DO NOWEJ PODSTAWY PROGRAMOWEJ

## Część 2
Programowanie obiektowe

# Kwalifikacja INF.04
## Projektowanie, programowanie i testowanie aplikacji

**Podręcznik do nauki zawodu**
**technik programista**

Piotr Siewniak

Helion

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 właścicieli.

Autor oraz wydawca dołożyli 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*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-8101-8

Copyright © Helion S.A. 2021

Printed in Poland.

- Kup książkę
- Poleć książkę
- 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



Kup ksi k

Pole ksi k

**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 kompilacji 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



Kup ksi k

Pole ksi k

**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 kopiujący .... 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



Kup ksi k    Pole ksi k

**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



Kup ksi k Pole ksi k

# 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 pozwalają 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 podyktowany 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 gruntownej 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.

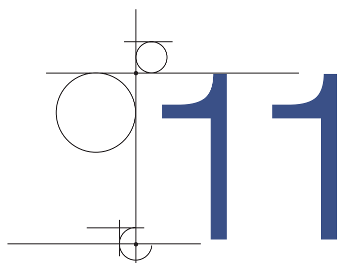


Kup ksi k

Pole ksi k

Kup ksi k

Pole ksi k



# 11 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 rozwiązania określonego problemu — metodę rozwiązania 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 rozwiązaniu 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 rozwiązanie 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 ściśle 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 rozwiązania 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



Kup ksi k Pole ksi k

podprogramów — **funkcje** (ang. *functions*). Funkcje znane z Pascala 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 paradygmat programowania, związany z deklaratywnym modelem programowania, a nie imperatywnym.

Funkcja odpowiada w programie za rozwiązanie 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 (zaimportować) 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 podzbiorem 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 ścisły 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.



Kup ksi k Pole ksi k

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 ogólne 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 rozgałę-**



Kup ksi k Pole ksi k

**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 określoną — zadaną 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 zagnieżdżać. 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 programiście 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 zagnieżdżona), która ma jedno wejście i jedno wyjście, jest równoważna (funkcjonalnemu) blokowi kodu.



Kup ksi k Pole ksi k

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 wywoływania 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 (pudełek) 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.



Kup ksi k Pole ksi k

**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.



Kup ksi k Pole ksi k

**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`.



Kup ksi k Pole ksi k

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 jawny 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.



Kup ksi k Pole ksi k

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 `Uczen`, która będzie miała następujące elementy członkowskie publiczne:

- zmienne członkowskie: `imie`, `nazwisko`, `klasa`, należące do typu łańcuchowego,



Kup ksi k Pole ksi k

- 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`.



Kup ksi k Pole ksi k

**Ćwiczenie 11.2**

Zmodyfikuj kod źródłowy z przykładu 11.2 — zamiast klasy `Pracownik` zdefiniuj klasę `Uczen`, 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 wspominano, 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 strukturowych (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 strukturowych):

```
nazwa_klasy zmienna_1, zmienna_2, ... ;
```

Na przykład utworzenie obiektu `pracownik` należącego do klasy `Pracownik` można zrealizować za pomocą wyrażenia `Pracownik pracownik;`.



Kup ksi k Pole ksi k

## 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_obiektowa.zmienna_członkowska`**

gdzie `zmienna_obiektowa` 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;
    }
};
```



Kup ksi k Pole ksi k

```
// 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



Kup ksi k Pole ksi k

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 zadanych 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;
}
```



Kup ksi k Pole ksi k

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` jako instancję 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 strukturowego Data.
};
```



Kup ksi k Pole ksi k

```
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 `Uczen`, 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 będące instancjami 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 zadanych 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;
}
```



Kup ksi k Pole ksi k

W programie zdefiniowano klasę `Pracownik`, która zawiera dwie zmienne klasowe, `s_szkola` i `s_stanowisko`. W zmiennej `s_szkola` przechowywana jest nazwa szkoły, w której jest zatrudniony pracownik, a w zmiennej `s_stanowisko` — stanowisko służbowe tego pracownika.

Zmienne klasowe `s_szkola` 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 `Uczen` 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 wewnątrz, 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.



Kup ksi k Pole ksi k

### 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 instancyjnych:
    string imie;
    string nazwisko;
    // Prototyp metody statycznej s_zwrocSzkola():
    static string s_zwrocSzkola();
    // Definicja metody statycznej s_ustawSzkola():
    static void s_ustawSzkola(string pSzkola) {
        s_szkola = pSzkola;
    }
    // Definicja metody instancyjnej 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_pobierzSzkola():
string Pracownik::s_zwrocSzkola() {
    return s_szkola;
}

// PROGRAM GŁÓWNY
int main() {
    // Utworzenie obiektu pracownik1:
    Pracownik pracownik1;
    // Nadanie wartości zmiennym członkowskim instancyjnym obiektu pracownik1:
    pracownik1.imie = "Jan";
    pracownik1.nazwisko = "Kowalski";
    cout << "Dane pracownika: " << endl;
    // Odwołanie się do zmiennych instancyjnych 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 kompilacji programu.



Kup ksi k Pole ksi k

**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`.



Kup ksi k Pole ksi k

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;
```



Kup książkę Poleć książkę

```
    // 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 kompilacji 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;
```



Kup ksi k Pole ksi k

```
    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 `Uczen`, 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



Kup ksi k Pole ksi k

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&);
```



Kup ksi k Pole ksi k

```
// PROGRAM GŁÓWNY
int main() {
    // Utworzenie obiektu pracownik 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.



Kup ksi k Pole ksi k

### 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.



Kup ksi k Pole ksi k

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" klasy. 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*).



Kup książkę Poleć książkę

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 strukturowe 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;
};
```



Kup ksi k Pole ksi k

```
/* 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 strukturowych (obiektów) `samochod1` i `samochod2` uzyskano w klasyczny sposób, tj. za pomocą ich identyfikatorów, operatora wyboru elementu członkowskiego `.` oraz nazw pól, np. `samochod2.marka`.



Kup ksi k   Pole ksi k

### 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 strukturowa) `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 (pól) struktury `samochod1` uzyskuje się za pomocą wskaźnika `wSamochod1` (w którym przechowywany jest jej adres) oraz operatora strzałkowego `->`.

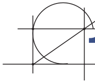
Oprócz tego w programie utworzono strukturę, dla której pamięć została przydzielona dynamicznie na stercie. Wspomniana struktura również jest obiektem będącym instancją struktury `Samochod`. Dostęp do tej struktury uzyskuje się za pomocą wskaźnika `wSamochod2`.

### 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 (pól) zmiennych strukturowych — obiektów — zrealizuj za pomocą wskaźników.



Kup ksi k Pole ksi k



# 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 instancyjne 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.



Kup ksi k Pole ksi k

5. Napisz program pozwalający zapamiętać w zmiennych członkowskich obiektu `uczen` należącego do klasy `Uczen` 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 `Uczen`.
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 strukturowego, 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.



Kup ksi k Pole ksi k

# 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 kompilacji, 104
  w czasie wykonywania programu, 115
Apache NetBeans, 34
argumenty, 198
arytmetyka wskaźnika, 133
automatyczne skalowanie, 133

## B

biblioteka
  cctype, 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
błędy, 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-struktury, 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 strukturowe, 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
  struktury, 172
  szablonów funkcji, 413
  typów danych, 247
  unii, 185
deklaracja
  C-napisu, 148
  funkcji, 194
  łańcucha, 158
  struktury, 172
  tablicy, 122, 129
  zmiennej, 41
    globalnej, 224
    obiektowej, 278
    strukturowej, 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
  kompilacji warunkowej, 251
  preprocesora, 69
dziedziczenie, 351, 362
  chronione, 353
  hierarchiczne, 359




Kup ksi k

Pole ksi k

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 zmienne lokalne, 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
kopiująca, 46
klamrowa, 46
konstruktorowa, 46, 324
listowa
bezpośrednia, 321
kopiująca, 321
wartościami
domyślnymi, 315
zadanymi, 320
zerowa, 46
inicjalizator wewnątrzklasowy, 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
kopiujące, 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



Kup ksi k

Pole ksi k

## Ł

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
statyczne, 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
wewnątrzklasowy, 315
konstruktor domyślny, 315
konstruktor kopiujący, 328
konstruktor parametryczny, 315
konstruktorowa, 324
lista inicjalizacyjna, 320, 326
listowa bezpośrednia, 321
listowa kopiująca, 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
dereferencji *, 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 rozwiązania 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 kopiującego, 328
przeciążanie
funkcji, 233, 428
konstruktorów, 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



Kup ksi k
Pole ksi k

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
obiektowe, 279
deklaracja, 278
przesłanianie, 231
statyczne, 124, 224
sterujące, 87
strukturowe, 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



Kup ksi k    Pole ksi k

# 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

Część 2
Programowanie obiektowe

# Kwalifikacja INF.04
## Projektowanie, programowanie i testowanie aplikacji

Podręcznik do nauki zawodu
**technik programista**

Informatycy i programiści należą obecnie do najbardziej poszukiwanych specjalistów. Stąd tytuł, który uzyskuje się po szkole średniej, nie stanowi jedynie świadectwa ukończenia pewnego etapu edukacji. Technik programista to zawód o wymiernej wartości rynkowej. Absolwenci tego kierunku kształcenia nie mają większych problemów ze znalezieniem intratnego i rozwojowego zajęcia, a pracodawcy chętnie inwestują w ich szkolenia, by mogli zdobywać coraz wyższe kwalifikacje. Dotyczą one między innymi programowania zorientowanego obiektowo, w tym umiejętności zastosowania mechanizmów: dziedziczenia, hermetyzacji, polimorfizmu i abstrakcji.

W tej publikacji, będącej podręcznikiem do nauki zawodu *technik programista*, położono szczególny nacisk na zagadnienie programowania obiektowego. Omówiony tu materiał obejmuje wszystkie efekty kształcenia wymienione w podstawie programowej z zakresu nauki zawodu *technik programista*, w dziale *INF.04.4. Programowanie obiektowe* kwalifikacji *INF.04. Projektowanie, programowanie i testowanie aplikacji*. Dotyczy to również kryteriów weryfikacji osiągnięć edukacyjnych ucznia.

Podręcznik składa się z dwudziestu rozdziałów, które można umownie podzielić na dwie części. Pierwsza, obejmująca rozdziały 1 – 10, zawiera wprowadzenie do programowania i środowiska programistycznego, a także podstawy algorytmiki i opis najważniejszych elementów języka C++. W części drugiej, w rozdziałach 11 – 20, omówiono zagadnienia dotyczące programowania zorientowanego obiektowo.

*Podręcznik do nauki zawodu technik programista* to charakteryzujący się wysoką jakością kompletny zestaw edukacyjny przygotowany przez dysponującego ogromnym doświadczeniem lidera na rynku książek informatycznych — wydawnictwo Helion.

**W skład zestawu podręczników do kwalifikacji INF.04 wchodzą także:**

- *Kwalifikacja INF.04. Projektowanie, programowanie i testowanie aplikacji. Część 1. Inżynieria programowania — projektowanie oprogramowania, testowanie i dokumentowanie aplikacji. Podręcznik do nauki zawodu technik programista*
- *Kwalifikacja INF.04. Projektowanie, programowanie i testowanie aplikacji. Część 3. Aplikacje webowe. Podręcznik do nauki zawodu technik programista*
- *Kwalifikacja INF.04. Projektowanie, programowanie i testowanie aplikacji. Część 4. Aplikacje mobilne. Podręcznik do nauki zawodu technik programista*
- *Kwalifikacja INF.04. Projektowanie, programowanie i testowanie aplikacji. Część 5. Aplikacje desktopowe. Podręcznik do nauki zawodu technik programista*

Podręczniki należące do tej serii przygotowano z myślą o wykształceniu kompetentnych techników, którzy bez trudu poradzą sobie z wyzwaniami, jakie stawia przed nimi współczesna informatyka. Wiedza zawarta w serii pomoże zdać egzamin zawodowy i uzyskać umiejętności praktyczne, przydatne w przyszłej pracy.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!

WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-8101-8

INFORMATYKA W NAJLEPSZYM WYDANIU