracownik3 klasy Wychowawca. W każdym z wymienionych przypadków dostęp do danego obiektu zrealizowano za pomocą wskaźnika w_pracownik typu statycznego Pracownik oraz operatora strzałkowego ->, czyli dereferencji w_pracownik->.

Skutkiem każdego ze wspomnianych wywołań jest wykonanie metody zwrocDane() zdefiniowanej w klasie bazowej Pracownik (a nie w klasach pochodnych), czyli wywołanie

Pracownik::zwrocDane(). Dzieje się tak dlatego, że wskaźnik `w_pracownik` stanowi (z definicji) referencję do typu bazowego `Pracownik`. Tym samym za jego pośrednictwem można uzyskać dostęp jedynie do elementów członkowskich klas pochodnych odziedziczonych po klasie bazowej — tj. odziedziczonych zmiennych członkowskich `imie` i `nazwisko` oraz odziedziczonej metody `zwrocDane()`. Dostęp do elementów członkowskich zdefiniowanych bezpośrednio w klasach pochodnych `Nauczyciel` i `Wychowawca` za pomocą dereferencji `w_pracownik->` nie jest możliwy.

Biorąc pod uwagę powyższe, w programie nie występuje przesłanianie metody `zwrocDane()`. Metoda ta jest metodą jednopostaciową, a nie wielopostaciową — jeśli spojrzeć z perspektywy jej wywołania za pośrednictwem wskaźnika `w_pracownik`. Tym samym żaden polimorfizm tutaj nie zachodzi.

Ćwiczenie 15.3

Zmodyfikuj program z przykładu 15.3 — zamiast dziedziczenia pojedynczego wielopoziomowego zastosuj dziedziczenie wielokrotne. W tym celu uzupełnij łańcuch dziedziczenia o zdefiniowane samodzielnie klasy `Przedmiot` i `Klasa`, zawierające odpowiednie zmienne członkowskie i/lub metody, oraz zmień relacje dziedziczenia pomiędzy klasami. Funkcjonalność zmodyfikowanego programu powinna być taka sama jak funkcjonalność programu z przykładu 15.3.

15.2.2. Metody wirtualne

Metody wirtualne (ang. *virtual methods*) to jedno z najważniejszych narzędzi polimorfizmu, czyli wielopostaciowości. Użycie metod wirtualnych jest związane z **polimorfizmem dynamicznym** (ang. *dynamic polymorphism*), a więc polimorfizmem zachodzącym **w czasie wykonywania** (ang. *runtime*) programu.

Metoda wirtualna to funkcja członkowska zdefiniowana w klasie bazowej, której deklaracja jest poprzedzona słowem kluczowym `virtual`. Zadeklarowanie metody w klasie bazowej jako wirtualnej pozwala na jej późniejsze „przedefiniowanie” (ang. *redefining*) w klasach pochodnych. Przy tym zachowane zostają charakterystyczne właściwości jej wywołań za pośrednictwem referencji do klasy bazowej.

W praktyce wspomniane powyżej odwołania są realizowane za pomocą wskaźnika należącego do typu statycznego klasy bazowej. Jednakże **typ dynamiczny wskaźnika** (ang. *dynamic type of pointer*, *pointer dynamic type*), który określa typ obiektu faktycznie wskazywanego przez wskaźnik, może się zmieniać w trakcie wykonywania programu.

UWAGA

Powiązanie wskaźnika z jego typem dynamicznym jest nazywane wiązaniem dynamicznym (ang. *dynamic linkage*) lub późnym wiązaniem (ang. *late binding*).

Wybór, która z metod zostanie wykonana jako skutek wywołania za pośrednictwem referencji do klasy bazowej — czy metoda zdefiniowana w klasie bazowej, czy któraś z metod o tej samej nazwie z klasy pochodnej — zależy od tego, czy metoda zdefiniowana w klasie bazowej została zadeklarowana jako wirtualna.

Stosowanie metod wirtualnych w programowaniu obiektowym jest ściśle powiązane z mechanizmem dziedziczenia. Uproszczone założenia umożliwiające wykorzystanie polimorfizmu dynamicznego wynikającego z użycia metod wirtualnych są następujące:

- W klasie bazowej zdefiniowano metodę wirtualną o określonych: nazwie, parametrach oraz typie wartości zwracanej.
- W klasach pochodnych zdefiniowano metody o nazwie identycznej z nazwą metody wirtualnej w klasie bazowej oraz takich samych parametrach (liczbie i typie) i typach zwracanych.
- Wywołania metod (o takiej samej nazwie jak nazwa funkcji wirtualnej w klasie bazowej) dotyczą obiektów będących instancjami różnych klas, które wchodzą w skład łańcucha dziedziczenia.
- Dostęp do obiektów należących do klas bazowej i pochodnych, połączonych ze sobą relacją dziedziczenia, jest realizowany przy użyciu wskaźnika należącego do typu statycznego klasy bazowej. Jednocześnie typ dynamiczny tego wskaźnika może się zmieniać w trakcie wykonywania programu.

Biorąc pod uwagę przedstawione powyżej założenia, wybór metody do wykonania zależy od typu dynamicznego wskaźnika, za którego pośrednictwem jest realizowany dostęp do danego obiektu. W rezultacie wybrana zostanie metoda zdefiniowana w klasie zgodnej z typem dynamicznym przetwarzanego obiektu. Metoda wirtualna z klasy bazowej zostaje wówczas **przesłonięta** (ang. *overridden*) przez metodę zdefiniowaną w tej klasie pochodnej, na której instancję (obiekt) faktycznie wskazuje wskaźnik.

Przykład 15.4

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

// Definicja klasy bazowej Pracownik:
class Pracownik {
public:
    // Prototyp metody wirtualnej zwrocDane():
    virtual void zwrocDane();
};

// Definicja klasy pochodnej Nauczyciel:
class Nauczyciel: public Pracownik {
public:
    // Prototyp metody zwrocDane():
```

```

    void zwrocDane();
};

// Definicja klasy pochodnej Wychowawca:
class Wychowawca: public Nauczyciel {
public:
    // Prototyp metody zwrocDane():
    void zwrocDane();
};

/* UWAGA
 * W każdej z klas wchodzących w skład łańcucha dziedziczenia zadeklarowano metodę publiczną o takiej samej nazwie
 * oraz identycznych parametrach (liczbie i typie) i typie zwracanej wartości.
 * Przy tym metoda zdefiniowana w klasie bazowej Pracownik jest metodą wirtualną.
 */

// Definicje metod zadeklarowanych w klasach Pracownik, Nauczyciel i Wychowawca:
void Pracownik::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Pracownik"
        << endl;
}

void Nauczyciel::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Nauczyciel"
        << endl;
}

void Wychowawca::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Wychowawca"
        << endl;
}

int main() {
    // Deklaracja wskaźnika w_pracownik typu statycznego Pracownik:
    Pracownik *w_pracownik;
    // Utworzenie obiektu pracownik1 jako instancji klasy Pracownik:
    Pracownik pracownik1;
    // Przypisanie wskaźnikowi w_pracownik adresu obiektu pracownik1:
    w_pracownik = &pracownik1;
    // Wywołanie metody zwrocDane():
    w_pracownik->zwrocDane(); // Zostanie wywołana metoda z klasy bazowej Pracownik.

    // Utworzenie obiektu pracownik2 jako instancji klasy Nauczyciel:
    Nauczyciel pracownik2;
    w_pracownik = &pracownik2;
    /* UWAGA
     * Realizacja przypisania w_pracownik = &pracownik2; powoduje zmianę typu dynamicznego wskaźnika
     * w_pracownik z typu Pracownik na typ Nauczyciel. Po wykonaniu przypisania wskaźnik ten wskazuje
     * na obiekt pracownik2 należący do typu Nauczyciel.
     */
}

```

```

// Wywołanie metody zwrocDane():
w_pracownik->zwrocDane();
/* UWAGA
 * Ze względu na to, że typem dynamicznym wskaźnika w_pracownik jest typ Nauczyciel, wywołana
 * zostaje metoda zwrocDane() zdefiniowana w klasie pochodnej Nauczyciel.
 * Tym samym metoda wirtualna zwrocDane() zdefiniowana w klasie bazowej została przesłonięta przez metodę
 * zwrocDane() z klasy pochodnej.
 */
// Utworzenie obiektu pracownik3 jako instancji klasy Wychowawca:
Wychowawca pracownik3;
w_pracownik = &pracownik3; // zmiana typu dynamicznego wskaźnika w_pracownik na typ
// Wychowawca
// Wywołanie metody zwrocDane():
w_pracownik->zwrocDane(); // zostanie wywołana metoda z klasy pochodnej Wychowawca

return 0;
}

```

W klasie bazowej Pracownik zdefiniowano metodę wirtualną zwrocDane(). W klasach pochodnych Nauczyciel i Wychowawca wchodzących w skład łańcucha dziedziczenia zdefiniowano metody o identycznych nazwach, liczbach i typach parametrów oraz typach zwracanych.

Wskaźnik w_pracownik należy do typu statycznego Pracownik, co wynika z jego deklaracji: Pracownik *w_pracownik;. Można go zatem używać do wskazywania zarówno na obiekty należące do klasy bazowej Pracownik, jak i na obiekty będące instancjami klas pochodnych Nauczyciel i Wychowawca.

Typ dynamiczny wskaźnika w_pracownik — czyli typ obiektu, na który faktycznie wskazuje ten wskaźnik — zmienia się w trakcie wykonywania programu. Po wykonaniu przypisania w_pracownik = &pracownik1; typ dynamiczny wskaźnika w_pracownik zostaje ustalony na typ Pracownik, ponieważ obiekt pracownik1 jest instancją klasy Pracownik. Na skutek zaś wykonania instrukcji przypisania, tj. w_pracownik = &pracownik2;, typ dynamiczny wskaźnika w_pracownik zmienia się na typ Nauczyciel. Wynika to z faktu, że obiekt pracownik2 należy do klasy Nauczyciel. Ostatnia zmiana typu dynamicznego wskaźnika w_pracownik następuje po wykonaniu instrukcji w_pracownik = &pracownik3;. Wówczas typem dynamicznym wskaźnika w_pracownik jest Wychowawca.

Wszystkie wywołania metody zwrocDane() są realizowane za pośrednictwem wskaźnika w_pracownik — niezależnie od typu obiektu wskazywanego przez ten wskaźnik — za pomocą dereferencji w_pracownik->.

Jeśli wskaźnik w_pracownik wskazuje na obiekt pracownik1, będący instancją klasy Pracownik, wywoływana jest metoda zwrocDane() zdefiniowana w klasie Pracownik. Dzieje się tak dlatego, że typem dynamicznym wskaźnika w_pracownik jest klasa Pracownik.

Jeśli wskaźnik w_pracownik wskazuje na obiekt pracownik2, będący instancją klasy pochodnej Nauczyciel, wywoływana jest metoda zwrocDane() zdefiniowana w klasie Nauczyciel —

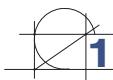
zgodnie z typem dynamicznym wskaźnika, którym jest typ `Nauczyciel`. Tym samym metoda `zwrocDane()` zdefiniowana w klasie pochodnej `Nauczyciel` przesłoniła metodę wirtualną o tej samej nazwie z klasy bazowej `Pracownik`.

Analogiczna sytuacja występuje po zmianie typu dynamicznego wskaźnika `w_pracownik` na typ `Wychowawca`. W tym przypadku wykonywana jest metoda `zwrocDane()` zdefiniowana w klasie pochodnej `Wychowawca`.

Biorąc pod uwagę powyższe rozważania, w programie zachodzi polimorfizm dynamiczny, który dotyczy metod `zwrocDane()` zdefiniowanych w klasie bazowej i klasach pochodnych łańcucha dziedziczenia. Wspomniany polimorfizm został uzyskany dzięki zadeklarowaniu funkcji `zwrocDane()` w klasie bazowej jako wirtualnej. W programie występują wiązania dynamiczne, czyli wiązania typu dynamicznego wskaźnika `w_pracownik` z typem obiektu, na który faktycznie wskazuje ten wskaźnik.

Ćwiczenie 15.4

Zmodyfikuj program zawarty w przykładzie 15.4 — zamiast obiektów: `pracownik1`, `pracownik2` i `pracownik3`, dla których pamięć operacyjna została przydzielona statycznie na stosie, wykorzystaj obiekty, dla których pamięć operacyjna została zaalokowana w sposób dynamiczny na stercie.



15.3. Pytania i zadania kontrolne

15.3.1. Pytania

- 1.** Na czym polega zjawisko polimorfizmu statycznego?
- 2.** Wyjaśnij znaczenie pojęcia przeciążania metod.
- 3.** Na czym polega polimorfizm dynamiczny? Wymień najważniejsze cechy polimorfizmu dynamicznego.
- 4.** Omów zagadnienie przesłaniania metod.
- 5.** Co to jest metoda wirtualna?
- 6.** Jakie są zalety stosowania polimorfizmu statycznego i dynamicznego?

15.3.2. Zadania

- 1.** Napisz program pozwalający przetwarzanie danych uczniów. Uwzględnij następujące dane: imię, nazwisko, numer w dzienniku, klasę, grupę. Wykorzystaj mechanizm polimorfizmu statycznego w odniesieniu do metody, której zadaniem jest wyświetlenie na ekranie monitora wszystkich lub wybranych danych ucznia. Wykonaj testy działania programu na danych kilku uczniów. Dane uczniów zainicjuj w treści programu.
- 2.** Napisz program pozwalający przetwarzanie wybranych danych smartfona: markę, model, rok produkcji, cenę, system operacyjny, przekątną wyświetlacza. Zastosuj mechanizm polimorfizmu dynamicznego.

morfizmu statycznego do metod, których zadaniem jest realizacja operacji wejścia/wyjścia. Zapewnij możliwość wprowadzenia wszystkich lub wybranych danych smartfona z klawiatury oraz wyświetlenia wszystkich lub wybranych danych na ekranie monitora. Wykonaj testy działania programu na danych kilku smartfonów.

3. Napisz program pozwalający przetwarzać dane pracowników szpitala, a w szczególności lekarzy i ordynatorów oddziałów szpitalnych. Wykorzystaj mechanizm dziedziczenia pojedynczego wielopoziomowego oraz polimorfizm dynamiczny. Załóż, że każdy lekarz ma określoną specjalizację (np. kardiologia, chirurgia), a każdy ordynator jest lekarzem i jednocześnie kieruje pojedynczym oddziałem w szpitalu (np. oddziałem wewnętrznym, dziecięcym). Dostęp do obiektów odpowiadających przykładowym lekarzom i ordynatorom zrealizuj w tradycyjny sposób, przy użyciu operatora dostępu . (kropki). Wykonaj testy działania programu na danych kilku lekarzy i ordynatorów.
4. Zrób tak jak w zadaniu 3., z tym że dostęp do obiektów odpowiadających przykładowym lekarzom i ordynatorom zrealizuj za pośrednictwem wskaźnika należącego do typu statycznego zgodnego z zaproponowanym przez siebie typem bazowym (np. klasą **Pracownik**) przy użyciu operatora strzałkowego ->.
5. Napisz program pozwalający przetwarzać dane wybranych pracowników na komendzie policji. Uwzględnij policjantów oraz naczelników wydziałów. Wykorzystaj mechanizm dziedziczenia wielokrotnego oraz polimorfizm dynamiczny. Załóż, że każdy policjant ma określony stopień służbowy (np. podkomisarz, komisarz, inspektor), a każdy naczelnik jest policjantem i jednocześnie kieruje jednym wydziałem na komendzie (np. wydziałem ruchu drogowego, wydziałem prewencji). Dostęp do obiektów odpowiadających przykładowym policjantom i naczelnikom zrealizuj w tradycyjny sposób, przy użyciu operatora dostępu . (kropki). Wykonaj testy działania programu na danych kilku policjantów i naczelników wydziałów.
6. Zrób tak jak w zadaniu 5., z tym że dostęp do obiektów odpowiadających przykładowym policjantom i naczelnikom zrealizuj za pośrednictwem wskaźników należących do typów statycznych zgodnych z zaproponowanymi przez siebie typami bazowymi (np. klasami: **Pracownik**, **Stopien**, **Wydzial**) z zastosowaniem operatora strzałkowego ->.

16

Mechanizm abstrakcji

Mechanizm abstrakcji to jedna z najważniejszych cech paradymatu obiektowego. W ogólności polega on na ukrywaniu przed użytkownikiem szczegółów implementacji klas (obiektów), a pokazywaniu jedynie ich funkcjonalności. Innymi słowy, abstrakcja pozwala ukrywać to, w jaki sposób osiągnięto daną funkcjonalność, i skupić się jedynie na tej funkcjonalności, czyli na tym, co robi (wykonuje) dana metoda (obiekt).

W popularnych językach programowania, takich jak Java i C#, mechanizm abstrakcji jest implementowany za pomocą tzw. klas abstrakcyjnych i interfejsów.

W wymienionych językach programowania **klasa abstrakcyjna** (ang. *abstract class*) to klasa bazowa, która zawiera deklaracje (prototypy) tzw. metod abstrakcyjnych oraz kompletne definicje innych elementów członkowskich, np. zwykłych — instancjacyjnych zmiennych i funkcji członkowskich (metod). Przy czym **metoda abstrakcyjna** (ang. *abstract method*) zadeklarowana w bazowej klasie abstrakcyjnej to metoda, która musi zostać zdefiniowana w jej klasie pochodnej. Tym samym każda metoda abstrakcyjna jest metodą polimorficzną. Zadeklarowanie metody jako abstrakcyjnej automatycznie łączy mechanizm polimorfizmu dynamicznego, czyli polimorfizmu zachodzącego w czasie wykonywania programu.

Drugie z wymienionych wcześniej narzędzi abstrakcji w językach Java i C#, **interfejs** (ang. *interface*), jest strukturą programistyczną, która ukrywa szczegóły implementacji metod, a pokazuje jedynie ich zachowanie — funkcjonalność. Interfejsy — w przeciwieństwie do klas abstrakcyjnych — mogą zawierać wyłącznie deklaracje metod abstrakcyjnych, stałe i metody statyczne oraz metodę domyślną. W interfejsach nie można definiować zwykłych, instancjacyjnych zmiennych członkowskich i metod. Natomiast klasa abstrakcyjna może, co już zostało wspomniane, zawierać również np. zmienne i metody instancjacyjne. W językach Java i C# interfejsy można definiować niezależnie od klas abstrakcyjnych, jako zupełnie odrębne struktury programistyczne.

Zgodnie zaś z regułami dziedziczenia w Javie i C#, a szczególnie ze względu na brak możliwości dziedziczenia wielokrotnego, w wymienionych językach programowania dana klasa może dziedziczyć tylko po jednej klasie bazowej, np. klasie abstrakcyjnej, ale jednocześnie może dziedziczyć po wielu interfejsach.

Przy czym w języku C++ mechanizm abstrakcji może być realizowany na kilka sposobów. Jednym z najważniejszych jest użycie klas abstrakcyjnych. Ponadto abstrakcję można osiągnąć przez odpowiednie zorganizowanie struktury wewnętrznej klasy, jak też dzięki wykorzystaniu zbiorów nagłówkowych.

Interfejsy (znane z języków Java i C#) w C++ są implementowane wyłącznie za pomocą klas abstrakcyjnych. Taka możliwość wynika z ważnej, unikatowej cechy języka C++, polegającej na tym, że dziedziczenie może w nim być wielokrotne, czyli klasa pochodna może dziedziczyć po wielu klasach bazowych.

Biorąc pod uwagę powyższe, klasa abstrakcyjna w języku C++ łączy cechy klasy abstrakcyjnej i interfejsu z Javy i C#. Jak wspomniano wcześniej, w Javie i C# dana klasa może dziedziczyć po wielu interfejsach, ale tylko po jednej klasie bazowej (np. klasie abstrakcyjnej). W C++ określona klasa może dziedziczyć po wielu klasach, w tym klasach abstrakcyjnych.

16.1. Klasa abstrakcyjna

Jednym z najważniejszych sposobów uzyskania abstrakcji w języku C++ jest użycie klas abstrakcyjnych. W ogólności **klasa abstrakcyjna** (ang. *abstract class*) to klasa bazowa, która zawiera deklarację (prototyp) przynajmniej jednej **metody abstrakcyjnej** (ang. *abstract method*). W C++ metody abstrakcyjne implementuje się za pomocą **metod czysto wirtualnych** (ang. *pure virtual methods*).

Deklaracja metody czysto wirtualnej jest analogiczna do deklaracji zwykłej metody wirtualnej, z tym że deklaracja tej pierwszej jest zakończona przypisaniem do niej wartości 0. Innymi słowy, ciało funkcji czysto wirtualnej jest zastępowane przypisaniem = 0. Na przykład kod `virtual void wyswietlDane() = 0;` zawarty w definicji klasy abstrakcyjnej określa deklarację bezparametrowej metody czysto wirtualnej o nazwie `wyswietlDane`, niezwracającej na zewnątrz żadnej wartości.

UWAGA

W dalszej części podręcznika metody czysto wirtualne są nazywane metodami abstrakcyjnymi.

Metoda abstrakcyjna, której deklaracja (prototyp) jest zawarta w abstrakcyjnej klasie bazowej, powinna zostać zdefiniowana w jej klasie pochodnej. Jeśli jednak klasa pochodna, będąca bezpośrednim potomkiem — „dzieckiem” — abstrakcyjnej klasy bazowej, nie zawiera tej definicji, to mimo że jest klasą pochodną, jest traktowana jako klasa abstrakcyjna. W takim przypadku definicja metody abstrakcyjnej powinna się znaleźć w którymś z potomków na-

leżących do łańcucha dziedziczenia — np. „wnuku” klasy abstrakcyjnej, w której tę metodę zadeklarowano.

Jedną z najważniejszych cech klas abstrakcyjnych jest to, że nie można utworzyć obiektu, który byłby jej instancją. Przy czym dotyczy to zarówno abstrakcyjnych klas bazowych, czyli tych, które zawierają deklaracje metod abstrakcyjnych, jak i określonych klas pochodnych tych klas — tych, w których nie zdefiniowano metod abstrakcyjnych.

Abstrakcja implementowana w języku C++ za pomocą klas abstrakcyjnych jest ściśle powiązana z inną cechą paradygmatu obiektowego — polimorfizmem. Mianowicie zadeklarowanie funkcji członkowskiej klasy jako wirtualnej automatycznie załącza mechanizm polimorfizmu. Przy czym jest to polimorfizm dynamiczny, czyli polimorfizm zachodzący w czasie wykonywania programu.

UWAGA

Tematyka polimorfizmu dynamicznego została omówiona szczegółowo w podrozdziale 15.2 podręcznika.

Przykład 16.1

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

// Definicja abstrakcyjnej klasy bazowej Osoba:
class Osoba {
public:
    // Deklaracje zmiennych członkowskich:
    string imie;
    string nazwisko;
/* UWAGA
* Klasa abstrakcyjna może zawierać deklaracje zwykłych — instancjacyjnych zmiennych i metod członkowskich.
*/
// Deklaracja (prototyp) metody abstrakcyjnej — funkcji czysto wirtualnej:
    virtual void wyswietlDane() = 0;
};

// Definicja klasy pochodnej Pracownik:
class Pracownik : public Osoba {
public:
    string stanowisko;
// Definicja metody abstrakcyjnej zadeklarowanej w bazowej klasie abstrakcyjnej:
    void wyswietlDane() {
        cout << imie << " " << nazwisko << ", " << stanowisko << endl;
    }
}
```

```

/* UWAGA
 * Metoda abstrakcyjna zadeklarowana w bazowej klasie abstrakcyjnej powinna zostać zdefiniowana
 * w jej klasie pochodnej.
 */
};

// Definicja klasy pochodnej Uczen:
class Uczen : public Osoba {
public:
    string klasa;
    // Definicja metody abstrakcyjnej zadeklarowanej w bazowej klasie abstrakcyjnej:
    void wyswietlDane() {
        cout << imie << " " << nazwisko << ", " << klasa << endl;
    }
};

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy pochodnej Pracownik:
    Pracownik pracownik;
    // Nadanie wartości zmiennym członkowskim obiektu pracownik:
    pracownik.imie = "Jan";
    pracownik.nazwisko = "Kowalski";
    pracownik.stanowisko = "nauczyciel";
    cout << "DANE PRACOWNIKA:" << endl;
    // Wywołanie funkcji wyswietlDane() zdefiniowanej w klasie pochodnej Pracownik:
    pracownik.wyswietlDane();

    // Utworzenie obiektu uczen jako instancji klasy pochodnej Uczen:
    Uczen uczen;
    // Nadanie wartości zmiennym członkowskim obiektu uczen:
    uczen.imie = "Maria";
    uczen.nazwisko = "Nowak";
    uczen.klasa = "4A";
    cout << "DANE UCZNIA:" << endl;
    // Wywołanie funkcji wyswietlDane() zdefiniowanej w klasie Uczen:
    uczen.wyswietlDane();

    return 0;
}

```

W programie zdefiniowano klasę abstrakcyjną Osoba, która jest klasą bazową dla dwóch klas pochodnych: Pracownika i Uczen. Definicja klasy Osoba zawiera deklarację metody abstrakcyjnej wyswietlDane(), która została zaimplementowana jako metoda czysto wirtualna. Oprócz tego w skład klasy Osoba wchodzą zmienne członkowskie imie i nazwisko.

Definicje obu klas pochodnych `Pracownik` i `Uczeń` zawierają definicje metody abstrakcyjnej `wyswietlDane()`, którą zadeklarowano w klasie bazowej `Osoba`. Definicje te są od siebie w pełni niezależne.

Zadeklarowanie w klasie bazowej metody `wyswietlDane()` jako (czysto) wirtualnej powoduje automatyczne załączenie mechanizmu polimorfizmu dynamicznego, czyli polimorfizmu zachodzącego w czasie wykonywania programu. Metodę polimorficzną stanowi oczywiście wspomniana powyżej metoda `wyswietlDane()`. W wyniku tego klasy wchodzące w skład łańcucha dziedziczenia tworzą hierarchię polimorficzną.

Wykorzystano tu zatem trzy założenia (cechy) paradygmatu obiektowego: dziedziczenie, polimorfizm i abstrakcję.

Zaimplementowana w programie abstrakcja dotyczy klas, a nie danych. W szczególności abstrakcja obejmuje wspólny interfejs klas `Pracownik` i `Uczeń`, którym jest metoda `wyswietlDane()`. Implementacja wymienionej metody na poziomie bazowej klasy abstrakcyjnej `Osoba` nie jest znana, ponieważ nie została tam określona. Jeśli spojrzeć z perspektywy klasy bazowej `Osoba`, wiadomo jest jedynie abstrakcyjny cel (zadanie) metody `wyswietlDane()`, czyli co ta metoda ma robić — ale nie wiadomo, w jaki sposób. Różne implementacje metody abstrakcyjnej `wyswietlDane()`, czyli opisanie, w jaki sposób osiągnięto jej założoną funkcjonalność, są zawarte w klasach pochodnych klasy abstrakcyjnej `Osoba`, tj. w klasach `Pracownik` i `Uczeń`.

Ćwiczenie 16.1

Zmodyfikuj program z przykładu 16.1 — wszystkie zmienne członkowskie klas `Osoba`, `Pracownik` i `Uczeń` zadeklaruj jako chronione (ang. `protected`). Zdefiniuj w wymienionych klasach publiczne metody dostępowe — settery i gettery odpowiadające każdej z tych zmieniemych. Ponadto uzupełnij definicję klasy abstrakcyjnej `Osoba` i definicje klas pochodnych `Pracownik` i `Uczeń` o definicje konstruktorów: domyślnych i parametrycznych.

Przykład 16.2

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

// Definicja abstrakcyjnej klasy bazowej Osoba:
class Osoba {
public:
    string imie;
    string nazwisko;
// Deklaracja metody abstrakcyjnej wyswietlDane():
    virtual void wyswietlDane() = 0; // funkcja czysto wirtualna
};

// Definicja abstrakcyjnej klasy bazowej Stanowisko:
class Stanowisko {
```

```
public:  
    string stanowisko;  
    // Deklaracja metody abstrakcyjnej wyswietlDane():  
    virtual void wyswietlDane() = 0; //funkcja czysto wirtualna  
};  
// Definicja abstrakcyjnej klasy bazowej Klasa:  
class Klasa {  
public:  
    string klasa;  
    // Deklaracja metody abstrakcyjnej wyswietlDane():  
    virtual void wyswietlDane() = 0; //funkcja czysto wirtualna  
};  
// Definicja klasy pochodnej Pracownik:  
class Pracownik : public Osoba, public Stanowisko {  
public:  
    // Definicja metody wyswietlDane():  
    void wyswietlDane() {  
        cout << imie << " " << nazwisko << ", " << stanowisko << endl;  
    }  
    /* UWAGA  
     * Metoda wyswietlDane() została zadeklarowana jako abstrakcyjna w klasach bazowych klasy Pracownik,  
     * tj. klasach Osoba i Stanowisko.  
     */  
};  
// Definicja klasy pochodnej Uczen:  
class Uczen : public Osoba, public Klasa {  
public:  
    // Definicja metody wyswietlDane():  
    void wyswietlDane() {  
        cout << imie << " " << nazwisko << ", " << klasa << endl;  
    }  
    /* UWAGA  
     * Metoda wyswietlDane() została zadeklarowana jako abstrakcyjna w klasach bazowych klasy Uczen,  
     * tj. klasach Osoba i Klasa.  
     */  
};  
  
int main() {  
    Pracownik pracownik;  
    pracownik.imie = "Jan";  
    pracownik.nazwisko = "Kowalski";  
    pracownik.stanowisko = "nauczyciel";  
    cout << "DANE PRACOWNIKA:" << endl;  
    pracownik.wyswietlDane();
```

```

Uczen uczen;
uczen.imie = "Maria";
uczen.nazwisko = "Nowak";
uczen.klasa = "4A";
cout << "DANE UCZNIA:" << endl;
uczen.wyswietlDane();

return 0;
}

```

Funkcjonalność programu przedstawionego w tym przykładzie jest analogiczna do funkcjonalności programu z przykładu 16.1. Różnica pomiędzy tymi programami tkwi w zastosowanych w nich rodzajach dziedziczenia. Mianowicie w programie z przykładu 16.1 wykorzystano dziedziczenie hierarchiczne, a tutaj — dziedziczenie wielokrotne.

Metoda `wyswietlDane()` została zadeklarowana jako czysto wirtualna (czyli abstrakcyjna) w trzech klasach bazowych: `Osoba`, `Stanowisko` i `Klasa`. Wymienione klasy są zatem klasami abstrakcyjnymi. Założona w klasach bazowych funkcjonalność metody `wyswietlDane()` to wyświetlenie danych. Nie zostało tam jednak określone, w jaki sposób ma to być zrealizowane.

Definicje tej metody abstrakcyjnej `wyswietlDane()` są zawarte w klasach pochodnych `Pracownik` i `Uczeń`. Należy zwrócić uwagę na to, że implementacje metody `wyswietlDane()` w wymienionych klasach pochodnych są odmienne. Stanowią one różne odpowiedzi na pytanie, w jaki sposób osiągnięto funkcjonalność tej metody założoną w klasach abstrakcyjnych.

Ćwiczenie 16.2

Porównaj szczegółowo kod programu z przykładu 16.2 z kodem programu z przykładu 16.1. Odpowiedz na następujące pytania:

- Które z zastosowanych rozwiązań jest bardziej przejrzyste?
- Które rozwiązanie umożliwia łatwiejszą rozbudowę o dodatkowe klasy `Nauczyciel` i `Wychowawca`? Załóż, że każdy nauczyciel jest pracownikiem, a każdy wychowawca — nauczycielem.

Uzasadnij udzielone odpowiedzi.



16.2. Interfejsy

Jak już wspomniano, **interfejs** (ang. *interface*) jest strukturą programistyczną, która ukrywa szczegóły implementacji funkcji członkowskich klas, czyli metod, a pokazuje jedynie ich funkcjonalność. Abstrakcja realizowana za pomocą interfejsów dotyczy zatem wyłącznie klas (metod) — nie danych.

W języku C++ interfejsy są implementowane wyłącznie za pomocą klas abstrakcyjnych będących klasami bazowymi dla innych klas, które wchodzą w skład łańcucha dziedziczenia. Taka możliwość wynika z unikatowej cechy języka C++ polegającej na tym, że można w nim wykorzystywać dziedziczenie wielokrotne, co oznacza, że dana klasa pochodna może dziedziczyć po wielu klasach bazowych.

W szczególności w języku C++ za interfejs można uważać klasę abstrakcyjną, która zawiera deklaracje metod abstrakcyjnych i ewentualnie definicje komponentów statycznych (np. metod statycznych). Jednocześnie zakłada się, że w skład interfejsu nie mogą wchodzić definicje instancjacyjnych elementów członkowskich, tj. zmiennych i metod instancjacyjnych.

Wykorzystanie interfejsów, jako jednych z ważniejszych narzędzi abstrakcji w programowaniu obiektowym, pozwala oddzielić wspólny interfejs klas wchodzących w skład łańcucha dziedziczenia od ich implementacji. Przy tym należy zwrócić uwagę, że ten wspomniany wspólny interfejs klas jest polimorficzny. Wynika to z właściwości metod abstrakcyjnych, które deklaruje się w języku C++ jako funkcje członkowskie czysto wirtualne. To z kolei pociąga za sobą automatyczne załączenie mechanizmu polimorfizmu dynamicznego.

Podsumowując, interfejs, jako abstrakcyjna klasa bazowa, może zawierać jedynie sygnatury, tj. deklaracje (prototypy) metod. Są to metody abstrakcyjne. Implementacje metod abstrakcyjnych zadeklarowanych w interfejsie zaś są definiowane w jego klasach pochodnych — zgodnie z zasadami polimorfizmu dynamicznego.

W polimorficznej hierarchii klas wchodzących w skład łańcucha dziedziczenia interfejsy są podnoszone (ang. *hoisting*) na samą górę tej hierarchii. Zapewnia to efektywne i przejrzyste oddzielenie interfejsu klas polimorficznych od ich implementacji. Klasa interfejsu wyraża jedynie (abstrakcyjne) możliwości, cele i zadania, natomiast ich konkretna implementacja nie jest w tym miejscu istotna. Za wspomnianą implementację odpowiadają klasy pochodne interfejsu. Dzięki temu implementacje można dodawać/zmieniać w zależności od potrzeb — bez konieczności zmiany istniejącego kodu interfejsu.

Przykład 16.3

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

// Definicja interfejsu Info:
class Info {
public:
    // Deklaracja metody abstrakcyjnej:
    virtual void wyswietlDane() = 0;
};

/* UWAGA
* Klasa abstrakcyjna — interfejs Info zawiera wyłącznie deklarację metody abstrakcyjnej wyswietlDane(),
* natomiast nie zawiera definicji żadnych innych zmiennych ani funkcji członkowskich.
*/
```

```

// Definicja klasy bazowej Osoba:
class Osoba {
public:
    string imie;
    string nazwisko;
    string szkola;
};

// Definicja klasy pochodnej Pracownik:
class Pracownik : public Info, public Osoba {
    /* UWAGA
     * Klasa pochodna Pracownik dziedziczy po interfejsie Info oraz klasie Osoba. Jest to dziedziczenie wielokrotne.
     */
public:
    // Deklaracja zmiennej członkowskiej:
    string stanowisko;
    // Definicja metody wyswietlDane() zadeklarowanej w interfejsie Info jako metoda abstrakcyjna:
    void wyswietlDane() {
        cout << "Imię: " << this->imie << endl;
        cout << "Nazwisko: " << this->nazwisko << endl;
        cout << "Stanowisko: " << this->stanowisko << endl;
    }
};

// Definicja klasy pochodnej Uczen:
class Uczen : public Info, public Osoba {
    /* UWAGA
     * Klasa pochodna Uczen dziedziczy po interfejsie Info i klasie Osoba.
     */
public:
    string klasa;
    // Definicja metody wyswietlDane() zadeklarowanej w interfejsie Info jako metoda abstrakcyjna:
    void wyswietlDane() {
        cout << "Imię: " << this->imie << endl;
        cout << "Nazwisko: " << this->nazwisko << endl;
        cout << "Klasa: " << this->klasa << endl;
    }
};

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy pochodnej Pracownik:
    Pracownik pracownik;
    // Nadanie wartości początkowych zmiennym instancyjnym obiektu pracownik:
    pracownik.imie = "Jan";
    pracownik.nazwisko = "Kowalski";
    pracownik.stanowisko = "nauczyciel";
}

```

```

cout << "DANE PRACOWNIKA:" << endl;
// Wywołanie metody wyswietlDane() zdefiniowanej w klasie Pracownik:
pracownik.wyswietlDane();
cout << endl;

Uczen uczen;
uczen.imie = "Maria";
uczen.nazwisko = "Nowak";
uczen.klasa = "4A";
cout << "DANE UCZNIA:" << endl;
uczen.wyswietlDane();

return 0;
}

```

W programie zaimplementowano mechanizm abstrakcji dotyczący klas. W szczególności zdefiniowano tutaj klasę abstrakcyjną `Info`. Należy zwrócić uwagę, że w jej skład nie wchodzą żadne instancynie zmienne i metody członkowskie, lecz jedynie deklaracja metody abstrakcyjnej `wyswietlDane()`. Dlatego też klasa `Info` odpowiada strukturze programistycznej nazywanej interfejsem — znanej z takich języków programowania jak Java i C#.

W językach Java i C# dana klasa może dziedziczyć po wielu interfejsach, ale tylko po jednej klasie bazowej. Natomiast w C++ można zaimplementować dziedziczenie wielokrotne — jak w omawianym programie. Mianowicie klasy pochodne `Pracownik` i `Uczen` dziedziczą po tych samych dwóch klasach bazowych: klasie abstrakcyjnej (interfejsie) `Info` i zwykłej klasie `Osoba`.

W klasach `Pracownik` i `Uczen` zdefiniowano metodę `wyswietlDane()`, zadeklarowaną w interfejsie `Info` jako metoda abstrakcyjna. Funkcjonalność i implementacja tej metody w klasach `Pracownik` i `Uczen` są odmienne. Tym samym, jeśli spojrzeć z perspektywy interfejsu `Info`, nastąpiło oddzielenie wspólnego interfejsu klas `Pracownik` i `Uczen`, wchodzących w skład łańcucha dziedziczenia, od ich implementacji, która jest w nich zawarta.

Na poziomie interfejsu `Info` implementacja metody abstrakcyjnej `wyswietlDane()` nie jest znana, ponieważ nie została tam określona. Wiadome jest jedynie jej zadanie ogólne, którym jest odpowiedź na pytanie: co robi (wykonuje) ta metoda?

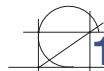
Jak wspomniano wcześniej, wspólny interfejs klas pochodnych `Pracownik` i `Uczen`, obejmujący metodę `wyswietlDane()`, został oddzielony od jej implementacji, która jest zawarta w wymienionych klasach pochodnych. Tym samym ewentualna modyfikacja kodu metody `wyswietlDane()` nie oznacza konieczności zmiany jej interfejsu zawartego w klasie (interfejsie) `Info`. Należy podkreślić, że interfejs `Info` został „podniesiony” na samą górę hierarchii klas wchodzących w skład łańcucha dziedziczenia.

Inną ważną cechą zaimplementowanej w programie hierarchii klas jest jej polimorficzność. Dotyczy to oczywiście polimorfizmu dynamicznego, czyli polimorfizmu zachodzącego w cza-

sie wykonywania programu. Wynika to z unikatowej cechy języka C++, w którym metoda abstrakcyjna `wyswietlDane()` została zadeklarowana w klasie abstrakcyjnej, interfejsie `Info`, jako funkcja (czysto) wirtualna.

Ćwiczenie 16.3

Zmodyfikuj program zawarty w przykładzie 16.3 — zdefiniuj dodatkowy interfejs o nazwie `InfoGminaSzkoła` zawierający definicje dwóch metod statycznych, które pozwolą wyświetlić na ekranie nazwę szkoły oraz prowadzącego je miasta (gminy). Wykorzystaj interfejs `InfoGminaSzkoła` w definicjach klas `Pracownik` i `Uczeń`.



16.3. Abstrakcja danych

W programowaniu obiektowym stosowanie abstrakcji oznacza udostępnianie świata zewnętrznemu tylko wybranych informacji, natomiast inne elementy, np. szczegóły implementacji, są ukrywane. Była już o tym mowa wcześniej w tym rozdziale w odniesieniu do klas abstrakcyjnych i interfejsów. Metody wirtualne, które można deklarować w klasach jako abstrakcyjne, są metodami „bez ciała”. Są zatem „bytami nieistniejącymi”, abstrakcyjnymi — to jedynie określenie celu, zadania, czyli operacji do wykonania.

Z punktu widzenia innych elementów członkowskich klasy pojęcie abstrakcji może dotyczyć również danych, które są w niej przechowywane w jej instancjach — obiektach. W szczególności **abstrakcja danych** (ang. *data abstraction*) dotyczy danych reprezentowanych w klasie (obiekcie) przez zmienne członkowskie. Idea abstrakcji danych sprowadza się w praktyce do ukrycia przed światem zewnętrznym szczegółów implementacji danych (np. ukrycia typów zmiennych członkowskich), a udostępnienia w otoczeniu klasy jedynie interfejsu tych danych (np. identyfikatorów zmiennych członkowskich).

W języku C++ abstrakcję danych można uzyskać dzięki odpowiedniej organizacji i budowie klas. Proces ten jest wspomagany przez użycie specyfikatorów dostępu do ich elementów członkowskich (ang. *members*): `private`, `protected` i `public`. Specyfikatory `private` i `protected` zapewniają ukrycie szczegółów implementacji klasy (np. danych) przed kodem z otoczenia klasy. Jeżeli trzeba zapewnić niejawność implementacji danych pojedynczej klasy, najwłaściwszym rozwiązaniem jest użycie specyfikatora `private`. Konieczność ukrycia implementacji danych w obrębie łańcucha dziedziczenia w hierarchii klas wymusza zastosowanie specyfikatora `protected`. Specyfikator `public` zaś jest używany w odniesieniu do metod dostępowych, które zapewniają obsługę interfejsu danych. Metody te odpowiadają za funkcjonalność obiektu (obiektów) oraz manipulowanie danymi.

Abstrakcja danych jest ściśle powiązana z inną zasadą programowania obiektowego, mechanizmem hermetyzacji — ukrywania danych. W praktyce hermetyzacja danych zazwyczaj jest realizowana przy użyciu setterów i getterów, które są publicznymi metodami dostępowymi do prywatnych/chronionych zmiennych członkowskich klas.

**UWAGA**

Tematyka hermetyzacji — ukrywania danych — została omówiona w podrozdziale 13.2.

Niewątpliwą zaletą stosowania zasad abstrakcji danych jest skuteczna ochrona implementacji tych danych. Ma to ogromne znaczenie np. w razie wystąpienia przypadkowych błędów na poziomie użytkownika (programisty), które mogą doprowadzić do niekontrolowanego stanu obiektu. Ponadto oddzielenie implementacji danych od ich interfejsu zapewnia łatwość modyfikowania klasy. Mianowicie nawet jeśli implementacja danych się zmieni, wystarczy przeprowadzić analizę kodu ich interfejsu, tj. kodu publicznych metod dostępowych, i w razie potrzeby wprowadzić niezbędne korekty. Ponadto w niektórych sytuacjach jest niepożądane, by użytkownik (tj. otoczenie klasy) miał wiedzę dotyczącą implementacji klasy (niekiedy jest to wręcz szkodliwe).

Przykład 16.4

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

// Definicja klasy Pracownik:
class Pracownik {
private:
    // Deklaracje prywatnych zmiennych członkowskich:
    string imie, nazwisko;
    string stanowisko;
    int nr_legit;
    /* UWAGA
     * Szczegóły implementacji danych pracownika przechowywanych w zmiennych członkowskich klasy
     * nie są dostępne dla świata zewnętrznego — są ukryte i niedostępne bezpośrednio dla kodu w otoczeniu klasy.
     */
public:
    // Definicje publicznych setterów i getterów:
    void setImie(string pImie) {
        this->imie = pImie;
    }
    string getImie() {
        return this->imie;
    }
    void setNazwisko(string pNazwisko) {
        this->nazwisko = pNazwisko;
    }
    string getNazwisko() {
        return this->nazwisko;
    }
    void setStanowisko(string pStanowisko) {
```

```

        this->stanowisko = pStanowisko;
    }
    string getStanowisko() {
        return this->stanowisko;
    }
    void setNrLegit(int pNrLegit) {
        const int temp = 1000;
        this->nr_legit = pNrLegit + temp;
    }
    int getNrLegit() {
        return this->nr_legit;
    }
    /* UWAGA
     * Settery i gettery stanowią publiczny interfejs danych pracownika, odpowiadający prywatnym
     * zmiennym członkowskim klasy.
     */
};

// Definicja zwykłej funkcji niezależnej od klasy Pracownik:
void wyswietlDane(Pracownik pPracownik) {
    cout << "Imię: " << pPracownik.getImie() << endl;
    cout << "Nazwisko: " << pPracownik.getNazwisko() << endl;
    cout << "Stanowisko: " << pPracownik.getStanowisko() << endl;
    cout << "Numer legitymacji: " << pPracownik.getNrLegit() << endl;
}

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy Pracownik:
    Pracownik pracownik;
    // Nadanie wartości początkowych zmiennym członkowskim obiektu pracownik:
    pracownik.setImie("Jan");
    pracownik.setNazwisko("Kowalski");
    pracownik.setStanowisko("nauczyciel");
    pracownik.setNrLegit(12);
    /* UWAGA
     * Ustalenie/zmiana danych pracownika jest realizowana za pomocą publicznych metod dostępowych — setterów.
     * Metody te stanowią interfejs wspomnianych danych, które, jako prywatne, nie są dostępne w sposób bezpośredni
     * w otoczeniu klasy.
     */
    // Prezentacja danych pracownika na ekranie monitora:
    cout << "DANE PRACOWNIKA:" << endl;
    // Wywołanie funkcji wyswietlDane(), niezależnej od klasy Pracownik:
    wyswietlDane(pracownik);

    return 0;
}

```

W programie zademonstrowano implementację abstrakcji danych pracownika, reprezentowanych przez zmienne członkowskie: `imie`, `nazwisko`, `stanowisko`, `nr_legit` obiektu `pracownik`. Obiekt `pracownik` jest instancją klasy `Pracownik`.

Szczegóły implementacji tych danych są ukryte przed światem zewnętrznym dzięki zadeklarowaniu ich w klasie `Pracownik` przy użyciu specyfikatora dostępu `private`. Interfejs ukrytych danych zapewniają publiczne metody dostępowe, czyli settery i gettery, poszczególnych zmiennych członkowskich oraz metoda instancyjna `wyswietlDane()`.

Implementacja danych w klasie `Pracownik` została skutecznie oddzielona od ich interfejsu. Oznacza to, że np. dodanie nowych elementów członkowskich do klasy `Pracownik`, takich jak zmienna członkowska odpowiadająca stażowi pracy pracownika, nie będzie miało żadnego wpływu na dotychczasowy interfejs danych w tej klasie. Jednocześnie uwzględnienie nowych danych pracownika zaimplementowanych w klasie `Pracownik` sprawi, że będzie trzeba zmienić kod funkcji `wyswietlDane()`, której zadaniem jest wyświetlenie na ekranie monitora wszystkich danych pracownika.

Ćwiczenie 16.4

Zmodyfikuj program zawarty w przykładzie 16.4 — uwzględnij w klasie `Pracownik` dodatkowe dane pracownika: staż pracy oraz wynagrodzenie podstawowe. Zaimplementuj abstrakcję nowych danych.

16.4. Mechanizm abstrakcji a pliki nagłówkowe

W języku C++ założenia abstrakcji można zrealizować również przez wykorzystanie plików nagłówkowych (ang. *header files*). W tym celu kod źródłowy zawierający określone elementy składowe definicji klasy można umieścić w plikach zewnętrznych:

- pliku nagłówkowym (`.h`),
- pliku z kodem źródłowym (`.cpp`).

Plik nagłówkowy zawiera definicję klasy. Przy czym w definicji klasy zamieszcza się jedynie jej interfejs, tj. deklaracje (a nie definicje) publicznych metod dostępowych do prywatnych (lub chronionych) zmiennych członkowskich. Kompletne definicje wspomnianych metod publicznych klasy, czyli ich implementacja, powinny być zawarte w pliku z kodem źródłowym `.cpp`.

UWAGA

Wykorzystanie w programie zbiorów nagłówkowych zostało omówione wcześniej — w podrozdziale 9.1.

Zgodnie z przedstawionymi wcześniej założeniami zarówno implementacja danych reprezentowanych przez prywatne (lub chronione) zmienne członkowskie klasy, jak i implementacja publicznych metod dostępowych do tych danych zostają ukryte przed światem zewnętrznym. Rolę „kontenera” (pojemnika) przeznaczonego na ukryte szczegóły implementacji funkcji członkowskich klasy odgrywa plik nagłówkowy. Jednocześnie należy zwrócić uwagę, że jest to kontener zewnętrzny. Kontenerem wewnętrzny jest klasa „sama w sobie”, w której zmienne członkowskie zostały zadeklarowane jako prywatne lub chronione. Mamy więc w tym przypadku do czynienia z abstrakcją dotyczącą zarówno danych, jak i innych komponentów klasy — w szczególności publicznych metod instancyjnych.

Implementacja hermetyzacji — ukrycia danych — zrealizowana na poziomie klasy z jednaczesnym wykorzystaniem plików nagłówkowych w celu ukrycia przed światem zewnętrznym szczegółów implementacji jej metod publicznych skutkuje efektywnym oddzieleniem implementacji klasy od jej interfejsu. Dzięki temu mechanizm abstrakcji dotyczy wszystkich komponentów składowych klasy.

Przykład 16.5

Program główny:

```
#include <iostream>
// Długość do aplikacji zdefiniowanego pliku nagłówkowego:
#include "kolo.h"
/* UWAGA
 * Kompilator będzie poszukiwał pliku nagłówkowego w katalogu bieżącym (czyli tym, w którym jest zapisany plik main.cpp).
 */
using namespace std;

int main() {
    // Utworzenie obiektu kolo klasy Kolo:
    Kolo kolo(1); // niejawne wywołanie konstruktora parametrycznego z argumentem 1
    // Odczyt bieżącej wartości promienia:
    cout << "Promień koła: " << kolo.getPromien() << endl;
    /* UWAGA
     * Odczyt wartości prywatnej zmiennej członkowskiej _r został zrealizowany z użyciem gettera getPromien().
     */
    // Obliczenie pola i obwodu koła:
    double poleK = kolo.pole();
    double obwodK = kolo.obwod();
    /* UWAGA
     * Obliczenie pola i obwodu koła zostało zrealizowane z zastosowaniem publicznych metod instancyjnych klasy Kolo.
     */
    // Prezentacja wyników na ekranie monitora:
    cout << "Pole koła: " << poleK << endl;
    cout << "Obwód koła: " << obwodK << endl;
```

```

        return 0;
    }

Zawartość pliku nagłówkowego kolo.h:
// PLIK NAGŁÓWKOWY
// Deklaracja klasy Kolo:
class Kolo {
private:
    // Deklaracja prywatnej zmiennej członkowskiej:
    double _r;
public:
    // Prototypy konstruktorów:
    Kolo();
    Kolo(double);
    // Deklaracje publicznych metod dostępowych:
    void setPromien(double); // setter
    double getPromien(); // getter
    // Prototypy (nagłówki) zwykłych metod instancjacyjnych:
    double pole();
    double obwod();
};


```

Zawartość pliku *kolo.cpp*:

```

// PLIK Z KODEM ŹRÓDLOWYM (IMPLEMENTACJĄ) METOD PUBLICZNYCH
#define _USE_MATH_DEFINES // w celu użycia stałej M_PI, która nie jest zdefiniowana w standardowym C/C++
#include <cmath>
#include "kolo.h"
// Definicje publicznych metod członkowskich klasy Kolo
// Definicje konstruktorów:
Kolo::Kolo() {}
Kolo::Kolo(double r) {
    _r = r;
}
// Definicja settera:
void Kolo::setPromien(double r) {
    _r = r;
}
// Definicja gettera:
double Kolo::getPromien() {
    return _r;
}
// Definicje zwykłych metod instancjacyjnych:
double Kolo::pole() {
    return M_PI * _r * _r;
}
double Kolo::obwod() {
    return 2 * M_PI * _r;
}

```

Implementacja danych reprezentowanych w klasie `Koło` przez prywatną zmienną członkowską `_r` została oddzielona od jej bezpośredniego, publicznego interfejsu — metod dostępowych: settera `setPromien()` i gettera `getPromien()`. Mamy więc tutaj do czynienia z abstrakcją danych zrealizowaną za pomocą klasy `Koło`, zorganizowanej wewnętrznie z użyciem specyfikatorów dostępu — odpowiednio: `private` i `public` — do jej elementów członkowskich.

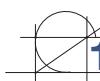
Oprócz tego wykorzystano tutaj plik nagłówkowy `kolo.h` oraz plik z kodem źródłowym `kolo.cpp`. W podanym pliku nagłówkowym ukryto przed światem zewnętrznym szczegóły implementacji klasy `Koło`. W szczególności dotyczy to implementacji publicznych metod instancjacyjnych należących do klasy `Koło`, w tym publicznych metod dostępowych do prywatnej zmiennej członkowskiej `_r` — settera `setPromien()` i gettera `getPromien()`. Kompletne definicje wspomnianych metod publicznych są zawarte w pliku `kolo.cpp`, którego zawartość nie jest udostępniana użytkownikowi w sposób bezpośredni.

Tak więc mechanizm abstrakcji dotyczy w zaprezentowanym programie wszystkich komponentów klasy `Koło` — zarówno prywatnych danych, jak i publicznych metod członkowskich:

- implementacja klasy `Koło` została oddzielona od jej interfejsu,
- szczegóły implementacji klasy `Koło` zostały ukryte przed światem zewnętrznym.

Ćwiczenie 16.5

Zmodyfikuj program zawarty w przykładzie 16.5 — zamiast klasy `Koło` zdefiniuj klasę `Prostokat` zawierającą elementy członkowskie, które umożliwiają ustalenie wartości i odczyt parametrów prostokąta oraz obliczenie jego pola i obwodu. Zaimplementuj mechanizm abstrakcji przy użyciu zbioru nagłówkowego `prostokat.h`. Kody źródłowe definicji publicznych funkcji członkowskich klasy `Prostokat` zapisz w pliku `prostokat.cpp`. Abstrakcję danych odpowiadających parametrom prostokąta zrealizuj przez odpowiednie wewnętrzne zorganizowanie klasy `Prostokat`.



16.5. Pytania i zadania kontrolne

16.5.1. Pytania

- 1.** Jaką rolę w programowaniu obiektowym odgrywają klasy abstrakcyjne? W jaki sposób definiuje się klasy abstrakcyjne w języku C++?
- 2.** Co to jest metoda abstrakcyjna? W jaki sposób definiuje się metody abstrakcyjne w C++?
- 3.** Czym różni się klasa abstrakcyjna od interfejsu w języku C++?
- 4.** Czy mechanizm abstrakcji jest związany wyłącznie z danymi, czy może dotyczyć również innych elementów członkowskich (komponentów) klasy?
- 5.** Na czym polega mechanizm abstrakcji danych?
- 6.** Czy wykorzystanie zbiorów nagłówkowych może być pomocne w implementacji mechanizmu abstrakcji?

16.5.2. Zadania

1. Napisz program pozwalający przetwarzanie danych pracowników szpitala, a dokładniej: lekarzy i ordynatorów oddziałów szpitalnych. Załóż, że każdy lekarz ma określoną specjalizację (np. kardiologia), a każdy ordynator jest lekarzem i jednocześnie kieruje pojedynczym oddziałem w szpitalu (np. oddziałem wewnętrznym). Zastosuj założenia dziedziczenia i abstrakcji. Mechanizm abstrakcji zrealizuj przy użyciu klas abstrakcyjnych. Wykonaj testy działania programu na danych kilku lekarzy i ordynatorów.
2. Zrób tak jak w zadaniu 1., lecz mechanizm abstrakcji zrealizuj za pomocą interfejsu (interfejsów).
3. Napisz program pozwalający przetwarzanie danych wybranych pracowników komendy policji. Uwzględnij policjantów oraz naczelników wydziałów. Załóż, że każdy policjant ma określony stopień służbowy (np. komisarz), a każdy naczelnik jest policjantem i jednocześnie kieruje jednym wydziałem w komendzie (np. wydziałem ruchu drogowego). Wykorzystaj mechanizmy dziedziczenia i abstrakcji. Mechanizm abstrakcji zrealizuj przy użyciu klas abstrakcyjnych. Wykonaj testy działania programu na danych kilku policjantów i naczelników wydziałów.
4. Zrób tak jak w zadaniu 3., lecz mechanizm abstrakcji zrealizuj za pomocą interfejsu (interfejsów).
5. Napisz program pozwalający obliczyć objętość, pole powierzchni całkowitej oraz długość wszystkich krawędzi prostopadłościanu. Dane wejściowe (parametry prostokąta) mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Wykorzystaj mechanizm abstrakcji danych.
6. Zrób tak jak w zadaniu 5., z tym że dodatkowo uwzględnij abstrakcję uzyskaną dzięki zastosowaniu pliku nagłówkowego *prostopadloscian.h* oraz pliku z kodem źródłowym *prostopadloscian.cpp*. Plik *prostopadloscian.h* powinien zawierać definicję klasy *Prostopadloscian* z deklaracjami prywatnych zmiennych i publicznych metod członkowskich, a plik *prostopadloscian.cpp* — kompletne definicje publicznych metod członkowskich klasy *Prostopadloscian*.

17

Funkcje i klasy zaprzyjaźnione

Prywatne elementy członkowskie klasy (np. prywatne zmienne członkowskie) są dostępne w sposób bezpośredni wyłącznie w obrębie klasy, w której zostały zdefiniowane. Natomiast chronione elementy członkowskie są dostępne bezpośrednio zarówno wewnątrz klasy, w której zostały zdefiniowane, jak i w jej klasach pochodnych.

Zgodnie z zasadami implementacji mechanizmu hermetyzacji (ukrywania danych) dostęp do zmiennych prywatnych i chronionych z poziomu otoczenia wymienionych wcześniej klas można zrealizować za pomocą publicznych metod dostępowych — instancjacyjnych setterów i getterów zdefiniowanych w tych klasach.

Prywatne elementy członkowskie klasy mogą być przetwarzane w jej otoczeniu także za pośrednictwem jej „przyjaciół” (ang. *friends*). To samo dotyczy możliwości przetwarzania chronionych elementów członkowskich klas wchodzących w skład łańcucha dziedziczenia na poziomie otoczenia tej hierarchii klas.

Wspomnianymi powyżej przyjaciółmi danej klasy (lub klas należących do łańcucha dziedziczenia) mogą być tzw. funkcje zaprzyjaźnione oraz klasy zaprzyjaźnione.

17.1. Funkcje zaprzyjaźnione

Funkcje zaprzyjaźnione (ang. *friend functions*) zapewniają dostęp do prywatnych członków klasy, tak samo jak metody (funkcje) członkowskie należące do tej klasy. To samo dotyczy chronionych elementów członkowskich, które z definicji są dostępne w klasie bazowej (czyli tej, w której zostały zadeklarowane) oraz w jej klasach pochodnych.

Funkcję zaprzyjaźnioną deklaruje się za pomocą prototypu w obrębie klasy, której przyjacielem jest ta funkcja. W tym celu wspomnianą deklarację należy poprzedzić słowem kluczowym **friend**. Na przykład wyrażenie: `friend double getPromienFriendKolo(Kolo);` oznacza deklarację funkcji zaprzyjaźnionej o nazwie `getPromienFriendKolo`, która ma jeden para-

metr wejściowy typu `Koło` i zwraca na zewnątrz wartość typu `double`. Przy tym jest zupełnie obojętne, czy deklaracja funkcji zaprzyjaźnionej znajduje się w sekcji prywatnej (chronionej) klasy, czy też w sekcji publicznej.

Funkcja zaprzyjaźniona danej klasy musi mieć co najmniej jeden parametr/argument typu obiektowego. Powinien on stanowić instancję klasy, której ta funkcja jest przyjacielem. Wynika to bezpośrednio ze sposobu uzyskiwania dostępu do prywatnych (chronionych) zmiennych członkowskich klasy przez tę funkcję. Mianowicie ten dostęp jest realizowany za pośrednictwem obiektu, który jest argumentem wywołania funkcji zaprzyjaźnionej.

Definicja funkcji zaprzyjaźnionej może się znajdować w dowolnym miejscu programu — tak samo jak definicje „zwykłych” funkcji. Bardzo ważne jest to, że dana funkcja zaprzyjaźniona może być „przyjacielem” kilku klas jednocześnie.

Należy podkreślić, że funkcja zaprzyjaźniona nie jest elementem członkowskim klasy (ang. *class member*), w której została zadeklarowana. Tym samym nie może ona być wywoływana za pośrednictwem obiektu będącego instancją tej klasy.

W ogólności funkcje zaprzyjaźnione można definiować jako:

- zwykłe funkcje globalne, które nie należą do żadnej klasy, albo
- funkcje członkowskie (metody) należące do innej klasy.

W praktyce wykorzystanie funkcji zaprzyjaźnionych może wspomagać i rozszerzać implementację mechanizmu hermetyzacji — ukrywania danych.

Poniżej przedstawiono dwa przykłady. W pierwszym z nich (przykład 17.1) wykorzystano funkcje zaprzyjaźnione, które nie należą do żadnej klasy, a w drugim (przykład 17.2) — funkcje zaprzyjaźnione, które są elementami członkowskimi innej klasy

Przykład 17.1

```
#include <iostream>
#define _USE_MATH_DEFINES // w celu użycia stałej M_PI, która nie jest zdefiniowana w standardowym C/C++
#include <cmath>
using namespace std;
// Definicja klasy Kolo:
class Kolo {
    // Deklaracja zmiennej członkowskiej (domyślnie prywatnej):
    double _r;
public:
    // Definicje konstruktorów:
    Kolo() {}
    Kolo(double r) {_r = r;
    }
    // Deklaracje (prototypy) funkcji członkowskich:
    void setPromien(double); // setter
}
```

```

double getPromien(); // getter
double pole();
double obwod();
// Prototypy funkcji zaprzyjaźnionych:
friend void setPromienFriendKolo(Kolo&, double);
friend double getPromienFriendKolo(Kolo);
/* UWAGA
 * Funkcja zaprzyjaźniona musi mieć co najmniej jeden parametr typu obiektowego należący do klasy,
 * której jest przyjacielem.
 * Wynika to z faktu, że funkcja zaprzyjaźniona uzyskuje dostęp do prywatnych zmiennych członkowskich klasy
 * za pośrednictwem obiektu będącego jej instancją, który jest argumentem jej wywołania.
 */
};

// Definicje metod członkowskich klasy Kolo:
void Kolo::setPromien(double r) {
    _r = r;
}
double Kolo::getPromien() {
    return _r;
}
double Kolo::pole() {
    return M_PI * _r * _r;
}
double Kolo::obwod() {
    return 2 * M_PI * _r;
}

// Definicje funkcji zaprzyjaźnionych klasy Kolo:
void setPromienFriendKolo(Kolo &kolo, double r) {
    kolo._r = r;
}
double getPromienFriendKolo(Kolo kolo) {
    return kolo._r;
}
/* UWAGA
 * Funkcje zaprzyjaźnione zostały zdefiniowane tutaj jako zwykłe funkcje globalne, które nie należą do żadnej klasy.
 * Definicje te mogą się znajdować w dowolnym miejscu programu.
 */

int main() {
    // Utworzenie obiektu kolo1 klasy Kolo:
    Kolo kolo1; // niejawne wywołanie konstruktora domyślnego
    // OBSŁUGA OBIEKTU ZA POMOCĄ METOD INSTANCYJNYCH
    // Nadanie wartości zmiennej prywatnej _r — wykorzystanie settera setPromien():
    kolo1.setPromien(1);
}

```

```

// Prezentacja wartości zmiennej _r — wywołanie gettera getPromien():
cout << "Promień koła: " << kolo1.getPromien() << endl;
// Wyświetlenie wartości pola i obwodu koła kolo1:
cout << "Pole wynosi " << kolo1.pole() << endl;
cout << "Obwód wynosi " << kolo1.obwod() << endl;

// Utworzenie obiektu kolo2 klasy Kolo:
Kolo kolo2; // niejawne wywołanie konstruktora domyślnego
// OBSŁUGA OBIEKTU ZA POMOCĄ FUNKCJI ZAPRZYJAŻNIONYCH
// Nadanie wartości zmiennej prywatnej _r — wykorzystanie funkcji zaprzyjaźnionej setPromienFriendKolo():
setPromienFriendKolo(kolo2, 2);
/* UWAGA
 * Obiekt kolo2 jest argumentem wywołania funkcji zaprzyjaźnionej setPromienFriendKolo().
 * Dostęp do prywatnej zmiennej członkowskiej _r klasy Kolo uzyskuje się za pośrednictwem tego właśnie obiektu.
 */
// Prezentacja wartości zmiennej _r — wykorzystanie funkcji zaprzyjaźnionej getPromienFriendKolo():
cout << "Promień koła: " << getPromienFriendKolo(kolo2) << endl;
/* UWAGA
 * Obiekt kolo2 jest argumentem wywołania funkcji zaprzyjaźnionej getPromienFriendKolo().
 */
// Wyświetlenie wartości pola i obwodu koła kolo1:
cout << "Pole wynosi " << kolo2.pole() << endl;
cout << "Obwód wynosi " << kolo2.obwod() << endl;

return 0;
}

```

W programie zdefiniowano klasę `Kolo` zawierającą deklarację prywatnej zmiennej członkowskiej `_r`. Dostęp do tej zmiennej spoza klasy `Kolo` (czyli z poziomu otoczenia klasy `Kolo`) można uzyskać za pośrednictwem publicznych metod dostępowych tej klasy — settera `setPromien()` i gettera `getPromien()`.

Ponadto w klasie `Kolo` zadeklarowano przy użyciu słowa kluczowego `friend` dwie funkcje będące przyjaciółmi tej klasy: `setPromienFriendKolo()` i `getPromienFriendKolo()`. Funkcjonalność funkcji zaprzyjaźnionej `setPromienFriendKolo()` odpowiada funkcjonalności settera `setPromien()`, a funkcji `setPromienFriendKolo()` — gettera `getPromien()`. Każda z wymienionych funkcji zaprzyjaźnionych ma parametr należący do typu obiektowego (klasy) `Kolo`.

Należy podkreślić, że zdefiniowane tutaj funkcje zaprzyjaźnione nie należą do żadnej klasy.

Funkcje zaprzyjaźnione `setPromienFriendKolo()` i `getPromienFriendKolo()` są wywoływanie w programie w sposób bezpośredni — bez konieczności użycia obiektu będącego instancją klasy `Kolo` jako pośrednika — jak w przypadku settera `setPromien()` i gettera `getPromien()`.

Z drugiej strony w każdym wywołaniu funkcji `setPromienFriendKolo()` i `getPromienFriendKolo()` występuje argument `kolo2`, który jest obiektem będącym instancją klasy `Kolo`. Obiekt

ten odgrywa rolę pośrednika w realizacji dostępu do prywatnej zmiennej członkowskiej `_r` klasy `Kolo` za pomocą funkcji zaprzyjaźnionych.

Omawiane tutaj funkcje zaprzyjaźnione rozszerzają interfejs klasy `Kolo` — stanowią komponent zewnętrzny tego interfejsu. Wspomagają one implementację mechanizmu hermetyzacji — ukrywania danych w klasie `Kolo`.

Pole i obwód są obliczane w programie dwukrotnie: pierwszy raz w odniesieniu do koła reprezentowanego przez obiekt `kolo1`, a drugi — do koła `kolo2`.

Ćwiczenie 17.1

Zmodyfikuj program zawarty w przykładzie 17.1 — zamiast klasy `Kolo` zdefiniuj klasę `Prostokat`. Klasa `Prostokat` powinna zawierać niezbędne deklaracje zmiennych i funkcji członkowskich pozwalających obliczyć pole i obwód prostokąta. Dane (parametry) prostokąta powinny być hermetyzowane (ukryte) przed światem zewnętrznym. Ponadto zdefiniuj zestaw funkcji zaprzyjaźnionych klasy `Prostokat`, których zadaniem jest wspomaganie i rozszerzenie implementacji mechanizmu hermetyzacji danych prostokąta. Zdefiniuj wspomniane funkcje jako globalne — niezależne od klasy `Prostokat`, czyli niebędące jej metodami.

Przykład 17.2

```
#include <iostream>
#define _USE_MATH_DEFINES // w celu użycia stałej M_PI, która nie jest zdefiniowana w standardowym C/C++
#include <cmath>
using namespace std;

// Deklaracja klasy Promien:
class Promien;

// Definicja klasy Kolo:
class Kolo {
public:
    // Deklaracje metod publicznych:
    double pole(Promien pPromien);
    double obwod(Promien pPromien);
    /* UWAGA
     * Metody pole() i obwod() stanowią funkcje zaprzyjaźnione klasy Promien.
    */
};

// Definicja klasy Promien:
class Promien {
private:
    double _r; // deklaracja prywatnej zmiennej członkowskiej
public:
```

```

// Definicje publicznych metod dostępowych do zmiennej prywatnej _r:
void setPromien(double r) { // setter
    _r = r;
}
double getPromien() { // getter
    return _r;
}

// Deklaracje (prototypy) funkcji zaprzyjaźnionych:
friend double Kolo::pole(Promien);
friend double Kolo::obwod(Promien);
/* UWAGA
* Zgodnie z deklaracjami powyżej definicje funkcji pole() i obwod() znajdują się w klasie Kolo.
*/

};

// Definicje metod pole() i obwod() z klasy Kolo:
double Kolo::pole(Promien pPromien) {
    return M_PI * pPromien._r * pPromien._r;
}
double Kolo::obwod(Promien pPromien) {
    return 2 * M_PI * pPromien._r;
}

int main() {
    // Utworzenie obiektu promien będącego instancją klasy Promien:
    Promien promien;
    promien.setPromien(1);

    // Utworzenie obiektu kolo należącego do klasy Kolo:
    Kolo kolo;
    // Obliczenie i wyświetlenie pola i obwodu koła:
    cout << "Pole wynosi " << kolo.pole(promien) << endl;
    cout << "Obwód wynosi " << kolo.obwod(promien) << endl;
/* UWAGA
* Powyżej wywołano metody pole() i obwod(), które stanowią zaprzyjaźnione klasy Promien.
*/

    return 0;
}

```

Funkcjonalność programu zawartego w niniejszym przykładzie jest podobna do funkcjonalności programu z przykładu 17.1. Jednakże tutaj obliczane są pole i obwód koła nie dla dwóch wartości promienia, a dla jednej.

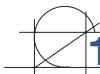
Najważniejsza różnica pomiędzy programem zaprezentowanym tutaj a programem z przykładu 17.1 tkwi w ich implementacji. Mianowicie tutaj zdefiniowano dwie klasy: `Promien` i `Kolo`.

Klasa `Promien` zawiera deklarację prywatnej zmiennej członkowskiej `_r` oraz definicje publicznych metod dostępowych: settera `setPromien()` i gettera `getPromien()`. Oprócz tego w klasie `Promien` zamieszczono deklaracje (prototypy) dwóch funkcji zaprzyjaźnionych: `pole()` i `obwod()`. Tym samym wymienione funkcje to funkcje zaprzyjaźnione klasy `Promien`. Należy zwrócić uwagę na to, że funkcje `pole()` i `obwod()` nie są metodami instancjnymi klasy `Promien`.

W klasie `Kolo` zdefiniowano dwie metody: `pole()` i `obwod()`, których zadaniem jest obliczenie, odpowiednio, pola i obwodu koła. Metody te są elementami członkowskimi klasy `Kolo` i jednocześnie — o czym już wspomniano — funkcjami zaprzyjaźnionymi klasy `Promien`.

Ćwiczenie 17.2

Zmodyfikuj program zawarty w przykładzie 17.2 — zdefiniuj dodatkową klasę `Kula` zawierającą metody `objetosc()` i `polePowierzchni()`, które umożliwiają obliczenie, odpowiednio, objętości i pola powierzchni kuli. W definicji klasy `Promien` zadeklaruj funkcje zaprzyjaźnione `objetosc()` i `polePowierzchni()`, które zostały zdefiniowane w klasie `Kula`, jako jej funkcje członkowskie. Wykonaj testy poprawności działania programu dla kuli o zadanym promieniu 1.



17.2. Klasy zaprzyjaźnione

W ogólności **klasa zaprzyjaźniona** (ang. *friend class*) to klasa, której obiekty (jako jej instancje) mogą uzyskać dostęp do prywatnych i chronionych elementów członkowskich innej klasy — tej, w której została zadeklarowana jako jej „przyjaciel” (ang. *friend*).

Tym samym prywatne i chronione składniki określonej klasy mogą być przetwarzane na poziomie jej otoczenia nie tylko za pomocą publicznych metod członkowskich tej klasy — setterów i getterów, ale również za pośrednictwem metod należących do jej klasy zaprzyjaźnionej.

Bardzo ważną cechą mechanizmu „przyjaźni” pomiędzy klasami jest to, że dana klasa może mieć wiele klas zaprzyjaźnionych.

Wykorzystując w praktyce omówiony powyżej mechanizm przyjaźni pomiędzy klasami, należy pamiętać o kilku ważnych zasadach, mianowicie:

- Przyjaźń nie podlega dziedziczeniu.
- Przyjaźń pomiędzy klasami nie jest automatycznie wzajemna — dwukierunkowa (ang. *mutual*). Oznacza to, że jeżeli klasa A jest przyjacielem klasy B, to odwrotna relacja przyjaźni nie zachodzi w sposób automatyczny.
- Zadeklarowanie zbyt wielu relacji przyjaźni pomiędzy klasami może znacznie skomplikować i zaciemnić zaimplementowany mechanizm hermetyzacji danych w tych klasach.

Przykład 17.3

```
#include <iostream>
#define _USE_MATH_DEFINES // w celu użycia stałej M_PI, która nie jest zdefiniowana w standardowym C/C++
#include <cmath>
using namespace std;

// Definicja klasy Promien:
class Promien {
    // Deklaracja prywatnej zmiennej członkowskiej:
    double _r;
public:
    // Prototypy publicznych funkcji dostępowych:
    void setPromien(double); // setter
    double getPromien(); // getter

    // Deklaracje klas zaprzyjaźnionych:
    friend class Kolo;
    friend class Kula;
/* UWAGA
 * Powyższe deklaracje z użyciem słowa kluczowego friend skutkują ustaleniem relacji przyjaźni pomiędzy klasą
 * Promien a klasami Kolo i Kula. Klasy Kolo i Kula stanowią klasy zaprzyjaźnione klasy Promien.
 */
};

// Definicje metod klasy Promien:
void Promien::setPromien(double r) {
    _r = r;
}
double Promien::getPromien() {
    return _r;
}

// Definicja klasy Kolo:
class Kolo {
/* UWAGA
 * Klasa Kolo jest klasą zaprzyjaźnioną klasy Promien.
 */
public:
    double pole(Promien);
    double obwod(Promien);
};

// Definicje metod klasy Kolo:
double Kolo::pole(Promien promien) {
    return M_PI * promien._r * promien._r;
/* UWAGA
 * W ciele funkcji członkowskiej pole() należącej do klasy Kolo, która jest „przyjacielem” klasy Promien,
 * wykorzystano prywatną zmienną członkowską _r zdefiniowaną w klasie Promien.
 */
}
```

```

double Kolo::obwod(Promien promien) {
    return 2 * M_PI * promien._r;
/* UWAGA
 * Metoda obwod() z klasy Kolo, będącej „przyjacielem” klasy Promien, wykorzystuje prywatną zmienną
 * członkowską _r zdefiniowaną w klasie Promien.
*/
}

// Definicja klasy Kula, która jest zaprzyjaźniona z klasą Promien:
class Kula {
public:
    double objetosc(Promien);
    double pole(Promien);
};

// Definicje metod klasy Kula:
double Kula::objetosc(Promien promien) {
    return double(4)/double(3) * M_PI * promien._r * promien._r * promien._r;
/* UWAGA
 * W treści funkcji członkowskiej objetosc() należącej do klasy Kula (która jest „przyjacielem” klasy Promien)
 * wykorzystano prywatną zmienną członkowską _r zdefiniowaną w klasie Promien.
*/
}

double Kula::pole(Promien promien) {
    return 4 * M_PI * promien._r * promien._r;
}

int main() {
    // Utworzenie obiektu promien jako instancji klasy Promien:
    Promien promien;
    // Utworzenie obiektu kolo:
    Kolo kolo; // Obiekt kolo jest instancją klasy Kolo, która jest „przyjacielem” klasy Promien.
    // Ustalenie promienia koła na 1:
    promien.setPromien(1); // wywołanie metody instancjowej klasy Promien
    // Obliczenie i prezentacja pola i obwodu koła dla zadanego promienia:
    cout << "Pole koła wynosi: " << kolo.pole(promien) << endl;
    cout << "Obwód koła wynosi: " << kolo.obwod(promien) << endl;
/* UWAGA
 * Za pośrednictwem metod pole() i obwod() obiektu kolo, będącego instancją klasy Kolo (która jest klasą
 * zaprzyjaźnioną klasy Promien), uzyskano dostęp do prywatnej zmiennej członkowskiej _r zdefiniowanej
 * w klasie Promien.
 * Należy zwrócić uwagę, że argument wywołania metod pole() i obwod() jest obiektem typu promien klasy Promien.
*/
}

// Utworzenie obiektu kula:
Kula kula; // obiekt kula jest instancją klasy Kula, która jest „przyjacielem” klasy Promien
promien.setPromien(2); // wywołanie metody instancjowej klasy Promien
cout << "Objętość kuli wynosi: " << kula.objetosc(promien) << endl;
cout << "Pole powierzchni kuli wynosi: " << kula.pole(promien) << endl;

```

```

/* UWAGA
 * Dostęp do prywatnej zmiennej członkowskiej _r zdefiniowanej w klasie Promien uzyskano za pomocą metod
 * objetosc() i pole() obiektu kula, będącego instancją klasy Kula. Przy tym klasa Kula jest klasą
 * zaprzyjaźnioną klasą Promien.
 * Obiektem promien klasy Promien jest argument wywołania metod objetosc() i pole().
 */
}

return 0;
}

```

W programie zdefiniowano klasę `Promien`, która zawiera deklarację prywatnej zmiennej członkowskiej `_r`. Dostęp do tej zmiennej z poziomu otoczenia klasy `Promien` można uzyskać za pomocą publicznych metod dostępowych należących do tej klasy: settera `setPromien()` i gettera `getPromien()`.

Oprócz tego definicja klasy `Promien` zawiera deklaracje dwóch klas zaprzyjaźnionych, `Kolo` i `Kula`: `friend class Kolo; friend class Kula;`. Oznacza to, że dostęp do prywatnej zmiennej `_r` należącej do klasy `Promien` można zrealizować również za pośrednictwem publicznych metod należących do wymienionych klas zaprzyjaźnionych, `Kolo` i `Kula`. W szczególności dostęp ten uzyskano za pośrednictwem metod `pole()` i `obwod()` — elementów członkowskich klasy `Kolo`, oraz metod `objetosc()` i `pole()` — elementów członkowskich klasy `Kula`.

Argumentem wywołania każdej z wymienionych metod jest obiekt `promien` należący do klasy `Promien`.

Ćwiczenie 17.3

Zmodyfikuj program zawarty w przykładzie 17.3 — zamiast relacji przyjaźni pomiędzy klasą `Promien` a klasami `Kolo` i `Kula` wykorzystaj mechanizm dziedziczenia. Załącz, że klasy `Kolo` i `Kula` to klasy pochodne klasy `Promien`, która odgrywa rolę klasy bazowej. Uwzględnij hermetyzację — ukrycie danych w klasie `Promien`. Przeprowadź analizę porównawczą wykonanego programu z programem z przykładu 17.3. Omów wyniki tej analizy.



17.3. Pytania i zadania kontrolne

17.3.1. Pytania

- Podaj definicję funkcji zaprzyjaźnionej. W jakim celu wykorzystuje się w praktyce funkcje zaprzyjaźnione?
- Co oznacza termin „klasa zaprzyjaźniona”?
- Czy relacje przyjaźni pomiędzy klasami podlegają dziedziczeniu?
- Omów wykorzystanie relacji przyjaźni zachodzącej pomiędzy:
 - funkcją a klasą,
 - między klasami

w implementacji mechanizmu hermetyzacji — ukrywania danych.

17.3.2. Zadania

1. Napisz program pozwalający obliczyć objętość, pole powierzchni całkowitej oraz długości wszystkich krawędzi prostopadłościanu. Dane wejściowe mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Wykorzystaj klasę `Prostopadloscian` zawierającą deklaracje i definicje wymaganych komponentów. Uwzględnij hermetyzację — ukrycie danych prostopadłościanu. Dostęp do prywatnych danych prostopadłościanu uzyskaj za pośrednictwem funkcji zaprzyjaźnionych z klasą `Prostopadloscian`. Przy czym wspomniane funkcje zaprzyjaźnione zdefiniuj jako funkcje globalne — niezależne od klasy `Prostopadloscian` (nienależące do żadnej klasy). Wykonaj testy działania programu dla prostopadłościanu o parametrach: długość podstawy 1, szerokość podstawy 2, wysokość 3.
2. Zrób tak jak w zadaniu 1., z tym że funkcje zaprzyjaźnione z klasą `Prostopadloscian` zdefiniuj jako metody instancyjne innej klasy.
3. Napisz program pozwalający przetwarzać dane pracownika i ucznia szkoły. Uwzględnij następujące dane pracownika: imię, nazwisko, stanowisko oraz następujące dane ucznia: imię, nazwisko, klasę. Wykorzystaj zdefiniowane samodzielnie klasy: `Osoba`, `Pracownik` i `Uczeń`, które nie będą ze sobą powiązane żadnymi relacjami dziedziczenia. Klasy `Pracownik` i `Uczeń` powinny być klasami zaprzyjaźnionymi klasą `Osoba`. Zastosuj mechanizm hermetyzacji — ukrywania danych w poszczególnych klasach. Dostęp do prywatnych danych klasy `Osoba` zrealizuj za pośrednictwem publicznych metod członkowskich należących do klas zaprzyjaźnionych — `Pracownik` i `Uczeń`. Wykonaj testy działania programu dla przykładowych danych pracownika i ucznia.
4. Zrób tak jak w zadaniu 3., z tym że zamiast relacji przyjaźni pomiędzy klasą `Osoba` a klasami `Pracownik` i `Uczeń` wykorzystaj mechanizm dziedziczenia — dziedziczenie hierarchiczne. Załącz, że klasą bazową jest klasa `Osoba`, a jej klasy pochodne to `Pracownik` i `Uczeń`. Dostęp do chronionych danych w poszczególnych klasach zrealizuj za pomocą funkcji zaprzyjaźnionych tych klas.

18

Szablony funkcji i klas

18.1. Szablony funkcji

18.1.1. Definiowanie szablonów funkcji

Szablon funkcji (ang. *function template*) to model funkcji, który można wykorzystać jako wzorzec (ang. *pattern*) do tworzenia innych funkcji o takiej samej funkcjonalności.

Ogólna idea konstrukcji szablonów funkcji opiera się na założeniu, aby na etapie ich definiowania nie określać w sposób jednoznaczny typów ich parametrów formalnych ani typu wartości zwracanych przez nie na zewnątrz. Zamiast na konkretnych — istniejących — typach danych (np. `int`, `float`) szablony funkcji operują na danych należących do **typów uogólnionych** (ang. *generic data types*).

UWAGA

W wielu publikacjach **szablony/wzorce funkcji** (ang. *function templates*) są nazywane **funkcjami szablonowymi/wzorcowymi** (ang. *template functions*), a także **funkcjami uogólnionymi** (ang. *generic functions*).

Jak wspomniano wcześniej, szablony funkcji pozwalają realizować tę samą funkcjonalność (wykonać ten sam kod) na danych należących do różnych typów. Dane te są reprezentowane przez:

- parametry uogólnione,
- uogólniony typ wyniku zwracanego przez funkcję na zewnątrz.

Z tego względu szablony funkcji należy stosować w sytuacjach, w których dany — identyczny — kod źródłowy musiałby zostać powielony dla kilku różnych typów danych.

W szablonie (wzorcu) funkcji wykorzystuje się tzw. **parametry szablonowe** (ang. *template parameters*), nazywane również **parametrami uogólnionymi** (ang. *generic parameters*). To specjalny rodzaj parametrów formalnych reprezentujących typ/typy danych, które na etapie definiowania szablonu nie zostały (jeszcze) określone.

W praktyce wykorzystanie parametrów szablonowych pozwala na dostarczenie do funkcji szablonowej w chwili jej wywołania **skonkretyzowanych typów danych** (ang. *instantiated data types*), np. `int`, `float` — tak samo jak innych, „zwykłych” danych. Parametry szablonowe wzorca funkcji są traktowane przez kompilator w taki sam sposób jak zwykłe parametry formalne funkcji.

Ogólna postać deklaracji parametru szablonowego wzorca funkcji jest następująca:

`template <typename T>`

lub:

`template <class T>`

gdzie:

- `template` — słowo kluczowe, które rozpoczyna definicję szablonu funkcji,
- `typename/class` — słowo kluczowe określające parametr szablonowy,
- `T` — identyfikator uogólnionego typu danych.

Słowa kluczowego `class` można używać wymiennie ze słowem kluczowym `typename` — nie ma żadnego znaczenia, które z nich zostanie wykorzystane. Zadeklarowana nazwa uogólnionego typu danych jest dowolna. Jednakże powszechnie przyjętym i akceptowanym identyfikatorem typu uogólnionego jest `T`.

W ogólności szablon (wzorzec) funkcji może mieć jeden lub więcej parametrów szablonowych. Jeżeli parametrów szablonowych jest kilka, oddziela się je od siebie przecinkami. Na przykład deklaracja `<typename T1, typename T2>` opisuje dwa parametry szablonowe reprezentujące typy uogólnione `T1` i `T2`.

18.1.2. Wywołanie funkcji szablonowej

W języku C++ funkcje szablonowe nie są kompliwane w sposób bezpośredni. Jeśli kompilator natrafia w kodzie źródłowym na wywołanie funkcji szablonowej, tworzy replikę (kopię) tej funkcji, w której parametry szablonowe (tj. parametry należące do typów uogólnionych) zastępuje parametrami należącymi do istniejących, konkretnych typów danych, np. `int`, `float`. Proces ten jest nazywany **konkretyzacją szablonu** (ang. *template instantiation*). Kopie funkcji szablonowych z faktycznymi (skonkretyzowanymi) typami danych są nazywane **instancjami szablonów funkcji** (ang. *function template instances*).

W wywołaniach funkcji szablonowych argumenty odpowiadające parametrom szablonowym można określić w dwojakim sposobie: jawnym (ang. *explicitly*) lub niejawnym (ang. *implicitly*).

Jawne — bezpośrednie — skonkretyzowanie typów szablonowych skutkuje konkretyzacją szablonu, która jest niezależna od typów argumentów odpowiadających zwykłym danym przekazanym do funkcji i z funkcji. Typy danych instancji szablonu funkcji są w tym przypadku zależne wyłącznie od skonkretyzowanych typów uogólnionych.

W drugim przypadku, tj. jeśli skonkretyzowane typy uogólnione nie są jawnie dostarczone do funkcji szablonowej w jej wywołaniu, kompilator utworzy instancję szablonu w sposób automatyczny — na podstawie typów argumentów odpowiadających „zwykłym” danym.

Oprócz tego, jeśli typy argumentów wywołania funkcji szablonowej są zależne wyłącznie od typów parametrów szablonu, kompilator samodzielnie wybierze typ wartości zwracanej przez funkcję szablonową na zewnątrz. Proces ten nazywa się **dedukcją typu** (ang. *data type deduction*).

Przykład 18.1

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

// Definicja szablonu — wzorca funkcji polePr():
template <typename T> // deklaracja parametru T szablonu
T polePr(T pBok1, T pBok2) {
    return (pBok1 * pBok2);
}

/* UWAGA
* Szablon (wzorzec) funkcji polePr() został sparametryzowany za pomocą jednego parametru szablonowego,
* którym jest uogólniony typ danych o nazwie T.
* Oprócz parametru szablonowego T funkcja szablonowa polePr() ma dwa „zwykłe” parametry wejściowe,
* odpowiadające danym (długościom boków prostokąta) przekazywanym do funkcji: pBok1 i pBok2.
* Parametry te należą do uogólnionego typu danych T.
* Wartość zwracana na zewnątrz funkcji również należy do typu uogólnionego T.
*/

// Definicja szablonu — wzorca funkcji polePr():
template <typename T> // deklaracja parametru szablonowego T
T obwodPr(T pBok1, T pBok2) {
    return (2 * pBok1 + 2 * pBok2);
}

int main() {
    float bok1 {1.5}, bok2 {2.5}; // zmienne pomocnicze reprezentujące boki prostokąta
    // Wywołania funkcji szablonowych polePr() i obwodPr()
    cout << "Pole wynosi " << polePr<int>(bok1, bok2) << endl;
    cout << "Obwód wynosi " << obwodPr<int>(bok1, bok2) << endl;
}
```

```

/* UWAGA
 * Wywołania funkcji szablonowych polePr() i obwodPr() mają po trzy argumenty (czyli parametry aktualne).
 * Pierwszym z nich jest typ int. Tym samym typ int jest skonkretyzowanym parametrem szablonowym,
 * czyli skonkretyzowanym typem uogólnionym T.
 * Dwa pozostałe argumenty (bok1 i bok2) należą do typu float, co wynika bezpośrednio z ich deklaracji.
 * Jednakże instancje szablonów funkcji polePr() i obwodPr() zostaną wygenerowane dla skonkretyzowanego
 * typu szablonowego int, ponieważ właśnie typ int został przekazany do obu funkcji w ich wywołaniach
 * w sposób jawnego — bezpośredni. Dlatego też kompilator wykona niejawną konwersję typu argumentów
 * bok1 i bok2 z typu float na typ int.
 */
cout << endl;
cout << "Pole wynosi " << polePr(bok1, bok2) << endl;
cout << "Obwód wynosi " << obwodPr(bok1, bok2) << endl;
/* UWAGA
 * Wywołania funkcji szablonowych polePr() oraz obwodPr() zostały określone dla argumentów (danych)
 * należących do typu float. Jednocześnie argument odpowiadający parametrowi szablonowemu T nie jest
 * przekazywany do funkcji w sposób bezpośredni (jawnym). Dlatego też w celu obsługi odpowiedniego typu
 * danych kompilator wykona konkretyzację funkcji szablonowych polePr() oraz obwodPr() w sposób
 * automatyczny („po cichu”) — na podstawie typu argumentów, czyli typu float.
 */
return 0;
}

```

W programie zdefiniowano szablony dwóch funkcji: `polePr()` i `obwodPr()`. Każdy z nich ma jeden parametr szablonowy — uogólniony typ danych o nazwie T .

Dzięki wykorzystaniu uogólnionego typu danych jako parametru szablonowego funkcje szablonowe `polePr()` i `obwodPr()` można zastosować do danych wejściowych należących do różnych typów. W programie zademonstrowano to dla dwóch typów danych: `int` i `float`.

Najpierw szablony funkcji `polePr()` i `obwodPr()` zostały skonkretyzowane dla typu danych `int`. Wynika to bezpośrednio z postaci ich wywołań — tj. `polePr<int>(bok1, bok2)` oraz `obwodPr<int>(bok1, bok2)` — w których uogólniony typ danych T skonkretyzowano jako `int` w nawiasach kątowych (ang. *angle brackets*, `<>`), w jawnym sposobie.

Kolejne wywołania funkcji szablonowych: `polePr(bok1, bok2)` oraz `obwodPr(bok1, bok2)`, już nie zawierają jawnego określenia typu szablonowego. Tym samym kompilator utworzy instancje szablonów omawianych funkcji w sposób automatyczny — na podstawie typu argumentów tych wywołań (`float`). Typ wyników funkcji `polePr()` i `obwodPr()` przekazywanych do otoczenia również zostanie wydedukowany przez kompilator automatycznie, ponieważ typy argumentów wywołań tych funkcji są zależne wyłącznie od typów parametrów szablonowych (czyli od typu uogólnionego T).

Ćwiczenie 18.1

Zmodyfikuj program zawarty w przykładzie 18.1 — zamiast obliczać pole i obwód prostokąta, wyznacz pole i obwód koła. Wykorzystaj zdefiniowane samodzielnie szablony funkcji `poleKo()` i `obwodKo()`. W wywołaniach wymienionych funkcji szablonowych w programie głównym nie określaj w sposób bezpośredni (jawnego) typu argumentu odpowiadającego parametrowi szablonowemu. Omów uzyskane rezultaty.

18.1.3. Specjalizowanie szablonów funkcji

Z materiału omówionego wcześniej w tym rozdziale wynika jednoznacznie, że wszystkie instancje szablonu funkcji tworzone przez kompilator dla każdego wywołania funkcji szablonowej mają taką samą implementację (kod źródłowy). Instancje te różnią się od siebie jedynie typami danych, które odpowiadają skonkretyzowanym typom szablonowym.

Jednakże w praktyce niekiedy jest konieczne zróżnicowanie szczegółów implementacji funkcji opartych na danym szablonie w zależności od typu/typów danych. W takim przypadku funkcja szablonowa może mieć inną implementację dla danych należących np. do typu rzeczywistego `float`, a inną np. dla danych typu `int`. Można to zrealizować z zastosowaniem tzw. specjalizacji szablonu funkcji.

Specjalizacja szablonu funkcji (ang. *function template specialization*) polega na zdefiniowaniu wersji specjalnej funkcji szablonowej — specjalizowanej — dla konkretnego, istniejącego typu danych, np. dla typu `float` czy `int`. Wspomniana wersja funkcji szablonowej specjalizowana dla określonego typu danych jest nazywana **funkcją specjalizowaną** (ang. *specialized function*). Implementacja funkcji specjalizowanej może (i powinna) być różna od implementacji instancji szablonu funkcji.

W definicji specjalizowanej funkcji szablonowej należy poinformować kompilator, że definicja ta dotyczy funkcji, która nie ma parametrów szablonowych (jej lista parametrów szablonowych jest pusta: `template <>`). Ponadto w nagłówku funkcji specjalizowanej zamiast parametru szablonowego reprezentującego uogólniony typ danych należy podać nazwę konkretnego typu danych (`float`, `int` itp.), dla którego szablon jest specjalizowany.

Przykład 18.2

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

// Definicja szablonu funkcji srednia():
template <typename T> // określenie parametru szablonowego
T srednia(T wyklad, T cwiczenia, T laboratorium, T seminarium) {
    return (wyklad + cwiczenia + laboratorium + seminarium) / 4;
}
```

```

// Definicja specjalizowanej wersji szablonu funkcji srednia() dla typu danych double:
template <>
double srednia(
    double wyklad,
    double cwiczenia,
    double laboratorium,
    double seminarium) {
    return (
        round(wyklad) + round(cwiczenia) + round(laboratorium) +
        round(seminarium)) / 4;
}

// Definicja specjalizowanej wersji szablonu funkcji srednia() dla typu danych float:
template <>
float srednia(
    float wyklad,
    float cwiczenia,
    float laboratorium,
    float seminarium) {
    return (
        round(wyklad) + round(cwiczenia) + round(laboratorium) +
        round(seminarium)) / 4;
}

/* UWAGA
 * Specjalizacje szablonu funkcji srednia() dla typów danych double i float zawierają inny kod źródłowy niż sam szablon,
 * czyli funkcja uogólniona. Innymi słowy, funkcja uogólniona i funkcje specjalizowane różnią się od siebie
 * implementacją.
 */

int main() {
    // Wywołanie funkcji szablonowej srednia():
    cout << "Średnia z ocen: " << srednia(3, 4, 3, 4) << endl;
    /* UWAGA
     * Kompilator, napotykając wywołanie funkcji szablonowej srednia() dla argumentów typu int, najpierw
     * sprawdza, czy istnieje wersja specjalizowana szablonu tej funkcji dla tego typu danych.
     * Jeśli kompilator nie znajdzie wersji specjalizowanej funkcji dla typu danych int, konkretyzuje szablon
     * funkcji (funkcji uogólnionej) dla tego typu danych.
     */
    // Ponowne wywołanie funkcji szablonowej srednia():
    cout << "Średnia z ocen: " << srednia<double>(3, 4, 3, 4) << endl;
    /* UWAGA
     * W wywołaniu funkcji szablonowej srednia() wyszczególniono w jawnym sposób argument szablonowy double.
     * Zatem uogólniony typ danych T został w jawnym sposób skonkretyzowany jako typ double.
     * Typ argumentów z pozostałymi danymi, int, ma niższy priorytet od typu argumentu szablonowego.
     */
}

```

- * Kompilator w pierwszej kolejności poszukuje wersji specjalizowanej szablonu funkcji `srednia()` dla typu danych `double`.
- * Funkcja specjalizowana dla typu `double` istnieje — zatem właśnie ona zostanie wykonana.
- */

```
// Ponowne wywołanie funkcji szablonowej srednia():
cout << "Średnia z ocen: " << srednia(3.0, 4.0, 3.0, 4.0) << endl;
/* UWAGA
 * Kompilator w pierwszej kolejności poszukuje wersji specjalizowanej szablonu funkcji srednia() dla typu danych
 * double, co wynika z typu, do którego należą argumenty wywołania tej funkcji.
 * Kompilator znajduje wersję specjalizowaną szablonu funkcji srednia() dla typu danych double i ją wykonuje.
 */
return 0;
}
```

W programie zdefiniowano szablon funkcji `srednia()`. Wyróżniono w nim jeden parametr szablonowy, którym jest uogólniony typ danych T .

Konkretyzacje wspomnianego typu T dla różnych typów danych (np. dla typów `int`, `long`) pozwalają na wyznaczenie średniej arytmetycznej ocen uzyskanych na uczelni technicznej z określonego przedmiotu, na który składają się: wykład, ćwiczenia, laboratorium i seminarium. Przy założeniu, że parametry: `wyklad`, `ćwiczenia`, `laboratorium` i `seminarium` należą do typu całkowitego, obliczenie średniej jest realizowane z zastosowaniem dzielenia całkowitego.

Oprócz tego w programie zdefiniowano dwie specjalizacje szablonu funkcji `srednia()` dla parametrów należących do typów rzeczywistych. Pierwsza z nich dotyczy typu danych `double`, a druga — typu `float`.

Funkcja szablonowa `srednia()` jest wywoływana w programie głównym trzykrotnie. W pierwszym wywołaniu, tj. `srednia(3, 4, 3, 4)`, argumenty z danymi (odpowiadającymi parametrom: `wyklad`, `ćwiczenia`, `laboratorium` i `seminarium`) należą do typu całkowitego `int`. Zarazem argument szablonowy nie został określony w jawnym — bezpośredni — sposób. Kompilator, napotkawszy omawiane wywołanie, będzie poszukiwał specjalizacji szablonu dla typu danych `int`. Tutaj takiej specjalizacji nie znajdzie, czego skutkiem będzie utworzenie instancji szablonu funkcji `srednia()` dla skonkretyzowanego typu danych `int`. Wykonana zostanie właśnie ta instancja.

W drugim wywołaniu funkcji szablonowej `srednia()`, tj. `srednia<double>(3, 4, 3, 4)`, argument szablonowy został ustalony w sposób jawnym jako `double`. Tym samym uogólniony typ danych T został skonkretyzowany jako `double`. Dlatego też kompilator będzie poszukiwał wersji specjalizowanej funkcji `srednia()` przeznaczonej dla typu `double`. Po jej znalezieniu zostanie wykonana właśnie ta specjalizowana wersja funkcji szablonowej.

W trzecim wywołaniu funkcji szablonowej `srednia()`, tj. `srednia(3.0, 4.0, 3.0, 4.0)`, parametr szablonowy nie został określony w sposób jawnym. Zarazem typy argumentów odpowiadających parametrom: `wyklad`, `ćwiczenia`, `laboratorium` i `seminarium` zostały w niejawnym

sposób ustalone jako `double`. Tym samym również w tym przypadku zostanie wykonana funkcja specjalizowana dla typu `double`.

Ćwiczenie 18.2

Zmodyfikuj program zawarty w przykładzie 18.2 — nie zmieniając wartości argumentów wywołań funkcji szablonowej `srednia()` jako literałów, zmodyfikuj dwie ostatnie postacie tych wywołań. Celem wspomnianej modyfikacji jest to, aby dla każdego z tych wywołań wykonywana była instancja szablonu funkcji `srednia()` skonkretyzowana dla typu danych `int`.



18.2. Szablony klas

18.2.1. Definiowanie szablonów klas

W programowaniu zdarza się tak, że zdefiniowano kilka klas o takiej samej implementacji i jednocześnie wspomniane klasy różnią się od siebie wyłącznie typami danych. Podobny problem występuje wówczas, gdy w obrębie jednej klasy zdefiniowano zmienne i funkcje członkowskie należące do różnych typów, ale o analogicznej funkcjonalności. Wymienione przypadki powodują niepotrzebne rozbudowanie kodu źródłowego oraz zapowiadają duże trudności w przyszłej konserwacji — utrzymaniu tego kodu.

Omówione powyżej problemy można rozwiązać z wykorzystaniem tzw. szablonów — wzorców klas, które wspomagają zastosowanie tego samego kodu do różnych typów danych. **Szablony klas** (ang. *class template*) to **klasa uogólniona** (ang. *generic class*), która pozwala zdefiniować na jej podstawie całą rodzinę klas w zależności od typu/typów danych, do których należą jej zmienne członkowskie i na których operują jej metody.

Szablony klas — analogicznie do szablonów funkcji — są **parametryzowane** (ang. *parameterized*) za pomocą **parametrów szablonowych** (ang. *template parameters*). Rolę parametrów szablonowych odgrywają **uogólnione typy danych** (ang. *generic data types*).

UWAGA

Szablony klas mogą być również parametryzowane za pomocą parametrów należących do „zwykłych” typów danych, np. `int`, `float`. Będzie o tym mowa w dalszej części tego podrozdziału.

Postać ogólna definicji szablonu klasy jest następująca:

```
template <class T>
class nazwa_klasy {
    deklaracje_i_definicje_elementow_czlonkowskich;
}
```

gdzie:

- `template` — słowo kluczowe, które rozpoczyna definicję szablonu klasy,
- `class T` — deklaracja parametru szablonowego określonego za pomocą uogólnionego typu danych `T`,
- `nazwa_klasy` — identyfikator szablonu,
- `deklaracje_i_definicje_elementów_członkowskich` — deklaracje i definicje elementów członkowskich szablonu klasy.

Dany szablon klasy może mieć jeden lub więcej parametrów szablonowych, np. `T1, T2, T3`. Elementy członkowskie szablonu klasy, tj. zmienne i funkcje członkowskie, definiuje się w taki sam sposób jak w przypadku zwykłej klasy. Analogicznie funkcje członkowskie szablonu można deklarować w obrębie (we wnętrzu) tego szablonu, a ich definicje umieszczać poza nim.

Parametryzacja szablonu klasy powoduje, że jego elementy składowe „same w sobie” stanowią szablony, ponieważ są (z definicji) parametryzowane parametrami szablonu.

W celu utworzenia obiektu należącego do „rzeczywistej” klasy utworzonej na podstawie szablonu kompilator generuje tę klasę jako **instancję szablonu** (ang. *template instance*) skonkretyzowaną dla określonego typu danych. Przy tworzeniu instancji szablonu klasy należy podać jej argumenty odpowiadające parametrom szablonowym w jego definicji. Na przykład instrukcja `Prostokat<int> prostokat;` powoduje wygenerowanie przez kompilator instancji szablonu klasy `Prostokat` skonkretyzowanego dla typu danych `int` — czyli konkretnej klasy, w której ciele uogólniony typ szablonu został zastąpiony typem całkowitym `int`. Następnie utworzony zostaje obiekt `prostokat`, jako instancja tej klasy.

W ogólności szablon klasy może, co już zostało powiedziane, zawierać większą liczbę parametrów typu uogólnionego. Jednakże szablony klas mogą być również parametryzowane parametrami należącymi do „zwykłych”, istniejących typów danych, np. `int, float, string`.

Jeżeli dany szablon klasy ma zarówno parametry szablonowe typów uogólnionych, jak i parametry należące do konkretnych, istniejących typów danych, w trakcie konkretyzacji tego szablonu, czyli podczas tworzenia jego instancji, należy określić dwa rodzaje argumentów. Pierwszy rodzaj argumentów stanowią konkretne typy danych (np. `int, float`) odpowiadające parametrom szablonowym należącym do typów uogólnionych, np. `T1, T2`. Drugi rodzaj omawianych argumentów to te „niebędące typami danych” (ang. *non-type arguments*). Argumenty te odpowiadają parametrom szablonowym należącym do „zwykłych” typów danych. Z definicji powinny one być stałymi lub wyrażeniami dającymi w wyniku stałe.

Na przykład wykonanie instrukcji `Tablica<int, 10> tablica;` spowoduje wygenerowanie klasy będącej konkretyzacją szablonu klasy `Tablica` dla typu danych `int` oraz wartości `10`, a następnie utworzenie obiektu `tablica` jako instancji tej klasy. Wartość `10` to argument, który „nie jest typem danych”.

Przykład 18.3

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

// Definicja szablonu klasy Prostokat:
template <class T>
/* UWAGA
* Parametr szablonu klasy Prostokat jest uogólnionym typem danych T. To jedyny parametr tego szablonu.
*/
class Prostokat {
public:
    // Deklaracje (szablonów) zmiennych członkowskich:
    T bok1, bok2;
    // Definicja (szablonu) metody pole():
    T pole() {
        return bok1 * bok2;
    }
    // Deklaracja (szablonu) metody obwod():
    T obwod();
};

// Definicja (szablonu) metody obwod():
template <class T>
T Prostokat<T>::obwod() {
    return 2 * bok1 + 2 * bok2;
}

int main() {
    // Utworzenie obiektu prostokat1:
    Prostokat<float> prostokat1 {1, 2}; // inicjalizacja listowa bezpośrednia
    /* UWAGA
    * Szablon klasy Prostokat został skonkretyzowany dla typu rzeczywistego float. Innymi słowy, została utworzona
    * instancja szablonu klasy Prostokat skonkretyzowana dla typu float.
    * Literaly całkowite 1 oraz 2, będące argumentami wywołania konstruktora obiektu prostokat1, zostają poddane
    * niejawnnej konwersji na literaly rzeczywiste (konwersja rozszerzająca).
    */
    cout << "Pole wynosi " << prostokat1.pole() << endl; // 2
    cout << "Obwód wynosi " << prostokat1.obwod() << endl; // 6
    /* UWAGA
    * Funkcje członkowskie pole() i obwod() zwracają na zewnątrz wartości typu float. Wynika to z faktu,
    * że kompilator jest w stanie samodzielnie wybrać typ wartości zwracanych przez te funkcje na podstawie
    * typu danych, na których operują.
    */
}
```

```

// Utworzenie obiektu prostokat2:
Prostokat<double> prostokat2 {1.0,2.0};
/* UWAGA
 * Została utworzona instancja szablonu klasy Prostokat skonkretyzowana dla typu rzeczywistego double.
 */
cout << "Pole wynosi " << prostokat2.pole() << endl;
cout << "Obwód wynosi " << prostokat2.obwod() << endl;
/* UWAGA
 * Funkcje członkowskie pole() i obwod() zwracają na zewnątrz wartości typu double.
 */
return 0;
}

```

W programie zdefiniowano szablon klasy `Prostokat` sparametryzowany za pomocą jednego parametru szablonowego. Parametr ten jest uogólnionym typem danych τ .

Zmienne członkowskie `bok1` i `bok2` to również szablony określone za pomocą parametru szablonowego — uogólnionego typu danych τ . To samo dotyczy funkcji członkowskich (metod), `pole()` i `obwod()`, które też same w sobie są szablonami.

W programie głównym utworzono dwie instancje szablonu klasy `Prostokat`. Pierwszy z nich został skonkretyzowany dla typu całkowitego `int`, a drugi — dla typu rzeczywistego `float`.

Ćwiczenie 18.3

Zmodyfikuj program zawarty w przykładzie 18.3 — zamiast szablonu klasy `Prostokat` zdefiniuj szablon klasy `Kwadrat` zawierający definicje szablonu zmiennej członkowskiej `bok` oraz szablonów metod `pole()` i `obwod()`. Utwórz instancje szablonu klasy `Kwadrat` w programie głównym skonkretyzowane dla typów danych `long` i `float`. Przyjmij, że zadana długość boku kwadratu wynosi 1.

Przykład 18.4

```

#include <iostream>
using namespace std;

// Definicja szablonu klasy Oceny:
template <class T, int n>
/* UWAGA
 * Szablon klasy Oceny ma dwa parametry szablonowe: uogólniony typ danych T oraz liczbę całkowitą n.
 */
class Oceny {
public:
    T oceny[n]; // zmienna członkowska
    T max() { //funkcja członkowska

```

```

T temp = oceny[0];
for (int i = 0; i < n; i++ ) {
    if (oceny[i] > temp) temp = oceny[i];
}
return temp;
}

T min() { //funkcja członkowska
    T temp = oceny[0];
    for (int i = 0; i < n; i++ ) {
        if (oceny[i] < temp) temp = oceny[i];
    }
    return temp;
}

};

int main() {
    // Utworzenie i inicjalizacja obiektu oceny:
    Oceny<int, 3> oceny {3, 5, 4};
    /* UWAGA
     * Szablon klasy Oceny został skonkretyzowany za pomocą argumentów szablonowych: typu danych int
     * oraz liczby 3. Tym samym została wygenerowana instancja szablonu klasy Oceny — nowa klasa, w której
     * uogólniony typ danych T został zastąpiony typem całkowitym int, a parametr n — liczbą 3. Obiekt oceny
     * jest instancją tej właśnie klasy.
     */
    cout << "Najwyższa z ocen: " << oceny.max() << endl;
    cout << "Najniższa z ocen: " << oceny.min() << endl;

    return 0;
}

```

W programie zdefiniowano szablon klasy `Oceny`. Szablon ten ma dwa parametry szablonowe, określone w deklaracji: `template <class T, int n>`. Pierwszy parametr jest uogólnionym typem danych `T`. Rolę drugiego parametru szablonowego odgrywa liczba `n` należąca do typu całkowitego `int`.

W szablonie klasy `Oceny` zdefiniowano zmienną członkowską o nazwie `oceny`, która jest tablicą o długości określonej za pomocą parametru szablonowego `n`. Ponadto zdefiniowano tam dwie funkcje członkowskie: `max()` i `min()`. Zadaniem tych funkcji jest wyznaczenie, odpowiednio, największej i najmniejszej wartości zapisanej w tablicy `oceny`.

W programie głównym szablon klasy `Oceny` został skonkretyzowany za pomocą dwóch argumentów odpowiadających parametrom szablonowym w jego definicji: `Oceny<int, 3>`. Pierwszy argument jest typem danych `int` reprezentującym parametr szablonowy `T`. Drugi argument nie jest typem danych. Jest nim liczba całkowita 3 odpowiadająca parametrowi `n` w definicji szablonu.

Innymi słowy, określenie argumentów szablonowych `<int, 3>` powoduje wygenerowanie instancji szablonu `Oceny`, czyli utworzenie nowej klasy, w której uogólniony typ danych `T` został zastąpiony typem całkowitym `int`, a parametr `n` — liczbą 3.

Obiekt `Oceny` został utworzony jako instancja tej nowo utworzonej klasy i zainicjowany wartościami `{3, 5, 4}`: `Oceny<int, 3> oceny {3, 5, 4};`.

Ćwiczenie 18.4

Zmodyfikuj program zawarty w przykładzie 18.4 — uzupełnij szablon klasy `Oceny` o definicję (szablonu) funkcji członkowskiej pozwalającej wyznaczyć średnią arytmetyczną elementów zapisanych w (szablonie) zmiennej członkowskiej `oceny`. Wykorzystaj tę funkcję w programie głównym. Szablon klasy `Oceny` skonkretyzuj tam dla typu danych `float` oraz `n` równego 4.

18.2.2. Specjalizowanie szablonów klas

Szablony klas są parametryzowane za pomocą parametrów szablonowych. Tym samym dany szablon klasy można konkretyzować dla jednego lub większej liczby typów danych — w zależności od liczby parametrów szablonowych. Implementacje instancji szablonów klas skonkretyzowane dla różnych typów danych są takie same — różnią się jedynie typami danych.

Jednakże w praktyce zdarza się sytuacja, w której implementacja pewnej (wyróżnionej) wersji szablonu klasy dla konkretnego, zadanego typu danych (typów danych) powinna być odmienna od implementacji instancji szablonu klasy skonkretyzowanych dla innych typów danych. W takim przypadku pomocna jest tzw. specjalizacja szablonu klasy dla tego wyróżnionego typu danych.

Specjalizacja szablonu klasy (ang. *class template specialization*) stanowi wersję specjalną szablonu klasy „wyspecjalizowaną” dla określonego typu lub typów danych. Ten drugi przypadek dotyczy sytuacji, w której szablon klasy ma wiele parametrów szablonowych. Specjalizacje klas nie mają żadnych parametrów szablonowych (ang. *template parameters*).

Specjalizacje szablonów klas są traktowane przez kompilator jako klasy zupełnie niezależne od szablonów, z którymi są skojarzone. Tym samym implementacja klasy specjalizowanej może być, o czym była mowa już wcześniej, całkowicie odmienna od implementacji (a więc również funkcjonalności) szablonu.

Jeśli w procesie komplikacji programu kompilator napotyka wyrażenie, którego zadaniem jest utworzenie obiektu na podstawie szablonu klasy, w pierwszej kolejności poszukuje w kodzie źródłowym definicji wersji specjalizowanej tego szablonu. Wspomniana specjalizacja szablonu dotyczy typu argumentu, który odpowiada parametrowi szablonowemu. W przypadku gdy specjalizacja szablonu klasy dla zadanego typu danych zostaje znaleziona, to właśnie ona jest wykorzystywana do utworzenia obiektu. W przeciwnym razie, tj. jeśli wersja specjalizowana szablonu klasy dla zadanego typu danych nie zostaje znaleziona, kompilator tworzy instancję szablonu klasy skonkretyzowaną dla tego typu danych.

Przykład 18.5

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

// Definicja szablonu klasy (klasy uogólnionej) Oceny:
template <class T> // szablon ma jeden parametr szablonowy T
class Oceny {
public:
    // Definicja konstruktora domyślnego:
    Oceny() {
        cout << "Wywołanie konstruktora klasy ogólnej ..." << endl;
    }
    // Definicje szablonów zmiennych członkowskich:
    T wyklad;
    T cwiczenia;
    T laboratorium;
    T seminarium;
    // Definicja szablonu funkcji członkowskiej:
    T srednia () {
        return (wyklad + cwiczenia + laboratorium + seminarium) / 4;
        /* UWAGA
         * Jeżeli zmienne członkowskie: wyklad, cwiczenia, laboratorium i seminarium należą do typu całkowitego,
         * mamy do czynienia z dzieleniem całkowitym.
         */
    }
};

// Definicja specjalizacji szablonu klasy Oceny dla typu double:
template <> class
Oceny <double> {
public:
    // Definicja konstruktora domyślnego:
    Oceny() {
        cout << "Wywołanie konstruktora klasy specjalizowanej ..." << endl;
    }
    // Deklaracje zmiennych członkowskich:
    double wyklad;
    double cwiczenia;
    double laboratorium;
    double seminarium;
    // Definicja funkcji członkowskiej:
    double srednia () {
        return (round(wyklad) +
               round(cwiczenia) +
```

```

        round(laboratorium) +
        round(seminarium)) / 4;
    }

    /* UWAGA
     * Kod (implementacja) metody srednia() w klasie specjalizowanej różni się od kodu szablonu metody o tej samej
     * nazwie zdefiniowanej w klasie uogólnionej.
    */

};

int main() {
    // Utworzenie obiektu oceny1:
    Oceny<int> oceny1 = Oceny<int>(); // jawne wywołanie konstruktora domyślnego
    /* UWAGA
     * Obiekt oceny1 został utworzony na podstawie klasy ogólnej Oceny po jej konkretyzacji dla typu danych int.
     * Wcześniej kompilator sprawdził, że nie jest dostępna specjalizacja szablonu Oceny dla typu int.
    */
    oceny1.wyklad = 3;
    oceny1.cwiczenia = 4;
    oceny1.laboratorium = 3;
    oceny1.seminarium = 4;
    cout << "Średnia ocen semestralnych: " << oceny1.srednia() << endl;

    // Utworzenie obiektu oceny2:
    Oceny<double> oceny2; // niejawne wywołanie konstruktora domyślnego
    /* UWAGA
     * Obiekt oceny2 zostaje utworzony na podstawie specjalizacji szablonu klasy Oceny dla typu danych double.
    */
    oceny2.wyklad = 3.0;
    oceny2.cwiczenia = 4.0;
    oceny2.laboratorium = 3.0;
    oceny2.seminarium = 4.0;
    cout << "Średnia ocen semestralnych: " << oceny2.srednia() << endl;

    return 0;
}

```

W programie zdefiniowano szablon klasy `Oceny`. Szablon ten zawiera definicję konstruktora domyślnego oraz definicje innych elementów członkowskich — zmiennych członkowskich: `wyklad`, `cwiczenia`, `laboratorium` i `seminarium` oraz funkcji członkowskiej `srednia()`. Oprócz tego w programie zdefiniowano specjalizację szablonu `Oceny` dla typu rzeczywistego `double`.

Należy zwrócić uwagę na to, że kod źródłowy — implementacja metody `srednia()` zdefiniowanej w klasie specjalizowanej — różni się od metody o tej samej nazwie, która została zdefiniowana w klasie uogólnionej.

W programie głównym utworzono dwa obiekty: `oceny1` i `oceny2`. Obiekt `oceny1` został utworzony na podstawie klasy uogólnionej po jej skonkretyzowaniu dla typu całkowitego `int`. Powody tego są dwa. Po pierwsze, typ `int` został przekazany jako argument konstruktora (domyślnego) odpowiadający parametrowi szablonowemu `T`: `Oceny<int> oceny1 = Oceny<int>();`. Po drugie, kompilator nie znalazł w kodzie definicji specjalizacji szablonu klasy `Oceny` dla typu `int`.

Obiekt `oceny2` z kolei został utworzony na podstawie wersji specjalizowanej szablonu klasy `Oceny` dla typu `double`. Wynika to z faktu, że kompilator znalazł w kodzie źródłowym programu definicję specjalizacji szablonu klasy `Oceny` dla typu `double`.

Ćwiczenie 18.5

Zmodyfikuj program zawarty w przykładzie 18.5 — zdefiniuj specjalizacje klasy uogólnionej `Oceny` dla typów danych `long` i `float`. Wykorzystaj wymienione specjalizacje w programie głównym.



18.3. Szablony a polimorfizm

Stosowanie szablonów funkcji oraz szablonów klas to kolejna możliwość implementacji mechanizmu polimorfizmu (wielopostaciowości) w języku C++. W szczególności wykorzystanie szablonów funkcji pozwala definiować funkcje o tych samych nazwach i tej samej implementacji, ale operujących na różnych typach danych. To samo dotyczy szablonów klas umożliwiających tworzenie rodziny klas o analogicznej implementacji, ale różniących się od siebie typem lub typami danych, do których należą zmienne członkowskie i na których operują funkcje członkowskie klas — członków tej rodziny klas.

Generowanie przez kompilator instancji szablonów funkcji skonkretyzowanych dla określonych typów danych jest podobne do mechanizmu przeciążania funkcji (ang. *function overloading*), które jest jednym z najważniejszych narzędzi polimorfizmu. Różnica pomiędzy instancjami szablonów funkcji a funkcjami przeciążonymi polega na tym, że instancje szablonów funkcji mają identyczną implementację — ten sam kod, a implementacja funkcji przeciążonych może być różna. Przy tym uwzględnienie dodatkowo specjalizacji szablonów funkcji jeszcze bardziej zbliża instancje szablonów do funkcji przeciążonych.

UWAGA

Szablony (funkcji i klas) są podstawą tzw. **programowania uogólnionego** (ang. *generic programming*), operującego na **danych uogólnionych** (ang. *generic data*).



18.4. Pytania i zadania kontrolne

18.4.1. Pytania

1. W jakim celu stosuje się szablony funkcji (ang. *function templates*)? Jaka jest różnica pomiędzy przeciążaniem funkcji (ang. *function overloading*) a wykorzystaniem szablonów funkcji?
2. Wyjaśnij pojęcia: funkcja szablonowa (ang. *template function*), parametr szablonowy (ang. *template parameter*), ogólny typ danych (ang. *generic data type*).
3. W jakim celu stosuje się szablony klas (ang. *class templates*)?
4. Co to jest instancja szablonu (ang. *template instance*)?
5. Na czym polega specjalizacja szablonów (ang. *template specialization*)?
6. Czy wykorzystanie szablonów (funkcji i klas) ma związek z polimorfizmem?

18.4.2. Zadania

1. Napisz program pozwalający obliczać pola powierzchni i obwody figur płaskich: kwadratu, prostokąta i koła. Dane wejściowe — parametry wymienionych figur — mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Uwzględnij założenie, że parametry figur mogą być zarówno liczbami całkowitymi, jak i rzeczywistymi. Wykorzystaj zdefiniowane samodzielnie szablony funkcji. Obsługę każdego z zastosowanych typów danych całkowitych i rzeczywistych zrealizuj za pomocą instancji szablonów funkcji skonkretyzowanych dla tych typów.
2. Zrób tak jak w zadaniu 1., lecz dodatkowo zdefiniuj specjalizacje szablonów funkcji dla parametrów figur będących liczbami rzeczywistymi.
3. Zrób tak jak w zadaniu 1., lecz zamiast szablonów funkcji wykorzystaj zdefiniowane samodzielnie szablony klas. Uwzględnij wersje specjalizowane tych szablonów dla danych należących do typów rzeczywistych.
4. Napisz program pozwalający wyznaczyć średnią ocen semestralnych uzyskanych z następujących przedmiotów: języka polskiego, języka angielskiego, matematyki, informatyki. Uwzględnij średnią arytmetyczną oraz średnią ważoną z wagami: język polski — 0.1, język angielski — 0.2, matematyka — 0.4, informatyka — 0.3. Przyjmij założenie, że oceny semestralne mogą być liczbami całkowitymi o wartościach: 1, 2, 3, 4, 5, 6 oraz liczbami rzeczywistymi o wartościach: 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6. Dane wejściowe, czyli oceny semestralne uzyskane z wymienionych przedmiotów, mają być wprowadzane z klawiatury. Wyniki, tj. średnia arytmetyczna i średnia ważona, powinny być wyświetlane na ekranie monitora. Wykorzystaj zdefiniowane samodzielnie szablony funkcji. Obsługę każdego z użytych w programie typów danych zrealizuj za pomocą instancji szablonów funkcji skonkretyzowanych dla tych typów.
5. Zrób tak jak w zadaniu 4., lecz dodatkowo zdefiniuj specjalizacje szablonów funkcji dla danych wejściowych będących liczbami rzeczywistymi.

- 6.** Zrób tak jak w zadaniu 4., lecz zamiast szablonów funkcji wykorzystaj zdefiniowane samodzielnie szablon klas. Uwzględnij wersje specjalizowane tych szablonów dla danych należących do typów rzeczywistych.

19

Obsługa błędów i wyjątków

W programowaniu często pojawiają się nietypowe okoliczności prowadzące do wystąpienia błędów, np. błędów w czasie wykonywania programu (ang. *runtime errors*). Te błędy trzeba przechwycić („złapać”) i obsłużyć.

W praktyce stosowane są dwa sposoby pozwalające na przechwycenie i obsługę błędów w programie:

- system komunikatów kontrolnych i kodów zwrotnych,
- mechanizm obsługi wyjątków.

System komunikatów i kodów zwrotnych został odziedziczony po języku C. Opiera się on na wykorzystaniu charakterystycznych cech programowania strukturalnego i funkcyjnego/proceduralnego — np. sterowania przepływem w programie, funkcji zdefiniowanych samodzielnie przez programistę — w połączeniu z efektywnym i prostym interfejsem zapewniającym skutecną reakcję programu na ewentualne błędy.

UWAGA

Język C zawiera zestaw predefiniowanych funkcji i makr wspomagających obsługę błędów, np. `assert()`, `signal()` czy `raise()`. Jednakże we współczesnych programach są one bardzo rzadko stosowane.

Drugi z wymienionych powyżej sposobów polega na wykorzystaniu tzw. **wyjątków** (ang. *exceptions*) zintegrowanych z konstrukcją językową `try-catch`.

UWAGA

Konstrukcja językowa `try-catch` występuje również w innych popularnych językach programowania, np. Java i C#.



19.1. System komunikatów i kodów zwrotnych

Inspekcję kodu źródłowego programu mającą na celu przechwycenie (wygenerowanie) błędów — inaczej: **złapanie błędów** (ang. *catching errors*) — a następnie **obsługę błędów** (ang. *handling errors*) można zrealizować przy wykorzystaniu reguł sterowania przepływem (ang. *flow control rules*) w programie. Stosowanie wspomnianych reguł jest charakterystyczne dla programów strukturalnych.



UWAGA

Zasady programowania strukturalnego zostały omówione szczegółowo w podrozdziale 11.1.

Dla przypomnienia — w celu efektywnego sterowania przepływem w programie wykorzystuje się struktury sterujące (ang. *control structures*), a w szczególności struktury wyboru (ang. *selection structures*), np. instrukcje warunkowe *if-else*. Zastosowanie instrukcji warunkowych umożliwia organizację i implementację kontrolowanych rozgałęzień (ang. *controlled branching*), co w kolejnym kroku pozwala przechwycić (wygenerować) ewentualne błędy, a następnie przeprowadzić ich kompleksową obsługę. Implementacja tych rozgałęzień wymaga zazwyczaj użycia zagnieźdzonych instrukcji warunkowych (ang. *nested conditional statements*). Zamiast zagnieźdzonych instrukcji *if-else* można oczywiście stosować zagnieźdzone instrukcje *if*.



UWAGA

Zagnieźdzone instrukcje warunkowe zostały przedstawione w podrozdziale 3.2.

Jak zasygnalizowano już wcześniej, wykorzystanie systemu komunikatów i/lub kodów zwrotnych wymaga użycia odpowiednio zaprojektowanego sterowania przepływem w programie. Dotyczy to w szczególności implementacji kontrolowanych rozgałęzień, do czego używa się instrukcji warunkowych *if* lub *if-else*. Warunki we wspomnianych instrukcjach warunkowych można formułować na podstawie funkcji kontrolnych, definiowanych samodzielnie przez programistę. Informacje zwrotne dla użytkownika w postaci komunikatów i/lub kodów zaś mogą być implementowane albo w sposób bezpośredni, czyli bezpośrednio w programie głównym, albo z zastosowaniem dodatkowych, specjalizowanych funkcji pomocniczych. Jest to zależne od tego, czy kod źródłowy odpowiedzialny za przechwytywanie i obsługę błędów jest wykorzystywany jednokrotnie, czy też jest to kod wielokrotnego użytku (ang. *reusable code*).

Definiowanie i wywoływanie specjalizowanych — pomocniczych — funkcji kontrolnych jest realizowane zgodnie z zasadami programowania proceduralnego.



UWAGA

Zasady programowania proceduralnego zostały omówione szczegółowo w podrozdziale 11.1.

Określony komunikat kontrolny pozwala na przekazanie informacji tekstowej o konkretnym rodzaju błędu — jest słownym opisem błędu. System kodów zwrotnych zaś umożliwia uzyskanie informacji o błędach w postaci kodów — liczb całkowitych odpowiadających różnym rodzajom błędów.

Przykład 19.1

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

// Definicje pomocniczych funkcji kontrolnych:
bool wejscieFormat(string sDana) {
    for (int i = 0; i < sDana.length(); i++) {
        if (isdigit(sDana[i]) == false) {
            return false;
        }
    }
    return true;
}
/* UWAGA
* Zadaniem funkcji wejscieFormat() jest sprawdzenie, czy format parametru sDana należącego do typu łańcuchowego
* odpowiada liczbie całkowitej bez znaku. Jeśli tak — funkcja wejscieFormat() zwraca na zewnątrz wartość true,
* w przeciwnym razie — false.
*/
bool wejscieZakres(int iDana) {
    if ((iDana >= 1) && (iDana <= 6)) return true;
    else return false;
}
/* UWAGA
* Zadaniem funkcji wejscieZakres() jest sprawdzenie, czy wartość parametru iDana będącego liczbą całkowitą
* odpowiada ocenie szkolnej, tj. czy mieści się w zakresie domkniętym <1, 6>.
* Jeśli tak — funkcja wejscieZakres() zwraca na zewnątrz wartość true, w przeciwnym razie — false.
*/

int main() {
    // Deklaracja i inicjalizacja danych wejściowych sOcena reprezentującej ocenę semestralną ucznia:
    string sOcena = "10"; // Wartość danych jest celowo niepoprawną oceną.
```

```
if (wejscieFormat(sOcena)) { // Sprawdzenie, czy format łańcucha sOcena odpowiada liczbie
    // całkowitej bez znaku.

    // Konwersja łańcucha sOcena na liczbę całkowitą ocenę:
    int ocena = stoi(sOcena);
    /* UWAGA
     * Funkcja stoi() została wprowadzona w standardzie C++11.
     * Jej użycie wymaga dołączenia do programu biblioteki cmath.
     */

    if (wejscieZakres(ocena)) { // Sprawdzenie, czy wartość zmiennej ocena należy do zakresu <1, 6>.
        // Przetwarzanie danych wejściowych:
        switch (ocena) {
            case 1:
                cout << "Uczeń nie otrzymuje promocji do następnej klasy ..."
                    << endl;
                break;
            case 2:
            case 3:
            case 4:
            case 5:
            case 6:
                cout << "Uczeń otrzymuje promocję do następnej klasy ..."
                    << endl;
                break;
        }
    }
    else { // Jeśli wartość danych wejściowych nie mieści się w zakresie <1, 6>.
        // Wyświetlenie na ekranie monitora komunikatu kontrolnego o błędzie:
        cout << "Uwaga błąd!" << endl;
        cout << "Wartość danej wejściowej nie mieści się w zakresie <1, 6>!"
            << endl;
    }
}
else { // Jeśli format danych wejściowych nie odpowiada liczbie całkowitej bez znaku.
    // Wyświetlenie komunikatu kontrolnego o błędzie:
    cout << "Uwaga błąd!" << endl;
    cout << "Format danej wejściowej nie odpowiada liczbie całkowitej!"
        << endl;
}

return 0;
}
```

W programie zaprezentowano obsługę błędów za pomocą systemu komunikatów kontrolnych. Przechwycenie i obsługa ewentualnych błędów jest realizowana z zastosowaniem odpowiednio zorganizowanego sterowania przepływem w programie, które prowadzi do uzyskania kontrolowanych rozgałęzień. Są one zaimplementowane przy użyciu zagnieżdżonych instrukcji warunkowych `if-else`.

Integralnymi składnikami wyrażeń warunkowych w wykorzystanych tutaj instrukcjach `if-else` są pomocnicze funkcje kontrolne `wejscieFormat()` i `wejscieZakres()`, które zostały zdefiniowane samodzielnie przez programistę. Co trzeba podkreślić, obie z wymienionych funkcji zwracają na zewnątrz wartości logiczne należące do typu `bool`: albo `true`, albo `false` — co wymaga podkreślenia.

Pierwsza z nich, `wejscieFormat()`, zwraca wartość logiczną `true`, jeśli format danych `s0cena` (odgrywającej rolę argumentu wywołania funkcji) odpowiada liczbie całkowitej bez znaku, lub wartość `false`, jeśli format danych `s0cena` nie jest prawidłowy.

Druga z wymienionych funkcji, `wejscieZakres()`, zwraca do swojego otoczenia (czyli programu głównego) wartość `true`, jeśli wartość zmiennej `ocena` (jako argumentu jej wywołania) mieści się w zakresie domkniętym `<1, 6>`, lub wartość `false`, jeśli się nie mieści.

Sterowanie przepływem w programie — zrealizowane za pomocą zagnieżdżonych instrukcji warunkowych `if-else` („drabinki” `if-else`) — zostało zaprojektowane w taki sposób, że skutkiem wystąpienia określonego błędu jest wyświetlenie na ekranie monitora odpowiedniego komunikatu kontrolnego, zawierającego szczegółowe informacje o zaistniałym błędzie. Po wyświetleniu komunikatu kontrolnego program kończy działanie.

Kod źródłowy pozwalający na rejestrowanie i obsługę błędów został użyty w programie jednokrotnie. W kontekście założonej funkcjonalności programu nie jest to zatem kod wielokrotnego użytku.

Podsumowując, w przedstawionym programie kod źródłowy wykorzystywany do przechwytywania i obsługi ewentualnych błędów za pomocą systemu komunikatów kontrolnych jest ściśle powiązany — a wręcz wymieszany — z kodem odpowiedzialnym za sterowanie przepływem w programie za pomocą instrukcji warunkowych. To duża wada tego programu, ponieważ znacznie ogranicza zarówno sposób rozmieszczenia kodu źródłowego, jak i możliwą implementację mechanizmu obsługi błędów. Tym samym również funkcjonalność tego mechanizmu jest ograniczona.

Ćwiczenie 19.1

Zmodyfikuj program zawarty w przykładzie 19.1 — zamiast pojedynczej oceny semestralnej ucznia przeanalizuj jego oceny semestralne z czterech wybranych przedmiotów, np. języka polskiego, języka obcego, matematyki i informatyki. Załącz, że kod źródłowy odpowiedzialny za obsługę błędów — wyświetlenie na ekranie zwrotnych komunikatów kontrolnych — jest kodem wielokrotnego użytku.

Przykład 19.2

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

// Definicje pomocniczych funkcji globalnych:

int wejscieFormat(string sDana, string& info) {
    string sTemp = "Format danej wejściowej nie odpowiada liczbie całkowitej!";
    for (int i = 0; i < sDana.length(); i++) {
        if (isdigit(sDana[i]) == false) {
            info = sTemp;
            return -1;
        }
    }
    info = "";
    return 0;
}

/* UWAGA
* Zadaniem funkcji wejscieFormat() jest sprawdzenie, czy format parametru sDana należącego do typu lańcuchowego
* odpowiada liczbie całkowitej bez znaku. Jeśli tak — zwraca na zewnątrz kod zwrotny 0, w przeciwnym razie zwraca
* kod -1. Oprócz tego funkcja przekazuje do swojego otoczenia informację — treść komunikatu o ewentualnym błędzie.
* Wspomniany komunikat jest reprezentowany przez parametr lańcuchowy info przekazywany przez referencję.
*/

int wejscieZakres(int iDana, string& info) {
    if ((iDana >= 1) && (iDana <= 6)) {
        info = "";
        return 0;
    }
    else {
        info = "Wartość danej wejściowej nie mieści się w zakresie <1, 6> ...";
        return -1;
    }
}

/* UWAGA
* Zadaniem funkcji wejscieZakres() jest sprawdzenie, czy wartość parametru iDana (będącego liczbą całkowitą)
* odpowiada ocenie szkolnej, tj. czy mieści się w zakresie domkniętym <1, 6>.
* Jeśli tak — funkcja zwraca na zewnątrz kod zwrotny 0, w przeciwnym razie — kod równy -1.
* Oprócz tego funkcja przekazuje na zewnątrz za pośrednictwem parametru info informację o ewentualnym błędzie.
*/
```

```

int main() {
    // Definicje zmiennych pomocniczych:
    string komunikat {}; // opis błędu (treść komunikatu o błędzie)
    int kod {}; // kod błędu

    // Deklaracja i inicjalizacja danych wejściowych sOcena reprezentującej ocenę semestralną ucznia:
    string sOcena = "10"; // Wartość danych celowo nie odpowiada ocenie szkolnej.

    // Wywołanie funkcji pomocniczej wejścieFormat():
    kod = wejścieFormat(sOcena, komunikat);
    if (kod == 0) { // Sprawdzenie, czy format łańcucha sOcena odpowiada liczbie całkowitej bez znaku:
        // Konwersja łańcucha sOcena na liczbę całkowitą ocena:
        int ocena = stoi(sOcena); // Funkcja stoi() została wprowadzona w C++11.
        // Wywołanie funkcji pomocniczej wejścieZakres():
        kod = wejścieZakres(ocena, komunikat);
        if (kod == 0) { // Sprawdzenie, czy wartość zmiennej ocena należy do zakresu <1, 6>:
            // Przetwarzanie danych wejściowych:
            switch (ocena) {
                case 1:
                    cout << "Uczeń nie otrzymuje promocji do następnej klasy ..."
                        << endl;
                    break;
                case 2:
                case 3:
                case 4:
                case 5:
                case 6:
                    cout << "Uczeń otrzymuje promocję do następnej klasy ..."
                        << endl;
                    break;
            }
            return 0;
        }
    }
    // Wyświetlenie informacji o błędzie (kodu zwrotnego i komunikatu kontrolnego):
    cout << "Uwaga błąd!" << endl;
    cout << "Kod: " << kod << endl;
    cout << "Opis: " << komunikat << endl;
    /* UWAGA
     * Instrukcje powyżej są odpowiedzialne za wyświetlenie informacji o błędzie, tj. kodu zwrotnego
     * i komunikatu kontrolnego.
     */
    return kod;
}

```

Funkcjonalność przedstawionego programu została rozszerzona względem funkcjonalności programu zawartego w przykładzie 19.1. Wspomniane rozszerzenie wynika z zastosowanych tutaj narzędzi obsługi błędów. Mianowicie w programie w przykładzie 19.1 wykorzystano wyłącznie system komunikatów dla użytkownika, natomiast tutaj — system komunikatów kontrolnych uzupełniony (rozszerzony) o zwrotne kody błędów.

Rejestrowanie i obsługa ewentualnych błędów są tu implementowane w analogiczny sposób jak w programie z przykładu 19.1 — tj. z zastosowaniem kontrolowanych rozgałęzień w programie. Jednakże w tym przykładzie sterowanie przepływem zrealizowano za pomocą „drabinki” instrukcji warunkowych `if` (czyli zagnieżdżonych instrukcji `if`), a nie zagnieżdżonych instrukcji `if-else`. Użycie drabinki instrukcji `if` zamiast zagnieżdżonych instrukcji `if-else` (jak w przykładzie 19.1) pozwala uprościć implementację interfejsu mechanizmu obsługi błędów. Mianowicie tutaj zestaw instrukcji odpowiedzialnych za komunikację z użytkownikiem w razie stwierdzenia błędu występuje w kodzie źródłowym jednokrotnie — niezależnie od rodzaju błędu. Natomiast w programie z przykładu 19.1 każdy rodzaj błędu reprezentują oddzielne, niezależne instrukcje stanowiące komponenty interfejsu programu związanego z obsługą błędów.

Funkcje globalne zdefiniowane samodzielnie przez programistę: `wejscieFormat()` i `wejscieZakres()`, zwracają do swojego otoczenia kody ewentualnych błędów, a nie wartości logiczne (`true`, `false`), jak w programie z przykładu 19.1. W szczególności każda z wymienionych funkcji zwraca na zewnątrz wartość całkowitą `0`, jeżeli wystąpienia określonego błędu nie złapano, lub wartość `-1`, jeśli błąd został przechwycony.

Ponadto funkcje `wejscieFormat()` i `wejscieZakres()` przekazują do swojego otoczenia opisy słowne ewentualnych błędów w postaci zwrotnych komunikatów kontrolnych. Wspomniane komunikaty są reprezentowane przez parametry funkcji o nazwie `info` należące do typu łańcuchowego `string`. Są one przekazywane do otoczenia omawianych funkcji przez referencję.

Informacje o ewentualnych błędach w programie są przechowywane w zmiennych pomocniczych komunikat i kod. W szczególności w zmiennej komunikat zapamiętywane są komunikaty o ewentualnych błędach, a w zmiennej kod — ich kody zwrotne. Wartości wymienionych zmiennych są „ustawiane” w programie poprzez wywołania funkcji pomocniczych `wejscieFormat()` i `wejscieZakres()`, które zostały omówione powyżej. Mianowicie do zmiennej kod podstawiane są kolejno wartości zwracane przez te funkcje podczas ich wywoływanego, a zmieniona komunikat jest jednym z argumentów tych wywołań.

Zmienna kod odgrywa rolę zmiennej sterującej instrukcji warunkowych `if`, odpowiedzialnych za sterowanie przepływem w programie.

Należy zwrócić uwagę na fakt, że funkcje `wejscieFormat()` i `wejscieZakres()` są wywoływanie w programie jednokrotnie. W razie przechwycenia określonego błędu jest on następnie obsługiwany. Obsługa błędu polega na wyświetleniu na ekranie monitora odpowiedniego komunikatu o błędzie oraz jego kodu zwrotnego. Następnie program kończy działanie — to jego wada.

Można by zmodyfikować opisywany program, a konkretnie: sterowanie jego przepływem, przez użycie — oprócz drabinki instrukcji `if` — dwóch pętli programowych `do-while`. Zadaniem wspomnianych pętli byłoby zapewnienie możliwości ponownego wprowadzenia nieprawidłowych danych wejściowych w razie zarejestrowania określonego błędu. Warunek zakończenia działania każdej z użytych pętli mógłby być zdefiniowany na podstawie kodu błędu (czyli wartości zmiennej `cod`) oraz ewentualnie maksymalnej liczby możliwych iteracji (np. 3).

Reasumując, identycznie jak w poprzednim przykładzie kod źródłowy wykorzystywany do obsługi błędów jest nierozerwalnie powiązany z mechanizmem ich przechwytywania (rejestrowania), implementowanym za pomocą instrukcji warunkowych. To duża wada tego programu, ponieważ ogranicza rozmieszczenie jego kodu źródłowego, jak również implementację mechanizmu przechwytywania i obsługi błędów.

Ćwiczenie 19.2

Zmodyfikuj program zawarty w przykładzie 19.2 — zaprojektuj funkcje `wejscieFormat()` i `wejscieZakres()` z parametrami będącymi C-napisami (czyli łańcuchami znaków w stylu języka C), a nie należącymi do typu `string`. To samo zrób ze zmienną pomocniczą komunikat. Ponadto uzupełnij program o dwie pętle `do-while` pozwalające na wielokrotne (np. maksymalnie trzykrotne) wprowadzenie danych wejściowych w razie przechwycenia określonego błędu.

Przedstawione podejście umożliwiające rejestrowanie i obsługę błędów w czasie wykonywania programu za pomocą mechanizmu komunikatów i/lub kodów zwrotnych zintegrowanych z systemem kontrolowanych rozgałęzień jest względnie proste do zaimplementowania w przypadku niewielkiej liczby sytuacji nietypowych (wyjątkowych), które mogą doprowadzić do powstania błędów. Sytuacja znacznie się komplikuje, jeśli programista musi przewidzieć wystąpienie dużej liczby błędów, a następnie zaimplementować ich rejestrację i obsługę. Często może to być niezwykle trudne. Skutkiem ubocznym jest wówczas znaczne zmniejszenie czytelności kodu źródłowego, które może doprowadzić do braku konsekwencji w procesie przechwytywania ewentualnych błędów.

Najważniejsze wady mechanizmu rejestrowania i obsługi błędów przy wykorzystaniu systemu komunikatów i/lub kodów zwrotnych to:

- Powiązanie i wymieszanie kodu źródłowego odpowiedzialnego za rejestrowanie i obsługę błędów ze sterowaniem przepływem w programie w postaci kontrolowanych rozgałęzień zaimplementowanych za pomocą instrukcji warunkowych.
- Każda funkcja pomocnicza, której zadaniem jest sprawdzenie, czy dany błąd wystąpił, czy nie, może zwrócić do swojego otoczenia wyłącznie pojedynczą wartość — co znacznie ogranicza jej funkcjonalność.
- Jeżeli funkcja pomocnicza jest wywoływaną w konstruktorze, kodu ewentualnego błędu (i komunikatu tekstopisu związanego z tym błędem) nie można przekazać do jego otoczenia, ponieważ konstruktory nie zwracają na zewnątrz żadnych wartości.

- Jeśli programista nie przewidział sytuacji, że funkcja wywołująca daną funkcję kontrolną powinna odebrać od niej kod ewentualnego błędu, może to doprowadzić do nieprzewidzianych okoliczności i generowania następnych błędów.



19.2. Mechanizm obsługi wyjątków

Drugi sposób umożliwiający przechwytywanie oraz obsługę błędów w programie to zastosowanie mechanizmu **obsługi wyjątków** (ang. *exception handling*), będącego rozszerzeniem języka C++ względem C. Przy czym wyjątek jest rozumiany jako pewna wyjątkowa (nietypową) okoliczność — swego rodzaju anomalia, która może wystąpić podczas wykonywania programu. W języku C++ wyjątki zwykle są rozumiane i traktowane jako **błędy w czasie wykonywania programu** (ang. *runtime errors*).

Z formalnego punktu widzenia w języku C++ wyjątki to obiekty będące instancjami określonych klas. Wspomniane klasy określają typy sytuacji wyjątkowych, czyli typy (rodzaje) błędów. Wyjątki będące obiektami mogą — analogicznie do „zwykłych” obiektów — zawierać elementy członkowskie (ang. *members*) pozwalające na przekazanie programiście szczegółowych informacji o błędzie.

Ponadto wyjątki w C++ mogą również należeć do typów wbudowanych — np. typów podstawowych (ang. *fundamental types*), takich jak `int` i `double`.

Działanie mechanizmu obsługi wyjątków w języku C++ dzieli się na kilka etapów. Wspomniane etapy można w uproszczeniu opisać tak:

- Pierwszy etap polega na inspekcji kodu źródłowego określonej funkcji (np. funkcji `main()`), która ma na celu wykrycie nietypowych okoliczności (anomalii), czyli błędów — np. próby dzielenia przez zero.
- Na drugim etapie wykryty błąd powinien zostać „**zgłoszony**”, inaczej: **wyrzucony** (ang. *thrown*), jako przeznaczony do dalszego przetwarzania (obsługi).
- Zgłoszenie błędu skutkuje przerwaniem wykonywania programu i rozpoczęciem trzeciego etapu — procesu jego przechwycenia, inaczej: **złapania** (ang. *catch*). Przechwycenie błędu polega na znalezieniu w programie kodu źródłowego odpowiedzialnego za jego obsługę.
- Ostatni etap to **obsługa błędu** (ang. *error handling*). Kod źródłowy, który za zadanie ma obsługę błędu, jest nazywany **procedurą obsługi błędu** (ang. *exception handler*).

Wymieniona powyżej inspekcja kodu źródłowego mająca na celu wykrycie w programie określonego błędu (lub błędów) jest realizowana w bloku kodu oznaczonym słowem kluczowym `try`:

```
try {
...
}
```

Wystąpienie określonego wyjątku (błędu) w bloku `try` jest sygnalizowane do jego otoczenia za pomocą słowa kluczowego `throw`:

`throw wyjątek;`

gdzie `wyjątek` to obiekt należący do dowolnego typu wbudowanego (np. `int`, `string`) lub typu klasowego.

Z formalnego punktu widzenia wykonanie instrukcji `throw wyjątek;` powoduje **wyrzucenie** (ang. *throwing*) wyjątku `wyjątek`. Na przykład we fragmencie kodu:

```
throw -1;
throw dana;
throw "Uwaga! Mianownik jest równy 0!";
throw Wyjatek("Uwaga błąd!");
```

zgłaszcane są cztery różne wyjątki. W szczególności wykonanie instrukcji `throw -1;` powoduje zgłoszenie jako wyjątku wartości literała `-1`. Z kolei instrukcja `throw dana;` „wyrzuca” zmienną `dana`. Następna instrukcja, `throw "Uwaga! Mianownik jest równy 0!";`, powoduje zgłoszenie jako wyjątku literała łańcuchowego typu `const char*` o wartości `"Uwaga! Mianownik jest równy 0!"`, a `throw Wyjatek("Uwaga błąd!");` — obiekt będący instancją klasy `Wyjatek`.

Jak wspomniano wcześniej, kod źródłowy odpowiedzialny za obsługę wyjątku (błędu) jest nazywany **procedurą obsługi błędu**. Procedura obsługi błędu jest oznaczana w programie za pomocą słowa kluczowego `catch`:

```
catch (parametr) {
    ...
}
```

gdzie `parametr` odpowiada wyjątkowi (błędowi), który jest obsługiwany.

Składnia deklaracji procedury obsługi wyjątku z parametrem `parametr` jest analogiczna do składni zwykłej funkcji z pojedynczym parametrem wejściowym. Na przykład w nagłówku procedury obsługi `catch (double mianownik)` parametrem jest zmienna o nazwie `mianownik` należąca do typu podstawowego `double`. Typ parametru powinien być zgodny z typem obsługiwanej wyjątku.

Zalecane jest, aby wyjątki należące do typów podstawowych — np. `double`, `int` — były przekazywane do procedury obsługi błędu przez wartość. Wyjątki, które nie należą do typów podstawowych, np. instancje struktur (`struct`), powinny zaś być przekazywane do procedury obsługi jako referencje do stałych `const`.

Uwzględniając przedstawione powyżej uwagi, mechanizm obsługi pojedynczego wyjątku (błędu) w programie można opisać w sposób ogólny tak:

```
try {
    zestaw_instrukcji_1;
    if (warunek) throw wyjątek;
    zestaw_instrukcji_2;
}
```

```
catch (parametr) {
    obsługa_wyjątku;
}
```

gdzie:

- `try`, `throw` i `catch` to słowa kluczowe,
- `zestaw_instrukcji_1`, `zestaw_instrukcji_2` to ciągi dowolnych instrukcji,
- `if (warunek) ...` to instrukcja warunkowa pozwalająca na inspekcję kodu źródłowego na podstawie sprawdzenia warunku (w przypadku ogólnym można stosować również instrukcję `if-else`),
- `wyjątek` to wyrażenie opisujące obiekt zgłaszający błąd,
- `parametr` to parametr procedury obsługi błędu (typ tego parametru powinien być zgodny z typem wyrażenia `wyjątek`),
- `obsługa_wyjątku` to zestaw instrukcji odpowiedzialnych za obsługę błędu odpowiadającego parametrowi, tj. „ciało” procedury obsługi błędu.

Jak wspomniano wcześniej, inspekcja kodu źródłowego, która ma na celu stwierdzenie wystąpienia określonego wyjątku (błędu), jest realizowana w bloku `try`. Zazwyczaj jest prowadzona z wykorzystaniem instrukcji warunkowych `if` lub `if-else` z odpowiednim warunkiem.

Wykrycie błędu jest sygnalizowane do otoczenia bloku `try` przez jego zgłoszenie. Następuje to jako skutek wykonania instrukcji `throw wyjątek`, w której wyrażenie `wyjątek` opisuje obiekt zgłaszający błąd.

W przypadku gdy określony wyjątek (błąd) nie zostanie wykryty (i zgłoszony), wówczas wykonanie dalszego kodu w programie jest kontynuowane w standardowy sposób. Tym samym wykonane zostaną instrukcje zawarte w `zestawie_instrukcji_2`.

Zgłoszenie błędu powoduje przeniesienie sterowania przebiegiem działania programu (sterowania przepływem) do procedury obsługi wyjątku, która jest oznaczana w kodzie źródłowym za pomocą słowa kluczowego `catch` — o ile oczywiście odpowiednia procedura obsługi istnieje i zostanie znaleziona.

Wyjątek (błąd) jest traktowany jako obsłużony, jeżeli zostanie znaleziony blok `catch`, reprezentujący procedurę obsługi tego błędu. Po znalezieniu procedury obsługi błędu jest ona wykonywana.

Cechą charakterystyczną mechanizmu obsługi wyjątków w języku C++ jest to, że nie ma możliwości powrotu z kodu źródłowego obsługującego wyjątek, tj. z procedury obsługi (bloku `catch`), do kodu, z którego poziomu następuje zgłoszenie jego wystąpienia, czyli do bloku `try`.

Po zakończeniu procesu obsługi wyjątku w bloku `catch` przepływ sterowania w programie przechodzi do następnej instrukcji po konstrukcji `try-catch`.

Przykład 19.3

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

int main() {
    try { // początek bloku kodu, w którym jest realizowana inspekcja kodu źródłowego
        // Dane wejściowe:
        double licznik = 1, mianownik = 0;
        if (mianownik == 0) // próba wykrycia wyjątku — błędu
            throw mianownik; // zgłoszenie wyjątku
        /* UWAGA
         * Obiektem zgłaszającym wyjątek (błąd) jest zmienna mianownik należąca do typu double.
         */
        cout << "Licznik = " << licznik << endl;
        cout << "Mianownik = " << mianownik << endl;
        cout << "Iloraz wynosi " << licznik/mianownik << endl;
        /* UWAGA
         * Jeśli zostanie zgłoszony wyjątek, zestaw instrukcji powyżej nie zostanie wykonany.
         */
    }
    catch (double mianownik) { // nagłówek procedury obsługi wyjątku
        cout << "Błąd! Próba dzielenia przez zero!" << endl;
        /* UWAGA
         * Jeżeli w treści procedury obsługi parametr formalny nie jest wykorzystywany — czyli jak tutaj —
         * to w nagłówku procedury obsługi można go pominąć.
         * W takim przypadku nagłówek procedury obsługi błędu miałby postać: catch (double).
         */
    }
    return 0;
}
```

W programie obliczany jest iloraz dwóch liczb przechowywanych w zmiennych `licznik` i `mianownik`, które należą do typu `double`.

Inspekcja kodu źródłowego, którego wykonanie może spowodować wygenerowanie wyjątku, jest realizowana w bloku kodu oznaczonym słowem kluczowym `try`. Polega ona na sprawdzeniu za pomocą instrukcji warunkowej `if`, czy wartość zmiennej `mianownik` jest równa `0`: `if (mianownik == 0)`. Jeśli tak (`true`), zgłoszany jest wyjątek. Obiektem zgłaszającym jest zmienna `mianownik`: `throw mianownik;`.

Po zgłoszeniu wyjątku sterowanie przebiegiem działania programu (przepływ sterowania) przechodzi do bloku kodu oznaczonego słowem kluczowym `catch`, w którym wyjątek ten jest obsługiwany. Wygenerowanie (przechwycenie) wyjątku polega tutaj na przekazaniu zmiennej `mianownik` z bloku `try` (w którym zaistniał problem, czego skutkiem jest zgłoszenie

wyjątku) do bloku `catch: catch (double mianownik)`, w którym wyjątek jest obsługiwany. Po przeniesieniu sterowania do bloku `catch` już nie wraca ono do bloku `try`.

Procedurę obsługi wyjątku stanowi tutaj pojedyncza instrukcja: `cout << "Błąd! Próba dzielenia przez zero!" << endl;`.

Po zakończeniu procesu obsługi błędu w bloku `catch` przepływ sterowania przechodzi do następnej instrukcji w programie, czyli `return 0;`.

Jeżeli błąd nie zostanie wykryty (i tym samym zgłoszony), wykonywany jest zestaw instrukcji zawartych w bloku `try` bezpośrednio po instrukcji `if`. Po zakończeniu tych działań przepływ sterowania w programie przechodzi do wykonania instrukcji `return 0`, czyli blok `catch` zostaje pominięty (przeskoczony).

Ćwiczenie 19.3

Zmodyfikuj program zawarty w przykładzie 19.3 — zamiast mechanizmu obsługi wyjątków (czyli konstrukcji `try-catch`) wykorzystaj mechanizm oparty na systemie komunikatów i kodów zwrotnych, który został omówiony w poprzednim podrozdziale.

W bloku `try` można przeprowadzać inspekcję kodu źródłowego programu względem nie tylko pojedynczej sytuacji wyjątkowej — pojedynczego błędu, ale także wielu błędów. W takim przypadku każdy z tych błędów może być obsługiwany przez niezależną procedurę obsługi, czyli odrębny blok `catch`.

Z formalnego punktu widzenia wiąże się to z koniecznością użycia wielu instrukcji `throw` zawartych w bloku `try` oraz wielu bloków `catch` zdefiniowanych bezpośrednio po bloku `try`.

Ogólna postać instrukcji opisujących mechanizm obsługi wielu błędów jest następująca:

```
try {
    zestaw_instrukcji_1;
    if (warunek_1) throw wyjątek_1;
    zestaw_instrukcji_2;
    if (warunek_2) throw wyjątek_2;
    ...
    zestaw_instrukcji;

}
catch (parametr_1) {
    obsługa_wyjątku_1;
}
catch (parametr_2) {
    obsługa_wyjątku_2;
}
catch (...) {
    obsługa_pozostałych_wyjątków;
}
```

gdzie:

- `try`, `throw` i `catch` to słowa kluczowe,
- `zestaw_instrukcji_1`, `zestaw_instrukcji_2`, `zestaw_instrukcji` to ciągi dowolnych instrukcji,
- `warunek_1`, `warunek_2` to warunki pozwalające na wykrycie określonych błędów i ich zgłoszenie,
- `wyjątek_1`, `wyjątek_2` to wyrażenia opisujące obiekty zgłaszające błędy (wynik każdego z tych wyrażeń powinien należeć do innego typu),
- `parametr_1`, `parametr_2` to parametry procedur obsługi błędów (`parametr_1` reprezentuje wyrażenie `wyjątek_1`, a `parametr_2` odpowiada `wyjątkowi_2`), przy czym każdy parametr powinien należeć do innego typu, a typy parametrów powinny być zgodne z odpowiadającymi im typami wyrażeń (`wyjątek_1`, `wyjątek_2`),
- `catch (...)` to domyślna procedura obsługi przechwytyująca wszystkie wyjątki (błędy), które nie zostały przechwycone przez inne procedury obsługi, zdefiniowane powyżej. W przypadku domyślnej procedury obsługi typ błędu jest nieistotny — może ona obsługiwać dowolne typy błędów. Deklaracja domyślnej procedury obsługi błędów jest opcjonalna.

Jeżeli inspekcja kodu źródłowego zawartego w bloku `try`, realizowana przy użyciu instrukcji warunkowych `if` (lub `if-else`), spowoduje zgłoszenie określonego wyjątku — błędu: `throw wyjątek_1` lub `throw wyjątek_2`, wykonywanie dalszych instrukcji w bloku `try` zostanie wstrzymane i nastąpi poszukiwanie procedury obsługi z odpowiednim parametrem. Typ parametru procedury obsługi powinien być zgodny z typem wyrażenia opisującego obiekt zgłaszający błąd (`wyjątek_1`, `wyjątek_2`). Poszczególne bloki `catch`, zawierające procedury obsługi błędów, są przeglądane sekwencyjnie jeden po drugim w kolejności ich zadeklarowania. Jeśli odpowiednia procedura obsługi została znaleziona, to jest ona wykonywana. Po zakończeniu obsługi wyjątku sterowanie przepływu w programie jest kierowane do następnej instrukcji po łańcuchu bloków `catch`.

Jeśli natomiast procedura obsługi przeznaczona do obsługi określonego błędu nie została znaleziona, wykonywana jest domyślna procedura obsługi — o ile została zdefiniowana. W przypadku gdy nie zdefiniowano domyślnej procedury obsługi, przepływ sterowania jest przekierowywany „wyżej”, do funkcji wywołującej, i tam proces poszukiwania odpowiedniej procedury obsługi jest kontynuowany. Jeśli i tam procedura obsługi zaistniałego błędu nie zostanie znaleziona, wykonywanie programu zostanie przerwane. Taki proces nazywamy **odwijaniem stosu** (ang. *stack unwinding*).

Przykład 19.4

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

```
// Definicje globalnych funkcji pomocniczych:  
bool wejscieFormat(string sDana) {  
    for (int i = 0; i < sDana.length(); i++) {  
        if (isdigit(sDana[i]) == false) {  
            return false;  
        }  
    }  
    return true;  
}  
  
bool wejscieZakres(int iDana) {  
    if ((iDana >= 1) && (iDana <= 6)) return true;  
    else return false;  
}  
  
int main() {  
    // Deklaracja i inicjalizacja danych wejściowych sOcena reprezentującej ocenę semestralną ucznia:  
    string sOcena = "10"; // Wartość danych jest celowo niepoprawna.  
  
    try { // początek bloku, w którym jest realizowana inspekcja kodu źródłowego  
        if (wejscieFormat(sOcena) == false) { // próba wykrycia błędu  
            throw sOcena; // zgłoszenie wyjątku sOcena  
        }  
        int ocena = stoi(sOcena);  
        /* UWAGA  
         * Instrukcja powyżej zostanie wykonana tylko w przypadku, jeśli zmieniona sOcena nie zgłosi  
         * wyjątku — błędu.  
         */  
        if (wejscieZakres(ocena) == false) { // próba wykrycia błędu  
            throw ocena; // zgłoszenie wyjątku ocena  
        }  
        // Instrukcje, które zostaną wykonane, jeśli nie zostanie zgłoszony żaden błąd:  
        switch (ocena) {  
            case 1:  
                cout << "Uczeń nie otrzymuje promocji do następnej klasy ..."   
                     << endl;  
                break;  
            case 2:  
            case 3:  
            case 4:  
            case 5:  
            case 6:  
                cout << "Uczeń otrzymuje promocję do następnej klasy ..."   
                     << endl;  
                break;  
        }  
    }
```

```

// Procedura obsługi wyjątku zgłoszonego przez zmienną s0cena:
catch (string s0cena) {
    cout << "Uwaga błąd!" << endl;
    cout << "Format danej wejściowej " << s0cena
        << " nie odpowiada liczbie całkowitej!" << endl;
}

// Procedura obsługi wyjątku zgłoszonego przez zmienną ocena:
catch (int ocena) {
    cout << "Uwaga błąd!" << endl;
    cout << "Wartość danej wejściowej " << ocena
        << " nie mieści się w zakresie <1, 6>!" << endl;
}

return 0;
}

```

Funkcjonalność przedstawionego programu jest taka sama jak funkcjonalność programu z przykładu 19.2. Jednakże w przykładzie 19.2 obsługę błędów zrealizowano na podstawie struktury kontrolowanych rozgałęzień zintegrowanych z systemem komunikatów i kodów zwrotnych, a tutaj — przy użyciu mechanizmu obsługi wyjątków opartego na konstrukcji językowej `try-catch`.

Inspekcja kodu źródłowego, którego wykonanie może spowodować wystąpienie sytuacji wyjątkowych, jest ograniczona „klamrami” bloku `try`. Uwzględniono tam możliwość wystąpienia dwóch problemów — błędów. Pierwszy z nich wynika z nieprawidłowego formatu danych wejściowych reprezentujących ocenę semestralną ucznia, a drugi — z nieprawidłowej wartości tych danych, która może nie mieścić się w założonym zakresie `<1, 6>`.

Sprawdzenie danych wejściowych jest realizowane w dwóch krokach przy wykorzystaniu funkcji pomocniczych: `wejscieFormat()` i `wejscieZakres()`. Obie z wymienionych funkcji zwracają na zewnątrz wartości logiczne: `true` (brak błędu) lub `false` (w razie wystąpienia błędu). Implementacja wymienionych funkcji jest identyczna z implementacją funkcji o tych samych nazwach w przykładzie 19.1.

W przypadku nieprawidłowego formatu danych wejściowych zgłaszany jest wyjątek `s0cena`: `throw s0cena`. Jeśli zaś wartość danych wejściowych wykracza poza zakres `<1, 6>`, zgłaszany jest wyjątek `ocena`: `throw ocena`.

Oba wymienione powyżej wyjątki — błędy — są przekazywane jako parametry/argumenty do odpowiadających im procedur obsługi. Wspomniane parametry/argumenty zostały określone w nagłówkach tych procedur: `catch(string s0cena)` oraz `catch(int ocena)`.

Jeśli w trakcie wykonywania programu zostanie wykryty i zgłoszony określony wyjątek — błąd — sterowanie przepływem (ang. *control flow*) zostanie skierowane do odpowiadającej mu procedury obsługi. Wyjątek (błąd) zostaje dostarczony do wybranej procedury obsługi jako jej argument i w następnym kroku jest obsługiwany przez wykonanie odpowiadającej mu procedury obsługi.

Jeśli żaden błąd nie zostaje wykryty (i tym samym nie jest zgłoszany), wykonywany jest zestaw instrukcji zawarty w bloku `try` bezpośrednio pod instrukcjami warunkowymi `if`. Następnie przepływ sterowania w programie jest przenoszony do instrukcji `return 0` — czyli kod zawarty w obu zdefiniowanych procedurach obsługi błędów (obu blokach `catch`) jest pomijany.

Funkcje globalne `wejscieFormat()` i `wejscieZakres()`, o takiej samej funkcjonalności, jaką mają funkcje o identycznych nazwach w przykładzie 19.1, wykorzystano tutaj w celach edukacyjnych. Dzięki temu można łatwo porównać implementację mechanizmu obsługi błędów opartego na systemie komunikatów zwrotnych z implementacją mechanizmu obsługi wyjątków zrealizowaną na podstawie konstrukcji językowej `try-catch`.

Z praktycznego punktu widzenia — wyjawszy wzgłydy edukacyjne — funkcjonalność wykorzystanych tutaj funkcji `wejscieFormat()` i `wejscieZakres()` należałoby zmienić. Mianowicie funkcje te nie powinny zwracać na zewnątrz wartości logicznych `true/false`, ale „same z siebie” powinny wyrzucić ze swojego wnętrza do otoczenia właściwy wyjątek (błąd). Na przykład definicja funkcji `wejscieZakres()` mogłaby mieć następującą postać:

```
void inputRange(int iDana) {
    if (!(iDana >= 1) && (iDana <= 6)) {
        throw iDana;
    }
}
```

Funkcję `wejscieFormat()` należałoby zdefiniować analogicznie.

Ćwiczenie 19.4

Zmodyfikuj program zawarty w przykładzie 19.4 — obsługę wyjątków zrealizuj przy wykorzystaniu elementów członkowskich zdefiniowanej samodzielnie klasy `Wyjatek`. Wspomniana klasa `Wyjatek` powinna zawierać definicje odpowiednich składników — np. definicje metod `inputFormat()` i `inputInRange()` — niezbędnych do obsługi tych samych błędów co w przykładzie 19.4.

Przykład 19.5

```
#include <iostream>
#include <string>
#include <sstream>
#include <cmath>
using namespace std;

// Definicja struktury (klasy) abstrakcyjnej Wyjatek:
struct Wyjatek {
    // Deklaracja metody abstrakcyjnej message():
    virtual string komunikat() = 0;
```

```

/* UWAGA
 * Definicje metody komunikat() o różnej funkcjonalności są zawarte w klasach Format i Zakres,
 * które są klasami pochodnymi klasy Wyjatek.
 */
};

// Definicja klasy Format, która jest klasą pochodną klasy Wyjatek:
class Format : public Wyjatek {
    // Deklaracja prywatnej zmiennej członkowskiej:
    string s0cena;
public:
    // Definicja konstruktora:
    Format(string s0cena) : s0cena(s0cena) {}
    // Definicja metody komunikat() odziedziczonej po klasie abstrakcyjnej Wyjatek:
    string komunikat() {
        // Deklaracja zmiennej pomocniczej sTemp:
        stringstream sTemp;
        /* UWAGA
         * Strumień stringstream odpowiada klasie, która pozwala wykonywać różne operacje na łańcuchach
         * należących do typu string, traktowanych jak strumienie.
         * Łańcuchy można zarówno zapisywać do strumienia stringstream, jak i z niego odczytywać.
         */
        sTemp << "Uwaga błąd! Format danej wejściowej " << s0cena
            << " nie odpowiada liczbie całkowitej!" << endl;
        return sTemp.str();
        /* UWAGA
         * Metoda str(), która należy do klasy stringstream, zwraca obiekt zawierający kopię wartości
         * strumienia sTemp.
         */
    }
};

// Definicja klasy Zakres, która jest klasą pochodną klasy Wyjatek:
class Zakres : public Wyjatek {
    int ocena;
public:
    Zakres(int ocena) : ocena(ocena) {}
    // Definicja metody komunikat() odziedziczonej po klasie abstrakcyjnej Wyjatek:
    string komunikat() {
        stringstream sTemp;
        sTemp << "Uwaga błąd! Wartość danej wejściowej " << ocena
            << " nie mieści się w zakresie <1, 6>!" << endl;
        return sTemp.str();
    }
};

```

```

// Definicje funkcji pomocniczych:
bool wejscieFormat(string sDana) {
    for (int i = 0; i < sDana.length(); i++) {
        if (isdigit(sDana[i]) == false) {
            return false;
        }
    }
    return true;
}

bool wejscieZakres(int iDana) {
    if ((iDana >= 1) && (iDana <= 6)) return true;
    else return false;
}

int main() {
    try {
        // Deklaracja i inicjalizacja danych wejściowych reprezentujących ocenę semestralną:
        string s0cena = "10"; // Wartość danych wejściowych celowo nie odpowiada ocenie szkolnej.

        // Inspekcja kodu źródłowego w celu wykrycia błędu związanego z nieprawidłowym formatem
        // danych wejściowych:
        if (wejscieFormat(s0cena) == false)
            throw Format(s0cena); // zgłoszenie obiektu stanowiącego instancję klasy Format
        // Konwersjałańca na liczbę całkowitą:
        int ocena = stoi(s0cena);

        // Inspekcja kodu źródłowego w celu wykrycia błędu wynikającego z błędnej wartości danych wejściowych:
        if (wejscieZakres(ocena) == false)
            throw Zakres(ocena); // zgłoszenie obiektu należącego do klasy Zakres
        // Zestaw instrukcji, który zostanie wykonany, jeśli nie wykryto (i nie zgłoszono) żadnego błędu:
        switch (ocena) {
            case 1:
                cout << "Uczeń nie otrzymuje promocji do następnej klasy ..." 
                    << endl;
                break;
            case 2:
            case 3:
            case 4:
            case 5:
            case 6:
                cout << "Uczeń otrzymuje promocję do następnej klasy ..." 
                    << endl;
                break;
        }
    }
}

```

```

}

// Procedura obsługi błędu należącego do klasy Wyjatek:
catch(Wyjatek& wyjatek) {
    cerr << wyjatek.komunikat() << endl;
    /* UWAGA
     * Strumień cerr jest standardowym strumieniem błędów.
     */
}
}

```

Funkcjonalność przedstawionego programu jest taka sama jak funkcjonalność programu zawartego w przykładzie 19.4. Jednakże tutaj mechanizm obsługi błędów został zaimplementowany przy użyciu klas: `Wyjatek`, `Format` i `Zakres`, zdefiniowanych samodzielnie przez programistę.

Klasa `Wyjatek` jest klasą abstrakcyjną, odgrywającą rolę interfejsu — struktury programistycznej znanej z takich języków programowania jak Java czy C#. Zawiera ona deklarację metody czysto wirtualnej (czyli metody abstrakcyjnej) `komunikat()`.

Każda metoda abstrakcyjna jest metodą polimorficzną. To samo dotyczy oczywiście metody `komunikat()` z klasy `Wyjatek`. Zadeklarowanie metody `komunikat()` jako abstrakcyjnej automatycznie załącza mechanizm polimorfizmu dynamicznego, czyli wielopostaciowości wspomnianej metody, która zachodzi w czasie wykonywania programu.

Klasa `Wyjatek` jest klasą bazową dla klas `Format` i `Zakres`. Tym samym klasy `Format` i `Zakres` są klasami pochodnymi klasy `Wyjatek`. W wymienionych klasach pochodnych zawarte są definicje metody `komunikat()`, odziedziczonej po klasie bazowej `Wyjatek`. Obie zdefiniowane postacie metody `komunikat()` charakteryzują się odmienną funkcjonalnością i tym samym implementacją.

UWAGA

Klasy abstrakcyjne, interfejsy i metody abstrakcyjne zostały omówione szczegółowo w rozdziale 16. podręcznika, polimorfizm — w rozdziale 15., a mechanizm dziedziczenia — w rozdziale 14.

Inspekcja kodu źródłowego jest realizowana w bloku `try` przy użyciu instrukcji warunkowych `if`. Warunki we wspomnianych instrukcjach `if` są formułowane na podstawie wyników wywołań funkcji globalnych `wejscieFormat()` i `wejscieZakres()`, które zwracają na zewnątrz wartości logiczne `true` lub `false`.

Wykrycie błędu związanego z niewłaściwym formatem danych wejściowych lub błędem wynikającym z nieprawidłowej wartości tych danych — wartości wykraczającej poza założony zakres, skutkuje zgłoszeniem odpowiedniego błędu za pomocą instrukcji `throw`. W razie wykrycia błędu związanego z formatem danych wejściowych zgłaszany jest błąd — obiekt

należący do klasy `Format`. Jeśli zaś wykryty zostanie błąd zakresu danych, zgłaszany jest obiekt będący instancją klasy `Zakres`.

Oba rodzaje błędów są obsługiwane przez jedną procedurę obsługi, czyli kod zawarty w tym samym bloku `catch`. Wynika to z typu parametru procedury obsługi: `Wyjatek& wyjatek`, który należy do typu bazowego `Wyjatek` i jest przekazywany przez referencję. Dzięki temu procedura obsługi ma możliwość przechwytywania błędów stanowiących instancje obu klas pochodnych (tj. `Format` i `Zakres`) klasy `Wyjatek` i ponadto może wykorzystywać mechanizm polimorfizmu (wielopostaciowości) metody `komunikat()`. Tym samym kod `wyjatek.komunikat()` reprezentuje faktycznie wywołanie metody `komunikat()` zdefiniowanej w tej klasie pochodnej (`Format` lub `Zakres`), której obiekt jest argumentem metody.

Ćwiczenie 19.5

Zmodyfikuj program zawarty w przykładzie 19.5 — zmień lub usuń, ewentualnie uzupełnij kod źródłowy odpowiedzialny za funkcjonalność jego komponentów składowych, aby błędy, które są instancjami klas `Format` i `Zakres`, były obsługiwane w niezależnych blokach `catch`, czyli za pomocą różnych procedur obsługi.

UWAGA

Więcej informacji dotyczących zgłaszania, rejestrowania i obsługi wyjątków można znaleźć w dokumentacji języka C++ na stronie: cppreference.com.

W programach demonstracyjnych zawartych w przykładach przedstawionych wcześniej w tym rozdziale zaprezentowano mechanizm obsługi wyjątków (błędów), które zostały zdefiniowane samodzielnie przez programistę. Przy tym — z formalnego punktu widzenia — uwzględniono zarówno błędy zgłasiane przez zmienne należące do typów podstawowych (jak w przykładach 19.3 i 19.4), jak i błędy będące obiektami (przykład 19.5).

Oprócz błędów definiowanych samodzielnie przez programistę mechanizm obsługi wyjątków w C++ zapewnia możliwość wykorzystania klasy `exception`, która jest integralnym składnikiem standardowej biblioteki C++ (ang. *C++ Standard Library*). Klasa ta odgrywa rolę klasy specjalnej, przeznaczonej do deklarowania obiektów, które mogą reprezentować ewentualne wyjątki (błędy) w programie. Wyjątki — obiekty udostępniane przez klasę `exception` — są nazywane **wyjątkami standardowymi** (ang. *standard exceptions*). Do wyjątków standardowych zaliczane są wszystkie wyjątki generowane przez zasoby standardowej biblioteki C++.

Wyjątki standardowe można podzielić na dwa rodzaje:

- **wyjątki uogólnione** (ang. *generic exceptions*) wynikające z błędnej logiki wewnętrznej w programie — **błędy logiczne** (ang. *logic errors*),
- wyjątki uogólnione wykrywane w trakcie trwania programu — **błędy w czasie wykonywania programu** (ang. *runtime errors*).

Pierwsza z wymienionych powyżej grup wyjątków jest reprezentowana w bibliotece `exception` przez klasę semantyczną `logic_error`, a druga — przez klasę `runtime_error`. Obie te klasy są klasami pochodnymi klasy `exception`.

Standardowe wyjątki logiczne są obsługiwane przez obiekty będąceinstancjami predefiniowanych klas pochodnych klasy `logic_error`, np. `invalid_argument`, `domain_error`, `length_error` czy też `out_of_range`. Za obsługę błędów standardowych w czasie wykonywania programu zaś odpowiedzialne są obiekty należące do predefiniowanych klas pochodnych klasy `runtime_error`, np. `range_error` i `overflow_error`.

Definicja klasy `exception` jest zawarta w bibliotece standardowej o tej samej nazwie, [exception](#). Dlatego użycie omawianej klasy w programie wymaga dołączenia do niego zasobów biblioteki, czyli dyrektywy `#include <exception>`.

Do mechanizmu obsługi wyjątków w programie mogą być dołączane — oprócz wymienionych wcześniej predefiniowanych klas należących do standardowej biblioteki C++ — klasy zdefiniowane samodzielnie przez programistę. Wynika to z faktu, że klasa `exception` może również z powodzeniem odgrywać rolę klasy bazowej dla klas pochodnych zdefiniowanych samodzielnie przez programistę. Tym samym klasa `exception` może stanowić komponent składowy interfejsu mechanizmu obsługi błędów.

W klasie `exception` została zdefiniowana specjalna publiczna funkcja członkowska (metoda) `what()`, która odgrywa w mechanizmie obsługi błędów bardzo ważną rolę. Ta rola jest ważna z dwóch powodów. Po pierwsze, funkcja `what()` zwraca na zewnątrz wskaźnik do C-napisu typu `char*`, który jest używany w programie jako komunikat informacyjny o ewentualnym wyjątku — błędzie. Po drugie, została zadeklarowana w klasie `exception` jako metoda wirtualna. Tym samym jest możliwe jej „przedefiniowanie” — nadpisanie (ang. [overwriting](#)) w każdej klasie pochodnej klasy `exception`, czyli np. klasie zdefiniowanej samodzielnie przez programistę.

Wykorzystanie polimorfizmu dynamicznego, polegającego na możliwości przesłonięcia (ang. [overriding](#)) metody wirtualnej `what()` z klasy `exception` w klasach wchodzących w skład łańcucha dziedziczenia — w tym w klasach zdefiniowanych samodzielnie przez programistę — pozwala na efektywną i prostą implementację mechanizmu obsługi błędów w programie.

Jak wspomniano wcześniej, w zdefiniowanej samodzielnie przez programistę klasie pochodnej klasy `exception` można przedefiniować (czyli zdefiniować inaczej) metodę `what()`. W definicji tej można określić dopuszczalny zbiór wyjątków, które mogą być przez nią zgłoszane. Ten zbiór wyjątków opisuje się za pomocą formalnej **specyfikacji wyjątków** (ang. [exception specification](#)). Specyfikację wyjątków zamieszcza się w formie listy na końcu sygnatury (nagłówka) funkcji `what(): throw(wyjątek_1, wyjątek_2, ...)`, gdzie `wyjątek_1, wyjątek_2` opisują zbiór dopuszczalnych wyjątków, które mogą być „wyrzucone” w trakcie wykonywania funkcji `what()`. Na przykład sygnatura `const char* what() const throw()` informuje, że funkcja `what()` nie może zgłosić żadnego wyjątku.

UWAGA

W standardzie C++11 i wyższych nastąpiły zmiany w sposobie specyfikowania wyjątków. Więcej informacji na ten temat można znaleźć w dokumentacji języka C++ na stronie https://en.cppreference.com/w/cpp/language/except_spec.

Specyfikacja wyjątków stanowi „pisemną” (formalną) gwarancję, że funkcja nie zgłosi żadnego innego wyjątku niż te, które zostały wymienione na liście po słowie kluczowym `throw`.

UWAGA

Więcej informacji dotyczących wyjątków standardowych można znaleźć w dokumentacji języka C++ na stronie <https://en.cppreference.com/w/cpp/error/exception>.

Przykład 19.6

```
#include <iostream>
#include <exception> // Dołączenie biblioteki exception pozwala wykorzystać w programie klasę exception.
using namespace std;

// Definicja klasy pochodnej Wyjatek względem klasy exception:
class Wyjatek : public exception {
public:
    // Definicja funkcji what():
    const char* what() const throw() {
        return "Błąd! Próba dzielenia przez zero!";
    }
    /* UWAGA
     * Funkcja what() została zdefiniowana w klasie bazowej exception jako metoda wirtualna.
     * Tym samym można ją „przedefiniować” (nadpisać) w klasie pochodnej klasy exception — jak tutaj.
     * Funkcja what() zwraca na zewnątrz wskaźnik do C-napisu.
     */
};

int main() {
    try { // początek kodu, w którym jest realizowana inspekcja kodu źródłowego
        double licznik = 1, mianownik = 0;
        if (mianownik == 0) {
            // Definicja obiektu wyjątek będącego instancją klasy Wyjatek:
            Wyjatek wyjątek;
            // Zgłoszenie wyjątku — błędu:
            throw wyjątek;
        }
        /* UWAGA
```

```

* Błąd zostanie zgłoszony tylko w przypadku, gdy wartość zmiennej mianownik jest równa 0.
*/
}

else {
    // Kod, który zostanie wykonany, jeśli nie stwierdzono błędu:
    cout << "Licznik = " << licznik << endl;
    cout << "Mianownik = " << mianownik << endl;
    cout << "Iloraz wynosi " << licznik/mianownik << endl;
}

// Procedura obsługi wyjątku:
catch(exception& e) {
    /* UWAGA
     * Wyjątek — obiekt należący do klasy bazowej exception jest przekazywany do procedury obsługi przez
     * referencję. Celem jest umożliwienie przechwycenia wyjątków będących obiektami klas pochodnych
     * względem klasy bazowej exception. Tutaj dotyczy to obiektu wyjątek będącego instancją klasy Wyjatek.
     */
    cout << e.what() << endl;
}

return 0;
}

```

Funkcjonalność przedstawionego programu jest taka sama jak funkcjonalność programu zawartego w przykładzie 19.3. Różnice pomiędzy nimi tkwią w implementacji mechanizmu obsługi wyjątków (błędów).

W programie zaprezentowanym tutaj wykorzystano klasę `Wyjatek` zdefiniowaną samodzielnie przez programistę. Jest ona klasą pochodną klasy `exception`, czyli klasą będącej integralnym składnikiem biblioteki standardowej C++. W klasie `Wyjatek` zdefiniowano — a dokładniej: przedefiniowano — metodę `what()`, która została zadeklarowana w klasie bazowej `exception` jako wirtualna. Innymi słowy, metoda `what()`, zadeklarowana w klasie bazowej `exception` jako wirtualna, została nadpisana — przesłonięta (ang. *overridden*) — w klasie `Wyjatek`.

W nagłówku definicji funkcji `what(): const char* what() const throw()` nie wyspecyfikowano żadnego wyjątku. Tym samym funkcja ta nie generuje (nie zgłasza) żadnego wyjątku.

Ewentualny wyjątek (błąd) w programie zostaje zgłoszony w bloku `try`, jako skutek wykonania instrukcji `throw wyjątek` — pod warunkiem że wartość zmiennej `mianownik` jest równa 0. Należy zwrócić uwagę, że obiekt `wyjatek` jest instancją klasy pochodnej `Wyjatek`, a nie klasy bazowej `exception`.

Do procedury obsługi błędu (`catch`) przekazywany jest jednak nie obiekt `wyjatek`, ale obiekt `e` należący do klasy bazowej `exceptions`. Takie rozwiązanie wybrano w celach demonstracyjnych.

Wspomniane powyżej przekazanie obiektu `e` do procedury obsługi jest realizowane za pomocą parametru referencyjnego: `exception& e`. Dzięki temu procedura obsługi może przechwytywać nie tylko błędy — obiekty należące do klasy bazowej `exception` — ale również błędy będące obiektami jej klas pochodnych. Tutaj przechwytywany jest błąd — obiekt `wyjatek`, który jest instancją klasy `Wyjatek`.

Ćwiczenie 19.6

Zmodyfikuj program zawarty w przykładzie 19.6 — w definicji procedury obsługi błędu (`catch`) zamiast parametru formalnego `e` należącego do klasy `exception` użyj jako parametru obiektu `wyjatek` będącego instancją klasy `Wyjatek`. Ponadto zmień odpowiednio kod (treść) wspomnianej procedury obsługi oraz sygnaturę (nagłówek) definicji funkcji `what()` w klasie `Wyjatek`. Uzasadnij uzyskane rezultaty.



19.3. Pytania i zadania kontrolne

19.3.1. Pytania

- Podaj i omów kilka przyczyn, które mogą prowadzić do powstawania błędów w programie. Szczególną uwagę zwróć na błędy powstające w czasie wykonywania programu (ang. *runtime errors*).
- Opisz mechanizm obsługi błędów oparty na systemie komunikatów i kodów zwrotnych. Uzeglądni następujące składniki tego mechanizmu:
 - sterowanie przepływem (ang. *control flow*) w programie,
 - implementację interfejsu, odpowiedzialnego za komunikację programu z użytkownikiem.
- Opisz rolę poszczególnych komponentów składowych mechanizmu obsługi wyjątków. Weź pod uwagę komponenty składowe tego mechanizmu reprezentowane przez słowa kluczowe: `try`, `throw` i `catch`.
- Omów parametry definicji procedury obsługi błędu w mechanizmie obsługi wyjątków. Czy mechanizm obsługi wyjątków pozwala na definiowanie wielu procedur obsługi błędów?

19.3.2. Zadania

- Napisz program pozwalający obliczyć pole i obwód pola uprawnego o kształcie prostokątnym. Dane wejściowe mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Załącz, że długość i szerokość rozpatrywanego pola uprawnego powinny mieć wartości dodatnie, ale nieprzekraczające wartości 100. Przyjmij dodatkowo, że wymiary pola należy wprowadzić z klawiatury w formacie liczby rzeczywistej z jednym miejscem po przecinku (nie po kropce!), np. `12,3`. Uzeglądni mechanizm

obsługi błędów w czasie wykonywania programu oparty na systemie komunikatów zwrotnych dla użytkownika.

- 2.** Zrób jak w zadaniu 1., ale mechanizm obsługi ewentualnych błędów w czasie wykonywania programu uzupełnij o zwrotne kody błędów.
- 3.** Zrób jak w zadaniu 1., ale obsługę ewentualnych błędów w czasie wykonywania programu zaimplementuj na podstawie mechanizmu obsługi wyjątków. Załóż, że wyjątki (błędy) należą do typu podstawowego `float`. Ponadto każdy rodzaj błędu powinien być obsługiwany przez niezależną procedurę obsługi.
- 4.** Zrób jak w zadaniu 1., ale obsługę ewentualnych błędów w czasie wykonywania programu zaimplementuj na podstawie mechanizmu obsługi wyjątków. Załóż, że wyjątki (błędy) są instancjami samodzielnie zdefiniowanych klas. Ponadto każdy rodzaj błędu powinien być obsługiwany przez niezależną procedurę obsługi.

20

Przykłady implementacji wybranych algorytmów

Podstawy algorytmiki, w tym zasady opisu algorytmów w postaci schematu blokowego i pseudokodu, zostały przedstawione w rozdziale 1. podręcznika. W niniejszym rozdziale zaś zaprezentowano przykłady implementacji wybranych, popularnych algorytmów. W szczególności dotyczy to:

- wyznaczania największego wspólnego dzielnika dwóch liczb całkowitych na podstawie algorytmu Euklidesa,
- algorytmów sortowania tablic za pomocą metody bąbelkowej i przez proste wstawianie,
- algorytmu wyszukiwania binarnego zadanego elementu (liczby) w określonym zbiorze liczb.

Ponadto przedstawiono wykorzystanie wbudowanych — predefiniowanych — funkcji standardowych języka C++ zawartych w bibliotece *algorithm*, które pozwalają efektywnie zrealizować wymienione powyżej zadania.

20.1. Wyznaczenie największego wspólnego dzielnika

Największy wspólny dzielnik (ang. *greatest common divisor*) dwóch liczb całkowitych, w skrócie NWD, to największa liczba naturalna, która dzieli obie te liczby bez reszty. Przy tym wspomniane liczby mają dowolny znak.

**UWAGA**

W przypadku ogólnym NWD można wyznaczać dla dowolnej ilości liczb całkowitych.

Wyznaczenie NWD dwóch zadanych liczb całkowitych jest realizowane najczęściej z zastosowaniem **algorytmu Euklidesa** (ang. *Euclidean algorithm*).

Wspomniany powyżej algorytm jest oparty na założeniu, że wykonanie operacji odejmowania mniejszej liczby od większej — skutkujące zmniejszeniem wartości większej z liczb — nie powoduje zmiany wartości NWD. Tym samym odjęcie liczby mniejszej od większej większą liczbę razy (co można zrealizować w pętli) również nie spowoduje zmiany NWD. Algorytm kończy działanie, jeśli spełniony jest warunek, że bieżące (zmniejszone) wartości obu liczb są identyczne.

Algorytm wyznaczania NWD można wykorzystać w praktyce np. do skracania ułamków.

Przykład 20.1

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

// Definicja funkcji globalnej obliczNWD(), pozwalającej wyznaczyć NWD:
int obliczNWD(int l1, int l2) {
    while (l1 != l2) { // Iteracje są wykonywane, dopóki liczby l1 i l2 są różne.
        // Ustalenie nowej wartości (zmniejszenie) większej z liczb przez odjęcie od niej liczby mniejszej:
        if (l1 > l2) // Jeśli wartość l1 jest większa od wartości l2.
            l1 = l1 - l2;
        else if (l2 > l1) // Jeśli wartość l2 jest większa od wartości l1.
            l2 = l2 - l1;
    }
    /* UWAGA
     * Po wykonanych iteracjach wartość l1 jest równa wartości l2.
     * Pętla (i funkcja) kończy działanie, jeśli wartości obu liczb są takie same.
     */
    return l1; // jeśli liczby l1 i l2 są równe, funkcja zwraca wartość l1
}

int main() {
    int liczba1, liczba2; // liczby, dla których wyznaczany jest NWD
    int nwd; // NWD

    // Pobranie wartości dwóch liczb całkowitych z klawiatury:
    cout << "Dane wejściowe " << endl;
    cout << "Podaj wartość pierwszej liczby: ";
    cin >> liczba1;
```

```

cout << "Podaj wartość drugiej liczby: ";
cin >> liczba2;
// Wyznaczenie NWD:
nwd = obliczNWD(liczba1, liczba2);
// Wyświetlenie wyniku — wyznaczonej wartości NWD — na ekranie monitora:
cout << "Wynik" << endl;
cout << "NWD = " << nwd << endl;

return 0;
}

```

Przedstawiony program pozwala wyznaczyć największy wspólny dzielnik (NWD) dwóch liczb całkowitych. Przy czym rozpatrywane są wyłącznie liczby dodatnie — wartość 0 oraz liczby ujemne nie są tutaj uwzględniane. Obliczenie NWD jest realizowane na podstawie algorytmu Euklidesa z użyciem funkcji globalnej `obliczNWD()`.

Algorytm Euklidesa użyty w zaprezentowanym programie w definicji funkcji `obliczNWD()` można zaimplementować również w inny sposób — z zastosowaniem operatora *modulo* (%). Definicja funkcji `obliczNWD()` mógłaby mieć wtedy postać:

```

int obliczNWD(int l1, int l2) {
    int reszta;
    while ((l1 % l2) > 0) {
        reszta = l1 % l2;
        l1 = l2;
        l2 = reszta;
    }
    return l2;
}

```

Do obliczenia NWD można wykorzystać również wbudowaną funkcję standardową `_gcd()`, dostępną w standardzie C++17 lub wyższym. Wyrażenie zawierające wywołanie tej funkcji mógłby mieć w omawianym programie postać: `nwd = _gcd(liczba1, liczba2);`. Użycie funkcji `_gcd()` wymaga dołączenia do programu zasobów biblioteki `#include <algorithm>`.

Ćwiczenie 20.1

Zmodyfikuj program z przykładu 20.1 — uzupełnij/zmień implementację funkcji `obliczNWD()` w taki sposób, aby przy wyznaczaniu NWD uwzględniane były liczby całkowite o dowolnym znaku oraz liczba 0.

Przykład 20.2

```

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

```

```

// Definicja funkcji globalnej pozwalającej wyznaczyć NWD:
int obliczNWD(int l1, int l2) {
    // Wyznaczenie wartości bezwzględnych obu liczb reprezentowanych przez parametry formalne funkcji:
    l1 = abs(l1); l2 = abs(l2);
    /* UWAGA
     * Wyznaczenie wartości bezwzględnych liczb l1 i l2 ma na celu uwzględnienie w obliczeniach liczb całkowitych
     * o dowolnym znaku — co wynika z definicji NWD. Wartość NWD z definicji jest nieujemna.
     */
    if (l1 == 0) // Jeśli wartość liczby l1 wynosi 0.
        return l2; // NWD jest równe wartości bezwzględnej liczby l2.

    if (l2 == 0) // Jeśli wartość liczby l2 wynosi 0.
        return l1; // NWD jest równe wartości bezwzględnej liczby l1.

    if (l1 == l2) // Jeśli wartości bezwzględne obu liczb są równe.
        return l1; // NWD jest równe wartości bezwzględnej liczby l1, która jest równa wartości bezwzględnej l2.

    if (l1 > l2) // Jeśli wartość bezwzględna l1 jest większa od wartości bezwzględnej l2.
        // Wywołanie rekurencyjne funkcji obliczNWD():
        return obliczNWD(l1 - l2, l2);
    /* UWAGA
     * Pierwszym argumentem wywołania funkcji obliczNWD() jest zmniejszona wartość l1, drugim — l2.
     */
    if (l2 > l1) // Jeśli wartość bezwzględna l2 jest większa od wartości bezwzględnej l1.
        // Wywołanie rekurencyjne funkcji obliczNWD():
        return obliczNWD(l1, l2 - l1);
    /* UWAGA
     * Pierwszym argumentem wywołania funkcji jest zmniejszona wartość l2, drugim — l1.
     */
}

int main() {
    int liczba1, liczba2, nwd;
    cout << "Dane wejściowe " << endl;
    cout << "Podaj wartość pierwszej liczby: ";
    cin >> liczba1;
    cout << "Podaj wartość drugiej liczby: ";
    cin >> liczba2;
    // Wyznaczenie NWD:
    nwd = obliczNWD(liczba1, liczba2);
    cout << "Wynik" << endl;
    cout << "NWD = " << nwd << endl;

    return 0;
}

```

Funkcjonalność przedstawionego programu została rozszerzona względem funkcjonalności programu zawartego w przykładzie 20.1. Wspomniane rozszerzenie polega na uwzględnieniu w wyznaczeniu NWD liczb całkowitych o dowolnym znaku wraz z liczbą 0, a nie jedynie liczb dodatnich — jak w przykładzie 20.1.

Implementacja funkcji `obliczNWD()` zdefiniowanej na listingu 20.2 różni się znaczco od implementacji funkcji o tej samej nazwie zawartej w przykładzie 20.1. Mianowicie funkcja `obliczNWD()` zdefiniowana tutaj jest funkcją rekurencyjną. Funkcja ta wywołuje w swoim ciele „samą siebie”. Mamy zatem do czynienia z rekurencją bezpośrednią (ang. *direct recursion*).

Algorytm Euklidesa w funkcji rekurencyjnej `obliczNWD()` zdefiniowanej w przedstawionym programie można zaimplementować również przy użyciu operatora *modulo* (%). Definicja rozpatrywanej funkcji mogłaby mieć wtedy postać:

```
int obliczNWD(int l1, int l2) {
    l1 = abs(l1); l2 = abs(l2);
    if (l2 == 0)
        return l1;
    return obliczNWD(l2, l1 % l2);
}
```

Ćwiczenie 20.2

Zmodyfikuj program zawarty w przykładzie 20.2 — zamiast funkcji globalnej `obliczNWD()` wykorzystaj zdefiniowaną samodzielnie klasę `NWD`. Załącz, że klasa `NWD` zawiera dwie zmienne członkowskie: `liczba1` i `liczba2`, reprezentujące liczby całkowite, dla których wyznaczany jest NWD, oraz metodę `obliczNWD()` o funkcjonalności analogicznej do funkcjonalności funkcji `obliczNWD()` z przykładu 20.2.

20.2. Sortowanie tablic

W ogólności **sortowanie** (ang. *sorting*) może dotyczyć różnych struktur (rodzajów) danych, np. tablic, łańcuchów znaków, plików. Polega ono na uporządkowaniu — ustaleniu kolejności — danych w określonej strukturze według zadanego klucza (reguły).

Przykładami struktur danych, których elementy składowe można poddać sortowaniu, są tabela 1-wymiarowa złożona z liczb rzeczywistych czy też tabela zawierająca łańcuchy znaków.

Istnieje wiele algorytmów sortowania tablic 1-wymiarowych: **sortowanie bąbelkowe** (ang. *bubble sort*), **sortowanie przez wstawianie** (ang. *insertion sort*), **sortowanie przez wybór** (ang. *selection sort*) itd.

W przykładach przedstawionych w tym punkcie zaprezentowano przykładowe implementacje algorytmów sortowania bąbelkowego (przykład 20.3) i sortowania przez wstawianie (przykład 20.4) wykonywane na tablicach 1-wymiarowych, w których elementami składowymi są liczby.

Przykład 20.3

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

// Definicja funkcji pozwalającej posortować elementy tablicy liczbowej metodą bąbelkową:
void sortowanieBabelkowe(float t[], int n) {
    int i, j;
    float temp;
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (t[j] < t[i]) {
                temp = t[i];
                t[i] = t[j];
                t[j] = temp;
            }
        }
    }
}

/* UWAGA
* Funkcja sortowanieBabelkowe() pozwala posortować elementy 1-wymiarowej tablicy liczbowej
* w porządku rosnącym — od elementu najmniejszego do największego.
*/

// Definicja funkcji pozwalającej wprowadzić wartości elementów tablicy z klawiatury:
void tablicaWejscie(float t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = ";
        cin >> t[i];
    }
}

// Definicja funkcji pozwalającej wyświetlić wartości elementów tablicy na ekranie:
void tablicaWyjscie(float t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = " << t[i] << endl;
    }
}

int main() {
    const int n = 5; // rozmiar tablicy
    float tablica[n]; // 1-wymiarowa tablica liczbową o rozmiarze n

    cout << "Wprowadź wartości elementów tablicy:" << endl;
    tablicaWejscie(tablica, n);

    cout << "Zawartość tablicy wejściowej: " << endl;
    tablicaWyjscie(tablica, n);
}
```

```
// Posortowanie elementów tablicy tablica:
sortowanieBabelkowe(tablica, n);

cout << "Zawartość tablicy posortowanej: " << endl;
tablicaWyjscie(tablica, n);

return 0;
}
```

Przedstawiony program pozwala posortować elementy składowe 1-wymiarowej tablicy `tablica`, którymi są liczby rzeczywiste należące do typu `float`. Rozmiar tablicy `tablica` jest określony za pomocą stałej `n` o wartości ustalonej w programie na 5.

Sortowanie jest realizowane metodą bąbelkową z zastosowaniem funkcji `sortowanieBabelkowe()`. Funkcje `tablicaWejscie()` i `tablicaWyjscie()` mają rolę pomocniczą. Są elementami składowymi interfejsu programu. Wszystkie z wymienionych funkcji są funkcjami globalnymi.

Ćwiczenie 20.3

Zmodyfikuj program z przykładu 20.3 — zamiast funkcji `sortowanieBabelkowe()`, `tablicaWejscie()` i `tablicaWyjscie()` wykorzystaj zdefiniowaną samodzielnie klasę `Tablica`. Wspomniana klasa powinna zawierać zmienne członkowskie reprezentujące 1-wymiarową tablicę liczbową i jej rozmiar oraz metody członkowskie odpowiadające wymienionym powyżej funkcjom globalnym.

Przykład 20.4

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

// Definicja funkcji pozwalającej posortować elementy tablicy liczbowej przez wstawianie:
void sortowaniePrzezWstawianie(float t[], int n) {
    int i, j;
    float temp;
    for (i = 1; i < n; i++) {
        temp = t[i];
        j = i - 1;
        while ((j >= 0) && (t[j] > temp)) {
            t[j+1] = t[j];
            j = j - 1;
        }
        t[j+1] = temp;
    }
}
```

```

/* UWAGA
 * Funkcja sortowaniePrzezWstawianie() pozwala posortować elementy 1-wymiarowej tablicy liczbowej
 * w porządku rosnącym — od elementu najmniejszego do największego.
 */
void tablicaWejscie(float t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = ";
        cin >> t[i];
    }
}
void tablicaWyjscie(float t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = " << t[i] << endl;
    }
}

int main() {
    const int n = 5; // rozmiar tablicy
    float tablica[n]; // 1-wymiarowa tablica liczbową o rozmiarze n
    cout << "Wprowadź wartości elementów tablicy:" << endl;
    tablicaWejscie(tablica, n);
    cout << "Zawartość tablicy wejściowej: " << endl;
    tablicaWyjscie(tablica, n);
    // Posortowanie elementów tablicy tablica w porządku rosnącym:
    sortowaniePrzezWstawianie(tablica, n);
    cout << "Zawartość tablicy posortowanej: " << endl;
    tablicaWyjscie(tablica, n);

    return 0;
}

```

Funkcjonalność zaprezentowanego programu jest analogiczna do funkcjonalności programu z przykładu 20.3. Wartości elementów składowych tablicy `tablica` są wprowadzane z klawiatury, wyświetlane kontrolnie na ekranie, sortowane, a na końcu ponownie wyświetlane na ekranie.

Zastosowano tutaj **sortowanie przez proste wstawianie** (ang. *simple insertion sort*). Jest ono realizowane z użyciem funkcji globalnej `sortowaniePrzezWstawianie()`.

Ćwiczenie 20.4

Zmodyfikuj program z przykładu 20.4 — w funkcji `sortowaniePrzezWstawianie()` zamiast sortowania przez wstawianie proste wykorzystaj algorytm **sortowania przez wstawianie binarne** (ang. *binary insertion sort algorithm*), który jest nazywany również sortowaniem przez wstawianie z wyszukiwaniem binarnym (ang. *binary search*).

UWAGA

Zamiast samodzielnie definiować funkcje mające za zadanie posortowanie elementów tablicy za pomocą wybranego algorytmu, można wykorzystać wbudowaną funkcję standardową `sort()`. Funkcja ta jest dostępna w standardzie C++17 lub wyższym. W ogólności funkcję `sort()` można stosować zarówno do tablic, jak i do wektorów należących do typu `vector`.

20.3. Wyszukiwanie binarne

W programowaniu często się zdarza, że trzeba wyszukać zadaną liczbę w zbiorze liczb przechowywanych w tablicy. Jeżeli wspomniana tablica jest posortowana, zadanie to można zrealizować na podstawie **algorytmu wyszukiwania binarnego** (ang. *binary search algorithm*).

Przykład 20.5

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

// Definicja funkcji pozwalającej wyszukać zadaną liczbę w tablicy:
int wyszukiwanieBinarne(int t[], int n, int liczba) {
    // t — posortowana tablica liczbowa
    // n — liczba elementów (rozmiar) tablicy
    // liczba — zdana liczba do wyszukania w tablicy

    int poczatek = 0;
    int koniec = n - 1;
    int srodek = (poczatek + koniec) / 2;
    int wynik;

    /* UWAGA
     * Jeśli w tablicy znaleziono element składowy o wartości liczba, to w zmiennej wynik zapisywany
     * jest indeks tego elementu.
     * Jeśli tablica nie zawiera elementu składowego o wartości liczba, to do zmiennej wynik podstawiana
     * jest wartość -1.
     */
    while (poczatek <= koniec) {
        if (t[srodek] < liczba) {
            poczatek = srodek + 1;
        }
        else {
            if (t[srodek] == liczba) {
                wynik = srodek;
                break;
            }
        }
    }
}
```

```

        else {
            koniec = srodek - 1;
        }
    }
    srodek = (początek + koniec) / 2;

}
if (początek > koniec) {
    wynik = -1;
}

return wynik;
}

// Funkcja pomocnicza:
void tablicaWyjscie(int t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = " << t[i] << endl;
    }
}
int main() {
    // Deklaracja i inicjalizacja tablicy:
    int tablica[] {10, 21, 32, 43, 54, 65, 76, 87, 98, 109};
    /* UWAGA
     * 1-wymiarowa tablica liczbową powinna być posortowana.
     */
    int n = sizeof(tablica) / sizeof(int); // liczba elementów tablicy

    cout << "Tablica wejściowa" << endl;
    // Prezentacja tablicy wejściowej na ekranie monitora:
    tablicaWyjscie(tablica, n);
    cout << endl;

    int liczba = 65; // szukana liczba
    cout << "Szukana liczba: " << liczba << endl;
    cout << endl;

    // Wyszukanie zadanej liczby w tablicy:
    int indeks = wyszukiwanieBinarne(tablica, n, liczba);

    // Prezentacja wyników na ekranie monitora:
    cout << "Wyniki" << endl;
    if (indeks != -1) {
        cout << "Pozycja (indeks) szukanej liczby w tablicy: "
            << indeks << endl;
    }
}

```

```

        cout << "Wartość elementu tablicy o indeksie " << indeks
        << ": " << tablica[indeks] << endl;
    }
else {
    cout << "Zadanej liczby nie znaleziono!" << endl;
}

return 0;
}

```

W zaprezentowanym programie przedstawiono przykładową implementację algorytmu wyszukiwania binarnego liczby całkowitej `liczba` w tablicy liczbowej `tablica`. Rozmiar (liczba elementów składowych) tej tablicy jest przechowywany w zmiennej `n`.

Wyszukiwanie binarne jest realizowane z wykorzystaniem funkcji `wyszukiwanieBinarne()`. Parametr `t` omawianej funkcji reprezentuje tablicę liczbową o rozmiarze `n`, która jest posortowana. Poszukiwana liczba odpowiada parametrowi `liczba`. Omawianą funkcję można zdefiniować również jako funkcję rekurencyjną.

Funkcja `wyszukiwanieBinarne()` zdefiniowana tutaj pozwala znaleźć zadaną liczbę w tablicy oraz indeks elementu, w którym liczba ta jest przechowywana. Implementacja tej funkcji nie uwzględnia jednak przypadku, w którym w tablicy zapisanych jest więcej elementów (liczb) o wartościach odpowiadających szukanej liczbie — co oczywiście można zmienić.

Wyszukiwanie binarne można również zrealizować z użyciem wbudowanej funkcji standardowej `binary_search()` z biblioteki `algorithm`. Funkcja ta jest dostępna w standardzie C++20 i nowszych. Wyrażenie zawierające wywołanie tej funkcji w omawianym programie mogłoby mieć postać: `bool wynik = binary_search(tablica, tablica + n, 65);`, gdzie:

- zmienna logiczna `wynik` reprezentuje wynik wyszukiwania: `true` — zadaną liczbę znaleziono, `false` — przeciwnie,
- `tablica` (a faktycznie `tablica + 0`) to początek zakresu, w którym jest prowadzone wyszukiwanie, a `tablica + n` to koniec tego zakresu,
- 65 odpowiada poszukiwanej liczbie.

Funkcję biblioteczną `binary_search()` można również stosować do zmiennych (wektorów) należących do typu `vector`.

Ćwiczenie 20.5

Zmodyfikuj kod źródłowy programu z przykładu 20.5 — zaimplementuj funkcję `wyszukiwanieBinarne()` jako funkcję rekurencyjną.



20.4. Pytania i zadania kontrolne

20.4.1. Pytania

1. Omów i opisz za pomocą pseudokodu wyznaczanie największego wspólnego dzielnika (NWD) dwóch liczb całkowitych na podstawie algorytmu Euklidesa bez stosowania rekurencji.
2. Zapisz algorytm Euklidesa do wyznaczania NWD dwóch liczb całkowitych za pomocą schematu blokowego. Wykorzystaj rekurencję.
3. Przedstaw zalety i wady sortowania tablic za pomocą metody bąbelkowej.
4. Przedstaw algorytm sortowania tablic metodą bąbelkową za pomocą pseudokodu i schematu blokowego.
5. Na czym polega **efektywność algorytmu** (ang. *algorithm efficiency*)?
6. Omów pojęcie **złożoności obliczeniowej** (ang. *computational complexity*) algorytmu, w tym jego **złożoność czasową** (ang. *time complexity*) i **złożoność pamięciową** (ang. *memory complexity*).

20.4.2. Zadania

1. Napisz program pozwalający wyznaczyć najmniejszą wspólną wielokrotność (NWW) dwóch zadanych liczb całkowitych. Wykonaj program w dwóch wariantach:
 - bez stosowania rekurencji,
 - z wykorzystaniem rekurencji.
2. Napisz program pozwalający posortować elementy składowe tablicy liczbowej w porządku malejącym. Wykorzystaj metodę **sortowania szybkiego** (ang. *quick sort, quicksort*). Omów założenia i ideę algorytmu „**dziel i zwyciężaj**” (ang. *divide and conquer*) oraz jego wykorzystanie w praktyce.
3. Napisz program pozwalający posortować w porządku rosnącym elementy tablicy 1-wymiarowej złożonej z łańcuchów znaków. Załącz, że wspomniane łańcuchy to nazwiska pracowników działu księgowości w zakładzie pracy.
4. Napisz program umożliwiający wyszukanie zadanego nazwiska w tablicy, w której elementy składowe to łańcuchy znaków reprezentujące nazwiska uczniów w grupie klasowej. Załącz, że dane nazwisko może występować w tablicy wielokrotnie. Określ liczbę wystąpień szukanego nazwiska w tablicy.
5. Napisz program pozwalający posortować elementy składowe wektora liczbowego należącego do typu `vector` za pomocą funkcji wbudowanej `sort()` z biblioteki *algorithm*.
6. Napisz program pozwalający wyszukać zadaną liczbę w zestawie liczb zapisanych w wektorze należącym do typu `vector`. Wykorzystaj wbudowaną funkcję biblioteczną `binary_search()`.

Bibliografia

Książki

B. Stroustrup, *Programowanie. Teoria i praktyka z wykorzystaniem C++*, wyd. 3,
Helion, Gliwice 2020.

Źródła internetowe

<https://en.cppreference.com>

<http://cplusplus.com>

<https://www.learncpp.com>

<https://www.geeksforgeeks.org>

<https://www.tutorialspoint.com>

<https://www.w3schools.com>

Skorowidz

A

abstrakcje, 383 – 388, 392
danych, 393
akcesory, 286, 346
algorytm, 10
Euklidesa, 460
sortowanie tablicy, 463 – 466
wyszukiwanie binarne, 467
wyznaczanie NWD, 459
algorytmika, 13
alias, 44
alokacja pamięci
dynamiczna, 114, 137
statyczna, 124
w czasie komplikacji, 104
w czasie wykonywania
programu, 115
Apache NetBeans, 34
argumenty, 198
arytmetyka wskaźnika, 133
automatyczne skalowanie, 133

B

biblioteka
ctype, 258
cmath, 256
cstdio, 261
cstdlib, 260, 263
cstring, 154
standardowa, 255
string, 162
blok, 14
decyzyjny, 17
komentarza, 19
końcowy, 15
operacji, 16
początkowy, 15
podprogramu, 18
wejścia/wyjścia, 16
błedy, 431
logiczne, 452
obsługa, 432

przechwytywanie, 432
system kodów zwrotnych,
433
w czasie wykonywania
programu, 440, 452

C

ciało funkcji, 196
CLion, 36
C-napisy, 148
deklarowanie, 148
inicjowanie, 149
operacje, 154
wejścia/wyjścia, 152
przekazywanie do funkcji,
218
Code::Blocks, 30
CodeLite, 31
C-strukturny, 171
definicja, 172
deklaracja, 172
dynamiczne, 182
elementy członkowskie, 171
inicjalizacja, 176
jako parametry funkcji, 211
odwołania do pól, 174
pola, 171
wskaźniki, 179
zagnieżdżanie, 173, 179
zmienne struktury, 174
C-unie, 184
definicja, 185
deklaracja zmiennej, 186
elementy członkowskie,
185 – 188
inicjalizacja zmiennej, 186

D

debugger, 12, 30
debugowanie, 12
dedukcja typu danych, 47, 415
definicja
funkcji, 196

klasy, 273, 274
stałej, 246
strukturny, 172
szablonów funkcji, 413
typów danych, 247
unii, 185
deklaracja
C-napisu, 148
funkcji, 194
łańcucha, 158
strukturny, 172
tablicy, 122, 129
zmiennej, 41
globalnej, 224
obiektoowej, 278
strukturyowej, 174
wskaźnikowej, 106
dekompozycja, 270
delegowanie konstruktorów, 332
destruktory, 337
dokumentacja programu, 9
techniczna, 13
użytkownika końcowego, 13

dyrektywa

#define, 245
definiowanie
makrofunkcji, 248
definiowanie stałych,
246
definiowanie typów
danych, 247
#elif, 251
#else, 251
#endif, 251
#if, 251
#ifdef, 251
#ifndef, 251
#include, 241

dyrektywy

komplikacji warunkowej, 251
preprocesora, 69
dziedziczenie, 351, 362
chronione, 353
hierarchiczne, 359

mieszane, 359
 pojedyncze, 358
 prywatne, 296, 354
 publiczne, 296, 353
 wielokrotne, 358
 wielopoziomowe, 359
 wskaźniki, 373

E

Eclipse CDT, 33
 elementy członkowskie
 klasy, 275
 funkcje, 276
 funkcje statyczne, 286
 funkcje typu inline, 288
 prywatne, 275
 publiczne, 275
 statyczne, 283
 zmienne, 275
 zmienne statyczne, 284
 obiektów, 279
 struktury, 171
 unii, 187, 188
 enkapsulacja, 343
 enumerator, 95

F

framework Qt, 33
 funkcja, 193, 269
 abs, 256
 append, 163
 assign, 163
 atof, 260
 atoi, 260
 atol, 260
 ceil, 256
 compare, 163
 cos, 256
 exp, 256
 find, 163
 floor, 256
 insert, 163
 isalnum, 258
 isalpha, 258
 iscntrl, 258
 isdigit, 258
 islower, 258
 isprint, 258

ispunct, 258
 isupper, 258
 isxdigit, 258
 length, 163
 log, 256
 log10, 256
 pow, 256
 rand, 263
 replace, 163
 round, 256
 sizeof, 156
 sqrt, 256
 srand, 263
 strcat, 154
 strchar, 154, 156
 strcmp, 154
 strcpy, 154
 strlen, 154
 strstr, 154, 156
 strtod, 260
 strtof, 260
 substr, 163
 tan, 256
 tolower, 258
 toupper, 258
 trunc, 256

funkcje
 argumenty, 198
 biblioteczne, 255
 ciało, 196
 członkowskie klasy, 276
 bazowej, 377
 statyczne, 286
 definicja, 196
 deklaracja, 194
 implementacja, 196
 inline, 235, 288
 konwersji typu danych, 260
 łańcuchowe, 154, 162
 matematyczne, 256
 parametry, 198
 C-struktury, 211
 łańcuchy znaków, 218
 obiekty, 292
 tablice, 213
 parametry domyślne, 222
 parametry formalne
 jako zmienna lokalna, 229

parametry wejściowe, 194
 jako referencje do
 stałych const, 200, 293
 jako wskaźniki do
 stałych const, 202, 293
 przekazywane przez
 wartość, 198
 parametry wyjściowe, 194
 przekazywane przez
 referencję, 204
 przekazywane przez
 wskaźniki, 208
 predefiniowane, 269
 przeciążone, 233, 428
 rekurencyjne, 236
 specjalizowane, 417
 standardowe, 194
 szablonowe, 413
 definiowanie, 413
 specjalizacja, 417
 wywołanie, 414
 uogólnione, 413
 wejścia/wyjścia, 261
 wywoływanie, 196, 414
 zaprzyjaźnione, 401
 zdefiniowane przez
 użytkownika, 194, 269
 zmienne
 globalne, 224
 lokalne, 226
 znakowe, 258

G

getter, 346
 globalna przestrzeń nazw, 224
 graficzny interfejs użytkownika,
 GUI, 31

H

hermetyzacja danych, 343
 ukrywanie danych, 345

I

IDE, integrated development
 environment, 12, 29
 Apache NetBeans, 34
 CLion, 36
 Code::Blocks, 31

environment
 CodeLite, 31
 Eclipse CDT, 33
 NetBeans, 33
 Visual Studio, 35
 Visual Studio Code, 36
identyfikator, 39
 T, 414
implementacja, 9
 abstrakcji, 385, 387, 392
 algorytmu, 459
 funkcji, 196
 interfejsu, 390
 polimorfizmu, 428
informacje o błędach, 433
inicjalizacja
 agregacyjna, 321
 bezpośrednia, 46
 jednolita
 bezpośrednia, 46
 kopiąca, 46
 klamrowa, 46
 konstruktorowa, 46, 324
listowa
 bezpośrednia, 321
 kopiąca, 321
wartościami
 domylnymi, 315
 zadanymi, 320
zerowa, 46
inicjalizator wewnętrzklasowy, 315
inicjowanie
 C-napisów, 149
 łańcuchów, 158
 obiektów, 315, 320, 324
 struktur, 176
 tablic, 125
 zmiennych, 45
instancja
 klasy, 273
 szablonu, 421
instrukcja
 break, 92
 continue, 93
 warunkowa
 if, 73
 if zagnieżdżone, 76
 if-else, 74

if-else zagnieżdżone, 78
switch, 82
instrukcje
 bloku, 72
 warunkowe, 72
 wyboru, 72
 złożone, 72
interfejsy, 383, 389
interpreter, 11
inżynieria oprogramowania, 9
iteracja, 87
J
język
 interpretowany, 11
 kompilowany, 11
 programowania, 11
K
kapsułkowanie, 343
klasa exception, 452
klasy, 273
 abstrakcyjne, 383, 384
 bazowe, 352
 definicja, 273, 274
 dziedziczenie
 chronione, 353
 hierarchiczne, 359
 mieszane, 359
 pojedyncze, 358
 prywatne, 354
 publiczne, 353
 wielokrotne, 358
 wielopoziomowe, 359
elementy członkowskie
 prywatne, 275
 publiczne, 275
 statyczne, 283
funkcje członkowskie, 276, 389
 statyczne, 286
 typu inline, 288
metody, 273
 dostępowe, 346
pochodne, 352
specyfikator dostępu, 274
uogólnione, 420
zaprzyjaźnione, 407
zmienne członkowskie, 273, 275
 statyczne, 284
kod
 maszynowy, 11
 obiektowy, 11
 źródłowy, 11
kodowanie programu, 11
kody zwrotne, 433
komentarze
 jednoliniowe, 40
 wieloliniowe, 40
kompilator, 11
 g++, 30
komunikat o błędzie, 433
konkatenacja, 164
konstrukcja try-catch, 431
konstruktory, 160, 303, 362
 delegowanie, 332
 domyślne, 304, 315, 337
 programisty, 308
 kopiące, 328
 jawne wywołanie, 332
 niejawne wywołanie, 329
 prototyp, 328
 niejawne, 304
parametryczne, 309, 315, 337, 348
 przeciążanie, 312
kontenery, 139
konwersja typu
 automatyczna, 62
 jawna, 64
 niejawna, 62
 standardowa, 63
kwalifikator zakresu, 66
L
linkowanie, 11
lista inicjalizacyjna, 320, 326
literały
 całkowite, 49
 logiczne, 50
 łańcuchowe, 50, 147
 zmiennoprzecinkowe, 50
 znakowe, 50

L

- łańcuch znaków, 147
 - deklarowanie, 158
 - inicjowanie, 158
 - jako parametry funkcji, 218
 - konkatenacja, 164
 - operacje wejścia/wyjścia, 161
 - typu string, 157
- łączniki
 - wewnętrzne, 18
 - zewnętrzne, 18

M

- makrodefinicje, 246
- makrofunkcje, 248
- mechanizm abstrakcji, 383
 - interfejsy, 389
 - klasy abstrakcyjne, 384
 - pliki nagłówkowe, 396
- metody, 273, 280, 367, 389
 - abstrakcyjne, 383, 451
 - czysto wirtualne, 384
 - dostępowe, 346
 - przeciążanie, 367
 - przesłanianie, 370, 378
 - stacyczne, 286, 288
 - wirtualne, 377, 378
- MinGW, Minimalist GNU for Windows, 30
- modyfikator, 42, 43, 49
 - const, 112

N

- nagłówek funkcji, 195
- nawiasy
 - kątowe, 242
 - klamrowe, 71, 226
- NetBeans, 33
- notacja
 - funkcyjna, 214
 - tablicowa, 214

O

- obiekty, 273
 - inicjalizacja
 - agregacyjna, 321
 - inicjalizator
 - wewnętrzny, 315

- konstruktor domyślny, 315
- konstruktor kopiący, 328
- konstruktor
 - parametryczny, 315
- konstruktorowa, 324
- lista inicjalizacyjna, 320, 326
- listowa bezpośrednia, 321
- listowa kopiąca, 321
- jako parametry funkcji, 292
- konstruktory, 303
- odwołania
 - do funkcji
 - członkowskich, 279
 - do zmiennych
 - członkowskich, 279
- wskaźniki, 290
- obsługa
 - błędów, 432, 440
 - wyjątków, 440, 442
- odwijanie stosu, 445
- operacje
 - na C-napisach, 154
 - wejścia/wyjścia, 152, 161
- operand, 51
- operator, 51
 - adresu &, 105
 - delete[], 115, 137
 - derefencji *, 109, 135, 181
 - dostępu do elementu
 - członkowskiego, 279
 - indeksowy, 123, 134
 - new, 115, 137
 - postdekrementacji, 54
 - postfiksowy, 52
 - postinkrementacji, 54
 - predekrementacji, 54
 - prefiksowy, 52
 - preinkrementacji, 54
 - przecinkowy, 61
 - przypisania prostego, 52
 - referencji &, 200
 - rozpoznawania zakresu, 66
 - sizeof, 60
 - strzałkowy, 180
 - warunkowy, 84

- wstawiania, 69, 152
- wyboru elementu
 - członkowskiego, 174, 180, 279, 296
- wyodrębnienia, 69, 152
- zakresu, 66, 284, 286, 303
- operatory
 - arytmetyczne, 53
 - bitowe, 56, 57
 - dwuargumentowe, 54
 - jednoargumentowe, 54
 - logiczne, 59
 - porównania, 58
 - priorytet, 55
 - przypisania, 52
 - złożonego, 53
- oprogramowanie otwarte, 30
- optymalizacja programu, 12

P

- paradygmat programowania, 267
- parametry
 - domyślne, 222
 - formalne, 195
 - jako zmienne lokalne, 229
 - funkcji, 198
 - C-struktury, 211
 - jako wskaźniki do stałych const, 202, 208
 - łańcuchy znaków, 218
 - obiekty, 292
 - opcjonalne, 222
 - przekazywane przez referencję, 200, 204
 - przekazywane przez wartość, 198
 - tablice, 213
 - szablonowe, 420
 - wejściowe, 194, 293
 - wyjściowe, 194, 293
- pętla, 85
 - do-while, 88
 - for, 89
 - foreach, 142
 - while, 86
- pętle zagnieżdżone, 94

planowanie rozwiązań
 problemu, 10
pliki nagłówkowe, 69, 242, 396
POD, plain old data, 296
podprogram, 193, *Patrz* funkcja
pola, 171
polimorfizm
 dynamiczny, 373, 385
 metody wirtualne, 377
 statyczny, 367
 szablony, 428
późne wiązanie, 373
preprocesor, 241
procedura, 268
 obsługi błędu, 440, 441
program
 główny, 71
 komputerowy, 9, 39
programowanie
 deklaratywne, 268
 imperatywne, 267
 komputerowe, 9
 obiektowe, 272
 proceduralne, 267, 268
 dekompozycja, 270
 strukturalne, 267
 struktury iteracyjne, 271
 struktury sekwencji, 270
 struktury wyboru, 270
 uogólnione, 428
projektowanie oprogramowania, 9
promocja typu, 63
prototyp
 funkcji, 195
 konstruktora kopiącego, 328
przeciążanie
 funkcji, 233, 428
 konstruktów, 312
 metod, 367
przesłanianie
 metod, 370
 zmiennych, 231
przestrzeń nazw, 65
pseudokod, 10, 24

R
rekord, 296
rekurencja
 bezpośrednia, 236
 pośrednia, 236
relacje dziedziczenia, 352
rzutowanie typu, 64

S
schemat blokowy, 10, 14, 19–23
setter, 346
słowo kluczowe, 39
 auto, 47
 bool, 43
 break, 82, 92
 catch, 441
 char, 43
 cin, 68, 152, 161
 class, 414
 const, 111, 225
 continue, 93
 cout, 68, 152, 161
 decltype, 47
 delete, 115, 137
 double, 43
 else, 76, 78
 float, 43
 if, 73
 int, 42
 new, 115, 137
 static, 286
 string, 147, 157
 struct, 172
 switch, 82
 template, 414, 421
 throw, 442
 try, 440
 typedef, 44, 45
 using, 44
 vector, 139
 virtual, 377
 void, 43
sortowanie
 bąbelkowe, 463
 przez proste wstawianie, 466
 przez wstawianie, 463
 przez wstawianie binarne, 466

przez wybór, 463
specjalizowanie szablonów
 funkcji, 417
 klas, 425
specyfikacja wyjątków, 453
specyfikator
 decltype, 47
 dostępu, 274, 353
 private, 274
 protected, 274
 public, 274
 typedef, 44
stała, 49, 246
 nazwana, 50
stan programu, 268
standardowa
 biblioteka C++, 67
 przestrzeń nazw, 67
standardowy strumień
 wejściowy, 68
 wyjściowy, 68
sterta, 114, 137
stos, 104, 124
struktura programu, 70
struktury, 171, 296, *Patrz także*
 C-struktury
 sterujące, 270, 271
szablon
 funkcji
 definiowanie, 413
 specjalizacja, 417
 klasy, 420
 specjalizacja, 425

Ś
środowisko wykonawcze, 30

T
tabela relacyjnej bazy danych, 296
tablice
 deklaracja, 122, 129
 dynamiczne, 137
 indeks, 122
 inicjalizacja, 125, 131
 jako parametry funkcji, 213
 jednowymiarowe, 121
 odwołanie się do elementu, 129

- stałe, 124
 statyczne, 121, 124
 wielowymiarowe, 121, 128
 testowanie programu, 12
 translacja, 11
typ
 dynamiczny wskaźnika, 377, 380
 statyczny wskaźnika, 374
 void, 43
typy danych
 agregacyjne, 122
 całkowite, 42
 logiczne, 43
 podstawowe, 42
 predefiniowane, 41
 proste, 42
 uogólnione, 413, 420
 wbudowane, 41
 wskaźnikowe, 107
 wyliczeniowe, 95
 zdefiniowanych przez użytkownika, 41
 złożone, 42
 zmiennoprzecinkowe, 43
 znakowe, 43
- U**
 ukrywanie danych, 345
 unia, *Patrz* C-unie
- V**
 Visual Studio, 35
 Visual Studio Code, 36
- W**
 walidacja programu, 12
 wartość null, 116, 148
 wczesne wiązanie, 368, 370
 wektory, 139
 weryfikacja programu, 12
 wiązanie, 368
 wskaźniki, 103, 106, 111, 132
 do obiektów, 290
 do struktur, 179
 jako parametry funkcji, 208
 na stałą, 112 – 114
 typ dynamiczny, 377, 380
- typ statyczny, 374
 w mechanizmie dziedziczenia, 373
 wyjątki, 431
 obsługa, 431, 440, 442
 standardowe, 452
 błędy logiczne, 452
 błędy
 w czasie wykonywania programu, 452
 specyfikacja, 453
- wyrażenia, 46, 61
 bitowe, 62
 całkowite, 62
 logiczne, 62
 rzeczywiste, 62
 stałe, 62
 złożone, 62
- wyszukiwanie binarne, 467
- wywołanie
 funkcji, 194, 196
 szablonowej, 414
- konstruktora
 domyślnego, 337
 parametrycznego, 337
- metody
 przeciążonej, 367
 statycznej, 286
- Z**
 zakres
 blokowy, 226
 globalny, 224
 klasy, 312
- zbiory nagłówkowe, 255
- zdefiniowanie problemu do rozwiązania, 10
- zintegrowane środowisko programistyczne, IDE, 12, 29
- zmienne, 40, 45
 członkowskie, 171
 klasy, 275
 statyczne, 284
- deklaracja z dedukcją typu, 47
- globalne, 224
 czas życia, 224
 deklaracja, 224
 identyfikator, 224
- jako stałe, 225
 inicjalizacja, 45
 instancyjne, 284
 klasowe, 284
 odwołania, 284
 lokalne, 226
 definiowane w bloku kodu, 228
 definiowane w ciele funkcji, 226
 parametry formalne, 229
- obiektywne, 279
 deklaracja, 278
- przesłanianie, 231
- statyczne, 124, 224
- sterujące, 87
- struktury, 174
- typu unijnego, 186
- wskaźnikowe, 106, 107
 deklaracja, 106
 wyliczeniowe, 96
- znak &, 105, 200
- znak *, 106, 109, 135, 181
- znak pusty, 148
- znak T, 414
- znaki specjalne, 50

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
<http://program-partnerski.helion.pl>



KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie z dostępem do nowoczesnych narzędzi - videokursów, e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL