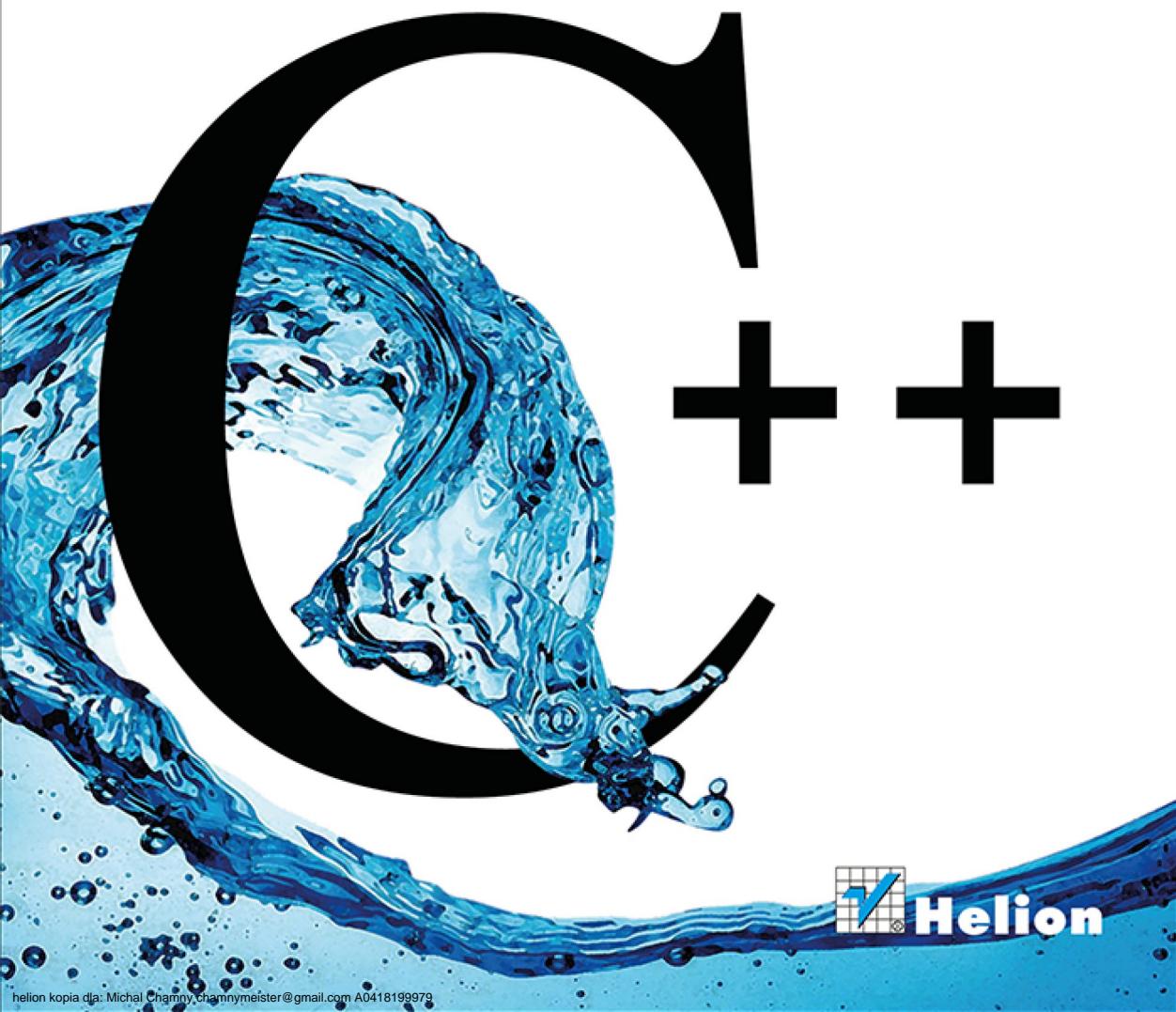


# ALEX ALLAIN

JUŻ DZIŚ NAUCZ SIĘ PROGRAMOWAĆ!

## PRZEWODNIK DLA POCZĄTKUJĄCYCH



 Helion

Tytuł oryginału: Jumping into C++

Tłumaczenie: Ireneusz Jakóbik

ISBN: 978-83-246-8923-1

Jumping into C++. Copyright © 2012 by F. Alexander Allain

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the author except for the use of brief quotations in a book review.

Polish edition copyright © 2014 by Helion S.A.

All rights reserved.

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

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

Autor oraz Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres  
[http://helion.pl/user/opinie/cpppo\\_ebook](http://helion.pl/user/opinie/cpppo_ebook)

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

# Spis treści

---

<b>Część I. Wskocz w C++ .....</b>	<b>13</b>
Podziękowania .....	14

<b>Rozdział 1. Wprowadzenie. Konfiguracja środowiska programistycznego .....</b>	<b>15</b>
Czym jest język programowania? .....	15
Słyszałem o języku, który nazywa się C. Jaka jest różnica między nim a C++? .....	15
Czy powiniem znać C, aby nauczyć się C++? .....	16
Czy żeby zostać programistą, muszę znać matematykę? .....	16
Terminologia .....	16
Programowanie .....	16
Plik wykonywalny .....	16
Edycja i kompilowanie plików źródłowych .....	17
Uwaga na temat przykładowych kodów źródłowych .....	17
Windows .....	17
Krok 1. Pobierz Code::Blocks .....	18
Krok 2. Zainstaluj Code::Blocks .....	18
Krok 3. Uruchom Code::Blocks .....	18
Rozwiązywanie problemów .....	21
Czym właściwie jest Code::Blocks? .....	23
Macintosh .....	23
Xcode .....	23
Instalowanie Xcode .....	24
Uruchamianie Xcode .....	24
Tworzenie pierwszego programu C++ w Xcode .....	24
Instalowanie Xcode 4 .....	28
Uruchamianie Xcode .....	28
Tworzenie pierwszego programu C++ w Xcode .....	28
Rozwiązywanie problemów .....	31
Linux .....	33
Krok 1. Instalowanie g++ .....	33
Krok 2. Uruchomienie g++ .....	34
Krok 3. Uruchomienie programu .....	34
Krok 4. Konfigurowanie edytora tekstopowego .....	35
Konfigurowanie nano .....	35
Korzystanie z nano .....	36

<b>Rozdział 2. Podstawy C++ .....</b>	<b>39</b>
Wprowadzenie do języka C++ .....	39
Najprostszy program w C++ .....	39
Co się dzieje, jeżeli nie możesz zobaczyć swojego programu? .....	41
Podstawowa struktura programu w C++ .....	42
Komentowanie programów .....	42
Specyfika myślenia programisty. Tworzenie kodu wielokrotnego użycia .....	43
Kilka słów na temat radości i bólu praktyki .....	44
Sprawdź się .....	44
Zadania praktyczne .....	45
<b>Rozdział 3. Interakcja z użytkownikiem. Zapisywanie informacji w zmiennych .....</b>	<b>47</b>
Deklarowanie zmiennych w C++ .....	47
Korzystanie ze zmiennych .....	48
Co zrobić, gdy program błyskawicznie kończy działanie? .....	48
Zmiana wartości zmiennych oraz ich porównywanie .....	49
Skrócone zapisy na dodawanie i odejmowanie jedynki .....	50
Poprawne i niepoprawne użycie zmiennych .....	51
Najczęściej popełniane błędy podczas deklarowania zmiennych w C++ .....	51
Rozróżnianie wielkości liter .....	52
Nazwy zmiennych .....	53
Przechowywaniełańcuchów tekstowych .....	53
No dobrze, rozumiem jużłańcuchy tekstowe, ale co z pozostałymi typami? .....	55
Mały sekret liczb zmiennoprzecinkowych .....	56
Mały sekret liczb całkowitych .....	57
Sprawdź się .....	57
Zadania praktyczne .....	58
<b>Rozdział 4. Instrukcje warunkowe .....</b>	<b>59</b>
Podstawowa składnia instrukcji if .....	59
Wyrażenia .....	60
Czym jest prawda? .....	61
Typ bool .....	61
Instrukcja else .....	62
Instrukcje else-if .....	62
Porównywaniełańcuchów tekstowych .....	63
Więcej interesujących warunków budowanych za pomocą operatorów logicznych .....	64
Logiczne nie .....	64
Logiczne ORAZ .....	65
Logiczne LUB .....	65
Łączenie wyrażeń .....	66
Przykładowe wyrażenia logiczne .....	67
Sprawdź się .....	67
Zadania praktyczne .....	68

---

<b>Rozdział 5. Pętle .....</b>	<b>69</b>
Pętla while .....	69
Najczęściej popełniany błąd .....	70
Pętla for .....	70
Inicjalizacja zmiennej .....	71
Warunek pętli .....	71
Aktualizacja zmiennej .....	71
Pętla do-while .....	72
Kontrolowanie przebiegu pętli .....	73
Pętle zagnieździone .....	75
Wybór właściwego rodzaju pętli .....	76
Pętla for .....	76
Pętla while .....	76
Pętla do-while .....	77
Sprawdź się .....	78
Zadania praktyczne .....	78
<b>Rozdział 6. Funkcje .....</b>	<b>81</b>
Składnia funkcji .....	81
Zmienne lokalne i zmienne globalne .....	83
Zmienne lokalne .....	83
Zmienne globalne .....	84
Ostrzeżenie dotyczące zmiennych globalnych .....	85
Przygotowanie funkcji do użycia .....	86
Definicja i deklaracja funkcji .....	86
Przykład użycia prototypu funkcji .....	87
Rozbijanie programu na funkcje .....	88
Kiedy wciąż na nowo powtarzasz ten sam kod .....	88
Kiedy chcesz, żeby kod był łatwiejszy do czytania .....	88
Nazwywanie i przeładowywanie funkcji .....	89
Podsumowanie wiadomości o funkcjach .....	89
Sprawdź się .....	90
Zadania praktyczne .....	90
<b>Rozdział 7. Instrukcje switch case oraz typ wyliczeniowy .....</b>	<b>91</b>
Porównanie instrukcji switch case z if .....	93
Tworzenie prostych typów za pomocą wyliczeń .....	94
Sprawdź się .....	96
Zadania praktyczne .....	96
<b>Rozdział 8. Dodawanie do programu elementu losowości .....</b>	<b>97</b>
Uzyskiwanie liczb losowych w C++ .....	98
Błędy i losowość .....	100
Sprawdź się .....	101
Zadania praktyczne .....	102

<b>Rozdział 9. Co zrobić, kiedy... nie wiesz, co robić? .....</b>	<b>103</b>
Krótka dygresja na temat wydajności i bezpieczeństwa kodu .....	106
Co robić, kiedy nie znasz algorytmu? .....	107
Zadania praktyczne .....	110
 <b>Część II. Praca z danymi .....</b>	 <b>111</b>
 <b>Rozdział 10. Tablice .....</b>	 <b>113</b>
Podstawowa składnia tablic .....	113
Przykładowe zastosowania tablic .....	114
Przechowywanie zamówień w tablicach .....	114
Odwzorowanie siatek w tablicach wielowymiarowych .....	115
Korzystanie z tablic .....	115
Tablice i pętle .....	115
Przekazywanie tablic do funkcji .....	116
Wypadnięcie poza ostatni element tablicy .....	118
Sortowanie tablic .....	118
Sprawdź się .....	122
Zadania praktyczne .....	123
 <b>Rozdział 11. Struktury .....</b>	 <b>125</b>
Wiązanie wielu wartości .....	125
Składnia .....	125
Przekazywanie struktur .....	127
Sprawdź się .....	129
Zadania praktyczne .....	130
 <b>Rozdział 12. Wprowadzenie do wskaźników .....</b>	 <b>131</b>
Zapomnij o wszystkim, co do tej pory słyszałeś .....	131
No dobrze, czym są wskaźniki? Dlaczego powinny mnie obchodzić? .....	131
Czym jest pamięć komputera? .....	132
Zmienne a adresy .....	133
Uwaga na temat nazewnictwa .....	133
Organizacja pamięci .....	134
Nieprawidłowe wskaźniki .....	135
Pamięć i tablice .....	136
Pozostałe zalety i wady wskaźników .....	136
Sprawdź się .....	137
Zadania praktyczne .....	137
 <b>Rozdział 13. Korzystanie ze wskaźników .....</b>	 <b>139</b>
Składnia wskaźników .....	139
Deklarowanie wskaźnika .....	139

---

Otrzymywanie adresu zmiennej za pomocą wskaźnika .....	140
Użycie wskaźnika .....	140
Niezainicjalizowane wskaźniki i wartość NULL .....	143
Wskaźniki i funkcje .....	144
Referencje .....	146
Referencje a wskaźniki .....	147
Sprawdź się .....	148
Zadania praktyczne .....	148
<b>Rozdział 14. Dynamiczna alokacja pamięci .....</b>	<b>151</b>
Pozyskiwanie pamięci za pomocą instrukcji new .....	151
Brak pamięci .....	152
Referencje i dynamiczna alokacja .....	152
Wskaźniki i tablice .....	152
Tablice wielowymiarowe .....	155
Arytmetyka wskaźników .....	155
Zrozumieć tablice dwuwymiarowe .....	156
Wskaźniki do wskaźników .....	157
Wskaźniki do wskaźników i tablic dwuwymiarowych .....	159
Oswajanie wskaźników .....	160
Sprawdź się .....	160
Zadania praktyczne .....	161
<b>Rozdział 15. Wprowadzenie do struktur danych: listy powiązane .....</b>	<b>163</b>
Wskaźniki i struktury .....	165
Tworzenie listy powiązanej .....	166
Pierwszy przebieg .....	167
Drugi przebieg .....	168
Przeglądanie listy powiązanej .....	169
Oswajanie list powiązanych .....	170
Tablice a listy powiązane .....	171
Sprawdź się .....	173
Zadania praktyczne .....	175
<b>Rozdział 16. Rekurencja .....</b>	<b>177</b>
Jak postrzegać rekurencję? .....	177
Rekurencja i struktury danych .....	179
Pętle i rekurencja .....	181
Stos .....	183
Zaleta stosu .....	185
Wady rekurencji .....	185
Debugowanie przepelnienia stosu .....	186
Wydajność .....	188
Oswajanie rekurencji .....	188
Sprawdź się .....	188
Zadania praktyczne .....	189

<b>Rozdział 17. Drzewa binarne .....</b>	<b>191</b>
Konwencje nazewnicze .....	193
Implementacja drzew binarnych .....	194
Wstawianie węzła do drzewa .....	194
Przeszukiwanie drzewa .....	196
Niszczanie drzewa .....	197
Usuwanie węzła z drzewa .....	199
Praktyczne zastosowanie drzew binarnych .....	206
Koszt tworzenia drzew i map .....	207
Sprawdź się .....	208
Zadania praktyczne .....	208
<b>Rozdział 18. Standardowa biblioteka szablonów .....</b>	<b>211</b>
Wektor — tablica o zmiennych rozmiarach .....	212
Przekazywanie wektorów do metod .....	213
Inne właściwości wektorów .....	213
Mapy .....	214
Iteratory .....	215
Sprawdzanie, czy wartość znajduje się w mapie .....	217
Oswajanie biblioteki STL .....	218
Więcej informacji o STL .....	219
Sprawdź się .....	219
Zadania praktyczne .....	220
<b>Rozdział 19. Więcej o łańcuchach tekstowych .....</b>	<b>221</b>
Wczytywanie łańcuchów tekstowych .....	221
Długość łańcucha i dostęp do jego elementów .....	223
Wyszukiwanie i podłańcuchy .....	224
Przekazywanie łańcucha przez referencję .....	225
Szerzenie się const .....	226
Const i STL .....	227
Sprawdź się .....	228
Zadania praktyczne .....	229
<b>Rozdział 20. Debugowanie w Code::Blocks .....</b>	<b>231</b>
Zaczynamy .....	232
Wstrzymywanie działania programu .....	233
Debugowanie awarii .....	239
Zaglądanie do zawieszonego programu .....	242
Modyfikowanie zmiennych .....	245
Podsumowanie .....	246
Zadania praktyczne .....	246
Zadanie nr 1. Problem z wykładnikiem .....	246
Zadanie nr 2. Problem z dodawaniem liczb .....	247
Zadanie nr 3. Problem z ciągiem Fibonacciego .....	247
Zadanie nr 4. Problem z odczytywaniem i wyświetlaniem listy .....	248

---

<b>Część III. Tworzenie większych programów .....</b>	<b>249</b>
<b>Rozdział 21. Rozbijanie programów na mniejsze części .....</b>	<b>251</b>
Proces komplikacji w języku C++ .....	251
Przetwarzanie wstępne .....	252
Kompilacja .....	253
Konsolidacja .....	253
Dlaczego komplikacja i konsolidacja przebiegają oddzielnie? .....	254
Jak rozbić program na wiele plików? .....	254
Krok 1. Oddzielanie deklaracji od definicji .....	255
Krok 2. Określenie, które funkcje powinny być wspólne .....	255
Krok 3. Przeniesienie wspólnych funkcji do nowych plików .....	255
Przykładowy program .....	256
Pozostałe zasady pracy z plikami nagłówkowymi .....	259
Praca z wieloma plikami źródłowymi w środowisku programistycznym .....	260
Sprawdź się .....	262
Zadania praktyczne .....	264
<b>Rozdział 22. Wprowadzenie do projektowania programów .....</b>	<b>265</b>
Powielony kod .....	265
Założenia dotyczące przechowywania danych .....	266
Projekt i komentarze .....	268
Sprawdź się .....	269
<b>Rozdział 23. Ukrywanie reprezentacji struktur danych .....</b>	<b>271</b>
Użycie funkcji w celu ukrycia układu struktury .....	272
Deklaracja metody i składnia wywołania .....	273
Przeniesienie definicji funkcji poza strukturę .....	274
Sprawdź się .....	275
Zadania praktyczne .....	275
<b>Rozdział 24. Klasa .....</b>	<b>277</b>
Ukrywanie sposobu przechowywania danych .....	278
Deklarowanie instancji klasy .....	279
Odpowiedzialności klasy .....	280
Co tak naprawdę znaczy private? .....	281
Podsumowanie .....	281
Sprawdź się .....	281
Zadania praktyczne .....	282
<b>Rozdział 25. Cykl życia klasy .....</b>	<b>283</b>
Konstruowanie obiektu .....	283
Co się stanie, jeśli nie utworzysz konstruktora? .....	285
Inicjalizacja składowych klasy .....	286
Użycie listy inicjalizacyjnej do pól stałych .....	287

Niszczanie obiektu .....	288
Niszczanie podczas usuwania .....	290
Niszczanie przy wyjściu poza zakres .....	290
Niszczanie przez inny destruktör .....	291
Kopiowanie klas .....	291
Operator przypisania .....	292
Konstruktor kopiący .....	295
Pełna lista metod generowanych przez kompilator .....	296
Całkowite zapobieganie kopiowaniu .....	296
Sprawdź się .....	297
Zadania praktyczne .....	298
<b>Rozdział 26. Dziedziczenie i polimorfizm .....</b>	<b>299</b>
Dziedziczenie w C++ .....	300
Pozostałe zastosowania oraz nieprawidłowe użycia dziedziczenia .....	304
Dziedziczenie, konstruowanie obiektów oraz ich niszczanie .....	304
Polimorfizm i dziedziczenie obiektów .....	306
Problem przycinania .....	308
Dzielenie kodu z podklasami .....	309
Dane chronione .....	309
Dane obejmujące całą klasę .....	310
W jaki sposób zaimplementowany jest polimorfizm? .....	311
Sprawdź się .....	313
Zadania praktyczne .....	314
<b>Rozdział 27. Przestrzenie nazw .....</b>	<b>317</b>
Kiedy stosować instrukcję using namespace .....	319
Kiedy należy utworzyć przestrzeń nazw? .....	319
Sprawdź się .....	320
Zadania praktyczne .....	320
<b>Rozdział 28. Plikowe operacje wejścia-wyjścia .....</b>	<b>321</b>
Podstawy plikowych operacji wejścia-wyjścia .....	321
Czytanie z plików .....	321
Formaty plików .....	323
Koniec pliku .....	324
Zapisywanie plików .....	326
Tworzenie nowych plików .....	327
Pozycja pliku .....	327
Pobieranie argumentów z wiersza poleceń .....	330
Obsługa argumentów liczbowych .....	332
Pliki binarne .....	332
Praca z plikami binarnymi .....	334
Konwersja na typ char* .....	335
Przykład binarnych operacji we/wy .....	335

---

Przechowywanie klas w pliku .....	336
Czytanie z pliku .....	338
Sprawdź się .....	340
Zadania praktyczne .....	341
<b>Rozdział 29. Szablony w C++ .....</b>	<b>343</b>
Szablony funkcji .....	343
Inferencja typów .....	345
Kacze typowanie .....	345
Szablony klas .....	346
Wskazówki dotyczące pracy z szablonami .....	347
Szablony i pliki nagłówkowe .....	349
Podsumowanie informacji o szablonach .....	349
Interpretacja komunikatów o błędach w szablonach .....	350
Sprawdź się .....	353
Zadania praktyczne .....	354
<b>Części IV. Zagadnienia rozmaite .....</b>	<b>355</b>
<b>Rozdział 30. Formatowanie danych wyjściowych za pomocą iomanip .....</b>	<b>357</b>
Rozwiązywanie problemów związanych z odstępami .....	357
Określanie szerokości pola za pomocą instrukcji setw .....	357
Zmiana znaku dopełniającego .....	358
Trwała zmiana ustawień .....	358
Korzystanie ze znajomości iomanip .....	359
Wyświetlanie liczb .....	360
Określanie precyzyji wyświetlanych liczb za pomocą instrukcji setprecision .....	361
A co z pieniędzmi? .....	361
Wyświetlanie liczb o różnych podstawkach .....	362
<b>Rozdział 31. Wyjątki i raportowanie błędów .....</b>	<b>363</b>
Zwalnianie zasobów po wystąpieniu wyjątku .....	364
Ręczne czyszczenie zasobów w bloku catch .....	365
Zgłaszanie wyjątków .....	366
Specyfikacja wyjątków .....	367
Korzyści płynące z wyjątków .....	368
Nieprawidłowe użycie wyjątków .....	369
Podsumowanie informacji o wyjątkach .....	370
<b>Rozdział 32. Końcowe przemyślenia .....</b>	<b>371</b>
Rozwiązywanie testu z rozdziału 2 .....	372
Rozwiązywanie testu z rozdziału 3 .....	373
Rozwiązywanie testu z rozdziału 4 .....	374

Rozwiązywanie testu z rozdziału 5 .....	374
Rozwiązywanie testu z rozdziału 6 .....	375
Rozwiązywanie testu z rozdziału 7 .....	375
Rozwiązywanie testu z rozdziału 8 .....	376
Rozwiązywanie testu z rozdziału 10. ....	377
Rozwiązywanie testu z rozdziału 11. ....	377
Rozwiązywanie testu z rozdziału 12. ....	378
Rozwiązywanie testu z rozdziału 13. ....	379
Rozwiązywanie testu z rozdziału 14. ....	380
Rozwiązywanie testu z rozdziału 15. ....	381
Rozwiązywanie testu z rozdziału 16. ....	382
Rozwiązywanie testu z rozdziału 17. ....	383
Rozwiązywanie testu z rozdziału 18. ....	384
Rozwiązywanie testu z rozdziału 19. ....	385
Rozwiązywanie testu z rozdziału 21. ....	385
Rozwiązywanie testu z rozdziału 22. ....	386
Rozwiązywanie testu z rozdziału 23. ....	386
Rozwiązywanie testu z rozdziału 24. ....	387
Rozwiązywanie testu z rozdziału 25. ....	387
Rozwiązywanie testu z rozdziału 26. ....	389
Rozwiązywanie testu z rozdziału 27. ....	390
Rozwiązywanie testu z rozdziału 28. ....	391
Rozwiązywanie testu z rozdziału 29. ....	391
<b>Skorowidz .....</b>	<b>393</b>

# I

## CZĘŚĆ I

### Wskocz w C++

---

Przygotuj się do programowania! Programowanie, tak jak inne formy sztuki, pozwala Ci tworzyć, ale w tym przypadku Twój potencjał jest zwielokrotniony przez prędkość i możliwości komputera. Możesz tworzyć pasjonujące gry, takie jak World of Warcraft, Bioshock, Gears of War albo Mass Effect. Możesz pisać szczegółowe i wciągające symulacje w rodzaju The Sims. Możesz tworzyć programy łączące ludzi: przeglądarki takie jak Chrome, edytory wiadomości e-mail, klienty czat albo witryny podobne do Facebooka lub Amazon.com. Możesz tworzyć aplikacje wykorzystujące zalety nowych urządzeń, takich jak iPhone albo telefony z Androidem. Rzecz jasna, aby zdobyć umiejętności niezbędne do tworzenia takich programów, potrzebny jest czas. Niemniej nawet na samym początku przygody z C++ możesz pisać interesujące aplikacje — programy pomagające rozwiązywać zadania domowe z matematyki, proste gry w stylu Tetrisa, które możesz pokazać znajomym, narzędzia automatyzujące żmudne czynności lub złożone obliczenia, które przeprowadzane ręcznie trwałyby dni albo tygodnie. Kiedy już zrozumiesz podstawy programowania komputerów — czego nauczy Cię ta książka — zdobędziesz umiejętności potrzebne do wybrania bibliotek graficznych lub sieciowych niezbędnych do pisania programów, które Cię interesują — gier, symulacji naukowych albo aplikacji łączących te gatunki.

C++ jest wydajnym językiem programowania, który zagwarantuje Ci solidne podstawy do korzystania z nowoczesnych technik programistycznych. Istotnie, C++ dzieli swoje koncepcje z wieloma innymi językami, dzięki czemu większość wiedzy, jaką zdobędziesz, będzie mieć zastosowanie także w odniesieniu do innych języków, które poznasz później (trudno znaleźć programistę piszącego wyłącznie w jednym języku).

Programiści C++ dysponują elastycznym zbiorem umiejętności, które pozwalają im pracować nad wieloma różnymi projektami. Większość aplikacji oraz programów, z których korzystasz każdego dnia, jest napisana w C++. To niewiarygodne, ale wszystkie aplikacje, które wymieniłem powyżej, zostały w całości utworzone w C++ lub w języku tym została zaprogramowana ogromna część ich ważnych komponentów<sup>1</sup>.

Prawdę mówiąc, zainteresowanie C++ nadal rośnie — nawet teraz, gdy nowe języki programowania, takie jak Java i C#, zdobywają popularność. W ciągu ostatnich kilku lat zauważylem znaczny wzrost ruchu w mojej witrynie *Cprogramming.com*. C++ nadal pozostaje językiem wyboru w przypadku wysokowydajnych aplikacji. Tworzone w nim programy działają bardzo szybko, często szybciej niż gdyby były napisane w Javie lub innych podobnych językach. C++ przez cały czas się rozwija, a jego nowa specyfikacja, C++11, uwzględnia funkcje ułatwiające

---

<sup>1</sup> Aplikacje te, a także wiele innych zastosowań C++, znajdziesz pod adresem <http://www.stroustrup.com/applications.html>.

i przyspieszające pracę programistów, zachowując jednocześnie jego wysokowydajne korzenie<sup>2</sup>. Dobra orientacja w C++ jest także zaletą na rynku pracy, a stanowiska wymagające znajomości tego języka stawiają przed programistami ciekawe wyzwania oraz są dobrze płatne.

Czy jesteś już gotów przystąpić do nauki? Część I traktuje o tym, jak przygotować się do pisania programów i jak używać podstawowych cegiełek C++. Kiedy już się z nią uporasz, będziesz w stanie pisać prawdziwe programy, które będą mogli zobaczyć Twoi znajomi (w każdym razie bliscy i życzliwi znajomi), a także nauczysz się myśleć jak programista. Nie staniesz się od razu mistrzem C++, chociaż będziesz dobrze przygotowany do poznania pozostałych funkcji tego języka, które będą Ci potrzebne do tworzenia naprawdę przydatnych i wydajnych programów.

Przekażę Ci tylko tyle podstawowej wiedzy oraz terminologii, abyś się nie pogubił, a bardziej skomplikowane wyjaśnienia pewnych zagadnień odłożę na później — gdy już opanujesz podstawy.

Pozostałe części tej książki będą Cię wprowadzać w coraz bardziej zaawansowane koncepcje. Dowiesz się, jak pisać programy pracujące z dużymi ilościami danych — w tym wczytujące informacje z plików — a także nauczysz się, jak łatwo i wydajnie te dane przetwarzać (przy okazji poznasz liczne sposoby na uproszczenie swojej pracy). Dowiesz się, jak pisać większe i bardziej skomplikowane programy, nie gubiąc się przy tym w gąszczu ich złożoności. Poznasz także narzędzia stosowane przez profesjonalnych programistów.

Pod koniec lektury tej książki powinieneś umieć czytać i pisać prawdziwe programy komputerowe, które robią przydatne i ciekawe rzeczy. Jeżeli interesuje Cię programowanie gier, będziesz gotowy do podjęcia wyzwań typowych dla pisania gier. Jeśli uczęszczasz na kurs C++ (albo przygotowujesz się do niego), powinieneś dysponować wiedzą, która pozwoli Ci rozwijać skrzydła. Jeżeli jesteś samoukiem, zdobędziesz wystarczająco dużo informacji do napisania prawie każdego programu, jaki tylko Cię zainteresuje, mając jednocześnie pod ręką niemal wszystkie narzędzia, jakie oferuje C++.

## Podziękowania

Pragnę podziękować Aleksandrze Hoffer za jej staranną i cierpliwaną edycję tekstu oraz szczegółowe sugestie, których udzielała mi w trakcie tworzenia tej książki. Bez jej wysiłku książka ta nie powstałyby. Chciałbym także podziękować Andreiowi Cheremskoyowi, Minyang Jiang i Johannesowi Peterowi za ich nieocenione opinie, sugestie oraz poprawki.

---

<sup>2</sup> Specyfikacja ta została zatwierdzona, gdy niniejsza książka była prawie na ukończeniu, w związku z czym nie zamieściłem w niej żadnych materiałów dotyczących nowego standardu. Cykl artykułów przedstawiających C++11 znajdziesz pod adresem <http://www.cprogramming.com/c++11/what-is-c++0x.html>.

# 1

## ■ ■ ■ R O Z D Z I A Ł 1

# Wprowadzenie. Konfiguracja środowiska programistycznego

---

## Czym jest język programowania?

Jeśli chcesz sprawować kontrolę nad swoim komputerem, musisz się z nim w jakiś sposób komunikować. W przeciwnieństwie do zwierząt, np. psa albo kota, które mają swoje zagadkowe języki, komputery posługują się językami programowania utworzonymi przez ludzi. Program komputerowy, podobnie jak książka albo wypracowanie, składa się z tekstu, z tym że ma on swoją szczególną strukturę. Język takiego programu, pozostając zrozumiałym dla ludzi, jest o wiele ściślej ustrukturyzowany niż zwykły język, a jego zasób leksykalny jest o wiele skromniejszy. Jednym z takich języków jest C++, i przy tym jest to język bardzo popularny.

Kiedy już napiszesz program komputerowy, potrzebny Ci będzie jakiś sposób, aby zaczął on działać w komputerze — sposób na wykonanie tego, co napisałeś. Zazwyczaj czynność ta jest nazywana uruchomieniem programu. Sposób, w jaki to zrobisz, zależy od języka programowania oraz środowiska programistycznego. Już wkrótce napiszę więcej o tym, jak uruchamiać programy.

Istnieje wiele języków programowania, z których każdy ma swoją strukturę oraz słownictwo, ale są one pod wieloma względami bardzo podobne. Kiedy już poznasz jeden z nich, nauka innego będzie łatwiejsza.

## Słyszałem o języku, który nazywa się C. Jaka jest różnica między nim a C++?

C jest językiem opracowanym pierwotnie w celu programowania w systemie operacyjnym Unix. Jest to język niskopoziomowy i wydajny, ale brakuje mu wielu nowoczesnych i przydatnych konstrukcji. C++ jest językiem nowszym, bazującym na C i uzupełnionym o liczne właściwości nowoczesnych języków programowania, które sprawiają, że programowanie w nim jest łatwiejsze niż w C.

C++ zachowuje pełnię możliwości języka C, jednocześnie oddając do dyspozycji programistów nowe funkcje, które upraszczają pisanie użytecznych i wyrafinowanych programów.

C++ ułatwia na przykład zarządzanie pamięcią komputera oraz udostępnia kilka funkcji umożliwiających programowanie „zorientowane obiektywne” oraz programowanie „generyczne”.

Później wyjaśnię, co to tak naprawdę znaczy. Na razie zapamiętaj, że C++ pozwala programistom nie myśleć o technicznych szczegółach działania komputera i skupić się na problemach, które starają się rozwiązać.

Jeśli zastanawiasz się nad wyborem między nauką C a C++, zdecydowanie polecam C++.

## Czy powiniem znać C, aby nauczyć się C++?

Nie. C++ jest nadzbiorem języka C. Wszystko, co możesz zrobić w C, możesz zrobić także w C++. Jeśli znasz już C, szybko przyzwyczaisz się do zorientowanych obiektywnie cech C++. Jeśli nie znasz C, wcale nie musisz się tym przejmować. Tak naprawdę nie ma żadnych korzyści w poznawaniu C przed C++, a poza tym od razu będziesz mógł skorzystać z przewagi wydajnych funkcji typowych wyłącznie dla C++ (z których pierwszą spośród wielu są łatwiejsze operacje wejścia i wyjścia).

## Czy żeby zostać programistą, muszę znać matematykę?

Gdybym dostawał pięciocentówkę za każdym razem, gdy ktoś zadaje mi to pytanie, potrzebowałbym kalkulatora, aby policzyć moją małą fortunę. Na szczęście odpowiedź brzmi: zdecydowanie nie! Większość programowania sprowadza się do projektowania oraz logicznego myślenia i nie wymaga umiejętności szybkiego wykonywania obliczeń ani głębokiej znajomości algebry albo analizy matematycznej. Programowanie i matematyka zachodzą na siebie zwłaszcza w zakresie logicznego myślenia oraz precyzyjnego rozumowania. Zdolności matematyczne będą Ci potrzebne tylko wtedy, gdy zechcesz zabrać się za programowanie zaawansowanych trójwymiarowych silników graficznych (<http://www.cprogramming.com/tutorial.html#3dtutorial>), pisać programy wykonujące analizy statystyczne lub zająć się innego rodzaju wyspecjalizowanym programowaniem numerycznym.

## Terminologia

Nowe terminy będę definiować w dalszych częściach tej książki, ale zaczniemy od kilku podstawowych pojęć, które będą potrzebne już na starcie.

## Programowanie

Programowanie to proces pisania instrukcji w taki sposób, który umożliwi komputerowi ich zrozumienie i wykonanie. Same instrukcje nazywane są **kodem źródłowym**. To właśnie taki kod będziesz pisać. Po raz pierwszy kod źródłowy zobaczysz w tej książce już kilka stron dalej.

## Plik wykonywalny

Końcowym wynikiem programowania jest plik **wykonywalny**. Jest to taki plik, który może być uruchomiony w Twoim komputerze. Jeśli pracujesz w systemie Windows, możesz znać te pliki pod nazwą „egzeków” (od rozszerzenia „exe”). Program taki jak Microsoft Word jest tego typu plikiem. Niektóre aplikacje zawierają dodatkowe pliki (graficzne, muzyczne itd.), ale każdy

program wymaga pliku wykonywalnego. W celu uzyskania pliku wykonywalnego potrzebny jest **kompilator** — program zamieniający plik źródłowy w plik wykonywalny. Bez kompilatora nie byłbyś w stanie zrobić nic innego, jak tylko oglądać swój kod źródłowy. Ponieważ jest to dość nudne zajęcie, następną rzeczą, jaką zrobimy, będzie skonfigurowanie kompilatora.

## Edycja i kompilowanie plików źródłowych

Pozostała część tego rozdziału poświęcona jest konfiguracji prostego, łatwego w użyciu środowiska programistycznego. Skonfigurujesz dwa specyficzne narzędzia, jakimi są kompilator i **edytor**. Wiesz już, do czego jest potrzebny kompilator — po to, aby programy mogły działać. Zastosowanie edytora jest mniej oczywiste, ale równie ważne: edytor umożliwia tworzenie kodu źródłowego we właściwym formacie.

Kod źródłowy musi być napisany **czystym tekstem**. Czysty tekst zawiera wyłącznie treść pliku; nie ma w nim dodatkowych informacji o formatowaniu ani wyświetlaniu jego zawartości. W odróżnieniu od czystego tekstu, plik utworzony za pomocą programu Microsoft Word (lub podobnego) nie jest czysty, ponieważ zawiera informacje o użytych czcionkach, rozmiarze tekstu, a także o jego formatowaniu. Nie zobaczysz tych wszystkich informacji, kiedy otworzysz plik tego typu w Wordzie, niemniej one tam są. Czyste pliki tekstowe zawierają wyłącznie samą treść. Możesz tworzyć je za pomocą narzędzi, które omówię już za chwilę.

Edytor udostępnia również inne przydatne funkcje, takie jak **wyróżnianie składni** oraz **automatyczne wcinanie tekstu**. Wyróżnianie składni oznacza po prostu dodawanie koloru, dzięki czemu można łatwo odróżnić poszczególne elementy programu. Automatyczne wcinanie tekstu pomaga formatować tekst w czytelny sposób.

Jeśli korzystasz z Windows albo Maca, pomożę Ci skonfigurować zaawansowany edytor nazywany **zintegrowanym środowiskiem programistycznym**, który łączy edytor z kompilatorem. Jeśli używasz Linuksa, będziesz mógł korzystać z prostego w użyciu edytora znanego pod nazwą nano. Wyjaśnię Ci wszystko, co jest potrzebne do skonfigurowania edytora i przystąpienia do pracy.

## Uwaga na temat przykładowych kodów źródłowych

Książka ta zawiera sporo przykładowych kodów źródłowych, z których możesz korzystać bez żadnych ograniczeń (ale też bez żadnych gwarancji) we własnych programach. Kody te znajdują się w pliku *przykładowe\_kody.zip*, dołączonym do tej książki. Wszystkie pliki z przykładowymi kodami źródłowymi zostały umieszczone w oddzielnnych folderach o nazwach utworzonych na podstawie rozdziałów, w których dane kody występują (na przykład pliki z tego rozdziału znajdują się w folderze *r1*). Każdy listing kodu źródłowego w tej książce, umieszczony w odpowiadającym mu pliku, został opatrzony podpisem będącym nazwą tego pliku (z pominięciem numeru rozdziału).

## Windows

Skonfigurujemy narzędzie o nazwie **Code:Blocks**, które jest darmowym środowiskiem programistycznym przeznaczonym do pracy w C++.

## Krok 1. Pobierz Code::Blocks

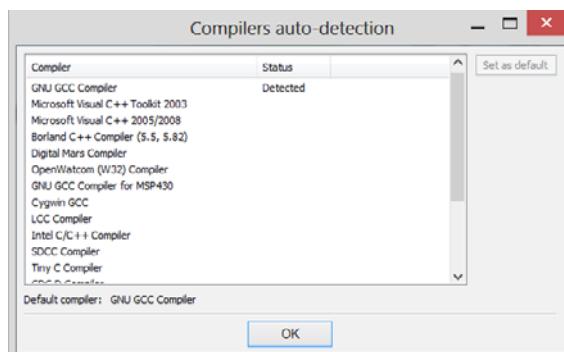
- Przejdź na stronę <http://www.codeblocks.org/downloads>.
- Kliknij łącze *Download the binary release* (<http://www.codeblocks.org/downloads/26>).
- Przejdź do sekcji *Windows 2000/XP/Vista/7/8*.
- Poszukaj pliku, który ma w nazwie *mingw* (w czasie tłumaczenia tej książki dostępny był plik *codeblocks-12.11mingw-setup*; obecna wersja pliku może być inna).
- Zapisz plik na pulpicie. W momencie pisania tych słów liczył on około 97 megabajtów objętości.

## Krok 2. Zainstaluj Code::Blocks

- Dwukrotnie kliknij plik instalatora.
- W kolejnych oknach klikaj przycisk *Next* (lub *I Agree* w oknie z licencją). Instalator zakłada, że program zostanie zainstalowany w katalogu *C:\Program Files (x86)\CodeBlocks* (jest to lokalizacja domyślna), ale możesz wskazać dowolne inne miejsce.
- Przeprowadź pełną instalację (z listy rozwijanej *Select the type of install* wybierz opcję *Full: All plugins, all tool, just everything*).
- Uruchom Code::Blocks.

## Krok 3. Uruchom Code::Blocks

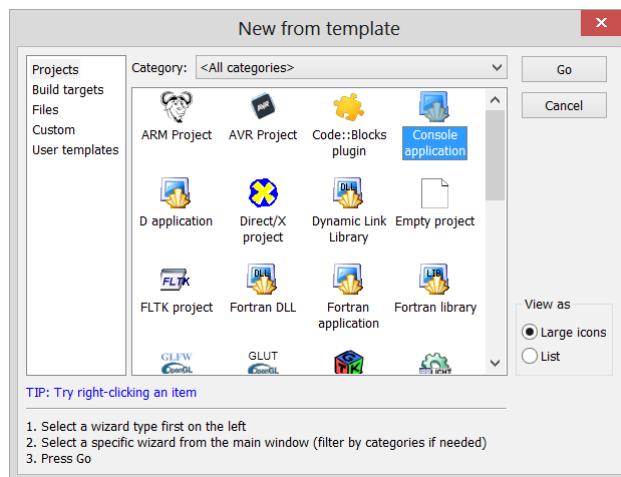
Zostanie wyświetlone okno automatycznego wykrywania kompilatorów.



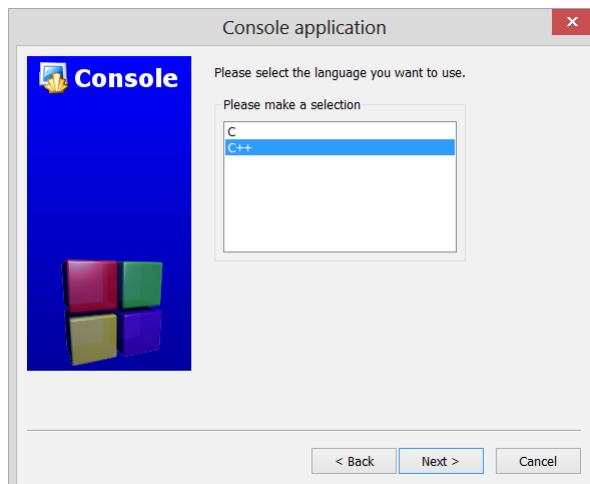
Kliknij w nim przycisk *OK*. Code::Blocks może zapytać Cię, czy chcesz, żeby stał się domyślnym edytorem plików C i C++. Sugeruję, abyś się na to zgodził (w oknie *File associations* wybierz opcję *Yes, associate Code::Blocks with C/C++ file types*). Kliknij menu *File*, po czym wybierz polecenie *New* oraz *Project*.

Zostanie wyświetlone poniższe okno (zobacz pierwszy rysunek na kolejnej stronie).

Wybierz opcję *Console application*, po czym kliknij przycisk *Go*. Wszystkie przykładowe kody zamieszczone w tej książce mogą być uruchamiane jako aplikacje konsolowe.



W dalszej kolejności klikaj przycisk *Next*, aż dojdiesz do okna dialogowego z wyborem języka.



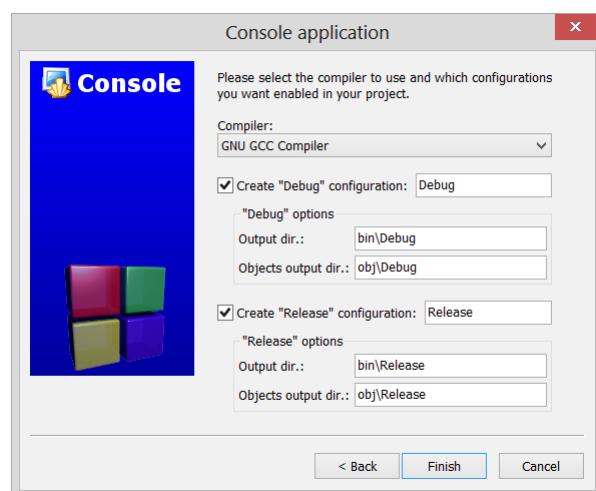
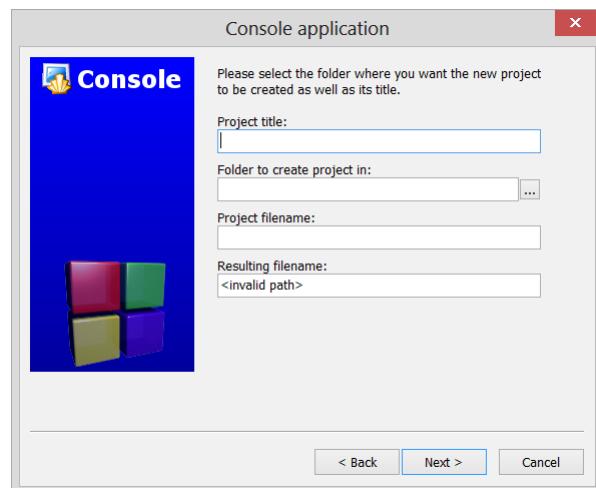
Zostaniesz zapytany, czy chcesz korzystać z C czy C++. Ponieważ będziemy się uczyć C++, wybierz tę drugą opcję.

Po kliknięciu przycisku *Next* Code::Blocks zapyta, gdzie chciałbyś zapisać swoją aplikację konsolową (zobacz pierwszy rysunek na kolejnej stronie).

Sugeruję, abyś umieścił ją w jej własnym folderze, ponieważ może ona wygenerować więcej plików (co ma miejsce zwłaszcza wtedy, gdy tworzysz projekty innych rodzajów). Swój projekt będziesz musiał jeszcze jakoś nazwać — dobra będzie każda opisowa nazwa.

Po kolejnym kliknięciu przycisku *Next* zostaniesz poproszony o skonfigurowanie kompilatora (zobacz drugi rysunek na kolejnej stronie).

W tym oknie nie musisz robić nic. Po prostu zaakceptuj ustawienia domyślne, klikając przycisk *Finish*.



Teraz możesz otworzyć plik *main.cpp*, widoczny po lewej stronie (zobacz pierwszy rysunek na kolejnej stronie).

Jeżeli nie widzisz pliku *main.cpp*, rozwiń zawartość folderu *Sources*.

W tym momencie gotowy jest Twój plik *main.cpp*, który możesz modyfikować, jak tylko zechcesz. Zwróć uwagę na jego rozszerzenie: chociaż pliki te zawierają czysty tekst, standardowym rozszerzeniem plików źródłowych języka C++ jest *.cpp*, a nie *.txt*. Na razie nasz program wyświetla tylko powitanie "Hello world!". Naciśnij klawisz *F9*, który spowoduje skompilowanie kodu, po czym jego uruchomienie (możesz także wybrać z głównego menu *Build* polecenie *Build and run*) (zobacz drugi rysunek na kolejnej stronie).

Masz program, który działa! Teraz możesz po prostu zmienić jego zawartość i znowu nacisnąć *F9*, aby go skompilować i uruchomić.

The screenshot shows the Code::Blocks IDE interface. The main window displays the code for `main.cpp` in the `moja_aplikacja` project:

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     cout << "Hello world!" << endl;
8     return 0;
9 }
10

```

The status bar at the bottom indicates the file is `C:\Aplikacje konsolowe\moja_aplikacja\main.cpp`, encoding is `WINDOWS-1252`, and the current position is Line 1, Column 1.

The screenshot shows a terminal window displaying the output of the `moja_aplikacja` program. The output is:

```

Hello world!
Process returned 0 (0x0)   execution time : 0.066 s
Press any key to continue.

```

## Rozwiązywanie problemów

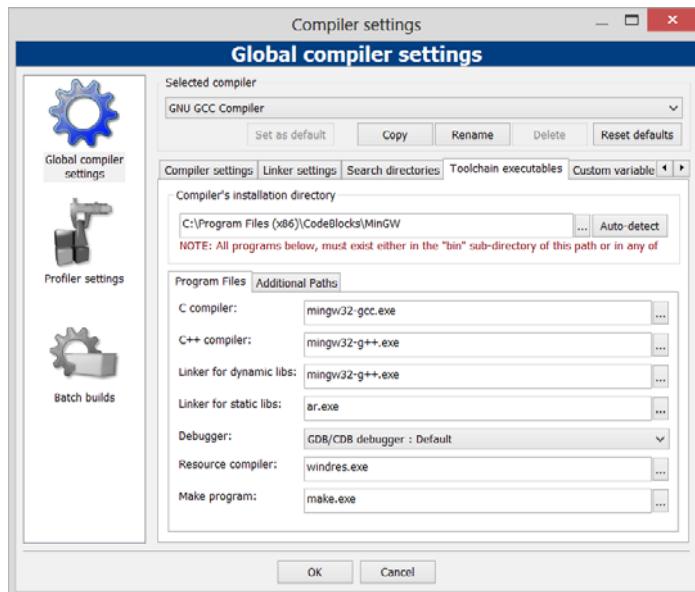
Jeśli z jakiegoś powodu nie udało Ci się uruchomić programu, prawdopodobnie wystąpiły błędy komplikacji lub nieprawidłowo skonfigurowałeś środowisko.

### Konfiguracja środowiska

Najczęściej pojawiający się błąd, jaki można zobaczyć, gdy program nie chce działać, jest komunikowany następująco: "CB01 – Debug" uses an invalid compiler. Probably the toolchain path within the compiler options is not setup correctly?! Skipping...<sup>1</sup>.

<sup>1</sup> Z ang. „CB01 – Debug” używa niewłaściwego kompilatora. Prawdopodobnie ścieżka toolchain w opcjach kompilatora została skonfigurowana nieprawidłowo?! Pomijanie... — przyp. tłum.

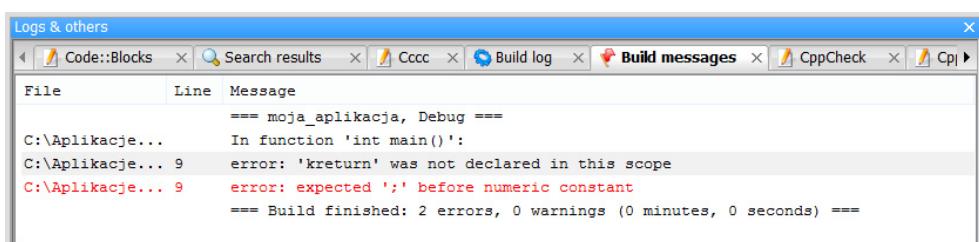
Najpierw sprawdź, czy pobrałeś poprawną wersję Code::Blocks — tę, która zawiera w nazwie MinGW. Jeśli problem nadal pozostaje nieroziwiążany, prawdopodobnie występuje kłopot z automatycznym wykrywaniem kompilatora. Aby sprawdzić bieżącą konfigurację tej opcji, wybierz z głównego menu *Settings* polecenie *Compiler...*, a następnie w grupie *Global Compiler Settings* (z symbolem koła zębatego) przejdź na kartę *Toolchain executables*. Na karcie tej znajduje się przycisk *Auto-detect*, z którego możesz skorzystać. Jeśli wciśnięcie tego przycisku nie rozwiązało problemu, możesz wypełnić pola tej karty ręcznie. Na poniższym rysunku pokazałem, jak konfiguracja ta wygląda w moim systemie. Jeśli zainstalowałeś Code::Blocks w innej lokalizacji, zmień ścieżkę znajdującą się w polu *Compiler's installation directory* i sprawdź, czy pozostałe pola są wypełnione zgodnie z rysunkiem.



Gdy już to zrobisz, ponownie naciśnij F9, aby sprawdzić, czy uda Ci się uruchomić program.

## Błędy kompilatora

Błędy kompilatora mogą wystąpić, jeśli zmodyfikujesz plik *main.cpp* w sposób niezrozumiały dla kompilatora. Aby dowiedzieć się, na czym polega problem, sprawdź okna *Build messages* i *Build log*. Okno *Build messages* pokazuje tylko błędy kompilatora, natomiast w oknie *Build log* znajdują się informacje dotyczące także innych problemów. Oto jak może wyglądać jedno z tych okien, kiedy pojawi się błąd.



W tym przypadku w oknie widoczna jest nazwa pliku, numer wiersza oraz krótki opis błędu. W kodzie zmieniłem instrukcję `return 0` na `return 0`, co nie jest poprawnym wyrażeniem w C++, w związku z czym wystąpił błąd.

Podczas programowania przekonasz się, że warto zaglądać do tego okna, kiedy Twój program nie chce się kompilować. Pomoże Ci to stwierdzić, co się stało.

W książce tej będziesz mieć do czynienia z wieloma przykładowymi kodami. Każdy z nich możesz utworzyć od początku albo zmodyfikować plik źródłowy swojego pierwszego programu. Zalecam, abyś dla każdego programu tworzył nową aplikację konsolową, dzięki czemu będziesz mógł wprowadzać zmiany w przykładowych kodach i zachowywać je do późniejszego oglądu.

## Czym właściwie jest Code::Blocks?

We wcześniejszej części tej książki przedstawiłem ideę zintegrowanego środowiska programistycznego. Code::Blocks jest takim środowiskiem, ponieważ ułatwia pisanie kodu źródłowego oraz kompilowanie programów z poziomu tej samej aplikacji. Powinieneś jednak zdawać sobie sprawę z pewnego faktu: Code::Blocks nie jest kompilatorem. Kiedy pobrzesz pakiet instalacyjny Code::Blocks, zawierał on **również** kompilator — w tym przypadku GCC od MinGW (<http://www.mingw.org/>), który jest bezpłatnym kompilatorem działającym pod Windows. Code::Blocks zajmuje się wszystkimi uciążliwymi szczegółami związanymi z konfiguracją i wywoływaniem kompilatora, który następnie zajmuje się rzeczywistą pracą.

## Macintosh

Podrozdział ten dotyczy wyłącznie konfigurowania środowiska programistycznego w systemie Mac OS X<sup>2</sup>.

Mac OS X udostępnia bazujące na powłoce Uniksa wydajne środowisko, z którego możesz korzystać, dzięki czemu będziesz miał dostęp do większości narzędzi omówionych w podrozdziale dotyczącym Linuksa. Możesz także wypróbować środowisko programistyczne Xcode firmy Apple. Niezależnie jednak od tego, czy będziesz korzystać z samego tylko środowiska Xcode, jego instalacja jest wymagana również wtedy, gdy będziesz chciał korzystać także ze standardowych narzędzi Linuksa.

Mimo że korzystanie ze środowiska Xcode nie jest konieczne do pisania programów w C++ na Maca, to jeśli masz zamiar zabrać się za programowanie z wykorzystaniem interfejsu użytkownika, powinieneś poznać także Xcode.

## Xcode

Xcode stanowi bezpłatną część Mac OS X, ale domyślnie nie jest instalowane. Xcode możesz znaleźć na swoim DVD z Mac OS X albo pobrać najnowszą jego wersję z internetu. Pakiet zawierający dokumentację jest bardzo duży, dlatego jeśli masz wolne połączenie z internetem,

<sup>2</sup> Jeśli używasz wersji Mac OS 9 albo wcześniejszej i nie masz możliwości zaktualizowania systemu, spróbuj pobrać z internetu środowisko Macintosh Programmer's Workshop. Ponieważ OS 9 jest już dość stary, nie będę mógł poprowadzić Cię przez proces konfiguracji Code::Blocks w tym systemie.

powinieneś poszukać Xcode na swoim dysku z Mac OS X. Zwróć uwagę, że nawet proste kompilatory, takie jak gcc albo g++, które zazwyczaj są domyślnie instalowane w środowisku Linuksa, nie zostaną domyślnie zainstalowane w Mac OS X. Aby mieć do nich dostęp, musisz pobrać Xcode Developer Tools.

## Instalowanie Xcode

W celu pobrania Xcode:

- Na stronie <https://developer.apple.com/register/> zarejestruj się jako programista Apple.
- Rejestracja jest bezpłatna. Na stronie Apple można odnieść wrażenie, że za rejestrację trzeba będzie zapłacić, ale powyższe łącze powinno przenieść Cię na stronę rejestracji bezpłatnej. Podczas rejestracji będziesz musiał podać podstawowe dane osobowe.
- Jeśli korzystasz z Liona albo nowszej wersji OS (10.7.x lub późniejszej), przejdź na stronę <https://developer.apple.com/downloads/index.action> i poszukaj Xcode. Powinieneś zobaczyć jego najnowszą wersję; kliknij ją. Pojawi się kilka łączów. Wybierz łącze opisane jako *Xcode* z dołączonym numerem wersji. Xcode możesz pobrać także z App store.
- Jeśli korzystasz ze Snow Leoparda (OS 10.6.x), powyższa wersja u Ciebie nie zadziała. Przejdź na stronę <https://developer.apple.com/downloads/index.action> i poszukaj *Xcode 3.2.6* — dostępny będzie tylko jeden taki wynik. Kliknij go, po czym kliknij łącze pobierania: *Xcode 3.2.6 and iOS SDK 43*.

Xcode jest dostępny pod postacią standardowego obrazu dysku, który możesz otworzyć. Otwórz go i uruchom plik *Xcode.mpkg*.

Na początku instalacji zostaniesz zapytany, czy akceptujesz umowę licencyjną, po czym zobacysz listę składników przygotowanych do zainstalowania. Składniki domyślne powinny wystarczyć. Zaakceptuj je i uruchom pozostałą część instalatora.

## Uruchamianie Xcode

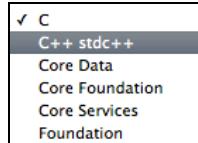
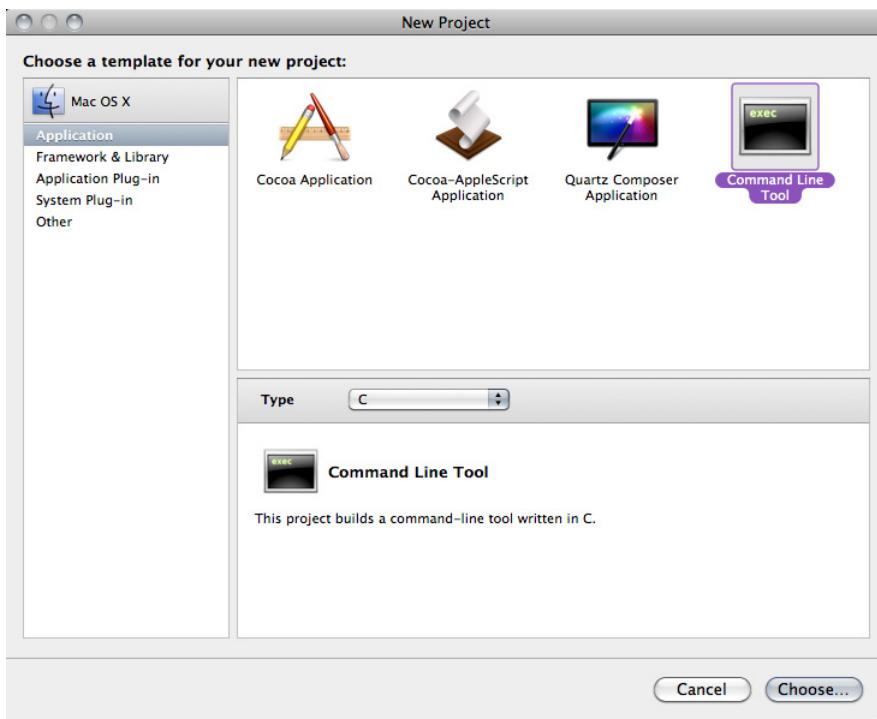
Po zakończeniu instalacji aplikacji Xcode będziesz mógł odnaleźć w *Developer/Applications/Xcode*. Uruchom ją. Xcode jest udostępniany z obszerną dokumentacją — być może zechcesz poświęcić trochę czasu na zapoznanie się z poradnikiem *Getting Started with Xcode*. W dalszej części tego podrozdziału założyłem jednak, że nie przeczytałeś żadnej innej dokumentacji.

## Tworzenie pierwszego programu C++ w Xcode

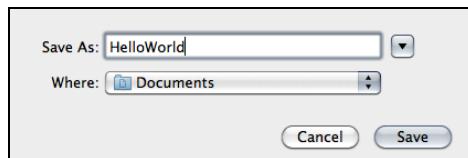
Zaczniemy więc. W głównym oknie Xcode, które pojawi się po uruchomieniu tej aplikacji, wybierz polecenie *Create a new Xcode project* (możesz także wybrać polecenie *File/New Project* albo nacisnąć klawisze *Shift+⌘+N*).

W pasku bocznym Mac OS X po lewej stronie wybierz opcję *Application*, a następnie *Command Line Tool* z prawej strony okna (zobacz pierwszy rysunek na kolejnej stronie).

Będziesz także musiał zmienić typ projektu widoczny na liście *Type* z *C* na *C++ stdc++* (zobacz drugi rysunek na kolejnej stronie).



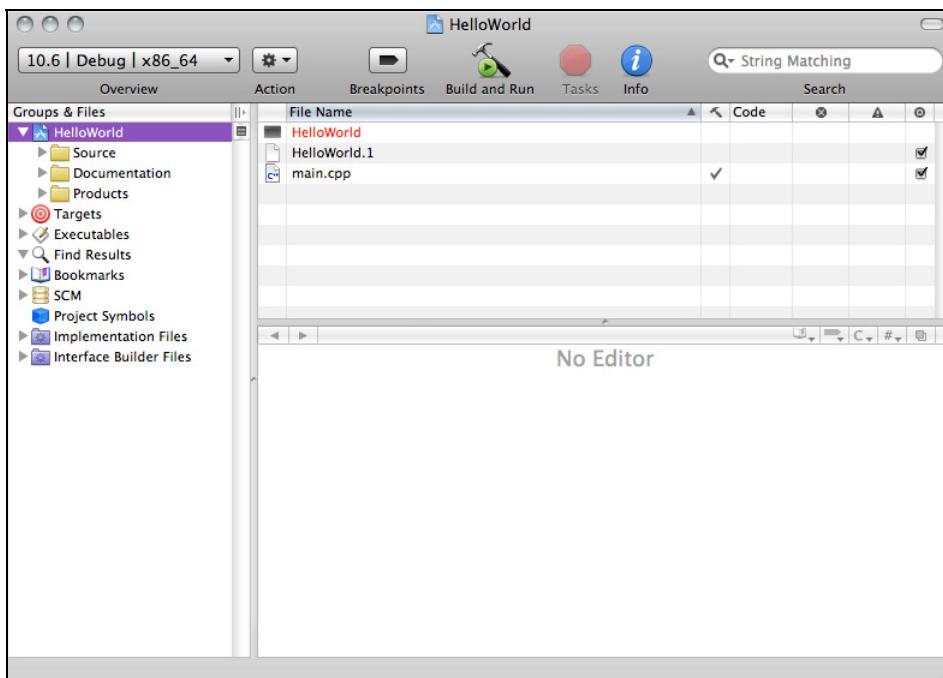
Kiedy już to zrobisz, kliknij przycisk *Choose...* i wybierz nazwę oraz lokalizację dla swojego nowego projektu. We wskazanej lokalizacji zostanie utworzony folder o takiej samej nazwie, jaką ma Twój projekt. Na potrzeby naszego przykładu wybrałem nazwę *HelloWorld*.



Teraz kliknij przycisk *Save*.

Po zapisaniu projektu zostanie otwarte nowe okno (zobacz rysunek na kolejnej stronie).

Na rysunku tym widać kilka elementów. Lewy pasek boczny udostępnia opcje *Source*, *Documentation* i *Products*. Folder *Source* zawiera pliki C++ przypisane do Twojego projektu, a folder *Documentation* całą dokumentację, jaką masz — zwykle jest to źródło „strony głównej” (na razie możesz go pominąć). Folder *Products* przechowuje rezultat komplikacji programu. Zawartość tych folderów możesz zobaczyć także w górnym środkowym oknie.



Popracujmy teraz nad plikiem źródłowym. W górnym środkowym oknie lub w folderze *Source* z lewej strony wybierz plik *main.cpp* (zobacz pierwszy rysunek na kolejnej stronie) (zwróć uwagę na rozszerzenie: standardowym rozszerzeniem plików źródłowych C++ nie jest *.txt*, ale *.cpp*, chociaż pliki te zawierają czysty tekst). Jeśli klikniesz raz, plik źródłowy zostanie otwarty w oknie zatytułowanym *No Editor*. Teraz możesz zacząć pisać bezpośrednio w pliku.

Jeśli potrzebujesz więcej miejsca, możesz dwukrotnie kliknąć nazwę pliku, co spowoduje otworzenie większego okna edytora.

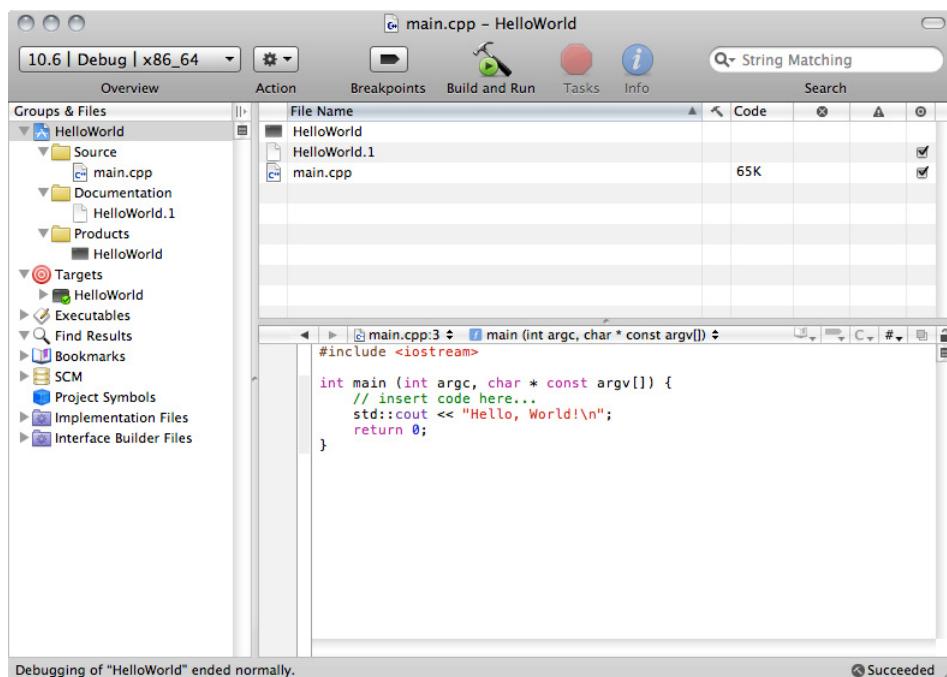
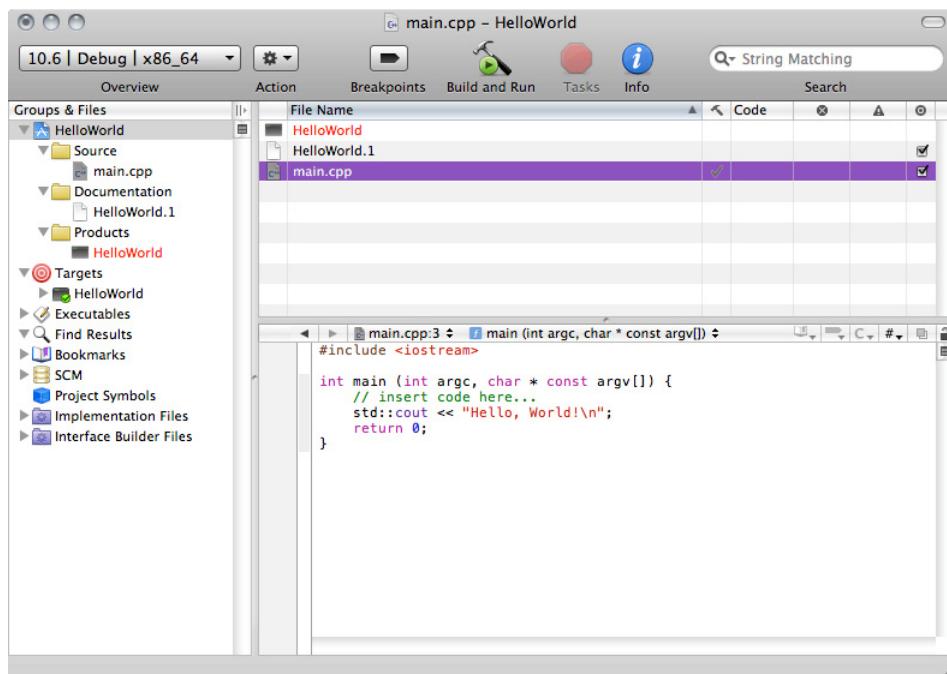
Xcode domyślnie udostępnia niewielki program przykładowy, z którym możesz rozpoczęć pracę. Spróbujmy go skompilować i uruchomić. Najpierw na pasku narzędzi kliknij przycisk *Build and Run*.

Po kliknięciu tego przycisku program skompiluje się, co oznacza, że zostanie utworzony plik wykonywalny. W Xcode 3 nie zobaczysz jednak, że coś zostało uruchomione. W tym celu powinieneś kliknąć plik wykonywalny *HelloWorld*. Zwróci uwagę, że wcześniej jego nazwa była wyróżniona kolorem czerwonym, a obecnie jest czarna (zobacz drugi rysunek na kolejnej stronie).

Teraz kliknij dwukrotnie nazwę pliku, aby uruchomić swój pierwszy program!

W rezultacie powinieneś zobaczyć okno, które wygląda mniej więcej tak (imię użytkownika ukrytem w celu ochrony prywatności osoby, która pożyczyła mi Macintosha na potrzeby wykonania tego rysunku) (zobacz rysunek na stronie 28).

Proszę bardzo — uruchomiłeś swój pierwszy program!





Od tej pory, gdy będziesz chciał uruchomić przykładowy program, będziesz mógł skorzystać z projektu, który właśnie utworzyłeś, albo przygotować nowy projekt. Tak czy owak, aby dodać własny kod, będziesz mógł zmodyfikować przykładowy program, który Xcode utworzy w pliku *main.cpp*.

## Instalowanie Xcode 4

Aby pobrać Xcode 4, odszukaj go w Mac App Store, po czym go zainstaluj. Zajmuje on około 4,5 GB miejsca na dysku. Pobrany z Mac App Store plik będzie widoczny na Twoim Docku jako ikona *Install Xcode*. Uruchom ją w celu rozpoczęcia procesu instalacji.

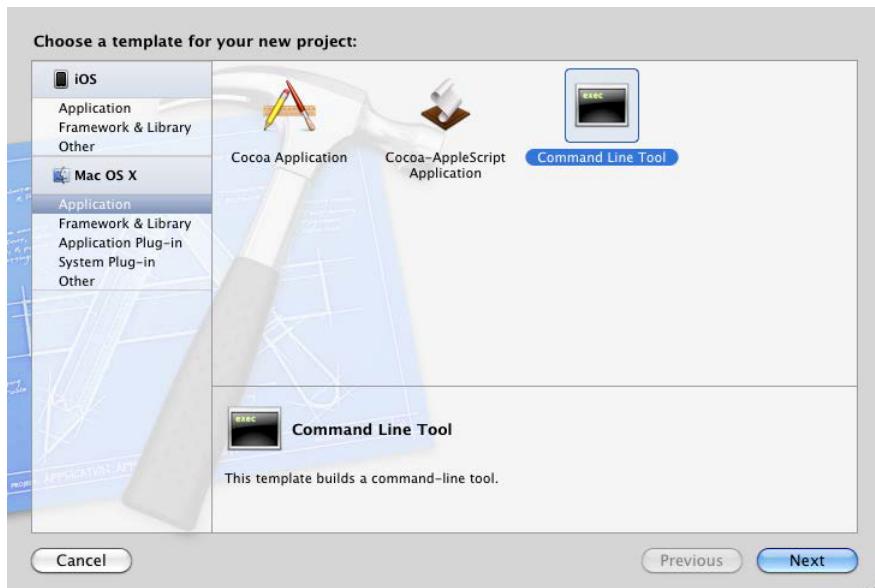
Na początku instalacji zostaniesz zapytany, czy akceptujesz umowę licencyjną, po czym zobaczysz listę składników przygotowanych do zainstalowania. Składniki domyślne powinny wystarczyć. Zaakceptuj je i uruchom pozostałą część instalatora.

## Uruchamianie Xcode

Po zakończeniu instalacji aplikacji Xcode będziesz mógł odnaleźć w *Developer/Applications/Xcode*. Uruchom ją. Xcode jest udostępniany z obszerną dokumentacją — być może zechcesz poświęcić trochę czasu na zapoznanie się z poradnikiem *Xcode Quick Start Guide*, który możesz otworzyć po kliknięciu łącza *Learn about using Xcode* na ekranie startowym. W dalszej części tego podrozdziału założyłem jednak, że nie przeczytałeś żadnej innej dokumentacji.

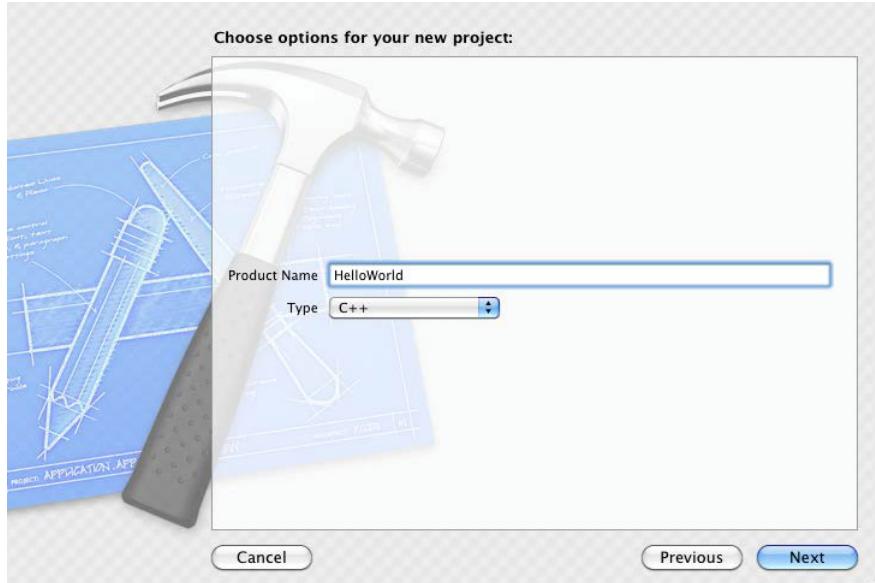
## Tworzenie pierwszego programu C++ w Xcode

Zacznijmy więc. W głównym oknie Xcode, które pojawia się po uruchomieniu tej aplikacji, wybierz polecenie *Create a new Xcode project* (możesz także wybrać polecenie *File/New/New Project* albo nacisnąć klawisze *Shift+⌘+N*). Zostanie otworzone okno, które wygląda następująco:



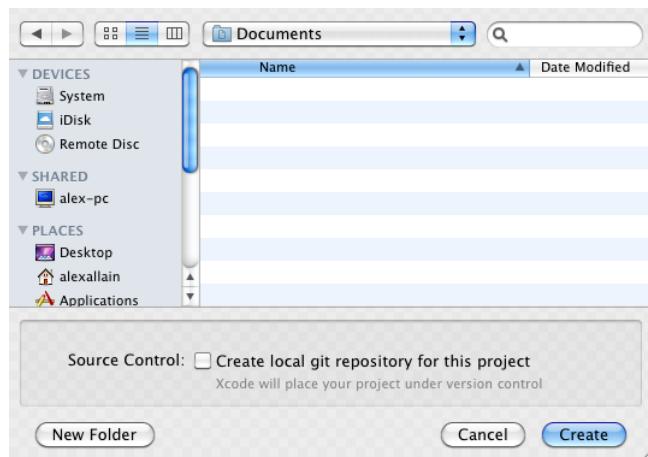
Z lewego paska bocznego wybierz w sekcji *Mac OS X* opcję *Application*, po czym w prawej części okna wskaz *Command Line Tool* (opcję *Application* możesz zobaczyć także w sekcji *iOS*, ale teraz jej nie wybieraj) i kliknij przycisk *Next*.

Po kliknięciu *Next* zobaczysz następujący ekran:



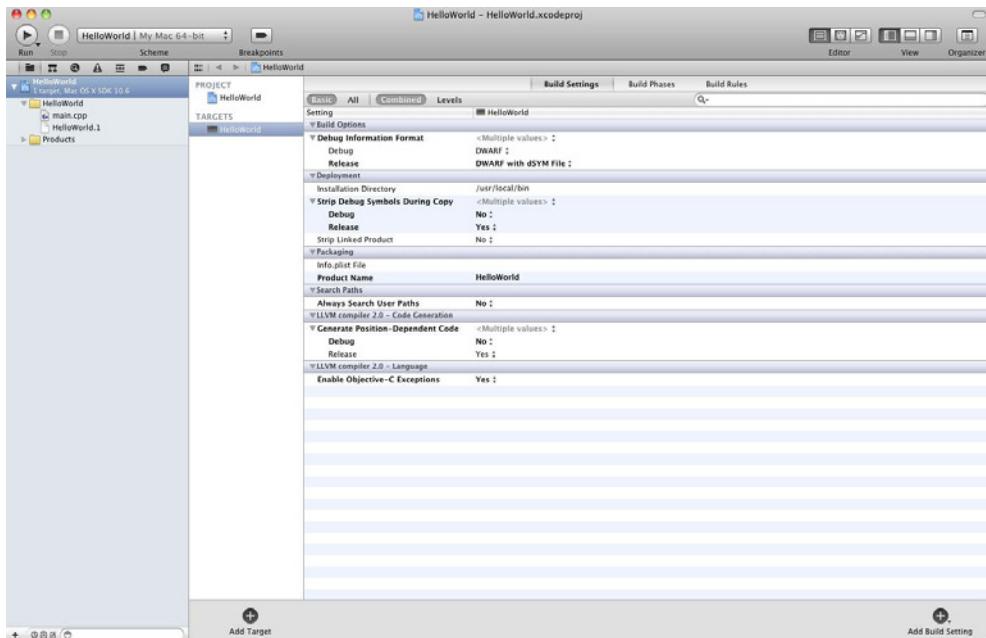
Wpisałem już nazwę programu — *HelloWorld* — i wybrałem typ C++ (domyślnie jest to C). Zrób to samo, po czym kliknij przycisk *Next*.

Po kliknięciu *Next* zostanie wyświetlony następujący ekran:



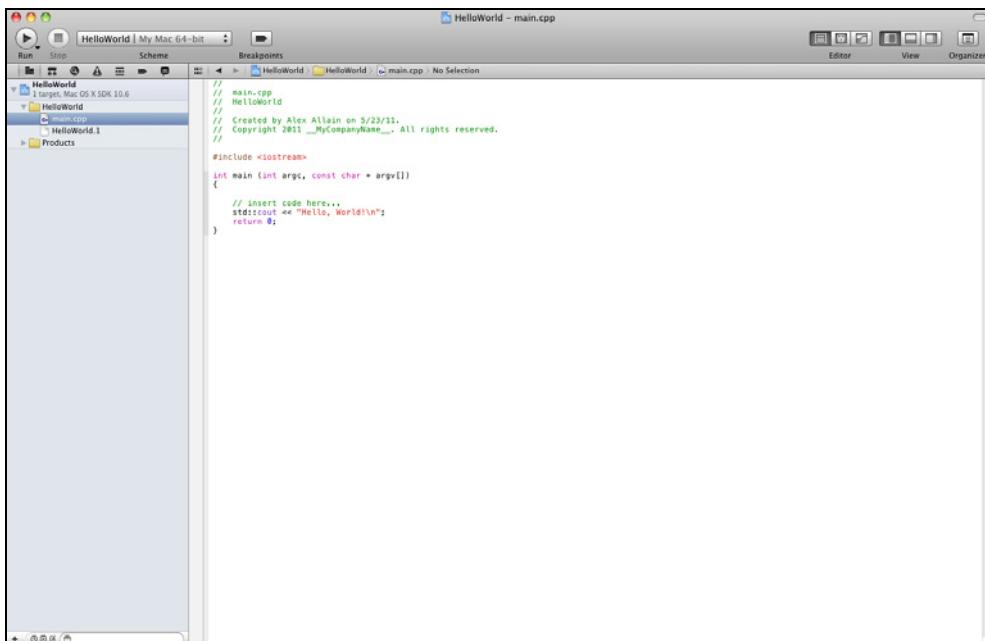
Jeśli zaznaczone jest pole wyboru *Create local git repository for this project*, usuń to zaznaczenie. Git to system kontroli źródła, który umożliwia przechowywanie wielu wersji projektu, ale jego omówienie wykracza poza zakres tej książki. Powinieneś także wybrać dla swojego projektu lokalizację — ja umieściłem go w folderze *Documents*. Kiedy już dokonasz powyższych wyborów, kliknij przycisk *Create*.

Zostanie wyświetcone nowe okno, które wygląda następująco:



Na rysunku tym widać kilka elementów. Lewy pasek boczny daje dostęp do kodu źródłowego oraz folderu *Products*. Kod źródłowy znajduje się w katalogu o nazwie zgodnej z nazwą Twojego projektu. W naszym przypadku jest to *HelloWorld*. Większość pozostałych elementów tego ekranu pokazuje konfigurację kompilatora — w tej chwili nie musimy nic z nią robić.

Popracujmy teraz nad plikiem źródłowym. W folderze w lewym pasku bocznym wybierz plik *main.cpp* (zwróć uwagę na rozszerzenie: standardowym rozszerzeniem plików źródłowych C++ nie jest *.txt*, ale *.cpp*, chociaż pliki te zawierają czysty tekst). Jeśli klikniesz raz, plik źródłowy zostanie otwarty w oknie głównym. Teraz możesz zacząć pisać bezpośrednio w pliku.



Możesz także dwukrotnie kliknąć nazwę pliku, aby otworzyć okno edytora, którego położenie na ekranie można zmieniać.

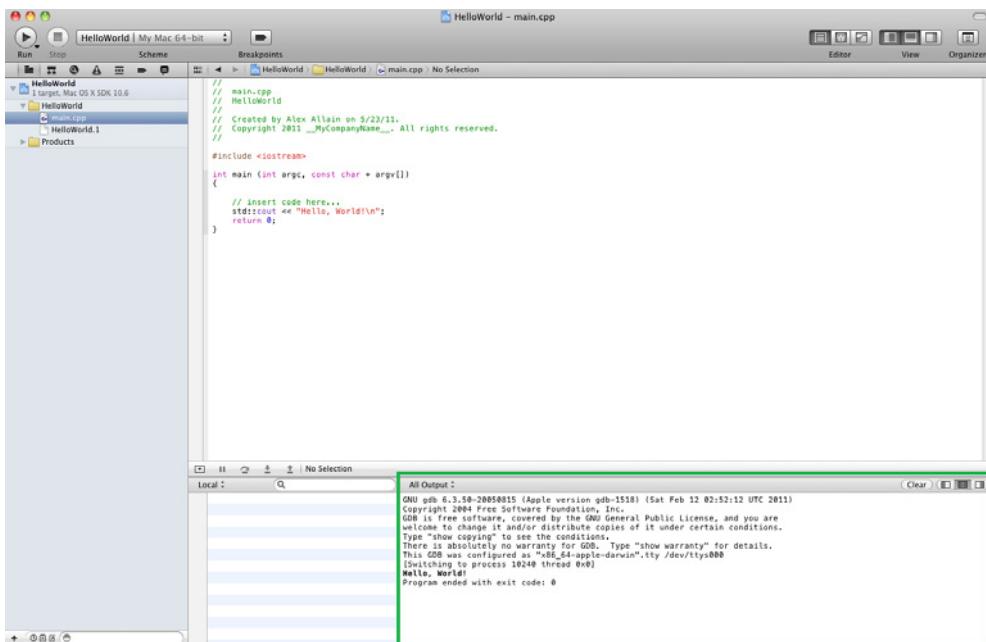
Domyślnie Xcode udostępnia niewielki program, od którego możesz rozpoczęć swoją pracę. Spróbujmy go skompilować i uruchomić. Wszystko, co musisz zrobić, sprowadza się do kliknięcia przycisku *Run* na pasku narzędzi. Wynik działania programu zostanie wyświetlony w prawym dolnym rogu okna (zobacz pierwszy rysunek na kolejnej stronie).

Proszę bardzo — uruchomisz swój pierwszy program!

Od tej pory, gdy będziesz chciał uruchomić przykładowy program, będziesz mógł skorzystać z projektu, który właśnie utworzyłeś, albo przygotować nowy projekt. Tak czy owak, aby dodać własny kod, będziesz mógł zmodyfikować przykładowy program, który Xcode utworzy w pliku *main.cpp*.

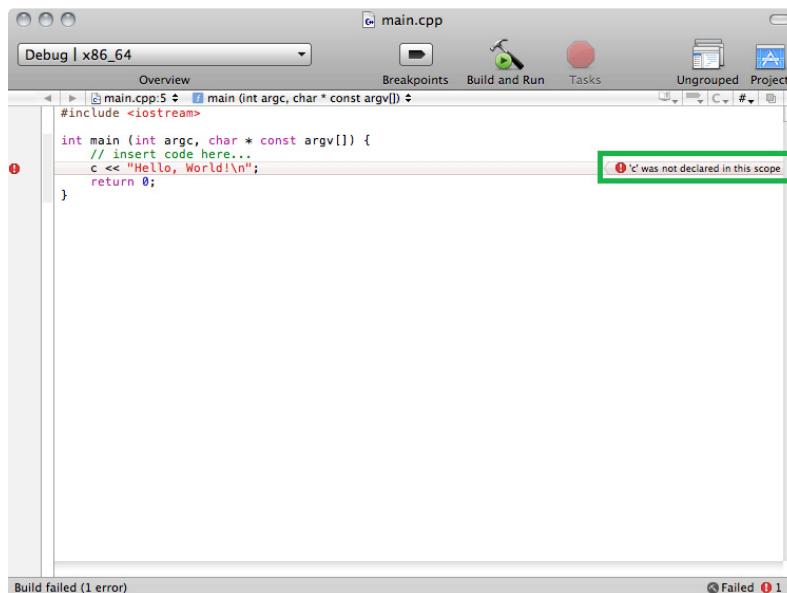
## Rozwiązywanie problemów

Zrzuty ekranowe zamieszczone w tym podrozdziale pochodzą z Xcode 3. Ewentualne różnice między wersjami Xcode 3 i Xcode 4 odpowiednio opiszę w dalszej części rozdziału.



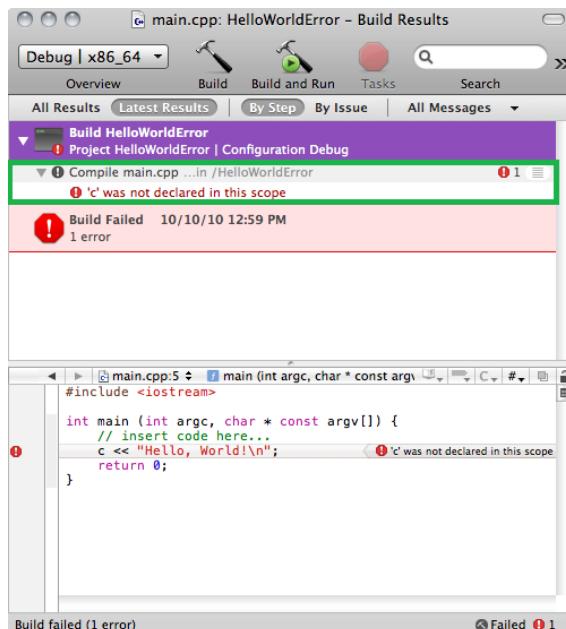
Czasami zdarza się, że z jakiegoś powodu program nie da się skompilować. Zwykle tak jest z powodu błędów kompilatora (na przykład pomyłka literowa w treści programu przykładowego albo rzeczywisty błąd w programie). Gdy tak się stanie, kompilator wyświetli jeden lub kilka komunikatów o błędach.

Xcode wyświetla komunikaty o błędach bezpośrednio w kodzie źródłowym, w wierszu, w którym wystąpił błąd. W naszym przykładzie zmieniłem oryginalny program w taki sposób, aby zamiast `std::cout` pozostała sama litera `c`.



W ramce możesz zobaczyć błąd komplikacji — Xcode nie wie, co znaczy c. W lewym dolnym rogu okna widać także komunikat o tym, że komplikacja nie powiodła się. Z kolei w prawym dolnym rogu widoczna jest liczba błędów (w tym przypadku jest to 1). W Xcode 4 ikona jest podobna, z tym że znajduje się w prawym górnym rogu.

Jeśli chcesz zobaczyć pełną listę błędów w Xcode 3, możesz kliknąć ikonę z młotkiem w prawym dolnym rogu okna, co spowoduje otwarcie okna dialogowego z wszystkimi błędami wykrytymi przez kompilator.



Ramką zaznaczyłem miejsce, w którym możesz przeczytać opis błędu. Jeśli je klikniesz, zostanie otwarte małe okno edytora, w którym będziesz mógł zobaczyć właściwy błąd.

W Xcode 4 prawy panel, zawierający pliki źródłowe, zostanie zastąpiony listą błędów kompilatora, jeśli komplikacja się nie powiedzie.

Kiedy już naprawisz błąd, możesz kliknąć przycisk *Build and Run*, aby spróbować ponownie skompilować kod.

## Linux

Jeżeli pracujesz z Linuksem, niemal na pewno masz zainstalowany kompilator C++. Zwykle użytkownicy Linuksa korzystają z kompilatora C++ o nazwie g++, który jest częścią *GNU Compiler Collection (GCC)*.

### Krok 1. Instalowanie g++

Aby sprawdzić, czy masz zainstalowany g++, otwórz okno terminala. Wpisz g++ i naciśnij *Enter*. Jeśli kompilator jest zainstalowany, powinieneś zobaczyć komunikat:

```
g++: no input files
```

Jeśli wyświetli się następująca informacja:

```
command not found
```

prawdopodobnie powinieneś zainstalować g++. Instalacja g++ zależy od systemu zarządzania pakietami Twojej dystrybucji Linuksa. Jeśli jest to na przykład Ubuntu, spróbuj wpisać po prostu:

```
aptitude install g++
```

Inne dystrybucje Linuksa mogą mieć podobnie proste systemy zarządzania pakietami lub będą wymagać dodatkowych czynności. Więcej informacji znajdziesz w dokumentacji swojej dystrybucji Linuksa.

## Krok 2. Uruchomienie g++

Uruchomienie g++ jest względnie łatwe. Teraz napiszesz swój pierwszy program. Utwórz prosty plik z rozszerzeniem *.cpp*, zawierający następującą treść:

### Przykładowy kod 1.: *hello.cpp*

```
#include <iostream>
int main ()
{
    std::cout << "Hello, world" << std::endl;
}
```

Zapisz go jako *hello.cpp* i zapamiętaj katalog, w którym go umieściłeś (zwróć uwagę na rozszerzenie: standardowym rozszerzeniem plików źródłowych C++ nie jest *.txt*, ale *.cpp*, chociaż pliki te zawierają czysty tekst).

Powróć do okna terminala i przejdź do katalogu, w którym zapisałeś plik.

Wpisz:

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

po czym naciśnij *Enter*.

Opcja *-o* informuje o nazwie pliku wynikowego. Jeśli jej nie użyjesz, domyślną nazwą tego pliku będzie *a.out*.

## Krok 3. Uruchomienie programu

Plikowi nadaliśmy nazwę *hello*, tak więc możesz uruchomić swój nowy program, wpisując:

```
./hello
```

Powinieneś zobaczyć następujący wynik jego działania:

```
Hello, world
```

Oto Twój pierwszy program, witający się z nowym, wspaniałym światem.

## Rozwiązywanie problemów

Jest możliwe, że z jakiegoś powodu Twój program nie skompiluje się; zwykle przyczyną jest błąd komplikacji (jeśli na przykład wprowadzisz program z błędem). Gdy tak się stanie, kompilator wyświetli jeden lub kilka komunikatów o błędach.

Jeżeli na przykład we wcześniejszym programie wstawisz `x` przed instrukcją `cout`, kompilator pokaże następujące informacje o błędach:

```
gcc_ex2.cc: In function 'int main ()':
gcc_ex2.cc:5: error: 'xcout' is not a member of 'std'
```

W każdej informacji znajduje się nazwa pliku, numer wiersza oraz komunikat błędu. W tym przypadku kompilator nic nie wie na temat instrukcji `xcout`, ponieważ poprawną instrukcją powinno być `cout`.

## Krok 4. Konfigurowanie edytora tekstowego

Jeśli używasz Linuksa, będzie Ci potrzebny także dobry edytor tekstów. Dla Linuksa istnieje kilka wysokiej klasy edytorów, takich jak Vim (<http://www.vim.org/>) albo Emacs (<http://www.gnu.org/software/emacs/>). Sam podczas pracy z Linuksem używam Vima. Edytory te są jednak stosunkowo trudne do opanowania i wymagają poświęcenia im czasu. W dłuższej perspektywie są tego warte, ale zapewne nie będziesz chciał poświęcać na nie czasu akurat teraz, kiedy właśnie zaczynasz uczyć się programowania. Jeżeli jeden z tych dwu edytorów jest już Ci znany, spokojnie możesz nadal z niego korzystać.

Jeśli jeszcze nie masz swojego ulubionego edytora, możesz wypróbować na przykład edytor **nano**. Nano (<http://www.nano-editor.org/>) jest relatywnie prostym edytorem, chociaż udostępnia kilka wartościowych funkcji, takich jak wyróżnianie składni oraz automatyczne wcinanie tekstu (dzięki czemu nie będziesz musiał naciskać klawisza tabulacji za każdym razem, kiedy wstawiasz nowy wiersz w swoim programie; może to wydawać się trywialne, ale taka funkcja naprawdę będzie Ci potrzebna). Nano bazuje na edytorze o nazwie pico, który jest bardzo łatwy w nauce, ale brakuje mu wielu funkcji przydatnych podczas programowania. Być może nawet używałeś pico, jeśli korzystałeś z programu pocztowego Pine. Jeżeli nie, to nic straconego — praca z nano nie wymaga wcześniejszego doświadczenia z pico.

Być może już masz nano. Aby się tego dowiedzieć, wpisz w oknie terminala nano. Program powinien się uruchomić. Jeżeli tak się nie stanie i zobaczysz jakąś odmianę komunikatu

```
command not found
```

będziesz musiał zainstalować nano. W tym celu zastosuj się do instrukcji dołączonej do menedżera pakietu Twojej dystrybucji Linuksa. Podrozdział ten pisałem, mając na myśl nano w wersji 2.2.4, chociaż zapewne będzie on pasował także do jego późniejszych wersji.

## Konfigurowanie nano

Aby mieć możliwość korzystania z pewnych funkcji nano, będziesz musiał dokonać edycji pliku konfiguracyjnego nano. Plik ten nazywa się `.nanorc` i — tak jak większość plików konfiguracyjnych zawierających ustawienia specyficzne dla użytkownika w Linuksie — znajduje się w katalogu domowym (`~/.nanorc`).

Jeśli plik ten już istnieje, będziesz mógł go od razu wyedytować; jeśli go nie ma, będziesz go musiał najpierw utworzyć (jeżeli nie masz żadnego doświadczenia w korzystaniu z edytorów tekstowych działających pod kontrolą systemu Linux, będziesz mógł posłużyć się nano; gdybyś potrzebował pomocy w posługiwaniu się podstawowymi funkcjami nano, czytaj dalej).

Aby poprawnie skonfigurować nano, skorzystaj z przykładowego pliku *.nanorc*, który jest dodatkowy do tej książki. Zapewnia on wyróżnianie składni oraz automatyczne wcinanie tekstu, co znacznie ułatwia edycję kodu źródłowego.

## Korzystanie z nano

Aby utworzyć nowy plik, uruchom nano bez argumentów. Aby otworzyć do edycji istniejący plik, podaj jego nazwę w wierszu poleceń:

```
nano hello.cpp
```

Jeśli taki plik nie istnieje, nano przejdzie do edycji skojarzonego z nim nowego bufora. Plik ten nie zostanie jednak utworzony na dysku, dopóki nie zapiszesz dokonanych w nim zmian.

Oto przykład, jak może wyglądać nano po uruchomieniu:



W ramce u góry znajduje się nazwa edytowanego pliku lub oznaczenie *New Buffer*, jeśli nano został uruchomiony bez podania nazwy pliku.

W ramce u dołu wymieniono szereg poleceń wywoływanych klawiszami. Znak  $\wedge$  znajdujący się przed literą oznacza, że łącznie z daną literą powinieneś na klawiaturze nacisnąć także klawisz *Ctrl*. Na przykład wyjście z nano jest oznaczone jako  $\wedge X$ , tak więc w celu opuszczenia edytora powinieneś nacisnąć klawisze *Ctrl+X*. Wielkość liter nie ma znaczenia.

Jeśli przychodzisz ze świata Windows, niektóre z okreseń używanych w nano mogą być Ci obce. Rzućmy zatem okiem na niektóre z podstawowych operacji wykonywanych w nano.

## Edytowanie tekstu

Kiedy uruchomisz nano, będziesz mógł utworzyć nowy plik albo otworzyć plik już istniejący. Chwilę potem możesz zacząć wpisywać w nim tekst. Pod tym względem nano bardzo przypomina Notatnik w Windows. Jeśli jednak zechcesz skorzystać z funkcji kopiowania i wklejania, to nazwy tych poleceń będą inne: *Cut Text* (*Ctrl+K*) oraz *UnCut Text* (*Ctrl+U*). Jeśli nie zaznaczyłeś żadnego fragmentu, polecenia te domyślnie wytną pojedynczy wiersz tekstu.

Możesz też wyszukiwać tekst za pomocą funkcji *Where Is* (*Ctrl+W*), która spowoduje wyświetlenie nowego zestawu opcji, chociaż najprostsze jej użycie polega na wpisaniu z klawiatury szukanego fragmentu.

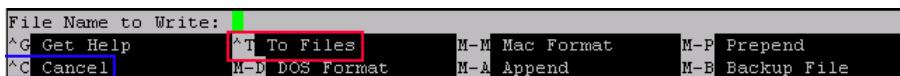
Możesz przechodzić do poprzedniej i następnej strony, korzystając z funkcji *Prev Page* (*Ctrl+Y*) i *Next Page* (*Ctrl+V*). Zwróć uwagę, że skróty klawiaturowe nie mają wiele wspólnego ze skortami używanymi w Windows.

Jedynie ważne funkcje, których brakuje w nano (w obecnej wersji 2.2), a w które wyposażona jest większość innych edytorów, to cofanie i powtarzanie czynności. Czynności cofania i powtarzania mają w nano charakter jedynie eksperymentalny i są domyślnie wyłączone.

Po naciśnięciu *Alt+R* możesz w nano korzystać z funkcji szukania i zastępowania tekstu. Najpierw zostaniesz poproszony o podanie tekstu do odnalezienia, a następnie wpisanie tekstu, który ma go zastąpić.

## Zapisywanie plików

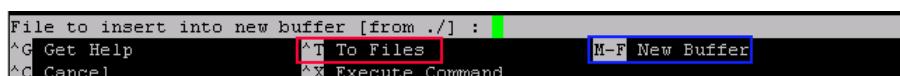
W języku nano zapisywanie tekstu to *WriteOut* (*Ctrl+O*).



Kiedy wywołasz polecenie *WriteOut*, zawsze będziesz musiał podać nazwę pliku do zapisania, nawet jeśli edytowany plik otworzyłeś wcześniej. Jeżeli edytujesz istniejący plik, jego nazwa zostanie domyślnie podpowiedziana, wystarczy zatem, że naciśniesz *Enter*, a plik zostanie zapisany. Jeśli chcesz zapisać plik w nowej lokalizacji, możesz podać nazwę pliku do zapisania albo skorzystać z opcji *To Files* (*Ctrl+T*) w celu wybrania tego pliku. Większość polecen ma opcję wycofania czynności — *Cancel* (*Ctrl+C*), jednak w odróżnieniu od Windows nie jest ona wywoływana klawiszem *Esc*. Pozostałymi opcjami nano nie musisz się teraz przejmować — przez większość czasu nie będziesz z nich korzystać.

## Otwieranie plików

Jeśli chcesz otworzyć plik do edycji, możesz skorzystać z polecenia *Read File* (*Ctrl+R*). Jego wywołanie spowoduje wyświetlenie nowego menu z opcjami.



Jeśli zamiast wstawiać tekst bezpośrednio do edytowanego właśnie pliku, wolisz otworzyć nowy plik, wybierz polecenie *New Buffer*. Skrót klawiaturowy tego polecenia to *M+F*, gdzie *M* oznacza klawisz Meta. Zwykle w takiej sytuacji klawiszem tym jest *Alt* (*Alt+F*)<sup>3</sup>. W ten sposób informujesz nano, że chcesz otworzyć plik. Kiedy już to zrobisz, będziesz mógł wpisać nazwę pliku albo za pomocą klawiszy *Ctrl+T* wywołać listę, która pozwoli wybrać pliki do edycji. Jak zwykle, przy użyciu klawiszy *Ctrl+C* można tę czynność odwołać.

<sup>3</sup> Niektóre osoby mogą mieć problem z użyciem klawisza *Alt* w roli metaklawisza. Jeżeli okaże się, że Twój *Alt* nie działa, zawsze przed naciśnięciem klawisza z literą możesz nacisnąć i puścić klawisz *Esc*. W tym przypadku naciśnięcie *Esc*, *F* działa tak samo jak naciśnięcie *Alt+F*.

## Przeglądanie pliku źródłowego

Teraz, kiedy znasz już podstawowe sposoby edytowania plików w nano, powinieneś wiedzieć, jak otworzyć plik i rozpoczęć z nim pracę. Jeżeli poprawnie skonfigurowałeś swój plik *.nanorc*, po otwarciu pliku źródłowego *hello.cpp* (który uruchomiliśmy wcześniej) powinieneś zobaczyć ekran wyglądający mniej więcej tak jak na poniższym rysunku. Tekst jest wyświetlany w różnych kolorach, które zależą od funkcji, jaką pełni. Na przykład napis "Hello, world" jest wyświetlany w kolorze różowym.

```
#include <iostream>

int main()
{
    std::cout << "Hello, world" << std::endl;
}
```

Wyróżnianie składni bazuje na rozszerzeniu pliku, dlatego dopóki nie zapiszesz pliku źródłowego jako *.cpp*, funkcja wyróżniania nie będzie aktywna.

Od tej pory, gdy tylko zechcesz uruchomić przykładowy program, będziesz mógł utworzyć dla niego nowy plik tekstowy w nano, po czym skompilować go, wykonując opisane wcześniej czynności.

## Więcej informacji

Teraz powinieneś już wiedzieć, jak edytować w nano podstawowe pliki, ale jeśli chciałbyś uzyskać więcej informacji, sięgnij po wbudowaną pomoc, która jest dostępna po naciśnięciu klawiszy *Ctrl+G*. Szczególnie przydatna okazała się dla mnie poniższa strona, która wyjaśnia bardziej zaawansowane funkcje nano:

<http://freethegnu.wordpress.com/2007/06/23/nano-shortcuts-syntax-highlight-and-nanorc-config-file-pt1/>.

# 2

## ■ ■ ■ R O Z D Z I A Ł 2

# Podstawy C++

---

## Wprowadzenie do języka C++

Jeśli skonfigurowałeś już swoje środowisko programistyczne, tak jak to opisałem w poprzednim rozdziale, masz szansę uruchomić swój pierwszy program. Gratuluję! To wielki krok.

W tym rozdziale przedstawię podstawowe cegiełki języka C++, które umożliwiają Ci rozpoczęcie pisania Twoich pierwszych programów. Pokażę kilka podstawowych koncepcji, z którymi będziesz mieć stale do czynienia. Przedstawię strukturę programu, jego funkcję `main`, ideę funkcji standardowych udostępnianych przez kompilator, sposób dodawania komentarzy do programu oraz pokrótkie omówię specyfikę myślenia programisty.

## Najprostszy program w C++

Najpierw rzućmy okiem na najprostszy możliwy program — taki, który nic nie robi — i przeanalizujmy go krok po kroku.

### Przykładowy kod 2.: `pusty.cpp`

```
int main ()  
{  
}
```

Widzisz? Wcale nie jest taki straszny!

Pierwszy wiersz:

```
int main ()
```

mówi kompilatorowi, że istnieje funkcja `main`, zwracająca liczbę całkowitą, której skrót w C++ to `int`. **Funkcja** to fragment napisanego przez kogoś kodu, zwykle korzystający z innych funkcji lub pewnych podstawowych cech języka. W tym przypadku nasza funkcja nic nie robi, ale już niedługo zobaczymy funkcję, która coś potrafi.

Funkcja `main` jest funkcją specjalną. To jedyna funkcja, która musi znajdować się we wszystkich programach napisanych w C++; to punkt, od którego program zaczyna działanie, kiedy go uruchamiasz. Funkcja `main` jest poprzedzona typem wartości, jaką zwraca — `int`. Gdy funkcja zwraca wartość, kod, który ją wywołał, będzie mieć do niej dostęp. W przypadku funkcji `main` wartość zwrotna jest przekazywana do systemu operacyjnego. Zwykle będziemy musieli otwarcie zwracać wartość w funkcjach, ale C++ pozwala na pominięcie instrukcji `return` w funkcji `main` i wówczas zwrócona zostanie wartość 0, która mówi systemowi operacyjnemu, że wszystko poszło jak należy.

**Nawiasy klamrowe**, { i }, oznaczają początek i koniec funkcji (i — jak się zaraz przekonamy — także początek i koniec innych bloków kodu). Możesz myśleć, że ich znaczenie to *rozpocznij* i *zakończ*. W naszym przypadku wiemy, że funkcja nic nie robi, ponieważ między nawiasami klamrowymi nic nie ma.

Kiedy uruchomisz ten program, nie zobaczyś żadnego efektu, przejdźmy zatem do programu, który jest trochę bardziej interesujący (ale tylko trochę).

### Przykładowy kod 3.: hello.cpp

```
#include <iostream>
using namespace std;
int main ()
{
    cout << "Hej, jestem tutaj! Och, i oczywiście Hello World!\n";
}
```

Przede wszystkim zwróć uwagę na to, że między nawiasami klamrowymi coś jest. Oznacza to, że program coś robi! Zbadajmy ten program krok po kroku.

Pierwszy wiersz:

```
#include <iostream>
```

to **instrukcja include**, która mówi kompilatorowi, żeby przed utworzeniem pliku wykonywalnego wstawił do naszego programu kod z pliku nagłówkowego o nazwie `iostream`. Plik nagłówkowy `iostream` znajduje się wśród plików kompilatora i umożliwia realizowanie operacji wejścia i wyjścia. Użycie instrukcji `#include` powoduje pobranie całej zawartości pliku nagłówkowego i wklejenie jej do Twojego programu. Dodażając pliki nagłówkowe, uzyskujesz możliwość pracy z wieloma funkcjami, które udostępnia Twój kompilator.

Jeśli chcemy uzyskać dostęp do podstawowych funkcji, musimy dołączyć plik nagłówkowy, który udostępnia te funkcje. Na razie większość potrzebnych nam funkcji znajduje się w pliku nagłówkowym `iostream`, tak więc będziesz go widzieć na początku prawie każdego programu. Niemal wszystkie programy, które napiszesz, będą zaczynać się jedną lub wieloma instrukcjami `include`.

Po instrukcji `include` znajduje się następujący wiersz:

```
using namespace std;
```

Jest to szablonowy kod, który pojawia się niemal w każdym programie napisanym w C++. Na razie będziemy go używać na początku wszystkich naszych programów, tuż pod instrukcją `include`. Kod ten ułatwia posługiwanie się krótszymi wersjami niektórych procedur udostępnionych w pliku nagłówkowym `iostream`. Później opowiem dokładniej, jak to działa — w tej chwili zapamiętaj tylko, że należy go dołączać.

Zauważ, że wiersz ten kończy się średnikiem. Taki średnik stanowi część składni języka C++. Mówi on kompilatorowi, że zakończyłeś swoją instrukcję. Średnik jest używany do końca większości instrukcji w C++. Pominięcie średnika to jeden z błędów najczęściej popełnianych przez początkujących programistów. Jeżeli Twój program z jakiegoś powodu nie działa, sprawdź, czy nie zapomniałeś postawić gdzieś średnika. Za każdym razem, gdy będę wprowadzać nowe pojęcia, powiem Ci, czy wymagają one użycia średnika.

Następnie mamy funkcję `main`, od której zaczyna się program:

```
int main ()
```

Następny wiersz programu, z zabawnym symbolem `<<`, może wydawać się dziwny.

```
cout << "Hej, jestem tutaj! Och, i oczywiście Hello World!\n";
```

W tym miejscu C++ korzysta z obiektu cout w celu wyświetlenia tekstu. Uzyskanie dostępu do tej instrukcji jest powodem, dla którego dołączymy plik nagłówkowy iostream.

Użyty został symbol <<, znany jako „operator wstawiania”, który wskazuje, co ma się pojawić na wyjściu. W skrócie: instrukcja cout << powoduje wywołanie funkcji, której argumentem jest tekst. Wywołanie funkcji sprawia, że uruchamiany jest kod z nią skojarzony. Funkcje zazwyczaj przyjmują **argumenty**, które są wykorzystywane w ich kodach. W tym przypadku jako argument został przekazany łańcuch tekstowy. O argumentach funkcji możesz myśleć jak o parametrach równania. Jeśli chcesz poznać pole powierzchni kwadratu, możesz użyć formuły, która pomnoży długość jego boku przez tę samą długość — wówczas argumentem Twojej formuły będzie długość boku kwadratu. Funkcje, podobnie jak formuły, przyjmują zmienne jako argumenty. W tym przypadku funkcja wyświetli na ekranie przekazany jej argument.

Cudzysłowy mówią kompilatorowi, że chcesz wypisać literał łańcuchowy w takiej postaci, w jakiej jest zapisany, z wyjątkiem sekwencji specjalnych. Ciąg \n jest jedną z takich sekwencji i w rzeczywistości jest traktowany jak jeden znak oznaczający nowy wiersz — podobnie jak naciśnięcie klawisza *Enter* (później opowiem o tym dokładniej). W rezultacie kursor zostanie przeniesiony do nowego wiersza. Czasami zobaczyż, że zamiast znaku nowego wiersza użyta została specjalna wartość endl. Zapisy cout << "Hej" << endl oraz cout << "Hej\n" są w zasadzie równoważne. Słowo endl to skrót od „end line”, czyli „koniec wiersza”, i na jego końcu znajduje się litera L, a nie cyfra 1. Napisanie endl z jedynką na końcu zamiast L jest błędem łatwym do popełnienia, a zatem bądź uważny.

Na koniec zwróć uwagę na jeszcze jeden średnik. Musimy go tu postawić, ponieważ wywołujemy funkcję.

Ostatni nawias klamrowy zamyka funkcję. Powinieneś spróbować skompilować ten program, po czym go uruchomić. Wpisz go w swoim środowisku albo otwórz plik źródłowy, który został dołączony do tej książki. Możesz go przekopiować i wkleić, ale zalecam własnoręczne wpisanie go. Program nie jest długi i pomoże Ci zwrócić uwagę na drobne szczegóły, które mają znaczenie dla kompilatora, takie jak użycie średników.

Jeśli Twój pierwszy program już działa, spróbuj poeksperymentować z funkcją cout, aby przyzwyczaić się do pisania w C++. Wypisz różne teksty albo wiele wierszy — sprawdź, do czego możesz skłonić kompilator.

## Co się dzieje, jeżeli nie możesz zobaczyć swojego programu?

W zależności od systemu operacyjnego oraz kompilatora, z których korzystasz, możesz nie ujrzeć wyniku działania programów dołączonych do tej książki — mogą one bardzo szybko mignąć na ekranie, po czym zostaną zamknięte. Jeśli używasz jednego ze środowisk, które zostały polecone w tej książce, nie powinieneś natknąć się na tego typu problem, ale może on wystąpić, gdy korzystasz z innego środowiska. Jeśli tak się stanie, możesz rozwiązać swój problem, dodaając na końcu programu następujący wiersz:

```
cin.get();
```

Spowoduje on, że przed zakończeniem działania program będzie czekać na naciśnięcie klawisza, dzięki czemu zanim zamkniesi okno, będziesz mógł zobaczyć wynik jego działania.

## Podstawowa struktura programu w C++

Uff, jak na tak krótki program sporo się w nim dzieje. Pomińmy wszystkie szczegóły i spójrzmy na zarys podstawowego programu napisanego w C++:

```
[instrukcje include]
using namespace std;
int main()
{
    [tutaj znajduje się Twój kod];
}
```

Co się stanie, jeśli opuścisz jeden z tych elementów?

Jeżeli pominiesz instrukcję `include` albo `using namespace std`, programu nie uda się skompilować. Jeśli program nie kompiluje się, oznacza to, że kompilator czegoś nie rozumie — prawdopodobnie użyłeś nieprawidłowej składni (na przykład pominąłeś średnik) albo brakuje pliku nagłówkowego. Przyczyny problemów z komplikacją mogą być trudne do określenia, jeżeli dopiero zaczynasz programować. Każde niepowodzenie kompilatora wygeneruje jeden lub więcej **błędów komplikacji**, które wyjaśniają przyczynę tego niepowodzenia. Oto przykład jednego z podstawowych komunikatów błędu kompilatora:

```
error: 'cout' was not declared in this scope
```

Jeżeli zobaczyłeś taki komunikat (błąd: 'cout' nie zostało zadeklarowane w tym zakresie), sprawdź, czy na początku Twojego programu znajduje się instrukcja dołączająca plik `iostream` oraz instrukcja `using namespace std`.

Komunikaty błędów kompilatora nie zawsze będą tak łatwe do zinterpretowania. Jeśli pominiesz średnik, prawdopodobnie zobaczyesz rozmaite rodzaje błędów — zwykle będą one dotyczyć wiersza znajdującego się *poniżej* linii, w której o nim zapomniałeś. Jeśli napotkasz mnóstwo niezrozumiałych błędów, spróbuj spojrzeć na wcześniejszy wiersz i sprawdź, czy zawiera on średnik. Bez obaw, wraz z upływem czasu staniesz się całkiem dobry w interpretowaniu błędów komplikacji i będziesz napotykać ich coraz mniej. Nie przejmuj się, że na początku otrzymujesz ich dużo. Nauka, jak sobie z nimi radzić, to prawie jak obrzęd przejścia!

## Komentowanie programów

Kiedy już uczysz się programowania, powinieneś także dowiedzieć się, jak dokumentować swoje programy (jeśli nie dla kogoś innego, to na własne potrzeby). W tym celu dodajesz do kodu **komentarze**. Ja także bardzo często będę stosować komentarze, aby wyjaśniać przykładowe kody.

Jeżeli poinformujesz kompilator, że pewna sekcja kodu jest komentarzem, kompilator zignoruje ją podczas komplikacji, umożliwiając Ci użycie w komentarzu dowolnego tekstu, którym chcesz opisać rzeczywisty kod. Aby utworzyć komentarz, skorzystaj ze znaków `//`, informujących kompilator, że pozostała część wiersza stanowi komentarz, albo `/*` i `*/`, oznaczających, że komentarzem jest cały blok znajdujący się między nimi.

```
// To jest komentarz jednowierszowy
Ten kod nie jest częścią komentarza

/* To jest komentarz wielowierszowy
Ten wiersz jest częścią komentarza
*/
```

Niektóre środowiska kompilacyjne zmieniają kolor fragmentu z komentarzem, aby ułatwić stwierdzenie, że dany **kod nie podlega wykonaniu** — jest to przykład działania funkcji wyróżniania składni.

Kiedy uczysz się programowania, przydatna będzie możliwość **wykomentowania** fragmentów kodu w celu sprawdzenia, w jaki sposób wpłynie to na wynik działania programu. Wykomentowanie kodu polega na umieszczeniu znaków komentarza przy wierszach, których nie chcesz kompilować. Jeśli na przykład chciałbyś poznać efekt braku instrukcji cout, możesz ją po prostu wykomentować:

#### **Przykładowy kod 4.: hello\_komentarz.cpp**

```
#include <iostream>
using namespace std;
int main ()
{
// cout << "Hej, jestem tutaj! Och, i oczywiście Hello World!\n";
}
```

Upewnij się tylko, czy przez przypadek nie wykomentowałeś potrzebnego kodu!

Jeżeli wykomentujesz kod, który jest wymagany, na przykład plik nagłówkowy, program może się nieprawidłowo kompilować. Jeśli podczas komplikacji napotykaš mnóstwo problemów, spróbuj wykomentować fragmenty programu, które według Ciebie mogą być błędne. Jeżeli program skompiluje się bez tego kodu, będziesz miał pewność, że problemy mają źródło w kodzie, który właśnie wykomentowałeś.

## **Specyfika myślenia programisty. Tworzenie kodu wielokrotnego użycia**

Zróbmy sobie krótką przerwę od składni C++ i porozmawiamy o doświadczeniach związanych z programowaniem. Pamiętam reklamę firmy ubezpieczeniowej, w której myjnja samocho- dowa oddawała klientom pojazdy pokryte pianą. Myjnja myła samochody, ale ich nie spłukiwała<sup>1</sup>. W reklamie tej chodziło o to, że niektóre firmy ubezpieczeniowe formułują polisy w taki sposób, że realizacja wynikających z nich roszczeń jest bardzo trudna, ponieważ wykluczają one wiele okoliczności, co do których wydawałoby się, że powinny zostać uwzględnione.

Reklama ta jest także doskonałą metaforą myślenia, które jest niezbędne podczas programowania. Komputery, podobnie jak myjnja z reklamy, traktują sprawy bardzo dosłownie. Robią tylko i wyłącznie to, co im kažesz, i nie rozumieją Twoich ukrytych intencji. Jeśli powiesz im, aby umyły samochód, zrobią to. Jeżeli chcesz, żeby samochód został także spłukany, lepiej im to powiedz. Poziom wymaganej szczegółowości może początkowo przytłaczać, ponieważ konieczne jest przemyślenie każdego pojedynczego kroku procesu oraz upewnienie się, że żaden z kroków nie został pominięty.

Na szczęście w czasie programowania — kiedy już powiesz komputerowi, aby coś zrobił — możesz daną czynność nazwać, a potem się do niej odwoływać, zamiast w kółko powtarzać te same kroki. Nie jest to tak żmudne, jak może się wydawać — nie musisz bez przerwy robić tego samego. Możesz napisać dokładne instrukcje tylko raz, po czym wykorzystać je ponownie. Już wkrótce zobaczyisz, jak to się robi, gdy zajmiemy się funkcjami.

<sup>1</sup> Możesz zobaczyć tę reklamę tutaj: <http://www.youtube.com/watch?v=QaTxIj7ZeLY>; trwa ona tylko 57 sekund.

## Kilka słów na temat radości i bólu praktyki

Dopiero zacząłeś czytać tę książkę, a już pod koniec tego rozdziału pojawiło się kilka problemów natury praktycznej. Z doświadczenia wiem, że sama nauka programowania przynosi tylko nieco lepsze rezultaty niż rozwiązywanie praktycznych problemów. Przede wszystkim programowanie wymaga przywiązywania wagi do detali i wiem z doświadczenia, że czasem można przeczytać fragment tekstu i pomyśleć „Hej, to ma sens!”, a potem przekonać się, że szczegóły problemu pozostają niejasne. Nie zacznesz dobrze radzić sobie ze składnią C++ i niuansami tego języka bez pisania własnego kodu. Ponieważ nie zawsze łatwo jest wpaść na dobre pomysły konstruowania prostych programów, kiedy dopiero zaczynasz naukę programowania, w większości rozdziałów tej książki zamieściłem przeznaczone dla Ciebie praktyczne zadania. Ich liczba nie przytłacza, dlatego szczerze zachęcam, abyś przed przejściem do kolejnego rozdziału spróbował rozwiązać każde z nich.

Gratulacje! Właśnie skończyłeś poznawanie swojego pierwszego programu, a nawet dowiedziałeś się, w jaki sposób zacząć myśleć jak programista. W rozdziale tym znalazło się mnóstwo materiału, ale pozostawiłem też trochę miejsca dla Ciebie, abyś mógł pobawić się przykładowym kodem i sprawdzić, co można z nim zrobić. W następnym rozdziale dowiesz się więcej na temat interakcji z użytkownikiem, w tym jak wczytywać do programu podawane przez niego informacje.

## Sprawdź się

- 1.** Jaka wartość jest zwracana do systemu operacyjnego po poprawnym zakończeniu działania programu?
  - A. -1
  - B. 1
  - C. 0
  - D. Programy nie zwracają wartości.
- 2.** Jaka jest jedyna funkcja, którą musi zawierać każdy program napisany w C++?
  - A. start ()
  - B. system ()
  - C. main ()
  - D. program ()
- 3.** W jaki sposób oznaczany jest początek i koniec bloku kodu?
  - A. { i }
  - B. -> i <-
  - C. BEGIN i END
  - D. ( i )
- 4.** Jaki znak kończy większość wierszy kodu w C++?
  - A. .
  - B. ;
  - C. :
  - D. '

- 5.** Który z poniższych zapisów stanowi poprawny komentarz?
- A. */\* Komentarz \*/*
  - B. *\*\* Komentarz \*\**
  - C. */\* Komentarz \*/*
  - D. *{ Komentarz }*
- 6.** Który plik nagłówkowy jest potrzebny, aby uzyskać dostęp do instrukcji cout?
- A. stream
  - B. Żaden, instrukcja cout jest dostępna domyślnie.
  - C. iostream
  - D. using namespace std;

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Napisz program, który wyświetli Twoje imię.
- 2.** Napisz program wyświetlający na ekranie wiele wierszy tekstu, z których każdy będzie zawierać imię jednego z Twoich znajomych.
- 3.** Spróbuj wykomentować poszczególne wiersze kodu w pierwszym programie, który razem napisaliśmy, i sprawdź, czy może on się skompilować bez któregoś z nich. Przyjrzyj się wyświetlonym komunikatom o błędach — czy mają one jakiś sens? Czy wiesz, dlaczego błędy te wystąpiły po zmianie kodu?



# 3

## ■ ■ ■ R O Z D Z I A Ł 3

# Interakcja z użytkownikiem. Zapisywanie informacji w zmiennych

---

Wiesz już, jak napisać prosty program wyświetlający informacje wprowadzone przez Ciebie — to jest przez programistę — oraz w jaki sposób opisywać programy za pomocą komentarzy. To całkiem nieźle, ale co zrobić, gdybyś chciał wejść w interakcję z użytkownikiem?

Aby współdziałać z użytkownikiem, powinieneś mieć możliwość przyjmowania **danych wejściowych**, czyli informacji pochodzących spoza programu. W tym celu musisz dysponować jakimś miejscem na przechowanie tych danych. W programach dane wejściowe, a także inne informacje, są zapisywane w **zmiennych**. Istnieje kilka różnych typów zmiennych, które przechowują różne rodzaje informacji (na przykład liczby albo litery). Kiedy mówisz kompilatorowi, że deklarujesz zmienną, powinieneś podać mu jej **typ**, a także nazwę.

Najczęściej używane podstawowe typy zmiennych, z których będziesz korzystać, to `char`, `int` i `double`. Zmienna typu `char` przechowuje pojedynczy znak, zmienne typu `int` przechowują liczby całkowite (czyli liczby bez miejsc dziesiętnych), natomiast zmienne typu `double` przechowują liczby z miejscami dziesiętnymi. Każdy z tych typów jest zarazem słowem kluczowym używanym podczas deklarowania zmiennej.

## Deklarowanie zmiennych w C++

Zanim użyjesz zmiennej, powinieneś poinformować o tym kompilator, deklarując ją (kompilator jest bardzo czuły na punkcie wcześniejszego informowania go o wszystkim). Aby zadeklarować zmienną, należy skorzystać ze składni typ `<zmienna>`; (zwróć uwagę na średnik!).

Oto kilka przykładów deklaracji zmiennych:

```
int liczba_calkowita;  
char litera;  
double liczba_dziesietna;
```

W jednym wierszu można zadeklarować wiele zmiennych tego samego typu. Wszystkie zmienne powinny być rozdzielone przecinkami.

```
int a, b, c, d;
```

Zalecam jednak deklarowanie każdej ze zmiennych w osobnym wierszu, co ułatwia czytanie kodu.

## Korzystanie ze zmiennych

Wiesz już, jak powiedzieć kompilatorowi o zmiennych, ale w jaki sposób z nich korzystać?

To proste — należy skorzystać z instrukcji `cin` w celu pobrania danych wejściowych, operatora wstawiania skierowanego w prawą stronę (`>>`) oraz ze zmiennej, do której chcemy „wstawić” wartość wpisaną przez użytkownika.

Oto przykładowy program ilustrujący użycie zmiennej:

### Przykładowy kod 5.: `czytaj_liczbe.cpp`

```
#include <iostream>

using namespace std;

int main ()
{
    int tojestliczba;
    cout << "Proszę wprowadź liczbę: ";
    cin >> tojestliczba;
    cout << "Wprowadziłeś: " << tojestliczba << "\n";
}
```

Rozłożymy nasz kod na składniki i przeanalizujmy go wiersz po wierszu. Pierwszą część programu już znasz, a zatem skupmy się na ciele funkcji `main`.

```
int tojestliczba;
```

W wierszu tym zadeklarowano zmienną typu całkowitego `tojestliczba`. Następna nowa linia to:

```
cin >> tojestliczba;
```

Funkcja `cin >> tojestliczba` powoduje zapisanie wprowadzonej wartości w zmiennej `tojestliczba`. Zanim liczba zostanie odczytana przez program, użytkownik musi nacisnąć *Enter*.

## Co zrobić, gdy program błyskawicznie kończy działanie?

Jeżeli Twój poprzedni program błyskawicznie kończył działanie i musiałeś użyć instrukcji `cin.get ()`, aby temu przeciwdziałać, może się zdarzyć, że okno z obecnym programem także od razu zostanie zamknięte, nawet jeśli dołączysz wywołanie funkcji `cin.get ()`. Możesz ominąć ten problem, dodając przed instrukcją `cin.get ()` następujący wiersz:

```
cin.ignore ();
```

Funkcja ta odczytuje i pozbywa się znaku — w tym przypadku jest to *Enter* naciśnięty przez użytkownika. Tak! Kiedy użytkownik wprowadza do komputera dane, program odczytuje także ten znak. Nie jest on nam potrzebny i dlatego możemy się go pozbyć. Zazwyczaj powyższy wiersz będzie wymagany tylko wtedy, gdy używasz instrukcji `cin.get ()` w celu wstrzymania działania programu w oczekiwaniu na naciśnięcie klawisza. Bez niego `cin.get ()` wczyta znak nowego wiersza (*Enter*), a działanie Twojego programu zostanie od razu zakończone.

Pamiętaj, że zmienna `tojestliczba` jest zadeklarowana jako liczba całkowita. Jeśli użytkownik wprowadzi liczbę z miejscami dziesiętnymi, zostanie ona **obcięta** (miejscami po przecinku będą

pomijane; na przykład 3,1415 zostanie odczytane jako 3). Po uruchomieniu programu spróbuj wpisać ciąg liter albo liczbę dziesiętną. Reakcja programu będzie zależeć od wprowadzonych danych, ale w żadnym przypadku nie będzie pożądana. Na razie nie zajmiemy się obsługą błędów, która jest potrzebna, aby radzić sobie w takich sytuacjach.

```
cout << "Wprowadziłeś: " << tojestliczba << "\n";
```

W wierszu tym wypisywana jest liczba podana przez użytkownika. Zwróć uwagę, że przy wyświetlanej zmiennej nie użyto znaków cudzysłowu. Gdybyśmy wokół zmiennej tojestliczba zastosowali cudzysłowy, na ekranie pokazałby się napis Wprowadziłeś: tojestliczba. Brak znaków cudzysłowu informuje kompilator, że ma do czynienia ze zmienną i że przed wyświetlением jej na ekranie powinien sprawdzić jej wartość w celu zastąpienia nazwy zmiennej jej treścią.

Przy okazji, nie dziw się dwóm oddzielnym operatorom wstawiania umieszczonym w jednym wierszu. Taki zapis jest jak najbardziej poprawny, a informacje wyjściowe zostaną wyświetcone w odpowiednich miejscach. W rzeczywistości literaly łańcuchowe (czyli napisy zawarte między cudzysłowami) muszą być oddzielane od zmiennych za pomocą operatorów wstawiania (<<). Próba wyświetlenia zmiennej razem z literałem łańcuchowym przy użyciu jednego tylko operatora << wywoła błąd o następującej treści:

#### ZŁY KOD

```
cout << "Wprowadziłeś: " tojestliczba;
```

Tak jak w przypadku wszystkich innych funkcji, wiersz kończy się średnikiem. Jeśli o nim zapomnisz, podczas próby skompilowania programu kompilator wyświetli komunikat informujący o błędzie.

## Zmiana wartości zmiennych oraz ich porównywanie

Odczytywanie i wyświetlanie zmiennych bardzo szybko robi się nudne. Zajmijmy się możliwością modyfikowania zmiennych oraz wpływania na zachowanie programu w zależności od ich wartości. Dzięki temu będziemy mogli w różny sposób reagować na różne dane wprowadzane przez użytkownika.

Zmiennej można przypisać wartość za pomocą **operatora przypisania**, którym jest znak równości =.

```
int x;  
x = 5;
```

Powyższy zapis nadaje zmiennej x wartość 5. Mógłbyś pomyśleć, że znak równości **porównuje** wartości z jego lewej i prawej strony, ale tak nie jest. W C++ do sprawdzania równości jest używany oddzielnny operator, składający się z dwóch znaków równości ==. Ze znaku tego będziesz często korzystać w instrukcjach warunkowych albo pętlach. W kilku kolejnych rozdziałach będziemy używać operatora porównania podczas poznawania sposobów na wybieranie różnych ścieżek działania programu w zależności od danych wprowadzonych przez użytkownika.

a == 5 // NIE przypisuje wartości 5 do zmiennej a, tylko sprawdza, czy zmieniona ta jest równa 5.

Na zmiennych możesz przeprowadzać operacje arytmetyczne.

*	Mnoży dwie wartości
-	Odejmuje jedną wartość od drugiej
+	Dodaje dwie wartości
/	Dzieli jedną wartość przez drugą

Oto kilka przykładów:

```
a = 4 * 6; // Daje w wyniku 24 (zwróć uwagę na użycie średnika oraz komentarza)  
a = a + 5; // Daje w wyniku poprzednią wartość zmiennej a powiększoną o 5
```

## Skrócone zapisy na dodawanie i odejmowanie jedynki

W C++ bardzo często zachodzi potrzeba zwiększenia zmiennej o jeden:

```
int x = 0;  
x = x + 1;
```

Z podobnym wzorcem będziesz mieć do czynienia w dalszej części tej książki, kiedy zaczniemy poznawać takie koncepcje jak pętle. Jest on do tego stopnia powszechny, że nawet istnieje operator, którego jedynym celem jest zwiększenie zmiennej o jeden: `++`.

Pokazany wcześniej kod można zapisać następująco:

```
int x = 0;  
x++;
```

Po wykonaniu powyższych instrukcji zmienna `x` przyjmie wartość 1. Operator ten jest znany pod nazwą operatora **inkrementacji**, a dodanie 1 do zmiennej nazywane jest **inkrementacją** tej zmiennej.

Operator `--` działa podobnie, tylko że odejmuje od zmiennej 1. Jest on znany pod nazwą operatora **dekrementacji**, natomiast odjęcie 1 od zmiennej nazywane jest **dekrementacją** tej zmiennej.

Wiedząc to, możesz domyślić się, skąd pochodzi nazwa C++. C++ bazuje na języku programowania C. C++ znaczy dosłownie „C plus jeden”. C++ to bardziej C z pewnymi dodatkami niż zupełnie nowy język. Myślę, że gdyby twórcy C++ wiedzieli, o ile potężniejszy będzie ten język od C, nadaliby mu nazwę C do kwadratu.

Istnieje podobny, skrócony operator umożliwiający dodanie dowolnej wartości do zmiennej:

```
x += 5; // Dodaje 5 do x
```

Są także operatory odejmowania, mnożenia i dzielenia:

```
x -= 5; // Odejmuje 5 od x  
x *= 5; // Mnoży x przez 5  
x /= 5; // Dzieli x przez 5
```

Operatory `++` i `--` możesz umieszczać nie tylko po zmiennej, ale także przed zmienią:

```
--x;  
++y;
```

Różnica między zapisami „przed” i „po” kryje się w wartości, jaka jest zwracana w danym wyrażeniu. Jeśli napiszesz:

```
int x = 0;
cout << x++;
```

Zostanie wyświetcone 0. Dzieje się tak dlatego, że pomimo zmodyfikowania zmiennej x została zwrócona jej wartość początkowa. Ponieważ operator ++ występuje po zmiennej, zwiększenie jej wartości ma miejsce już po jej odczytaniu.

Jeżeli umieścis operator przed zmienną, uzyskasz nową wartość:

```
int x = 0;
cout << ++x;
```

Zostanie wyświetcone 1, ponieważ najpierw nastąpi zwiększenie zmiennej x o 1, a dopiero później pobranie jej wartości. Znając wszystkie te operacje, możesz napisać w C++ niewielki kalkulator.

### **Przykładowy kod 6.: *kalkulator.cpp***

```
#include <iostream>

using namespace std;

int main()
{
    int pierwszy_argument;
    int drugi_argument;
    cout << "Podaj pierwszy argument: ";
    cin >> pierwszy_argument;
    cout << "Podaj drugi argument: ";
    cin >> drugi_argument;
    cout << pierwszy_argument << " * " << drugi_argument << " = " <<
    ↵pierwszy_argument * drugi_argument << endl;
    cout << pierwszy_argument << " + " << drugi_argument << " = " <<
    ↵pierwszy_argument + drugi_argument << endl;
    cout << pierwszy_argument << " / " << drugi_argument << " = " <<
    ↵pierwszy_argument / drugi_argument << endl;
    cout << pierwszy_argument << " - " << drugi_argument << " = " <<
    ↵pierwszy_argument - drugi_argument << endl;
}
```

## **Poprawne i niepoprawne użycie zmiennych**

### **Najczęściej popełniane błędy podczas deklarowania zmiennych w C++**

Deklarowanie zmiennych daje Ci w programach wiele nowych możliwości, ale niepoprawne zadeklarowanie zmiennej może wywołać szereg problemów. Jeśli na przykład spróbujesz użyć zmiennej, której nie zadeklarowałeś, komplikacja nie powiedzie się i wystąpi błąd dotyczący **niezadeklarowanej zmiennej**. Jeżeli skorzystasz ze zmiennej, która nie była zadeklarowana, kompilator zazwyczaj wygeneruje następujący błąd (w tym przypadku zmienną tą jest x):

```
error: 'x' was not declared in this scope
```

Dokładna treść komunikatu zależy od kompilatora. Powyższy przykład dotyczy kompilatora MinGW oraz środowiska Code::Blocks.

Chociaż może istnieć wiele zmiennych tego samego typu, nie można mieć wielu zmiennych o tej samej nazwie. Nie ma na przykład możliwości utworzenia jednej zmiennej o typie zmiennoprzecinkowym oraz drugiej zmiennej o typie całkowitym, z których obie będą mieć nazwę `moja_wartosc`. Komunikat błędu dotyczący zadeklarowania dwóch zmiennych o tej samej nazwie może wyglądać następująco:

```
error: conflicting declaration 'double moja_wartosc'  
error: 'moja_wartosc' has a previous declaration as 'int moja_wartosc'  
error: declaration of 'double moja_wartosc'  
error: conflicts with previous declaration 'int moja_wartosc'
```

Trzeci najczęściej występujący błąd polega na pominięciu średnika na końcu wiersza:

**ZŁY KOD**  
`int x`

W zależności od instrukcji znajdującej się po deklaracji zmiennej brak średnika może wywołać ze strony kompilatora wiele różnych komunikatów o błędach. Zwykle komunikat będzie się odnosić do wiersza występującego bezpośrednio po deklaracji.

Niektóre błędy nie pojawią się podczas komplikacji, ale w czasie działania programu. Kiedy zadeklarujesz zmienną, pozostaje ona **niezainicjalizowana**. Zanim użyjesz zmiennej, musisz ją **zainicjalizować**. Aby przeprowadzić inicjalizację zmiennej, powinieneś nadać jej wartość. Jeśli tego nie zrobisz, program będzie się zachowywać w sposób nieprzewidywalny. Często spotykany problem wygląda mniej więcej tak:

```
int x;  
int y;  
y = 5;  
x = x + y;
```

W powyższym przykładzie tylko zmiennej `y` nadano wartość 5 przed jej użyciem. Początkowa wartość `x` pozostaje nieznana. Podczas działania programu zostanie ona wybrana losowo, tak więc ostateczna wartość zmiennej `x` może być zupełnie dowolna! Nie należy zakładać, że zmienne są inicjalizowane jakąś wygodną wartością, taką jak na przykład 0.

Jedna z technik, z której możesz korzystać, polega na inicjalizowaniu zmiennych podczas ich deklarowania:

```
int x = 0;
```

Takie rozwiązanie w zupełności wystarczy, aby po utworzeniu zmiennej jej wartość była znana. Przyjęcie takiego nawyku z pewnością pozwoli Ci uniknąć w przyszłości popełnienia wielu paskudnych błędów, a czym wśród przyjaciół jest tych kilka dodatkowych stuknięć w klawisze?

## Rozróżnianie wielkości liter

Mamy teraz dobrą okazję do porozmawiania o kolejnym ważnym zagadnieniu, przez które łatwo możesz się pogubić. Jest nim **rozróżnienie wielkości liter**. W C++ ma znaczenie, czy użyjesz liter wielkich, czy małych. Nazwy `Kot` i `kot` mają dla kompilatora różne znaczenie. We wszystkich słowach kluczowych, nazwach funkcji oraz zmiennych w C++ rozróżniana jest wielkość liter.

Różnica w wielkości liter (na przykład `X` i `x`) między deklaracją zmiennej i miejscami jej użycia jest jednym z powodów, dla których może wystąpić błąd niezadeklarowanej zmiennej, gdy jesteś przekonany, że zmienną jednak zadeklarowałeś.

## Nazwy zmiennych

Wybieranie zrozumiałych i opisowych nazw zmiennych także jest bardzo ważne. Oto przykład złego doboru nazw zmiennych:

```
wart1 = wart2 * wart3;
```

Co to znaczy? Tego nie wie nikt. Nazwy użyte w powyższej instrukcji są praktycznie bezużyteczne. Kiedy tworzysz program, myślisz, że kod, który piszesz, jest dość oczywisty — jesteś o tym przekonany w dniu, w którym programujesz. Następnego dnia ten sam kod będzie dla Ciebie niezrozumiały. Dzięki nadawaniu opisowych nazw zmiennym będziesz mniej zdeorientowany, gdy następnym razem będziesz czytać swój kod. Oto przykład:

```
powierzchnia = szerokosc * wysokosc;
```

Taki zapis jest o wiele czytelniejszy niż pierwsze równanie, i to tylko dzięki zmianie nazw.

## Przechowywanie łańcuchów tekstowych

Zapewne zauważyłeś, że wszystkie wspomniane do tej pory typy zmiennych pozwalają na przechowywanie prostych wartości, na przykład pojedynczej liczby całkowitej albo znaku. Za ich pomocą można całkiem sporo działać, ale C++ oferuje także inne typy danych<sup>1</sup>.

Jednym z najprzydatniejszych typów danych jest typ łańcuchowy **string**. Umożliwia on przechowywanie wielu znaków. Widziałeś już typ łańcuchowy w działaniu, gdy na ekranie był wyświetlany napis:

```
cout << "Hej, jestem tutaj! Och, i oczywiście Hello World!\n";
```

Klasa **string** w C++ umożliwia przechowywanie, modyfikowanie oraz wykonywanie innych działań na łańcuchach tekstowych.

Zadeklarowanie zmiennej łańcuchowej jest proste:

### Przykładowy kod 7.: *string.cpp*

```
#include <string>

using namespace std;

int main ()
{
    string my_string;
}
```

Zwróć uwagę, że w odróżnieniu od innych typów wbudowanych, w celu użycia typu **string** powinieneś dołączyć plik nagłówkowy **<string>**. Jest tak dlatego, że typ **string** nie jest wbudowany bezpośrednio w kompilator, tak jak to jest w przypadku typów całkowitych. Łańcuchy tekstowe są udostępniane poprzez standardową bibliotekę C++, która jest ogromnym źródłem kodu przeznaczonego do wielokrotnego użycia.

Tak samo jak w przypadku pozostałych typów podstawowych dostępnych w C++, łańcuchy tekstowe wpisywane przez użytkownika można wczytywać za pomocą instrukcji **cin**.

---

<sup>1</sup> C++ umożliwia programistom tworzenie własnych typów danych. Powrócimy do tego tematu w dalszej części książki, podczas omawiania struktur.

**Przykładowy kod 8.: string\_imie.cpp**

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string imie_uzytownika;
    cout << "Proszę podać swoje imię: ";
    cin >> imie_uzytownika;
    cout << "Cześć " << imie_uzytownika << "\n";
}
```

Program ten tworzy zmienną łańcuchową, prosi użytkownika o podanie swojego imienia, po czym wyświetla je na ekranie.

Tak jak w przypadku innych zmiennych, zmienne łańcuchowe można inicjalizować wartością początkową:

```
string imie_uzytownika = '<niewiadome>'
```

Jeśli chcesz zestawić ze sobą dwa łańcuchy tekstowe (operacja ta jest nazywana **dolaczaniem**<sup>2</sup> jednego łańcucha do drugiego), możesz skorzystać ze znaku +:

**Przykładowy kod 9.: string\_dolacz.cpp**

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string imie_uzytownika;
    string nazwisko_uzytownika;

    cout << "Proszę podać swoje imię: ";
    cin >> imie_uzytownika;
    cout << "Proszę podać swoje nazwisko: ";
    cin >> nazwisko_uzytownika;
    string pełne_imie_uzytownika = imie_uzytownika + " " + nazwisko_uzytownika;

    cout << "Nazywasz się: " << pełne_imie_uzytownika << "\n";
}
```

Program bierze trzy różne łańcuchy tekstowe: imię użytkownika, pojedynczy odstęp oraz nazwisko użytkownika, i łączy je w jeden łańcuch.

Kiedy wczytujesz łańcuchy tekstowe, czasami chciałbyś pobrać jednorazowo cały wiersz. Istnieje specjalna funkcja, `getline`, której można w tym celu użyć; pozbywa się ona nawet końcowego znaku nowego wiersza.

---

<sup>2</sup> Czasami spotkasz się z terminem **konkatenacja**, oznaczającym łączenie łańcuchów tekstowych. Słowo „konkatenacja” wywodzi się z łacińskiego określenia łączenia za pomocą łańcucha; *catena* w łacinie to łańcuch.

Aby skorzystać z funkcji `getline`, należy przekazać jej źródło danych wejściowych (w tym przypadku `cin`), łańcuch, do którego nastąpi wczytanie, oraz znak, na którym należy zakończyć wczytywanie łańcucha. Na przykład poniższy kod pobierze imię użytkownika:

```
getline( cin, user_first_name, '\n' );
```

Funkcja `getline` jest przydatna również wtedy, gdy chcesz pobrać od użytkownika dane wejściowe tylko do pewnego znaku, na przykład przecinka (choć użytkownik nadal musi naciągnąć *Enter*, zanim program przyjmie dane).

```
getline( cin, moj_lancuch, '.' );
```

Jeśli użytkownik wpisze teraz:

```
Hello, World
```

do zmiennej `moj_lancuch` wejdzie wartość `Hello`. Dalsza część tekstu, w tym przypadku `World`, będzie pozostawać w buforze wejściowym do chwili, w której program odczyta bufor za pomocą następnej instrukcji pobierającej z niego dane.

## No dobrze, rozumiem już łańcuchy tekstowe, ale co z pozostałymi typami?

*Uwaga! Podrozdział ten zawiera zaawansowany materiał, z którego nie musisz jeszcze korzystać. Jeśli jest on dla Ciebie zbyt trudny, nie ma przeszkód, abyś go teraz pominął i powrócił do niego później.*

Być może zastanawiasz się teraz, dlaczego istnieje aż tyle różnych podstawowych typów zmiennych. Poświęćmy więc chwilę na zaznajomienie się z dwoma elementarnymi składnikami wszystkich programów komputerowych, jakimi są **bit** i **bajt**. Bit stanowi podstawową jednostkę pamięci komputera. Jest on przełącznikiem o stanach włączony i wyłączony, przechowującym w zależności od ustawienia wartość 1 lub 0. Bajt składa się z ośmiu bitów. Ponieważ w bajcie znajduje się osiem bitów, istnieje 256 różnych zestawień zer i jedynek. Jest tak, ponieważ istnieje osiem pozycji, z których każda może przyjmować dwie wartości. Omówię to dokładniej. Jeden bit może przechowywać 0 lub 1, czyli dwie wartości. Drugi bit podwaja liczbę możliwości: 00, 01, 10 i 11. Trzeci bit ponownie podwaja liczbę możliwości przez dodanie zera lub jedynki do każdej dwubitowej kombinacji. Każdy bit podwaja zatem liczbę możliwych do przedstawienia wartości. Innymi słowy, dla  $n$  bitów mamy  $2^n$  wartości. Ponieważ bajt zawiera osiem bitów, może on przyjmować  $2^8$  możliwych konfiguracji. Jeśli masz dwa bajty, masz tym samym 16 bitów, co przekłada się na  $2^{16}$  (65 536) wartości.

Nie przejmuj się, jeśli nie zrozumiałeś tego wszystkiego. Najważniejsze to zapamiętać, że im więcej masz bitów, tym większy zakres danych możesz przechowywać.

Na przykład w zmiennej typu `char` można zapisać dane o ograniczonym zakresie — tylko 256 różnych wartości; to pojedynczy bajt. Zmienna typu `integer` składa się zwykle z czterech bajtów, co oznacza, że może ona przedstawiać około cztery miliardy różnych liczb.

Dobry przykład dwóch typów zmiennych, które różnią się między sobą tylko wielkością miejsca zajmowanego w pamięci, stanowi `double` oraz jego mniejszy brat `float`. `Float` był tak naprawdę pierwszym typem, który mógł przechowywać liczby zmiennoprzecinkowe, a samo pojęcie liczby zmiennoprzecinkowej wiąże się z faktem, że przecinek może zmieniać swoje położenie w liczbie. Innymi słowy, przed przecinkiem dziesiątnym mogą znajdować się dwie cyfry, a po

przecinku cztery (12,1234), albo cztery cyfry przed przecinkiem i dwie po nim (1234,12). Nie jesteś ograniczony do konkretnej liczby cyfr przed przecinkiem dziesiętnym i po nim.

Nie martw się, jeżeli nie wszystko z tego do końca rozumiesz — liczby zmiennoprzecinkowe są tak nazywane głównie ze względów historycznych. Zapamiętaj tylko, że nazwa ta odnosi się do liczb z miejscami po przecinku (czyli miejscami dziesiętnymi). Typ `float` zajmuje tylko cztery bajty pamięci i nie może przechowywać tylu różnych wartości, ile typ `double`, który dysponuje ośmioma bajtami. Dawniej, kiedy komputery miały mniej pamięci niż teraz, gra była warta świeczki, a programiści zadawali sobie wiele trudu, aby zaoszczędzić kilka bajtów. W obecnych czasach prawie zawsze lepiej będzie korzystać z typu `double`, chociaż w przypadku ograniczonych zasobów (na przykład w systemach z niewielką pamięcią, takich jak telefony komórkowe) nadal istnieje możliwość użycia typu `float`.

Najmniejszym typem danych jest `char`. Być może zastanawiasz się, dlaczego nadal istnieje ten typ, skoro pamięć nie stanowi już wielkiego problemu. Oto odpowiedź: typ `char` ma pewne specjalne znaczenie — operacje wejścia i wyjścia są realizowane raczej poprzez znaki niż za pośrednictwem liczb. Kiedy dla zmiennej wybierzesz typ `char`, użytkownik będzie mógł wpisać z klawiatury znak, a gdy zechcesz wyświetlić zmienną, cout wypisze znak w niej przechowywany zamiast liczby, która rzeczywiście tam się znajduje. „Moment — zapytasz — co to oznacza? Dlaczego liczby są znakami?”. Odpowiedź na to pytanie jest następująca: gdy komputer przechowuje to, co według nas jest znakiem (takim jak na przykład litera „a”), tak naprawdę zapamiętuje liczbę reprezentującą ten znak. Istnieje tabela zawierająca sparowane liczby oraz znaki, zwana **tabelą ASCII**, która pokazuje, które liczby odpowiadają danym znakom. Kiedy program drukuje znak, nie wyświetla liczby, tylko odszukuje w tabeli ASCII znak, który powinien zostać pokazany<sup>3</sup>.

## Mały sekret liczb zmiennoprzecinkowych

Chciałbym Ci coś wyjawić na temat liczb zmiennoprzecinkowych, takich jak `float` albo `double`. Z pewnością są one bardzo praktyczne, ponieważ mogą przyjmować szeroki zakres wartości. Największa liczba, którą może przedstawić typ `double`, to około  $1,8 \times 10^{308}$  — jest to liczba z 308 zerami na końcu. Jednak typ ten zajmuje tylko 8 bajtów; czy to oznacza, że może on przechowywać tylko  $2^{32}$  (18 446 744 073 709 551 616) możliwych wartości? Taka liczba ma mnóstwo zer, ale nie jest ich 308.

No właśnie! W rzeczywistości typ `double` może reprezentować około 18 trylionów liczb. To bardzo dużo — tak dużo, że aż musiałem sprawdzić, jak nazywa się liczba z 18 zerami. Nadal nie jest to jednak 308 zer. Liczby zmiennoprzecinkowe pozwalają na dokładne przedstawienie tylko niektórych wartości, które są objęte ich zakresem, przez użycie formatu podobnego do notacji naukowej.

W notacji naukowej liczby są zapisywane w postaci  $x \times 10^y$ . Składnik  $x$  przechowuje zazwyczaj kilka pierwszych cyfr liczby, natomiast  $y$  — nazywany **wykładnikiem** — to potęga, do której należy podnieść liczbę. Na przykład odległość Ziemi od Słońca można zapisać jako  $1,496 \times 10^8$  kilometrów (około 150 milionów kilometrów).

<sup>3</sup> Poważnym zaniedbaniem z mojej strony byłoby pominięcie informacji, że tabela ASCII jest raczej mała — zawiera tylko 256 pozycji. Oznacza to, że nie nadaje się ona do pracy z takimi językami jak chiński albo japoński, które dysponują więcej niż 256 znakami. Obejście tego problemu wiąże się z wprowadzeniem koncepcji Unicode, która to tematyka wykracza poza zakres niniejszej książki. Więcej informacji na ten temat znajdziesz na stronie <http://www.cprogramming.com/tutorial/unicode.html>.

Dzięki umieszczeniu dużych wartości w wykładniku taki sposób zapisu pozwala komputerowi na przechowywanie naprawdę dużych liczb. Część niewykładnicza nie może jednak zawierać 300 cyfr, tylko mniej więcej 15. W związku z tym, kiedy pracujesz z liczbami zmiennoprzecinkowymi, masz do dyspozycji tylko 15 cyfr **znaczących**. Jeśli masz do czynienia z względnie małymi wartościami, wówczas różnica między liczbą przechowywaną przez komputer a faktyczną liczbą będzie bardzo niewielka. Jeżeli jednak pracujesz z dużymi wartościami, to błąd bezwzględny także będzie spory, nawet jeśli błąd wzajemny będzie mały. Na przykład przy dwóch liczbach znaczących mogłbym napisać, że Ziemia znajduje się w odległości  $1,5 \times 10^8$  kilometrów od Słońca. Względnie jest to całkiem dobra wartość (błąd jest mniejszy niż 0,1%), ale w mierze bezwzględnej jest to ponad 70 tysięcy kilometrów — prawie dwa razy tyle, ile wynosi obwód Ziemi! Błąd taki ma miejsce oczywiście wtedy, gdy zakładamy użycie dwóch cyfr znaczących. Przyjmując 15 cyfr znaczących, możemy przybliżyć się do liczb tak małych jak na przykład milion.

W większości przypadków niedokładności liczb zmiennoprzecinkowych nie będą mieć większego znaczenia, chyba że przeprowadzasz poważne obliczenia matematyczne lub naukowe.

## Mały sekret liczb całkowitych

Liczby całkowite również mają swój mały sekret. Faktem jest, że liczby całkowite oraz zmiennoprzecinkowe nie mają ze sobą zbyt wiele wspólnego. Liczby całkowite, w przeciwieństwie do zmiennoprzecinkowych, przechowują dokładnie taką wartość, jaka zostanie w nich zapisana, ale szczerze nienawidzą przecinka dziesiętnego. Kiedy przeprowadzasz obliczenia na liczbach całkowitych, a wynikiem nie będzie liczba całkowita, rezultat zostanie obcięty. Część całkowita będzie dokładna, ale część dziesiętna zostanie odrzucona.

Prawdopodobnie nie zaliczyłeś żadnego matematycznego testu, gdybyś napisał, że  $5/2 = 2$ , ale właśnie taki wynik poda komputer! Jeśli potrzebujesz uzyskać odpowiedź z miejscami dziesiętnymi, powinieneś użyć typu niecałkowitego.

Kiedy w swoim programie zapisujesz liczby, kompilator przyjmuje, że są one całkowite, i stąd  $5/2$  daje w wyniku 2. Jeśli umieściszt w liczbie przecinek dziesiętny — na przykład  $5,0/2,0$  — kompilator zinterpretuje taką operację jako działanie na liczbach całkowitych i w rezultacie poda wynik, którego oczekujesz: 2,5.

## Sprawdź się

1. Jakiego typu zmiennej powinieneś użyć, kiedy chcesz zapisać taką liczbę jak 3,1415?
  - A. int
  - B. char
  - C. double
  - D. string
2. Który z poniższych zapisów przedstawia poprawny operator służący do porównywania dwóch zmiennych?
  - A. :=
  - B. =
  - C. equal
  - D. ==

- 3.** W jaki sposób można uzyskać dostęp do danych typu `string`?
  - A. Typ `string` jest wbudowany w język, tak więc nie trzeba nic robić.
  - B. Ponieważ typ `string` jest używany podczas operacji wejścia-wyjścia, należy dołączyć plik nagłówkowy `iostream`.
  - C. Należy dołączyć plik nagłówkowy `string`.
  - D. C++ nie obsługuje łańcuchów tekstowych
- 4.** Który z poniższych typów nie jest poprawnym typem zmiennej?
  - A. `double`
  - B. `real`
  - C. `int`
  - D. `char`
- 5.** W jaki sposób można wczytać cały wiersz wprowadzony przez użytkownika?
  - A. Za pomocą `cin>>`.
  - B. Za pomocą `readline`.
  - C. Za pomocą `getline`.
  - D. Nie da się tego zrobić w prosty sposób.
- 6.** Co zostanie wyświetlone na ekranie w wyniku wykonania następującej instrukcji w C++:  
`cout << 1234/2000?`
  - A. 0
  - B. 0,617
  - C. W przybliżeniu 0,617, ale dokładny wynik nie może być zapisany w liczbie zmiennoprzecinkowej.
  - D. To zależy od typów znajdujących się z obu stron równania.
- 7.** Dlaczego w C++ jest potrzebny typ `char`, skoro istnieją już typy całkowite?
  - A. Ponieważ znaki i liczby całkowite są zupełnie różnymi rodzajami danych; pierwsze z nich to litery, a drugie to liczby.
  - B. Ze względu na wstępную zgodność z C.
  - C. Aby łatwiej było wczytywać oraz wyświetlać znaki zamiast liczb, ponieważ znaki są w rzeczywistości przechowywane jako liczby.
  - D. Ze względu na wsparcie wielojęzykowości, aby możliwa była obsługa takich języków jak chiński albo japoński, w których występuje wiele znaków.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Napisz program, który wyświetli Twoje imię.
- 2.** Napisz program, który wczyta dwie liczby i zsumuje je.
- 3.** Napisz program, który wykonuje dzielenie na dwóch liczbach podanych przez użytkownika i wyświetla dokładny wynik tego działania. Nie zapomnij przetestować swojego programu zarówno dla liczb całkowitych, jak i dziesiętnych.

# ■ ■ ■ R O Z D Z I A Ł 4

## Instrukcje warunkowe

---

Do tej pory dowiedziałeś się, jak napisać program, który przechodzi bezpośrednio od jednej instrukcji do następnej i nie potrafi różnicować swojego zachowania z wyjątkiem wyświetlania różnych wartości obliczanych na podstawie danych wprowadzonych przez użytkownika. **Instrukcja warunkowa if** pozwala decydować, czy program przejdzie do określonej sekcji kodu w zależności od tego, czy został spełniony określony warunek. Innymi słowy, instrukcja ta pozwala programowi podejmować różne działania na podstawie danych uzyskanych od użytkownika. Za pomocą instrukcji warunkowej można na przykład sprawdzić, czy użytkownik podał właściwe hasło, i zdecydować, czy może on uzyskać dostęp do programu.

### Podstawowa składnia instrukcji if

Struktura instrukcji warunkowej if jest prosta:

```
if ( <wyrażenie jest prawdziwe> )
    Wykonaj tę instrukcję
```

Albo:

```
if ( <wyrażenie jest prawdziwe> )
{
    Wykonaj wszystko w tym bloku
}
```

Kod, który następuje po instrukcji if (i jest warunkowo wykonywany), jest nazywany **ciałem** tej instrukcji (tak samo jak kod funkcji main nazywany jest jej ciałem).

Oto prosty (i dość głupawy) przykład składni instrukcji if:

```
if ( 5 < 10 )
    cout << "A teraz 5 jest mniejsze od 10. Co za niespodzianka!"
```

W przykładzie tym sprawdzamy po prostu, czy twierdzenie, że „pięć jest mniejsze od dziesięć” jest prawdziwe, czy nie. Jeśli będziemy mieć szczęście, okaże się, że nie jest! Jeśli chcesz, możesz napisać swój program w pełnej wersji; dołącz do niego plik iostream, umieść poniższy kod w funkcji main oraz uruchom program w celu przetestowania go.

Oto przykład ilustrujący użycie nawiasów klamrowych w celu wykonania wielu instrukcji:

```
if ( 5 < 10 )
{
    cout << "A teraz 5 jest mniejsze od 10. Co za niespodzianka!\n";
    cout << "Mam nadzieję, że ten komputer funkcjonuje poprawnie.\n";
}
```

Jeżeli po instrukcji `if` znajduje się więcej niż jeden wiersz, powinieneś użyć nawiasów klamrowych, aby zagwarantować, że w przypadku spełnienia warunku zostanie wykonany cały blok. Zawsze zalecam zamknięcie między nawiasami ciała instrukcji warunkowej. Jeśli tak zrobisz, nie będziesz musiał pamiętać o dodaniu nawiasów, kiedy będziesz chciał wykonać więcej niż tylko jedną instrukcję, a samo ciało instrukcji warunkowej zostanie w ten sposób lepiej wyróżnione. Częstym błędem jest dodanie do ciała instrukcji `if` drugiego wiersza z pominięciem nawiasów klamrowych, co powoduje, że instrukcja w drugim wierszu wykona się zawsze.

```
if ( 5 < 10 )
    cout << "A teraz 5 jest mniejsze od 10. Co za niespodzianka!\n";
    cout << "Mam nadzieję, że ten komputer funkcjonuje poprawnie.\n";
```

Ze względu na wcięcia dostrzeżenie takiego błędu może być trudne. Bezpieczniej będzie zawsze korzystać z nawiasów klamrowych.

Instrukcje `if`, którym się do tej pory przyglądaliśmy, były dość nudne. Rzućmy okiem na instrukcję `if` z prawdziwego zdarzenia, która działa na podstawie danych wprowadzonych przez użytkownika.

### Przykładowy kod 10.: *zmienna.cpp*

```
#include <iostream>

using namespace std;

int main ()
{
    int x;
    cout << "Podaj liczbę: ";
    cin >> x;
    if ( x < 10 )
    {
        cout << "Twoja liczba jest mniejsza od 10" << '\n';
    }
}
```

Powyższy program różni się od programu z poprzedniego przykładu tym, że porównywana wartość nie została umieszczona w kodzie na stałe, ale jest wprowadzana przez użytkownika. Powinniśmy być podekscytowani takim rozwiązaniem, ponieważ po raz pierwszy uzyskaliśmy program, którego działanie w znacznym stopniu zależy od tego, co zrobi użytkownik. Spójrzmy tymczasem na elastyczność instrukcji `if`.

## Wyrażenia

Instrukcja `if` sprawdza pojedyncze wyrażenie. **Wyrażenie** to stwierdzenie lub kilka powiązanych ze sobą stwierdzeń, których wynik sprowadza się do pojedynczej wartości. W większości elementów, które przyjmują zmienne lub stałe wartości (na przykład liczby), można umieszczać także wyrażenia. W rzeczywistości zarówno wartości zmienne, jak i stałe są tak naprawdę prostymi wyrażeniami. Takie działania jak dodawanie albo mnożenie to tylko nieco bardziej złożone formy wyrażeń. Kiedy zostaną one użyte w kontekście porównania (jak to jest w przypadku instrukcji `if`), wynik wyrażenia jest zamieniany w prawdę albo fałsz.

## Czym jest prawda?

Dla poetów piękno jest prawdą, prawda pięknem i to wszystko, co wiedzieć trzeba<sup>1</sup>. Kompiatory jednak nie są poetami. Dla kompilatora wyrażenie jest **prawdziwe**, jeśli jego wartość jest różna od zera. Wynikiem wyrażenia **falszywego** jest zero, w związku z czym wyrażenie:

```
if ( 1 )
```

spowoduje, że ciało instrukcji if zawsze zostanie wykonane, natomiast wyrażenie:

```
if ( 0 )
```

spowoduje, że ciało instrukcji if nigdy nie zostanie wykonane.

W C++ istnieją specjalne słowa kluczowe, true i false, oznaczające, odpowiednio: prawdę i fałsz, które można umieszczać bezpośrednio w kodzie. Gdybyś chciał wyświetlić wartość całkowitą odpowiadającą słowu true, byłoby to 1, z kolei wartością odpowiadającą słowu false jest oczywiście 0.

Kiedy dokonujesz porównania za pomocą **operatora relacyjnego**, zwróci on prawdę albo fałsz. Na przykład porównanie `0 == 2` da w wyniku fałsz (zwróć uwagę, że porównanie wymaga użycia dwóch znaków równości; pojedynczy znak równości powoduje przypisanie wartości do zmiennej). Porównanie `2 == 2` da w wyniku prawdę. W instrukcji if nie na potrzeby sprawdzania, czy operacja porównania daje w wyniku prawdę czy fałsz; zapis:

```
if ( x == 2 )
```

jest równoważny z:

```
if ( ( x == 2 ) == true )
```

Do tego pierwsza wersja jest łatwiejsza do odczytania!

Podczas programowania często będziesz musiał sprawdzać, czy jedna wartość jest większa, mniejsza, czy też równa drugiej wartości.

Oto tabela z operatorami relacyjnymi, które umożliwiają porównywanie dwóch wartości:

>	większe od	$5 > 4$ daje w wyniku prawdę
<	mniejsze od	$4 < 5$ daje w wyniku prawdę
$\geq$	większe lub równe	$4 \geq 4$ daje w wyniku prawdę
$\leq$	mniejsze lub równe	$3 \leq 4$ daje w wyniku prawdę
$\equiv$	równe	$5 \equiv 5$ daje w wyniku prawdę
$\neq$	różne od	$5 \neq 4$ daje w wyniku prawdę

## Typ bool

C++ umożliwia przechowywanie wyników porównań przy użyciu typu `bool`<sup>2</sup>. Typ ten nie różni się znacznie od typu `integer`, ale ma jedną zaletę: daje jasno do zrozumienia, że będziesz korzystać tylko z dwóch możliwych wartości, którymi są `true` i `false`. Te słowa kluczowe oraz typ

<sup>1</sup> [http://wiersze.wikia.com/wiki/Oda\\_do\\_greckiej\\_urny](http://wiersze.wikia.com/wiki/Oda_do_greckiej_urny)

<sup>2</sup> Nazwa `bool` pochodzi od nazwiska Georga Boole'a, który stworzył logikę boolowską — logikę bazującą wyłącznie na wartościach prawda i fałsz; stanowi ona fundament technologii cyfrowej.

bool sprawiają, że Twoje intencje są wyraźniejsze. Wynik działania wszystkich operatorów relacyjnych również jest boolowski.

```
int x;
cin >> x;
bool czy_x_wynosi_dwa = x == 2; // Zwróć uwagę na podwójny znak równości w porównaniu

if ( czy_x_wynosi_dwa )
{
    // Wykonaj jakieś działanie, bo x wynosi dwa!
}
```

## Instrukcja else

W wielu przypadkach będziesz chciał, aby Twój program przeprowadził jedno porównanie, a następnie wykonał pewne czynności, jeśli wynikiem porównania jest prawda (na przykład hasło podane przez użytkownika jest poprawne), a inne czynności, jeśli porównanie dało w rezultacie fałsz (hasło było nieprawidłowe).

Instrukcja `else` umożliwia dokonywanie porównań typu „jeśli... w przeciwnym razie”. Kod następujący po `else` (pojedynczy wiersz albo kod zamknięty między nawiasami klamrowymi) zostanie wykonany, gdy warunek sprawdzany w instrukcji `if` okaże się niespełniony. Oto przykładowy kod sprawdzający, czy użytkownik podał liczbę ujemną:

### Przykładowy kod 11.: *niewujemna.cpp*

```
#include <iostream>

using namespace std;

int main()
{
    int liczba;
    cout << "Podaj liczbę: ";
    cin >> liczba;
    if ( liczba < 0 )
    {
        cout << "Twoja liczba jest ujemna\n";
    }
    else
    {
        cout << "Twoja liczba jest nieujemna\n";
    }
}
```

## Instrukcje else-if

Kolejne zastosowanie instrukcji `else` ma miejsce wtedy, gdy istnieje wiele wyrażeń warunkowych, które mogą dać w wyniku prawdę, ale chcesz, aby zostało wykonane ciało tylko jednej instrukcji `if`. Możesz na przykład zmodyfikować poprzedni kod w taki sposób, aby wykrywał trzy różne przypadki: liczby ujemne, dodatnie oraz zero. W tym celu można użyć instrukcji `else-if`, która następuje po wcześniejszej instrukcji `if` z jej ciałem. Dzięki temu jeśli pierwszy warunek zostanie spełniony, instrukcja `else-if` zostanie pominięta; jeżeli jednak warunek nie zostanie spełniony, wówczas nastąpi sprawdzenie warunku w instrukcji `else-if`. W przy-

padku, gdy warunek zapisany w tej instrukcji `if` zostanie spełniony, sprawdzenie warunku w części `else` nie odbędzie się. W celu zagwarantowania, że zostanie wykonany tylko jeden blok instrukcji, można użyć serii instrukcji `else-if`.

Oto, jak można zmienić poprzedni kod, aby za pomocą instrukcji `else-if` sprawdzać wystąpienie zera:

### **Przykładowy kod 12.: `else_if.cpp`**

```
#include <iostream>

using namespace std;

int main()
{
    int liczba;
    cout << "Podaj liczbę: ";
    cin >> liczba;
    if ( liczba < 0 )
    {
        cout << "Twoja liczba jest ujemna\n";
    }
    else if ( liczba == 0 )
    {
        cout << "Twoja liczba to zero\n";
    }
    else
    {
        cout << "Twoja liczba jest dodatnia\n";
    }
}
```

## **Porównywanie łańcuchów tekstowych**

Obiekty łańcuchowe w C++ umożliwiają stosowanie wobec nich wszystkich operatorów porównania, które poznałeś w tym rozdziale. Dzięki porównywaniu łańcuchów tekstowych możesz napisać program sprawdzający hasło!

### **Przykładowy kod 13.: `haslo.cpp`**

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string hasło;
    cout << "Podaj hasło: " << "\n";
    getline( cin, hasło, '\n' );
    if ( hasło == "xyzzy" )
    {
        cout << "Dostęp przyznany" << "\n";
    }
    else
    {
        cout << "Niepoprawne hasło. Odmowa dostępu!" << "\n";
        // Zwrócenie wartości to wygodny sposób na zatrzymanie wykonywania programu
    }
}
```

```
        return 0;
    }
    // Pracujemy dalej!
}
```

Program wczytuje wiersz od użytkownika i porównuje go z hasłem "xyzzy". Jeśli wprowadzony wiersz nie jest taki sam jak hasło, program bezzwłocznie wyjdzie z funkcji `main`<sup>3</sup>.

W stosunku do łańcuchów tekstowych można używać także pozostałych operatorów relacyjnych, takich jak operatory porównania, aby stwierdzić, który łańcuch jest wcześniejszy w kolejności alfabetycznej, lub operator `!=`, aby sprawdzić, czy dwa łańcuchy różnią się między sobą.

## Więcej interesujących warunków budowanych za pomocą operatorów logicznych

Do tej pory mogliśmy sprawdzić tylko jeden warunek naraz. Jeśli chciałbyś sprawdzić dwa elementy, takie jak poprawność hasła oraz właściwa nazwa użytkownika, musiałbyś napisać jakąś dziwną instrukcję `if/else`. Na szczęście C++ udostępnia możliwość jednoczesnego sprawdzania wielu warunków dzięki zastosowaniu **operatorów boolowskich** (nazwa ta ma związek z typem boolowskim, który został omówiony wcześniej; operatory boolowskie działają na wartościach boolowskich), zwanych też operatorami logicznymi.

Operatory boolowskie umożliwiają tworzenie złożonych instrukcji warunkowych. Jeśli na przykład chcesz sprawdzić, czy zmienna `wiek` jest jednocześnie większa od 5 oraz mniejsza od 10, możesz użyć logicznego ORAZ, aby zagwarantować, że oba wyrażenia `wiek > 5` i `wiek < 10` będą prawdziwe.

Operatory logiczne działają podobnie jak operatory porównania, zwracając prawdę albo fałsz w zależności od wyniku wyrażenia.

### Logiczne nie

**Logiczne nie** (negacja) przyjmuje jeden argument. Jeśli na wejściu pojawi się prawda, operator zwróci fałsz, a jeśli na wejściu znajdzie się fałsz, zostanie zwrocona prawda. Na przykład wynikiem wyrażenia `nie(prawda)` jest fałsz, a wynikiem `nie(fałsz)` jest prawda. Wyrażenie `nie` (dowolna liczba oprócz 0) daje w wyniku fałsz.

Symbolem logicznego NIE w C++ jest `!` (wykrzyknik).

Na przykład:

```
if ( ! 0 )
{
    cout << "! 0 daje w wyniku prawdę";
```

---

<sup>3</sup> Oczywiście żaden kod sprawdzający hasła nie jest aż tak prosty. Z pewnością nie chciałbyś na stałe umieszczać hasła w kodzie źródłowym!

## Logiczne ORAZ

**Logiczne ORAZ** zwraca prawdę, jeśli oba argumenty wejściowe są prawdziwe (jeśli „to” ORAZ „tamto” są prawdziwe). Wyrażenie prawda ORAZ fałsz daje w wyniku fałsz, ponieważ jeden z argumentów jest fałszywy (aby wynikiem była prawda, oba argumenty muszą być prawdziwe); prawda ORAZ prawda daje w wyniku prawdę, natomiast (dowolna liczba oprócz 0) ORAZ fałsz daje w wyniku fałsz.

Operator ORAZ jest zapisywany w C++ jako `&&`. Nie powinieneś myśleć, że sprawdza on, czy między dwoma liczbami zachodzi równość — sprawdza tylko, czy oba argumenty są prawdziwe.

```
if ( 1 && 2 )
{
    cout << "Zarówno 1, jak i 2 dają w wyniku prawdę";
```

## Skrócone wartościowanie

Jeśli pierwsze wyrażenie logicznego ORAZ jest fałszywe, wówczas drugie wyrażenie nie będzie obliczane. Innymi słowy, stosowane jest **skrócone wartościowanie**.

Skrócone wartościowanie jest przydatne, ponieważ możesz dzięki niemu pisać wyrażenia, w których drugi warunek powinien zostać sprawdzony tylko wtedy, gdy pierwszy warunek jest prawdziwy, na przykład aby zapobiec dzieleniu przez zero. Spójrzmy na poniższą instrukcję `if`, która sprawdza, czy 10 podzielone przez `x` jest mniejsze od 2:

```
if ( x != 0 && 10 / x < 2 )
{
    cout << "10 / x jest mniejsze od 2";
```

Podczas wykonywania instrukcji `if` program najpierw sprawdza, czy `x` jest różne od zera. Jeśli `x` wynosi zero, nie ma potrzeby sprawdzania drugiego warunku i zostanie on pominięty. Dzięki temu nie musisz obawiać się, że dzielenie przez zero spowoduje awarię Twojego programu. Bez możliwości skróconego wartościowania musiałbyś napisać taki kod:

```
if ( x != 0 )
{
    if ( 10 / x < 2 )
    {
        cout << "10 / x jest mniejsze od 2";
    }
}
```

Skrócone wartościowanie sprawia, że można pisać bardziej przejrzysty i zwięzły kod.

## Logiczne LUB

**Logiczne LUB** zwraca prawdę, jeśli jedna z dwóch lub obie wartości są prawdziwe. Na przykład wyrażenie prawda LUB fałsz daje w wyniku prawdę, a wyrażenie fałsz LUB fałsz daje w wyniku fałsz. Logiczne LUB jest zapisywane w C++ jako `||`. Znaki te to symbole potoku. Na klawiaturze komputera może on wyglądać jak pionowa kreska z przerwą w środku, chociaż na ekranie będzie wyświetlany jako kreska bez przerwy. Na wielu klawiaturach symbol potoku znajduje się na tym samym klawiszku co znak `\` i wymaga naciśnięcia klawisza `Shift`.

Podobnie jak logiczne ORAZ, logiczne LUB podlega skróconemu wartościowaniu — jeśli pierwszy warunek jest spełniony, drugi warunek nie zostanie sprawdzony.

## Łączenie wyrażeń

Z pomocą operatorów logicznych można jednocześnie sprawdzać dwa warunki. A gdybyś tak chciał mieć jeszcze większe możliwości? Czy pamiętasz, w jaki sposób można tworzyć wyrażenia ze zmiennych, operatorów i wartości? Wyrażenia można konstruować także z innych wyrażeń.

Możesz na przykład sprawdzić, czy  $x$  wynosi 2, a  $y$  3, łącząc operatory porównania z logicznym operatorem ORAZ:

```
x == 2 && y == 3
```

Spójrzmy na przykład zastosowania operatorów logicznych w programie sprawdzającym zarówno nazwę użytkownika, jak i jego hasło.

### Przykładowy kod 14.: *uzytkownik\_haslo.cpp*

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string nazwa;
    string haslo;
    cout << "Podaj nazwę użytkownika: " << "\n";
    getline( cin, nazwa, '\n' );

    cout << "Podaj hasło: " << "\n";
    getline( cin, haslo, '\n' );
    if ( nazwa == "admin" && haslo == "xyzzy" )
    {
        cout << "Dostęp przyznany" << "\n";
    }
    else
    {
        cout << "Niepoprawna nazwa użytkownika lub hasło. Odmowa dostępu!" << "\n";
        // Zwrócenie wartości to wygodny sposób na zatrzymanie programu
        return 0;
    }
    // Pracujemy dalej!
}
```

Po uruchomieniu program ten przyzna dostęp wyłącznie użytkownikowi o nazwie admin, który dysponuje właściwym hasłem. Za pomocą instrukcji `else-if` łatwo możesz rozszerzyć program o kolejnych użytkowników, z których każdy będzie mieć własne hasło.

## Kolejność wykonywania działań

W języku C++ operatory mają swoje **priorytety**, które decydują o kolejności wykonywania działań. W przypadku operatorów arytmetycznych (+, -, / i \*) ich priorytety są takie same jak w zwykłej matematyce; mnożenie i dzielenie są wykonywane przed dodawaniem i odejmowaniem.

Spośród operatorów logicznych najpierw wykonywana jest negacja, po której przeprowadzane są porównania. Logiczne ORAZ opracowywane jest przed logicznym LUB.

W poniższej tabeli przedstawiono kolejność wykonywania działań dla operatorów logicznych oraz operatorów porównania.

!
<code>==, &lt;, &gt;, &lt;=, &gt;=, !=</code>
<code>&amp;&amp;</code>
<code>  </code>

Aby mieć kontrolę nad kolejnością opracowywania operatorów logicznych oraz arytmetycznych, takich jak dodawanie i odejmowanie, zawsze można użyć nawiasów.

Powróćmy do naszego wcześniejszego przykładu:

```
x == 2 && y == 3
```

Gdybyś chciał zastosować warunek „jeśli powyższe wyrażenie NIE jest prawdziwe”, mógłbyś skorzystać z nawiasów:

```
! ( x == 2 && y == 3 )
```

## Przykładowe wyrażenia logiczne

Rzućmy okiem na kilka bardziej skomplikowanych wyrażeń logicznych, dzięki którym przekonasz się, czy dobrze zrozumiałeś operatory boolowskie.

Jaki wynik da następujące wyrażenie?

```
! ( true && false )
```

Wynikiem będzie fałsz. Jest tak, ponieważ wynikiem wyrażenia `true && false` jest fałsz, natomiast `! false` daje w wyniku prawdę.

Oto trzy kolejne wyrażenia; ich wyniki zostały podane w przypisach:

```
! ( true || false )4
```

```
! ( true || true && false )5
```

```
! ( ( true || false ) && false )6
```

## Sprawdź się

1. Które z poniższych wyrażeń są prawdziwe?

- A. 1
- B. 66
- C. 0,1
- D. -1
- E. Każde z powyższych.

<sup>4</sup> Fałsz.

<sup>5</sup> Fałsz (ORAZ jest opracowywane przed LUB).

<sup>6</sup> Prawda.

- 2.** Który z poniższych operatorów jest w C++ boolowskim ORAZ?
  - A. &
  - B. &&
  - C. |
  - D. ||
- 3.** Jaką wartość ma wyrażenie `! ( true && ! ( false || true ) )`?
  - A. Prawda.
  - B. Fałsz.
- 4.** Który z poniższych zapisów instrukcji if ma prawidłową składnię?
  - A. if *wyrażenie*
  - B. if { *wyrażenie*
  - C. if ( *wyrażenie* )
  - D. *wyrażenie* if

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Poproś użytkownika o podanie wieku dwóch osób i wskaż, która z nich jest starsza.  
Jeśli obie osoby mają powyżej 100 lat, program powinien zachować się w szczególny sposób.
- 2.** Zaimplementuj prosty system weryfikacji haseł, który pobiera hasła w postaci liczb.  
Ważne powinno być jedno z dwóch haseł, ale w celu ich sprawdzenia użyj tylko jednej instrukcji if.
- 3.** Napisz niewielki kalkulator, który pobiera na wejściu jeden z operatorów arytmetycznych oraz dwa argumenty, po czym wyświetla wynik obliczeń otrzymany na podstawie tych danych.
- 4.** Rozszerz program kontrolujący haseła, który został zamieszczony w tym rozdziale, w taki sposób, aby akceptował wielu użytkowników, z których każdy ma swoje hasło. Zagwarantuj, aby właściwe haseła były przypisane właściwym użytkownikom. Udostępnij możliwość ponownego zalogowania użytkownika, jeśli pierwsza próba nie powiodła się. Zastanów się, jak łatwo (albo trudno) można zrealizować taką funkcjonalność w przypadku dużej liczby użytkowników i haseł.
- 5.** Pomyśl o tym, jakie elementy lub funkcje języka ułatwiłyby dodawanie nowych użytkowników bez potrzeby ponownej komplikacji programu weryfikującego hasła (nie czuj się zmuszony do rozwiązania tego problemu za pomocą funkcji C++, które poznajesz do tej pory; możesz powrócić do tego zadania, gdy poznasz narzędzia, które omówię w kolejnych rozdziałach).

# 5

## ■ ■ ■ R O Z D Z I A Ł 5

## Pętle

---

Z poprzednich rozdziałów wiesz już, jak pisać programy, które zachowują się w różny sposób w zależności od danych wprowadzonych przez użytkownika, z tym że działanie tych programów było jednoprzebiegowe. Na razie nie potrafisz napisać programu, który wciąż na nowo prosi użytkownika o kolejne dane. Jeśli zabrałeś się za zadania praktyczne z poprzedniego rozdziału i pracowałeś nad programem weryfikującym hasła, który prosi użytkownika o ponowne zalogowanie się w przypadku podania błędnego hasła, prawdopodobnie musiałeś wpisać w kodzie serię instrukcji `if` sprawdzających hasło po raz kolejny. Nie było sposobu na umożliwienie użytkownikowi wprowadzania haseł, dopóki podaje on niepoprawne dane logowania.

Do tego właśnie służą pętle, które wielokrotnie wykonują blok kodu. Pętle są potężnym narzędziem i tworzą zasadnicze części większości programów. Liczne programy oraz strony internetowe, które prezentują bardzo złożone informacje (takie jak fora dyskusyjne), w rzeczywistości wykonują wiele razy jedno zadanie. Pomyśl, co to może oznaczać: pętla umożliwia napisanie bardzo prostej instrukcji, która przyniesie wspaniałe wyniki tylko dlatego, że jest powtarzana. Możesz prosić użytkownika o hasło tyle razy, ile tylko zechce on je podać. Możesz wyświetlić tysiące postów na forum internetowym. Wspaniale!

W C++ istnieją trzy rodzaje pętli, z których każda służy nieco innym celom: `while`, `for` i `do-while`. Zajmiemy się po kolei każdą z nich.

### Pętla while

**Pętla while** jest najprostszym rodzajem pętli. Jej podstawowa struktura wygląda następująco:

```
while ( <warunek> ) { Kod do wykonania, gdy warunek jest spełniony }
```

W rzeczywistości pętla `loop` bardzo przypomina instrukcję `if`, z tym że jej ciało jest wykonywane wielokrotnie. Tak jak w przypadku instrukcji `if`, warunek jest wyrażeniem logicznym. Tak może wyglądać pętla `loop` z dwoma warunkami:

```
while ( i == 2 || i == 3 )
```

A oto przykład podstawowej pętli `loop`:

```
while ( true )
{
    cout << "Powtarzam pętle\n";
}
```

**Uwaga!** Jeśli uruchomisz taką pętlę, nigdy się nie zatrzyma. Jej warunek zawsze będzie spełniony. Taka pętla nazywa się **pętlą nieskończoną**. Ponieważ pętla nieskończona nigdy się nie zatrzyma,

będziesz musiał ręcznie przerwać działanie programu (za pomocą klawiszy *Ctrl+C*, *Ctrl+Break* albo zamykając okno konsoli). Aby uniknąć pętli nieskończonych, musisz mieć pewność, że ich warunki nie zawsze będą spełnione.

## Najczęściej popełniany błąd

Nadarzył się dobry moment, aby wskazać, że częstą przyczyną pętli nieskończonej jest użycie w warunku pętli pojedynczego znaku równości zamiast znaku podwójnego:

### ZŁY KOD

```
int i = 1;
while ( i = 1 )
{
    cin >> i;
}
```

Pętla ta powinna wczytywać wartości wprowadzane przez użytkownika do chwili, w której poda on liczbę różną od 1. Niestety, warunek pętli wygląda następująco:

*i = 1*

zamiast:

*i == 1*

Wyrażenie *i = 1* przypisze po prostu zmiennej *i* wartość 1. Jak się okazuje, wyrażenie przypisujące zwraca także przypisywaną wartość, którą w tym przypadku jest 1. Ponieważ 1 jest różne od zera, pętla będzie powtarzana w nieskończoność.

Spójrzmy teraz na pętlę, która działa prawidłowo. Oto program, który demonstruje funkcjonowanie pętli *loop*, wyświetlając liczby od 0 do 9:

### Przykładowy kod 15.: *while.cpp*

```
#include <iostream>

using namespace std;

int main ()
{
    int i = 0; // Nie zapomnij o zadeklarowaniu zmiennej

    while ( i < 10 ) // Dopóki i jest mniejsze od 10
    {
        cout << i << '\n';
        i++; // Zaktualizuj i, aby warunek został w końcu spełniony
    }
}
```

Jeśli masz problem z połapaniem się w działaniu tej pętli, pomyśl o niej następująco: kiedy program dochodzi do klamry na końcu ciała pętli, powraca na jej początek, gdzie ponownie sprawdza warunek i decyduje, czy powtórzyć blok jeszcze raz, czy też go pominąć i przejść do pierwszej instrukcji znajdującej się po bloku.

## Pętla for

Pętla *for* jest uniwersalna i bardzo wygodna w użyciu. Jej składnia wygląda następująco:

```

for (inicjalizacja zmiennej; warunek; aktualizacja zmiennej )
{
    // Wykonywany kod, jeśli warunek został spełniony
}

```

W pętli for wiele się dzieje. Spójrzmy na krótki przykład takiej pętli i omówmy każdy z jej elementów. Poniższa pętla for zachowuje się dokładnie tak jak pętla while, którą niedawno poznaliśmy.

```

for (int i = 0; i < 10; i++)
{
    cout << i << '\n';
}

```

## Inicjalizacja zmiennej

Inicjalizacja zmiennej, w tym przypadku  $i = 0$ , umożliwia zadeklarowanie zmiennej i nadanie jej wartości (lub nadanie wartości istniejącej już zmiennej). W naszym przykładzie deklarujemy zmienną  $i$ . Kiedy w pętli sprawdzana jest wartość jednej zmiennej, czasami jest ona nazywana **zmienną pętli**. W programowaniu do oznaczania zmiennych pętli tradycyjnie używa się liter  $i$  oraz  $j$ . Zmienna, której wartość w każdym przebiegu pętli jest zwiększana o 1, nazywana jest **licznikiem pętli**, ponieważ odlicza ona kolejne wartości, począwszy od zadanej wielkości początkowej.

## Warunek pętli

Warunek pętli informuje program, że pętla powinna być powtarzana, dopóki wartością wyrażenia warunkowego jest prawda (tak samo jak w przypadku pętli while). W naszym programie sprawdzamy, czy  $x$  jest mniejsze od 10. Podobnie jak w przypadku pętli while, warunek jest sprawdzany przed wykonaniem ciała pętli, a następnie po każdym jej przebiegu w celu podjęcia decyzji, czy pętla powinna zostać powtórzona.

## Aktualizacja zmiennej

Sekcja aktualizacji zmiennej to miejsce, w którym można aktualizować wartość zmiennej. Można w niej dokonywać takich operacji jak  $i++$ ,  $i = i + 10$  albo wywołać funkcję. Jeśli zechcesz, możesz wywoływać funkcje, które nic nie robią z samą zmienną, ale wpływają na kod w jakiś inny przydatny sposób.

Ponieważ w tak wielu pętlach występuje jedna zmienna, jeden warunek i jedna sekcja aktualizacji zmiennej, pętla for stanowi wygodny sposób na pisanie zwartych pętli, w których wszystkie istotne elementy mieszczą się w jednym wierszu.

Zwróć uwagę, że w wierszu tym do rozdzielenia poszczególnych sekcji należy użyć średników i nie można ich pominąć. Każda z sekcji może być pusta, ale średniki należy pozostawić na swoich miejscach. Jeśli warunek będzie pusty, zostanie on uznany za prawdziwy i pętla będzie powtarzana, aż przerwie ją jakieś inne zdarzenie — jest to następny sposób na napisanie pętli nieskończonej.

Aby zrozumieć, kiedy realizowana jest każda z sekcji pętli for, porównajmy ją z pętlą while wykonującą to samo zadanie, którą już poznaliśmy w tym rozdziale:

```

int i = 0; // Deklaracja oraz inicjalizacja zmiennej
while ( i < 10 ) // Warunek
{
    cout << i << '\n';
    i++; // Aktualizacja zmiennej
}

```

Pętla for to po prostu bardziej zwarty sposób na realizację tego samego zadania.

Spójrzmy na jeszcze jeden przykład pętli for, która robi coś bardziej interesującego niż tylko wypisanie serii liczb. Oto pełna wersja programu drukującego na ekranie kwadraty wszystkich liczb całkowitych od 0 do 9:

### Przykładowy kod 16.: **for.cpp**

```

#include <iostream>

using namespace std;

int main ()
{
    // Pętla jest wykonywana, dopóki i < 10; i zwiększa się przy każdym przebiegu pętli
    for ( int i = 0; i < 10; i++ )
    {
        // Pamiętaj, że warunek pętli jest sprawdzany
        // przed kolejnym wykonaniem pętli. W związku
        // z tym pętla zatrzyma się, gdy i będzie równe 10.
        // Zmienna i jest aktualizowana przed sprawdzeniem
        // warunku.
        cout << i << " do kwadratu to " << i * i << endl;
    }
}

```

Program ten to bardzo prosty przykład pętli for. Aby dobrze zrozumieć, kiedy wykonywane są poszczególne jej części, przeanalizujmy go:

1. Uruchamiana jest sekcja inicjalizacyjna: zmienna i otrzymuje wartość 0.
2. Sprawdzany jest warunek. Ponieważ i jest mniejsze od 10, wykonywane jest ciało pętli.
3. Wykonywana jest aktualizacja zmiennej i; jest ona zwiększana o 1.
4. Sprawdzany jest warunek — jeśli nie jest spełniony, pętla zostanie zakończona.
5. Jeżeli warunek jest spełniony, ciało pętli zostanie wykonane i wszystkie czynności będą powtarzane począwszy od kroku 3, dopóki zmienna i będzie mniejsza od 10.

Pamiętaj, że aktualizacja zmiennej ma miejsce tylko po wykonaniu pętli. Nie jest ona przeprowadzana przed uruchomieniem ciała pętli for po raz pierwszy.

## Pętla do-while

**Pętla do-while** ma szczególne zastosowanie i jest spotykana dość rzadko. Głównym jej celem jest uproszczenie stosowania ciał pętli, które powinny wykonać się przynajmniej raz. Struktura pętli do-while wygląda następująco:

```

do
{
    // Ciało...
} while (warunek );

```

Warunek nie jest sprawdzany na początku ciała pętli, tylko na jej końcu. W związku z tym ciało pętli zostanie wykonane przynajmniej raz. Jeśli warunek jest spełniony, nastąpi przejście na początek bloku i ponowne wykonanie ciała pętli. Pętla do-while jest w zasadzie odwróconą wersją pętli while. Można wyobrazić sobie, że pętla while mówi „Powtarzaj pętlę, dopóki warunek jest spełniony, wykonując przy tym następujący blok kodu”. Tymczasem pętla do-while mówi „Wykonaj następujący blok kodu, a następnie, jeśli warunek jest spełniony, powróć na początek pętli”. Oto przykładowy program pozwalający użytkownikowi na wprowadzenie hasła:

### **Przykładowy kod 17.: dowhile.cpp**

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string haslo;
    do
    {
        cout << "Proszę podać hasło: ";
        cin >> haslo;
    } while ( haslo != "foobar" );
    cout << "Witaj, podałeś poprawne hasło";
}
```

Pętla wykona swoje ciało przynajmniej raz, umożliwiając użytkownikowi podanie hasła. Jeżeli hasło nie jest prawidłowe, pętla zostanie powtórzona i użytkownik zostanie poproszony o ponowne wprowadzenie hasła — i tak aż do chwili, w której poda on właściwe hasło.

W powyższym przykładzie zwrócić uwagę na średnik kończący wiersz z instrukcją while. Łatwo zapomnieć o jego dodaniu, ponieważ w innych pętlach nie jest on wymagany. W rzeczywistości pozostałych pętli w ogóle **nie należy** kończyć średnikiem, co tylko przyczynia się do zwiększenia całego zamieszania.

## **Kontrolowanie przebiegu pętli**

Chociaż decyzja o wyjściu z pętli jest zwykle podejmowana na podstawie sprawdzenia jej warunku, czasem zachodzi potrzeba wcześniejszego opuszczenia pętli. W C++ istnieje słowo kluczowe break, którego można w tym celu użyć. Instrukcja break powoduje natychmiastowe zakończenie działania każdej pętli, która jest właśnie wykonywana.

W poniższym przykładzie użyto instrukcji break w celu zakończenia pętli, która w innym przypadku wykonywałaby się w nieskończoność. Jest to inna wersja kodu sprawdzającego hasło z poprzedniego przykładu.

### **Przykładowy kod 18.: break.cpp**

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string haslo;
```

```
while ( 1 )
{
    cout << " Proszę podać hasło: ";
    cin >> hasło;
    if ( hasło == "foobar" )
    {
        break;
    }
}
cout << "Witaj, podałeś poprawne hasło";
```

Instrukcja `break` powoduje natychmiastowe zakończenie pętli i przejście do klamry zamkającej. W naszym przykładzie pętla zakończy się po wprowadzeniu prawidłowego hasła. Ponieważ instrukcja `break` może pojawić się w dowolnym miejscu pętli — w tym pod sam jej koniec — możesz korzystać z pętli nieskończonych jako alternatywnego sposobu na tworzenie pętli `do-while`, tak jak to właśnie zrobiliśmy. W naszym przykładzie `break` faktycznie pełni rolę sprawdzania warunku znajdującego się na końcu pętli `do-while`.

Instrukcje `break` są przydatne, kiedy szukasz sposobu na wyjście z obszernej pętli, chociaż zbyt wiele takich instrukcji powoduje, że kod staje się mało czytelny.

Druga metoda kontrolowania przebiegu pętli polega na opuszczeniu jednej iteracji za pomocą słowa kluczowego `continue`. Kiedy program natrafi na instrukcję `continue`, bieżąca iteracja pętli zostanie zakończona wcześniej, ale nie nastąpi wyjście z samej pętli. Możesz na przykład napisać pętlę, która pomija wyświetlanie liczby 10:

```
int i = 0;
while ( true )
{
    i++;
    if ( i == 10 )
    {
        continue;
    }
    cout << i << "\n";
}
```

Pętla w powyższym przykładzie nigdy się nie skończy, ale kiedy `i` osiągnie 10, instrukcja `continue` spowoduje przejście programu na początek pętli, co spowoduje pominięcie wywołania funkcji `cout`. Jednak warunek zakończenia pętli nadal będzie sprawdzany. Kiedy w pętli zostanie użyta instrukcja `continue`, etap aktualizacji zmiennej nastąpi bezpośrednio po jej wykonaniu.

Instrukcja `continue` jest najbardziej przydatna wtedy, gdy chcesz pominąć część kodu wewnątrz ciała pętli. Możesz na przykład sprawdzić dane wejściowe wprowadzone przez użytkownika, a gdy są one niepoprawne, pominąć ich przetwarzanie, korzystając z następującej pętli:

```
while ( true )
{
    cin >> dane_wejściowe;
    if ( ! sprawdzDane( dane_wejściowe ) )
    {
        continue;
    }
    // Przejdź do przetwarzania danych w normalnym trybie
}
```

# Pętle zagnieżdżone

Bardzo często zdarza się, że chcesz przetwarzać w pętli nie tylko jedną wartość, ale dwie różne, chociaż powiązane ze sobą wartości. Być może będziesz chciał wydrukować listę postów z forum dyskusyjnego (jedna pętla), a dla każdego z nich szereg różnych informacji, takich jak temat postu, jego autor oraz treść. Dane te można pobrać w drugiej pętli. Ta druga pętla musi jednak wykonywać się w pierwszej pętli — raz dla każdej wiadomości. Tego typu pętle są nazywane **pętlami zagnieżdżonymi**, ponieważ jedna pętla jest umieszczona wewnątrz innej pętli.

Spójrzmy na prosty przykład, który nie wymaga złożoności potrzebnej do obsługi forum dyskusyjnego. Wydruk tabliczki mnożenia doskonale nadaje się do realizacji za pomocą zagnieżdżonych pętli:

## Przykładowy kod 19.: zagniezdzzone\_petle.cpp

```
#include <iostream>

using namespace std;

int main ()
{
    for ( int i = 0; i < 10; i++ )
    {
        // Symbol \t reprezentuje znak tabulacji, który
        // spowoduje czytelne sformatowanie wydruku
        cout << '\t' << i;
    }
    cout << '\n';

    for ( int i = 0; i < 10; ++i )
    {
        cout << i;

        for ( int j = 0; j < 10; ++j )
        {
            cout << '\t' << i * j;
        }
        cout << '\n';
    }
}
```

W przypadku pętli zagnieżdżonych można mówić o **pętli zewnętrznej** oraz **pętli wewnętrznej**, co pozwala je od siebie odróżnić. W naszym przykładzie pętla ze zmienną *j* jest pętlą wewnętrzną, natomiast pętla, którą ją zawiera — pętla ze zmienną *i* — jest pętlą zewnętrzna.

Uważaj tylko, aby nie stosować tej samej zmiennej zarówno do pętli wewnętrznej, jak i zewnętrznej:

**ZŁY KOD**

```
for ( int i = 0; i < 10; i++ )
{
    // Ups, przypadkowo ponownie zadeklarowałem i tutaj!
    for ( int i = 0; i < 10; i++ )
    {
    }
}
```

Można zagnieźdzać więcej niż dwie pętle. Możesz utworzyć tyle poziomów zagnieżdżenia, ile tylko potrzebujesz — pętlę w pętli, w pętli, w pętli — pętle aż do samego końca!

## Wybór właściwego rodzaju pętli

Poznałeś zatem trzy różne rodzaje pętli w C++. Zapewne zastanawiasz się, po co. Do czego w ogóle są potrzebne trzy rodzaje pętli?

Tak naprawdę nie potrzebujesz wszystkich trzech rodzajów pętli. Pętle `do-while` częściej spotykam w podręcznikach niż w rzeczywistym kodzie. O wiele częściej używane są pętle `for` oraz `while`.

Poniżej zamieszczam kilka krótkich wskazówek dotyczących wyboru właściwego rodzaju pętli. To tylko ogólne zasady. Z upływem czasu uzyskasz lepsze wyczucie względem tego, co sprawia, że określony rodzaj pętli lepiej się sprawdzi w danym kodzie. Nie powinieneś wtedy pozwolić, aby wskazówki te wzięły góre nad Twoim doświadczeniem.

### Pętla for

Użyj pętli `for`, jeśli dokładnie wiesz, ile razy należy powtórzyć pętlę. Pętla `for` idealnie nadaje się na przykład do liczenia od 1 do 100 albo podczas generowania tabliczek mnożenia. Pętle `for` są także standardowo stosowane do iterowania po tablicach, o czym dowiesz się, gdy zajmiemy się tą strukturą danych (rozdział „Tablice”). Z drugiej jednak strony, nie użyjesz pętli `for`, jeśli zmienność musi być aktualizowana w bardzo skomplikowany sposób. Kiedy wszystko na temat funkcjonowania pętli można pokazać w jednej zwartej instrukcji, wówczas pętla `for` znakomicie spełni swoje zadanie. Jeżeli etap aktualizacji wymaga wielu wierszy kodu, stracisz przewagę, jaką daje pętla `for`.

### Pętla while

Jeśli warunek pętli jest skomplikowany albo musisz dokonać wielu obliczeń, aby uzyskać kolejną wartość zmiennej pętli, zastanów się nad użyciem pętli `while`. Pętle `while` bardzo ułatwiają dostrzeżenie momentu, w którym kończą swoje działanie, ale trudniej w nich zauważać, co ulega zmianie podczas każdego ich przebiegu. Jeśli aktualizacja zmiennej jest skomplikowana, lepiej wybierz pętlę `while`, gdyż osoba czytająca Twój kod dowie się przynajmniej, że obliczenie nowej wartości zmiennej nie jest proste.

Spójrzmy na przykład, w którym występują dwie różne zmienne pętli:

```
int j = 5;
for ( int i = 0; i < 10 && j > 0; i++ )
{
    cout << i * j;
    j = i - j;
}
```

Zwróć uwagę, że nie wszystko, co jest istotne, zmieściło się w jednym wierszu z instrukcją pętli. Jedna z aktualizacji ma miejsce pod koniec ciała pętli. W ten sposób czytelnik kodu może zostać wprowadzony w błąd. W takim przypadku lepiej będzie użyć pętli `while`.

```

int i = 0;
int j = 5;
while ( i < 10 && j > 0 )
{
    cout << i * j;
    j = i - j;
    i++;
}

```

Kod nie wygląda elegancko, ale przynajmniej nie prowadzi na manowce.

Pętle while idealnie sprawdzają się także w sytuacji, w której pętla ma się powtarzać prawie w nieskończoność, na przykład w programie szachowym, gdzie każda ze stron wykonuje swoje ruchy aż do zakończenia rozgrywki.

## Pętla do-while

Jak już wspomniałem, pętle do-while są używane bardzo rzadko. Jedynym powodem ich użycia jest konieczność zrobienia czegoś przynajmniej raz. Dobry przykład stanowi pokazany już wcześniej kod, który prosi użytkownika o podanie hasła, lub w ogólności dowolny rodzaj interfejsu użytkownika, który wymaga wprowadzenia danych i wielokrotnie prezentuje prośbę o ich wpisanie aż do chwili, w której uzyska poprawne informacje. W niektórych jednak przypadkach, nawet gdy ciało pętli ma się powtarzać, do-while może nie być najlepszym wyborem, gdy w pierwszym przebiegu pętli kod ma być nieco inny; na przykład jeśli komunikat powinien mieć inną treść w przypadku podania błędnego hasła za pierwszym razem.

W jaki sposób napisałbyś taki program:

```

string haslo;

cout << "Podaj hasło: ";

cin >> haslo;
while ( haslo != "xyzzy" )
{
    cout << "Niepoprawne hasło - spróbuj ponownie: ";
    cin >> haslo;
}

```

korzystając z pętli do-while?

```

string haslo;

do
{
    if ( haslo == "" )
    {
        cout << "Podaj hasło: ";
    }
    else
    {
        cout << "Niepoprawne hasło - spróbuj ponownie: ";
    }
    cin >> haslo;
} while ( haslo != "xyzzy" );

```

Widzisz, jak użycie pętli `do-while` skomplikowało program, zamiast go uprościć? Rzecz w tym, że ciało pętli nie jest takie samo. Nawet jeśli wczytujemy od użytkownika te same dane, musimy wyświetlić mu różne komunikaty.

## Sprawdź się

- 1.** Jaka będzie ostateczna wartość zmiennej `x` po uruchomieniu kodu `x; for( x=0; x<10; x++ ) {}?`
  - A. 10
  - B. 9
  - C. 0
  - D. 1
- 2.** Kiedy zostanie wykonany blok następujący po instrukcji `while ( x<100 )?`
  - A. Kiedy `x` będzie mniejsze od 100.
  - B. Kiedy `x` będzie większe od 100.
  - C. Kiedy `x` będzie równe 100.
  - D. Kiedy zechce.
- 3.** Która z poniższych instrukcji nie tworzy struktury pętli?
  - A. `for`
  - B. `do-while`
  - C. `while`
  - D. `repeat until`
- 4.** Ile razy na pewno wykona się pętla `do-while`?
  - A. 0
  - B. Nieskończoność wiele razy.
  - C. 1
  - D. To zależy.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Napisz program, który drukuje pełny tekst piosenki *99 Bottles of Beer*<sup>1</sup>.
- 2.** Napisz program udostępniający menu, które pozwala użytkownikowi dokonać wyboru spośród różnych opcji. Jeśli odpowiedź udzielona przez użytkownika nie jest zgodna z żadną z opcji, wyświetl menu jeszcze raz.
- 3.** Napisz program obliczający sumę narastającą liczb wprowadzanych przez użytkownika, który zakończy swoje działanie, gdy użytkownik wprowadzi 0.
- 4.** Napisz program służący do weryfikacji haseł, który daje użytkownikowi tylko kilka szans na podanie poprawnego hasła, dzięki czemu użycie łamacza haseł będzie trudne.

---

<sup>1</sup> Jeśli nie znasz tej piosenki, jej słowa znajdziesz na stronie [http://pl.wikipedia.org/wiki/99\\_Bottles\\_of\\_Beer](http://pl.wikipedia.org/wiki/99_Bottles_of_Beer).

- 5.** Spróbuj rozwiązać powyższe zadania, korzystając z wszystkich rodzajów pętli. Zwróć uwagę, która pętla sprawdza się najlepiej w każdym z zadań.
- 6.** Napisz program wyświetlający pierwszych 20 liczb kwadratowych.
- 7.** Napisz program udostępniający opcję sumowania wyników ankiety, w której mogą wystąpić trzy różne wartości. Dane wejściowe wprowadzane do programu to pytanie ankietowe oraz trzy możliwe odpowiedzi. Pierwszej odpowiedzi przypisywana jest wartość 1, drugiej 2, a trzeciej 3. Odpowiedzi są sumowane do chwili, w której użytkownik wprowadzi 0 — wtedy program powinien pokazać wyniki ankiety. Postaraj się wygenerować wykres słupkowy, pokazujący wyniki przeskalowane w taki sposób, aby zmieściły się na ekranie bez względu na liczbę udzielonych odpowiedzi.



## Funkcje

---

Teraz, kiedy poznałeś już pętle, masz możliwość pisania wielu interesujących programów. Niestety, Twoje programy muszą w całości zawierać się w funkcji `main`. Jeśli będziesz chciał zrobić w niej coś skomplikowanego, stanie się ona wyjątkowo duża i trudna do zrozumienia. Być może już zwróciłeś na to uwagę, pracując nad rozwiązyaniem bardziej złożonych problemów z poprzednich rozdziałów. Co więcej, natrafisz na sytuacje, w których będziesz chciał wykonać to samo zadanie w wielu różnych miejscach programu, w związku z czym przyjdzie Ci wielokrotnie kopować i wklejać ten sam kod.

W takich właśnie sytuacjach przydatne są **funkcje**. Dzięki rozbiciu programu na funkcje będziesz mógł korzystać z zawartego w nich kodu w różnych miejscach programu bez konieczności kopiowania go i wklejania. Tak naprawdę to używałeś już kilku standardowych funkcji podczas wykonywania operacji wejścia i wyjścia.

Większość zagadnień, które poznałeś do tej pory, wiązała się z robieniem nowych rzeczy; funkcje polegają na organizowaniu elementów, ułatwiając wielokrotne ich używanie i poprawiając czytelność Twojego kodu.

### Składnia funkcji

Widziałeś już, jak tworzyć funkcje; przecież każdy Twój program zawiera w sobie funkcję `main`!

Abyśmy mieli o czym porozmawiać, weźmiemy pod uwagę inną funkcję i rozłożymy ją na poszczególne części:

```
int dodaj (int x, int y)
{
    return x + y;
}
```

No dobrze, co takiego się tu dzieje? Najpierw zwróć uwagę na to, że wygląda ona bardzo podobnie do funkcji `main`, którą pisałeś już kilkanaście razy. Istnieją między nimi jedynie dwie zasadnicze różnice:

1. Powyższa funkcja przyjmuje dwa argumenty — `x` oraz `y`. Funkcja `main` nie przyjmowała żadnych argumentów.
2. Funkcja ta jawnie zwraca wartość (zapewne pamiętasz, że funkcja `main` także zwraca wartość, ale nie musisz w niej osobiście umieszczać instrukcji `return`).

W wierszu

```
int dodaj ( int x, int y )
```

najpierw podany jest typ zwracanej wartości — jeszcze przed nazwą funkcji. Jej dwa argumenty są wymienione po nazwie. Jeżeli funkcja nie przyjmuje argumentów, należy po prostu umieścić po jej nazwie parę nawiasów:

```
int funkcja_bez_arg ()
```

Jeśli potrzebna jest funkcja, która nie zwraca żadnej wartości, na przykład funkcja drukująca coś na ekranie, możesz zadeklarować zwracany przez nią typ jako void. W ten sposób nie pozwolisz na użycie takiej funkcji w wyrażeniu (takim jak przypisanie wartości zmiennej albo sprawdzenie warunku w instrukcji if).

Wartość zwracana zostanie udostępniona za pomocą instrukcji return. Cała nasza funkcja składa się z tylko jednego wiersza:

```
return x + y;
```

Może jednak zawierać ona więcej linii, tak samo jak funkcja main. Funkcja zatrzyma swoje działanie po wykonaniu instrukcji return, zwracającej wartość do elementu, który ją wywołał.

Kiedy już zadeklarujesz swoją funkcję, będziesz mógł ją wywołać w następujący sposób:

```
dodaj( 1, 2 ); // Zignoruj zwracaną wartość
```

Możesz także użyć funkcji w wyrażeniu, aby przypisać wynik jej działania zmiennej albo wydrukować ten wynik na ekranie:

### Przykładowy kod 20.: funkcja\_dodaj.cpp

```
#include <iostream>

using namespace std;

int dodaj ( int x, int y )
{
    return x + y;
}

int main ()
{
    int wynik = dodaj( 1, 2 ); // Wywołaj funkcję dodaj i przypisz wynik do zmiennej
    cout << "Pierwszy wynik to " << wynik << '\n';
    cout << "Wynikiem dodawania 3 i 4 jest " << dodaj( 3, 4 );
}
```

W powyższym przykładzie mogłoby się zdawać, że polecenie cout wydrukuje nazwę funkcji dodaj, ale tak samo jak to się dzieje w przypadku zmiennych, cout spowoduje wyświetlenie wyniku wyrażenia zamiast frazy dodaj(3, 4). Efekt działania programu będzie taki sam, jak gdybyśmy uruchomili następujący wiersz kodu:

```
cout << "Wynikiem dodawania 3 i 4 jest " << 3 + 4;
```

W powyższym programie zauważ, że zamiast powtarzać ten sam kod w różnych miejscach, wywołujemy kilka razy funkcję dodaj. W przypadku tak krótkiej funkcji nie stanowi to szczególnej zalety, ale jeśli później zdecydujemy się na rozszerzenie funkcji dodaj o nowy kod (na przykład o instrukcje drukujące argumenty i wyniki funkcji na czas debugowania programu), przyjdzie nam poddać zmianom o wiele mniej kodu, niż gdybyśmy musieli modyfikować wszystkie miejsca, w których występuje powielony kod.

# Zmienne lokalne i zmienne globalne

Teraz, kiedy możesz mieć już więcej niż tylko jedną funkcję, prawdopodobnie będziesz mieć do czynienia z większą liczbą zmiennych. Niektóre z nich mogą znajdować się nawet w każdej funkcji. Zajmijmy się przez chwilę nazwami, jakie nadajesz zmiennym. Kiedy deklarujesz zmienną wewnątrz funkcji, w jakiś sposób ją nazywasz. W których miejscach programu możesz korzystać z tej nazwy, aby odwoływać się do danej zmiennej?

## Zmienne lokalne

Spójrzmy na pewną prostą funkcję:

```
int dodajDziesiec (int x)
{
    int wynik = x + 10;
    return wynik;
}
```

Występują w niej dwie zmienne — `x` oraz `wynik`. Najpierw porozmawiamy o zmiennej `wynik`. Jest ona dostępna wyłącznie w obrębie nawiasów klamrowych, między którymi została zdefiniowana, czyli zasadniczo w dwóch wierszach funkcji `dodajDziesiec`. Oznacza to, że mógłbyś napisać kolejną funkcję zawierającą zmienną `wynik`:

```
int pobierzDziesiec ()
{
    int wynik = 10;
    return wynik;
}
```

Moglibyś nawet użyć funkcji `pobierzDziesiec` z poziomu funkcji `dodajDziesiec`:

```
int dodajDziesiec (int x)
{
    int wynik = x + pobierzDziesiec();
    return wynik;
}
```

Istnieją dwie różne zmienne o nazwie `wynik`; jedna z nich należy do funkcji `dodajDziesiec`, a druga do `pobierzDziesiec`. Nie ma między nimi konfliktu, ponieważ funkcja `pobierzDziesiec` w chwili, gdy się wykonuje, ma dostęp wyłącznie do swojej kopii zmiennej `wynik`. Ta sama zasada dotyczy funkcji `dodajDziesiec`.

Widoczność zmiennej w zależności od miejsca w programie nazywana jest jej **zasięgiem** lub **zakresem**. Zasięg zmiennej oznacza po prostu sekcję kodu, w której można korzystać z jej nazwy w celu uzyskania do niej dostępu. Zmienne deklarowane w funkcji są dostępne wyłącznie w jej zasięgu — wówczas, gdy funkcja ta jest wykonywana. Zmienna zadeklarowana w zasięgu jednej funkcji nie będzie dostępna dla innych funkcji, które zostaną wywołane podczas działania pierwszej funkcji. Gdy jedna funkcja wywoła drugą, dostępne będą wyłącznie zmienne z tej drugiej funkcji.

Argumenty funkcji również są deklarowane w jej zasięgu. Zmienne te nie są dostępne dla obiektów wywołujących daną funkcję, pomimo że przekazują im one wartości. Zmienna `x` funkcji `dodajDziesiec` jest jej argumentem i może być używana tylko w obrębie tej funkcji. Co więcej, tak samo jak w przypadku każdej innej zmiennej zadeklarowanej w ramach funkcji, zmienna `x` nie może być używana przez obiekt, który wywołuje funkcję `dodajDziesiec`.

Argumenty funkcji w pewnym sensie pełnią rolę dublerów zmiennych przekazywanych do funkcji. Zmiana wartości argumentu funkcji nie ma wpływu na oryginalną zmienną. Aby możliwe było zachowanie tej wartości, podczas przekazywania zmiennej do funkcji jest ona kopiwana do argumentu funkcji:

### Przykładowy kod 21.: *zmienna\_lokalna.cpp*

```
#include <iostream>

using namespace std;

void zmienArgument (int x)
{
    x = x + 5;
}

int main()
{
    int y = 4;
    zmienArgument( y ); // y nie zostanie zmienione przez wywołanie funkcji
    cout << y; // i nadal ma wartość 4
}
```

Zasięg zmiennej może być nawet mniejszy niż cała funkcja. Każda para nawiasów klamrowych definiuje nowy, węższy zakres, na przykład:

```
int podziel (int dzielna, int dzielnik)
{
    if ( 0 == dzielnik )
    {
        int wynik = 0;
        return wynik;
    }
    int wynik = dzielna / dzielnik;
    return wynik;
}
```

Pierwsza deklaracja zmiennej `wynik` znajduje się w zasięgu ograniczonym nawiasami klamrowymi instrukcji `if`. Zakres drugiej deklaracji zmiennej `wynik` sięga od miejsca, w którym została zadeklarowana, do końca funkcji. Zwykle kompilator nie powstrzyma Cię od utworzenia dwóch zmiennych o tej samej nazwie, jeśli tylko ich zasięgi będą różne. W przypadkach takich jak w przykładzie z funkcją `podziel`, stosowanie w podobnych zakresach wielu zmiennych o tej samej nazwie może utrudnić czytanie kodu osobie, która będzie próbowała go później zrozumieć.

Każda zmienna zadeklarowana w zasięgu funkcji lub wewnętrz bloku nazywana jest **zmienną lokalną**. Możesz także tworzyć zmienne, które będą dostępne w szerszym zakresie — tak zwane zmienne globalne.

## Zmienne globalne

Czasami chciałbyś mieć do dyspozycji zmienną, która będzie dostępna we wszystkich funkcjach programu. Na przykład plansza w grze logicznej mógłaby być przechowywana jako zmienna globalna, dzięki czemu liczne funkcje z niej korzystające nie musiałyby bez przerwy przekazywać jej sobie nawzajem.

Taki stan rzeczy można osiągnąć, posługując się zmienną globalną. **Zmienna globalna** to zmienna, która została zadeklarowana poza wszystkimi funkcjami. Zmienna taka jest dostępna w każdym punkcie programu, począwszy od miejsca, w którym znajduje się jej deklaracja.

Oto prosty przykład programu ze zmienną globalną pokazujący, jak ją deklarować i jak można z niej korzystać.

### **Przykładowy kod 22.: zmienna\_globalna.cpp**

```
#include <iostream>

using namespace std;

int zrobCos () // Mała funkcja demonstrująca zakres
{
    return 2 + 3;
}

// Zmienne globalne można inicjalizować tak samo jak wszystkie inne zmienne
int liczba_wywolan_funkcji = 0;

void fun ()
{
    // Zmienna globalna jest tu dostępna
    liczba_wywolan_funkcji++;
}

int main ()
{
    fun();
    fun();
    fun();
    // Zmienna globalna jest dostępna także tutaj!
    cout << "Funkcja fun została wywołana " << liczba_wywolan_funkcji << " razy";
}
```

Zasięg zmiennej `liczba_wywolan_funkcji` zaczyna się tuż przed funkcją `fun`. Funkcja `zrobCos` nie ma do niej dostępu, ponieważ zmienna ta została zadeklarowana już po funkcji `zrobCos`, natomiast funkcje `fun` i `main` mogą po nią sięgać, ponieważ obie zostały zadeklarowane po zmiennej.

## **Ostrzeżenie dotyczące zmiennych globalnych**

Mogliby się wydawać, że zmienne globalne upraszczają sprawy, ponieważ wszyscy mogą z nich korzystać. Korzystanie ze zmiennych globalnych utrudnia jednak zrozumienie kodu. Aby zrozumieć, jak zmienna globalna jest używana, musiałbyś zająrzyć dosłownie wszędzie! Rzadko zdarza się, że używanie zmiennych globalnych jest właściwym rozwiązaniem. Powinieneś z nich korzystać wyłącznie wtedy, gdy naprawdę potrzebujesz elementu, który ma być szeroko dostępny. Zamiast jednak udostępniać funkcjom zmienne globalne, lepiej przekazywać do funkcji argumenty. Nawet jeśli sądzisz, że określony element będzie używany globalnie, może się później okazać, że wcale tak nie jest.

Weźmy na przykład wspomnianą wcześniej planszę. Możesz podjąć decyzję o utworzeniu funkcji wyświetlającej planszę, która korzysta ze zmiennej globalnej. Co jednak się stanie, jeśli

zechcesz wyświetlić inną planszę niż bieżącą, na przykład w celu pokazania alternatywnego posunięcia? Twój kod nie przyjmuje planszy jako argumentu, wyświetla tylko jedną, globalną planszę. Nie jest to zbyt wygodne rozwiązań.

## Przygotowanie funkcji do użycia

Reguły dotyczące zasięgu, które odnoszą się do zmiennych — takie jak dostępność zmiennej dopiero po jej deklaracji — mają zastosowanie również do funkcji (czyż konsekwencja nie jest czymś wspaniałym?).

Na przykład taki program nie skompiluje się:

### Przykładowy kod 23.: *zly\_kod.cpp*

#### ZŁY KOD

```
#include <iostream> // Potrzebne dla instrukcji cout

using namespace std;

int main ()
{
    int wynik = dodaj( 1, 2 );
    cout << "Pierwszy wynik to " << wynik << '\n';
    cout << "Wynikiem dodawania 3 i 4 jest " << dodaj( 3, 4 );
}

int dodaj (int x, int y)
{
    return x + y;
}
```

Jeśli spróbujesz skompilować ten program, zobaczysz następujący (albo podobny) komunikat błędu:

```
zly_kod.cpp: 8: error: 'dodaj' was not declared in this scope
```

Problem polega na tym, że w miejscu, w którym następuje wywołanie funkcji dodaj, nie była ona jeszcze zadeklarowana, a zatem znajdowała się poza zasięgiem. Kiedy kompilator widzi, że próbujesz wywołać funkcję, która nie została zadeklarowana, zaczyna się gubić — biedaczysko!

Jedno z rozwiązań, z którego korzystałem we wcześniejszych przykładach, polega na umieszczeniu funkcji przed miejscami, w których jest ona używana. Inne wyjście to **zadeklarowanie** funkcji, zanim zostanie ona **zdefiniowana**.

Chociaż słowa „deklaracja” i „definicja” brzmią podobnie, mają one zupełnie inne znaczenia. Zajmijmy się teraz ich wyjaśnieniem.

## Definicja i deklaracja funkcji

Definiowanie funkcji oznacza napisanie pełnej wersji tej funkcji, łącznie z jej ciałem. Na przykład sposób, w jaki utworzyliśmy funkcję dodaj, jest równoważny z jej zdefiniowaniem, ponieważ od razu pokazujemy, co funkcja ta robi. Definicja funkcji także **jest** jej deklaracją, gdyż do zdefiniowania funkcji potrzebne są również wszystkie informacje, jakie podaje deklaracja.

Deklarowanie funkcji powoduje przekazanie kompilatorowi podstawowych informacji na jej temat, które są potrzebne obiektom wywołującym tę funkcję. Zanim jakikolwiek element będzie mógł ją wywołać, funkcja musi zostać zadeklarowana — bądź w postaci samej deklaracji, bądź też za pomocą pełnej definicji.

W celu zadeklarowania funkcji należy napisać jej **prototyp**. Deklaracja funkcji informuje kompilator, jaki typ wyniku ona zwróci, jaka będzie jej nazwa i jakie argumenty będą jej przekazywane. Prototyp funkcji można traktować jak instrukcję jej obsługi.

```
zwracany_typ nazwa_funkcji (typ_arg arg1, ..., typ_arg argN);
```

W powyższym zapisie typ\_arg oznacza po prostu typ argumentu funkcji, na przykład int, double albo char. W dokładnie taki sam sposób zadeklarowałbyś zmienną.

Rzućmy okiem na taki prototyp funkcji:

```
int dodaj (int x, int y)
```

Prototyp ten określa, że funkcja dodaj będzie przyjmować dwa argumenty, oba całkowite, i że będzie zwracać wartość całkowitą. Średnik informuje kompilator, że nie ma do czynienia z pełną definicją funkcji, tylko z jej prototypem. Postaraj się być miły dla kompilatora i nie zapominaj o średniku kończącym wiersz; w przeciwnym razie kompilator może się pogubić.

## Przykład użycia prototypu funkcji

Spójrzmy na poprawioną wersję wcześniejszego kodu, w którym brakowało prototypu funkcji.

### Przykładowy kod 24.: *prototyp\_funkcji.cpp*

```
#include <iostream>

using namespace std;

// Prototyp funkcji dodaj
int dodaj (int x, int y);
int main ()
{
    int wynik = dodaj( 1, 2 );
    cout << "Pierwszy wynik to " << wynik << '\n';
    cout << "Wynikiem dodawania 3 i 4 jest " << dodaj( 3, 4 );
}

int dodaj (int x, int y)
{
    return x + y;
}
```

Jak zwykle program rozpoczyna się od dołączenia potrzebnych plików oraz instrukcji using namespace std;.

Następnie znajduje się prototyp funkcji dodaj, zakończony średnikiem. Począwszy od tego miejsca dowolny element programu, łącznie z funkcją main, może korzystać z funkcji dodaj, chociaż jest ona zdefiniowana później — poniżej funkcji main. Ponieważ prototyp naszej funkcji znajduje się powyżej funkcji main, kompilator wie, że została ona zadeklarowana, i zna jej argumenty oraz zwracaną przez nią wartość.

Pamiętaj, że chociaż funkcję można wywoływać jeszcze przed jej zdefiniowaniem, to aby program mógł się skompilować, konieczne będzie podanie jej definicji<sup>1</sup>.

## Rozbijanie programu na funkcje

Teraz, kiedy już wiesz, jak pisać funkcje, powinieneś dowiedzieć się, *kiedy* należy je tworzyć.

### Kiedy wciąż na nowo powtarzasz ten sam kod

Głównym zastosowaniem funkcji jest uproszczenie wielokrotnego używania kodu. Funkcje w znacznym stopniu ułatwiają wielorazowe korzystanie z logiki programu, ponieważ wszystko, co trzeba w tym celu zrobić, to wywołać funkcję; nie ma potrzeby kopiowania i wklejania kodu. Kopiowanie i wklejanie, chociaż może wydawać się łatwiejsze, przyniesie skutek w postaci wielu powielonych bloków tego samego kodu w programie. Korzystanie z funkcji pozwala również zaoszczędzić sporo miejsca, dzięki czemu program będzie czytelniejszy, a także ułatwia wprowadzanie zmian. Czy zamiast dokonać jednej modyfikacji w ciele funkcji wolałbyś wprowadzić w kodzie czterdzieści drobnych zmian rozsianych po całym wielkim programie? Oso-biście wolałbym funkcję.

Warto przyjąć ogólną zasadę, że jeśli napisales ten sam kod trzy razy, zamiast powtarzać go po raz czwarty, przekształć go w funkcję.

### Kiedy chcesz, żeby kod był łatwiejszy do czytania

Nawet jeśli nie musisz wielokrotnie używać kodu, czasami obecność w programie długiego bloku instrukcji, które robią coś szczególnego i skomplikowanego, może utrudnić dostrzeżenie pełnego obrazu tego, czym zajmuje się Twój kod. Utworzenie funkcji umożliwia Ci stwier-dzenie: „Oto idea, z której chcę skorzystać” i zrealizowanie tej idei. Łatwo jest na przykład zrozumieć ideę „wczytania danych, które wpisał użytkownik”, gdy masz jedną funkcję, która się tym zajmuje. Kod realizujący odczytywanie wciśnięć klawiszy, przekształcający je w sygnały elektryczne i wczytujący je do zmiennej — to dopiero skomplikowane zadanie! Czyż nie jest o wiele łatwiej czytać następujący kod:

```
int x;  
cin >> x;
```

niż kod implementujący szczegółowo proces wczytywania danych? Jeśli pracujesz nad jakimś kodem i okazuje się, że ogarnięcie obrazu całości robi się trudne, może nadszedł czas, aby napisać kilka funkcji, które pomogą lepiej zorganizować pracę.

Dzięki napisaniu funkcji będziesz mógł skupić się wyłącznie na tym, co ta funkcja przyjmuje na wejściu i jaki wynik zwraca, zamiast przez cały czas zajmować się szczegółami jej działania.

Możesz zapytać: „Tylko czy nie powiniensem znać szczegółów?”. To prawda, czasami szczegóły będą Ci potrzebne, ale wtedy będziesz mógł zająć się funkcji, ponieważ wszystko, co musisz o niej wiedzieć, znajduje się właśnie tam — w jednym miejscu. Jeżeli detale będą wmi-ezane w większą strukturę programu, poznanie ich będzie trudniejsze.

---

<sup>1</sup> Z technicznego punktu widzenia nie powiedzie się konsolidacja. O różnicach między komplikacją a konsolidacją opowie w dalszej części tej książki.

Weźmy na przykład program, który uruchamia skomplikowany kod, kiedy użytkownik wybiera polecenie w menu. Dla każdej z pozycji menu powinna istnieć odrębna funkcja. Działanie poszczególnych poleceń można zrozumieć, patrząc na powiązane z nimi funkcje, a struktura głównego kodu realizującego pobieranie informacji od użytkownika także jest łatwa do ogniecia. Najgorsze programy mają zwykle tylko jedną, wymaganą funkcję `main`, która jest wypełniona całymi stronami pogmatwanego kodu. W następnym rozdziale będziesz miał okazję poznać taki właśnie kod.

## Nazwanie i przeładowywanie funkcji

Bardzo ważne jest dokonanie właściwego wyboru nazw dla funkcji, zmiennych oraz niemal wszystkich pozostałych elementów programu. Nazwy pomagają zrozumieć, co robi Twój kod. Wywołanie funkcji nie pokazuje Ci jej implementacji, tak więc istotny jest taki dobór nazwy funkcji, która będzie odzwierciedlać najważniejsze jej zadanie. Ponieważ nazwy są tak istotne, czasami będziesz chciał nadać tę samą nazwę więcej niż tylko jednemu elementowi. Założymy na przykład, że masz funkcję obliczającą powierzchnię trójkąta zdefiniowanego za pomocą trzech współrzędnych:

```
int obliczPowierzchnieTrojkata (int x1, int y1, int x2, int y2, int x3, int y3);
```

A gdybyś tak chciał mieć drugą funkcję, która oblicza powierzchnię trójkąta, ale tym razem na podstawie długości jego podstawy i wysokości? Mógłbyś jeszcze raz skorzystać z nazwy `oblicz PowierzchnieTrojkata`, ponieważ bardzo dobrze opisuje ona zadanie tej funkcji. Czy jednak nazwa ta nie spowoduje konfliktu z istniejącą już funkcją `obliczPowierzchnieTrojkata`? Nie w C++! Język ten umożliwia **przeładowywanie** (lub też przeciążanie) funkcji. Możesz użyć tej samej nazwy w odniesieniu do więcej niż tylko jednej funkcji, pod warunkiem że listy ich argumentów będą różne. Można zatem napisać:

```
int obliczPowierzchnieTrojkata (int x1, int y1, int x2, int y2, int x3, int y3);
```

oraz:

```
int obliczPowierzchnieTrojkata (int podstawa, int wysokosc)
```

Kompilator będzie w stanie odróżnić od siebie wywołania tych funkcji, ponieważ mają one inną liczbę argumentów (kompilator daje sobie radę także z funkcjami o tej samej liczbie argumentów, o ile są one różnych typów). Możesz więc napisać:

```
obliczPowierzchnieTrojkata ( 1, 1, 1, 4, 1, 9 );
obliczPowierzchnieTrojkata ( 5, 10 );
```

a kompilator będzie wiedzieć, którą z dwóch funkcji wywołać.

Nie powinieneś nadużywać możliwości przeładowywania funkcji. Sam fakt, że dwa elementy mogą mieć tę samą nazwę, nie oznacza od razu, że tak powinno być. Przeładowywanie ma sens wtedy, gdy dwie funkcje robią to samo, ale przy użyciu różnych argumentów.

## Podsumowanie wiadomości o funkcjach

Razem ze zmiennymi, pętlami oraz instrukcjami warunkowymi, funkcje są jednymi z podstawowych narzędzi, jakimi posługują się programiści w C++. Funkcje pozwalają ukryć złożone obliczenia za prostym interfejsem i pozbyć się powielonych fragmentów kodu. Dzięki temu wielokrotne korzystanie z kodu jest o wiele łatwiejsze.

## Sprawdź się

- 1.** Który z poniższych zapisów nie tworzy poprawnego prototypu funkcji?
  - A. int funk(char x, char y);
  - B. double funk(char x)
  - C. void funk();
  - D. char x();
- 2.** Który z poniższych typów zwraca funkcja o prototypie `int funk(char x, double v, float t);?`
  - A. char
  - B. int
  - C. float
  - D. double
- 3.** Który z poniższych zapisów przedstawia poprawne wywołanie funkcji (przy założeniu, że funkcja ta istnieje)?
  - A. funk;
  - B. funk x, y;
  - C. funk();
  - D. int funk();
- 4.** Który z poniższych zapisów przedstawia kompletną funkcję?
  - A. int funk();
  - B. int funk(int x) {return x=x+1;}
  - C. void funk(int) {cout<<"Witaj";}
  - D. void funk(x) {cout<<"Witaj";}

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Weź program z menu, który napisałeś wcześniej, i rozbij go na serię wywołań funkcji, z których każda odpowiada za jedną pozycję menu. Jako dwie nowe pozycje menu dodaj kalkulator oraz *99 Bottles of Beer*.
- 2.** Zmień program kalkulatora w taki sposób, aby każdy z rodzajów operacji był przeprowadzany w osobnej funkcji.
- 3.** Zmodyfikuj program weryfikujący hasła w taki sposób, aby cała logika sprawdzania hasła znalazła się w odrębnej funkcji.

## Instrukcje switch case oraz typ wyliczeniowy

Zróbmy sobie na chwilę przerwę od dokładania do naszego arsenału nowych technik programowania i powróćmy do czegoś bardziej podstawowego, mianowicie do kodu wykonywanego warunkowo (niestety — nie wszystko, czego się nauczysz, będzie również eksytujące jak funkcje i projektowanie programów). Często będziesz tworzyć długie serie instrukcji if-else sprawdzających rozmaite warunki. Jeśli na przykład odczytasz naciśnięty przez użytkownika klawisz, możesz mieć potrzebę sprawdzenia pięciu lub większej liczby możliwych wartości. Podczas pisania gry będziesz musiał zaimplementować sprawdzanie takich zdarzeń jak naciśnięcie strzałki w lewo, w prawo, w górę i w dół oraz spacji. W rozdziale tym dowiesz się, jak wygodnie pisać tego typu testy warunkowe za pomocą instrukcji switch case, a także uzyskasz nieco informacji na temat tworzenia własnych prostych typów, które bardzo dobrze sprawdzają się w zestawieniu z tymi instrukcjami.

**Instrukcje switch case** zastępują długie ciągi instrukcji if przy porównywaniu pojedynczej wartości z wieloma wartościami **całkowitymi**. Wartość całkowita to wartość, którą można wyrazić w postaci liczby całkowitej, taka jak typu int albo char.

Podstawowy sposób korzystania z instrukcji switch case przedstawiono poniżej. Wartość zmiennej umieszczonej przy instrukcji switch jest porównywana z wartościami znajdującymi się przy każdej instrukcji case. Program będzie kontynuowany od miejsca, w którym obie te wartości są ze sobą zgodne, aż do końca bloku case albo wystąpienia instrukcji break.

```
switch ( <zmienna> )
{
    case ta-wartosc:
        // Kod, który wykona się, gdy <zmienna> == ta-wartosc
        break;
    case tamta-wartosc:
        // Kod, który wykona się, gdy <zmienna> == tamta-wartosc
        break;
    // ...
    default:
        // Kod, który wykona się, gdy <zmienna> nie jest zgodna z żadną z wartościami przy instrukcjach case
        break;
}
```

Pierwszy blok instrukcji, który zostanie wykonany, to blok znajdujący się po instrukcji case, przy której znajduje się wartość zgodna z wartością zmiennej. Instrukcje bloku default wykonają się wtedy, gdy nie zostanie uruchomiony żaden z bloków instrukcji case. Użycie instrukcji default nie jest obowiązkowe, ale dobrze jest ją zamieszczać w celu obsłużenia nieprzewidzianych przypadków.

Zwróć uwagę na instrukcję break kończące każdy blok kodu. Zapobiegają one **przebiegnięciu** przez kolejne bloki case i wykonaniu się wszystkich instrukcji, które są w nich umieszczone. Tak, to dziwne zachowanie, ale właśnie tak to działa. Nie zapominaj więc o umieszczeniu w blokach case instrukcji break, chyba że istotnie zależy Ci na takiej reakcji programu.

Wartość umieszczonej przy każdej instrukcji case musi być stałym wyrażeniem całkowitym. Niestety, nie można pisać takich instrukcji case:

**ZŁY KOD**

```
int a = 10;
int b = 10;
switch ( a )
{
    case b:
        //kod
        break;
}
```

Jeśli spróbujesz skompilować powyższy kod, otrzymasz mniej więcej taki błąd komplikacji:

```
badcode.cpp:9: error: 'b' cannot appear in a constant-expression
```

Poniżej znajduje się przykładowy program demonstrujący działanie instrukcji switch case, który możesz uruchomić.

### Przykładowy kod 25.: switch.cpp

```
#include <iostream>

using namespace std;

void graj ()
{}

void wczytaj ()
{}

void multiplayer ()
{}

int main ()
{
    int wybor;

    cout << "1. Graj\n";
    cout << "2. Wczytaj stan gry\n";
    cout << "3. Tryb multiplayer\n";
    cout << "4. Koniec\n";
    cout << "Wybierz: ";
    cin >> wybor;
    switch ( wybor )
{
```

```

case 1: // Zwróć uwagę na dwukropek po instrukcjach case — nie ma tu średnika!
    graj();
    break;
case 2:
    wczytaj();
    break;
case 3:
    multiplayer();
    break;
case 4:
    cout << "Dziękuję za grę!\n";
    break;
default: // Zwróć uwagę na dwukropek po default — nie ma tu średnika!
    cout << "Niepoprawny wybór. Wyjście z programu.\n";
    break;
}
}

```

Powyższy program skompiluje się i po uruchomieniu przedstawi Ci prosty interfejs użytkownika, chociaż sama gra bardziej będzie przypominać *Czekając na Godota*.

Jeden z problemów, na który być może zwróciłeś uwagę, polega na tym, że użytkownik ma tylko jedną szansę, zanim program skończy działanie. Jeśli wprowadzi on błędą wartość, nie będzie mieć możliwości poprawy. Z łatwością możesz usunąć tę niedogodność, umieszczając cały blok case w pętli, tylko co z instrukcjami break? Czy nie spowodują one wyjścia z pętli? Mam dla Ciebie dobrą wiadomość — instrukcja break sprawi tylko, że program przejdzie na koniec instrukcji switch.

## Porównanie instrukcji switch case z if

Jeśli masz problem ze zrozumieniem logiki instrukcji switch, pomyśl o niej tak, jakby każdemu blokowi case odpowiadała instrukcja if:

```

if ( 1 == wybór )
{
    graj();
}
else if ( 2 == wybór )
{
    wczytaj();
}
else if ( 3 == wybór )
{
    multiplayer();
}
else if ( 4 == wybór )
{
    cout << "Dziękuję za grę!\n";
}
else
{
    cout << "Niepoprawny wybór. Wyjście z programu.\n";
}

```

Jeśli możemy osiągnąć ten sam cel za pomocą instrukcji `if/else`, do czego w ogóle potrzebny jest `switch`? Główna zaleta stosowania instrukcji `switch` polega na tym, że w bardzo czytelny sposób pokazuje ona przebieg programu — jedna zmienna kontroluje ścieżkę, po której zmierza kod. W przypadku serii instrukcji `if/else` należy starannie zapoznać się z każdym warunkiem.

## Tworzenie prostych typów za pomocą wyliczeń

Czasami podczas pisania programu potrzebna jest zmienna, która może przyjmować tylko kilka możliwych wartości, przy czym wartości te są wcześniej znane. Możesz na przykład mieć stałe wartości dostępnych kolorów tła, które może wybrać użytkownik. Bardzo wygodnie byłoby mieć zarówno zbiór takich stałych, jak i specjalnie przeznaczoną do ich przechowywania zmienną. Co więcej, taki rodzaj zmiennej doskonale sprawdziłby się w zestawieniu z instrukcjami `switch-case`, ponieważ z góry znasz jej wszystkie możliwe wartości!

Przyjrzyjmy się, jak osiągnąć taki skutek za pomocą **typu wyliczeniowego**. Typ `enum` tworzy nowy typ zmiennej przy użyciu stałej („wyliczonej”) listy wartości. W roli typu wyliczeniowego dobrze sprawdzą się kolory tęczy:

```
enum KolorTeczy {KT_CZERWONY, KT_POMARANCZOWY, KT_ZOLTY, KT_ZIELONY, KT_NIEBIESKI,  
→KT_INDYGO, KT_FIOLETOWY};
```

W powyższym zapisie należy zwrócić uwagę na następujące elementy:

1. Nowe wyliczenie jest wprowadzane za pomocą słowa kluczowego `enum`.
2. Nowy typ uzyskuje własną nazwę — `KolorTeczy`.
3. Wszystkie możliwe wartości typu znajdują się na liście (poprzedziłem je przedrostkiem `KT_` na wypadek, gdyby ktoś z jakiegoś powodu także chciał skorzystać w kolejnym typie wyliczeniowym z nazw kolorów).
4. Instrukcja kończy się, rzecz jasna, średnikiem.

Teraz można zadeklarować zmienną typu `KolorTeczy` w następujący sposób:

```
KolorTeczy wybrany_kolor = KT_CZERWONY
```

Można również napisać taki kod:

```
switch (wybrany_kolor)  
{  
case KT_CZERWONY: /* Zmień kolor ekranu na czerwony */  
case KT_POMARANCZOWY: /* Zmień kolor ekranu na pomarańczowy */  
case KT_ZOLTY: /* Zmień kolor ekranu na żółty */  
case KT_ZIELONY: /* Zmień kolor ekranu na zielony */  
case KT_NIEBIESKI: /* Zmień kolor ekranu na niebieski */  
case KT_INDYGO: /* Zmień kolor ekranu na indygo */  
case KT_FIOLETOWY: /* Zmień kolor ekranu na fioletowy */  
default: /* Obsługa niespodziewanych typów */  
}
```

Ponieważ mamy do czynienia z typem wyliczeniowym, możemy być pewni, że uwzględniliśmy wszystkie możliwe wartości zmiennej. Za kulisami typ wyliczeniowy jest jednak tylko typem całkowitym; można mu nadawać wartości spoza wyliczenia, ale nie powinieneś tego robić, chyba że naprawdę nie cierpisz programistów zajmujących się konserwacją oprogramowania!

Być może zastanawiasz się nad tym, jakie wartości tak naprawdę przyjmują zmienne wyliczeniowe. Jeżeli nie zdefiniujesz konkretnych wartości podczas deklarowania typu wyliczeniowego, wartością danego elementu listy będzie wartość poprzedniego elementu powiększona o 1, przy czym wartością pierwszego elementu listy jest 0. W naszym przypadku `KT_CZERWONY` przyjmie wartość 0, a `KT_POMARANCZOWY` 1.

Możesz także definiować własne wartości, co może być przydatne podczas pisania kodu korzystającego z danych, które pochodzą z innego systemu — jakiegoś urządzenia albo gotowego kodu — i chciałbyś nadać im czytelne nazwy.

```
enum KolorTeczy {
    KT_CZERWONY = 1, KT_POMARANCZOWY = 3, KT_ZOLTY = 5, KT_ZIELONY = 7, KT_NIEBIESKI = 9,
    KT_INDYGO = 11, KT_FIOLETOWY = 13
};
```

Jeden z głównych powodów, dla których typ wyliczeniowy jest przydatny, jest taki, że umożliwia on nadawanie nazw wartościom, które bez tego typu musiałyby zostać umieszczone na stałe w kodzie programu. Jeśli na przykład chciałbyś napisać grę w kółko i krzyżyk, potrzebowałbyś jakiegoś sposobu na przedstawienie kólek i krzyżyków na planszy. Dla pustego pola mógłbyś wybrać 0, kółka byłyby reprezentowane przez 1, natomiast krzyżyk przez 2. Jeśli tak zrobisz, prawdopodobnie napiszesz także kod porównujący wybrane pole planszy z wartościami 0, 1 i 2:

```
if ( pole_planszy == 2 )
{
    /* Wykonaj coś, bo na tym polu jest krzyżyk */
}
```

Taki zapis jest trudny do odczytania, gdyż w kodzie znajduje się **magiczna liczba** o jakimś znaczeniu, którego jednak nie da się poznać, czytając sam kod (chyba że znajduje się w nim komentarz, taki jak ten, który tam umieściłem). Typ wyliczeniowy pozwala nadawać nazwy takim wartościom:

```
enum PolePlanszyKolkoKrzyzyk {PPKK_PUSTE, PPKK_0, PPKK_X };
if ( pole_planszy == PPKK_X )
{
    /* Jakiś kod */
}
```

Teraz jakiś biedny programista (tym programistą możesz być Ty!), któremu w przyszłości przyjdzie naprawiać błędy w kodzie, będzie mógł zrozumieć, co program robi.

Typ wyliczeniowy bardzo dobrze nadaje się do pracy z wartościami stałymi, natomiast instrukcja `switch case` znakomicie sprawdza się podczas obsługiwania danych wprowadzanych przez użytkownika. Żadne z tych narzędzi nie rozwiązuje jednak problemu jednokrotniej pracy z wieloma wartościami. Wyobraź sobie na przykład, że musisz wczytać ogromną ilość danych statystycznych dotyczących meczów piłkarskich, po czym je przetworzyć. W takich sytuacjach bloki `switch case` nie będą przydatne; będziesz potrzebował jakiejś metody na przechowywanie i przetwarzanie dużych ilości danych.

O tym właśnie będzie traktować część II tej książki. Zanim jednak do niej przejdziemy, dowiesz się, jak sprawić, aby programy zachowywały się interesująco bez potrzeby dostarczania im dużych zestawów danych. W szczególności nauczysz się dodawać do programów element losowości (na wypadek, gdybyś chciał napisać grę).

## Sprawdź się

- 1.** Co następuje po instrukcji case?  
A. :  
B. ;  
C. -  
D. Nowy wiersz.
- 2.** Co jest potrzebne, aby kod nie przebiegał przez kolejne bloki case?  
A. end;  
B. break;  
C. stop;  
D. Średnik.
- 3.** Jakie słowo kluczowe obsługuje niespodziewane przypadki?  
A. all  
B. contingency  
C. default  
D. other
- 4.** Jaki będzie wynik wykonania poniższego kodu?

```
int x = 0;
switch( x )
{
    case 1: cout << "Jeden";
    case 0: cout << "Zero";
    case 2: cout << "Witaj Świecie";
}
```

A. Jeden  
B. Zero  
C. Witaj Świecie  
D. ZeroWitaj Świecie

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

1. Zmodyfikuj program obsługujący menu, który napisałeś jako rozwiązanie zadania praktycznego z rozdziału 6. poświęconego funkcjom, w taki sposób, aby program ten korzystał z instrukcji switch-case.
2. Napisz program, który przy pomocy instrukcji switch-case wyświetli tekst kolędy *The Twelve Days of Christmas*<sup>1</sup> (podpowiedź: możesz skorzystać z przewagi, jaką daje przebieganie przez bloki case).
3. Napisz grę w kółko i krzyżyk, w której dwie osoby grają przeciw sobie. Jeśli to możliwe, użyj typu wyliczeniowego w celu przechowywania wartości pól planszy.

---

<sup>1</sup> [http://en.wikipedia.org/wiki/The\\_Twelve\\_Days\\_of\\_Christmas\\_\(song\)](http://en.wikipedia.org/wiki/The_Twelve_Days_of_Christmas_(song))

# 8

## ■ ■ ■ ROZDZIAŁ 8

# Dodawanie do programu elementu losowości

Odmienne zachowanie programu przy każdym jego uruchomieniu można tak naprawdę uzyskać na dwa sposoby:

1. Jeśli użytkownik wprowadzi różne dane wejściowe (albo różne dane zostaną pobrane z plików).
2. Jeśli program inaczej zareaguje na te same dane otrzymane od użytkownika.

W wielu przypadkach pierwszy sposób sprawdzi się doskonale, a użytkownicy często chcą, aby ich programy były przewidywalne. Jeżeli na przykład piszesz edytor tekstów albo przeglądarkę internetową, prawdopodobnie będziesz chciał uzyskać ten sam efekt za każdym razem, gdy użytkownik wpisze w komputerze fragment tekstu albo adres WWW. Nie chciałbyś, aby przeglądarka losowo decydowała, którą stronę odwiedzisz, o ile tylko nie korzystasz ze *StumbleUpon*<sup>1</sup>.

W niektórych przypadkach zachowywanie się programu w ten sam sposób może jednak stanowić poważny problem — na przykład wiele gier komputerowych bazuje na losowości. Dobrym przykładem takiej gry może być *Tetris*. Gdyby w każdej rozgrywce klocki spadały w takiej samej kolejności, gracze mogliby zapamiętywać długie sekwencje klocków i coraz bardziej poprawiać swoje wyniki wyłącznie dzięki znajomości klocków, które spadną jako następne. Gra taka dawałaby tyle samo rozrywki, co zapamiętywanie liczby *pi* z dokładnością do tysiąca miejsc po przecinku. Aby *Tetris* zapewniał zabawę, potrzebny jest jakiś sposób na losowe wybieranie klocków.

W tym celu musimy skłonić komputer do generowania liczb losowych. Komputer oczywiście dokładnie zrobi, co mu każesz. Oznacza to, że gdy go o coś poprosisz, zawsze uzyskasz to samo, co utrudnia otrzymywanie prawdziwie losowych liczb. Na szczęście nie zawsze musimy mieć rzeczywiście losowe wartości. Często wystarczą liczby, które wyglądają jak losowe; są to tak zwane **liczby pseudolosowe**.

Do wygenerowania liczby pseudolosowej komputer będzie potrzebować pewnej wartości, zwanej **ziarnem**, która zostanie poddana transformacjom matematycznym przekształcającym ją w kolejną liczbę. Ta nowa liczba stanie się następnym ziarnem, które zostanie wykorzystane przez generator liczb losowych. Jeżeli Twój program za każdym razem będzie wybierać inne ziarno, to praktycznie nigdy nie uzyskasz tej samej sekwencji liczb losowych. Używane w tym

<sup>1</sup> *StumbleUpon* to witryna, która pozwala „natknąć się” na interesujące strony internetowe: <http://www.stumbleupon.com/>.

celu przekształcenia matematyczne zostały starannie dobrane, dzięki czemu wszystkie generowane liczby występują z takim samym prawdopodobieństwem i nie wykazują oczywistych wzorców (na przykład nie powstają przez dodanie 1 do podanej przez Ciebie liczby).

C++ zajmuje się tym wszystkim w Twoim imieniu. Nie musisz przejmować się obliczeniami — istnieją funkcje, z których możesz korzystać. Wszystko, co potrzebujesz zrobić, to podać losowe ziarno, co jest równie proste jak odczytanie bieżącego czasu. Przyjrzyjmy się teraz szczegółowo.

## Uzyskiwanie liczb losowych w C++

W C++ istnieją dwie funkcje losowe; jedna z nich służy do nadania wartości ziarnu, natomiast druga generuje liczbę losową na podstawie ziarna:

```
void srand (int seed)
```

Funkcja `srand` pobiera liczbę i na jej podstawie ustawia wartość ziarna. Powinieneś wywołać ją raz, na początku programu. Zwykle funkcję `srand` wywołuje się w taki sposób, aby przekazać jej wynik działania funkcji `time`, która zwraca liczbę reprezentującą bieżący czas<sup>2</sup>.

```
srand ( time( NULL3 ) );
```

Gdybyś co chwilę wywoływał funkcję `srand`, wciąż na nowo nadawałbyś wartość ziarnu generatora losowego, co spowodowałoby, że liczby stałyby się mniej losowe, ponieważ bazują one na sekwencji blisko spokrewnionej z wartościami czasu. Aby korzystać z funkcji `srand`, powinieneś dołączyć plik nagłówkowy `cstdlib`, natomiast funkcja `time` wymaga pliku `ctime`.

### Przykładowy kod 26.: `srand.cpp`

```
#include <cstdlib>
#include <ctime>

int main ()
{
    // Wywołaj tylko raz, na samym początku
    srand( time( NULL ) );
}
```

Wygenerujmy teraz naszą liczbę losową. W tym celu należy wywołać funkcję `rand`, której prototyp wygląda następująco:

```
int rand ();
```

Zwróć uwagę, że funkcja `rand` nie przyjmuje żadnych argumentów, zwraca ona tylko liczbę. Wyświetlmy wynik jej działania:

---

<sup>2</sup> Funkcja `time` zwraca liczbę sekund, jakie upłynęły od pierwszego stycznia 1970. roku. Taka reprezentacja czasu pochodzi z systemu operacyjnego Unix i była nazywana **czasem uniksowym**. W większości przypadków czas jest przechowywany jako 32-bitowa liczba całkowita ze znakiem. Takie rozwiązanie skutkuje interesującą możliwością przepchnięcia się rozmiaru liczby całkowitej i uzyskania w rezultacie liczby ujemnej, która reprezentuje przeszłość. Okazuje się, że coś takiego nastąpi w roku 2038. Zagadnienie to zapoczątkowało dyskusję nad możliwym problemem roku 2038, kiedy to komputery bazujące na czasie uniksowym przedstawią się na rok 1901.Więcej informacji na ten temat znajdziesz na stronie [http://pl.wikipedia.org/wiki/Problem\\_roku\\_2038](http://pl.wikipedia.org/wiki/Problem_roku_2038).

<sup>3</sup> Na razie nie przejmuj się parametrem `NULL`; możesz go traktować jako formalność. Nabierze on więcej sensu, gdy dotrzemy do rozdziału poświęconego wskaźnikom.

**Przykładowy kod 27.: rand.cpp**

```
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

int main ()
{
    // Wywołaj tylko raz, na samym początku
    srand( time( NULL ) );
    cout << rand() << '\n';
}
```

Hura! Program ten zachowuje się inaczej za każdym razem, gdy go uruchamiasz, co oznacza, że czekają Cię długie godziny eksytującej rozrywki. Ciekawe, jaka liczba będzie następna?

No dobrze, może ta zabawa nie jest aż tak eksytująca. Liczby, które otrzymujesz, przyjmują bardzo szeroki zakres wartości. Mógłbyś robić o wiele ciekawsze rzeczy, gdybyś miał możliwość uzyskania liczb pochodzących z określonego zakresu. Jak się okazuje, funkcja rand zwraca liczby z zakresu od 0 do stałej RAND\_MAX (która ma wartość co najmniej 32767). To całkiem sporo, a Tobie prawdopodobnie będzie potrzebny tylko niewielki podzakres spośród tych wartości. Mógłbyś oczywiście wywoływać funkcję rand w pętli i czekać, aż pojawi się liczba z Twojego zakresu:

```
int zakresRand ( int minimum, int maksimum )
{
    while ( 1 )
    {
        int wynik_rand = rand();
        if ( wynik_rand >= minimum && wynik_rand <= maksimum )
        {
            return wynik_rand;
        }
    }
}
```

Jest to jednak okropne rozwiązanie! Pierwszy problem polega na tym, że program działa wolno; jeśli potrzebna Ci będzie liczba między 1 a 4, upłynie wiele czasu, zanim uzyskasz jedną z tych wartości, gdyż rand zwraca liczby pochodzące z o wiele większego zakresu. Drugim problemem jest brak gwarancji, że powyższy kod kiedykolwiek zakończy działanie. Jest możliwe (choć bardzo mało prawdopodobne), że nigdy nie uzyskasz liczb z zakresu, który wyznaczyłeś. Po co podejmować ryzyko, skoro masz inną możliwość uzyskania potrzebnych wyników?

W C++ istnieje operacja zwracająca resztę z dzielenia (na przykład  $4/3$  daje w wyniku 1 z resztą wynoszącą 1). Jest to operacja dzielenia modulo, realizowana za pomocą znaku %. Jeśli podzielasz dowolną liczbę przez 4, resztą z tego dzielenia będzie liczba od 0 do 3. Jeżeli podzielasz liczbę losową przez rozmiar swojego zakresu, uzyskasz liczbę od 0 do górnej granicy tego zakresu (ale z pominięciem tej ostatniej wielkości).

Oto przykład:

**Przykładowy kod 28.: modulo.cpp**

```
#include <ctime>
#include <cstdlib>
```

```
#include <iostream>

using namespace std;

int zakresRand ( int minimum, int maksimum )
{
    // Dostajemy liczbę losową od 0 do górnej granicy naszego
    // zakresu, po czym dodajemy najniższą możliwą wartość
    return rand() % ( maksimum - minimum + 1 ) + minimum;
}

int main ()
{
    srand( time( NULL ) );
    for ( int i = 0; i < 1000; ++i )
    {
        cout << zakresRand( 4, 10 ) << '\n';
    }
}
```

W powyższym kodzie należy zwrócić uwagę na dwie sprawy. Najpierw do różnicę maksimum - minimum musimy dodać 1. Aby zrozumieć, dlaczego taki zabieg jest konieczny, wyobraź sobie, że potrzebne są nam liczby z zakresu od 0 do 10, co daje 11 różnych możliwości wyboru. Odejmowanie da w rezultacie różnicę między wartością największą i najmniejszą zamiast łącznej liczby wartości w zakresie, w związku z czym do uzyskanego wyniku musimy dodać 1. Po drugie, zwróć uwagę na to, że w celu otrzymania pożądanego zakresu liczb powinniśmy do wyniku dodać wartość minimalną. Jeśli na przykład zależy nam na wartościach od 10 do 20, najpierw musimy uzyskać liczbę losową od 0 do 10, po czym dodać do niej 10.

Mając możliwość otrzymywania liczb losowych pochodzących z określonego zakresu, będziesz mógł robić wiele interesujących rzeczy, takich jak tworzenie zgadywanek albo symulowanie rzutu kostką.

## Błędy i losowość

Liczby losowe mogą być przyczyną problemów, kiedy opracowujesz swój program. Rzecz w tym, że gdy usiłujesz zlokalizować błąd, zwykle byłoby lepiej, aby program za każdym razem robił dokładnie to samo. Jeśli tak nie jest, błąd może się nie pojawić, a Ty poświęcisz sporo czasu, sprawdzając przebiegi programu, w trakcie których nie występują problemy. Mogą pojawić się również błędy, których nie oczekiwaleś. Podczas pierwszych testów programu lub w trakcie szukania błędów przydatne może okazać się przekształcenie w komentarz wywołań funkcji srand. Bez nadania wartości ziarnu generatora liczb losowych funkcja rand będzie przy wszystkich uruchomieniach programu zwracać wciąż tę samą sekwencję wartości, dzięki czemu za każdym razem będziesz miał okazję zaobserwować takie samo zachowanie kodu.

Co robić, kiedy po przywróceniu wywołań funkcji srand natrafisz na błędy? Jedno z rozwiązań polega na zapisywaniu wartości ziarna przy każdym uruchomieniu programu:

```
int ziarno_srand = time( NULL );
cout << ziarno_srand << '\n';
srand( ziarno_srand );
```

Jeśli odnajdziesz błąd, będziesz mógł zmienić swój kod w sposób, który umożliwi debugowanie programu przy takiej samej wartości ziarna generatora liczb losowych, jaka pozwoliła Ci odszukać błąd. Jeżeli ziarno miało wartość na przykład 35434333, będziesz mógł napisać:

```
int ziarno_srand = 35434333; // time( NULL );
cout << ziarno_srand << '\n';
srand( ziarno_srand );
```

Od tej pory podczas każdego uruchomienia programu będziesz otrzymywać przewidywalne wyniki.

## Sprawdź się

- 1.** Co się stanie, jeśli przed wywołaniem funkcji rand nie uruchomisz funkcji srand?
  - A. Wywołanie funkcji rand nie powiedzie się.
  - B. Funkcja rand zawsze będzie zwracać 0.
  - C. Podczas każdego uruchomienia programu funkcja rand będzie zwracać tę samą sekwencję liczb.
  - D. Nic się nie stanie.
- 2.** Dlaczego funkcję srand wywołujesz z bieżącym czasem?
  - A. Aby zagwarantować, że program zawsze będzie działać tak samo.
  - B. Aby przy każdym uruchomieniu programu wygenerować nowe liczby losowe.
  - C. Aby zagwarantować, że komputer wygeneruje rzeczywiste liczby losowe.
  - D. To się robi automatycznie; funkcję srand należy wywołać tylko wtedy, gdy za każdym razem chcesz nadawać ziarnu tę samą wartość.
- 3.** Jaki zakres wartości zwraca funkcja rand?
  - A. Jaki chcemy.
  - B. Od 0 do 1000.
  - C. Od 0 do RAND\_MAX.
  - D. Od 1 do RAND\_MAX.
- 4.** Jaki wynik zwróci wyrażenie  $11 \% 3$ ?
  - A. 33
  - B. 3
  - C. 8
  - D. 2
- 5.** Kiedy i ile razy powinieneś wywołać funkcję srand?
  - A. Za każdym razem, gdy potrzebujesz liczby losowej.
  - B. Nigdy, to tylko taki ozdobnik w Windows.
  - C. Raz, na początku programu.
  - D. Sporadycznie — aby poprawić losowość wyników, kiedy funkcja rand była używana już przez jakiś czas.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Napisz program symulujący rzut monetą. Uruchom go wiele razy. Czy uzyskane wyniki wyglądają Twoim zdaniem na losowe?
- 2.** Napisz program, który wybiera liczbę od 1 do 100 i następnie pozwala użytkownikowi ją odgadnąć. Program powinien podpowiadać, czy liczba wskazana przez użytkownika jest za niska, za wysoka, czy poprawna.
- 3.** Napisz program rozwiążający zgadywkę z zadania nr 2. Ile pytań potrzebuje Twój program do odgadnięcia wylosowanej liczby?
- 4.** Napisz program symulujący „jednorękiego bandytę”, który wyświetla użytkownikowi losowe wyniki. Niech każdy z bębnów maszyny przyjmuje co najmniej trzy możliwe wartości. Nie przejmuj się animacją tekstu „przelatującego” przed użytkownikiem; po prostu wylosuj wynik, wyświetl go i zaprezentuj wygraną (wygrywające kombinacje wybierz sam).
- 5.** Napisz program do gry w pokera. Rozdaj użytkownikowi pięć kart, pozwól mu dobrać nowe karty, a następnie określ, jak dobry układ ma on w ręku. Zastanów się, czy zadanie to będzie łatwe w realizacji. Na jakie problemy możesz natrafić, próbując zapamiętać, które karty są już rozdane? Czy zaprogramowanie gry w pokera jest łatwiejsze, czy może bardziej skomplikowane niż napisanie programu symulującego „jednorękiego bandytę”?

# 9

## ■ ■ ■ R O Z D Z I A Ł 9

# Co zrobić, kiedy... nie wiesz, co robić?

---

Teraz, kiedy poznaleś już wiele podstawowych elementów C++, być może zacząłeś dziko biegać wokół, pisząc programy wszędzie, gdzie tylko się da. Ale poczekaj! Skąd wiesz, co pisać? Nawet jeśli znasz problem, który chcesz rozwiązać, możesz poczuć się jak gnomi gaciowe:

**1.** Kradniemy ludziom gacie.

**2.** ?

**3.** Jesteśmy bogaci.

Wiesz, dokąd zmierzasz, i znasz punkt startu, ale krok 2. nie jest już tak oczywisty.

Być może nie zdajesz sobie z tego sprawy, gdy zapoznajesz się z przykładami kodu źródłowego, ale kiedy wezmiesz się za pisanie własnych programów, mogą zacząć się wyłaniać całkiem wyraźne problemy (albo i nie — w takim przypadku masz szczęście, a poza tym udało Ci się wyprzedzić harmonogram; weź sobie wolny wieczór, otwórz piwo i do zobaczenia jutro).

Jesteś zatem jedną z wielu osób, które utkwiły na drugim etapie. Nie ma w tym nic złego! Tak naprawdę jest to etap, który da Ci największą frajdę (tylko nie mów o tym swojemu koledze otwierającemu piwo, bo zrobi mu się smutno).

Muszę Ci powiedzieć, że jest to aspekt programowania, który stanowi dla programisty największe wyzwanie, większe niż na przykład składnia języka, ale daje on też ogromną satysfakcję. Kiedy od podstaw projektujesz nową aplikację, która robi coś, co wydaje się trudne, dzieje się magia. Nie ma nic wspanialszego, jak zobaczyć swój rodzący się program, dzięki któremu trudny problem staje się łatwy. Im więcej będziesz ćwiczyć, tym lepszym programistą się staniesz, ale powinieneś wiedzieć, co ćwiczyć. O tym właśnie traktuje ten rozdział. Niestety, zła wiadomość jest taka, że krok 2. prawdopodobnie przybierze postać kroków od 2. do 22., ponieważ cała sztuka rozwiązywania problemów sprowadza się do rozbicia ich na małe i możliwe do przełknięcia kąski.

Wydobądźmy zatem nasze noże, wędliny oraz trochę majonezu i weźmy się za przyrządżanie przekąsek. Tak naprawdę to zaczniemy raczej od radzenia sobie z sytuacjami, w których w zasadzie wiesz, jak rozwiązać dany problem — gdy masz dość dobre rozeznanie, co się dzieje, i tylko nie jesteś pewien, jak zapisać to w kodzie, czyli wówczas, gdy znasz podstawy algorytmu. **Algorytm** to seria kroków potrzebnych do rozwiązania problemu. Nawet jeśli znasz algorytm, przełożenie go na kod nie zawsze okaże się proste. Być może liczba zadań, z którymi będzie musiał poradzić sobie Twój program, jest przytaczająca. Na szczęście istnieją narzędzia, które są pomocne w takich sytuacjach.

Czy pamiętasz, jak mówiłem, że programowanie polega na rozbijaniu problemów na mniejsze fragmenty, które będą zrozumiałe dla komputera? Wspaniała cecha funkcji polega na tym, że nie zmuszają nas one do pracy z nieprzetworzonymi surowcami, ale pozwalają tworzyć klocki, które potrafi zrozumieć komputer. Co mam na myśli? Założymy, że chcesz wydrukować liczby pierwsze od 1 do 100. Z pewnością wiąże się z tym zadaniem więcej niż tylko jedna operacja, w związku z czym musimy je rozbić na etapy, które pojmie komputer.

Problem jednak polega na tym, że aby osiągnąć zamierzony cel, trzeba się sporo naprawocować! Już przez samo myślenie o tym, jak to wszystko jednocześnie zrealizować, możemy się zupełnie zniechęcić.

A gdybyśmy zastanowili się nad naszym zadaniem w inny sposób: jak moglibyśmy je rozbić na mniejsze części? Takie etapy nie muszą być pojedynczymi instrukcjami. Spróbujmy po prostu uzyskać kroki, które będą łatwiejsze do wykonania niż zadanie, które stoi przed nami obecnie. Oto kilka takich rozsądkowych kroków:

- 4.** Weź po kolej wszystkie liczby od 1 do 100.
- 5.** Sprawdź, czy każda z tych liczb jest pierwsza.
- 6.** Wydrukuj liczbę, jeśli jest pierwsza.

Świetnie! Podzieliliśmy nasz problem na kilka różnych, mniejszych zadań, ale najwyraźniej nie możemy ich jeszcze przełożyć na program. Co powinniśmy w tym celu zrobić? Czy znasz jakiś sposób na sprawdzenie po kolej liczb od 1 do 100? Jak nic wygląda na to, że potrzebna będzie pętla. Prawdę powiedziawszy, jej kod możemy napisać nawet teraz:

```
for ( int i = 0; i < 100; i++ )
{
    // Sprawdź, czy liczba jest pierwsza; jeśli tak, wydrukuj ją
}
```

Wstawmy do pętli wywołanie funkcji o nazwie `czyPierwsza`. Jeśli jej argument będzie liczbą pierwszą, zwróci ona wartość `true`; w przeciwnym razie funkcja zwróci wartość `false`. Musimy jeszcze wymyślić, jak zaimplementować funkcję `czyPierwsza`, ale jeśli tylko wyobrażymy sobie, że już ją mamy, będziemy mogli nieco uzupełnić nasz kod. Jesteśmy w stanie napisać większość funkcji, które możemy sobie wyobrazić — poza tym nasz problem stał się mniejszy, gdyż sprawdzenie jednej liczby jest łatwiejsze do wykonania niż przetestowanie stu wartości. Zmierzamy zatem we właściwym kierunku.

```
for ( int i = 0; i < 100; i++ )
{
    if ( czyPierwsza ( i ) )
        { cout << i << endl;
    }
}
```

Czyż ten kod nie wygląda wspaniale? Mamy podstawową strukturę, z którą możemy pracować. Wszystko, co musimy teraz zrobić, to napisać funkcję `czyPierwsza`. Liczba jest pierwsza, jeśli dzieli się bez reszty tylko przez 1 i siebie samą. Czy definicja ta daje nam wystarczająco wiele informacji, abyśmy mogli rozbić nasz problem na mniejsze podproblemy? Myślę, że tak. W celu sprawdzenia, czy liczba ma dzielnicę, musimy sprawdzić, czy istnieją jakieś liczby (różne od 1 i od niej samej), które ją dzielą bez reszty. Ponieważ będziemy mieć do czynienia z dzieleniem przez wiele różnych wartości, sugerowałbym użycie kolejnej pętli. Oto poszczególne kroki potrzebne do zrealizowania tej części algorytmu:

1. Dla każdej wartości z zakresu od 1 do testowanej liczby.
2. Sprawdź, czy testowana liczba jest podzielna przez zmienną pętli.
  - a. Jeśli tak, zwróć false.
3. Jeśli nie jest podzielna przez żadną z wartości, zwróć true.

Zobaczmy, czy uda nam się przełożyć powyższe kroki na kod źródłowy. Nie wiemy jeszcze, jak sprawdzić, czy dana liczba jest podzielna przez inną liczbę. Przyjmijmy zatem na razie, że potrafimy to zrobić, i w odpowiednim miejscu umieśćmy wywołanie funkcji `czyPodzielna`, która zastąpi nam logikę tego testu.

```
bool czyPierwsza (int liczba)
{
    for ( int i = 2; i < liczba; i++ )
    {
        if ( czyPodzielna( liczba, i ) )
        {
            return false;
        }
    }
    return true;
}
```

Raz jeszcze umieściliśmy w pętli sprawdzanie zakresu wartości. Udało nam się też z łatwością przełożyć warunek „jeśli” w naszej logice na instrukcję warunkową w kodzie.

W jaki sposób zaimplementujemy teraz funkcję `czyPodzielna`? Jeden ze sposobów polega na skorzystaniu z operatora dzielenia modulo:

`10 % 2 == 0 // 10 / 2 = 5 bez reszty`

Wszystko, co musimy zrobić, to sprawdzić, czy dana liczba pozostaje bez reszty, gdy jest dzielona:

```
bool czyPodzielna (int liczba, int dzielnik)
{
    return liczba % dzielnik == 0;
}
```

Hej, spójrz tylko! Zredukowaliśmy nasz problem do zaledwie kilku zadań, które potrafi zrozumieć komputer. Nie ma już więcej funkcji, które musielibyśmy napisać. Wszystko w naszym programie składa się z gotowych instrukcji, które są już zdefiniowane, albo z funkcji, które sami zdefiniowaliśmy. Złożym wszystko razem i spójrzmy na obraz całości.

```
#include <iostream>

// Zwróć uwagę na użycie prototypów funkcji
bool czyPodzielna (int liczba, int dzielnik);
bool czyPierwsza (int liczba);

using namespace std;

int main ()
{
    for ( int i = 0; i < 100; i++ )
    {
        if ( czyPierwsza ( i ) )
        {
```

```
        cout << i << endl;
    }
}

bool czyPierwsza (int liczba)
{
    for ( int i = 2; i < liczba; i++)
    {
        if ( czyPodzielna ( liczba, i ) )
        {
            return false;
        }
    }
    return true;
}

bool czyPodzielna (int liczba, int dzielnik)
{
    return liczba % dzielnik == 0;
}
```

Korzystając z prototypów funkcji, mogliśmy nawet uszeregować kod w takiej samej kolejności, w jakiej myśleliśmy o naszym projekcie! Co więcej, z łatwością możemy czytać kod programu, począwszy od uzyskania obrazu całości (tak jak w przypadku naszego projektu), a skończywszy na szczegółach implementacji **funkcji pomocniczych**.

## Krótką dygresja na temat wydajności i bezpieczeństwa kodu

Przy okazji chciałbym nadmienić, że nasz kod można nieco poprawić, dzięki czemu stanie się on wydajniejszy. Tak naprawdę w pętli funkcji czyPierwsza nie musimy sprawdzać wszystkich wartości, począwszy od 2 aż do liczba. Jeśli potrafimy szybko wymyślić jakiś algorytm, nie oznacza to od razu, że algorytm ten jest najlepszy i najwydajniejszy. W naszym przypadku możemy sprawdzać wartości od 2 do pierwiastka kwadratowego ze zmiennej liczba. Ponieważ jednak wykonujemy test pierwszości dla niewielu liczb, wydajność nie jest aż tak istotna. Z drugiej jednak strony niektóre ważne algorytmy, takie jak kryptograficzny algorytm RSA z kluczem publicznym, używany przez większość banków oraz sklepów internetowych, a także wykorzystywany do zabezpieczania wrażliwych danych, bazuje na możliwości generowania dużych liczb pierwszych w celu tworzenia kluczy szyfrujących<sup>1</sup>. Generowanie dużych liczb pierwszych wymaga oczywiście sprawdzania, czy dana liczba jest pierwsza. Jeśli chciałbyś utworzyć wiele kluczy szyfrujących RSA, potrzebowałbyś szybkiego i wydajnego generatora liczb pierwszych.

Kiedy tylko pracujesz nad jakimś problemem, który wydaje się zbyt duży do ogarnięcia, postaraj się rozbić go na mniejsze podproblemy, nad którymi można łatwiej zapanować. Nie musisz od razu wiedzieć, jak je rozwiązać (oczywiście nie zaszkodzi, jeśli będziesz miał o tym pojęcie).

<sup>1</sup> Więcej informacji na temat algorytmu RSA znajdziesz w Wikipedii: [http://pl.wikipedia.org/wiki/RSA\\_\(kryptografia\)](http://pl.wikipedia.org/wiki/RSA_(kryptografia)).

Ważne jest, abyś na podstawie tych małych problemów zrozumiał, jakie dane wejściowe są potrzebne i co z nich wynika. Jeżeli będziesz w stanie napisać program zawierający funkcje, które rozwiążają te małe podproblemy, będziesz gotów na zmierzenie się z następnym wyzwaniem, jakim jest rozwiązanie większego problemu. Jeśli wystarczająco długo będziesz wytrwały w swoim postępowaniu, otrzymasz w końcu kod źródłowy.

Czasami okaże się, że rozwiązanie podproblemu jest z jakiegoś powodu niemożliwe. Projektowanie programów nie zawsze jest łatwe (gdyby było, mielibyśmy o wiele więcej znudzonych inżynierów oprogramowania). Jeśli podczas rozbijania problemu natrafisz na przeszkody, spróbuj cofnąć się o krok i poszukać innego sposobu na podzielenie go na mniejsze fragmenty. W ten sposób sprawdzisz, czy uda Ci się odnaleźć więcej możliwych do rozwiązania podproblemów.

Takie podejście, polegające na rozbijaniu problemów na mniejsze części, jest nazywane **projektowaniem od góry do dołu**. Inna metoda, **projektowanie od dołu do góry**, polega na tworzeniu wpierw funkcji pomocniczych, a następnie użyciu ich do rozwiązania większego problemu. Projektowanie od dołu do góry może prowadzić do sytuacji, w których tworzysz funkcje pomocnicze, które nie są Ci do niczego potrzebne, chociaż dzięki nim łatwiej jest rozpocząć pracę nad projektem, ponieważ masz do dyspozycji działające funkcje. Osoby początkujące powinny jednak przyjąć metodę projektowania od góry do dołu, co pozwoli im skupić się na pisaniu funkcji rozwiązujących określony problem. W ten sposób, zamiast zgadywać, które funkcje mogą być przydatne podczas rozwiązywania problemu, zoptymalizujesz swój projekt w kierunku poszukiwania dokładnie takich funkcji, które będą Ci potrzebne<sup>2</sup>.

Nie musisz także projektować wszystkiego od razu w kodzie źródłowym. Zapisywanie projektu na papierze albo tablicy zmywalnej pozwoli Ci zobaczyć, jak poszczególne elementy pasują do siebie, bez konieczności wgłębiania się w składnię C++ i bez błędów komplikacji. Projektowanie bezpośrednio w kodzie może czasami przesłonić obraz całości, ponieważ zajmujesz się wówczas wszystkimi drobnymi elementami składni języka, które są Ci potrzebne. Nie stanie się zatem nic złego, jeśli zdecydujesz się na zapisywanie poszczególnych etapów projektu i rozbijanie każdego z nich na serie mniejszych czynności bez natychmiastowego przekształcania ich w kod. Takie podejście jest jak najbardziej sprawiedliwione i naturalne.

Jeśli jest jeszcze coś, o czym powinieneś wiedzieć, to fakt, że opracowanie problemu nie zawsze będzie łatwe. To, co Ci przekazałem, będzie pomocne, ale nie traktuj tego, jakby to był magiczny pocisk. Pomoże Ci praktyka. Zdobędziesz ją i będziesz coraz lepszy w tym, co robisz. Być może będziesz potrzebował na to nieco czasu, ale nie poddawaj się.

## Co robić, kiedy nie znasz algorytmu?

W przypadku szukania liczb pierwszych nasze zadanie było łatwe, ponieważ już sama definicja liczby pierwszej podsuwa nam algorytm na przeprowadzenie testu pierwszości. Cały problem sprowadzał się „tylko” do przetłumaczenia algorytmu na kod. Przeważnie jednak rozwiązanie zadania nie będzie aż tak proste. Aby poradzić sobie z problemem, będziesz musiał obmyślić algorytm.

<sup>2</sup> Nie myśl jednak, że chcę Cię powstrzymać od wypróbowania metody projektowania od dołu do góry. Niektóre osoby świetnie odnajdują się w niej; być może będzie to również Twój metoda. Jeśli nie możesz zrozumieć idei projektowania od góry do dołu, odwróć swój projekt o 180°, zanim się poddasz.

Wyobraź sobie na przykład próbę utworzenia algorytmu dla programu, który wyświetla nazwę danej liczby, na przykład 1204 jako „tysiąc dwieście cztery”. Kiedy wypowiadasz liczby, robisz to tak naturalnie, że prawdopodobnie nawet nie zastanawiasz się nad strukturą algorytmu; po prostu to robisz. Aby prawidłowo podejść do rozwiązywania takiego problemu, powinieneś wśród danych odkryć powtarzalny schemat, który umożliwi Ci skonstruowanie algorytmu.

Dobrym punktem wyjścia będzie zanotowanie kilku przykładów i zastanowienie się nad występującymi wśród nich podobieństwami oraz różnicami w celu odnalezienia jakiegoś schematu. Postąpmy właśnie w taki sposób:

1	jeden
10	dziesięć
101	sto jeden
1001	tysiąc jeden
10001	dziesięć tysięcy jeden
100001	sto tysięcy jeden
1000001	milion jeden
10000001	dziesięć milionów jeden
100000001	sto milionów jeden

Czy widzisz, jak wyłania się pewien schemat?

1	jeden
10	dziesięć
101	sto jeden
1001	tysiąc jeden
10001	dziesięć tysięcy jeden
100001	sto tysięcy jeden
1000001	milion jeden
10000001	dziesięć milionów jeden
100000001	sto milionów jeden

Po każdych trzech cyfrach następuje przejście w górę o jeden poziom — od niczego do tysięcy i milionów, ponadto można też zaobserwować powtarzający się wzorzec „nic, dziesięć, sto”, który można z kolei połączyć ze wzorcem wyższego poziomu: „nic tysiąc”, „dziesięć tysięcy” i „sto tysięcy” (pomińmy na chwilę kwestię liczby mnogiej słów tysiąc i milion).

Nasz algorytm musi zatem rozbić liczbę na trzycyfrowe grupy, określić „rząd wielkości” danej grupy (tysiąc, milion, miliard), po czym przekształcić tę grupę na tekst i połączyc ją z właściwym rzędem wielkości. Każda z trzycyfrowych grup jest mniejsza od tysiąca, w związku z czym pozostał nam do rozwiązywania o wiele mniejszy problem — to bardzo dobra wiadomość. Poszukajmy kolejnego schematu:

5	pięć
15	piętnaście
25	dwendzieścia pięć
35	trzydzięści pięć
45	czterdzięści pięć
105	sto pięć
115	sto piętnaście
125	sto dwadzieścia pięć
135	sto trzydzięści pięć
145	sto czterdzięści pięć

Mamy tu do czynienia z podobnym schematem. Jeśli dana liczba jest większa od 100, tekst ma postać „sto”, „dwadzieścia”, „trzysta” itd., z dołączonym zapisem liczby dwucyfrowej. Jeśli liczba jest mniejsza od 100, mamy tylko tekst odzwierciedlający fragment dwucyfrowy.

Wszystko, co musimy teraz zrobić, to zdecydować, jak poradzić sobie z setkami oraz liczbami dwucyfrowymi. Czy widzisz, że także i w tym przypadku występuje schemat? Z wyjątkiem liczb mniejszych od 20, zawsze mamy do czynienia ze wzorcem „nazwa dziesiątek” „nazwa jedności”, który możemy odtworzyć przy pomocy prostej serii instrukcji `if/else`.

Aby rozprawić się z liczbami od 1 do 19 oraz setkami, zapiszemy je na stałe w kodzie programu; nie pomoże nam w tym żaden algorytm — przynajmniej nie taki, który potrafiłbym sobie wyobrazić!

Nasz algorytm będzie więc wyglądać mniej więcej tak:

1. Rozbij liczby na trzycyfrowe grupy.
2. Dla każdej z tych grup wyznacz odpowiedni tekst, dołącz do niej rząd wielkości, po czym połącz ze sobą poszczególne grupy.
3. Aby uzyskać słowny zapis grupy trzycyfrowej, wyznacz, w której setce grupa ta się znajduje, odszukaj w kodzie odpowiadający jej tekst i dołącz do niego zapis grupy dwucyfrowej.
4. Aby uzyskać słowny zapis dwucyfrowej grupy mniejszej niż 20, odczytaj go z kodu programu. Jeśli dwucyfrowa grupa jest większa lub równa 20, określ, w której dziesiątce grupa ta się znajduje, odszukaj odpowiadający jej tekst i dołącz do całości słowny zapis ostatniej cyfry.

W celu przełożenia powyższego algorytmu na kod źródłowy potrzebny będzie kolejny przebieg, gdyż jeszcze nie wszystkie szczegóły są w pełni określone (musimy między innymi zadbać o odpowiednią odmianę liczebników). Podstawy algorytmu zostały jednak zarysowane w wystarczającym stopniu, abyś w celu jego implementacji mógł przyjąć metodę projektowania od góry do dołu.

Czy widzisz już, jak przebiega taki proces? Zastanawiając się nad przykładami różnych liczb, mogliśmy odnaleźć pewne powtarzające się schematy, zgodnie z którymi ustrukturyzowane są liczby. Udało nam się dotrzeć do sedna algorytmu. Nie opracowaliśmy w pełni jego szczegółów, ale nie ma w tym nic złego. Na kolejnych etapach procesu projektowania będziemy coraz bardziej precyzować detale, aż pod koniec wszystko stanie się jasne.

## Zadania praktyczne

1. Zaimplementuj kod źródłowy, który przekształca liczby od -999999 do 999999 na ich zapis słowny (podpowiedź: mógłbyś skorzystać z przewagi, jaką daje fakt, że w typie całkowitym obcinane są miejsca dziesiętne; pamiętaj też, że Twój algorytm nie musi działać dla wszystkich wprowadzanych liczb, a tylko dla tych, które mają do sześciu cyfr).
2. Zastanów się, jak mógłbyś opracować przypadek odwrotny — przekładanie zapisu słownego na liczby. Czy jest on łatwiejszy do zrealizowania niż algorytm z poprzedniego zadania? W jaki sposób obsłużyłbyś błędne dane wejściowe?
3. Zaprojektuj program znajdujący wszystkie liczby od 1 do 1000, których czynniki pierwsze sumują się do liczby pierwszej (na przykład czynnikami pierwszymi liczby 12 są 2, 2 i 3, sumujące się do 7, która jest liczbą pierwszą). Zaimplementuj dla swojego algorytmu kod (nie przejmuj się, jeśli nie znasz algorytmu odszukującego czynniki pierwsze danej liczby i masz problem z jego obmyśleniem — poszukaj w Google! Właśnie to miałem na myśli, gdy twierdziłem, że nie musisz znać matematyki, aby zostać programistą).

# III

■ ■ ■ CZĘŚĆ III

## Praca z danymi

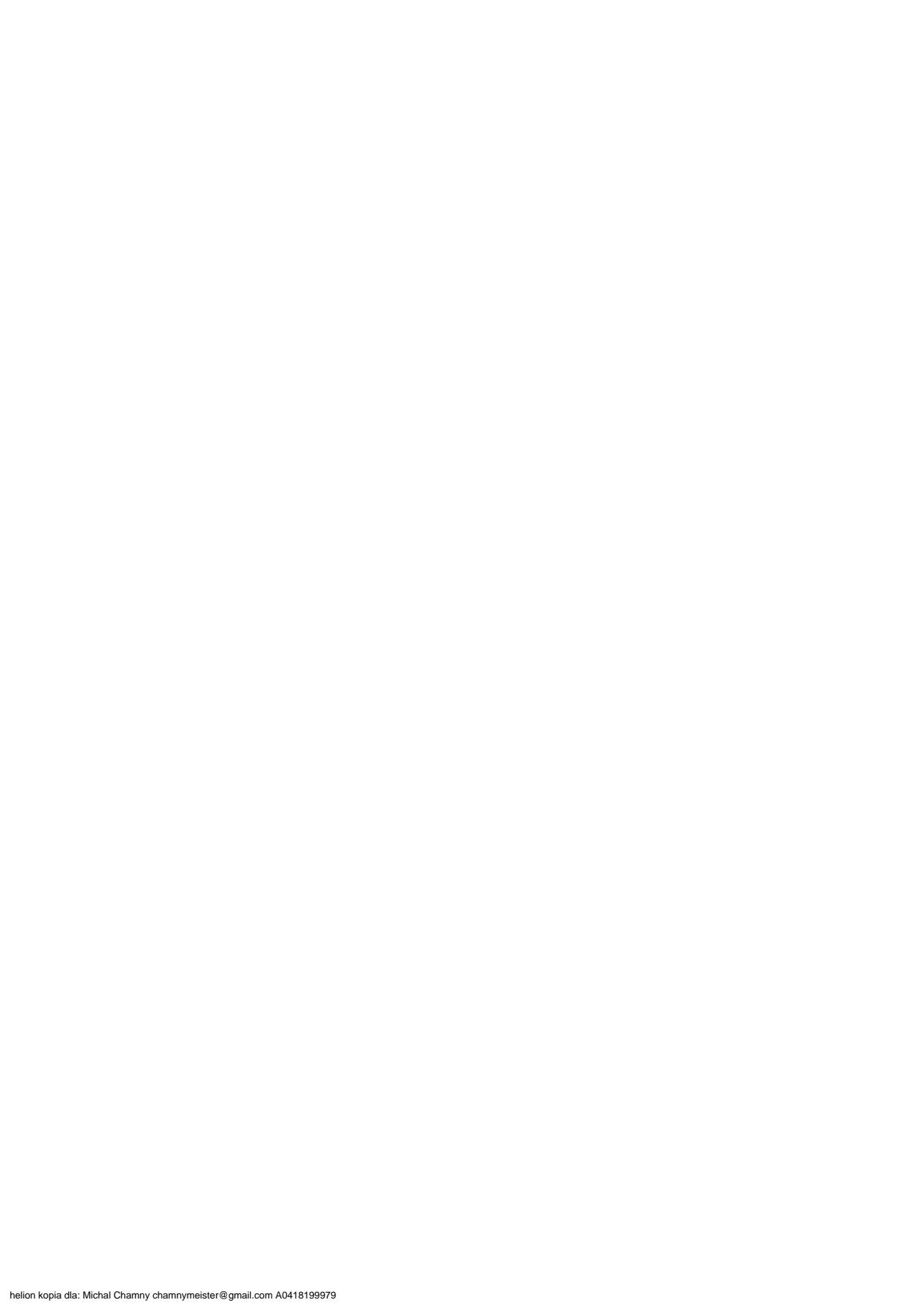
---

W części I sporo dowiedziałeś się o tym, jak pisać proste programy, które robią ciekawe rzeczy — wyświetlają napisy (na przykład Twoje imię), reagują na działania użytkownika, podejmują decyzje na podstawie wprowadzonych danych, powtarzają cyklicznie proste operacje, a nawet prowadzą gry losowe.

Wszystko to jest bardzo pozyteczne, ale po pewnym czasie możesz stwierdzić, że programy te stają się nudnawe; robienie ciekawych rzeczy przy użyciu małych ilości danych jest dość trudne. Powróć myślami do „pokerowego” zadania z jednego z poprzednich rozdziałów. Czy łatwo będzie zapamiętać karty, które zostały rozdane i rozegrane? Jak trudne będzie potasowanie całej talii kart i wyświetlenie ich w nowej kolejności?

Zapewne będzie to trudne. Najpierw musisz znaleźć jakiś sposób na przechowanie 52 różnych wartości; będą Ci potrzebne 52 zmienne. Za każdym razem, kiedy nadasz zmiennej wartość reprezentującą pewną kartę, będziesz musiał sprawdzić wszystkie pozostałe zmienne w celu stwierdzenia, czy dana karta nie została już wcześniej rozdana. Zanim dotrzesz do 52. karty, będziesz mieć bardzo dużo kodu i bardzo mało chęci do dalszego programowania. Na szczęście programiści są leniwi i nie lubią wykonywać pracy, za którą nie muszą się brać. Udało im się wymyślić elegancki sposób na rozwiązywanie tego problemu.

Ta część książki jest poświęcona rozwiązywaniu problemów, które pojawiają się podczas pracy z dużymi ilościami danych; dowiesz się o tym, jak takie dane wczytywać, przechowywać je w pamięci oraz przetwarzać. Zaczniemy od poznania techniki umożliwiającej przechowywanie dużych ilości danych bez potrzeby tworzenia wielu zmiennych, dzięki której rozwiążemy problem pokerowy.



# ■ ■ ■ R O Z D Z I A Ł 1 0

## Tablice

---

Tablice są odpowiedzią na pytanie „Jak mogę w łatwy sposób przechowywać dużo danych?”. **Tablica** to w zasadzie zmienna o jednej nazwie, mogącą przyjmować wiele wartości, z których każda ma swój indeks. Tablicę możesz uważać za listę numerowaną, do której elementów masz dostęp poprzez ich numery.

Tablice bardzo łatwo można sobie wyobrazić:

wart0	wart1	wart2	wart3	wart4	wart5
-------	-------	-------	-------	-------	-------

Ja zawsze myślałem o tablicach jak o długim rzędzie pudełek, które ustawiono jedno za drugim. Każde pudełko jest **elementem** tablicy. Pobranie wartości z tablicy to jak sięgnięcie do pudełka o określonym numerze: „Proszę o pudełko nr 5!”. Do tego właśnie sprowadza się cała ta magia. Ponieważ tablice przechowują wszystkie swoje wartości pod jedną nazwą, możliwe jest **programowe** wskazanie, który element zostanie wybrany. Pisząc „programowe”, mam na myśli to, że nie trzeba w kodzie podawać nazwy konkretnej zmiennej — program sam wybierze właściwą wartość na podstawie jej numeru. Jeśli zechcesz w pokerze dobrać pięć kart, będziesz mógł przechować je w tablicy o rozmiarze pięć. Dobranie kolejnej karty nie będzie wymagać użycia nowej zmiennej, tylko sprowadzi się do zmiany indeksu tablicy. Dzięki zastosowaniu zmiennej do pamiętania tego indeksu możliwe będzie korzystanie z jednego kodu w celu dobrania każdej z pięciu kart. Nie będzie konieczne pisanie oddzielnego kodu dla każdej zmiennej. Istnieje wyraźna różnica między takim kodem:

```
Karta1 = wylosujKarte();
Karta2 = wylosujKarte();
Karta3 = wylosujKarte();
Karta4 = wylosujKarte();
Karta5 = wylosujKarte();
```

a takim:

```
for ( int i = 0; i < 5; i++ )
{
    karta[ i ] = wylosujKarte();
}
```

Wyobraź sobie teraz różnicę przy stu kartach!

## Podstawowa składnia tablic

Aby zadeklarować tablicę, oprócz jej nazwy należy określić jeszcze dwa elementy: typ i rozmiar.

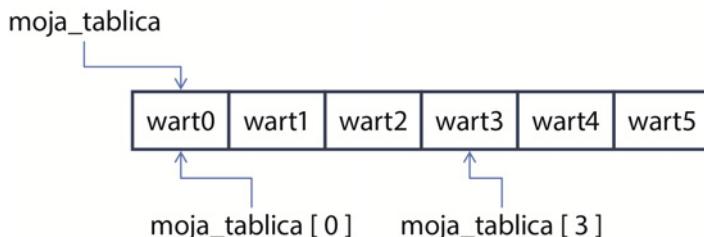
```
int moja_tablica[ 6 ]
```

W ten sposób zostanie zadeklarowana tablica z sześcioma elementami typu całkowitego. Zwróć uwagę, że rozmiar tablicy jest podawany w nawiasach prostokątnych, które występują po jej nazwie.

Aby uzyskać dostęp do wybranego elementu tablicy, również należy skorzystać z nawiasów prostokątnych, z tym że zamiast rozmiaru znajdzie się między nimi indeks potrzebnego elementu:

```
moja_tablica [3]
```

Możemy to sobie wyobrazić następująco:



Nazwa `moja_tablica` odnosi się do tablicy jako całości, podczas gdy `moja_tablica[0]` oznacza pierwszy element, a `moja_tablica[3]` czwarty. Jeśli jeszcze raz musiałeś przeczytać poprzednie zdanie, znaczy to, że jesteś uważnym Czytelnikiem. Nie ma tam jednak błędu — **indeksy** tablic zaczynają się od 0. Przez indeks rozumiem liczbę, z jakiej korzystasz w celu odczytania z tablicy konkretnej wartości. Do takiego sposobu liczenia raczej nie jesteś przyzwyczajony, chyba że Twój rodzic (lub ktoś inny, kto uczył Cię liczyć) byli programistami.

Oto prosty sposób na zapamiętanie tej zasady. Indeks to odległość, jaką należy pokonać wzduż listy, aby dotrzeć do właściwego pudełka. Z pewnością przedżej czy później spotkasz się z określeniem **przesunięcie**. Przesunięcie to wyszukiwany sposób na powiedzenie tego samego — że wartości w tablicy są przesunięte względem jej początku o wielkość indeksu. Ponieważ pierwszy element tablicy znajduje się na jej początku, przesunięcie — a tym samym indeks — wynosi 0.

Kiedy wybierzesz już określony element tablicy, możesz traktować go jak każdą inną zmienną. Możesz modyfikować go w następujący sposób:

```
int moja_tablica [ 4 ]; // Deklaracja tablicy
moja_tablica[ 2 ] = 2; // Niech trzeci (tak, trzeci!) element tablicy ma wartość 2
```

## Przykładowe zastosowania tablic

### Przechowywanie zamówień w tablicach

Czy pamiętasz pytanie, które wcześniej zadałem: „Jak posortowałbyś talię 52 kart?”. Część problemu stanowi sposób na zapamiętanie 52 kart. Teraz już masz rozwiązanie — możesz skorzystać z tablicy. Kolejną częścią problemu była prezentacja kolejności kart w talii. Tutaj również czeka nas dobra wiadomość. Ponieważ dostęp do tablic realizowany jest poprzez liczby, za naturalny porządek kart w talii można przyjąć kolejność, w jakiej poszczególne elementy występują w tablicy. Jeśli zatem zapiszemy w tablicy 52 losowo wybrane i niepowtarzalne wartości, będziemy mogli przyjąć, że pierwszy element tablicy (o indeksie 0) jest pierwszą kartą w talii, a ostatni element (o indeksie 52) to karta ostatnia.

Kolejnym szerokim zastosowaniem tablic jest przechowywanie sortowanych wartości. W jaki sposób mógłbyś wczytać na przykład 100 liczb i pokazać je w kolejności posortowanej? Odłóżmy na bok problem sortowania; liczby można umieścić w tablicy i skorzystać z przewagi, jaką daje naturalne uporządkowanie tablic.

## Odwzorowanie siatek w tablicach wielowymiarowych

Tablice można wykorzystywać także w celu przechowywania danych **wielowymiarowych**, takich jak plansze do gry w szachy albo warcaby (lub w kółko i krzyżyk, jeśli wolisz coś prostszego). Dane wielowymiarowe oznaczają, że do ich reprezentacji musisz użyć więcej niż jednego indeksu.

Aby zadeklarować tablicę dwuwymiarową, należy podać oba jej wymiary:

```
int plansza_kolko_krzyzyk[ 3 ][ 3 ]
```

Oto proste wyobrażenie tablicy `plansza_kolko_krzyzyk[ 3 ][ 3 ]`:

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

Ponieważ tablice dwuwymiarowe są prostokątne, istnieją dwa indeksy, które powinieneś podać — jeden dla wiersza i drugi dla kolumny. Na rysunku pokazałem indeksy, w których musisz skorzystać w celu uzyskania dostępu do każdego z elementów. Potrzebujesz po prostu dwóch wartości, z których jedna znajdzie się w pierwszej przegródce, a druga w przegródce drugiej.

Możesz tworzyć także tablice trójwymiarowe, chociaż prawdopodobnie nie będziesz miał takiej potrzeby. W rzeczy samej można deklarować tablice cztero-, pięcio- i więcej wymiarowe, ale są one dość trudne do wyobrażenia i praktycznie nie będziesz z nich korzystać, w związku z czym nie będę ich rysował.

Tablice w kształcie siatki umożliwiają lepszą organizację danych. W przypadku gry w kółko i krzyżyk możesz nadawać poszczególnym elementom tablicy wartości odzwierciedlające bieżącą sytuację na planszy. Tablice mogą być również używane do odwzorowania labiryntu lub układu poziomu w grze *RPG*.

## Korzystanie z tablic

### Tablice i pętle

Tablice i pętle doskonale ze sobą współpracują. Dostęp do tablicy można uzyskać poprzez zainicjalizowanie zmiennej wartością 0, a następnie zwiększanie wartości tej zmiennej, aż osiągnie ona wartość równą rozmiarowi tablicy — takie postępowanie znakomicie wpisuje się w schemat działania pętli.

Oto niewielki program, który demonstruje zastosowanie pętli w celu utworzenia tabliczki mnożenia i zapisania jej w tablicy dwuwymiarowej:

**Przykładowy kod 29.: tablica\_wielowymiarowa.cpp**

```
#include <iostream>

using namespace std;

int main ()
{
    int tablica[ 8 ][ 8 ]; // Deklaracja tablicy, która wygląda jak szachownica

    for ( int i = 0; i < 8; i++ )
    {
        for ( int j = 0; j < 8; j++ )
        {
            tablica [ i ][ j ] = i * j; // Nadanie wartości każdemu elementowi
        }
    }
    cout << "Tabliczka mnożenia:\n";
    for ( int i = 0; i < 8; i++ )
    {
        for ( int j = 0; j < 8; j++ )
        {
            cout << "[ " << i << " ][ " << j << " ] = ";
            cout << tablica [ i ][ j ] << " ";
            cout << "\n";
        }
    }
}
```

## Przekazywanie tablic do funkcji

Szybko nauczysz się, które z elementów języka współpracują ze sobą. Teraz, kiedy poznaleś tablice, mógłbyś zadać pytanie o to, jak przekazywać tablice do funkcji. Na szczęście nie wiąże się z tym jakaś szczególna składnia.

Podczas wywoływania funkcji należy po prostu użyć nazwy tablicy:

```
int wartosci[ 10 ];
sumuj_tablice( wartosci );
```

W deklaracji funkcji należy z kolei podać nazwę tablicy w następujący sposób:

```
int sumuj_tablice (int wartosci[]);
```

„Hej! — słyszę, jak pytasz. — O co tu chodzi? Przecież tu nie ma rozmiaru tablicy!”. Zgadza się. W przypadku tablic jednowymiarowych nie ma potrzeby podawania ich rozmiaru. Wielkość tablicy jest potrzebna tylko podczas definiowania tablicy, ponieważ kompilator musi zarezerwować dla niej miejsce. Kiedy przekazujesz tablicę do funkcji, kompilator przesyła do niej oryginalną tablicę. W związku z tym nie jest konieczne podawanie jej rozmiaru, gdyż nie tworzy się wówczas nowa tablica. Fakt ten oznacza, że jeśli funkcja zmodyfikuje istniejącą tablicę, wprowadzone w tablicy zmiany pozostaną w niej po zakończeniu działania funkcji. Jak już zobaczyliśmy wcześniej, zwykłe zmienne są kopowane — kiedy funkcja pobiera argument i modyfikuje zmienną, która przechowuje wartość przekazaną w argumencie, zabieg ten nie ma wpływu na wartość oryginalnej zmiennej.

Jeśli funkcja nie zna rozmiaru tablicy, należy rzecz jasna przekazać jej tę informację w postaci parametru funkcji:

```

int sumujTablice (int wartosci[], int rozmiar)
{
    int suma = 0;
    for ( int i = 0; i < rozmiar; i++ )
    {
        suma += wartosci[ i ];
    }
    return suma;
}

```

Z drugiej jednak strony w przypadku korzystania z tablicy wielowymiarowej należy podać wszystkie jej rozmiary, z wyjątkiem pierwszego:

```
int zaznacz_kolko_krzyzyk (int plansza[] [ 3 ]);
```

Dziwne! Na razie zapamiętał tylko, że nie musisz uwzględniać pierwszego rozmiaru (jeśli chcesz, możesz to zrobić, chociaż informacja ta zostanie pominięta).

O przekazywaniu tablic do funkcji napiszę nieco więcej, gdy przejdziemy do wprowadzenia do wskaźników w rozdziale 12. Wy tłumaczę wówczas, jakie obliczenia wykonuje za kulisami kompilator. Do tego czasu możesz traktować pomijanie pierwszego wymiaru tablicy jako pewne dziwactwo składni C++.

Napiszmy teraz pełną wersję programu demonstrującego funkcję `sumujTablice`.

### **Przykładowy kod 30.: *sumuj\_tablice.cpp***

```

#include <iostream>

using namespace std;

int sumujTablice (int wartosci[], int rozmiar)
{
    int suma = 0;
    // Pętla zatrzyma się, gdy i == rozmiar. Dlaczego? Bo ostatni element ma wartość rozmiar - 1
    for ( int i = 0; i < rozmiar; i++ )
    {
        suma += wartosci[ i ];
    }
    return suma;
}

int main ()
{
    int wartosci[ 10 ];
    for ( int i = 0; i < 10; i++ )
    {
        cout << "Podaj wartość " << i << ": ";
        cin >> wartosci[ i ];
    }
    cout << sumujTablice ( wartosci, 10 ) << endl;
}

```

Zastanów się, jak mógłbyś napisać taki program bez tablicy. Nie miałbyś możliwości zapamiętania wszystkich wartości, w związku z czym musiałbyś przechowywać bieżącą sumę narastającą. Za każdym razem, gdy użytkownik wprowadzi nową wartość, musiałbyś ją od razu dodawać. Nie miałbyś możliwości łatwego przechowywania wprowadzonych wartości na wypadek, gdybyś chciał skorzystać z nich później (na przykład w celu wyświetlenia liczb, które zostały zsumowane).

## Wypadnięcie poza ostatni element tablicy

Chociaż możesz dowolnie sięgać do wszystkich elementów tablicy, nigdy nie powinieneś próbować wyjść poza ostatni jej element, tak jak na przykład wtedy, gdy w przypadku tablicy 10-elementowej użyłbyś indeksu równego 10:

### ZŁY KOD

```
int moja_tablica[ 10 ];
moja_tablica[ 10 ] = 4; // Próba zapisania jedenastego elementu tablicy
```

Powyzsza tablica liczy tylko dziesięć elementów, zatem ostatni jej poprawny indeks ma wartość 9. Użycie indeksu równego 10 jest nieprawidłowe i może spowodować awarię Twojego programu (dlaczego tak się dzieje, wyjaśnię, gdy przejdziemy do omawiania funkcjonowania pamięci). Najczęściej spotykany scenariusz, gdy następuje przekroczenie indeksu tablicy, ma miejsce podczas działania pętli przebiegającej przez tablicę:

### ZŁY KOD

```
int wartosci[ 10 ];
for ( int i = 0; i <= 10; i++ )
{
    cin >> wartosci[ i ];
}
```

Tablica w powyższym przykładzie ma dziesięć elementów, ale warunek pętli sprawdza, czy *i* jest mniejsze lub równe 10, co znaczy, że nastąpi próba zapisania w elemencie `wartosci[ 10 ]`, czego nie powinno się robić. Niestety, pomimo swojej drobiazgowości kompilator nie poinformuje Cię o takim błędzie. Dowiesz się o problemie dopiero wtedy, gdy w Twoim programie nastąpi awaria lub zacznie się on dziwnie zachowywać, ponieważ zmieniona wartość została użyta w jakimś innym kodzie.

## Sortowanie tablic

Spróbujmy teraz odpowiedzieć na pytanie, które zadałem wcześniej: „W jaki sposób można zapamiętać sto wartości i posortować je?”. Podstawowa struktura kodu powinna być teraz w miarę oczywista; potrzebna jest pętla wczytująca 100 liczb całkowitych podanych przez użytkownika:

### Przykładowy kod 31.: wczytaj\_calkowite.cpp

```
#include <iostream>

using namespace std;

int main ()
{
    int wartosci[ 100 ];
    for ( int i = 0; i < 100; i++ )
    {
        cout << "Podaj wartość " << i << ": ";
        cin >> wartosci[ i ];
    }
}
```

Ta część była prosta — masz już wczytane liczby, ale jak je posortujesz? Sposób, w jaki większość osób naturalnie sortuje elementy, polega na odnalezieniu wśród nich najmniejszej wartości

i przesunięciu jej na początek. Następnie szukany jest drugi najmniejszy element i umieszczany bezpośrednio za pierwszym. Potem należy znaleźć trzeci najmniejszy element i przesunąć go za element drugi.

Gdybyś miał posortować następującą listę:

3, 1, 2

Najpierw przesunąłbyś 1 na początek listy:

1, 3, 2

Następnie przeniósłbyś 2 na drugie miejsce listy:

1, 2, 3

Czy sądzisz, że korzystając z poznanych do tej pory elementów C++, mógłbyś napisać kod realizujący takie zadanie? Według mnie można tu zastosować pętlę. W pętli tej będziesz iterować po tablicy, począwszy od jej pierwszego elementu. Decyzję, jaką wartość umieścić w tym miejscu, podejmiesz po odnalezieniu najmniejszego elementu w pozostałej części tablicy (czyli w jej nieposortowanym fragmencie). Następnie zamienimy miejscami odszukaną wartość z wartością elementu, który znajduje się na pozycji wskazywanej przez bieżący indeks (wartość tego elementu musi zostać gdzieś zachowana). Zaczniemy od napisania krótkiego kodu, korzystając z projektowania metodą od góry do dołu:

```
void sortuj (int tablica [])
{
    for ( int i = 0; i < 100; i++ )
    {
        int indeks = znajdzNajmniejszyPozostalyElement( tablica, i );
        zamien( tablica, i, indeks );
    }
}
```

Teraz możemy pomyśleć o zaimplementowaniu dwóch funkcji pomocniczych: `znajdzNajmniejszyPozostalyElement` oraz `zamien`. Zastanówmy się nad funkcją `znajdzNajmniejszyPozostalyElement`. Począwszy od indeksu równego `i`, powinna ona przechodzić przez tablicę w celu odszukania w niej najmniejszego elementu. Wygląda to na kolejną pętlę, prawda? Przyglądamy się każdemu elementowi tablicy i jeśli jest on mniejszy od najmniejszego elementu, z którym mieliśmy do tej pory do czynienia, przyjmujemy indeks tego elementu za bieżący indeks naszego najmniejszego elementu.

```
int znajdzNajmniejszyPozostalyElement (int tablica [], int indeks)
{
    int indeks_najmniejszej_wartosci = indeks;
    for (int i = indeks + 1; i < ???; i++)
    {
        if ( tablica[ i ] < tablica[ indeks_najmniejszej_wartosci ] )
        {
            indeks_najmniejszej_wartosci = i;
        }
    }
    return indeks_najmniejszej_wartosci;
}
```

Nasz kod wygląda całkiem rozsądnie. Jest jednak pewien problem — skąd mamy wiedzieć, kiedy pętla powinna się zakończyć? W argumentach funkcji `znajdzNajmniejszyPozostalyElement` brakuje informacji, która wskazywałaby, jak duża jest tablica. Musimy dodać taki parametr,

a także uzupełnić go w wywołaniu funkcji. Zauważ, że natrafiliśmy na sytuację, w której przyjęte przez nas podejście od góry do dołu wymaga, abyśmy powrócili do wcześniejszego kodu i dokonali w nim zmian. Nie ma w tym nic złego — taki powrót stanowi część procesu projektowania i nie powinieneś przejmować się, że musisz to zrobić. Przy okazji uporządkujmy nieco nasz kod sortujący, aby nie był w nim na stałe zapisany 100-elementowy rozmiar tablicy.

```
int znajdzNajmniejszyPozostalyElement (int tablica [], int rozmiar, int indeks)
{
    int indeks_najmniejszej_wartosci = indeks;
    for (int i = indeks + 1; i < rozmiar; i++)
    {
        if ( tablica [ i ] < tablica [indeks_najmniejszej_wartosci ] )
        {
            indeks_najmniejszej_wartosci = i;
        }
    }
    return indeks_najmniejszej_wartosci;
}

void sortuj (int tablica[], int rozmiar)
{
    for ( int i = 0; i < rozmiar; i++ )
    {
        int indeks = znajdzNajmniejszyPozostalyElement ( tablica, rozmiar, i );
        zamien( tablica, i, indeks );
    }
}
```

Na koniec musimy zaimplementować funkcję zamien. Ponieważ funkcja ta będzie modyfikować oryginalną tablicę, która zostanie do niej przekazana, możemy po prostu zamienić miejscami dwie wartości, korzystając ze zmiennej tymczasowej przechowującej pierwszą, nadpiswaną wartość.

```
void zamien (int tablica[], int pierwszy_indeks, int drugi_indeks)
{
    int tymczas = tablica [ pierwszy_indeks ];
    tablica[ pierwszy_indeks ] = tablica[ drugi_indeks ];
    tablica[ drugi_indeks ] = tymczas;
}
```

Funkcja zamien modyfikuje oryginalną tablicę, która jest do niej przekazywana, w związku z czym nie musimy w niej już nic więcej robić.

Aby udowodnić, że nasz algorytm działa, możemy wypełnić tablicę losowo wygenerowanymi liczbami, a następnie ją posortować. Oto pełny listing programu:

### **Przykładowy kod 32.: sortowanie\_przez\_wstawianie.cpp**

```
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

int znajdzNajmniejszyPozostalyElement (int tablica[], int rozmiar, int indeks);
void zamien (int tablica[], int pierwszy_indeks, int drugi_indeks);

void sortuj (int tablica[], int rozmiar)
{
```

```

for ( int i = 0; i < rozmiar; i++ )
{
    int indeks = znajdzNajmniejszyPozostalyElement( tablica, rozmiar, i );
    zamien( tablica, i, indeks );
}
}

int znajdzNajmniejszyPozostalyElement (int tablica[], int rozmiar, int indeks)
{
    int indeks_najmniejszej_wartosci = indeks;
    for (int i = indeks + 1; i < rozmiar; i++)
    {
        if ( tablica[ i ] < tablica[ indeks_najmniejszej_wartosci ] )
        {
            indeks_najmniejszej_wartosci = i;
        }
    }
    return indeks_najmniejszej_wartosci;
}

void zamien (int tablica[], int pierwszy_indeks, int drugi_indeks)
{
    int tymczas = tablica[ pierwszy_indeks ];
    tablica[ pierwszy_indeks ] = tablica[ drugi_indeks ];
    tablica[ drugi_indeks ] = tymczas;
}

// Nieduża funkcja pomocnicza wyświetlająca tablicę przed i po sortowaniu
void wyswietlTablice (int tablica[], int rozmiar)
{
    cout << "{";
    for ( int i = 0; i < rozmiar; i++ )
    {
        // Wzorcem tym będziesz mieć często do czynienia — służy on
// do czytelnego formatowania list; kod sprawdza, czy jesteśmy
// już poza pierwszym elementem, i jeśli tak, dodaje przecinek
        if ( i != 0 )
        {
            cout << ", ";
        }
        cout << tablica[ i ];
    }
    cout << "}";
}

int main ()
{
    int tablica[ 10 ];
    srand( time( NULL ) );
    for ( int i = 0; i < 10; i++ )
    {
        // Liczby będą łatwiejsze do odczytania, gdy będą małe
        tablica[ i ] = rand() % 100;
    }
    cout << "Tablica wyjściowa: ";
    wyswietlTablice( tablica, 10 );
    cout << '\n';
}

```

```
sortuj( tablica, 10 );
cout << "Tablica posortowana: ";
wyswietlTablice( tablica, 10 );
cout << '\n';
}
```

Algorytm sortujący, który właśnie poznaleś, nazywany jest **sortowaniem przez wstawianie**. Nie jest to najszybszy algorytm służący do sortowania liczb, ale ma jedną zaletę: jest łatwy do zrozumienia i zimplementowania. Jeśli chciałbyś posortować duży zbiór danych, mógłbyś wybrać algorytm szybszy, ale trudniejszy do zaprogramowania i zrozumienia. Tego typu kompromisowe decyzje będziesz musiał podejmować jako programista. W wielu przypadkach najprostszy do zimplementowania algorytm będzie także najlepszy. Jeżeli jednak obsługujesz stronę internetową z milionami odwiedzin dziennie, najłatwiejszy algorytm może nie wystarczyć. Na podstawie spodziewanej ilości napływających danych oraz tego, jak ważne jest szybkie działanie, podejmiesz świadomą decyzję, z którego algorytmu skorzystać. Jeżeli istnieje możliwość uruchomienia nocnego przetwarzania wsadowego, algorytm może być wolniejszy. Jeżeli jednak musisz w czasie rzeczywistym udzielać odpowiedzi na pytania użytkowników (tak jak Google), rozwiązanie takie nie będzie dobre.

Jak widzisz, tablice dają nam spore możliwości. Możemy pracować ze znacznie większymi ilościami danych oraz organizować więcej informacji, niż to było możliwe wcześniej. Nadal jednak pozostaje do rozwiązania kilka problemów. Co moglibyśmy zrobić, gdybyśmy zamiast zapamiętywania pojedynczych wartości zechcieli skojarzyć wiele różnych, chociaż powiązanych ze sobą danych? Tablice pozwalają organizować odrębne dane, ale nie pomogą nam w organizowaniu informacji, które mają ze sobą coś wspólnego. Dowiemy się, jak rozwiązywać tego typu problemy, w następnym rozdziale, przy okazji omawiania struktur.

Drugi problem polega na tym, że tablice zawsze udostępniają stałą ilość pamięci — nawet bardzo dużo, jeśli potrzebujemy — ale jest to ilość określona z góry podczas pisania programu. Jeżeli zechcesz napisać program, który może przechowywać i przetwarzać nieograniczone ilości danych, tablice o stałych rozmiarach nie wystarczą. Już niedługo zabierzemy się za rozwiązywanie także i tego typu problemów.

Pomimo wszystkich tych ograniczeń tablice znacznie poprawiają naszą sytuację i niejeden raz będziemy korzystać z indeksów w celu odczytywania i zapisywania danych.

## Sprawdź się

1. Który z poniższych zapisów poprawnie definiuje tablicę?
  - A. int jakas\_tablica[ 10 ];
  - B. int jakas\_tablica;
  - C. jakas\_tablica{ 10 };
  - D. array jakas\_tablica[ 10 ];
2. Jaki jest indeks ostatniego elementu tablicy liczącej 29 elementów?
  - A. 29
  - B. 28
  - C. 0
  - D. Zdefiniowany przez programistę.

- 3.** Który z poniższych zapisów definiuje tablicę dwuwymiarową?
- A. array jakas\_tablica[ 20 ][ 20 ];
  - B. int jakas\_tablica[ 20 ][ 20 ];
  - C. int jakas\_tablica[ 20, 20 ];
  - D. char jakas\_tablica[ 20 ];
- 4.** Który z poniższych zapisów poprawnie odczyta siódmy element tablicy foo liczącej 100 elementów?
- A. foo[ 6 ];
  - B. foo[ 7 ];
  - C. foo( 7 );
  - D. foo;
- 5.** Który z poniższych zapisów poprawnie deklaruje funkcję, której argumentem jest tablica dwuwymiarowa?
- A. int funk ( int x[][] );
  - B. int funk ( int x[ 10 ][] );
  - C. int funk ( int x[] );
  - D. int funk ( int x[] [ 10 ] );

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

1. Przekształć kod realizujący sortowanie przez wstawianie w taki sposób, aby działał z tablicami o dowolnych rozmiarach.
2. Napisz program, który wczytuje 50 wartości, po czym w osobnych wierszach wypisuje wartość najmniejszą, największą, średnią oraz każdą z wprowadzonych liczb.
3. Napisz program, który potrafi stwierdzić, czy tablica jest posortowana. Jeśli nie jest, powinien ją posortować.
4. Napisz niewielki program do gry w kółko i krzyżyk, który pozwoli dwóm graczom grać przeciwko sobie. Program powinien sprawdzać, czy któryś z graczy wygrał oraz czy plansza została w całości wypełniona (co oznacza zakończenie rozgrywki remisem). Zadanie dodatkowe: czy potrafisz sprawić, aby Twój program wiedział, że rozgrywka nie może zostać wygrana przez żadnego uczestnika przed zapelnieniem całej planszy?
5. Rozszerz swój program do gry w kółko i krzyżyk o możliwość gry na planszach większych niż 3x3 pola, z tym że do uzyskania wygranej konieczne będzie zdobycie czterech pól w rzędzie. Pozwól graczom na określanie rozmiaru planszy podczas działania programu (podpowiedź: na razie musisz tak definiować tablicę, aby podczas komplikacji kodu miała ona określony rozmiar, w związku z czym będziesz musiał ograniczyć maksymalną wielkość planszy, jaką mogą podać użytkownicy).
6. Napisz program do gry w warcaby dla dwóch osób, który umożliwia graczom wykonywanie ruchów, weryfikuje dopuszczalność poszczególnych posunięć oraz sprawdza, czy gra została zakończona. Nie zapomnij o możliwości zdobycia damki! Możesz ponadto uwzględnić dowolne dodatkowe reguły, które stosujesz podczas gry w warcaby. Zastanów się nad udostępnieniem użytkownikowi możliwości wyboru określonych reguł gry na starcie programu.



# 11

## ■ ■ ■ ROZDZIAŁ 11

## Struktury

---

### Wiązanie wielu wartości

Teraz, kiedy potrafisz już zapisywać pojedyncze wartości w tablicach, zyskałeś możliwość pisania programów, które przetwarzają duże ilości danych. Podczas pracy z większymi ilościami danych zdarzy się, że będziesz mieć do czynienia z rozmaitymi informacjami, które są ze sobą powiązane. Możesz na przykład chcieć przechowywać współrzędne ekranowe (wartości x oraz y) oraz imiona graczy w grze komputerowej. Obecnie mógłbyś to zrealizować za pomocą trzech tablic:

```
int wspolrzedne_x[ 10 ];
int wspolrzedne_y[ 10 ];
string imiona[ 10 ];
```

Musiałbyś jednak pamiętać, że każda tablica łączy się z pozostałymi. Jeśli więc zmieniłbyś pozycję jednego z elementów w pierwszej tablicy, konieczne byłoby przeniesienie elementów także w dwóch pozostałych tablicach.

Takie rozwiązanie stałoby się bardzo niewygodne, gdybyś potrzebował kontrolować jeszcze czwartą wartość. Musiałbyś dodać kolejną tablicę i pamiętać o synchronizowaniu jej z pierwszymi trzema. Na szczęście osoby projektujące języki programowania nie są masochistami, w związku z czym wymyślono lepszy sposób na kojarzenie ze sobą wartości. Sposobem tym są struktury. Umożliwiają one przechowywanie pod wspólną nazwą różnych wartości zapisanych w wielu zmiennych. Struktury są pozyteczne, kiedy różne informacje powinny zostać zgrupowane.

### Składnia

Definiowanie struktury wygląda następująco:

```
struct StatekKosmiczny
{
    int wspolrzedna_x;
    int wspolrzedna_y;
    string nazwa;
}; // <- Zwróć uwagę na ten nieznośny średnik — musisz o nim pamiętać
```

W powyższym przykładzie `StatekKosmiczny` jest nazwą zdefiniowanej przez nas struktury. Innymi słowy, utworzyliśmy własny typ, taki jak `double` albo `int`, z którego można korzystać w celu deklarowania zmiennych:

```
StatekKosmiczny moj_statek;
```

Elementy `wspolrzedna_x`, `wspolrzedna_y` oraz `imie` to pola naszego nowego typu. Moment! Pola? Właściwie co to takiego?

Oto wyjaśnienie. Utworzyliśmy właśnie typ złożony — zmienną, która przechowuje wiele powiązanych ze sobą wartości (takich jak współrzędne albo imię i nazwisko). Sposób, w jaki informujesz zmienną, którą z jej wartości chcesz odczytać, polega na podaniu nazwy pola, którym jesteś zainteresowany. To tak, jakby mieć dwie zmienne o różnych nazwach, z tym że są one zgrupowane, a ich nazewnictwo zostało uspójnione. Strukturę możesz traktować jak formularz (wyobraź sobie na przykład podanie o prawo jazdy) z polami. Formularz przechowuje wiele danych, a każde z jego pól zawiera powiązane ze sobą informacje. Zadeklarowanie struktury to sposób na zdefiniowanie formularza, natomiast zadeklarowanie zmiennej o typie zgodnym z tą strukturą tworzy kopię formularza, którą można wypełnić i wykorzystać do przechowywania całej masy danych.

Aby uzyskać dostęp do pól, należy zapisać nazwę zmiennej (ale nie nazwę struktury, ponieważ każda zmienna przechowuje w swoich polach inne wartości), postawić kropkę `.`, po czym podać nazwę pola:

```
// Deklaracja zmiennej
StatekKosmiczny moj_statek;

// Użycie zmiennej
moj_statek.wspolrzedna_x = 40;
moj_statek.wspolrzedna_y = 40;
moj_statek.nazwa = "USS Enterprise (NCC-1701-D)";
```

Jak widać, w strukturze może znajdować się wiele pól — praktycznie tyle, ile tylko chcesz — i nie muszą być one tego samego typu.

Spójrzmy teraz na przykładowy program demonstrujący odczytywanie imion pięciu graczy w grze komputerowej (gra została pominięta), w którym wykorzystano tablice oraz struktury:

```
#include <iostream>

using namespace std;

struct Informacja_o_Graczu
{
    int poziom_trudnosci;
    string imie;
};

int main ()
{
    // Tak jak w przypadku innych typów, można tworzyć tablice struktur
    Informacja_o_Graczu gracze[ 5 ];
    for ( int i = 0; i < 5; i++ )
    {
        cout << "Podaj imię gracza: " << i << '\n';
        // Najpierw za pomocą zwykłej składni tablicowej
        // wskaz element tablicy, a następnie przy użyciu składni
        // z kropką wczytaj wartość pola struktury
        cin >> gracze[ i ].imie;
        cout << "Podaj poziom trudności dla gracza " << gracze[ i ].imie << '\n';
        cin >> gracze[ i ].poziom_trudnosci;
    }
    for ( int i = 0; i < 5; ++i )
```

```

    {
        cout << gracze[ i ].imie << " osiągnął poziom trudności " <<
        gracze[ i ].poziom_trudnosci << '\n';
    }
}

```

W strukturze `Informacja_o_Graczu` zadeklarowane są dwa pola — jedno zawiera imię gracza, a drugie poziom trudności. Ponieważ ze struktury można korzystać tak jak z każdego innego typu zmiennej (na przykład `int`), możesz utworzyć tablicę z informacjami o graczech. Po utworzeniu tablicy struktur każdy element takiej tablicy będziesz mógł traktować tak jak pojedynczy egzemplarz struktury. Aby uzyskać dostęp do pola z imieniem gracza w pierwszej strukturze tablicy, wystarczy, że napiszesz `gracze[ 0 ].imie`.

W programie tym wykorzystano przewagę, jaką daje łączenie tablic i struktur w celu wczytania w pierwszej pętli dwóch różnych informacji o pięciu gracach i wyświetlenia tych informacji w drugiej pętli. Nie ma potrzeby deklarowania kilku powiązanych ze sobą tablic dla każdego rodzaju informacji; niepotrzebne będą dwie odrębne tablice `imie_gracza` i `poziom_gracza`.

## Przekazywanie struktur

Często będziesz musiał napisać funkcję, która otrzymuje strukturę jako argument albo zwraca strukturę. Gdybyś na przykład pracował nad grą, w której lata statek kosmiczny, mógłbyś potrzebować funkcję inicjalizującą strukturę wrogiego statku.

```

struct WrogiStatekKosmiczny
{
    int wspolrzedna_x;
    int wspolrzedna_y;
    int sila_broni;
};

WrogiStatekKosmiczny generujNowegoWroga ();

```

W takim przypadku funkcja `generujNowegoWroga` powinna zwracać wartość, w której zainicjalizowane są wszystkie pola struktury. Oto jak mógłbyś ją napisać:

```

WrogiStatekKosmiczny generujNowegoWroga ()
{
    WrogiStatekKosmiczny statek;
    statek.wspolrzedna_x = 0;
    statek.wspolrzedna_y = 0;
    statek.sila_broni = 20;
    return statek;
}

```

Funkcja ta sporządzi kopię zmiennej lokalnej `statek`, którą następnie zwróci. Oznacza to, że do nowej zmiennej skopiowane zostaną po kolejne wszystkie pola struktury. Chociaż kopiowanie każdego z pól może wydawać się powolne, przez większość czasu komputer jest na tyle szybki, że nie będzie to mieć większego znaczenia. Kopiowanie pól nabierze jednak znaczenia, gdy zaczniesz pracować z dużą liczbą struktur. W następnym rozdziale — poświęconym wskaźnikom — pomówimy zatem o unikaniu tworzenia tych dodatkowych kopii.

W celu otrzymania zwracanej zmiennej należy skorzystać z takiego kodu jak poniżej:

```
WrogiStatekKosmiczny statek = generujNowegoWroga ();
```

Teraz możesz stosować zmienną `statek` tak jak każdą inną zmienną struktury.

Przekazywanie struktury do funkcji będzie wyglądać następująco:

```
WrogiStatekKosmiczny ulepszBron (WrogiStatekKosmiczny statek)
{
    statek.sila_broni += 10;
    return statek;
}
```

Kiedy do funkcji przekazywana jest struktura, zostanie ona skopiowana (tak samo jak wtedy, gdy zwracamy strukturę), w związku z czym wszelkie zmiany dokonane w strukturze przez funkcję zostaną utracone! To właśnie z tego powodu funkcja musi zwrócić kopię struktury po jej zmodyfikowaniu — ponieważ wyjściowa struktura nie uległa zmianie.

Aby użyć funkcji ulepszBron w celu zmodyfikowania zmiennej statek o strukturze WrogiStatekKosmiczny, należy napisać:

```
statek = ulepszBron ( statek );
```

Kiedy funkcja ulepszBron zostanie wywołana, zmienna statek zostanie skopiowana do argumentu tej funkcji. Z kolei gdy funkcja zwraca wartość, pola struktury WrogiStatekKosmiczny są kopiowane do zmiennej statek, nadpisując poprzednie pola.

Oto przykładowy program demonstrujący tworzenie i ulepszanie wrogiego statku:

### Przykładowy kod 33.: *ulepszanie.cpp*

```
struct WrogiStatekKosmiczny
{
    int wspolrzedna_x;
    int wspolrzedna_y;
    int sila_broni;
};

WrogiStatekKosmiczny generujNowegoWroga()
{
    WrogiStatekKosmiczny statek;
    statek.wspolrzedna_x = 0;
    statek.wspolrzedna_y = 0;
    statek.sila_broni = 20;
    return statek;
}

WrogiStatekKosmiczny ulepszBron (WrogiStatekKosmiczny statek)
{
    statek.sila_broni += 10;
    return statek;
}

int main ()
{
    WrogiStatekKosmiczny wrog = generujNowegoWroga();
    wrog = ulepszBron( wrog );
}
```

Być może zastanawiasz się, co zrobić, gdybyś chciał utworzyć nieograniczoną liczbę wrogich statków kosmicznych i pamiętać o nich w miarę postępu gry. W jaki sposób można byłoby je wygenerować? Zapewne wywoływałbyś funkcję generujNowegoWroga. Gdzie jednak byś je zapamiętywał? Na razie znasz jedynie tablice o stałych rozmiarach. Mógłbyś utworzyć tablicę z elementami typu WrogiStatekKosmiczny:

---

```
WrogiStatekKosmiczny moje_wrogie_statki[ 10 ]
```

Jednak w ten sposób mógłbyś zapamiętać tylko nie więcej niż dziesięć wrogów jednocześnie. Być może tyle wrogich statków wystarczy, a być może nie. Rozwiążanie tego problemu kryje się w kilku kolejnych rozdziałach, z których pierwszy zawiera wprowadzenie do wskaźników.

## Sprawdź się

- 1.** Który z poniższych zapisów umożliwia dostęp do zmiennej w strukturze b?
  - A. b->zmn;
  - B. b.zmn;
  - C. b-zmn;
  - D. b>zmn;
- 2.** Który z następujących zapisów poprawnie definiuje strukturę?
  - A. struct {int a;}
  - B. struct struktura\_a {int a};
  - C. struct struktura\_a int a;
  - D. struct struktura\_a {int a;};
- 3.** Który z następujących zapisów deklaruje zmienną struktury typu foo o nazwie moje\_foo?
  - A. moje\_foo as struct foo;
  - B. foo moje\_foo;
  - C. moje\_foo;
  - D. int moje\_foo;
- 4.** Jaka będzie ostateczna wartość, którą wyświetli poniższy kod?

```
#include <iostream>

using namespace std;

struct MojaStruktura
{
    int x;
};

void aktualizujStrukture (MojaStruktura moja_struktura)
{
    moja_struktura.x = 10;
}

int main ()
{
    MojaStruktura moja_struktura;
    moja_struktura.x = 5;
    aktualizujStrukture( moja_struktura );
    cout << moja_struktura.x << '\n';
}
```

- A. 5
- B. 10
- C. Ten kod się nie skompiluje.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

1. Napisz program, który pozwoli użytkownikowi wypełnić strukturę nazwiskiem, adresem i numerem telefonu jednej osoby.
2. Utwórz tablicę statków kosmicznych i napisz program, który na bieżąco aktualizuje ich pozycje, aż znikną z ekranu. Przyjmij, że ekran ma rozmiary 1024 na 768 pikseli.
3. Bazując na zadaniu nr 1, napisz program implementujący książkę adresową. Tym razem użytkownik powinien mieć możliwość wypełnienia nie tylko pojedynczej struktury, ale także dodawania nowych pozycji, z których każda będzie zawierać nazwisko oraz numer telefonu. Pozwól użytkownikowi dodawać tyle pozycji, ile zechce. Czy taki program będzie łatwy do zrealizowania? Czy w ogóle będzie możliwy? Dodaj opcję wyświetlania wszystkich lub tylko niektórych wpisów i pozwól użytkownikowi na ich przeglądanie.
4. Napisz program pozwalający użytkownikom wprowadzać najlepsze wyniki w grze, który będzie pamiętał imię użytkownika oraz jego punktację. Dodaj możliwość wyświetlania najwyższej punktacji każdego z graczy, wszystkich wyników wybranego gracza, wszystkich wyników wszystkich graczy oraz listy graczy.

# 12

■ ■ ■ ROZDZIAŁ 12

## Wprowadzenie do wskaźników

---

### Zapomnij o wszystkim, co do tej pory słyszałeś

Niestety, koncepcja wskaźników nabrała wśród początkujących (a nawet profesjonalnych) programistów pewnej aury mistyczności. Jeśli słyszałeś, że wskaźników trudno się nauczyć, są pogmatwane albo zbyt ciężkie do zrozumienia, zapomnij o tym i zignoruj tego typu opinie.

Fakty są takie, że kiedy uczyłem programowania, prawie wszystkie osoby, które dotrwały do wskaźników, dały sobie z nimi radę. Czytasz **moją** książkę i **obiecuję** Ci, że zrozumiesz, jak działają wskaźniki oraz dlaczego i w jaki sposób z nich korzystać — jeśli tylko poświęcisz im nieco czasu.

Być może przez parę dni będziesz łamał sobie głowę nad wskaźnikami, ale trochę ćwiczeń intelektualnych nie powinno Ci zaszkodzić. Postaram się, aby kilka następnych rozdziałów było w miarę krótkich, dzięki czemu Twój umysł będzie mógł odpocząć podczas wielu przerw. Zacznę od wyjaśnienia koncepcji wskaźników i zanim przejdę do omawiania szczegółów ich składni, wy tłumaczę, dlaczego będziesz ich potrzebować.

### No dobrze, czym są wskaźniki? Dlaczego powinny mnie obchodzić?

Aż do tej pory mogliśmy pracować tylko ze stałą ilością pamięci, która była ustalana jeszcze przed uruchomieniem programu. Podczas deklarowania zmiennej za kulisami aplikacji alokowana jest pewna pamięć, potrzebna do przechowywania informacji zapisanej w tej zmiennej. Kiedy deklarujesz zmienną, rozmiar potrzebnej pamięci zostanie określony podczas komplikacji; w trakcie działania programu nie będziesz mógł zmienić tej ilości. Wiemy już, jak tworzyć tablice danych przechowujące wiele zmiennych — to całkiem sporo pamięci — ale w tablicy nie można zapisać więcej elementów, niż zadeklarowano podczas pisania programu. Z kilku kolejnych rozdziałów dowiesz się, w jaki sposób uzyskać dostęp do większej ilości pamięci, niż mamy do dyspozycji na początku działania programu. Nauczysz się tworzyć nieograniczoną liczbę wrogich statków kosmicznych, które latają w tym samym czasie po ekranie (z pominięciem ich wyświetlania).

Aby uzyskać dostęp do praktycznie nieograniczonych zasobów pamięci, będziemy potrzebować jakiegoś rodzaju zmiennej potrafiącej odwoływać się bezpośrednio do pamięci, w której zapisywane są zmienne. Taki rodzaj zmiennej nazywamy **wskaźnikiem**.

Wskaźniki noszą dość trafną nazwę — są to zmienne, które „wskazują” pewną lokalizację w pamięci komputera. Są one bardzo podobne do hiperłączy. Strona internetowa znajduje się

w pewnym miejscu, na czymś serwerze WWW. Czy wysłałbyś e-mailem całą stronę internetową, gdybyś chciał komuś ją pokazać? Nie, raczej posłałbyś samo łącze do niej. Podobnie wskaźnik umożliwia przechowywanie lub przesyłanie „łącza” do zmiennej, tablicy lub struktury, zamiast tworzenia jej kopii.

Wskaźnik, podobnie jak hiperłącze, zawiera lokalizację pewnych danych. Ponieważ wskaźniki mogą przechowywać lokalizację (czyli **adres**) innych danych, będziesz mógł z nich korzystać w celu sięgania do pamięci uzyskanej od systemu operacyjnego. Innymi słowy, Twój program będzie mógł poprosić o więcej pamięci i użyć jej za pośrednictwem wskaźników.

Tak naprawdę miałeś już do czynienia z przykładowym zastosowaniem wskaźnika. Kiedy przekazywaliśmy tablicę do funkcji, nie była ona kopiowana, prawda? Zamiast tego funkcja otrzymywała oryginalną tablicę. Proces ten odbywał się przy użyciu wskaźników. Jak widzisz, nie są one takie straszne.

Zanim jednak udamy się dalej, porozmawiamy jeszcze o pamięci.

## Czym jest pamięć komputera?

Pamięć komputera można bardzo łatwo wyobrazić sobie jako arkusz Excela. Arkusze kalkulacyjne składają się w zasadzie z dużej liczby „komórek”, z których każda może przechowywać pewne dane. Tym samym jest pamięć komputera: wielką liczbą sekwencyjnie ułożonych danych. W przeciwieństwie jednak do Excela, każda „komórka” pamięci może przechowywać bardzo mało danych, mianowicie 1 bajt, który może przyjąć tylko 256 możliwych wartości (od 0 do 255). Ponadto, w odróżnieniu od Excela, pamięć nie jest zorganizowana w postaci siatki, tylko „liniowo”. W rzeczy samej możesz uważać pamięć za bardzo długą tablicę typu char.

Tak jak w przypadku każdej komórki Excela istnieje sposób na jej zlokalizowanie (numer wiersza i litera kolumny), tak też każda komórka pamięci ma swój adres. Adres ten jest wartością, którą zapamiętuje wskaźnik, gdy ma przechować określona lokalizację w pamięci (w Excelu wskaźnik byłby komórką przechowującą adres innej komórki, na przykład gdyby w komórce C1 znajdowała się wartość A1).

Oto rysunek pokazujący, jak można wyobrazić sobie fragment pamięci komputera. Zwróć uwagę, że rysunek bardzo przypomina tablicę — tablica to po prostu zorganizowana sekwencyjnie pamięć:

0	4	8	12	16	20
??	16	??	??	??	??

Pola reprezentują lokalizacje w pamięci, w których można przechowywać dane, natomiast liczby to **adresy pamięci**, które pozwalają identyfikować te lokalizacje. Ich wartości zwiększą się co 4, ponieważ większość zmiennych zajmuje w pamięci po cztery bajty. Spoglądamy zatem na pamięć zajętą przez sześć różnych, czterabajtowych zmiennych<sup>1</sup> (przy okazji — bardzo często będziesz mieć do czynienia z adresami pamięci zapisanymi w kodzie szesnastkowym, który

<sup>1</sup> W zasadzie stwierdzenie to jest prawdziwe tylko w przypadku komputerów 32-bitowych (32 bity tworzą cztery bajty), w których większość operacji procesora przebiega na wartościach 4-bajtowych, chociaż nawet wtedy mamy do czynienia z częściową prawdą, gdyż pewne zmienne zajmują więcej niż cztery bajty (na przykład typ double). Nie wnioskajmy jednak teraz w szczegóły, ponieważ łatwiej będzie nam myśleć w nieco uproszczony sposób.

może Ci się wydawać nonsensowny, jeśli nie widziałeś go nigdy wcześniej; w książce tej jednak będę używał zwykłych liczb<sup>2</sup>).

W powyższym przykładzie można zauważyc, że pamięć pod adresem 4 przechowuje wartość 16, która może być kolejnym adresem pamięci. Pamięć pod tym adresem należy do zmiennej wskaźnikowej. Pozostałe wartości zostały oznaczone symbolem ??, co oznacza, że nie mają one żadnej określonej wartości, chociaż rzecz jasna przez cały czas pod adresami tymi coś się znajduje. Przed zainicjalizowaniem pamięci wartości te są jednak bezużyteczne — może tam istnieć cokolwiek.

## Zmienne a adresy

Zrozumienie różnicy między zmienną i adresem może sprawiać pewne problemy. Zmienna to reprezentacja pewnej wartości; wartość ta jest przechowywana w określonym miejscu pamięci, pod określonym adresem. Innymi słowy, kompilator korzysta z adresów pamięci, aby implementować zmienne w programie. Wskaźnik jest specjalnym rodzajem zmiennej, który umożliwia przechowywanie adresu dotyczącego innej zmiennej.

Najciekawsze jest to, że kiedy już będziesz dysponował adresem zmiennej, będziesz mógł się wybrać pod ten adres i odczytać dane, które są tam przechowywane. Jeśli zdarzy się, że zechcesz przekazać do funkcji dane o sporych rozmiarach, o wiele efektywniej będzie (podczas działania programu) przekazać samą informację o ich lokalizacji, zamiast kopiować każdy element tych danych — tak jak to się odbywa w przypadku tablic. Z takiej samej techniki możemy korzystać, aby uniknąć kopiowania struktur podczas przekazywania ich do funkcji. Cały pomysł polega na tym, że pobieramy adres, pod którym przechowywane są dane związane ze zmienną struktury, i zamiast kopiować wszystkie informacje zapisane w strukturze, przekazujemy do funkcji ten właśnie adres.

Najważniejszą rolą wskaźników jest udostępnianie w dowolnym momencie, za pośrednictwem systemu operacyjnego, większych ilości pamięci. W jaki sposób można uzyskać od systemu operacyjnego<sup>3</sup> tę pamięć? System operacyjny podaje adres pamięci, a w celu jego zapamiętania potrzebny jest wskaźnik. Jeśli w przyszłości będziesz potrzebować więcej pamięci, będziesz mógł o to poprosić, po czym zmienić adres, na który wskazujesz. W rezultacie wskaźniki umożliwiają wykraczanie poza stałą ilość pamięci, pozwalając na określanie podczas działania programu, ile pamięci potrzebujemy.

## Uwaga na temat nazewnictwa

Słowo „wskaźnik” może odnosić się do:

<sup>2</sup> Liczby szesnastkowe bazują na liczbie 16 i zwykle są zapisywane w postaci, która wygląda mniej więcej tak: 0x10ab0200, gdzie 0x informuje, że liczba ma format szesnastkowy, natomiast litery od a do f w jej dalszej części oznaczają liczby od 10 do 15.

<sup>3</sup> System operacyjny zarządza pamięcią, tak więc zdanie to jest zasadniczo prawdziwe, chociaż w rzeczywistości istnieje zazwyczaj kilka różnych „warstw” kodu obsługującego przydzielanie pamięci. System operacyjny jest jedną z tych warstw, ale wyżej istnieją kolejne warstwy. Na razie pominę to rozróżnienie, ponieważ wprowadza ono trochę zamieszania. Nie przejmuj się, jeżeli nie rozumiesz tego wszystkiego. Gdyby różnice te były ważne, nie wspominałbym o nich w przypisie. Nabiorą one sensu później, ale w tej chwili nie powinny mieć dla nas większego znaczenia.

- adresu w pamięci komputera,
- zmiennej, która przechowuje taki adres.

Zwykle rozróżnienie to nie jest aż tak istotne. Kiedy przesyłasz do funkcji zmienną wskaźnikową, tak naprawdę przekazujesz wartość zapisaną we wskaźniku, czyli adres pamięci komputera.

Pisząc o adresie pamięci, będę używał terminu „adres pamięci” lub po prostu „adres”, natomiast zmienną przechowującą taki adres będę nazywał „wskaźnikiem”.

Kiedy zmienna przechowuje adres innej zmiennej, będę pisał, że **wskazuje** ona tę zmienną.

## Organizacja pamięci

Skąd dokładnie bierze się pamięć? Dlaczego w ogóle trzeba o nią prosić system operacyjny?

W Excelu dysponujesz jedną, bardzo dużą grupą komórek, do której masz dostęp. W komputerze również masz do dyspozycji ogromną ilość pamięci, jednak jest ona bardziej ustrukturyzowana. Niektóre z jej fragmentów, które są dostępne dla Twojego programu, mogą być zajęte. Jedna z części pamięci komputera — używana do przechowywania wartości zmiennych, które są deklarowane w wykonywanych właśnie funkcjach — jest nazywana **stosem**. Nazwa ta wzięła się stąd, że podczas wywoływania kolejnych funkcji ich zmienne lokalne są odkładane w tym obszarze pamięci jedna na drugiej („w stos”). Wszystkie zmienne, z którymi pracowaliśmy do tej pory, były przechowywane na stosie.

Inna część pamięci nazywana jest **pamięcią dostępną** (lub czasami **sterią**). Jest to niealokowana pamięć, której fragmentami możesz dysponować. Ta część pamięci jest zarządzana przez system operacyjny. Kiedy fragment pamięci zostanie zadysponowany, powinien on być używany wyłącznie przez kod, który tę pamięć alokował, albo przez kod, któremu dany adres został przekazany przez alokator pamięci. Korzystanie ze wskaźników umożliwia uzyskanie dostępu do tak przydzielanej pamięci.

Możliwość siegania do pamięci komputera daje dużo władzy, ale z dużą władzą wiąże się duża odpowiedzialność. Pamięć komputera jest zasobem ograniczonym. Może nie w takim stopniu jak wtedy, gdy gigabajty RAM-u nie były jeszcze standardem, niemniej nadal ma ona swoje limity. Każdy fragment pamięci, który rezerwowałaś z dostępnych zasobów, powinien zostać do nich zwrócony, kiedy Twój program już go nie potrzebuje. Część kodu, która jest odpowiedzialna za zwracanie określonych fragmentów pamięci, jest nazywana **właścicielem** tej pamięci. Kiedy właściciel pamięci już jej nie potrzebuje (na przykład statek kosmiczny w grze został zniszczony), kod, który dysponuje tą pamięcią, powinien zwrócić ją do wolnych zasobów, dzięki czemu będzie mógł z niej korzystać inny kod. Jeśli pamięć nie zostanie zwrócona, Twojemu programowi zacznie w końcu jej brakować, co spowoduje spowolnienie jego działania lub nawet awarię. Być może słyszałeś narzekania, że przeglądarka internetowa Firefox zużywa zbyt dużo pamięci, co do tego stopnia spowalnia jej działanie, że wydaje się, jakby pełzała. Dzieje się tak dlatego, że ktoś nie oddawał z powrotem pamięci, która powinna zostać zwrócona, co powodowało problemy zwane **wyciekami pamięci**<sup>4</sup>.

---

<sup>4</sup> W obronie Firefoksa chciałbym nadmienić, że źródło części tych błędów w większym stopniu tkwiło w kiepsko napisanych rozszerzeniach (pisanych przez użytkowników dodatków do programu) niż w kodzie samego Firefoksa. Wynik końcowy pozostawał jednak taki sam — wyczerpywanie się pamięci powodowało poważne utrudnienia dla użytkowników.

Idea własności pamięci stanowi część interfejsu łączącego funkcję z obiektami z niej korzystającymi — pojęcie to nie jest częścią języka. Kiedy piszesz funkcję otrzymującą wskaźnik, powinieneś udokumentować, czy przejmuje ona pamięć na własność, czy nie. C++ nie zrobi tego za Ciebie i podczas działania programu nigdy nie zwolni pamięci, którą jawnie alokowałeś, chyba że jawnie tego zażadasz.

Fakt, że tylko określony kod powinien korzystać z określonej pamięci, wyjaśnia, dlaczego nie możesz tak po prostu do niej siegać. Co by się stało, gdybyś wygenerował liczbę losową i potraktował ją jako adres pamięci? Technicznie mógłbyś tak zrobić, ale nie byłby to dobry pomysł. Nie wiesz, co alokuje tę pamięć — może to być nawet sam stos. Kiedy więc zmodyfikujesz zawartość takiej pamięci, zniszczysz dane, które się tam znajdowały! Aby ułatwić wykrywanie tego typu zdarzeń, system operacyjny chroni pamięć, która nie została przekazana Ci do użycia. Taka pamięć jest **nieprawidłowo zaalokowana** i próba uzyskania do niej dostępu spowoduje awarię programu, co umożliwi Ci wykrycie problemu<sup>5</sup>.

Poczekaj chwilę! Czy nie zasugerowałem właśnie, że awaria programu to zdarzenie pomyślne? Przecież tak jest! Awarie spowodowane próbą dostępu do nieprawidłowo zaalokowanej pamięci prawie zawsze są łatwiejsze do zdiagnozowania niż błędy wynikające z wpisania niewłaściwych danych do poprawnie zaalokowanej pamięci. Zwykle tego typu awarie następują dość szybko, ponieważ problem z alokacją pojawia się bezzwłocznie. Jeśli zmienisz prawidłowo zaalokowaną pamięć, której nie jesteś właściwym, błęd nie pojawi się, dopóki kod dysponujący prawem do tej pamięci nie spróbuje z niej skorzystać, co może nastąpić znacznie później, długo po zapisaniu w tej pamięci danych. Mój kolega z pracy wyjaśnił to następująco: „Właśnie w samochodzie odpadło koło, ale śruba, która je trzymała, urwała się kilometr wcześniej”. Spróbuj poszukać tej śruby. Powodzenia!

Niektóre osoby mogą Ci powiedzieć, że awarie powodowane niepoprawną alokacją pamięci są naprawdę trudne do zdiagnozowania. No cóż, nie czytały one tej książki. W rozdziale „Debugowanie w Code::Blocks” opiszę, jak niemal bezzwłocznie odnajdować awarie, których przyczyną jest zła alokacja pamięci.

## Nieprawidłowe wskaźniki

Jedną z przyczyn powodujących przypadkowe użycie niepoprawnie zaalokowanej pamięci jest zastosowanie wskaźnika bez jego zainicjalizowania. Kiedy deklarujesz wskaźnik, początkowo będą się w nim znajdować przypadkowe dane. Będzie on wskazywać jakieś miejsce w pamięci, które może być poprawne, chociaż wcale takie być nie musi. Korzystanie z tego miejsca z pewnością jest niebezpieczne. Faktycznie jego zawartością równie dobrze mógłaby być liczba losowa. Użycie tej wartości spowoduje awarię lub uszkodzenie danych. Zawsze przed użyciem wskaźników musisz je zainicjalizować!

<sup>5</sup> Przy okazji losowego generowania adresów pamięci muszę dodać, że istnieje jeszcze jeden mały problem — adresy pamięci zazwyczaj muszą być odpowiednio wyrównane. Aby uzyskać dostęp do typu całkowitego, adres pamięci powinien być wielokrotnością czwórki (4, 8, 12, 16 itd.). Jeśli wygenerujesz adres losowo, będziesz musiał go odpowiednio wyrównać. Wymogi związane z wyrównywaniem pamięci różnią się w zależności od architektury systemu, ale zwykle obowiązują ze względów wydajnościowych.

## Pamięć i tablice

Czy pamiętasz, jak pisałem, że wypadnięcie poza koniec tablicy spowoduje problemy? Teraz, kiedy mamy już trochę więcej informacji o pamięci, możemy zrozumieć, dlaczego tak się dzieje. Z tablicą jest skojarzona pewna ilość pamięci, która zależy od rozmiaru tej tablicy. Jeżeli sięgniesz do elementu znajdującego się poza tablicą, uzyskasz dostęp do pamięci, która do niej nie należy — pamięć ta nie jest elementem tablicy. To, co się w tym miejscu znajduje, zależy od kodu oraz implementacji kompilatora, ale nie stanowi części tablicy. Korzystanie z takiej pamięci z pewnością spowoduje problemy.

## Pozostałe zalety i wady wskaźników

Teraz, kiedy masz już nieco więcej szczegółowych informacji o wskaźnikach, możemy powrócić do naszej wcześniejszej analogii i przyjrzeć się niektórym kompromisom, które wiążą się z ich użyciem. Hiperłącza i wskaźniki mają sporo wspólnych zalet i wad. Zaczniemy od tych pierwszych:

1. Nie musisz robić kopii — jeśli strona jest duża albo złożona, skopiowanie jej może być trudne (wyobraź sobie próbę wysłania komuś Wikipedii!). Podobnie, dane w pamięci komputera również mogą być dość skomplikowane, a sporządzenie ich wiernej kopii może być trudne (o czym będzie mowa w dalszej części tej książki) albo czasochłonne (kopianie dużych ilości pamięci może trwać bardzo długo).
2. Nie musisz przejmować się, czy masz najnowszą wersję strony. Jeżeli autor zaktualizuje swoją stronę, dowieš się o zmianach, kiedy ponownie ją odwiedzisz. Dysponując wskaźnikiem do pamięci, zawsze będziesz mieć możliwość odczytania bieżącej wartości zapisanej pod danym adresem.

Oczywiście są też wady przesyłania łącza zamiast kopii:

1. Strona może zostać przeniesiona albo skasowana. Podobnie, pamięć może zostać zwolniona przez system operacyjny, chociaż nadal istnieje do niej wskaźnik. Aby uniknąć takich sytuacji, kod, który jest właścicielem pamięci, musi sprawdzać, czy nie korzysta z niej jakiś inny obiekt.
2. Aby mieć dostęp do strony, musisz być online. Warunek ten w zasadzie nie dotyczy wskaźników.

Myślenie o wskaźnikach jak o łączach na stronie internetowej powinno pomóc Ci zrozumieć, kiedy z nich korzystać, chociaż analogia ta wiąże się z kilkoma problemami. Po pierwsze hiperłącza i strony internetowe to dwie różne rzeczy, podczas gdy wskaźniki i zmienne są tym samym. Co mam na myśli? Wskaźnik to tylko inny rodzaj zmiennej (choć ma on szczególnie właściwości), a hiperłącze nie będzie stroną internetową, nawet gdybyś się bardzo o to starał. Z drugiej jednak strony wskaźnik jest innym *typem* zmiennej, tak samo jak hiperłącze jest innym obiektem niż witryna internetowa.

Czy udało Ci się wszystko do tej pory zrozumieć? Obiecałem, że podzielę cały materiał o wskaźnikach na wiele krótkich rozdziałów, aby Twój mózg mógł robić sobie przerwy. Nadeszła zatem pora na zakończenie bieżącego rozdziału. A skoro poznaleś już część kluczowych idei, które są Ci potrzebne, w następnym rozdziale zajmę się szczegółami stosowania wskaźników.

## Sprawdź się

- 1.** Która z poniższych sytuacji NIE jest dobrym powodem do zastosowania wskaźnika?
  - A. Chcesz pozwolić funkcji, aby modyfikowała argument, który zostanie do niej przekazany.
  - B. Chcesz zaoszczędzić pamięć i uniknąć kopiowania dużej zmiennej.
  - C. Chcesz mieć możliwość uzyskania pamięci od systemu operacyjnego.
  - D. Chcesz mieć możliwość szybszego odczytywania zmiennych.
- 2.** Co przechowuje wskaźnik?
  - A. Nazwę innej zmiennej.
  - B. Wartość całkowitą.
  - C. Adres pamięci innej zmiennej.
  - D. Adres pamięci, ale niekoniecznie innej zmiennej.
- 3.** Skąd możesz uzyskać więcej pamięci podczas działania programu?
  - A. Nie można uzyskać więcej pamięci.
  - B. Ze stosu.
  - C. Z dostępnej pamięci.
  - D. Poprzez zadeklarowanie kolejnej zmiennej.
- 4.** Co niekorzystnego może się wydarzyć podczas stosowania wskaźników?
  - A. Możesz siegnąć do pamięci, z której nie powinieneś korzystać, co doprowadzi do awarii programu.
  - B. Możesz siegnąć pod niewłaściwy adres pamięci, co spowoduje zniszczenie danych.
  - C. Możesz zapomnieć o zwróceniu pamięci do systemu operacyjnego, przez co w programie skończy się pamięć.
  - D. Każde zdarzenie z powyższych.
- 5.** Skąd bierze się pamięć używana przez zwykłą zmienną zadeklarowaną w funkcji?
  - A. Z dostępnej pamięci.
  - B. Ze stosu.
  - C. Zwykłe zmienne nie używają pamięci.
  - D. Z samego programu — to dlatego pliki EXE są tak duże!
- 6.** Co powinieneś zrobić po zaalokowaniu pamięci?
  - A. Nic, ta pamięć już na zawsze będzie Twoja.
  - B. Zwrócić ją do systemu operacyjnego, kiedy nie będzie już potrzebna.
  - C. Nadać wskazywanej zmiennej wartość 0.
  - D. Zapisać we wskaźniku wartość 0.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Weź jakiś niewielki program, który wcześniej napisałeś — może to być program z zadań praktycznych z jednego z wcześniejszych rozdziałów tej książki. Przyjrzyj się wszystkim jego zmiennym i wyobraź sobie, że każda z nich ma przypisane swoje miejsce w pamięci.

Spróbuj narysować dla nich schemat pudełkowy (podobny do rysunku, który sporządziłem na początku rozdziału), który ukazywałby powiązanie każdej zmiennej z pamięcią komputera. Zastanów się, w jaki sposób mógłbyś przedstawić serię zmiennych, które nie stanowią części jednej tablicy, ale są usytuowane w pamięci jedna za drugą.

- 2.** Zastanów się, ile „przegródek” w pamięci potrzebuje następujący program:

```
int main ()  
{  
    int i;  
    int glosy[ 10 ];  
}
```

Czy z całkowitą pewnością możesz powiedzieć coś na temat miejsc w pamięci zajmowanych przez zmienne glosy[ 0 ], glosy[ 9 ] oraz i? Podpowiedź: możesz nie wiedzieć, gdzie znajduje się zmienna i, ale wiesz, gdzie jej na pewno nie ma. Spróbuj narysować możliwe konfiguracje pamięci w tym programie.

# 13

## ■ ■ ■ R O Z D Z I A Ł 1 3

## Korzystanie ze wskaźników

---

Z lektury poprzednich rozdziałów wiesz już, czym jest pamięć i jak powinieneś o niej myśleć; ale jak pisać kod, który będzie mógł z tej pamięci korzystać? W tym rozdziale, przy pomocy wielu rysunków, przedstawię składnię potrzebną do pracy ze wskaźnikami. Następnie omówię kilka podstawowych przykładów zastosowania wskaźników w rzeczywistych programach. Nie zabierzemy się jeszcze za pozyskiwanie pamięci z wolnych zasobów — o tym będzie traktować kolejny rozdział — ale dostaniemy do dyspozycji wszystkie narzędzia, które będą do tego potrzebne.

## Składnia wskaźników

### Deklarowanie wskaźnika

W C++ istnieje specjalna składnia służąca do deklarowania zmiennych wskaźnikowych. Informuje ona, że zmienna jest wskaźnikiem, oraz określa, jaki typ pamięci ta zmienna wskazuje.

Deklaracja wskaźnika wygląda następująco:

```
<typ> *<nazwa_wsk>;
```

Przy pomocy poniższej składni można na przykład zadeklarować wskaźnik, który przechowuje adres zmiennej całkowitej:

```
int *w_wskaznik_calkowity;
```

Zwróci uwagę na użycie znaku \*. Przy deklarowaniu wskaźnika jest on kluczowy. Jeśli umieszczisz go przed nazwą zmiennej, deklarujesz, że zmienna ta ma być wskaźnikiem. Z dowolnej strony gwiazdki można wstawić odstęp. Następujące dwie deklaracje są równorzędne:

```
int *w_wskaznik_calkowity;
```

oraz

```
int* w_wskaznik_calkowity;
```

Prefiks `w_` nie jest wymagany w C++, ale zawsze z niego korzystam, aby było wiadomo, że zmienna jest wskaźnikiem. Mała uwaga: jeśli deklarujesz wiele wskaźników w tym samym wierszu, każdą nazwę zmiennej powinieneś poprzedzić gwiazdką:

```
// Jeden wskaźnik i jedna zwykła zmienna  
int *w_wskaznik1, niewskaznik1;
```

```
// Dwa wskaźniki  
int *w_wskaznik1, *w_wskaznik2;
```

Być może zastanawiasz się, czy nie dałoby się tego zrobić w jakiś prostszy sposób, na przykład za pomocą instrukcji pointer `w_wskaznik`. Powód jest następujący: aby kompilator mógł użyć adresu pamięci, musi wiedzieć, jaki typ danych znajduje się pod tym adresem; w przeciwnym razie nie będzie mógł tych danych poprawnie zinterpretować (na przykład ten sam bajt ma różne znaczenie w typach `double` oraz `int`). Zamiast stosować oddzielne nazwy wskaźników dla każdego typu (`int_wsk`, `char_wsk` itd.), lepiej w celu otrzymania wskaźnika postawić gwiazdkę przy nazwie typu zmiennej.

## Otrzymywanie adresu zmiennej za pomocą wskaźnika

Chociaż wskaźniki mogą być używane do odczytywania adresu nowej pamięci, zaczniemy od zobaczenia, jak z nich korzystać podczas pracy z istniejącymi zmiennymi. Aby otrzymać adres zmiennej (czyli jej lokalizację w pamięci), należy przed jej nazwą postawić znak `&`, który jest nazywany **operatorem adresu**, gdyż zwraca on adres zmiennej:

```
int x;
int *w_x = & x;
*w_x = 2; // Inicjalizacja x wartością 2
```

Korzystanie z operatora adresu bardziej przypomina spojrzenie na pasek adresu w przeglądarce w celu odczytania adresu strony niż oglądanie samej zawartości tej strony.

Powinno być zrozumiałe, że w celu otrzymania miejsca w pamięci, w którym zapisana jest zmienna, należy zrobić coś specjalnego — przecież w większości przypadków wszystko, czego żądamy od zmiennej, to jej bieżąca wartość.

## Użycie wskaźnika

Użycie wskaźnika wymaga nowej składni, ponieważ posługując się wskaźnikiem, należy mieć możliwość zrobienia dwóch różnych rzeczy, którymi są:

- pobranie lokalizacji pamięci, którą przechowuje wskaźnik,
- pobranie wartości przechowywanej w tej lokalizacji.

Kiedy użyjesz wskaźnika, jakby był zwykłą zmienną, dostaniesz lokalizację w pamięci, która jest zapisana w tym wskaźniku.

Poniższy fragment kodu spowoduje wyświetlenie adresu zmiennej `x`, który jest wskazywany (przechowywany) przez wskaźnik `w_wskaznik_calkowity`:

```
int x = 5;
int *w_wskaznik_calkowity = & x;
cout << w_wskaznik_calkowity; // Wyświetla adres zmiennej x
// Zapis ten jest równoważny instrukcji cout << &x;
```

Aby uzyskać dostęp do pamięci w tej lokalizacji, należy użyć symbolu `*`. Oto przykład, w którym inicjalizowany jest wskaźnik wskazujący na inną zmienną:

```
int x = 5;
int *w_wskaznik_calkowity = & x;
cout << *w_wskaznik_calkowity; // Wyświetla wartość 5
// Zapis ten jest równoważny instrukcji cout << x;
```

Zapis `*w_wskaznik_calkowity` oznacza: „odczytaj wskaźnik i pobierz wartość przechowywaną w pamięci pod adresem wskazywanym przez ten wskaźnik”. Ponieważ w naszym przypadku wskaźnik `w_wskaznik_calkowity` odnosi się do zmiennej `x`, a w zmiennej tej zapisana jest wartość 5, na ekranie zostanie wyświetcone 5.

W łatwy sposób można zapamiętać, że symbol `*` jest używany do odczytywania wartości zapisanej we wskazywanym miejscu dzięki temu, że zmienna wskaźnikowa działa tak samo jak zwykła zmienna — aby uzyskać zapisaną w niej wartość, należy po prostu użyć nazwy tej zmiennej. Przechowywana w niej wartość jest adresem pamięci. Jeśli chcesz zrobić coś wyjątkowego i niezwykłego, czyli pobrać wartość przechowywaną pod tym adresem, musisz skorzystać ze specjalnej składni. Pomyśl, że gwiazdka oznacza coś niezwykłego — ktoś mógłby postawić taką gwiazdkę na przykład przy wyniku meczu, który wygrała polska reprezentacja piłkarska.

Korzystanie z operatora `*` w celu pobrania wartości spod wskazywanego adresu jest nazywane **derefencją wskaźnika**. Określenie to wzięło się stąd, że aby uzyskać potrzebną wartość, należy pobrać referencję do jakiegoś adresu w pamięci i tam przejść.

Derefencja zmiennej umożliwia także zapisywanie wartości pod wskazanym adresem.

```
int x;
int *w_wskaznik_calkowity = & x;
*w_wskaznik_calkowity = 5; // Teraz x wynosi 5
cout << x;
```

Można pogubić się w tym, kiedy należy użyć znaku `*`, a kiedy `&`. Oto tabelka, do której możesz zaglądać w przypadku wątpliwości:

Czynność	Potrzebny znak	Przykład
Deklaracja wskaźnika	<code>*</code>	<code>int *w_x;</code>
Odczytanie adresu zapisanego we wskaźniku	Żaden	<code>cout &lt;&lt; w_x;</code>
Zapisanie adresu we wskaźniku	Żaden	<code>int *w_x; w_x = /* adres */;</code>
Odczytanie wartości spod adresu	<code>*</code>	<code>cout &lt;&lt; *w_x;</code>
Zapisanie nowej wartości pod tym adresem	<code>*</code>	<code>*w_x = 5;</code>
Deklaracja zmiennej	Żaden	<code>int y;</code>
Odczytanie wartości zapisanej w zmiennej	Żaden	<code>int y; cout &lt;&lt; y;</code>
Nadanie wartości zmiennej	Żaden	<code>int y; y = 5;</code>
Odczytanie adresu zmiennej	<code>&amp;</code>	<code>int y; int *w_x; w_x = &amp; y;</code>
Zmiana adresu zmiennej	Nie dotyczy	Brak możliwości — nie można zmienić adresu zmiennej.

Ponadto mam jeszcze dwie proste reguły:

**Wskaźnik przechowuje adres, w związku z czym, kiedy użyjesz „gołego” wskaźnika, dostaniesz adres. Aby otrzymać albo zmienić wartość przechowywaną pod tym adresem, musisz dodać coś ekstra — gwiazdkę.**

**Zmienna przechowuje wartość, w związku z czym, kiedy użyjesz zmiennej, dostaniesz wartość. Aby otrzymać adres tej zmiennej, musisz dodać coś ekstra — znak `&`.**

Spójrzmy teraz na krótki program ilustrujący powyższe reguły i przyjrzyjmy się przydatnym technikom, które służą do sprawdzania, co się dzieje w pamięci komputera:

### Przykładowy kod 34.: wskaznik.cpp

```
#include <iostream>

using namespace std;

int main ()
{
    int x; // Zwykła zmieniona całkowita
    int *w_int; // Wskaźnik do zmiennej całkowitej

    w_int = & x; // Czytaj: „przypisz adres zmiennej x do wskaźnika w_int”
    cout << "Podaj liczbę: ";
    cin >> x; // Wczytaj wartość do x; można tu użyć także zapisu *w_int
    cout << *w_int << '\n'; // Zwróć uwagę na użycie * w celu odczytania wartości
    *w_int = 10;
    cout << x; // Znowu wyświetla 10
}
```

Pierwsza instrukcja cout wyświetla wartość znajdująca się w zmiennej x. Dlaczego? Przeanalizujmy krok po kroku program i zobaczymy, jaki wpływ wywiera on na pamięć komputera. Za pomocą strzałek oznaczmy, co wskazuje wskaźnik, natomiast w przypadku zwykłych (niewskaźnikowych) zmiennych pokażemy ich wartości.

Zacznijmy od zmiennej całkowitej x oraz wskaźnika całkowitego w\_int.

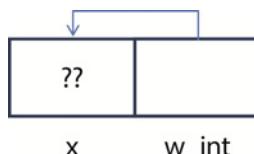
Można to sobie wyobrazić jako dwie zmienne (być może jedna obok drugiej) o nieznanych wartościach.



Następnie za pomocą operatora adresu & kod programu zapisuje lokalizację zmiennej x we wskaźniku w\_int, dzięki czemu uzyskuje adres tej zmiennej.

```
w_int = & x; // Czytaj: „przypisz adres zmiennej x do wskaźnika w_int”
```

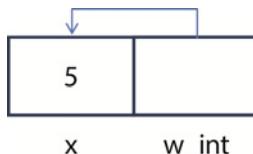
Możemy narysować strzałkę prowadzącą od zmiennej wskaźnikowej w\_int do zmiennej całkowitej x, aby zaznaczyć, że w\_int wskazuje x.



Teraz użytkownik wprowadza liczbę, która zostanie zapisana w zmiennej x; jest to lokalizacja wskazywana przez w\_int.

```
cin >> x; // Wczytaj wartość do x; można tu użyć także zapisu *w_int
```

Załóżmy, że użytkownik wprowadził 5, co można zobrazować następująco:



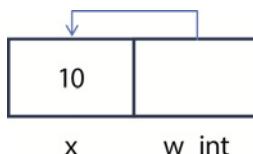
W następnym wierszu wskaźnik *\*w\_int* zostaje przekazany do cout. Zapis *\*w\_int* oznacza dereferencję wskaźnika *w\_int*; następuje sprawdzenie adresu zapisanego w *w\_int*, przejście pod ten adres i zwrot przechowywanej tam wartości, co można sobie wyobrazić jako przejście programu wzdłuż strzałki widocznej na rysunku.

```
cout << *w_int << '\n'; // Zwróć uwagę na użycie * w celu odczytania wartości
```

Ostatnia para wierszy pokazuje, że modyfikacja wskaźnika zmienia także oryginalną zmienianą — w pamięci wskazywanej przez *w\_int* (gdzie przechowywana jest wartość zmiennej *x*) zostaje zapisana wartość 10:

```
*w_int = 10;
```

Teraz stan pamięci wygląda następująco:



Mam nadzieję, że przekonałeś się, iż korzystanie z takich schematów ułatwia prześledzenie, co zachodzi w programie podczas stosowania wskaźników. Gdy natrafisz na problemy ze zrozumieniem, co się dzieje, narysuj początkowy stan pamięci, a następnie prześledź strzałka po strzałce działanie programu, kreśląc przy tym zachodzące zmiany. Jeśli wskaźnik zmieni miejsce, które wskazuje, narysuj nową strzałkę. Jeżeli zmieniona uzyska nową wartość, odnotuj to na schemacie. W ten sposób będziesz mógł ogarnąć i zrozumieć nawet skomplikowane systemy.

## Niezainicjalizowane wskaźniki i wartość NULL

Zwróć uwagę, że w naszym przykładzie wskaźnik (*w\_int*) jest inicjalizowany określonym adresem pamięci, jeszcze zanim zostanie użyty. W przeciwnym razie wskazywałby on cokolwiek, co mogłoby prowadzić do bardzo nieprzyjemnych konsekwencji, jak nadpisanie pamięci, w której przechowywana jest jakaś inna wartość, albo wywołanie awarii programu. Aby uniknąć takich sytuacji, zawsze powinieneś inicjalizować wskaźniki przed ich użyciem.

Czasami jednak chciałbyś mieć możliwość powiedzenia, że pewien wskaźnik celowo nie został jeszcze zainicjalizowany. W C++ istnieje specjalna wartość służąca do jawnego oznaczania wskaźnika jako niezainicjalizowanego. Tą wartością jest **NULL**. Jeśli wskaźnik wskazuje adres **NULL** (przechowuje tę wartość), wiadomo, że nie został zainicjalizowany. Za każdym razem, gdy tworzysz nowy wskaźnik, przypisz mu wartość **NULL**, co umożliwi Ci późniejsze sprawdzenie, czy została mu nadana jakaś użyteczna wartość. W przeciwnym wypadku nie będziesz mieć możliwości przekonania się, czy wskaźnik nadaje się do użycia bez ryzykowania awarii programu:

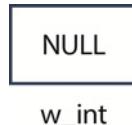
```

int *w_int = NULL;

// Kod, który może (albo i nie) nadać wartość wskaźnikowi w_int
if ( w_int != NULL )
{
    *w_int = 2;
}

```

Aby na schemacie pamięci uwzględnić wskaźnik NULL, zamiast rysować strzałkę, napisz po prostu NULL w odpowiednim polu:



## Wskaźniki i funkcje

Wskaźniki pozwalają na przekazywanie adresu lokalnej zmiennej do funkcji, która może tę zmienną modyfikować. Prawie każdy autor podaje przykład, w którym para prostych funkcji próbuje zamienić miejscami dwie wartości przechowywane w dwóch zmiennych:

### Przykładowy kod 35.: zamiana.cpp

```

#include <iostream>

using namespace std;

void zamien1 (int lewy, int prawy)
{
    int tymczas = lewy;
    lewy = prawy;
    prawy = tymczas;
}

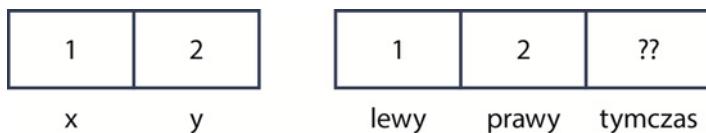
void zamien2 (int *w_lewy, int *w_prawy)
{
    int tymczas = *w_lewy;
    *w_lewy = *w_prawy;
    *w_prawy = tymczas;
}

int main ()
{
    int x = 1, y = 2;
    zamien1( x, y );
    cout << x << " " << y << '\n';
    zamien2( &x, &y );
    cout << x << " " << y << '\n';
}

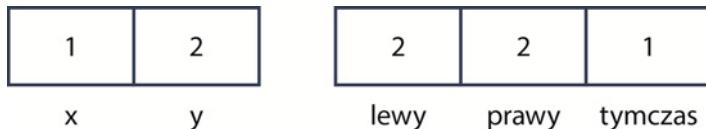
```

Zastanów się przez chwilę i postaraj się wskazać funkcję, która prawidłowo zamienia miejscami dwie wartości.

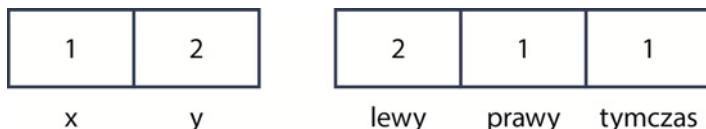
Zgadza się! Funkcja zamien1 zamienia miejscami tylko swoje zmienne lokalne. Nie ma ona wpływu na wartości do niej przekazywane, ponieważ są to tylko kopie oryginalnych wartości (przechowywanych w zmiennych x i y). Na schemacie można zobaczyć, że wywołanie tej funkcji powoduje przekopiowanie zmiennych x oraz y do zmiennych lewy i prawy:



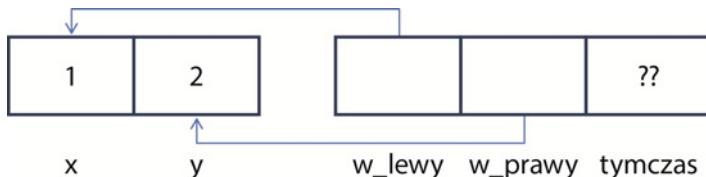
Teraz wartość zmiennej `lewy` jest zapisywana w zmiennej `tymczas`, natomiast zmiennej `lewy` jest przypisywana wartość zmiennej `prawy`:



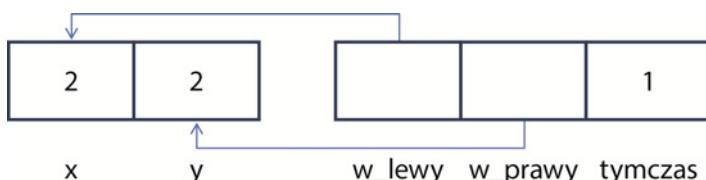
Na koniec wartość zmiennej `tymczas` jest kopiwana do zmiennej `prawy`, co powoduje zamianę wartości zmiennych `lewy` i `prawy`, przy czym zmienne `x` i `y` pozostają niezmienione:



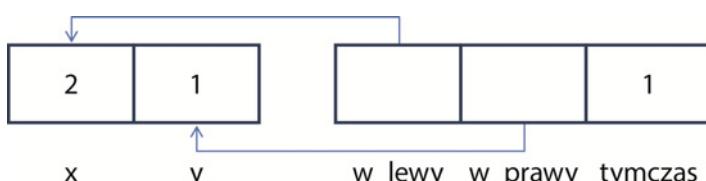
Funkcja `zamien2` jest bardziej interesująca. Pobiera ona adresy zmiennych lokalnych `x` i `y`. Zmienne `w_lewy` i `w_prawy` wskazują teraz zmienne `x` oraz `y`:



W ten sposób funkcja uzyskała dostęp do pamięci, w której przechowywane są obie zmienne. Kiedy zatem wykonuje ona zamianę, modyfikuje pamięć zmiennych `x` i `y` — najpierw kopiuje wartość wskazywaną przez `w_lewy` do zmiennej `tymczas`, po czym kopiuje do zmiennej `w_lewy` wartość wskazywaną przez `w_prawy`:



Zauważ, że tym razem pamięć przechowującą zmienną `x` została zmodyfikowana. Na koniec wartość zapisana w zmiennej `tymczas` jest przypisywana pamięci wskazywanej przez `w_prawy`, co kończy proces zamiany:



Możliwość zamieniania wartości zmiennych w taki sposób nie jest jednak najważniejszą zaletą wskaźników. Istnieje inna funkcjonalność języka C++, która ułatwia pisanie tego typu funkcji bez potrzeby pełnego zagłębiania się w działanie wskaźników. Funkcjonalnością tą są referencje.

## Referencje

Czasami będą Ci potrzebne niektóre z możliwości udostępnianych przez wskaźniki (jak na przykład brak konieczności kopiowania dużych ilości danych), chociaż nie będziesz potrzebować ich pełnej funkcjonalności. W takich sytuacjach możesz użyć **referencji**. Referencja to zmienna, która odwołuje się do innej zmiennej, dzieląc z nią tę samą pamięć. Z referencji można korzystać tak jak ze zwykłych zmiennych. O referencji można myśleć jak o uproszczonym wskaźniku, podczas stosowania którego nie ma potrzeby stosowania specjalnej gwiazdki ani znaku & w celu odczytania wartości, do której referencja się odnosi. Referencje, w przeciwieństwie do wskaźników, zawsze muszą odwoływać się do poprawnych adresów. Są one deklarowane następująco:

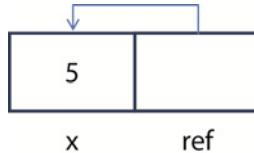
```
int &ref;
```

Powyższa deklaracja jest jednak nieprawidłowa, ponieważ referencje zawsze muszą być inicjalizowane (referencja zawsze musi odwoływać się do poprawnego adresu).

```
int x = 5;
int &ref = x; // Zwróć uwagę, że przed zmienną x nie musisz stawać znaku &
```

Referencję można pokazać na schemacie w taki sam sposób jak wskaźnik. Różnica polega na tym, że podczas stosowania referencji zamiast adresu pamięci otrzymuje się wartość zapisaną pod tym adresem.

```
int x = 5;
int &ref = x;
```



Na powyższym rysunku pamięć zmiennej ref przechowuje wskaźnik do zmiennej x. Komplator wie, że kiedy napiszesz samo ref, tak naprawdę masz na myśli wartość przechowywaną we wskazywanej lokalizacji w pamięci. W pewnym sensie referencje to wskaźniki o przeciwnym zachowaniu „domyślnym”, które jest realizowane, gdy w kodzie wprowadzasz nazwę zmiennej.

Z referencji można korzystać w celu przekazywania struktur do funkcji bez konieczności kopiowania całej struktury i bez potrzeby przejmowania się wskaźnikami NULL.

```
struct mojaDuzaStruktura
{
    int x[ 100 ]; // Duża struktura, zajmująca mnóstwo pamięci!
};

void pobierzStrukture (mojaDuzaStruktura& moja_struktura)
{
    moja_struktura.x[ 0 ] = 23;
}
```

Ponieważ struktury zawsze odnoszą się do oryginalnych obiektów, podczas przekazywania tych obiektów do funkcji unikasz ich kopiowania, a także możesz je modyfikować. Powyższy przykład pokazuje, jak można zmodyfikować zawartość elementu `moja_struktura.x[ 0 ]`, dzięki czemu oryginalna struktura przekazana do funkcji będzie zawierać wartość 23, gdy zostanie z tej funkcji zwrocona.

Niedawno poznaliśmy funkcję zamieniającą przy użyciu wskaźników wartości zmiennych; zobaczymy teraz, jak można napisać taką funkcję jeszcze prościej przy wykorzystaniu referencji.

```
void zamien ( int& lewy, int& prawy )
{
    int tymczas = prawy;
    prawy = lewy;
    lewy = tymczas;
}
```

Zwrć uwagę, że powyższa funkcja jest znacznie prostsza niż jej wskaźnikowy odpowiednik. Referencję rzeczywiście można traktować jak zastępstwo oryginalnej wartości. Oczywiście kompilator podczas implementacji referencji posługuje się wskaźnikami; samo odczytywanie danych, czyli dereferencja, jest już wykonywane za Ciebie.

## Referencje a wskaźniki

Referencje zastępują wskaźniki, kiedy potrzebujesz odwołać się do pewnej zmiennej przy użyciu wielu nazw, na przykład gdy chcesz przekazać do funkcji argumenty bez ich kopiowania albo kiedy funkcja powinna modyfikować swoje parametry w sposób widoczny dla obiektów ją wywołujących.

Referencje nie zapewniają takiej elastyczności jak wskaźniki, ponieważ zawsze muszą wskazywać poprawny adres. Nie ma możliwości, aby zawierały wartość `NULL`; za pomocą referencji nie da się zapisać, że adres nie jest dokładnie określony — referencje nie zostały po prostu w tym celu zaprojektowane. Ponieważ referencje nie mogą przyjmować wartości `NULL`, nie pozwalają one na tworzenie złożonych struktur danych. Strukturom danych poświęcimy o wiele więcej miejsca w kilku kolejnych rozdziałach; przy różnych okazjach zadaj sobie wówczas pytanie, czy mógłbyś osiągnąć to samo, korzystając z referencji.

Kolejna różnica polega na braku możliwości zmiany miejsca w pamięci, które wskazuje referencja po jej zainicjalizowaniu. Referencja trwale odwołuje się do tej samej zmiennej, co także ogranicza możliwości jej stosowania podczas budowania złożonych struktur danych.

W dalszej części tej książki będę korzystał z referencji, gdy przyjęcie takiego rozwiązania okaże się właściwe; prawie zawsze wtedy, gdy funkcja pobiera jako argument instancję struktury (albo klasy, gdy już je omówię). Wzorzec taki niemal zawsze będzie wyglądać podobnie jak w poniższym przykładzie:

```
void (mojTypStruktury& arg);
```

## Sprawdź się

- 1.** Który z poniższych zapisów poprawnie definiuje wskaźnik?
  - A. int x;
  - B. int &x;
  - C. wsk x;
  - D. int \*x;
- 2.** Który z poniższych zapisów podaje adres pamięci zmiennej całkowej a?
  - A. \*a;
  - B. a;
  - C. &a;
  - D. address( a );
- 3.** Który z poniższych zapisów podaje adres pamięci zmiennej wskazywanej przez wskaźnik w\_a?
  - A. w\_a;
  - B. \*w\_a;
  - C. &w\_a;
  - D. address( w\_a );
- 4.** Który z poniższych zapisów podaje wartość przechowywaną pod adresem wskazywanym przez wskaźnik w\_a?
  - A. w\_a;
  - B. wart( w\_a );
  - C. \*w\_a;
  - D. &w\_a;
- 5.** Który z poniższych zapisów poprawnie definiuje referencję?
  - A. int \*w\_int;
  - B. int &moja\_ref;
  - C. int &moja\_ref = & moja\_oryg\_wart;
  - D. int &moja\_ref = moja\_oryg\_wart;
- 6.** Która z poniższych sytuacji nie jest dobrą okazją do użycia referencji?
  - A. Przechowywanie adresu, który został dynamicznie zaalokowany w wolnej pamięci.
  - B. Uniknięcie kopiowania dużej wartości podczas przekazywania jej do funkcji.
  - C. Wymuszenie, aby parametr przekazywany do funkcji nigdy nie przyjmował wartości NULL.
  - D. Zezwolenie funkcji na dostęp do oryginalnej zmiennej, która jest do niej przekazywana, bez konieczności korzystania ze wskaźników

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Napisz funkcję, która prosi użytkownika o podanie w dwóch osobnych zmiennych imienia i nazwiska. Funkcja ta powinna zwracać obie wartości za pośrednictwem dodatkowych parametrów wskaźnikowych (lub referencji) przekazywanych do niej podczas jej wywołania.

Najpierw postaraj się napisać taką funkcję, korzystając ze wskaźników, a następnie przy użyciu referencji (podpowiedź: sygnatura tej funkcji będzie podobna do sygnatury funkcji zamieniającej ze sobą wartości dwóch zmiennych z wcześniejszej części tego rozdziału).

- 2.** Narysuj schemat podobny do rysunku demonstrującego działanie funkcji zamieniającej wartości zmiennych, z tym że dotyczący funkcji z zadania 1.
- 3.** Zmodyfikuj program napisany w ramach pierwszego zadania w taki sposób, aby prosił użytkownika o podanie nazwiska tylko wtedy, gdy w parametrze dotyczącym nazwiska funkcja otrzymała wskaźnik o wartości `NULL`.
- 4.** Napisz funkcję, która pobiera dwa argumenty i zwraca dwa odrębne wyniki. Jednym z wyników powinien być iloczyn obu argumentów, a drugim ich suma. Ponieważ funkcja może bezpośrednio zwracać tylko jedną wartość, druga wartość powinna być zwracana poprzez parametr wskaźnikowy albo referencję.
- 5.** Napisz program, który porównuje adresy pamięci dwóch różnych zmiennych na stosie i wyświetla zmienne na podstawie kolejności liczbowej ich adresów. Czy kolejność ta budzi Twoje zdziwienie?



## Dynamiczna alokacja pamięci

---

Jeśli udało Ci się przebrnąć przez kilka ostatnich rozdziałów, to znaczy, że całkiem nieźle dajesz sobie radę. Teraz zabierzemy się za aspekt wskaźników dający największą frajdę, czyli za korzystanie z nich podczas rozwiązywania prawdziwych problemów. Jak już dawałem do zrozumienia, nareszcie jesteśmy gotowi, aby się dowiedzieć, w jaki sposób w trakcie działania programu uzyskać tyle pamięci, ile potrzebujemy. Tak! Nareszcie będziemy mogli zmonopolizować pamięć w komputerze, chociaż raczej nie powinniśmy tego robić.

### Pozyskiwanie pamięci za pomocą instrukcji new

**Dynamiczna alokacja** oznacza możliwość żądania w czasie działania programu takiej ilości pamięci, jaka jest potrzebna. Zamiast pracować ze stałym zbiorem zmiennych o określonej wielkości, Twój program obliczy, ile pamięci potrzebuje. W tym rozdziale poznasz podstawy alokowania pamięci, a kolejne rozdziały wyjaśniają Ci, jak w pełni korzystać z przewagi, jaką daje dynamiczna alokacja.

Zobaczmy najpierw, jak uzyskać więcej pamięci. Aby zainicjalizować wskaźnik adresem z dostępnej pamięci, należy użyć słowa kluczowego `new`. Pamiętaj, że dostępną pamięć stanowi nieużywany obszar pamięci komputera, do którego Twój program może żądać dostępu. Oto podstawowa składnia instrukcji `new`:

```
int *w_int = new int;
```

Operator `new` korzysta z „przykładowej” zmiennej, na podstawie której oblicza potrzebną ilość pamięci. W tym przypadku weźmie pod uwagę typ `int` i zwróci tyle pamięci, ile potrzeba do przechowania liczby całkowitej.

Wskaźnik `w_int` zostanie ustawiony na to właśnie miejsce w pamięci i razem z kodem, który z niej korzysta, stanie się jej właścicielem. Kod, w którym użyto wskaźnika `w_int`, będzie w końcu musiał zwrócić swoją pamięć do wolnych zasobów. Operacja taka nazywana jest **zwalnianiem** pamięci. Dopóki pamięć używana przez `w_int` jest zajęta, będzie oznaczona jako znajdująca się w użyciu i nie zostanie ponownie przydzielona. Jeśli będziesz alokować pamięć, nigdy jej nie zwalniając, wyczerpiesz całą wolną pamięć w komputerze.

Aby zwrócić pamięć do wolnego zasobu, należy użyć słowa kluczowego `delete`. Operacja `delete` zwalnia pamięć zaalokowaną za pomocą instrukcji `new`. Oto zwolnienie pamięci zajmowanej przez `w_int`:

```
delete w_int;
```

Dobrą praktyką po usunięciu wskaźnika jest ponowne nadanie mu wartości NULL:

```
delete w_int;  
w_int = NULL;
```

Taki zabieg nie jest konieczny — po usunięciu wskaźnika nie będziesz mógł korzystać z pamięci, którą wskazywała, ponieważ zostanie ona zwrócona do wolnego zasobu (i może zostać przydzielona później). Nadając wskaźnikowi wartość NULL, zabezpieczasz się jednak przed sytuacją, w której kod próbuje dokonać dereferencji wskaźnika po jego zwolnieniu (co zdarza się dość często, nawet doświadczonym programistom), o czym szybko się dowiesz, ponieważ w programie wystąpi awaria. Jest to o wiele lepsze rozwiązywanie, niż gdybyś miał się dowiedzieć o takim błędzie później, gdy Twój program zniszczy czyjeś dane.

## Brak pamięci

Pamięć nie jest zasobem nieograniczonym — możesz dosłownie zagarnąć ją w całości. Jeśli tak się stanie, nie będziesz mógł uzyskać więcej pamięci. Jeśli w C++ wywołanie instrukcji new nie powiedzie się ze względu na brak wolnej pamięci w systemie, zostanie zgłoszony wyjątek. Zwykle nie będziesz musiał przejmować się taką sytuacją, ponieważ w nowoczesnych systemach występuje ona na tyle rzadko, że większość programistów decyduje się ją zignorować (sytuacja taka jest szczególnie mało prawdopodobna w przypadku dobrze napisanych programów, które prawidłowo zwalniają pamięć; kod, który nigdy nie zwalnia pamięci, ma większe szanse na jej wyczerpanie). Wyjątki, których obsłużenie stanowi dość zaawansowane zagadnienie, opisano pod koniec tej książki. Zwykle najlepsze rozwiązywanie polega po prostu na każdorazowym zwalnianiu alokowanej pamięci i nie przejmowaniu się tym, że wywołanie instrukcji new może się nie udać.

## Referencje i dynamiczna alokacja

Zazwyczaj nie powinieneś przechowywać zaalokowanej pamięci w referencji:

```
int &wart = *(new int);
```

Lepiej unikać takiej konstrukcji, ponieważ referencje nie gwarantują bezpośredniego dostępu do „surowych” adresów pamięci. Możesz je uzyskać za pomocą operatora &, ale w zasadzie rola referencji powinno być zapewnianie dodatkowych nazw zmiennych, a nie przechowywanie dynamicznie zaalokowanej pamięci.

## Wskaźniki i tablice

Być może zastanawiasz się, jak za pomocą instrukcji new można uzyskać więcej pamięci, niż się miało na początku programu, skoro wszystko, co można przy jej użyciu zrobić, to inicjalizacja pojedynczego wskaźnika. Odpowiedź na to pytanie jest taka, że wskaźnik potrafi wskazywać również sekwencję wartości. Innymi słowy, wskaźniki można traktować tak jak tablice. Tablica jest przecież serią wartości rozmieszczonych sekwencyjnie w pamięci komputera. Ponieważ wskaźnik przechowuje adres pamięci, można w nim zapisać adres pierwszego elementu tablicy. Aby uzyskać dostęp do jej innego elementu, wystarczy po prostu posłużyć się wartością, która odzwierciedla odległość tego elementu od początku tablicy.

Dlaczego takie rozwiązanie miałoby być przydatne? Ponieważ tablice można tworzyć dynamicznie w wolnej pamięci, co pozwala określić ilość potrzebnej pamięci w trakcie działania programu. Zaraz pokażę na przykładzie, jak to zrobić, ale najpierw podam kilka podstawowych informacji.

Tablicę można przypisać do wskaźnika bez konieczności posługiwania się operatorem adresu:

```
int liczby[ 8 ];
int* w_liczby = liczby;
```

Teraz można korzystać ze wskaźnika `w_liczby` tak jak z tablicy:

```
for ( int i = 0; i < 8; ++i )
{
    w_liczby[ i ] = i;
}
```

Tablica `liczby` po przypisaniu jej do wskaźnika działa w taki sposób, jakby sama była wskaźnikiem. **Ważne jest, aby pamiętać, że tablice nie są wskaźnikami, chociaż można je przypisywać do wskaźników.** Kompilator C++ rozumie, jak przełożyć tablicę na wskaźnik wskazujący jej pierwszy element (taka konwersja zachodzi w C++ bardzo często; kiedy na przykład przypisujesz zmienną typu `char` do zmiennej typu `int` — chociaż `char` nie jest tym samym co `int`, kompilator będzie wiedzieć, jak dokonać potrzebnej konwersji).

Za pomocą instrukcji `new` można dynamicznie zaalokować tablicę w pamięci i przypisać ją do wskaźnika:

```
int *w_liczby = new int[ 8 ];
```

Dzięki użyciu w argumencie instrukcji `new` składni tablicowej kompilator wie, ile pamięci potrzeba — tyle, ile wymaga 8-elementowa tablica liczb całkowitych. Od tej pory można korzystać ze wskaźnika `w_liczby` tak, jakby wskazywał tablicę. W przeciwieństwie jednak do tablic, powinieneś zwolnić pamięć, którą wskazuje `w_liczby`, co w ogóle nie jest konieczne w przypadku wskaźnika odnoszącego się do tablicy zadeklarowanej statycznie. Aby zwolnić pamięć, należy posłużyć się specjalną składnią operatora `delete`:

```
delete[] w_liczby;
```

Nawiasy prostokątne informują kompilator, że wskaźnik nie odwołuje się do pojedynczej wartości, ale do tablicy.

A teraz przykład, na który czekasz — dynamiczne określenie ilości potrzebnej pamięci:

```
int ile_liczb;
cin >> ile_liczb;
int *w_liczby = new int[ ile_liczb ];
```

Kod ten pyta użytkownika, ile liczb potrzebuje, po czym korzysta z uzyskanej odpowiedzi w celu określenia rozmiaru dynamicznie alokowanej tablicy. Tak naprawdę nie musimy znać z góry dokładnej liczby; możemy alokować pamięć w miarę wzrostu potrzeb użytkownika. Oznacza to, że czeka nas trochę kopowania, ale jest to jak najbardziej wykonalne. Spójrzmy na kod demonstrujący omawianą technikę. Poniższy program wczytuje liczby od użytkownika, a jeśli okaże się, że użytkownik wprowadził więcej liczb, niż można zmieścić w tablicy, zmienia jej wielkość.

### **Przykładowy kod 36.: zmiana rozmiaru tablicy.cpp**

```
#include <iostream>

using namespace std;

int *zwięksTablice (int* w_wartosci, int *rozmiar);
void drukujTablice (int* w_wartosci, int rozmiar, int zajete_pola);

int main ()
```

```

{
    int nastepny_element = 0;
    int rozmiar = 10;
    int *w_wartosci = new int[ rozmiar ];
    int wart;
    cout << "Podaj liczbę: ";
    cin >> wart;
    while ( wart > 0 )
    {
        if ( rozmiar == nastepny_element + 1 )
        {
            // Wszystko, co musimy teraz zrobić, to zaimplementować
            // funkcję zwiększaTablice. Zauważ, że rozmiar musimy
            // przekazać jako wskaźnik, ponieważ w każdej chwili
            // powinniśmy znać na bieżąco wielkość rosnącej tablicy!
            w_wartosci = zwiększaTablice( w_wartosci, &rozmiar );
        }
        w_wartosci[ nastepny_element ] = wart;
        nastepny_element++;
        cout << "Oto bieżące parametry tablicy: " << endl;
        drukujTablice( w_wartosci, rozmiar, nastepny_element );
        cout << "Podaj liczbę (0 - wyjście z programu): ";
        cin >> wart;
    }
    delete [] w_wartosci;
}

void drukujTablice (int *w_wartosci, int rozmiar, int zajete_pola)
{
    cout << "Całkowity rozmiar tablicy: " << rozmiar << endl;
    cout << "Liczba zajętych pól tablicy: " << zajete_pola << endl;
    cout << "Wartości w tablicy: " << endl;
    for ( int i = 0; i < zajete_pola; ++i )
    {
        cout << "w_wartosci[" << i << "] = " << w_wartosci[ i ] << endl;
    }
}

```

Pomyślmy przez chwilę, jak powiększyć tablicę. Co powinniśmy zrobić? Nie możemy tak po prostu poprosić o zwiększenie pamięci, którą dysponujemy — w przeciwieństwie do Excela nie da się tu wstawić nowej kolumny, gdy zabraknie nam miejsca. Musimy poprosić o więcej pamięci i przekopiować do niej istniejące wartości.

Pozostaje jeszcze pytanie, ile pamięci powinniśmy otrzymać. Jednorazowe powiększanie tablicy o jedną wartość całkowitą będzie nieefektywne. Takie rozwiązanie pociągnęłoby za sobą wiele niepotrzebnych alokacji pamięci — pamięć by się nam nie skończyła, ale sam proces pozyskiwania dodatkowego miejsca spowolniłby działanie programu. Dobrą strategią będzie odczytanie bieżącej wielkości tablicy i podwojenie jej. Jeżeli przestaniemy wprowadzać nowe wartości, tablica nie zabierze dużo miejsca — nie więcej niż dwa razy tyle, ile zajmowała na początku, a przy tym nie będziemy musieli bezustannie alokować pamięci. Jak widać, musimy znać bieżącą wielkość tablicy, a także początkowe jej wartości, aby je przekopiować.

### **Przykładowy kod 37.: zmiana rozmiaru tablicy.cpp (ciąg dalszy)**

```

int *zwiększaTablice (int* w_wartosci, int *rozmiar)
{
    *rozmiar *= 2;
    int *w_nowe_wartosci = new int[ *rozmiar ];

```

```

for ( int i = 0; i < *rozmiar; ++i )
{
    w_nowe_wartosci[ i ] = w_wartosci[ i ];
}
delete [] w_wartosci;
return w_nowe_wartosci;
}

```

Zwróć uwagę na to, w jaki sposób zadbane o usunięcie wartości ze wskaźnika `w_wartosci` po zakończeniu kopiowania danych z tablicy. W przeciwnym razie mielibyśmy do czynienia z wyciekiem pamięci, ponieważ nadpisujemy wskaźnik przechowujący naszą tablicę do chwili powrotu z funkcji `zwięksTablice`.

## Tablice wielowymiarowe

Zwiększanie wielkości pojedynczej dużej tablicy jest bardzo przydatne; to technika, którą z pewnością będziesz chciał zapamiętać. Czasami jednak przyjdzie Ci mieć do czynienia z czymś więcej niż tylko jedną sporą tablicą. Czy pamiętasz, jak eksytująca była praca z tablicami wielowymiarowymi? Czy nie byłoby miło mieć możliwość decydowania o wielkości takich tablic? Możemy się za to zabrać — będzie to całkiem dobre ćwiczenie, które pomoże Ci lepiej zrozumieć wskaźniki, poza tym omawiana technika okaże się bardzo przydatna. Aby ją zgłębić, potrzebna Ci będzie dodatkowa wiedza. W kilku następnych podrozdziałach omówię podstawy tej wiedzy, po czym pokażę Ci, jak można dynamicznie alokować wielowymiarowe struktury danych.

## Arytmetyka wskaźników

*Podrozdział ten nieco głębiej traktuje tematykę wskaźników i może wymagać od Ciebie większego wysiłku umysłowego! Przedstawione w nim idee, chociaż trudne, mają sens. Jeżeli nie zrozumiesz zagadnień tu przedstawionych podczas pierwszego czytania, przeczytaj go ponownie. Jeśli możesz pojąć wszystko, co znajduje się w tym podrozdziale, łącznie z tematyką alokacji tablic dwuwymiarowych, otworzy się przed Tobą możliwość zrozumienia wszelkich zagadnień dotyczących wskaźników. Nie będzie zatem łatwo i — w przeciwieństwie do niektórych rozdziałów — odniesione korzyści nie będą od razu oczywiste, ale czas poświęcony na zrozumienie przedstawionych tu zagadnień oznacza mniej czasu przeznaczonego na zapoznanie się z resztą treści tej książki. Możesz mi zaufać.*

Porozmawiajmy przez chwilę o adresach pamięci i o tym, jak o nich myśleć. Wskaźniki reprezentują adresy pamięci, które z kolei są tylko liczbami, w związku z czym — tak samo jak w przypadku innych liczb — można wykonywać na nich operacje matematyczne, jak na przykład dodanie liczby do wskaźnika lub odjęcie jednego wskaźnika od drugiego. W jakim celu miałbyś to robić? Po pierwsze, zdarza się, że gdy chcesz zapisać blok pamięci, wiesz dokładnie, z jakim przesunięciem względem pewnego miejsca chcesz ten blok zapisać. Jeżeli opis ten brzmi dla Ciebie zbyt żargonowo, powinieneś wiedzieć, że już niejednokrotnie miałeś do czynienia z taką sytuacją, mianowicie przy okazji pracy z tablicami!

Okazuje się, że gdy napiszesz:

```

int x[ 10 ];
x[ 3 ] = 120;

```

stosujesz **arytmetykę wskaźników** w celu wypełnienia trzeciego elementu tablicy wartością 120. Nawiąsy prostokątne pełnią tu tylko rolę **lukru składniowego** (co jest terminem oznaczającym specjalną, uproszczoną składnię), zastępującego arytmetykę wskaźników. Tę samą operację można przeprowadzić za pomocą następującego zapisu:

$$*(x + 3) = 120;$$

Rozłożmy na składniki to, co się tutaj dzieje. Co dziwne (lub mylące), **nie** zachodzi tu dodawanie 3 do bieżącej wartości zmiennej  $x$ , tylko dodanie do  $x$  wyniku mnożenia  $3 * \text{sizeof}(\text{int})$ . `sizeof` to słowo kluczowe podające w bajtach rozmiar zmiennej danego typu. Podczas pracy z pamięcią komputera taka informacja często będzie Ci potrzebna. W arytmetyce wskaźników zawsze dodawane są „przegródki” pamięci, a same wskaźniki nie są traktowane jak liczby (podobnie jak użycie nawiasów prostokątnych udostępnia określony element tablicy). Dodawanie wielkości będących wielokrotnością rozmiaru zmiennej zapobiega w arytmetyce wskaźników przypadkowemu zapisaniu lub odczytaniu wartości znajdującej się między dwoma zmiennymi (na przykład pobraniu ostatnich dwóch bajtów jednej zmiennej i pierwszych dwóch bajtów drugiej zmiennej)<sup>1</sup>.

W większości przypadków powinieneś jednak korzystać ze składni tablicowej, zamiast wkładać się w próby poprawnego stosowania arytmetyki wskaźników. Podczas przeprowadzania operacji arytmetycznych na wskaźnikach bardzo trudno połąpać się w obliczeniach i łatwo można zapomnieć, że nie dodajesz poszczególnych bajtów, tylko „przegródki” pamięci. Zrozumienie arytmetyki wskaźników ułatwi Ci jednak realizowanie pewnych wyszukanych zadań, a znajomość jej pełnych możliwości będzie Ci potrzebna w kolejnych rozdziałach tej książki. Dzięki niej zrozumiesz ponadto, jak dynamicznie alokować tablice wielowymiarowe.

## Zrozumieć tablice dwuwymiarowe

Zanim przystąpimy do dynamicznego alokowania tablic wielowymiarowych, powinieneś zrozumieć, czym tak naprawdę jest taka tablica. Także i w tym podrozdziale powinieneś zdobyć się na wąsłyku związanym ze zrozumieniem opisanych tu zagadnień — wbrew ich trudności. To Ci się *opłaci*.

Zacznijmy od pewnej ciekawostki: kiedy deklarujesz, że funkcja może przyjmować jako argument tablicę dwuwymiarową, nie musisz podawać obu jej wymiarów, a tylko drugiego.

Możesz podać oba wymiary:

```
int sumTablicaDwyWym( int array[ 4 ][ 4 ] );
```

Albo tylko drugiego:

```
int sumTablicaDwyWym( int array[] [ 4 ] );
```

Nigdy nie możesz jednak pominąć obu wymiarów:

```
int sumTablicaDwyWym( int array[] [] );
```

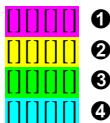
---

<sup>1</sup> A tak przy okazji, to w celu obliczenia odległości dzielącej dwa wskaźniki można je od siebie odejmować. W tym przypadku otrzymany dystans również nie będzie wyrażony liczbą bajtów, tylko liczbą „przegródek”. Z tego powodu nie ma możliwości odjęcia od siebie wskaźników dwóch różnych typów, ponieważ ich „przegródki” mogą mieć różne rozmiary. Z odejmowaniem wskaźników mam jednak do czynienia niezwykle rzadko. Nie ma natomiast możliwości dodawania do siebie wskaźników — do wskaźników można dodać jedynie przesunięcia. Czy to nie ciekawe, że operacje związane z odejmowaniem i dodawaniem wskaźników dają w wyniku różne typy wartości?

Ani podać tylko pierwszego wymiaru:

```
int sumTablicaDwyWym( int array[ 4 ][ ] );
```

Dzieje się tak dlatego, że w celu poprawnego przeprowadzania arytmetyki wskaźników potrzebne są tylko niektóre wymiary. Tablice dwuwymiarowe tak naprawdę ułożone są obok siebie w pamięci; kompilator umożliwia programiście, traktować je jak prostokątne bloki pamięci, ale w rzeczywistości są one tylko liniowym zbiorem adresów pamięci. Kompilator przekształca zapis tablicowy, taki jak `tablica[ 3 ][ 2 ]`, na pozycję w pamięci. Oto, jak w prosty sposób można o tym myśleć. Tablicę 4x4 można wyobrazić sobie tak:



Tablica ta jest w rzeczywistości umieszczona w pamięci następująco:



Aby skorzystać z elementu `[ 3 ][ 2 ]` (który znajduje się w części ④), kompilator musi uzyskać dostęp do pamięci położonej trzy wiersze w dół (przez części ①, ②, i ③) i dwie kolumny w prawo. W celu przejścia w dół o trzy wiersze, z których każdy ma szerokość czterech liczb całkowitych, należy ominąć  $4 * 3$  całkowite „przegródki”, po czym udać się dalej jeszcze o 2 „przegródki” (aby dotrzeć do trzeciego elementu w ostatnim rzędzie).

Innymi słowy, `tablica[ 3 ][ 2 ]` przekształca się w następującą arytmetykę wskaźników:

```
*(tablica + 3 * <szerokość tablicy> + 2)
```

Teraz już wiesz, dlaczego potrzebna jest szerokość tablicy — bez niej nie da się przeprowadzić potrzebnych obliczeń, a drugi wymiar tablicy dwuwymiarowej jest jej szerokością. Nie można zrobić tego samego z wysokością tablicy ze względu na fizyczne rozmieszczenie danych w pamięci (wysokość byłaby wymagana, gdyby tablica była ułożona w drugim kierunku, czyli według wierszy). Z tego względu można jako argument funkcji przekazywać tablice o różnych wysokościach, ale drugi wymiar musi być ścisłe określony. Prawdę mówiąc, dla dowolnej tablicy wielowymiarowej można określać wszystkie jej wymiary z wyjątkiem wysokości. Tablicę jednowymiarową możesz traktować jak specjalny przypadek tablicy, która ma tylko wysokość.

Niestety, ponieważ podczas deklarowania tablicy dwuwymiarowej wymagane jest wpisanie bezpośrednio w kod jej szerokości, dynamiczna alokacja dwuwymiarowej tablicy o dowolnej szerokości wymaga jeszcze jednej funkcjonalności języka C++, którą są wskaźniki do wskaźników.

## Wskaźniki do wskaźników

Oprócz wskazywania zwykłych danych wskaźniki mogą wskazywać także inne wskaźniki. W końcu wskaźnik, tak samo jak każda inna zmienna, ma swój adres, do którego można używać dostęp.

Aby zadeklarować wskaźnik do wskaźnika, należy napisać:

```
int **w_w_x;
```

Wskaźnik `w_w_x` wskazuje adres pamięci zawierający wskaźnik do liczby całkowitej. Prefiks `w_w` użytem w celu pokazania, że wskaźnik wskazuje inny wskaźnik, co znaczy, że należy przekazać mu adres pamięci jakiegoś wskaźnika, np.:

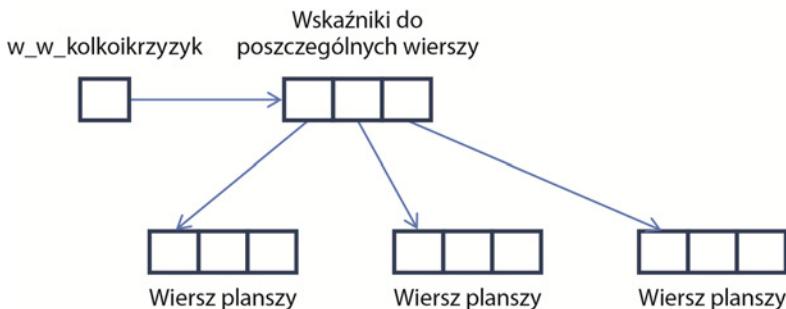
```
int *w_y;
int **w_w_x;
w_w_x = & w_y;
```

Teraz można za pomocą `w_w_x` przypisać wskaźnik do `w_y`:

```
*w_w_x = new int;
```

Wskaźników do wskaźników można używać w celu tworzenia tablic dwuwymiarowych tak samo, jak korzystaliśmy z pojedynczych wskaźników w celu tworzenia tablic jednowymiarowych o dowolnych rozmiarach.

Można o tym myśleć jak o jednowymiarowej tablicy wskaźników, w której każdy wskaźnik odnosi się do drugiej jednowymiarowej tablicy. Spójrzmy, jak wyglądałaby na rysunku deklaracja wskaźnika do wskaźnika przechowującego planszę do gry w kółko i krzyżyk:



Pierwszy wskaźnik wskazuje zbiór wskaźników, z których każdy wskazuje jeden wiersz planszy. Oto kod, który jest potrzebny do alokacji takiej struktury danych:

```
int **w_w_kolkoikrzyzyk;
// Zauważ, że nie stosujemy zapisu int*, ponieważ alokujemy tablice wskaźników
w_w_kolkoikrzyzyk = new int*[ 3 ];

// Każdy wskaźnik będzie przechowywał adres tablicy wartości całkowitych
for ( int i = 0; i < 3; i++ )
{
    w_w_kolkoikrzyzyk[ i ] = new int[ 3 ];
```

Od tego miejsca można korzystać z zaalokowanej pamięci tak samo jak z tablicy dwuwymiarowej. Możemy na przykład zainicjalizować całą planszę za pomocą dwóch pętli:

```
for ( int i = 0; i < 3; i++ )
{
    for ( int j = 0; j < 3; j++ )
    {
        w_w_kolkoikrzyzyk[ i ][ j ] = 0;
    }
}
```

Aby zwolnić pamięć, musimy postąpić w odwrotnej kolejności w stosunku do tego, co robiliśmy, aby ją zainicjalizować: najpierw zwalniamy każdy wiersz, a następnie wskaźnik przechowujący wiersze:

```

for ( int i = 0; i < 3; i++ )
{
    delete []w_w_kolkoikrzyzyk[ i ];
}

delete []w_w_kolkoikrzyzyk;

```

Zwykle nie będziesz stosował powyższego rozwiązania, jeśli będziesz znał potrzebną wielkość pamięci (jak to jest na przykład w grze w kółko i krzyżyk), ponieważ jest ono nieco bardziej skomplikowane niż napisanie po prostu:

```
int plansza_kolko_krzyzyk[ 3 ][ 3 ];
```

Jeżeli jednak chciałbyś utworzyć grę o dowolnie dużej planszy, będzie Ci potrzebne takie właśnie rozwiązanie.

## Wskaźniki do wskaźników i tablic dwuwymiarowych

Zwróć uwagę, że położenie w pamięci wskaźnika do wskaźnika, gdy jest on używany w celu przechowywania tablicy dwuwymiarowej, jest inne niż położenie takiej tablicy. Standardowa tablica dwuwymiarowa zajmuje ciągły obszar pamięci, natomiast w przypadku wskaźnika wcale tak nie jest! Na wcześniejszym rysunku (z planszą do gry w kółko i krzyżyk) widać, że każdy wiersz tworzy odrębną grupę danych; w rzeczywistości każdy wiersz zapisany w pamięci komputera może znajdować się dość daleko od pozostałych wierszy.

Takie rozwiązanie pociąga za sobą konsekwencje dla każdej funkcji, która przyjmuje tablicę jako argument. Jak już wiesz, do wskaźnika można przypisać tablicę:

```
int x[ 8 ];
int *y = x;
```

Nie można jednak przypisać dwuwymiarowej tablicy do wskaźnika do wskaźnika:

**ZŁY KOD**

```
int x[ 8 ][ 8 ];
int **y = x; // Nie skompiluje się!
```

W pierwszym przypadku tablicę można traktować jako wskaźnik do bloku pamięci, który zawiera wszystkie dane. W drugim jednak przypadku tablica nadal jest tylko pojedynczym wskaźnikiem do bloku pamięci.

Najważniejszą konsekwencją różnic w układzie danych jest brak możliwości przekazania wskaźnika do wskaźnika funkcji, która oczekuje tablicy wielowymiarowej (nawet jeśli możesz przekazać wskaźnik funkcji, która przyjmuje tablicę jednowymiarową).

```

int sumuj_macierz (int wartosci[][ 4 ], int liczba_wierszy)
{
    int suma_narast = 0;
    for ( int i = 0; i < liczba_wierszy; i++ )
    {
        for ( int j = 0; j < 4; j++ )
        {
            suma_narast += wartosci[ i ][ j ];
        }
    }
    return suma_narast;
}

```

Jeśli natomiast zaalokujesz wskaźnik do wskaźnika i przekażesz go do funkcji, kompilator zgłosi błąd:

**ZŁY KOD**  
int \*\*x;  
// Zaalokuj x, aby miał 10 wierszy  
sumuj\_macierz( x, 10 ); // Nie skompiluje się

W przypadku jednowymiarowym obie operacje wykonują po prostu pewne przesunięcie względem adresu wskaźnika. W przypadku dwuwymiarowym podejście typu wskaźnik do wskaźnika wymaga jednak przeprowadzenia dwóch dereferencji wskaźnika — pierwszej w celu odnalezienia właściwego wiersza i drugiej, aby przejść do potrzebnej wartości w wierszu. W odniesieniu do tablic w celu otrzymania właściwej wartości stosowana jest arytmetyka wskaźników. Ponieważ w przypadku wskaźnika do wskaźnika taka arytmetyka nie jest możliwa, kompilator nie pozwoli na przekazanie do funkcji wskaźnika do wskaźnika, tak jakby to była tablica dwuwymiarowa, nawet jeśli napiszesz kod, który poza tym wygląda identycznie!

## Oswajanie wskaźników

Wskaźniki początkowo mogą wydawać się trudnym zagadniением, jednak można je zrozumieć. Jeśli nie przyswoiłeś sobie wszystkich wiadomości na ich temat, weź kilka głębszych oddechów, ponownie przeczytaj ten rozdział, odpowiedz na pytania testowe i spróbuj rozwiązać zadania praktyczne. Nie musisz mieć poczucia, że w pełni ogarniasz wszelkie niuanse związane ze stosowaniem wskaźników, chociaż powinieneś znać składnię umożliwiającą ich stosowanie, wiedzieć, jak je inicjalizować, oraz rozumieć, w jaki sposób alokować pamięć.

## Sprawdź się

1. Które z poniższych słów kluczowych jest właściwym poleceniem służącym do alokowania pamięci w C++?  
  - A. new
  - B. malloc
  - C. create
  - D. value
2. Które z poniższych słów kluczowych jest właściwym poleceniem służącym do zwalniania zaallokowanej pamięci w C++?<sup>2</sup>  
  - A. free
  - B. delete
  - C. clear
  - D. remove
3. Które z poniższych stwierdzeń jest prawdziwe?  
  - A. Tablice i wskaźniki są tym samym.
  - B. Tablic nie można przypisywać do wskaźników.

---

<sup>2</sup> No dobrze — masz także rację, jeśli na dwa pierwsze pytania odpowiedziałeś malloc i free; funkcje te pochodzą z języka C, ale to oznacza, że nie przeczytałeś tego rozdziału!

- C. Wskaźniki można traktować jak tablice, chociaż nimi nie są.  
 D. Ze wskaźników można korzystać tak samo jak z tablic, ale nie można ich alokować tak jak tablic.
- 4.** Jakie będą ostateczne wartości `x`, `w_int` oraz `w_w_int` po wykonaniu poniższego kodu?  
 Zwróć uwagę, że kompilator nie przyjmie bezpośrednio tego kodu, ponieważ liczby całkowite i wskaźniki są różnych typów. Aby jednak stwierdzić, co dzieje się z takimi wielokrotnymi wskaźnikami, ćwiczenie to można wykonać na kartce papieru.
- ```
int x = 0;
int *w_int = & x;
int **w_w_int = & w_int;
*w_int = 12;
**w_w_int = 25;
w_int = 12;
*w_w_int = 3;
w_w_int = 27;
```
- A.  $x = 0, w_w_int = 27, w_int = 12$   
 B.  $x = 25, w_w_int = 27, w_int = 12$   
 C.  $x = 25, w_w_int = 27, w_int = 3$   
 D.  $x = 3, w_w_int = 27, w_int = 12$

- 5.** W jaki sposób można zaznaczyć, że wskaźnik nie wskazuje ważnej wartości?
- A. Nadać mu wartość ujemną.  
 B. Nadać mu wartość NULL.  
 C. Zwolnić pamięć skojarzoną z tym wskaźnikiem.  
 D. Nadać mu wartość false.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- Napisz funkcję tworzącą dwuwymiarową tabliczkę mnożenia (<http://www.gry-dla-dzieci.eu/tabliczka-mnozenia/>) o dowolnej wielkości w każdym z dwóch wymiarów.
- Napisz funkcję, która pobiera trzy argumenty — długość, szerokość i wysokość, dynamicznie alokuje trójwymiarową tablicę utworzoną na podstawie tych wartości, po czym wypełnia ją tabliczkami mnożenia. Nie zapomnij o zwolnieniu pamięci po zakończeniu działania programu.
- Napisz program, który pokazuje adresy pamięci wszystkich elementów tablicy dwuwymiarowej. Sprawdź, czy wyświetcone adresy są zgodne z wyjaśnieniami, których wcześniej udzieliłem.
- Napisz program, który pozwala użytkownikom zapisywać, kiedy ostatnio rozmawiali ze swoimi znajomymi. Użytkownicy powinni mieć możliwość dodawania nowych znajomych (tylko, ile tylko chcą!) oraz przechowywania liczby dni, które uplynęły od czasu ostatniej rozmowy z danym znajomym. Pozwól użytkownikom aktualizować tę wartość (ale nie zezwalaj na wprowadzanie nieprawidłowych danych, na przykład liczb ujemnych). Program powinien pozwalać na wyświetlanie listy znajomych posortowanej według ich nazwisk lub liczby dni, które uplynęły od rozmowy z każdym ze znajomych.

5. Napisz grę w czwórki<sup>3</sup> dla dwóch osób, w której użytkownik może określić szerokość i wysokość planszy, a każdy z graczy na przemian wrzuca żeton do wybranej przez siebie kolumny. Wyświetl planszę, w której jedna krawędź będzie reprezentowana przez znaki +, druga krawędź przez x, natomiast \_ będzie oznaczać wolne miejsca.
6. Napisz program, który pobiera od użytkownika szerokość i wysokość, po czym dynamicznie generuje labirynt o podanych wymiarach. W labiryncie zawsze musi znaleźć się możliwa do przejścia ścieżka (jak można to zagwarantować?). Po wygenerowaniu labiryntu wyświetl go na ekranie.

Każde z powyższych zadań spróbuj rozwiązać w dwóch wersjach — przy użyciu wskaźników i za pomocą referencji. Nie zapomnij o zwalnianiu zaallokowanej pamięci.

---

<sup>3</sup> <http://pl.wikipedia.org/wiki/Czwórki>

## Wprowadzenie do struktur danych: listy powiązane

---

W poprzednim rozdziale dowiedziałeś się, jak dynamicznie alokować tablice w celu wykorzystania przewagi, jaką daje możliwość alokowania pamięci. W tym rozdziale poznasz jeszcze bardziej elastyczne zastosowanie dynamicznego alokowania pamięci. Dobrą stroną posiadania dużej ilości pamięci jest możliwość dysponowania ogromną przestrzenią na przechowywanie danych. Można tam zapisać całkiem sporo informacji. Wyfania się jednak pytanie o to, w jaki sposób informacje te szybko zapisywać i jak uzyskać do nich łatwy dostęp w przyszłości. Właśnie to zagadnienie omówię w niniejszym rozdziale.

Najpierw wyjaśnię kilka terminów. **Struktura danych** to sposób organizowania informacji w pamięci komputera. Tablica jest na przykład bardzo prostą strukturą, w której pamięć jest zorganizowana liniowo. Każdy element tablicy stanowi element struktury danych. Tablica dwuwymiarowa zaimplementowana przy pomocy wskaźników do wskaźników jest nieco bardziej wyszukaną strukturą danych.

Problem w przypadku tablic polega jednak na tym, że nie ma możliwości wstawienia elementu do tablicy, w której nie ma wolnych miejsc. Jak już wiesz, musimy wówczas zacząć od początku — zaalokować nową tablicę, po czym przekopiować do niej wszystkie elementy ze starej tablicy. Takie rozwiązanie programiści nazywają **kosztowną** operacją. Biorąc pod uwagę standardy procesora w Twoim komputerze, jest ona bardzo czasochłonna. Kosztowne operacje mogą nie stanowić problemu z punktu widzenia użytkownika. Jeśli nie będziesz prowadzić ich zbyt często, może okazać się, że tak naprawdę nikt ich nie zauważy. Procesory w komputerach są bardzo szybkie. W większości przypadków kosztowne operacje mogą być jednak problematyczne.

Drugi problem z tablicami polega na tym, że nie ma możliwości łatwego wstawiania danych między istniejące elementy tablicy. Jeśli chcesz na przykład umieścić nowy element między pozycją pierwszą a drugą, a tablica liczy 1000 elementów, musiałbyś przestawić wszystkie elementy, począwszy od drugiego aż do tysięcznego! Takie rozwiązanie także jest zbyt kosztowne.

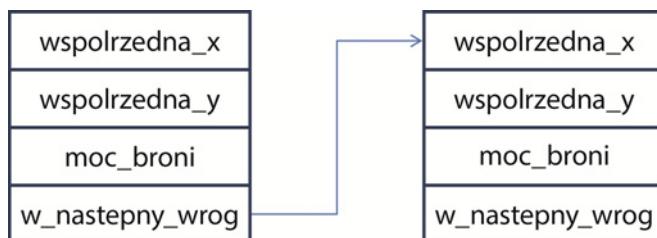
Czy znasz sytuacje, w których Twój komputer aż sapie z wysiłku, a czekanie na jego reakcję staje się coraz bardziej uciążliwe? To dlatego, że wykonuje on jakąś kosztowną operację. Struktury danych pomagają znaleźć efektywne sposoby przechowywania informacji, dzięki czemu użytkownicy nie będą musieli oglądać na ekranie ikony z wirującym kółkiem.

Kolejnym powodem stosowania różnych struktur danych jest możliwość przejścia na wyższy poziom w myśleniu o programowaniu. Zamiast myśleć o potrzebie użycia „pętli”, zacznesz myśleć o skorzystaniu z „listy”. Struktury danych udostępniają logiczne sposoby na organizowanie

informacji, a także zwarte metody komunikowania podstawowych operacji, które Twój program będzie przeprowadzać. Jeśli na przykład stwierdzisz, że potrzebna jest Ci „lista”, jasno dasz do zrozumienia, że musisz przechować jakieś dane w sposób umożliwiający ich efektywne dodawanie i usuwanie. W miarę poznawania struktur danych coraz częściej będziesz myśleć o swoich programach w kategoriach danych, które będą w nich potrzebne, oraz o sposobach ich organizowania. Na razie wystarczy jednak tego teoretyzowania; porozmawiajmy przez chwilę o listach powiązanych.

Czy pamiętasz problem związany z łatwym dodawaniem nowych danych? Przypominasz sobie, jak przy użyciu tablicy trzeba było wszystko kopiować? Mam nadzieję, że pamiętasz — przecież to było zaledwie kilka akapitów temu! Czyż nie byłoby wspaniale, gdybyś mógł utworzyć strukturę danych, w której każdy element pokazuje, gdzie znajduje się następny element? Mógłbyś wówczas z łatwością dodać nowy element na końcu takiej struktury, zapisując w ostatnim elemencie odnośnik do jej nowego elementu. Mógłbyś także wstawać nowe dane między dwa istniejące elementy, zmieniając po prostu miejsce wskazywane przez jeden z tych elementów. Powróćmy do przykładu, z którego już korzystałem — przechowywania wrogów w grze komputerowej. Z pewnością chciałbyś dysponować jakąś listą wrogów, w której każdy element byłby strukturą przechowującą informacje o danym wrogu. (Po co w ogóle potrzebna jest lista wrogów? Musiałbyś mieć taką listę, gdyby podczas każdej rundy gry wrogowie wykonywali jakieś czynności, na przykład gdybyś musiał sprawdzać na podstawie listy każdego wroga i decydować o jego ruchu. Nie jest to taka „lista” jak lista zakupów albo lista uczniów w klasie. Czasami potrzebujesz list, aby zapisywać w nich informacje o każdym obiekcie, którym dysponujesz, w tym przypadku są to wrogowie). Chciałbyś mieć także możliwość szybkiego dodawania i usuwania wrogów. A gdyby jeszcze tak każdy wróg miał informację o następnym wrogu?

Rzućmy okiem na graficzną reprezentację elementu, który zawiera współrzędne x i y oraz broń o określonej mocy.



Mamy tu strukturę `WrogiStatekKosmiczny`, która zawiera jakiś rodzaj łącza do następnej struktury. Czym mogłoby być takie łącze? Wskaźnikiem! Każdy statek kosmiczny zawiera wskaźnik do kolejnego statku kosmicznego.

```
struct WrogiStatekKosmiczny
{
    int wspolrzedna_x;
    int wspolrzedna_y;
    int moc_broni;
    WrogiStatekKosmiczny* w_nastepny_wrog;
};
```

Poczekaj chwilę! Właśnie użyliśmy struktury wewnętrz struktury `WrogiStatekKosmiczny`. Czy tak można? Można! Język C++ znakomicie radzi sobie z obsługą takiej autoreferencji. Problem pojawiłby się, gdybyś napisał:

```
WrogiStatekKosmiczny nastepny_wrog;
```

W takim przypadku mielibyśmy strukturę, która powtarza sama siebie w nieskończoność. Zadeklarowanie pojedynczego statku wymagałoby całej pamięci dostępnej w systemie. Zwróć jednak uwagę na znak wskaźnika, znajdujący się przy elemencie `WrogiStatekKosmiczny`; nie jest to rzeczywista zmienna tego typu. Ponieważ wskaźniki nie muszą odnosić się do prawidłowych miejsc w pamięci, nie zostanie utworzona nieskończona liczba statków kosmicznych, a tylko jeden statek, który *może* (ale nie musi) wskazywać inny statek. Jeśli istotnie zostanie wskazany inny statek, będzie potrzebna jakąś pamięć, ale dopóki tak się nie stanie, jedyną zajętą pamięcią będzie pamięć, w której zapisano wskaźnik — zaledwie kilka bajtów. Użycie wskaźnika oznacza, że może on wskazywać jakieś rzeczywiste miejsce w pamięci; wymaga on tylko kilku bajtów na przechowanie adresu. Kiedy deklarujesz instancję struktury `WrogiStatek` →`Kosmiczny`, potrzebna jest jedynie ilość pamięci wystarczająca na zapisanie pól `wspolrzedna_x`, `wspolrzedna_y`, `moc_broni` oraz wskaźnika. Nie trzeba przechowywać kolejnego statku kosmicznego (co samo w sobie wymagałoby miejsca na kolejny statek), a tylko wskaźnik.

Pozwól mi na jeszcze jedną, ostatnią analogię. Wyobraź sobie pociąg. Każdy wagon ma sprzęg umożliwiający połączenie go z innym wagonem. Aby dołączyć nowy wagon, podczepiasz go po prostu do wagonu znajdującego się przed nim i za nim. Sprzęg może pozostać nieużywany, jeśli nie ma wagonu, który można do niego przyłączyć — taki mógłby być odpowiednik wskaźnika o wartości `NULL`.

Teraz, kiedy znasz już ideę tworzenia takiego rodzaju list, omówię szczegóły składni umożliwiającej wykorzystanie wskaźników w listach.

## Wskaźniki i struktury

Aby za pomocą wskaźnika uzyskać dostęp do pól struktury, należy zamiast operatora `.` użyć operatora `->`:

```
w_moja_struktura->moje_pole;
```

Każde pole struktury ma inny adres pamięci, zwykle oddalony o kilka bajtów od początku struktury. Składnia ze strzałką oblicza dokładne przesunięcie potrzebne do uzyskania dostępu do określonego pola struktury. Wszystkie pozostałe cechy wskaźników (wskaźnik wskazuje miejsce w pamięci, nie należy używać nieważnych wskaźników itd.) pozostają w mocy. Składnia ze strzałką stanowi odpowiednik zapisu:

```
(*w_moja_struktura).moje_pole;
```

Jest ona jednak prostsza do odczytania (i napisania!), gdy już się do niej przyzwyczaisz.

Jeśli funkcja pobiera jako argument wskaźnik do struktury, wówczas może ona modyfikować pamięć pod adresem skojarzonym z tą strukturą, umożliwiając tym samym wprowadzanie zmian w otrzymanej strukturze. Proces ten przebiega dokładnie tak samo, jak w przypadku przekazywania do funkcji tablic. Zobaczmy, jak to zadziała z naszą strukturą `WrogiStatek` →`Kosmiczny`:

### Przykładowy kod 38.: aktualizacja.cpp

```
// Ten plik nagłówkowy jest potrzebny do korzystania z wartości NULL;
// zwykle jest on dołączany w innych plikach nagłówkowych,
// tylko że teraz nie potrzebujemy innych plików
#include <cstddef>
```

```
struct WrogiStatekKosmiczny
```

```
{  
    int wspolrzedna_x;  
    int wspolrzedna_y;  
    int moc_broni;  
    WrogiStatekKosmiczny* w_nastepny_wrog;  
};  
  
WrogiStatekKosmiczny* generujNowegoWroga ()  
{  
    WrogiStatekKosmiczny* w_statek = new WrogiStatekKosmiczny;  
    w_statek->wspolrzedna_x = 0;  
    w_statek->wspolrzedna_y = 0;  
    w_statek->moc_broni = 20;  
    w_statek->w_nastepny_wrog = NULL;  
    return w_statek;  
}  
  
void ulepszBron (WrogiStatekKosmiczny* w_statek)  
{  
    w_statek->moc_broni += 10;  
}  
  
int main ()  
{  
    WrogiStatekKosmiczny* w_wrog = generujNowegoWroga();  
    ulepszBron( w_wrog );  
}
```

Funkcja generujNowegoWroga korzysta z instrukcji new w celu zaalokowania pamięci dla nowego statku. W funkcji ulepszBron możemy bezpośrednio modyfikować wskaźnik w\_statek, ponieważ wskazuje on blok pamięci zawierający wszystkie pola struktury.

## Tworzenie listy powiązanej

Teraz, kiedy znamy już składnię wskaźników do struktur, możemy tworzyć nasze listy. Lista utworzona przy wykorzystaniu struktury zawierającej wskaźnik do następnego elementu nazywana jest **listą powiązaną**. Aby zidentyfikować daną listę, potrzebny będzie jakiś sposób na zapisanie początku listy. Powróćmy do naszego przykładu i dodajmy zmienną służącą do przechowywania statków kosmicznych:

```
struct WrogiStatekKosmiczny  
{  
    int wspolrzedna_x;  
    int wspolrzedna_y;  
    int moc_broni;  
    WrogiStatekKosmiczny* w_nastepny_wrog;  
};  
  
WrogiStatekKosmiczny* w_wrogowie = NULL;
```

Zmienna w\_wrogowie będzie używana w celu przechowywania listy wrogów. Za każdym razem, gdy w grze utworzymy nowego wroga, dodamy go do listy (na liście tej będziemy prowadzić różne operacje na wrogach). Zmienna ta mogłaby nosić nazwę w\_pierwszy albo w\_początek, informując, że zawiera ona pierwszy element naszej listy.

Kiedy w grze dodamy nowego wroga, wprowadzimy go na początek listy.

```

WrogiStatekKosmiczny* generujNowegoWroga ()
{
    WrogiStatekKosmiczny* w_statek = new WrogiStatekKosmiczny;
    w_statek->wspolrzedna_x = 0;
    w_statek->wspolrzedna_y = 0;
    w_statek->moc_broni = 20;
    w_statek->w_nastepny_wrog = w_wrogowie;
    w_wrogowie = w_statek;
    return w_statek;
}

```

Zaczynamy z pustą (NULL) zmienną `w_wrogowie`. Kiedy dodajemy nowego wroga, aktualizujemy go, aby wskazywał poprzedni pierwszy element listy (przechowywanej w zmiennej `w_wrogowie`), po czym aktualizujemy zmienną `w_wrogowie`, aby wskazywała nowego wroga. Każdy nowy element dodajemy na początku listy, przesuwając pozostałe jej elementy w dół. Przesunięcie to nie wymaga żadnego kopiowania; modyfikujemy jedynie dwa wskaźniki.

Proces ten może wydawać się nieco zawikłany, tak więc przyjrzymy mu się krok po kroku i wspomóżmy się dodatkowymi rysunkami.

## Pierwszy przebieg

Początkowo zmienna `w_wrogowie` ma wartość NULL; innymi słowy nie ma wrogów (zawsze będziemy używać wartości NULL, aby wskazać, że znajdujemy się na końcu listy).

### Stan początkowy



### Krok 1. Utworzenie statku



### Kroki 2. i 3. Aktualizacja wskaźników w\_nastepny\_wrog i w\_wrogowie



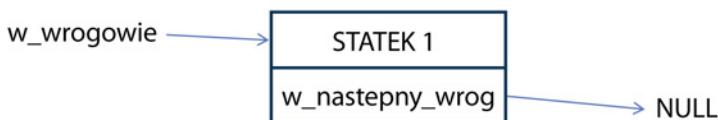
1. Zostaje zaalokowany nowy statek `w_statek`. Mamy teraz wroga — nazwę go STATEK 1, który nie znajduje się jeszcze na liście łączy. Na rysunku możesz zobaczyć, że pole `w_nastepny_wrog` nie ma nadanej wartości i wskazuje jakieś nieokreślone miejsce w pamięci komputera.
2. Pole `w_nastepny_wrog` STATKU 1 zostaje zaktualizowane w taki sposób, aby wskazywało aktualną listę wrogów (w tym przypadku jest to NULL).
3. Zmienna `w_wrogowie` zostaje zaktualizowana i wskazuje nowy statek.
4. Nasza funkcja zwraca teraz wskaźnik `w_statek` obiektowi, który ją wywołał i który może z niego skorzystać w dowolny sposób, podczas gdy zmienna `w_wrogowie` zapewnia dostęp do całej listy (która na razie zawiera tylko jeden element).

## Drugi przebieg

Podczas drugiego przebiegu wskaźnik `w_wrogowie` wskazuje statek, który właśnie utworzyliśmy.

1. Zostaje zaalokowany nowy statek — `w_statek`. Teraz mamy drugiego wroga, którego pole `w_nastepny_wrog` wskazuje nieokreślone miejsce w pamięci komputera.
2. Następnie pole `w_nastepny_wrog` jest aktualizowane, aby wskazywało bieżącą listę wrogów, w tym przypadku wroga, którego utworzyliśmy w pierwszym przebiegu.
3. Zmienna `w_wrogowie` zostaje zaktualizowana i wskazuje nowo utworzony statek (wskazuje ona teraz drugi statek, który z kolei wskazuje statek pierwszy).
4. Nasza funkcja zwraca teraz wskaźnik `w_statek` obiektowi, który ją wywołał i który może z niego skorzystać w dowolny sposób, podczas gdy zmienna `w_wrogowie` zapewnia dostęp do całej listy, w tym przypadku do dwóch elementów.

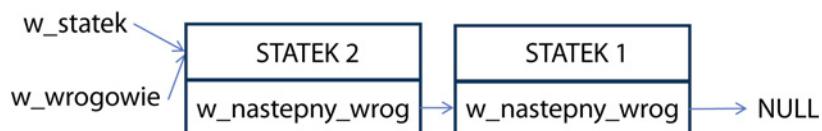
### Stan wyjściowy



### Krok 1. Utworzenie statku



### Kroki 2. i 3. Aktualizacja wskaźników `w_nastepny_wrog` i `w_wrogowie`



O całej tej operacji możesz myśleć jak o przesuwaniu wzdułu listy istniejących elementów za każdym razem, gdy dodawany jest do niej nowy element. Przesuwanie to nie wiąże się z kopowaniem całej listy, jak to jest w przypadku tablic. Zamiast tego aktualizowany jest wskaźnik do początku listy, dzięki czemu wskazuje on jej nowy pierwszy element. Pierwszy element listy nazywany jest jej **głową**. Zwykle będziesz dysponować wskaźnikiem wskazującym głowę listy, którym jest w naszym przypadku `w_wrogowie`. Pod koniec działania funkcji zarówno `w_statek`, jak i `w_wrogowie` wskazują to samo miejsce, ale wcześniej potrzebowaliśmy, aby wskaźnik `w_statek` odnosił się do nowej pamięci, dzięki czemu mogliśmy zmodyfikować pole `w_nastepny_wrog` w taki sposób, aby wskazywało poprzednią głowę listy, zapisaną w zmiennej `w_wrogowie`.

Chociaż funkcja, którą napisałem, korzysta ze zmiennej globalnej, mógłbyś przekształcić ją w funkcję pobierającą głowę listy, dzięki czemu mogłaby ona działać z dowolną listą, a nie tylko z jedną listą globalną. Taka funkcja mogłaby wyglądać następująco:

```

WrogiStatekKosmiczny* dodajDoListyNowegoWroga (WrogiStatekKosmiczny* w_lista)
{
    WrogiStatekKosmiczny* w_statek = new WrogiStatekKosmiczny;
    w_statek->wspolrzedna_x = 0;
    w_statek->wspolrzedna_y = 0;
    w_statek->moc_broni = 20;
    w_statek->w_nastepny_wrog = w_lista;
    return w_statek;
}

```

Zwróć uwagę, że funkcja ta różni się od funkcji generujNowegoWroga, gdyż nie zwraca nowego wroga, tylko wskaźnik do listy. Ponieważ nie może ona modyfikować zmiennej globalnej skojarzonej z listą ani wskaźnika, który jest jej przekazywany (a tylko element, który ten wskaźnik wskazuje), potrzebny jest sposób na zwrócenie nowego początku listy<sup>1</sup>. W takim przypadku w celu dodania nowego elementu do listy wywołanie funkcji może wyglądać następująco:

```
w_lista = dodajDoListyNowegoWroga (w_lista);
```

Interfejs funkcji dodajDoListyNowegoWroga pozwala wybrać bazową listę, która zostanie użyta, oraz zdecydować, gdzie zapisać zwróconą listę.

Aby odtworzyć zachowanie, które było zaimplementowane wcześniej przy użyciu zmiennej globalnej w\_wrogowie, należy napisać:

```
w_wrogowie = dodajDoListyNowegoWroga( w_wrogowie );
```

## Przeglądanie listy powiązanej

Na razie wszystko w porządku — wiemy już, jak przechowywać elementy na liście. Ale w jaki sposób korzystać z takiej listy w celu robienia różnych rzeczy? Poznaliśmy sposoby odczytywania poszczególnych elementów tablicy przy użyciu pętli, co umożliwia **iterowanie** (wyszukane słowo oznaczające działanie w pętli) po tablicach. Dowiedzmy się teraz, jak zrobić to samo w odniesieniu do list powiązanych — technika ta nazywana jest **przegląaniem** listy.

Aby przejść do następnego elementu listy, potrzebny jest jedynie element bieżący. Możesz napisać pętlę, która zawiera zmienną przechowującą wskaźnik do bieżącego elementu listy. Po przeprowadzeniu operacji na tym elemencie zmienna aktualizuje wskaźnik, aby wskazywał następny element listy.

Spójrzmy na przykładowy kod, który ulepsza wszystkie bronie wrogów w grze (na przykład dlatego, że gracz dostał się na kolejny poziom):

```

WrogiStatekKosmiczny *w_biezacy = w_wrogowie;

while ( w_biezacy != NULL )
{
    ulepszBron( w_biezacy );
    w_biezacy = w_biezacy->w_nastepny_wrog;
}

```

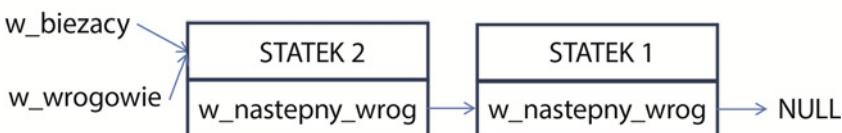
Hej! Ten kod jest prawie tak krótki jak podczas przechodzenia przez tablicę! Zmienna w\_biezacy przechowuje namiary na bieżący element listy, z którym mamy do czynienia. Początkowo

<sup>1</sup> Jeśli chcesz poćwiczyć umysł, zamiast zwracać oryginalną wartość, spróbuj rozwiązać ten problem przy użyciu wskaźnika do wskaźnika.

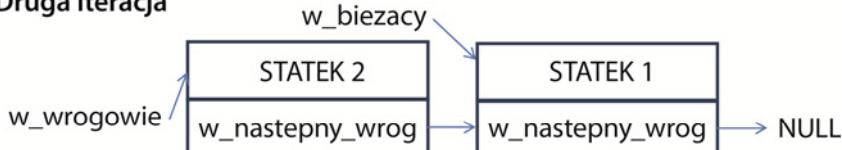
wskazuje ona pierwszego wroga na liście (niezależnie od tego, co wskazuje `w_wrogowie`). Dopóki zmienna `w_biezacy` nie ma wartości `NULL` (co oznacza, że nie dotarliśmy jeszcze do końca listy), ulepszamy bronie bieżącego wroga i aktualizujemy ją, aby wskazywała kolejnego wroga na liście.

Zwróć uwagę, że `w_biezacy` zmienia wskazywane przez siebie elementy, natomiast `w_wrogowie` i pozostałe zmienne wciąż wskazują to samo miejsce. Na tym polega moc wskaźników! Możesz przemieszczać się wzduł struktury danych, zmieniając po prostu wskazywane przez nie miejsce bez konieczności kopiowania czegokolwiek. Przez cały czas istnieje tylko jedna kopia każdego ze statków, co umożliwia ulepszanie przez kod programu broni oryginalnych statków zamiast ich kopii. Oto graficzna reprezentacja wyglądu struktur i zmiennych podczas iterowania po liście.

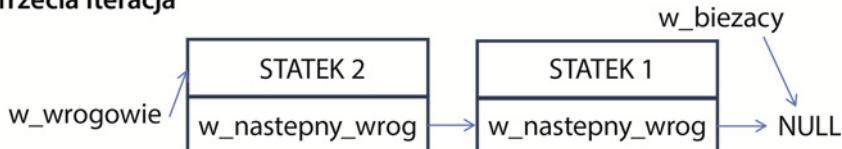
### Pierwsza iteracja



### Druga iteracja



### Trzecia iteracja



## Oswajanie list powiązanych

Listy powiązane umożliwiają łatwe dodawanie nowej pamięci do struktur danych bez potrzeby kopiowania dużych ilości pamięci i bez konieczności przetasowywania tablic. Dzięki listom powiązanym możliwe jest także realizowanie takich operacji jak wstawianie elementów w środk listy oraz ich usuwanie. Pełna implementacja list powiązanych wymaga, aby wszystkie te operacje były wspierane.

Listy powiązane mają swój mały sekret: prawdopodobnie nigdy nie będziesz musiał implementować ich samodzielnie! Zawsze będziesz mógł skorzystać ze standardowej biblioteki szablonów, co omówię już wkrótce. Ważność list powiązanych polega jednak na tym, że bardzo często będziemy korzystać z podobnych technik w celu tworzenia interesujących struktur danych. Nie wpuściłem Cię w maliny — to, czego dowiedziałeś się w tym rozdziale, z pewnością będzie przydatne, nawet jeśli nigdy nie napiszesz własnej listy powiązanej. Co więcej, rozumiejąc sposób implementacji list powiązanych, będziesz lepiej orientować się w kompromisach związanych z podjęciem decyzji, czy użyć takiej listy, czy może tablicy.

## Tablice a listy powiązane

Najważniejszą zaletą list powiązanych (w porównaniu z tablicami) jest łatwość zmiany wielkości listy oraz dodawania do niej nowych elementów, co nie wymaga przesuwania wszystkich elementów listy. Na przykład bardzo proste jest umieszczenie na liście nowego węzła.

A gdybyś tak chciał wstawić do listy nowy element, a przy tym zachować ją posortowaną? Jeśli elementy Twojej listy to 1, 2, 5, 9 i 10, a chciałbyś do niej wstawić 6, musiałbyś umieścić szóstkę między piątką a dziewiątką. W przypadku tablicy wiązałoby się to ze zmianą jej wielkości w celu zrobienia miejsca na nowy element i z przesunięciem każdego elementu, począwszy od 9, w kierunku końca tablicy. Gdyby po dziesiątce Twoja lista miała jeszcze tysiąc elementów, musiałbyś przesunąć je wszystkie o jeden element. Innymi słowy, wydajność wstawiania elementów w środku listy jest proporcjonalna do długości tablicy. W przypadku listy powiązanej wystarczy zmodyfikować element piąty, aby wskazywał nowy element, oraz nowy element, aby wskazywał element dziewiąty — to wszystko! Taka operacja zabiera tyle samo czasu niezależnie od wielkości listy.

Najważniejsza przewaga tablicy nad listą powiązaną polega na tym, że tablica umożliwia bardzo szybki dostęp do dowolnego jej elementu na podstawie indeksu. Z kolei lista powiązana wymaga przejrzenia wszystkich jej elementów, aż do natrafienia na element szukany. Właściwość ta oznacza, że aby można było skorzystać z przewagi tablic, indeks powinien mieć logiczny związek z wartościami zapisanymi w zbiorze elementów. W przeciwnym razie w celu odszukania potrzebnego elementu będziesz musiał sprawdzić cały ich zbiór.

Możesz na przykład utworzyć ankietę wyborczą, w której głosujący za pomocą liczb od 0 do 9 wyrażają swoje preferencje wyborcze względem kandydatów, którym nadano te same numery. Indeks tablicy odpowiada numerowi kandydata, a wartość tablicy w danej lokalizacji jest równa liczbie głosów oddanych na tego kandydata. Nie ma naturalnego związku między kandydatami a numerami, ale możemy taki związek utworzyć, przypisując jeden numer każdemu z kandydatów. Dzięki temu, korzystając z tych numerów, będziemy mogli uzyskać informacje o poszczególnych kandydatach.

Oto prosta implementacja pokazująca, jak można rozwiązać powyższy problem za pomocą tablicy:

### Przykładowy kod 39.: glosowanie.cpp

```
#include <iostream>

using namespace std;

int main ()
{
    int glosy[ 10 ];

    // Gwarantujemy, że głosowanie nie będzie zafalsowane (zerujemy tablice)
    for ( int i = 0; i < 10; ++i )
    {
        glosy[ i ] = 0;
    }

    int kandydat;
    cout << "Proszę zagłosować na swoich kandydatów za pomocą ich numerów:
    ↗0) Józef 1) Robert 2) Maria 3) Zuzanna 4) Małgorzata 5) Elżbieta 6) Aleksander
    ↗7) Tomasz 8) Andrzej 9) Irena" << '\n';
```

```
cin >> kandydat;

// Odczytujemy glosy, dopóki użytkownik nie wprowadzi nieważnego numeru kandydata
while ( 0 <= kandydat && kandydat <= 9 )
{
    // Zwróć uwagę, że nie możemy użyć pętli do-while, ponieważ musimy
    // sprawdzić, czy wprowadzono poprawny numer kandydata jeszcze przed
    // aktualizacją tablicy. Pętla do-while wymagałaby oddania głosu
    // na kandydata, sprawdzenia ważności tego głosu oraz zwiększenia
    // liczby głosów oddanych na danego kandydata
    glosy[ kandydat ]++;
    cout << "Proszę oddać następny głos: ";
    cin >> kandydat;
}
// Pokaż wyniki głosowania
for ( int i = 0; i < 10; ++i )
{
    cout << glosy[ i ] << '\n';
}
}
```

Zwróć uwagę na to, jak łatwa jest aktualizacja liczby głosów oddanych na poszczególnych kandydatów.

Moglibyśmy udoskonalić ten program i zastosować tablicę struktur, w której każda struktura przechowywałaby liczbę głosów razem z imieniem kandydata. Takie rozwiązanie umożliwiłoby wyświetlanie kandydatów łącznie z oddanymi na nich głosami.

Wyobraź sobie, co by trzeba zrobić, aby ten sam cel osiągnąć za pomocą listy powiązanej. Kod musiałby sprawdzać każdy element, aż dotarłby do potrzebnego kandydata. Sprawdzenie głosów oddanych na kandydata nr 5 wymagałoby pętli przehodzącej od węzła kandydata nr 0 do węzła kandydata nr 1, następnie od węzła kandydata nr 1 do węzła kandydata nr 2 itd.; nie ma możliwości bezpośredniego przejścia do środka listy powiązanej.

Czas potrzebny na uzyskanie dostępu do elementu tablicy na podstawie indeksu jest **stały**, co oznacza, że nie ulega zmianie wraz ze wzrostem rozmiaru tablicy. Z drugiej strony czas niezbędny do odnalezienia elementu na liście powiązanej jest proporcjonalny do wielkości listy, niezależnie od tego, czy indeks istnieje, czy nie. Wraz ze wzrostem wielkości listy jej przeszukiwanie staje się coraz wolniejsze.

Gdybyś chciał zrealizować ankietę wyborczą przy użyciu listy powiązanej, przypisywanie kandydatom numerów nie miałoby sensu. Równie dobrze mógłbyś szukać kandydatów na podstawie nazwisk (porównywanie nazwisk przebiega wolniej od porównywania indeksów, ale jeśli decydujesz się na użycie listy powiązanej, prawdopodobnie nie zależy Ci na maksymalnej efektywności kodu).

## Ile miejsca potrzeba na listę powiązaną?

Wielkość miejsca zajmowanego przez strukturę danych jest ważnym czynnikiem, gdy liczba elementów tej struktury jest bardzo duża. W przypadku małych struktur różnica nie będzie widoczna, ale zajmowanie dwukrotnie większej przestrzeni może mieć duże znaczenie w odniesieniu do ogromnych struktur danych.

Tablice zwykle zajmują mniej miejsca w przeliczeniu na jeden element. Listy powiązane wymagają zarówno samego elementu, jak i wskaźnika do następnego elementu listy, co oznacza, że

już na samym początku wymagają one mniej więcej dwa razy tyle miejsca w przeliczeniu na jeden jej element. Listy powiązane mogą jednak zajmować mniej miejsca, jeśli z góry wiesz, ile elementów należy zapisać. Zamiast alokować wielką tablicę i pozostawiać większość jej elementów pustych, można alokować nowe węzły tylko wtedy, gdy jest to potrzebne, dzięki czemu nie będzie konieczne alokowanie dodatkowej pamięci, która nie zostanie wykorzystana (w celu uniknięcia tego problemu można alokować tablicę dynamicznie, co jednak będzie wymagało kopiowania elementów tablicy przy każdym alokowaniu nowej pamięci, przez co utracimy niektóre korzyści wynikające z oszczędzania pamięci<sup>2</sup>).

## Uwagi dodatkowe

Tablice mogą być wielowymiarowe. Na przykład przedstawienie szachownicy za pomocą tablicy o wymiarach 8 na 8 jest łatwe, podczas gdy odwzorowanie szachownicy przy użyciu listy powiązanej wymagałoby utworzenia listy składającej się z innych list, a uzyskanie dostępu do poszczególnych jej pól byłoby o wiele wolniejsze i trudniejsze do zrozumienia niż w przypadku tablicy.

## Kilka zasad ogólnych

Oto kilka ogólnych zasad, które powinieneś rozważyć, dokonując wyboru między listami powiązanymi a tablicami:

Wybierz **tablice**, gdy potrzebujesz stałego czasu dostępu do elementów na podstawie indeksu i kiedy z góry wiesz, ile elementów będziesz przechowywać lub musisz ograniczyć miejsce przypadające w pamięci na jeden element.

Wybierz **listy powiązane**, gdy powinieneś mieć możliwość nieustanego dodawania nowych elementów<sup>3</sup> lub wielokrotnie musisz wstawiać elementy w środek listy.

Innymi słowy, listy powiązane oraz tablice służą realizacji uzupełniających się celów. Użycie jednego bądź drugiego rozwiązania będzie zależeć od tego, co chcesz osiągnąć.

## Sprawdź się

1. Na czym polega przewaga list powiązanych nad tablicami?
  - A. Każdy element w liście powiązanej zajmuje mniej miejsca.
  - B. Listy powiązane można dynamicznie powiększać w celu umieszczenia w nich nowych elementów bez konieczności kopiowania elementów istniejących.
  - C. W listach powiązanych można szybciej odszukać konkretny element niż w tablicach.
  - D. Elementami przechowywanymi w listach powiązanych mogą być struktury.

<sup>2</sup> Mogłbyś przyjąć takie rozwiązanie tak czy inaczej, zwłaszcza jeśli zależy Ci na stałym czasie dostępu, możliwym do uzyskania dzięki indeksom tablicowym. W przypadku struktur danych nie ma zawsze słusznej odpowiedzi, szczególnie gdy bierzemy pod uwagę rozwiązania, z których żadne nie jest ewidentnie złe.

<sup>3</sup> Klasa vector ze standardowej biblioteki szablonów (SLT) ułatwia dodawanie nowych elementów do struktur danych przypominających tablice, umniejszając tym samym zalety list powiązanych. W rezultacie klasa vector będzie zazwyczaj lepszym wyborem niż listy powiązane albo tablice. Już niedługo zajmiemy się wektorami.

- 2.** Które z poniższych stwierdzeń jest prawdziwe?
- A. Nie ma żadnych powodów, aby w ogóle korzystać z tablic.
  - B. Listy powiązane i tablice charakteryzują się taką samą wydajnością.
  - C. Zarówno listy powiązane, jak i tablice dzięki indeksom gwarantują stały czas dostępu do swoich elementów.
  - D. Dodanie nowego elementu w środku listy powiązanej jest szybsze niż dodanie nowego elementu w środku tablicy.
- 3.** Kiedy zwykle będziesz używać list powiązanych?
- A. Kiedy trzeba przechować tylko jeden element.
  - B. Kiedy liczba elementów, które należy przechować, jest znana podczas komplikacji.
  - C. Kiedy trzeba dynamicznie dodawać i usuwać elementy.
  - D. Kiedy potrzebny jest błyskawiczny dostęp do dowolnego elementu posortowanej listy bez potrzeby przeprowadzania jakichkolwiek iteracji.
- 4.** Dlaczego można zadeklarować listę powiązaną zawierającą referencję do elementu typu tej listy (struct Wezel { Wezel\* w\_nastepny; };)?
- A. To nie jest dozwolone.
  - B. Ponieważ kompilator wie, że tak naprawdę elementy autoreferencyjne nie potrzebują pamięci.
  - C. Ponieważ typ jest wskaźnikiem, potrzebne jest tylko tyle miejsca, ile zajmuje jeden wskaźnik. Pamięć niezbędna do realizacji kolejnego węzła zostanie zaalokowana później.
  - D. Takie rozwiązanie jest dopuszczalne, dopóki zmienna w\_nastepny nie wskazuje następnej struktury.
- 5.** Dlaczego ważne jest, aby na końcu listy powiązanej znajdowała się wartość NULL?
- A. Wartość ta wskazuje koniec listy i nie pozwala, aby kod uzyskał dostęp do niezainicjalizowanej pamięci.
  - B. Wartość NULL przeciwdziała powstawaniu w liście serii odwołań cyklicznych.
  - C. W ten sposób wspomaga się debugowanie kodu — kiedy spróbujesz wyjść poza listę, w programie nastąpi awaria.
  - D. Jeśli nie zapiszemy wartości NULL, lista ze względu na autoreferencję będzie potrzebować nieskończonej pamięci.
- 6.** W czym podobne są tablice i listy powiązane?
- A. Zarówno jedne, jak i drugie umożliwiają szybkie dodawanie nowych elementów w środku swoich struktur.
  - B. Zarówno jedne, jak i drugie umożliwiają sekwencyjne zapisywanie oraz odczytywanie danych.
  - C. Zarówno tablice, jak i listy powiązane mogą z łatwością zwiększać swoje rozmiary poprzez dodawanie do nich nowych elementów.
  - D. Zarówno jedne, jak i drugie gwarantują szybki dostęp do wszystkich swoich elementów.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

1. Napisz program usuwający element z listy powiązanej. Funkcja usuwająca powinna przyjmować tylko element przeznaczony do skasowania. Czy taka funkcja jest łatwa do napisania i czy będzie szybka? Czy można ją uprościć albo przyspieszyć, umieszczając w niej dodatkowy wskaźnik do listy?<sup>4</sup>
2. Napisz program dodający elementy do listy powiązanej, który nie umieszcza elementów na początku listy, ale w kolejności posortowanej.
3. Napisz program szukający elementu listy powiązanej na podstawie jego nazwy.
4. Zaimplementuj grę w kółko i krzyżyk dla dwóch osób. Najpierw w celu odwzorowania planszy użyj listy powiązanej. Następnie wykorzystaj tablicę. Które rozwiązanie jest prostsze? Dlaczego?

---

<sup>4</sup> Podpowiedź: a gdybyś tak miał wskaźnik do poprzedniego węzła? Czy to będzie pomocne?



## Rekurencja

---

Widziałeś już wiele algorytmów bazujących na pętlach, które wykonują wciąż na nowo pewne zadania. Istnieje jeszcze inny sposób na powtarzalne wykonywanie kodu, który nie wymaga pętli i polega na wielokrotnym wywoływaniu funkcji. Technika ta jest nazywana **rekurencją** (lub rekursją). Rekurencja polega na definiowaniu wykonywanych operacji w kategoriach tych operacji. Innymi słowy, sprowadza się ona do pisania funkcji, które wywołują same siebie. Rekurencja przypomina w działaniu pętle, z tym że udostępnia o wiele większe możliwości. Dzięki niej napisanie programów, których realizacja przy zastosowaniu pętli jest praktycznie niemożliwa, staje się zadaniem niemal trywialnym. Rekurencja jest szczególnie przydatna, kiedy zostanie zastosowana w odniesieniu do takich struktur danych jak listy powiązane albo drzewa binarne (które poznasz już wkrótce). W tym oraz następnym rozdziale uzyskasz możliwość zrozumienia podstawowych idei związanych z rekurencją oraz zobaczysz kilka konkretnych przykładów ich zastosowania.

### Jak postrzegać rekurencję?

Rekurencję możemy wyobrazić sobie jako proces, w którym jedna z instrukcji brzmi „powtórz proces”, co wygląda bardzo podobnie do pętli, ponieważ następuje powtórzenie tego samego kodu. Istotnie, pod pewnymi względami rekurencja przypomina w działaniu pętle, ale z drugiej strony ułatwia ona wyrażanie idei, w których wynik wywołania rekurencyjnego jest niezbędny do ukończenia zadania. Rzecz jasna musi istnieć jakiś sposób zakończenia procesu z pominięciem kolejnego przebiegu. Za prosty przykład niech posłuży tu budowa wysokiej na 10 metrów ściany. Jeśli zechć zbudować taką ścianę, najpierw zbuduję ścianę o wysokości 9 metrów, na której położę kolejny metr cegieł. Konceptualnie to jakby stwierdzić, że funkcja „zbuduj ścianę” pobiera wysokość ściany do zbudowania i jeśli wysokość ta jest większa od 1, wywołuje samą siebie, żeby przystąpić do budowania ściany niższej o metr, po czym dokłada 1 metr cegiel.

Oto podstawowa struktura takiego kodu (za chwilę omówię kilka występujących w nim błędów). Ważna idea, którą chcemy wyrazić, polega na tym, że zbudowanie ściany o określonej wysokości można wyrazić w kategoriach budowania niższej ściany.

```
void budujSciane (int wysokosc)
{
    budujSciane( wysokosc - 1 );
    dodajMetrCegiel();
}
```

Czy w kodzie tym nie tkwi jednak pewien problem? Kiedy skończy się wywoływanie funkcji buduj Sciane? Niestety, odpowiedź brzmi: nigdy. Rozwiążanie tego problemu jest łatwe — musimy

zakończyć wywołania rekurencyjne, gdy ściana będzie mieć wysokość 0. Osiągnąwszy wysokość 0, powinniśmy tylko dodać metr cegieł i nie wywoływać już funkcji budujSciane z niższą wartością.

```
void budujSciane (int wysokosc)
{
    if ( wysokosc > 0 )
    {
        budujSciane( wysokosc - 1 );
    }
    dodajMetrCegiel();
}
```

Warunek, po spełnieniu którego funkcja nie wywoła samej siebie, nazywany jest **przypadkiem bazowym** tej funkcji. W naszym przykładzie funkcja budująca ścianę wie, że jeśli osiągniemy poziom ziemi, powinniśmy w celu zbudowania ściany położyć pierwszy metr cegieł. W przeciwnym razie najpierw musimy wybudować niższą ścianę, po czym na jej wierzchu ułożyć metr cegieł. Jeśli masz problem ze zrozumieniem kodu (a rekurencja może wydawać się nieco dziwna, gdy widzisz ją po raz pierwszy), pomyśl, jak przebiega faktyczny proces budowania ściany. Zaczynasz od postanowienia zbudowania ściany o określonej wysokości, po czym stwierdzasz: „Żeby cegły znalazły się na tej wysokości, potrzebuję ścianę niższą o jeden metr cegieł” i tak dalej, aż w końcu oznajmiasz: „Nie potrzebuję już niższej ściany; cegły mogę ułożyć bezpośrednio na ziemi” — to jest Twój przypadek bazowy.

Zwróć uwagę, że powyższy algorytm redukuje problem wyjściowy do mniejszego problemu (zbudowanie niższej ściany), po czym rozwiązuje ten mniejszy problem. Na pewnym etapie mniejszy problem robi się na tyle mały (położenie na ziemi pierwszego metra cegieł), że nie musimy go już bardziej redukować i zyskamy możliwość szybkiego poradzenia sobie z prostym przypadkiem. W rzeczywistym świecie moglibyśmy zbudować ścianę, natomiast w C++ mamy gwarancję, że funkcja w końcu zatrzyma swoje wywołania rekurencyjne. Przypomina to proces projektowania z góry do dołu, który poznaliśmy wcześniej: rozbijaliśmy problem na mniejsze podproblemy, tworzyliśmy funkcje rozwiązujące te podproblemy i budowaliśmy z nich pełny program. W takim przypadku rozkładaliśmy dany problem na podproblemy *różniące się* od problemu, który rozwiązywaliśmy. W przypadku rekurencji rozbijamy problem na jego *mniejsze wersje*.

Kiedy funkcja wywoła sama siebie i powróci z tego wywołania, będzie gotowa, aby przejść do następnego wiersza, który znajduje się po wierszu z jej wywołaniem. Gdy funkcja rekurencyjna wraca z wywołania, może wykonywać inne operacje lub wywoływać następne funkcje. Funkcja budująca ścianę, po wybudowaniu niższej ściany, będzie kontynuować swoje działanie i ułożyć nowy metr cegieł.

Spójrzmy na przykładowy kod, który możesz uruchomić i zobaczyć wynik jego działania. Jak mógłbyś napisać rekurencyjną funkcję wyświetlającą ciąg znaków 123456789987654321? Można to zrobić, pisząc funkcję, która wyświetla daną cyfrę dwa razy: pierwszy raz przed wywołaniem rekurencyjnym i drugi raz po powrocie z wywołania.

### Przykładowy kod 40.: *wyswietl\_liczby.cpp*

```
#include <iostream>

using namespace std;

void wyswietlLiczby (int liczba)
{
```

```

// Dwa wywołania tej funkcji w instrukcji cout otoczą wewnętrzną
// sekwencję zawierającą liczby (liczba+1)...99...(liczba+1)
cout << liczba;
// Ponieważ zaczynamy od wartości mniejszej od 9, musimy rekurencyjnie wyświetlać
// sekwencje dla wartości (liczba+1) ... 99 ... (liczba+1)
if ( liczba < 9 )
{
    wyswietlLiczby( liczba + 1 );
}
cout << liczba;
}

int main ()
{
wyswietlLiczby( 1 );
}

```

Rekurencyjne wywołanie funkcji `wyswietlLiczby( liczba + 1 )` powoduje wyświetlenie sekwencji `(liczba+1)...99...(liczba+1)`. Wyświetlenie zmiennej `liczba` zarówno przed, jak i po wywołaniu funkcji `wyswietlLiczby( liczba + 1 )` sprawia, że z obu stron sekwencji `(liczba+1)...99...(liczba+1)` zostanie wydrukowana zmienna `liczba`, dzięki czemu sekwencja ta przybierze postać `(liczba)(liczba+1)...99...(liczba+1)(liczba)`. Jeśli zmienna `liczba` ma wartość 1, otrzymamy sekwencję `12...99...21`.

O naszej funkcji `wyswietlLiczby` można myśleć także w inny sposób: najpierw pokazuje ona liczby od 1 do 9, za każdym razem wywołując samą siebie. Kiedy zostanie osiągnięty przypadek bazowy, funkcja powróci do każdego z wywołań rekurencyjnych, ponownie wyświetlając wszystkie liczby, lecz tym razem w kolejności, w jakiej następują powroty z wywołań. Ponieważ ostatnie wywołanie funkcji `wyswietlLiczby` miało miejsce przy wartości 9, która jest przypadkiem bazowym, nie nastąpi ponowne wywołanie funkcji, tylko liczba ta zostanie od razu wyświetlona.

Podczas powrotu z ostatniego wywołania funkcja `wyswietlLiczby` znajdzie się w miejscu, w którym zmienna `liczba` wynosiła 8 — wartość ta zostanie wyświetlona. Nastąpi wówczas kolejny powrót, a zmienna `liczba` przyjmie wartość 7, i tak dalej, aż do ukończenia wszystkich pozostałych wywołań rekurencyjnych, gdy funkcja dotrze do swojego pierwszego wywołania z wartością 1. Zostanie wtedy wydrukowana jedynka i program zakończy działanie.

## Rekurencja i struktury danych

Niektóre struktury danych prowadzą wprost do algorytmów rekurencyjnych, ponieważ informacje są w nich zorganizowane w taki sposób, jakby zawierały mniejsze kopie swoich własnych struktur. Algorytmy rekurencyjne działają na zasadzie rozbijania problemu na mniejsze wersje tego samego problemu i dlatego bardzo dobrze współdziałają ze strukturami danych, które są złożone z mniejszych wersji ich samych — przykładem takiej struktury mogą być listy powiązane.

Do tej pory traktowaliśmy listy powiązane jak listy, na początku których można dodawać kolejne węzły. O listach powiązanych można też myśleć w inny sposób: lista taka składa się z pierwszego węzła, który wskazuje następną, mniejszą listę powiązaną. Innymi słowy, listę powiązaną tworzą poszczególne węzły, z których każdy wskazuje kolejny węzeł będący początkiem „reszty listy”.

Takie ujęcie jest bardzo ważne, ponieważ zyskujemy dzięki niemu istotną wskazówkę: możemy pisać programy przetwarzające listy powiązane za pomocą kodu, który obsługuje pierwszy węzeł albo „resztę listy”. Na przykład w celu odszukania określonego węzła listy można wykorzystać następujący prosty algorytm:

- Jeśli znajdujemy się na końcu listy, należy zwrócić NULL.
- W przeciwnym razie, jeśli bieżący węzeł jest przypadkiem bazowym, należy go zwrócić.
- W przeciwnym razie należy przeszukać resztę listy.

Kod realizujący powyższy algorytm może wyglądać następująco:

```
struct wezel
{
    int wartosc;
    wezel *nastepny;
};

wezel* szukaj (wezel* lista, int szukana_wartosc)
{
    if ( lista == NULL )
    {
        return NULL;
    }
    if ( lista->wartosc == szukana_wartosc )
    {
        return lista;
    }
    else
    {
        return szukaj( lista->nastepny, szukana_wartosc );
    }
}
```

Pisząc o wywołaniu rekurencyjnym, wspomniałem, że wywoływana funkcja wykonuje jakąś pracę. Zadanie, które zrealizuje pewna funkcja po otrzymaniu określonych danych wejściowych, jest nazywane jej **kontraktem**. Kontrakt funkcji podsumowuje wykonywaną przez nią pracę. W przypadku funkcji szukającej kontraktem jest odnalezienie na liście określonego węzła. Nasza funkcja jest zaimplementowana następująco: „Jeśli bieżący węzeł jest węzłem, którego potrzebujemy, zwróć go; w przeciwnym razie kontraktem funkcji szukającej będzie odszukanie węzła na liście; według takich samych zasad należy przeszukać resztę listy”.

Ważne jest, aby ponownie nie wywoływać funkcji szukającej dla całej listy, tylko dla jej reszty. Rekurencja zadziała wyłącznie wtedy, gdy:

1. Możesz opracować sposób rozwiązyania problemu, pracując nad rozwiązyaniem mniejszych wersji tego problemu.
2. Możesz rozwiązać przypadek bazowy.

Funkcja szukająca rozwiązuje dwa możliwe przypadki bazowe — albo dotarliśmy do końca listy, albo odnaleźliśmy węzeł, którego szukamy. Jeśli nie zachodzi żaden z tych przypadków, musimy skorzystać z funkcji szukającej w celu rozwiązania mniejszej wersji naszego problemu. Oto cała tajemnica rekurencji. Ma ona zastosowanie wtedy, gdy można rozwiązać mniejszą wersję danego problemu i skorzystać z tego rozwiązania w celu poradzenia sobie z dużym problemem.

Czasami wartość otrzymana z wywołania rekurencyjnego nie jest natychmiast zwracana, tylko zostanie użyta ponownie. Spójrzmy na taki właśnie przykład; będzie to funkcja implementująca silnię (wszyscy podają silnię w swoich przykładach na rekurencję!).

```
silnia( x ) = x * ( x - 1 ) * ( x - 2 ) ... * 1
```

Albo, wyrażając to samo w inny sposób:

```
silnia( x ) =
    if ( x == 1 ) 1
    else x * silnia( x - 1 )
```

Innymi słowy, silnię można obliczyć, mnożąc bieżącą wartość przez silnię mniejszej wartości. W takim przypadku korzystamy z wartości zwróconej przez wywołanie rekurencyjne i robimy z tą wartością coś innego, w naszym przykładzie mnożymy ją przez kolejną wartość.

Oto kod:

```
int silnia (int x)
{
    if ( x == 1 )
    {
        return 1;
    }
    return x * silnia( x - 1 );
}
```

W powyższym kodzie możemy rozwiązać przypadek bazowy dla  $x$  równego 1 albo zabrać się za mniejszy problem  $\text{silnia}(x - 1)$  i na podstawie uzyskanego wyniku obliczyć silnię  $x$ . Także i w tym przypadku każde wywołanie funkcji  $\text{silnia}$  zmniejsza  $x$ , co spowoduje, że w końcu osiągniemy przypadek bazowy.

Zwróci uwagę na to, że rozwiązujeśmy pewien podproblem, po czym korzystamy z uzyskanego wyniku, aby zrobić coś jeszcze. Podczas przeszukiwania listy powiązanej wszystko, co robiliśmy, sprowadzało się do zwracania wyniku otrzymywanego podczas rozwiązywania podproblemu. Rekurencję można stosować na dwa sposoby — albo całość problemu rozwiązać za pomocą wywołania rekurencyjnego, albo uzyskać rozwiązanie podproblemu i otrzymany wynik wykorzystać w celu przeprowadzenia dodatkowych obliczeń.

## Pętle i rekurencja

W niektórych przypadkach algorytm rekurencyjny można wyrazić w prosty sposób w postaci pętli, która ma taką samą strukturę jak algorytm. Przeszukiwanie listy można wyrazić na przykład następująco:

```
wezel *szukaj (wezel *lista, int szukana_wartosc)
{
    while ( 1 )
    {
        if ( lista == NULL )
        {
            return NULL;
        }
        if ( lista->wartosc == szukana_wartosc )
        {
            return lista;
        }
    }
}
```

```
        else
        {
            lista = lista->nastepny;
        }
    }
}
```

W powyższym kodzie użyto dokładnie takich instrukcji warunkowych, jak w wersji rekurencyjnej, dzięki czemu z łatwością możesz porównać oba warianty. Jedyna różnica między dwoma algorytmami polega na tym, że w miejsce rekurencji zastosowano pętlę, a zamiast wywołań rekurencyjnych kod skraca listę za każdym razem, gdy zmiennej `lista` przypisywana jest wartość oznaczająca „resztę listy”. Jest to przypadek, w którym wersja rekurencyjna oraz **iteracyjna** (czyli bazująca na pętli) działają w podobny sposób.

Zwykle bardzo łatwo napisać algorytm rekurencyjny w postaci pętli (i odwrotnie), kiedy nie trzeba nic robić z wynikiem zwracanym z wywołania funkcji rekurencyjnej. Taki rodzaj rekurencji — gdy ostatnią czynnością dokonywaną przez funkcję jest przeprowadzenie znajdującego się na jej końcu wywołania rekurencyjnego — nazywany jest **rekurencją ogonową**. Ponieważ wywołanie rekurencyjne jest ostatnią operacją, nie różni się ono od przejścia do następnej instrukcji pętli. Kiedy nastąpi powrót z kolejnego wywołania funkcji, nie są potrzebne żadne informacje z wcześniejszego jej wywołania. Przypadek z przeszukiwaniem listy stanowi przykład rekurencji ogonowej.

Z drugiej jednak strony, zastanów się nad silnią. Kiedy bazując na implementacji rekurencyjnej, spróbujemy przekształcić ją w pętlę, napotkamy pewien problem.

```
int silnia (int x)
{
    while ( 1 )
    {
        if ( x == 1 )
        {
            return 1;
        }
        // A co wstawić tutaj??
        // return x * silnia( x - 1 );
    }
}
```

Musimy coś zrobić z wynikiem zwracanym przez wywołanie `silnia ( x - 1 )`, w związku z czym nie możemy zastosować tu pętli. Zanim rozwiążemy nasz problem, rzeczywiście najpierw powinniśmy rozwiązać podproblem (ukończenie reszty pętli).

Silnię można jednak dość łatwo przekształcić w pętlę, jeśli jeszcze raz przemyślimy cały problem! Zastanów się nad początkową definicją:

$$\text{silnia}(x) = x * (x - 1) * (x - 2) \dots * 1$$

Jeśli uda nam się zapamiętać bieżącą wartość, będziemy mogli obliczyć silnię, obserwując aktualny wynik mnożenia  $x * (x - 1) * (x - 2) \dots$

```
int silnia (int x)
{
    int biez = x;
    while ( x > 1 )
    {
```

```

        x--;
        biez *= x;
    }
    return x;
}

```

Zauważ, że zamiast rozwiązywać problem, korzystając z wyników uzyskanych na podstawie rozwiązania podproblemów (mniejszych silni), przeprowadzamy nasze obliczenia w odwrotnym kierunku. Gdybyśmy na przykład musieli policzyć 5 silnia, funkcja rekurencyjna wykonałaby mnożenie w następującej kolejności:

1 \* 2 \* 3 \* 4 \* 5

Z kolei w rozwiązaniu iteracyjnym obliczenia przebiegają w przeciwnym kierunku:

5 \* 4 \* 3 \* 2 \* 1

W tym przypadku możliwe jest zarówno rozwiązywanie iteracyjne, jak i rekurencyjne, z tym że mają one inne struktury. Ponowne przemyślenie struktury algorytmu umożliwiło napisanie funkcji w postaci bardzo prostej pętli. W niektórych jednak przypadkach uzyskanie rozwiązania bazującego na pętli może być o wiele bardziej skomplikowane niż w przykładzie z silnią. Od tego, czy uda Ci się znaleźć algorytm iteracyjny, będzie zależeć Twoja decyzja o zastosowaniu rekurencji. W przypadku silni było to dość łatwe, ale w innych przypadkach znalezienie rozwiązania iteracyjnego może być bardzo trudne. Już niedługo będziesz mógł się o tym przekonać.

## Stos

Nadeszła pora, aby opowiedzieć nieco więcej o tym, jak przebiega wywołanie funkcji, i rzucić okiem na kilka ładnych rysunków. Kiedy już dowiesz się, na czym polega wywoływanie funkcji, lepiej zrozumiesz rekurencję i zorientujesz się, dlaczego niektóre algorytmy są łatwiejsze do zrealizowania przy użyciu rekurencji niż za pomocą pętli.

Wszystkie informacje wykorzystywane przez funkcję są przechowywane wewnętrznie na **stosie**. Wyobraź sobie stos talerzy. Możesz dołożyć nowy talerz na szczyt stosu albo zdjąć talerz z jego wierzchu. Stos w pamięci komputera działa tak samo jak stos talerzy, z tym że zamiast talerzy mamy coś, co jest nazywane **ramkami stosu**. Kiedy funkcja jest wywoływana, otrzymuje nową ramkę stosu i korzysta z niej w celu przechowywania wszystkich lokalnych zmiennych, które zostaną użyte. Gdy zostanie wywołana inna funkcja, wcześniejsza ramka stosu jest zachowywana, a na wierzchu stosu odkładana jest nowa ramka, udostępniająca nowej funkcji miejsce na jej zmienne. Aktualnie wykonywana funkcja zawsze korzysta z ramki stosu znajdującej się na wierzchu stosu.

W najprostszym przypadku, gdy wykonywana jest tylko funkcja `main`, stos wygląda następująco:

Zmienne w main

Mamy tylko jedną funkcję, `main`, a stos zawiera wyłącznie zmienne z tej funkcji.

Jeśli w funkcji `main` nastąpi wywołanie innej funkcji, nowa funkcja utworzy ramkę stosu na wierzchu ramki funkcji `main`, co będzie wyglądać następująco:

|                           |
|---------------------------|
| Zmienne w drugiej funkcji |
| Zmienne w main            |

Bieżąca funkcja uzyskała miejsce na swoje zmienne oraz możliwość pracy z nimi bez kolidowania ze zmiennymi, z których korzystała funkcja `main`. Jeżeli druga funkcja wywoła trzecią funkcję, stos przyjmie następujący wygląd:

|                            |
|----------------------------|
| Zmienne w trzeciej funkcji |
| Zmienne w drugiej funkcji  |
| Zmienne w main             |

Wywołana teraz funkcja ma do dyspozycji własną ramkę stosu. Każde wywołanie funkcji tworzy nową ramkę stosu. Po powrocie z funkcji stos przyjmie swój poprzedni wygląd:

|                           |
|---------------------------|
| Zmienne w drugiej funkcji |
| Zmienne w main            |

Jeśli druga funkcja powróci do `main`, stos ponownie przyjmie formę, jaką miał z jedną tylko ramką stosu:

|                |
|----------------|
| Zmienne w main |
|----------------|

Aktywna ramka stosu każdorazowo jest przypisywana do funkcji, która właśnie jest wykonywana, i zawsze znajduje się na wierzchu stosu.

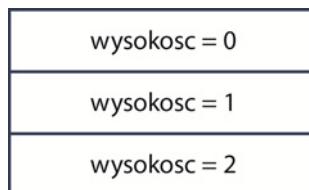
Oprócz zmiennych, które są używane w funkcjach, ramka stosu zawiera także argumenty przekazywane do funkcji oraz wiersz kodu, do którego funkcja powinna powrócić po zakończeniu swojego działania. Innymi słowy, ramka stosu przechowuje informację o miejscu, w którym funkcja się znajduje, i wszystkie dane, z których funkcja korzysta. Rekurencyjne wywołanie funkcji tworzy nową ramkę stosu, nawet jeśli wywołana zostanie ta sama funkcja. To właśnie dlatego rekurencja działa — każde wywołanie funkcji ma swoją niepowtarzaną ramkę stosu, argumenty i zmienne. Dzięki temu poszczególne wywoływanie funkcji dysponują własnymi informacjami, w związku z czym możliwa jest praca nad mniejszą wersją wyjściowego problemu, który jest reprezentowany przez swoje zmienne.

Kiedy funkcja kończy działanie, jak widzieliśmy to na rysunku, usuwa własną ramkę z wierzchu stosu i wraca do punktu wywołania. W ten sposób oddaje funkcji, która ją wywołała, możliwość korzystania z ramki tej funkcji.

Istotne jest, że ramka funkcji przechowuje miejsce, do którego należy powrócić, oraz że można ją usunąć ze stosu po zakończeniu działania funkcji. Bez właściwej ramki stosu funkcja wywołująca nie mogłaby poprawnie kontynuować pracy po zakończeniu działania funkcji wywoływanej — nie miałaby na przykład prawidłowych wartości w swoich zmiennych lokalnych.

Możesz o tym myśleć następująco: kiedy wywoływana jest nowa funkcja, wszystko, co jest potrzebne do działania poprzedniej funkcji, jest zapisywane. To tak, jakbyś pracował nad pewnym projektem i zdecydował się wyjść na obiad. Sporządziłbyś wtedy notatki, aby po powrocie z obiadu pamiętać, na którym etapie projektu się znajdujesz, i aby móc go dokończyć. Stos umożliwia komputerowi szczegółowe pamiętanie, czym w każdej chwili się zajmował — o wiele bardziej szczegółowe, niż sami moglibyśmy to opisać.

Oto stos pokazujący trzy rekurencyjne wywołania funkcji budujSciane, począwszy od wysokości równej 2. Jak widzisz, każda ramka stosu przechowuje nową wysokość ściany, która została przekazana do funkcji (zwróć uwagę, że wywołanie ze zmiennej wysokość o wartości 0 znajduje się na wierzchu stosu, która to wartość znajduje się na *dole rzeczywistej ściany*).



Taki sposób rysowania stosu często jest przedstawiany następująco:

```

budujSciane( wysokość = 0 )
budujSciane( wysokość = 1 )
budujSciane( wysokość = 2 )
main()
  
```

Każda funkcja jest pokazywana powyżej funkcji, która ją wywołuje, i podawane są jej argumenty. Z techniki tej można korzystać, aby łatwiej było zrozumieć, jak działa dana funkcja rekurencyjna. Czasami może okazać się, że oprócz umieszczenia obok każdej ramki nazwy funkcji i jej argumentów, pomocne będzie rozpisanie także jej zmiennych lokalnych.

## Zaleta stosu

Najważniejszą zaletą rekurencji jest możliwość dysponowania stosem wywołań funkcji zamiast pojedynczą ramką stosu. Dzięki algorytmom rekurencyjnym istnieje możliwość korzystania z przewagi, jaką daje posiadanie dodatkowych informacji przechowywanych w każdej ramce stosu, podczas gdy pętle udostępniają jedynie zbiór zmiennych lokalnych. W rezultacie funkcja rekurencyjna może czekać, aż nastąpi powrót z wywołania, i kontynuować pracę od miejsca, w którym to wywołanie nastąpiło. Aby napisać pętlę działającą w taki sposób, musiałbyś zaimplementować własną wersję stosu.

## Wady rekurencji

Stos ma stałą wielkość, co oznacza, że rekurencja nie może przebiegać w nieskończoność. W pewnym momencie zabraknie pamięci na umieszczenie kolejnej ramki na wierzchu stosu — tak jak może zabraknąć miejsca na kolejny talerz na stosie naczyń w kredensie.

Oto prosty przykład rekurencji, która teoretycznie powinna działać w nieskończoność:

```

void rekurencja ()
{
    rekurencja(); // Funkcja wywołuje samą siebie
}
  
```

```
int main ()
{
    rekurencja(); // Rozpoczęcie rekurencji
}
```

Ostatecznie zostanie wyczerpana cała przestrzeń stosu, a w programie wystąpi awaria związana z **przepelnieniem stosu**. Przepelenie stosu zachodzi wtedy, gdy na stosie nie ma już miejsca. W takiej sytuacji nie można przeprowadzić kolejnego wywołania funkcji i kiedy program podejmie próbę jej wywołania, nastąpi awaria. Tego typu wydarzenia, chociaż nieczęste, są zwykle wynikiem działania funkcji rekurencyjnej o złe napisanym kodzie. Na przykład w napisanej przez mnie funkcji obliczającej silnię występuje pewien drobny błąd — nie sprawdza ona, czy podstawą obliczeń jest liczba ujemna. Gdyby funkcja wywołująca silnię przekazała wartość  $-1$ , niemal na pewno nastąpiłoby przepelenie stosu (wypróbuj taki przypadek osobiście — przepelenie stosu wywoła w programie awarię, ale nie uszkodzi Twojego komputera).

Oto prosty przykład pokazujący, ile rekurencyjnych wywołań niewielkiej funkcji potrzeba, aby wyczerpała się pamięć stosu (im większa jest ramka stosu funkcji, tym mniej rekurencyjnych wywołań można przeprowadzić, chociaż bardzo rzadko napotkasz takie ograniczenie, gdy przypadki bazowe będą zdefiniowanie poprawnie).

```
#include <iostream>

using namespace std;

void rekurencja (int licznik) // Każde wywołanie funkcji dostanie własny licznik
{
    cout << licznik << "\n";
    // Nie ma potrzeby zwiększenia licznika, ponieważ zmienne
    // wszystkich funkcji są niezależne od siebie (dzięki czemu
    // licznik w każdej ramce stosu jest inicjalizowany wartością
    // o 1 większą niż przy poprzednim wywołaniu)
    rekurencja( licznik + 1 );
}

int main ()
{
    rekurencja( 1 ); // Jest to pierwsze wywołanie, dlatego zaczynamy od 1
}
```

## Debugowanie przepelnienia stosu

Gdy próbujesz debugować przepelenie stosu, najważniejsze jest stwierdzenie, która funkcja (albo ich grupa) wielokrotnie dodaje nowe ramki stosu. Jeżeli na przykład podczas uruchamiania programu z poprzedniego przykładu korzystałbyś z debugera (który omówię w rozdziale „Debugowanie w Code::Blocks”), dowiedziałbyś się, że w chwili wystąpienia awarii stos wyglądał następująco:

```
rekurencja( 10000 );
rekurencja( 9999 );
rekurencja( 9998 );
...
rekurencja( 1 );
main()
```

W tym przypadku łatwo stwierdzić, która funkcja była zamieszana w awarię — najwidoczniej w funkcji tej brakuje jakiegoś przypadku bazowego, prawdopodobnie związanego z zatrzymaniem wywołań, gdy argument rekurencyjny osiągnie określony rozmiar.

Czasami możesz mieć do czynienia z **rekurencją wzajemną**, gdy dwie funkcje wywołują się nawzajem.

Oto zmyślony przykład, który ponownie dotyczy obliczania silni, gdzie użyto dwóch funkcji w celu jej obliczenia — pierwsza oblicza silnię liczb nieparzystych, a druga parzystych:

```
int silnia_nieparzyste (int x)
{
    if ( x == 0 )
    {
        return 1;
    }
    return silnia_parzyste( x - 1 );
}

int silnia_parzyste (int x)
{
    if ( x == 0 )
    {
        return 1;
    }
    return silnia_nieparzyste( x - 1 );
}

int silnia (int x)
{
    if ( x % 2 == 0 )
    {
        return silnia_parzyste( x );
    }
    else
    {
        return silnia_nieparzyste( x );
    }
}
```

Zauważ, że przypadki bazowe nie strzegą nas przed wprowadzeniem liczb ujemnych. Wywołanie `silnia(-1)` doprowadzi stos do mniej więcej takiego stanu:

```
silnia_parzyste( -10000 )
silnia_nieparzyste( -9999 )
silnia_parzyste( -9998 )
silnia_nieparzyste( -9997 )
```

Wystarczy spojrzeć na stos, aby dojść do wniosku, że występuje problem z przypadkiem bazowym i że dwie funkcje nie przestają wywoływać się nawzajem. Następnym krokiem powinno być sprawdzenie kodu i próba określenia, która z funkcji powinna uwzględnić przypadek bazowy z liczbą ujemną. Podczas obliczania silni sensowne jest, aby obie funkcje przeprowadzały ten sam test niezależnie od siebie. W innych przypadkach tylko jedna z funkcji może być odpowiedzialna za pilnowanie przypadku bazowego.

Jeśli debugujesz złożone wywołania rekurencyjne, pomocne będzie odnalezienie funkcji, które cyklicznie wywołują się nawzajem, w tym przypadku `silnia_parzyste` i `silnia_nieparzyste`. W niektórych przypadkach mogą występować o wiele dłuższe okresy dzielące poszczególne powtórki; powinieneś odkryć cały zestaw cyklicznych wywołań, a następnie określić przyczynę powtarzania się takiego zestawu funkcji.

## Wydajność

Rekurencja wymaga wykonania wielu wywołań funkcji. Każda funkcja musi skonfigurować ramkę stosu i przekazać argumenty, co wiąże się z dodatkowym narzutem, który nie występuje w przypadku pętli. W większości przypadków taki narzut nie będzie miał znaczenia podczas pracy na nowoczesnych komputerach, ale jeśli ma być wykonywany bardzo często (miliony albo miliardy wywołań w krótkim czasie), może stać się dostrzegalny.

## Oswajanie rekurencji

Rekurencja umożliwia tworzenie algorytmów, które rozwiązuje problemy, rozbijając je na ich mniejsze wersje. Gwarantuje ona także większe możliwości niż pętle, ponieważ funkcje rekurencyjne korzystają ze stosu, który przechowuje bieżący stan każdego ich wywołania, umożliwiając im kontynuowanie działania po uzyskaniu rozwiązania podproblemu.

Rekurencyjna implementacja danego algorytmu często będzie się wydawać bardziej naturalna niż implementacja bazująca na pętli. W następnym rozdziale zobaczymy kilka tego typu przykładów, które dotyczą drzew binarnych. W miarę pracy nad kolejnymi programami stwierdzisz, że rekurencja ułatwia rozpatrywanie o wiele szerszego zakresu problemów, niż by to było możliwe w przypadku rozwiązywania ich wyłącznie za pomocą pętli.

Oto kilka ogólnych zasad, które powinieneś rozważyć, dokonując wyboru między rekurencją a pętlami:

Wybierz rekurencję, gdy:

1. Rozwiązywanie problemu wymaga rozbicia go na mniejsze wersje jego samego i nie ma oczywistego sposobu na rozpisanie go w postaci pętli.
2. Rekurencyjna jest struktura danych, z jaką pracujesz (na przykład lista powiązana).

Wybierz pętlę, gdy:

1. Jest oczywiste, jak rozwiązać problem za pomocą prostej pętli; na przykład dodawanie liczb z listy można zrealizować za pomocą funkcji rekurencyjnej, jednak przyjęcie takiego rozwiązania nie będzie opłacalne.
2. Kiedy korzystasz z indeksowanej liczbowo struktury danych, na przykład z tablicy.

## Sprawdź się

1. Kiedy masz do czynienia z rekurencją ogonową?
  - A. Kiedy wołasz swojego psa.
  - B. Kiedy funkcja wywołuje sama siebie.
  - C. Kiedy ostatnią czynnością, jaką wykonuje funkcja rekurencyjna przed powrotem, jest wywołanie siebie samej.
  - D. Kiedy algorytm rekurencyjny można napisać w postaci pętli.
2. Kiedy skorzystasz z rekurencji?
  - A. Kiedy algorytmu nie można rozpisać w postaci pętli.
  - B. Kiedy naturalniejsze jest wyrażenie algorytmu pod postacią podproblemów niż w formie pętli.

- C. Nigdy, rekurencja jest zbyt trudna ☺.
  - D. Podczas pracy z tablicami i listami łączonymi.
- 3.** Jakie elementy są wymagane w algorytmie rekurencyjnym?
- A. Przypadek bazowy i wywołanie rekurencyjne.
  - B. Przypadek bazowy i jakiś sposób na rozbicie problemu na jego mniejsze wersje.
  - C. Jakiś sposób na połączenie mniejszych wersji problemu.
  - D. Każde z powyższych.
- 4.** Co może się wydarzyć, jeśli przypadek bazowy jest niekompletny?
- A. Algorytm może zbyt wcześnie zakończyć działanie.
  - B. Kompilator wykryje błąd i zgłosi swoje zastrzeżenia.
  - C. To nie stanowi problemu.
  - D. Może wystąpić przepełnienie stosu.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

1. Napisz algorytm rekurencyjny obliczający funkcję potęgową  $\text{pot}(x, y) = x^y$ .
2. Napisz funkcję rekurencyjną pobierającą tablicę i wyświetlającą jej elementy w odwrotnej kolejności, która jednak nie zaczyna swojej pracy od ostatniego elementu tablicy (innymi słowy, nie pisz ekwiwalentu pętli rozpoczynającej wyświetlanie od końca tablicy).
3. Napisz algorytm rekurencyjny usuwający elementy z listy powiązanej oraz algorytm rekurencyjny dodający elementy do listy powiązanej. Spróbuj napisać takie same algorytmy z zastosowaniem iteracji. Czy bardziej naturalna wydaje Ci się implementacja rekurencyjna, czy może iteracyjna?
4. Napisz funkcję rekurencyjną, która pobiera posortowaną tablicę, element docelowy oraz odszukuje ten element w tablicy (jeśli nie znajdzie elementu, powinna zwrócić -1). Jak szybkie może być takie szukanie? Czy można osiągnąć lepszy wynik bez potrzeby sprawdzania każdego elementu?
5. Napisz rekurencyjny algorytm rozwiązujejący problem. Pod adresem <http://wipos.p.lodz.pl/zylia/games/hanoi5p.html> znajdziesz stronę zawierającą opis problemu, a także pozwalającą Ci się z nim zmierzyć.



## Drzewa binarne

---

W rozdziale tym opisano jedną z najciekawszych i najbardziej przydatnych struktur danych, jaką są drzewa binarne. Stanowią one doskonały przykład, jak za pomocą rekurencji i wskaźników można osiągać zdumiewające rezultaty. Czy jednak wymagają one, abyś w pełni rozumiał rekurencję oraz idee leżące u podstaw list powiązanych? Skąd miałbym to wiedzieć? Widziałem już niejednego studenta zmagającego się z drzewami binarnymi, chociaż wcześniej świetnie sobie radził ze wskaźnikami i rekurencją. W drzewach binarnych nie ma nic z gruntu trudnego; nic, co powstrzymałoby Cię od ich zrozumienia. Potrzebne Ci jednak będą solidne fundamenty. Jeśli koncepcje przedstawione w tym rozdziale sprawią Ci kłopoty, prawdopodobnie będziesz musiał lepiej zrozumieć wskaźniki i rekurencję. Spróbuj ponownie przeczytać wcześniejsze rozdziały i wykonaj zaproponowane w nich ćwiczenia.

Listy powiązane znakomicie nadają się do sporządzania zestawień różnych elementów. Niestety, odszukanie w nich konkretnego elementu może zajść mnóstwo czasu. Co więcej, nawet tablice nie będą pomocne, jeśli dane tworzą jedną długą listę pozbawioną jakiekolwiek struktury. Tablicę można posortować, co umożliwia przeprowadzanie w niej bardzo szybkich wyszukiwań, jednak wstawianie do niej elementów nadal będzie trudne. Jeśli zechcesz zachować tablicę w porządku posortowanym, wstawienie każdego nowego elementu będzie wymagać dokonania w niej wielu przesunięć. Poza tym szybkie odnajdowanie elementów jest bardzo ważne. Oto kilka przykładów:

1. Jeśli tworzysz grę typu MMORPG, taką jak *World of Warcraft*, i chciałbyś, aby gracze mogli się szybko do niej logować, musisz mieć możliwość błyskawicznego odszukania danego gracza.
2. Jeśli pracujesz nad oprogramowaniem przetwarzającym dane z kart kredytowych i w każdej godzinie musisz obsłużyć miliony transakcji, powinieneś mieć możliwość szybkiego ustalania stanu konta związanego z numerem danej karty kredytowej.
3. Jeśli pracujesz z urządzeniami niskonapięciowymi, takimi jak smartfony, i wyświetlasz użytkownikowi książkę adresową, nie chciałbyś, aby proces ten trwał długo z powodu wolnej struktury danych.

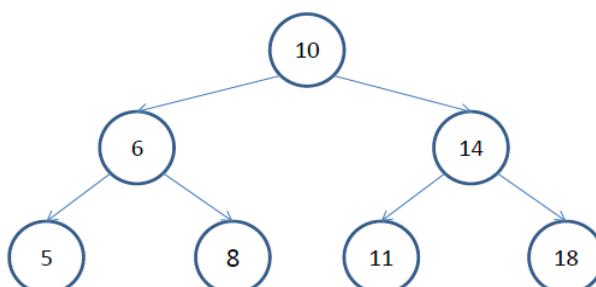
Rozdział ten w całości traktuje o narzędziach, które są potrzebne do rozwiązywania takich (a także innych) problemów.

Podstawowa idea leżąca u podstaw tego typu rozwiązania polega na przechowywaniu elementów w strukturze przypominającej listę powiązaną, co znaczy, że w celu ustrukturyzowania pamięci — tak samo jak w przypadku list powiązanych — użyto wskaźników, ale w sposób ułatwiający odszukiwanie elementów. W tym celu pamięć musi jednak mieć bardziej złożoną strukturę, niż to jest w przypadku prostej listy.

Zastanówmy się, co takie ustrukturyzowanie danych naprawdę oznacza. Na początku tablice były wszystkim, czym mogłeś dysponować. Tablice nie dawały żadnych możliwości realizowania jakiejkolwiek innej struktury danych niż lista sekwencyjna. W listach powiązanych użyto wskaźników w celu stopniowego rozbudowywania listy sekwencyjnej, jednak nie skorzystano z elastyczności, jaką dają wskaźniki, aby skonstruować bardziej wyrafinowane struktury.

Co rozumiem przez wyrafinowaną strukturę pamięci? Przede wszystkim można utworzyć strukturę przechowującą więcej niż tylko jeden węzeł. Dlaczego mielibyśmy to robić? Jeśli masz dwa węzły typu „następny”, jeden z nich może reprezentować elementy mniejsze od bieżącego elementu, natomiast drugi elementy większe. Taki rodzaj struktury jest nazywany **drzewem binarnym**. Drzewo binarne zostało tak nazwane, ponieważ przy każdym jego węźle istnieją co najmniej dwie gałęzie. Węzły typu „następny” nazywane są **synami**, natomiast węzeł połączony z synem nosi nazwę **ojca**.

Drzewo binarne można przedstawić następująco:



Zwróć uwagę na to, że w powyższym drzewie lewy syn każdego elementu ma mniejszą wartość niż dany element, natomiast prawy syn jest od tego elementu większy. Węzeł 10 jest ojcem całego drzewa. Jego synowie 6 i 14 są ojczami swoich mniejszych drzew. Drzewa te są nazywane **poddrzewami**.

Jedna z ważnych cech drzewa binarnego polega na tym, że każdy syn danego węzła sam jest ojcem mniejszego drzewa binarnego. Właściwość ta, w połączeniu z regułą, że synowie po lewej stronie są mniejsi od bieżącego węzła, a synowie po prawej są większe, sprawia, że projektowanie algorytmów wyszukujących węzły w drzewie jest łatwe. Najpierw należy sprawdzić wartość bieżącego węzła; jeśli jest ona równa szukanej wartości, kończymy szukanie. Jeśli szukany element jest mniejszy od bieżącego węzła, idziemy w lewo; w przeciwnym razie idziemy w prawo. Algorytm ten działa, ponieważ każdy węzeł po lewej stronie drzewa jest mniejszy od bieżącego węzła, natomiast każdy węzeł z prawej strony jest większy.

Idealną sytuacją byłoby posiadanie **drzewa zrównoważonego**, co oznacza, że zarówno z prawej strony drzewa, jak i z jego lewej strony znajduje się tyle samo węzłów. W takim przypadku każde poddrzewo binarne ma wielkość równą mniej więcej połowie całego drzewa. Jeśli szukasz wartości w takim drzewie, przy każdym przejściu do węzła potomnego możesz usunąć połowę elementów. Jeżeli zatem masz drzewo 1000-elementowe, od razu możesz pozbyć się około 500 elementów. Teraz Twoje poszukiwania zostały ograniczone do drzewa 500-elementowego. Szukanie wśród tych elementów ponownie daje Ci możliwość usunięcia mniej więcej połowy z nich, czyli około 250. Odnalezienie szukanej wartości nie zabierze Ci dużo czasu, jeśli za każdym razem pozbywasz się połowy elementów. Ile razy powinieneś podzielić drzewo, aby pozostał Ci jeden element? Odpowiedzią jest  $\log_2 n$ , gdzie  $n$  to liczba elementów drzewa. Wartość ta jest mała, nawet w przypadku dużych drzew (dla drzewa o 4 miliardach elementów

wynosi ona 32, co daje prędkość szukania około sto milionów razy większą niż w przypadku listy powiązanej o tej samej liczbie elementów, w której konieczne jest sprawdzenie każdego elementu). Jeśli jednak drzewo binarne nie jest zrównoważone, możesz nie mieć możliwości przecięcia go dokładnie na pół. W najgorszym przypadku każdy węzeł będzie mieć tylko jednego syna, co znaczy, że drzewo takie jest po prostu naszą ulubioną listą powiązaną (z paroma dodatkowymi wskaźnikami), zmuszającą nas do sprawdzenia wszystkich  $n$  elementów.

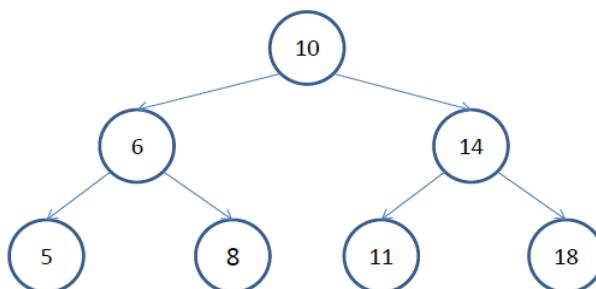
Jak widzisz, jeśli drzewo jest mniej więcej zrównoważone (ta równowaga nie musi być idealna), przeszukiwanie węzłów jest o wiele szybsze niż przeglądanie listy powiązanej. Dzieje się tak dlatego, że zamiast być skazanym na proste listy, możesz nadawać pamięci strukturę stosownie do swoich potrzeb<sup>1</sup>.

## Konwencje nazewnicze

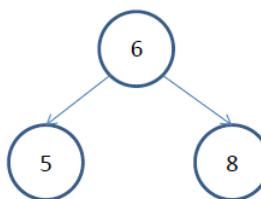
Abyśmy mogli zapoznać się z przykładowym kodem drzewa binarnego, potrzebny nam będzie wygodny sposób odnoszenia się do jego różnych części. Ustalmy więc kilka podstawowych konwencji mających zastosowanie podczas rysowania drzew binarnych i pisania o nich.

Najbardziej podstawowym rodzajem drzewa jest **drzewo puste**, reprezentowane przez wartość `NULL`. Rysując schematy drzew binarnych, nie będę pokazywał łączyc prowadzących do drzew pustych.

Kiedy będę chciał odnieść się do konkretnego drzewa, napiszę "`<drzewo o wierzchołku [wartość ojca]>`". Na przykład w przypadku poniższego drzewa:



`<drzewo o wierzchołku 6>` będzie oznaczać następujące poddrzewo:



<sup>1</sup> Podstawowe drzewo binarne, które tu omówię, może w rzadkich przypadkach przyjąć strukturę listy połączoną, na co ma wpływ kolejność dodawania poszczególnych węzłów. Istnieją bardziej wyszukane drzewa binarne, które zawsze przyjmują właściwą równowagę, ale ich omówienie wykracza poza tematykę tej książki. Jedną z takich struktur danych jest drzewo czerwono-czarne: [http://pl.wikipedia.org/wiki/Drzewo\\_czerwono-czarne](http://pl.wikipedia.org/wiki/Drzewo_czerwono-czarne).

# Implementacja drzew binarnych

Spójrzmy na kod niezbędny do prostej implementacji drzewa binarnego. Zaczniemy od zadeklarowania struktury węzła:

```
struct wezel
{
    int wartosc_klucza;
    wezel *w_lewy;
    wezel *w_prawy;
};
```

Nasza struktura `wezel` może przechowywać wartości w zmiennej całkowitej `wartosc_klucza` i zawiera dwóch synów: `w_lewy` i `w_prawy`.

Istnieje kilka często spotykanych funkcji, które zapewne chciałbyś wykonywać podczas pracy z drzewami binarnymi; główne z nich to wstawianie węzła do drzewa, szukanie wartości w drzewie, usuwanie węzła z drzewa i niszczenie drzewa w celu zwolnienia pamięci.

```
wezel* wstaw (wezel* w_drzewo, int klucz);
wezel *szukaj (wezel* w_drzewo, int klucz);
void zniszcz_drzewo (wezel* w_drzewo);
wezel *usun (wezel* w_drzewo, int klucz);
```

## Wstawianie węzła do drzewa

Zaczniemy od wstawiania węzłów do drzewa za pomocą algorytmu rekurencyjnego. W przypadku drzew binarnych rekurencja naprawdę może pokazać, co potrafi, ponieważ każde drzewo zawiera dwa mniejsze drzewa, tak więc cała struktura jest ze swojej natury rekurencyjna (co nie miałoby miejsca, gdyby drzewo binarne, zamiast kolejnych drzew, zawierało na przykład tablicę albo wskaźnik do listy powiązanej).

Nasza funkcja będzie pobierać klucz oraz istniejące drzewo (być może puste) i zwracać nowe drzewo zawierające wstawioną wartość.

```
wezel* wstaw (wezel *w_drzewo, int klucz)
{
    // Przypadek bazowy — mamy już puste drzewo
    // i musimy wstawić tu nasz nowy węzeł.
    if ( w_drzewo == NULL )
    {
        wezel* w_nowe_drzewo = new wezel;
        w_nowe_drzewo->w_lewy = NULL;
        w_nowe_drzewo->w_prawy = NULL;
        w_nowe_drzewo->wartosc_klucza = klucz;
        return w_nowe_drzewo;
    }
    // W zależności od wartości węzła zdecyduj, czy wstawić
    // do lewego, czy do prawego poddrzewa.
    if( klucz < w_drzewo->wartosc_klucza )
    {
        // Zbuduj drzewo bazujące na w_drzewo->lewy przez dodanie klucza.
        // Następnie zastąp istniejący wskaźnik w_drzewo->lewy wskaźnikiem
        // do nowego drzewa. Musimy skonfigurować wskaźnik w_drzewo->w_lewy
        // na wypadek, gdyby w_drzewo->lewy miał wartość NULL (jeśli nie
        // jest NULL, w_drzewo->w_lewy nie zmieni się, ale taka
        // konfiguracja i tak nie zaszkodzi).
    }
}
```

```

        w_drzewo->w_lewy = wstaw( w_drzewo->w_lewy, klucz );
    }
else
{
    // Wstawianie z prawej strony jest symetryczne względem
    // wstawiania z lewej strony.
    w_drzewo->w_prawy = wstaw( w_drzewo->w_prawy, klucz );
}
return w_drzewo;
}

```

Podstawowa logika naszego algorytmu jest następująca: jeśli mamy puste drzewo, tworzymy nowe. W przeciwnym razie, jeśli wstawiana wartość jest większa od wartości bieżącego węzła, wstawiamy ją w lewe poddrzewo i zastępujemy lewe poddrzewo nowo utworzonym poddrzewem. W przeciwnym razie wstawiamy ją w prawe poddrzewo i dokonujemy analogicznego zastąpienia.

Zobaczmy nasz kod w akcji — jak z pustego drzewa tworzy się drzewo z kilkoma węzłami. Jeśli do pustego drzewa (NULL) wstawimy wartość 10, otrzymamy na przypadek bazowy. W rezultacie powstanie bardzo proste drzewo:



W drzewie tym oba drzewa potomne wskazują NULL.

Jeśli teraz wstawimy do drzewa wartość 5, wykonamy następujące wywołanie:

```
wstaw( <drzewo o wierzchołku 10>, 5 )
```

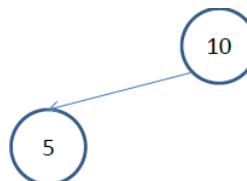
Ponieważ 5 jest mniejsze niż 10, dostaniemy rekurencyjne wywołanie w lewym poddrzewie:

```
wstaw( NULL, 5 )
wstaw( <drzewo o wierzchołku 10>, 5 )
```

Wywołanie `wstaw( NULL, 5 )` utworzy nowe drzewo i je zwróci:



Po otrzymaniu zwróconego drzewa wywołanie `wstaw( <drzewo o wierzchołku 10>, 5 )` połączy oba drzewa. W tym przypadku lewy syn wierzchołka 10 miał wartość NULL, w związku z czym stanie się on zupełnie nowym drzewem:



Jeśli teraz do drzewa dodamy liczbę 7, uzyskamy następujące wywołania:

```
wstaw( NULL, 7 )
wstaw( <drzewo o wierzchołku 5>, 7 )
wstaw( <drzewo o wierzchołku 10>, 7 )
```

Najpierw wywołanie

```
wstaw( NULL, 7 )
```

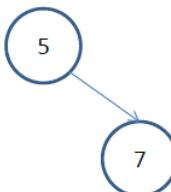
zwróci nowe drzewo:



Następnie wywołanie

```
wstaw( <drzewo o wierzchołku 5>, 7 )
```

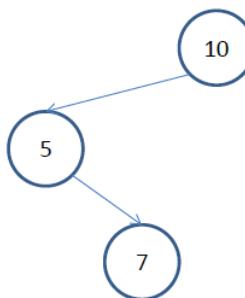
dołączy poddrzewo 7 w następujący sposób:



I ostatecznie drzewo to zostanie przekazane w wywołaniu

```
wstaw( <drzewo o wierzchołku 10>, 7 )
```

które je dołączy do wierzchołka 10.



Ponieważ 10 ma już wskaźnik do węzła zawierającego 5, ponowne dołączanie lewego syna 10 do drzewa z rodzicem 5 nie jest konieczne, ale dzięki temu pozbywamy się dodatkowej instrukcji warunkowej sprawdzającej, czy dołączane poddrzewo jest puste.

## Przeszukiwanie drzewa

Zobaczmy teraz, jak można zaimplementować przeszukiwanie drzewa. Podstawowa logika będzie prawie taka sama jak w przypadku wstawiania węzłów do drzewa — najpierw sprawdzamy oba przypadki bazowe (znaleźliśmy węzeł albo mamy do czynienia z pustym drzewem), po czym — jeśli nie natrafiliśmy na przypadek bazowy — określamy, które poddrzewo przeszukać.

```
wezel *szukaj (wezel *w_drzewo, int klucz)
{
    // Jeśli dotrzymy do pustego drzewa, szukanej wartości na pewno tu nie ma!
    if ( w_drzewo == NULL )
```

```

    {
        return NULL;
    }
    // Jeśli znajdziemy klucz, kończymy!
    else if ( klucz == w_drzewo->wartosc_klucza )
    {
        return w_drzewo;
    }
    // W przeciwnym razie próbujemy szukać w lewym albo prawym poddrzewie
    else if ( klucz < w_drzewo->wartosc_klucza )
    {
        return szukaj( w_drzewo->w_lewy, klucz );
    }
    else
    {
        return szukaj( w_drzewo->w_prawy, klucz );
    }
}

```

Pokazana powyżej funkcja szukająca dokonuje najpierw sprawdzenia dwóch warunków bazowych: czy dotarliśmy na koniec gałęzi drzewa albo czy znaleźliśmy nasz klucz. W obu przypadkach wiemy, co zwrócić — NULL, jeśli osiągnęliśmy koniec drzewa, albo drzewo, jeśli znaleźliśmy klucz.

Jeśli nie mamy do czynienia z przypadkiem bazowym, redukujemy problem do odszukania klucza w jednym z drzew potomnych — w zależności od wartości klucza lewym lub prawym. Zauważ, że za każdym razem, kiedy podczas szukania węzła przeprowadzamy wywołanie rekurencyjne, zmniejszamy rozmiar drzewa mniej więcej o połowę. Dzieje się dokładnie tak, jak to opisałem na początku tego rozdziału, gdy wykazałem, że czas potrzebny na przeszukanie zrównoważonego drzewa jest proporcjonalny do  $\log_2 n$ , co w przypadku znacznych ilości danych jest o wiele wydajniejsze niż przeszukiwanie dużych list powiązanych albo tablic.

## Niszczanie drzewa

Funkcja `zniszcz_drzewo` także powinna być rekurencyjna. Poniższy algorytm zniszczy dwa poddrzewa bieżącego węzła, po czym usunie także sam węzeł.

```

void zniszcz_drzewo (wezel* w_drzewo);
{
    if ( w_drzewo != NULL )
    {
        zniszcz_drzewo( w_drzewo->w_lewy );
        zniszcz_drzewo( w_drzewo->w_prawy );
        delete w_drzewo;
    }
}

```

Abyś lepiej mógł zrozumieć, jak działa ten proces, wyobraź sobie, że przed usunięciem węzła drukujesz jego wartość:

```

void zniszcz_drzewo (wezel *w_drzewo)
{
    if ( w_drzewo != NULL )
    {
        zniszcz_drzewo( w_drzewo->w_lewy );
        zniszcz_drzewo( w_drzewo->w_prawy );

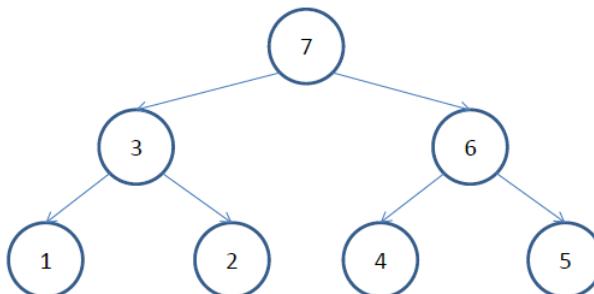
```

```

        cout << "Usuwam węzel: " << w_drzewo->wartosc_klucza;
        delete w_drzewo;
    }
}

```

Zobaczysz, że drzewo jest usuwane „z dołu do góry”. Najpierw kasowane są węzły 5. i 8., a następnie węzeł 6. W dalszej kolejności usuwana jest druga strona drzewa: węzły 11. i 18., po czym 14. Na koniec, kiedy nie ma już synów, usuwany jest węzeł 10. Wartości w drzewie nie mają znaczenia; ważne jest, gdzie znajduje się węzeł. Poniżej widać drzewo binarne, w którym zamiast wartości pokazałem, w jakiej kolejności będą kasowane poszczególne węzły:



Pomocne może okazać się ręczne przećwiczenie na kilku drzewach tego, co wykonuje kod. Dzięki temu proces ten stanie się o wiele bardziej zrozumiały.

Usuwanie elementów z drzewa stanowi przykład algorytmu rekurencyjnego, który nie byłby prosty do zaimplementowania w postaci iteracyjnej. Musiałbyś napisać pętlę, która mogłaby w jakiś sposób zajmować się jednocześnie lewą i prawą gałęzią drzewa. Problem polega na tym, że musisz mieć możliwość skasowania jednego poddrzewa, zajmując się w tym samym czasie drugim poddrzewem przeznaczonym do usunięcia, co ponadto trzeba zrealizować na wszystkich poziomach drzewa. W tym przypadku orientację w drzewie pomaga zachować stos. Można to sobie wyobrazić w taki sposób, że każda ramka stosu przechowuje informację o gałęzi drzewa, która właśnie została usunięta:

```

zniszcz_drzewo( <poddzewo> )
zniszcz_drzewo( <poddzewo> ) - wie, czy to poddrzewo znajdowało się z lewej,
→czy też z prawej strony

```

Każda ramka stosu wie, które części drzewa należy usunąć, ponieważ zna miejsce w funkcji, od którego należy kontynuować wykonywanie kodu. Podczas pierwszego wywołania funkcji `zniszcz_drzewo` ramka stosu nakazuje programowi kontynuować od drugiego wywołania tej funkcji. Podczas drugiego wywołania funkcji `zniszcz_drzewo` ramka stosu nakazuje programowi wykonanie polecenia `delete drzewo`. Ponieważ każde wywołanie funkcji ma własną ramkę stosu, pamiętany jest bieżący stan niszczenia drzewa — każdorazowo jeden jego poziom.

Jedyna możliwość nierekurencyjnej implementacji tego procesu mogłaby polegać na utworzeniu struktury danych, która przechowywałaby odpowiednik tych informacji. Moglibyśmy na przykład zasymulować stos, pisząc funkcję zapamiętującą listę powiązaną (emulującą stos) zawierającą drzewa, które są akurat niszczone. Taka lista mogłaby przechowywać informację o stronach drzewa, które należy jeszcze usunąć. Mając taką strukturę, moglibyśmy napisać bazujący na pętli algorytm dodający do listy poddrzewa i usuwający je z niej, gdy zostaną już one w całości skasowane. Innymi słowy, rekurencja pomaga nam skorzystać z przewagi, jaką daje wbudowana stosowa struktura danych, bez konieczności pisania własnej struktury. W ramach

ćwiczenia mógłbyś spróbować zaimplementować nierekurencyjną odmianę funkcji `zniszcz_drzewo`. Przekonasz się, o ile łatwiej jest wyrazić ten algorytm bez tworzenia własnego stosu, a przy okazji zyskasz lepsze zrozumienie rekurencji.

## Usuwanie węzła z drzewa

Algorytm usuwający węzły z drzewa binarnego jest bardziej złożony. Podstawowa struktura jest podobna do wzorca, który już widzieliśmy: jeśli mamy puste drzewo, kończymy; jeśli kasowana wartość znajduje się w lewym poddrzewie, usuwamy ją z lewego poddrzewa; jeśli kasowana wartość znajduje się w prawym poddrzewie, usuwamy ją z prawego poddrzewa. Jeżeli odnaleźliśmy wartość, usuwamy ją.

```
wezel* usun (wezel* w_drzewo, int klucz)
{
    if ( w_drzewo == NULL )
    {
        return NULL;
    }
    if ( w_drzewo->wartosc_klucza == klucz )
    {
        // Co zrobić?
    }
    else if ( klucz < w_drzewo->wartosc_klucza )
    {
        w_drzewo->lewy = usun( w_drzewo->lewy, klucz );
    }
    else
    {
        w_drzewo->prawy = usun( w_drzewo->prawy, klucz );
    }
    return w_drzewo;
}
```

W jednym z przypadków bazowych napotkamy jednak na problem. Co właściwie powinniśmy zrobić, kiedy znajdziemy kasowaną wartość? Pamiętaj, że drzewo binarne musi zachować następującą właściwość:

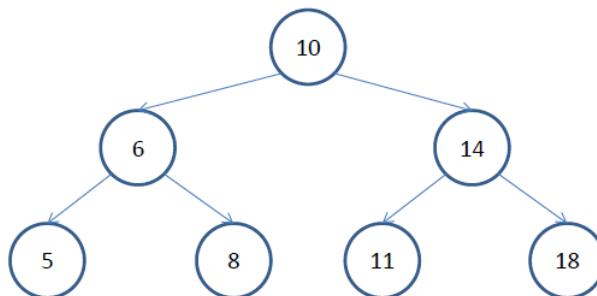
*Każda wartość w drzewie na lewo od bieżącego węzła musi być mniejsza od swojej wartości klucza; każda wartość w drzewie na prawo do bieżącego węzła musi być większa od swojej wartości klucza.*

Należy rozpatrzyć trzy podstawowe przypadki:

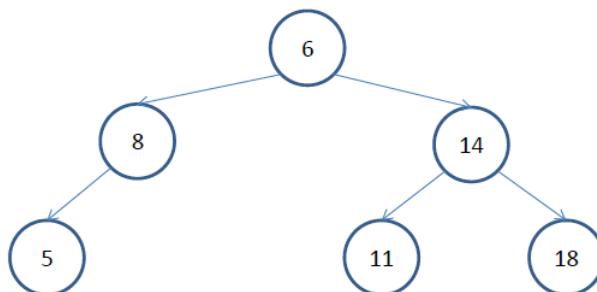
1. Usuwany węzeł nie ma synów.
2. Usuwany węzeł ma jednego syna.
3. Usuwany węzeł ma dwóch synów.

Przypadek nr 1 jest najprostszy; jeśli usuwamy węzeł bez syna, wszystko, co powinniśmy zrobić, to zwrócić NULL. Przypadek nr 2 także jest łatwy — jeżeli istnieje tylko jeden syn, zwracamy tego właśnie syna. Przypadek nr 3 jest jednak trudniejszy.

Nie możemy wziąć po prostu jednego z synów i przenieść go w górę. Co by się na przykład stało, gdybyśmy wybrali węzeł z lewej strony usuwanego elementu? Co by się wówczas stało z elementami z prawej strony węzła? Weźmy pod uwagę przykładowe drzewo z początku tego rozdziału:

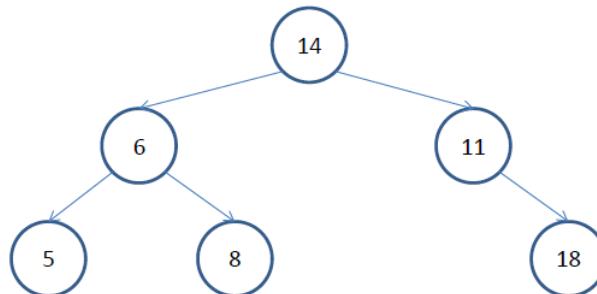


A gdybyśmy tak zechcieli usunąć element 10? Nie możemy go zastąpić elementem 6, ponieważ otrzymalibyśmy w rezultacie następujące drzewo.



8 znajduje się na lewo od 6, chociaż ósemka jest większa od szóstki. Takie rozwiązańe zaburza układ drzewa. Funkcja szukająca wartości 8 skierowałaby się na prawo od 6 i nigdy nie znalazłaby ósemki.

Z podobnej przyczyny nie można wybrać elementu z prawej strony.

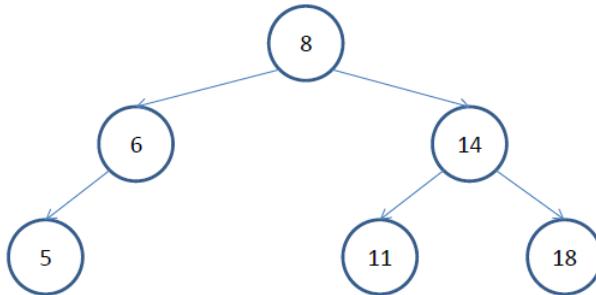


W tym przypadku 11 jest mniejsze od 14, chociaż znajduje się z prawej strony drzewa — niedobre. W drzewach binarnych nie można przemieszczać węzłów zgodnie z własnym widzimisię.

Co w takim razie zrobić? Wszystko, co znajduje się na lewo od węzła, musi mieć wartości mniejsze od wartości tego węzła. Dlaczego by nie odszukać największej wartości z lewej strony usuwanego węzła i przesunąć ją na wierzch drzewa? Ponieważ jest to największa wartość po lewej stronie drzewa, zastąpienie nią bieżącego węzła jest całkowicie bezpieczne. Mamy gwa-

rancję, że będzie ona większa od każdego innego węzła z lewej strony i ponieważ znajduje się po tej właśnie stronie, wiemy, że jest także mniejsza od każdego z węzłów umieszczonych po prawej stronie<sup>2</sup>.

W naszym przykładzie chcemy uzyskać następujące drzewo, ponieważ 8 jest największą wartością, jaka znajduje się na lewo od 10:



W tym celu potrzebny nam będzie algorytm, który potrafi odszukać największą wartość zapisaną z lewej strony drzewa — potrzebujemy funkcji `znajdz_max`. Funkcję taką możemy zaimplementować, wykorzystując prawidłowość, wedle której większe wartości zawsze znajdują się w prawym poddrzewie. Wystarczy zatem podążać wzduż prawej gałęzi drzewa, aż dotrzymy do wartości `NULL`. Innymi słowy, na potrzeby funkcji `znajdz_max`, która pobiera drzewo i zwraca największą znalezioną w nim wartość, będziemy traktować drzewo, jakby to była lista powiązana składająca się z prawych wskaźników tego drzewa:

```

wezel* znajdz_max (wezel* w_drzewo)
{
    if ( w_drzewo == NULL )
    {
        return NULL;
    }
    if ( w_drzewo->w_prawy == NULL )
    {
        return w_drzewo;
    }
    return znajdz_max( w_drzewo->w_prawy );
}
  
```

Zwróć uwagę, że potrzebne są nam dwa przypadki bazowe — pierwszy, gdy w ogóle nie mamy drzewa, i drugi, gdy docieramy do końca listy drzew potomnych po prawej stronie<sup>3</sup>. W celu zwrócenia wskaźnika do ostatniego węzła powinniśmy „zajrzeć” o jeden węzeł dalej, znajdujący się jednocześnie przy bieżącym węźle.

<sup>2</sup> Korzystając z takiej samej logiki, możemy wybrać węzeł o najmniejszej wartości z prawej strony drzewa. W praktyce dobry algorytm nie powinien stale wybierać tylko jednego bądź drugiego kierunku, aby drzewo nie utraciło równowagi. My jednak zaimplementujemy prostszą wersję algorytmu, w której nie będziemy zajmować się wyborem kierunku.

<sup>3</sup> Sposób, w jaki zaimplementujemy usuwanie, sprawi, że pierwszy przypadek bazowy (sprawdzenie, czy drzewo jest puste) będzie niepotrzebny, niemniej zgodnie z dobrym stylem programowania powinniśmy zabezpieczyć się przed nieprawidłowymi danymi wejściowymi.

Przekonajmy się, czy możemy skorzystać z tych założeń w celu napisania funkcji usuwającej węzły. W przypadku bazowym, jeśli funkcja znajdz\_max zwróci wartość NULL, będzie to znaczyć, że usuwany węzeł można zastąpić lewym drzewem, ponieważ nie ma już wartości większej od tego węzła. W przeciwnym razie węzeł należy zastąpić wynikiem zwróconym przez funkcję znajdz\_max.

```
wezel* usun (wezel* w_drzewo, int klucz)
{
    if ( w_drzewo == NULL )
    {
        return NULL;
    }
    if ( w_drzewo->wartosc_klucza == klucz )
    {
        // Dwa pierwsze przypadki obsługują węzły bez synów
        // lub z jednym synem.
        if ( w_drzewo->w_lewy == NULL )
        {
            wezel* w_prawe_poddruzwo = w_drzewo->w_prawy;
            delete w_drzewo;
            // Jeśli nie ma węzłów potomnych, może tu zostać zwrócona
            // wartość NULL, ale przecież o to nam chodzi.
            return w_prawe_poddruzwo;
        }
        if ( w_drzewo->w_prawy == NULL )
        {
            wezel* w_lewe_poddruzwo = w_drzewo->w_lewy;
            delete w_drzewo;
            // W tym miejscu zawsze zostanie zwrócony właściwy węzeł,
            // ponieważ na podstawie poprzedniego warunku wiemy,
            // że nie ma on wartości NULL.
            return w_lewe_poddruzwo;
        }
        wezel* w_max_wezel = znajdz_max( w_drzewo->w_lewy );
        w_max_wezel->w_lewy = w_drzewo->w_lewy;
        w_max_wezel->w_prawy = w_drzewo->w_prawy;
        delete w_drzewo;
        return w_max_wezel;
    }
    else if ( klucz < w_drzewo->wartosc_klucza )
    {
        w_drzewo->w_lewy = usun( w_drzewo->w_lewy, klucz );
    }
    else
    {
        w_drzewo->w_prawy = usun( w_drzewo->w_prawy, klucz );
    }
    return w_drzewo;
}
```

Jednak czy takie rozwiązanie zadziała? Wkradł się w nie pewien subtelny błąd. Otóż zapomnieliśmy o usunięciu węzła max\_wezel z miejsca, które zajmuje on w drzewie! Oznacza to, że gdzieś w drzewie istnieje wskaźnik do max\_wezel, który wskazuje w góre drzewa. Co więcej, synowie węzła max\_wezel nie są już dostępni.

Musimy usunąć z drzewa węzeł `max_wezel`. Na szczęście wiemy, że `max_wezel` nie ma prawego poddrzewa, a tylko lewe, co znaczy, że ma on co najwyżej jednego syna<sup>4</sup>. Dzięki temu mamy do czynienia z przypadkiem, który jest prosty do obsłużenia. Potrzebujemy tylko zmodyfikować ojca `max_wezel` w taki sposób, aby wskazywał lewe poddrzewo tego węzła.

Możemy napisać prostą funkcję, która po otrzymaniu wskaźnika do `max_wezel` oraz wierzchołka drzewa zawierającego `max_wezel` zwraca nowe drzewo z poprawnie usuniętym maksymalnym węzłem. Zauważ, że takie rozwiązanie bazuje na założeniu, że węzeł `max_wezel` nie ma prawego poddrzewa!

```
wezel* usun_max_wezel (wezel* w_drzewo, wezel* w_max_wezel)
{
    // Kod na wszelki wypadek — warunek ten nigdy nie powinien się spełnić.
    if ( w_drzewo == NULL )
    {
        return NULL;
    }
    // Znaleźliśmy węzeł i możemy go zastąpić.
    if ( w_drzewo == w_max_wezel )
    {
        // Robimy to tylko dlatego, że wiemy, że
        // w_max_wezel->w_prawy ma wartość NULL, tak więc nie tracimy
        // żadnej informacji. Jeśli w_max_wezel nie ma lewego poddrzewa,
        // zwróciśmy po prostu z tej gałęzi wartość NULL, co spowoduje,
        // że węzeł w_max_wezel zostanie zastąpiony pustym drzewem,
        // a właśnie to chcemy osiągnąć.
        return w_max_wezel->w_lewy;
    }
    // Każde wywołanie rekurencyjne następuje prawe poddrzewo nowym
    // poddrzewem, które nie zawiera węzła w_max_wezel.
    w_drzewo->w_prawy = usun_max_wezel( w_drzewo->w_prawy, w_max_wezel );
    return w_drzewo;
}
```

Mając taką funkcję pomocniczą, z łatwością możemy zmodyfikować funkcję `usun` w taki sposób, że przed zastąpieniem usuwanego węzła nowym węzłem maksymalnym możemy usunąć go z lewego poddrzewa.

```
wezel* usun (wezel* w_drzewo, int klucz)
{
    if ( w_drzewo == NULL )
    {
        return NULL;
    }
    if ( w_drzewo->wartosc_klucza == klucz )
    {
        // Dwa pierwsze przypadki obsługują węzły bez synów
        // lub z jednym synem.
        if ( w_drzewo->w_lewy == NULL )
        {
            wezel* w_prawe_poddzewo = w_drzewo->w_prawy;
            delete w_drzewo;
```

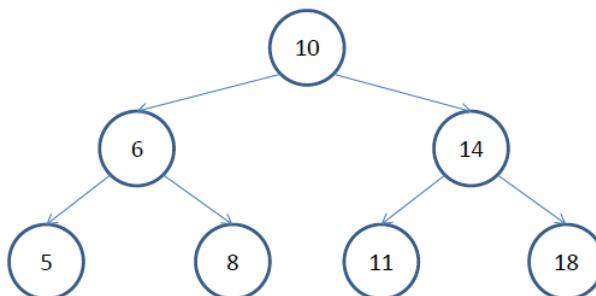
<sup>4</sup> Wiemy o tym, ponieważ jest to maksymalna wartość poddrzewa, a zatem z prawej strony nie może znajdować się węzeł.

```

// Jeśli nie ma węzłów potomnych, może tu zostać zwrocona
// wartość NULL, ale przecież o to nam chodzi.
return w_prawe_poddrzewo;
}
if ( w_drzewo->w_prawy == NULL )
{
    wezel* w_lewe_poddrzewo = w_drzewo->w_lewy;
    delete w_drzewo;
    // W tym miejscu zawsze zostanie zwrocony właściwy węzeł,
    // ponieważ na podstawie poprzedniego warunku wiemy,
    // że nie ma on wartości NULL.
    return w_lewe_poddrzewo;
}
wezel* w_max_wezel = find_max( w_drzewo->w_lewy );
// Ponieważ węzeł w_max_wezel pochodzi z lewego poddrzewa,
// musimy go stamtąd usunąć, zanim z powrotem połączymy
// to drzewo z pozostałą resztą drzewa.
w_max_wezel->w_lewy =
    usun_max_wezel( w_drzewo->w_lewy, w_max_wezel );
w_max_wezel->w_prawy = w_drzewo->w_prawy;
delete w_drzewo;
return w_max_wezel;
}
else if ( klucz < w_drzewo->wartosc_klucza )
{
    w_drzewo->w_lewy = usun( w_drzewo->w_lewy, klucz );
}
else
{
    w_drzewo->w_prawy = usun( w_drzewo->w_prawy, klucz );
}
return w_drzewo;
}

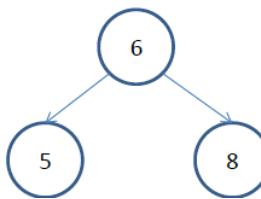
```

Zobaczmy teraz, co ten kod zrobi ze znany już nam z wcześniejszych przykładów drzewem:

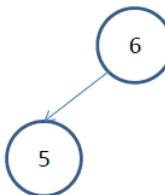


Jeśli zechcemy usunąć z drzewa węzeł 10., funkcja usun od razu natrafi na przypadek „znaleziono” — stwierdzi, że węzeł ma zarówno lewe, jak i prawe poddrzewo, w związku z czym w poddrzewie o wierzchołku 6. odszuka węzeł o maksymalnej wartości. Tym węzłem jest 8. Lewe poddrzewo tego węzła zostanie umieszczone w taki sposób, aby wskazywało nowe poddrzewo o wierzchołku 6., które jednak nie będzie już zawierać węzła 8.

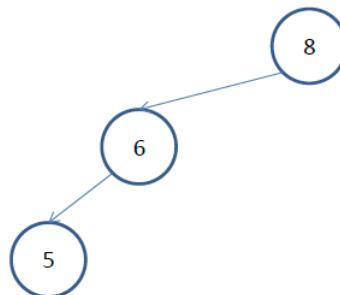
Usunięcie z poddrzewa węzła 8. jest proste. Zaczynamy od następującego poddrzewa:



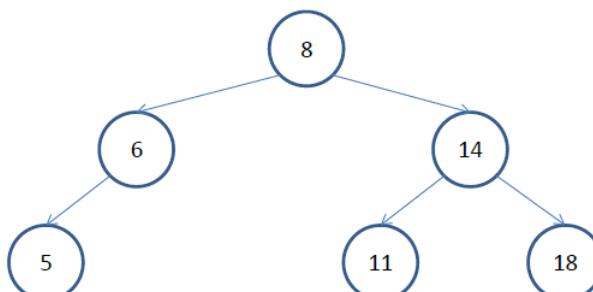
Przy pierwszym wywołaniu funkcja `usun_max_wezel` stwierdza, że 6 nie jest właściwym węzłem do usunięcia, tak więc wywołuje się rekurencyjnie z drzewem o wierzchołku 8. Ponieważ 8 jest szukanym węzłem, zostanie zwrócone jego lewe poddrzewo (czyli `NULL`), a prawy wskaźnik węzła 6. także otrzyma wartość `NULL`. Teraz mamy następujące drzewo:



W wywołaniu funkcji `usun` przekazane zostanie teraz drzewo zwrócone przez funkcję `usun_max_wezel` (pokazane powyżej), na które wskazuje lewy wskaźnik węzła 8. Nasze nowe drzewo wygląda zatem następująco:



Na koniec prawy wskaźnik węzła 8. jest kierowany na prawe poddrzewo o wierzchołku 14.; nasze drzewo zostaje tym samym odtworzone:



Teraz możemy zwolnić z pamięci węzeł 10.

Cały kod źródłowy zamieszczony w tym rozdziale, łącznie z prostym programem umożliwiającym manipulowanie drzewem, znajdziesz w pliku *drzewo\_binarne.cpp*.

## Praktyczne zastosowanie drzew binarnych

Chociaż bardzo dużo pisałem o potrzebach szybkiego wyszukiwania informacji, możesz zapytać, czy szybkie szukanie określonych wartości w strukturach danych naprawdę ma jakieś znaczenie. Czy komputery nie są wystarczająco szybkie? W jakich sytuacjach potrzebne są tego rodzaju wyszukiwania?

Szukanie jest zazwyczaj ważne w dwóch sytuacjach. Pierwszą z nich jest sprawdzanie, czy dysponujesz już określona wartością. Jeśli na przykład masz do czynienia z grą pozwalającą użytkownikom zarejestrować się, chciałbyś dysponować możliwością sprawdzenia, czy dana nazwa gracza została już wykorzystana. Jeśli pracujesz nad taką grą jak *World of Warcraft*, chciałbyś mieć możliwość bardzo szybkiego przeprowadzenia takiego testu w odniesieniu do milionów użytkowników. Ponieważ nazwy użytkowników składają się raczej z łańcuchów tekstowych niż liczb, ich sprawdzenie zabierze więcej czasu, ponieważ porównana musi zostać każda pojedyncza litera. Taki proces nie potrwa długo, jeśli przeprowadzisz go tylko kilka razy, ale stanie się wystarczająco wolny, gdy będzie się składać z milionów porównań. Użycie drzewa binarnego do przechowywania nazw użytkowników sprawi, że logowanie się użytkowników będzie o wiele wygodniejsze. Jeśli będziesz zachęcał użytkowników, aby grali online w Twoją grę, z pewnością zechcesz, aby logowanie się do gry było łatwe.

Inna często spotykana sytuacja ma miejsce wtedy, gdy potrzebne jest odszukiwanie dodatkowych informacji skojarzonych z przechowywanymi wartościami. Taka struktura danych nosi nazwę **mapy**. Mapa przechowuje klucz oraz wartość z nim skojarzoną (wartość ta nie musi być pojedynczą informacją; jeśli zachodzi potrzeba przechowywania wielu danych, może to być struktura, lista, a nawet inna mapa).

Weźmy na przykład taką grę jak *World of Warcraft*. Każda masowa gra wieloosobowa będzie wymagać w celu obsługi logowania mapy kojarzącej nazwę użytkownika z jego hasłem<sup>5</sup> oraz być może statystykami jego postaci. Za każdym razem, kiedy gracz loguje się, podając swoją nazwę oraz hasło, *World of Warcraft* odszukuje jego imię w mapie, znajduje odpowiadające mu hasło, porównuje je z wpisany hasłem i jeśli dane logowania są poprawne, pobiera pozostałe informacje o postaci i pozwala użytkownikowi dołączyć do gry.

Mapę taką mógłbyś zaimplementować przy pomocy drzewa binarnego. W takim rozwiązaniu drzewo binarne korzystałoby z klucza w celu wstawiania węzłów (w tym przypadku byłby to nazwy użytkowników) i zapisywania informacji (hasła) w tym samym węźle, tuż obok klucza.

Koncepcja mapy pojawia się bardzo często. Na przykład w dużej skali firmy obsługujące karty kredytowe także chcą korzystać z jakiegoś rodzaju map. Za każdym razem, kiedy kupujesz coś przy pomocy karty kredytowej, trzeba zmienić jakieś dane dotyczące Twojego konta. Kartami

<sup>5</sup> W rzeczywistości hasła jako takie nie będą przechowywane w mapie; zapisane tam zostaną ich wersje „wymieszczone”. **Algorytm mieszający** (haszujący) przekształca łańcuch tekstowy na inny łańcuch (albo liczbę) w taki sposób, który uniemożliwia odzyskanie oryginalnej wersji hasła. W tym przypadku wymieszana wersja hasła nie pozwoli na odczytanie hasła rzeczywistego. Przechowywanie haseł w takiej postaci zapobiega ich kradzieży na skutek podejrzenia pliku lub bazy danych, w której hasła są zapisane. Hasła są mieszcane przy użyciu algorytmów gwarantujących niskie prawdopodobieństwo uzyskania tego samego łańcucha znaków na podstawie dwóch różnych haseł.

kredytowymi dysponują setki milionów osób; przeszukiwanie takiej liczby kart przy okazji każdej transakcji wstrzymałoby handel na całym świecie. Podstawowy zamysł jest więc taki, że musi istnieć możliwość szybkiego ustalenia salda konta dla danego numeru karty kredytowej. Aby to zrealizować, mógłbyś użyć drzewa binarnego w celu utworzenia mapy kojarzącej wszystkie numery kart kredytowych ze stanami kont przypisanych do tych numerów. Teraz każda transakcja kartą kredytową będzie się sprowadzać do prostego odszukania węzła w drzewie binarnym i aktualizowania salda zapisanego w tym węźle.

Jeśli dysponujesz milionem numerów kart kredytowych zapisanych w zrównoważonym drzewie binarnym, takie szukanie będzie wymagać sprawdzenia przeciętnie  $\log_2 1000000$  węzłów, co daje ich około 20. Jest to wynik 50000 razy lepszy w porównaniu z liniowym przeszukiwaniem listy węzłów. Nie ma wątpliwości co do tego, że firmy obsługujące płatności kartami kredytowymi korzystają z o wiele bardziej wyszukanych struktur danych niż drzewa binarne. Przede wszystkim informacje na temat kont muszą być trwałe przechowywane w bazie danych zamiast tymczasowo w pamięci komputera. Struktury danych także mogą być bardziej złożone i skomplikowane niż proste mapy. Najważniejsza jest jednak idea, że drzewa binarne oraz mapy mogą być cegiełkami, z których można konstruować te bardziej wyszukane struktury.

Możliwość szybkiego wyszukiwania ma znaczenie nawet w małej skali. Na przykład Twój telefon komórkowy ma prawdopodobnie funkcję wyświetlaczą na podstawie książki adresowej imię osoby, która do Ciebie dzwoni. To kolejny przykład sytuacji, w której chcesz mieć możliwość szybkiego odszukiwania nazw na podstawie numerów (w tym przypadku numeru telefonu). Nie wiem, jak faktycznie zaimplementowana jest ta funkcja w telefonach — książka adresowa może nie być wystarczająco duża, aby odnieść korzyści z zastosowania drzewa binarnego — ale jest to jeszcze jeden przypadek, w którym mógłbyś wykorzystać koncepcję map ze względu na ich możliwości organizowania informacji. Mapy bardzo często są wbudowywane w strukturę drzew binarnych, co umożliwia przeprowadzanie szybkich poszukiwań<sup>6</sup>.

## Koszt tworzenia drzew i map

Utworzenie mapy przy wykorzystaniu drzewa binarnego zabierze nieco czasu. Wszystkie węzły należy wstawić w drzewo, co dla każdego węzła będzie wymagać przeciętnie  $\log_2 n$  operacji (tyle samo co w przypadku poszukiwania węzła, ponieważ zarówno dodawanie, jak i szukanie węzłów wiąże się z obcinaniem za każdym razem drzewa o połowę). Oznacza to, że zbudowanie całego drzewa może wymagać przeprowadzenia  $n \log_2 n$  operacji. Ponieważ każde liniowe przeszukanie listy powiązanej wymaga sprawdzenia przeciętnie  $\frac{n}{2}$  węzłów, jeśli przeprowadzasz  $2 \log_2 n$  poszukiwań na liście powiązanej, wówczas poświętasz na nie mniej więcej tyle samo czasu, ile potrwa zbudowanie drzewa binarnego. (Dlaczego? Ponieważ łączny czas jest średnim czasem wszystkich poszukiwań pomnożonych przez ich liczbę:  $\frac{n}{2} \times 2 \log_2 n = n \log_2 n$ ). Innymi

słowy, lepiej nie konstruować drzewa binarnego, jeśli masz zamiar skorzystać z niego tylko raz, ale jeżeli wiesz, że będzie ono używane wielokrotnie, zrób to (nawet mapa z milionem węzłów wymaga zaledwie około 40 sprawdzeń, aby zwiększyła się jej ogólna wydajność). W przypadku firm obsługujących karty kredytowe jest to czysty zysk. Jeśli natomiast chodzi

<sup>6</sup> Istnieją także inne struktury danych, w tym tablice mieszające ([http://pl.wikipedia.org/wiki/Tablica\\_mieszająca](http://pl.wikipedia.org/wiki/Tablica_mieszająca)), przy pomocy których również można implementować mapy.

o telefony komórkowe, opłacalność stosowania map zależy od liczby otrzymywanych połączeń telefonicznych oraz wielkości książki adresowej (spróbuj przeprowadzić odpowiednie obliczenia, aby przekonać się, czy takie rozwiązanie będzie opłacalne w przypadku telefonu).

## Sprawdź się

- 1.** Jaka jest podstawowa zaleta drzew binarnych?
  - A. Korzystają ze wskaźników.
  - B. Mogą przechowywać dowolne ilości danych.
  - C. Umożliwiają szybkie odszukiwanie informacji.
  - D. Usuwanie z nich danych jest łatwe.
- 2.** Kiedy użyjesz raczej listy powiązanej niż drzewa binarnego?
  - A. Kiedy musisz przechowywać dane w taki sposób, który umożliwia ich szybkie wyszukiwanie.
  - B. Kiedy musisz mieć możliwość odczytywania danych w kolejności posortowanej.
  - C. Kiedy musisz mieć możliwość szybkiego dodawania danych na początku lub końcu, ale nigdy w środku.
  - D. Kiedy nie musisz zwalniać pamięci, z której korzystasz.
- 3.** Które z poniższych stwierdzeń jest prawdziwe?
  - A. Kolejność, w jakiej dodajesz dane do drzewa binarnego, może zmienić jego strukturę.
  - B. Aby zagwarantować najlepszą strukturę drzewa binarnego, należy wstawać w nie elementy w kolejności posortowanej.
  - C. Szukanie elementów w liście powiązanej będzie szybsze niż w przypadku drzewa binarnego, jeśli elementy drzewa binarnego są powstawane w dowolnej kolejności.
  - D. Drzewa binarne nigdy nie da się przekształcić w taki sposób, aby miało strukturę listy powiązanej.
- 4.** Które z poniższych zdań wyjaśnia, dlaczego szukanie węzłów w drzewie binarnym przebiega szybko?
  - A. Wcale tak nie jest. Dwa wskaźniki oznaczają, że przeglądanie drzewa wymaga więcej pracy.
  - B. Przejście w dół o każdy poziom drzewa powoduje, że liczba węzłów, które pozostały do sprawdzenia, jest redukowana mniej więcej o połowę.
  - C. Drzewa binarne tak naprawdę wcale nie są lepsze od list powiązanych.
  - D. Wywołania rekurencyjne w drzewach binarnych są szybsze niż pętle w listach powiązanych.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Napisz program wyświetlający zawartość drzewa binarnego. Czy potrafisz napisać program, który pokażę węzły w kolejności posortowanej? A co z kolejnością malejącą?
- 2.** Napisz program pokazujący liczbę węzłów w drzewie binarnym.
- 3.** Napisz program, który sprawdza, czy drzewo binarne jest poprawnie zrównoważone.

- 4.** Napisz program, który sprawdza, czy drzewo binarne jest prawidłowo posortowane, tj. czy wszystkie węzły leżące na lewo od danego węzła mają mniejszą wartość niż ten węzeł oraz czy wszystkie węzły leżące od niego na prawo mają wartość większą.
- 5.** Napisz program, który bez użycia rekurencji usuwa wszystkie węzły z drzewa binarnego.
- 6.** Zaimplementuj w postaci drzewa binarnego prostą mapę, która przechowuje książkę adresową. Kluczem mapy powinno być nazwisko osoby, a wartością jej adres e-mailowy. Powinieneś udostępnić możliwość dodawania do mapy nowych adresów, ich usuwania oraz aktualizowania, a także, rzecz jasna, szukania. Kiedy program będzie kończyć działanie, książka adresowa powinna być czyszczona. Przypomnę tylko, że w celu porównywania dwóch łańcuchów tekstowych możesz korzystać ze standardowych w języku C++ operatorów porównania (takich jak ==, < albo >).



# 18

■ ■ ■ R O Z D Z I A Ł 1 8

## Standardowa biblioteka szablonów

---

Możliwość pisania własnych struktur danych to świetna rzecz, jednak nie jest to tak popularne zajęcie, jak mógłbyś sądzić na podstawie poprzedniego rozdziału. Ale bez obaw, nie nakłaniam Cię do jego przeczytania bezpodstawnie. Bardzo dużo dowiedziałeś się o tym, jak konstruować własne struktury danych, gdy ich **naprawdę** potrzebujesz, oraz poznaleś właściwości kilku często spotykanych struktur. Zdarzają się przypadki, w których implementowanie własnych struktur danych jest bardzo uzasadnione.

Obok możliwości tworzenia własnych struktur danych, jedną z najprzydatniejszych właściwości języka C++ (która nie jest dostępna w C) jest duża biblioteka kodu wielokrotnego użycia, udostępniona wraz z kompilatorem. Biblioteka ta nazywana jest **standardową biblioteką szablonów** (STL, od ang. *Standard Template Library*). STL jest zbiorem często używanych struktur danych, takich jak listy powiązane albo struktury bazujące na drzewach binarnych. Każda z tych struktur umożliwia definiowanie typów danych, które będą w nich przechowywane, dzięki czemu można w nich zapisywać, co tylko się chce — liczby całkowite,łańcuchy tekstowe albo ustrukturyzowane dane.

Ze względu na tę elastyczność w wielu przypadkach standardowa biblioteka szablonów może wykluczyć potrzebę tworzenia niestandardowych struktur danych na własne potrzeby programistyczne. W rzeczy samej STL pozwala Ci wynieść Twój kod na wyższy poziom dzięki kilku różnym cechom:

1. Możesz zacząć myśleć o swoich programach w kategoriach potrzebnych Ci struktur danych, bez konieczności zastanawiania się, czy będziesz w stanie te struktury zaimplementować we własnym zakresie.
2. Masz dostęp do światowej klasy struktur danych, których wydajność i zajmowana przez nie pamięć w zupełności wystarczy do rozwiązania większości Twoich problemów.
3. Nie musisz pamiętać o alokowaniu i zwalnianiu pamięci zajmowanej przez struktury, z których korzystasz.

Istnieje jednak kilka kompromisów, które wiążą się ze stosowaniem standardowej biblioteki szablonów:

1. Musisz poznać interfejsy standardowej biblioteki szablonów oraz nauczyć się, jak z nich korzystać.
2. Błędy komplikacji generowane na skutek nieprawidłowego użycia STL są niewiarygodnie trudne do odczytania.
3. Nie każda struktura danych, która może Ci być potrzebna, jest dostępna w STL.

Standardowa biblioteka szablonów to obszerny temat. Istnieją całe książki poświęcone wyłącznie STL<sup>1</sup>, nie mam więc szans na pełne jej tu opisanie. Celem tego rozdziału jest przedstawienie Ci w zarysie najbardziej przydatnych i najczęściej spotykanych struktur danych z STL. Od tej chwili będę z nich korzystał, gdy tylko będzie to sprawiedliwione.

## Wektor — tablica o zmiennych rozmiarach

W STL istnieje odpowiednik tablicy, którym jest **wektor**. W STL wektory bardzo przypominają tablice, z tym że można zmieniać ich rozmiary, a Ty — programista, nie musisz przejmować się szczegółami alokacji pamięci i przenoszeniem z miejsca na miejsce istniejących elementów wektora.

Składnia związana z użyciem wektora różni się jednak od składni tablicy. Oto deklaracja tablicy:

```
int jakas_tablica[ 10 ];  
i wektora:
```

```
#include <vector>  
  
using namespace std;  
vector<int> jakis_wektor( 10 );
```

Przede wszystkim zwróć uwagę na konieczność dołączenia pliku nagłówkowego `vector`, żeby w ogóle można było korzystać z wektora, a także na użycie przestrzeni nazw `std`. Jest tak dlatego, że podobnie jak `cin` i `cout`, wektor stanowi część biblioteki standardowej.

Po drugie, gdy deklarujesz wektor, musisz w nawiasach ostrokątnych podać typ danych, jakie będą w nim przechowywane:

```
vector<int>
```

W powyższej składni wykorzystano funkcjonalność języka C++ nazywaną szablonem (stąd nazwa: standardowa biblioteka **szablonów**). Typ `vector` jest zaimplementowany w taki sposób, że można w nim przechowywać dane dowolnego typu, o ile tylko kompilator zostanie poinformowany, jaki rodzaj danych zostanie umieszczony w konkretnym wektorze. Innymi słowy, tak naprawdę mamy tu do czynienia z dwoma typami: jeden to typ struktury danych, jaka będzie zarządzać organizacją danych, a drugi to typ danych znajdujących się w tej strukturze. Szablony umożliwiają łączenie różnych typów struktur danych z różnymi typami danych przechowywanych w tych strukturach.

Na koniec, podając rozmiar wektora, należy użyć nawiasów okrągłych zamiast prostokątnych:

```
vector<int> jakis_wektor( 10 );
```

Składnia taka jest używana podczas inicjalizacji określonych rodzajów zmiennych. W tym przypadku do procedury inicjalizującej (czyli konstruktora) przekazywana jest wartość 10, która ustali taką właśnie wielkość wektora. W kolejnych rozdziałach przedstawię więcej informacji o konstruktorach oraz obiektach, które z nich korzystają.

---

<sup>1</sup> Polecam książkę *STL w praktyce. 50 sposobów efektywnego wykorzystania Scotta Meyersa* (Helion, Gliwice 2004).

Po utworzeniu wektora można korzystać z jego elementów w taki sam sposób, jak w przypadku tablic:

```
for ( int i = 0; i < 10; i++ )
{
    jakis_wektor[ i ] = 0;
    jakas_tablica[ i ] = 0;
}
```

## Przekazywanie wektorów do metod

Wektory udostępniają jednak więcej możliwości niż tylko podstawowe funkcjonalności związane z tablicami. W przypadku wektora można przykładowo wstawać elementy poza jego koniec. Operacje takie są realizowane przez funkcje stanowiące część wektora. Ich składnia różni się od tego, z czym miałeś do czynienia do tej pory. Wektory korzystają z funkcjonalności C++ znanej jako **metoda**. Metoda jest funkcją deklarowaną łącznie z typem danych (w tym przypadku wektorem), a z wywołaniem metody wiąże się użycie nowej składni:

```
jakis_wektor.size();
```

Powyższy kod wywołuje metodę `size` w wektorze `jakis_wektor` i zwraca wielkość tego wektora. To trochę jak odczytywanie pola w strukturze, z tym że za pośrednictwem metody należącej do tej struktury. Chociaż metoda `size` najwyraźniej coś robi z wektorem `jakis_wektor`, nie musisz udostępniać go tej metodzie jako odrębnego argumentu! Składnia metody wie, że należy do niej przekazać `jakis_wektor` jako argument domyślny.

O następującej składni:

```
<zmienna>.<wywołanie funkcji>( <argumenty> );
```

możesz myśleć jak o wywołaniu funkcji, która należy do typu tej zmiennej. Innymi słowy, to mniej więcej tak, jakby napisać:

```
<wywołanie funkcji>( <zmienna>, <argumenty> );
```

W naszym przykładzie zapis:

```
jakis_wektor.size();
```

byłby równoważny z napisaniem:

```
size( jakis_wektor );
```

W kolejnych rozdziałach o wiele więcej miejsca poświęćę metodom, sposobom ich deklarowania i korzystania z nich. Na razie wystarczy, że będziesz wiedzieć, iż istnieje wiele metod, które można wywoływać z wektorami i że w tym celu należy użyć specjalnej składni. Taka składnia jest jedynym sposobem przeprowadzania tego typu wywołań — nie można napisać `size( jakis_wektor )`.

## Inne właściwości wektorów

Jakie więc wspaniałe możliwości zyskujemy dzięki wektorom? Ułatwiają one zwiększanie liczby przechowywanych w nich wartości bez konieczności przeprowadzania jakichkolwiek żmudnych alokacji pamięci. Jeśli na przykład chciałbyś dodać do swojego wektora więcej elementów, mógłbyś napisać:

```
jakis_wektor.push_back( 10 );
```

Taki zapis spowoduje dodanie do wektora nowego elementu. Znaczy on „dodaj element 10 na końcu bieżącego wektora”. Sam wektor zajmie się obsługą zmiany swojego rozmiaru! Aby zrobić to samo z tablicą, musiałbyś dokonać alokacji nowej tablicy, przekopiować do niej wszystkie wartości i dopiero wtedy dodać nowy element. Rzecz jasna wektory wewnętrznie alokują pamięć i kopią elementy, ale robią to w inteligentny sposób. Jeśli zatem regularnie dodajesz nowe elementy, wektor nie będzie alokować pamięci przy każdej zmianie swojego rozmiaru.

Muszę Cię jednak przed czymś ostrzec. Nawet jeśli za pomocą metody `push_back` możesz dodać nowy element na końcu wektora, nie uzyskasz tego samego efektu, korzystając z nawiasów prostokątnych. Jest to swego rodzaju osobliwość związana z implementacją tej funkcjonalności, gdyż nawiasy kwadratowe pozwalają na pracę wyłącznie z zaalokowaną pamięcią. Prawdopodobnie przyczyną przyjęcia takiego rozwiązania była chęć uniemożliwienia alokowania pamięci bez woli użytkownika kodu.

W związku z tym następujący kod:

```
vector<int> jakis_wektor( 10 );
jakis_wektor[ 10 ] = 10; // Ostatnim poprawnym elementem jest 9
```

nie zadziała. Może on spowodować awarię programu i z pewnością jest niebezpieczny w użyciu. Jeśli natomiast napiszesz:

```
vector<int> jakis_wektor( 10 );
jakis_wektor.push_back( 10 ); // Dodaje do wektora nowy element
```

wektor zostanie rozszerzony do nowego rozmiaru, który wyniesie teraz 11.

## Mapy

Pisałem już nieco na temat idei kryjącej się za mapami, która polega na pobraniu jednej wartości i skorzystaniu z niej w celu sprawdzenia innej wartości. Taka potrzeba stale pojawia się w trakcie programowania — implementacja książki adresowej, w której adresy szukane są na podstawie nazwy, szukanie informacji o koncie bankowym na podstawie jego numeru, czy też umożliwienie użytkownikom logowania się do gry.

STL udostępnia bardzo wygodny rodzaj mapy, który umożliwia określanie typów klucza i wartości. Na przykład struktura danych przechowująca prostą książkę adresową — podobną do tej, jaką być może napisałeś w ramach jednego z ćwiczeń z poprzedniego rozdziału — może zostać zaimplementowana następująco:

```
#include <map>
#include <string>

using namespace std;

map<string, string> nazwa_email;
```

W kodzie tym musimy poinformować strukturę danych przechowującą mapę o dwóch różnych typach. Pierwszy typ, `string`, dotyczy klucza, natomiast drugi typ, także `string`, określa wartość, którą w naszym przykładzie jest adres e-mail.

Jedna z interesujących funkcjonalności map w STL polega na tym, że gdy z nich korzystasz, możesz stosować taką samą składnię jak w przypadku tablic!

Dodając wartość do mapy, traktujesz ją jak tablicę, z tym że zamiast korzystać z liczby całkowitej, używasz typu klucza:

```
nazwa_email[ "Alex Allain" ] = "webmaster@cplusplus.com"
```

Odczytanie wartości z mapy przebiega niemal tak samo:

```
cout << nazwa_email[ "Alex Allain" ];
```

Co za wygoda! Prostota, jak w przypadku korzystania z map, przy jednoczesnej możliwości przechowywania dowolnych typów. Co lepsze, w przeciwieństwie do wektorów, przed użyciem operatora [] w celu dodania nowych elementów nie trzeba nawet określać rozmiaru mapy.

Usuwanie elementów z mapy również jest bardzo łatwe.

Załóżmy, że nie chcesz już do mnie wysyłać e-maili; możesz usunąć mnie ze swojej książki adresowej za pomocą metody `erase`:

```
nazwa_email.erase( "Alex Allain" );
```

Żegnaj!

Möżesz także sprawdzić rozmiar mapy przy użyciu metody `size`:

```
nazwa_email.size();
```

Z kolei za pomocą metody `empty` można sprawdzić, czy mapa jest pusta:

```
if ( nazwa_email.empty() )
{
    cout << "Twoja książka adresowa jest pusta. Może lepiej było nie kasować Aleksa?"; }
```

Metody tej nie należy mylić z metodą `clear`, która „czyści” tablicę, tj. usuwa z niej wszystkie elementy:

```
nazwa_email.clear();
```

A tak przy okazji — kontenery STL używają spójnej konwencji nazewnictwa, w związku z czym z metod `clear`, `empty` i `size` można korzystać także w odniesieniu zarówno do wektorów, jak i map.

## Iteratory

Oprócz przechowywania danych w strukturze i odczytywania z niej poszczególnych elementów, czasami chciałbyś po prostu przejrzeć po kolejnych wszystkie elementy tej struktury. Gdybyś miał do czynienia z tablicą albo wektorem, mógłbyś odczytać ich długości i dzięki temu zyskać możliwość sprawdzania poszczególnych elementów. Co jednak z mapami? Ponieważ mapy często mają klucze nienumeryczne, nie zawsze istnieje możliwość iterowania po nich przy użyciu zmiennej licznikowej.

Aby rozwiązać ten problem, w STL wprowadzono ideę zwaną iteratorem. **Iterator** to zmienna umożliwiająca realizowanie sekwencyjnego dostępu do kolejnych elementów struktury danych, nawet jeśli struktura ta nie udostępnia normalnie takiej możliwości. Zaczniemy od sprawdzenia, jak korzystać z iteratora w odniesieniu do wektorów, a następnie dowiemy się, jak stosować iteratory w celu odczytywania elementów w mapach. Podstawowa koncepcja iteratora sprawdza się do tego, że przechowuje on bieżącą pozycję w strukturze danych, udostępniając dane znajdujące się na tej pozycji. Wywołując metodę dla iteratora, można przejść do następnego elementu struktury danych.

Zadeklarowanie iteratora wymaga użycia dość niezwykłej składni. Oto, jak wygląda deklaracja iteratora dla wektora liczb całkowitych:

```
vector<int>::iterator
```

Taki zapis oznacza, że mamy do czynienia z wektorem liczb całkowitych i potrzebny nam jest iterator działający z tym typem, stąd konstrukcja `::iterator`. Jak więc skorzystać z iteratora? Ponieważ iterator przechowuje pozycję w strukturze danych, należy odwołać się do iteratora w tej strukturze:

```
vector<int> wekt;  
wekt.push_back( 1 );  
wekt.push_back( 2 );  
  
vector<int>::iterator itr = wekt.begin();
```

Wywołanie metody `begin` zwraca iterator umożliwiający uzyskanie dostępu do pierwszego elementu wektora. Możesz przyjąć, że wektor jest bardzo podobny do wskaźnika. Pozwala on określić położenie elementu w strukturze danych i można przy jego pomocy odczytać (lub zapisać) ten element. W naszym przypadku możesz odczytać pierwszy element wektora przy użyciu następującej składni:

```
cout << *itr; // Wyświetl pierwszy element wektora
```

Użyto operatora gwiazdki, zupełnie jakby to był wskaźnik. Taka składnia ma sens; iterator przechowuje lokalizację, dokładnie tak samo jak wskaźnik.

Aby przejść do następnego elementu wektora, należy zwiększyć iterator:

```
itr++;
```

Taki zapis nakazuje przejście do kolejnego elementu wektora.

Można również użyć operatora w prefiksie:

```
++itr;
```

W przypadku niektórych iteratorów takie rozwiązanie jest nieco skuteczniejsze<sup>2</sup>.

Możemy sprawdzić, czy dotarliśmy do końca iteracji, porównując nasz iterator z iteratorem `end`, co wymaga następującego zapisu:

```
wekt.end();
```

Zatem aby napisać kod przechodzący w pętli przez cały wektor, należy utworzyć następującą pętlę:

```
for ( vector<int>::iterator itr = wekt.begin(); itr != wekt.end(); ++itr )  
{  
    cout << *itr << endl;  
}
```

Kod ten oznacza: utwórz iterator `i` z wektora liczb całkowitych pobierz pierwszy element. Dopóki iterator jest różny od iteratora `end`, kontynuuj iterowanie po wektorze i wyświetlaj po kolei jego elementy.

---

<sup>2</sup> Jest tak dlatego, że operator w prefiksie (`++itr`) zwraca wartość wyrażenia po wykonaniu inkrementacji, podczas gdy użycie operatora w postfiksie (`itr++`) zwraca wcześniejszą wartość `itr`, co oznacza, że iterator musi tę wartość przechowywać. Operator w prefiksie zwraca potrzebną wartość, ponieważ w iteratorze znajduje się już wynik inkrementacji.

Istnieje kilka poprawek, które możemy wprowadzić do tej pętli. Powinniśmy unikać wywoływania metody `wekt.end` przy każdym przebiegu pętli:

```
vector<int>::iterator koniec = wekt.end();
for ( vector<int>::iterator itr = wekt.begin(); itr != koniec; ++itr )
{
    cout << *itr << endl;
}
```

W pierwszej części pętli możemy umieścić więcej zmiennych, dzięki czemu nasz kod stanie się nieco bardziej elegancki:

```
for ( vector<int>::iterator itr = wekt.begin(), koniec = wekt.end();
      itr != koniec; ++itr )
{
    cout << *itr << endl;
}
```

Z podobnego podejścia można korzystać w celu przechodzenia w pętlach przez mapy. Mapa nie przechowuje jednak tylko jednej wartości; znajdują się w niej zarówno klucz, jak i wartość. W jaki sposób uzyskać je na podstawie iteratora? Kiedy dokonujesz dereferencji iteratora, zawiera on dwa pola — `first` i `second`. Pole `first` to pole z kluczem, natomiast pole `second` to pole z wartością:

```
int klucz = itr->first; // Odczytaj z iteratora klucz
int wartosc = itr->second; // Odczytaj z iteratora wartość
```

Spójrzmy na kod, który wyświetla zawartość mapy w czytelnym formacie:

```
void wyswietlMape (map<string, string> mapa_do_wyswietlenia)
{
    for ( map<string, string>::iterator itr = mapa_do_wyswietlenia.begin(),
          koniec = mapa_do_wyswietlenia.end();
          itr != koniec;
          ++itr )
    {
        cout << itr->first << " --> " << itr->second << endl;
    }
}
```

Powyższy kod jest podobny do kodu iterującego po wektorach. Jedyna istotna różnica między tymi kodami sprowadza się do użycia struktury danych typu `mapa` i zastosowania z iteratorem pól `first` i `second`.

## Sprawdzanie, czy wartość znajduje się w mapie

Czasami podczas pracy z mapami chciałbyś mieć możliwość sprawdzenia, czy określony klucz jest już zapisany w mapie. Jeśli na przykład szukasz kogoś w książce adresowej, mógłbyś przekonać się, czy osoba ta znajduje się już w książce. Metoda `find` wywołana dla mapy jest dokładnie tym, czego potrzebujesz w celu sprawdzenia, czy dana wartość występuje w mapie, i pobrania jej, jeśli została odnaleziona. Metoda `find` zwraca iterator; będzie to iterator przechowujący lokalizację obiektu o zadanym kluczu albo iterator `end`, gdy obiekt nie został odszukany.

```
map<string, string>::iterator itr = nazwa_email.find( "Alex Allain" );
if ( itr != nazwa_email.end() )
{
    cout << "Jak miło znowu zobaczyć Aleksa! Jego email to: " << itr->second;
}
```

Z drugiej jednak strony, gdy za pomocą zwykłej składni z nawiasami prostokątnymi spróbowujesz uzyskać dostęp do elementu mapy, który nie jest obecny na liście:

```
nazwa_email[ "Jan NN" ];
```

wówczas mapa wstawi za Ciebie pusty element, jeśli podana wartość nie zostanie odnaleziona. Jeżeli chcesz jedynie wiedzieć, czy określona wartość znajduje się w mapie, użyj metody `find`. W przeciwnym razie spokojnie możesz skorzystać ze składni z nawiasami prostokątnymi.

## Oswajanie biblioteki STL

Biblioteka STL kryje w sobie znacznie więcej możliwości niż tylko te, które tu opisałem. Poznałeś ją jednak na tyle, że będziesz mógł korzystać z przewagi, jaką dają podstawowe typy STL. Typ `vector` całkowicie zastępuje tablice i można z niego korzystać zamiast z list powiązanych, kiedy czas wstawiania elementów do listy lub jej modyfikowania nie gra roli. Istnieje niewiele powodów, dla których miałbyś korzystać z tablic, gdy masz do dyspozycji wektory — w większości przypadków będą to zaawansowane zastosowania tablic, jak na przykład podczas plikowych operacji wejścia-wyjścia.

Struktura map jest chyba najprzydatniejszym typem danych w STL. Ciągle korzystam z mapopodobnych struktur, dzięki czemu pisanie wyrafinowanych programów przychodzi mi o wiele łatwiej, ponieważ nie muszę zastanawiać się, jak tworzyć różnorodne struktury danych. Zamiast tego mogę skupić się na opracowywaniu problemów, które wymagają rozwiązania. Mapy pozwalają zastępować podstawowe drzewa binarne na wiele różnych sposobów — w większości przypadków prawdopodobnie nie będziesz musiał implementować własnego drzewa binarnego, chyba że masz szczególne wymagania dotyczące wydajności albo rzeczywiście musisz użyć struktury drzewa binarnego. Taka jest prawdziwa moc biblioteki STL — w okolo 80% przypadków udostępnii Ci ona odpowiednie struktury danych, dzięki czemu będziesz mógł napisać kod rozwiążający Twoje specyficzne problemy. Pozostałe 20% przypadków tłumaczy, dlaczego powinieneś wiedzieć, jak tworzyć własne struktury danych<sup>3</sup>.

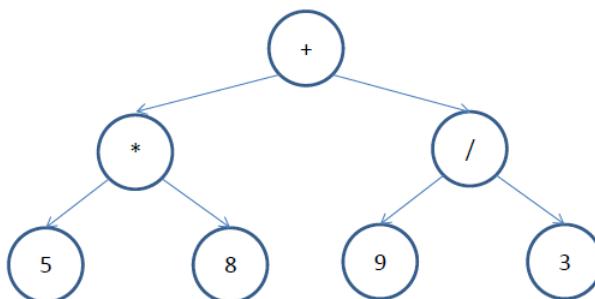
Niektórzy programiści cierpią na syndrom „wyważania otwartych drzwi”, który przejawia się pisaniem własnego kodu zamiast stosowania kodu opracowanego przez kogoś innego. W większości przypadków **nie powinieneś** implementować własnych struktur danych. Struktury wbudowane są zwykle lepsze, szybsze i pełniejsze od struktur, które mógłbyś zbudować samodzielnie. Wiedza o tym, jak je tworzyć, pozwoli Ci jednak lepiej rozumieć, jak z nich korzystać i jak tworzyć własne struktury, gdy zajdzie taka potrzeba.

Kiedy zatem możesz potrzebować własnej struktury danych? Założymy, że chcesz zbudować mały kalkulator umożliwiający użytkownikom wprowadzanie wyrażeń arytmetycznych i obliczający te wyrażenia zgodnie z właściwym priorytetem operatorów, na przykład po wczytaniu działania  $5 * 8 + 9 / 3$  mnożenie i dzielenie zostaną obliczone przed dodawaniem.

Okazuje się, że w bardzo naturalny sposób można myśleć o tego typu strukturze jak o drzewie. Oto sposób na przedstawienie w formie drzewa działania  $5 * 8 + 9 / 3$ :

---

<sup>3</sup> Informacje te nie zostały uzyskane w żaden naukowy sposób. Tak naprawdę wymyśliłem je sam. Wielkości te mogą się różnić, ale wątpię, aby któraś z nich osiągnęła 100%.



Każdy węzeł można wyznaczyć następująco:

- 1.** Jeśli węzeł zawiera liczbę, zwróć jego wartość.
- 2.** Jeśli węzeł zawiera operator, pobierz wartości obu poddrzew i wykonaj daną operację.

Konstruowanie takiego drzewa wymaga pracy z nieprzetworzoną strukturą danych — mając do dyspozycji samą tylko mapę, nie dasz rady go zbudować. Jeśli Twoim jedynym narzędziem jest STL, rozwiążanie takiego problemu będzie trudne, jeśli natomiast rozumiesz drzewa binarne i rekurencję, zadanie to stanie się o wiele prostsze.

## Więcej informacji o STL

Jeśli chciałbyś dowiedzieć się, co takiego jeszcze udostępnia STL, mogę polecić kilka dobrych źródeł informacji. SGI ma stronę internetową z obfitą dokumentacją na temat STL: <http://www.sgi.com/tech/stl/>. Kolejnym niezłym źródłem informacji jest książka Scotta Meyersa *STL w praktyce. 50 sposobów efektywnego wykorzystania*, która wyjaśnia wiele koncepcji oraz idiomów STL. Strona <http://en.cppreference.com/w/cpp> także zawiera świetną dokumentację dotyczącą wielu elementów STL i chociaż nie stanowi ona wprowadzenia do tej biblioteki, to jednak udostępnia znakomity materiał na temat standardowej biblioteki C++.

## Sprawdź się

- 1.** Kiedy wskazane jest użycie wektora?
  - A. Kiedy trzeba przechowywać związek zachodzący między kluczem a wartością.
  - B. Kiedy podczas wymieniania zbioru elementów potrzebna jest maksymalna wydajność.
  - C. Kiedy nie musisz przejmować się szczegółami aktualizowania swojej struktury danych.
  - D. Tak jak garnitur w przypadku rozmowy o pracę, wektor jest zawsze odpowiedni.
- 2.** W jaki sposób można jednocześnie usunąć wszystkie elementy z mapy?
  - A. Zamienić element na łańcuch pusty.
  - B. Wywołać metodę `erase`.
  - C. Wywołać metodę `empty`.
  - D. Wywołać metodę `clear`.
- 3.** Kiedy powinieneś implementować własne struktury danych?
  - A. Kiedy potrzebujesz czegoś naprawdę szybkiego.
  - B. Kiedy potrzebujesz czegoś solidnego.

- C. Kiedy trzeba skorzystać z przewagi, jaką dają nieprzetworzone struktury danych, na przykład w przypadku tworzenia drzewa wyrażeń arytmetycznych.
  - D. Tak naprawdę nie trzeba implementować własnych struktur danych, chyba że z jakichś powodów chcemy to zrobić.
- 4.** Który z poniższych zapisów poprawnie definiuje iterator, którego można użyć z wektorem całkowitym `vector<int>`?
- A. `iterator<int> itr;`
  - B. `vector::iterator itr;`
  - C. `vector<int>::iterator itr;`
  - D. `vector<int>::iterator<int> itr;`
- 5.** Który z poniższych zapisów udostępnia klucz elementu, przy którym w danej chwili znajduje się iterator na mapie?
- A. `itr.first`
  - B. `itr->first`
  - C. `itr->klucz`
  - D. `itr.klucz`
- 6.** Jak podczas korzystania z iteratora sprawdzisz, czy dotarłeś do ostatniej iteracji?
- A. Porównam iterator z wartością `NULL`.
  - B. Porównam iterator z wartością wywołania metody `end()` dla kontenera, po którym iteruję.
  - C. Porównam iterator z wartością `0`.
  - D. Porównam iterator z wartością wywołania metody `begin()` dla kontenera, po którym iteruję.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

1. Zaimplementuj program obsługujący niewielką książkę adresową, która umożliwia użytkownikom wprowadzanie nazw oraz adresów e-mailowych, usuwanie i zmienianie wpisów, a także wyświetlanie informacji w niej zapisanych. Nie przejmuj się brakiem możliwości zapisywania adresów na dysku. Nic się nie stanie, jeśli dane zostaną utracone po zakończeniu działania programu<sup>4</sup>.
2. Skorzystaj z wektorów w celu zaimplementowania listy zawierającej najlepsze wyniki osiągnięte w grze komputerowej. Lista powinna być aktualizowana automatycznie, a nowe wyniki wstawiane we właściwe miejsce listy. Na stronie SGI, którą polecałem wcześniej, możesz znaleźć więcej przydatnych operacji, które można wykonywać na wektorach.
3. Napisz program udostępniający dwie opcje: rejestrowanie użytkownika oraz logowanie. Funkcja rejestracji umożliwi nowemu użytkownikowi utworzenie nazwy i hasła. Funkcja logowania pozwoli z kolei przejść do nowych opcji — zmiany hasła i wylogowania się. Funkcja zmiany hasła pozwoli użytkownikowi zmienić bieżące hasło, natomiast wylogowanie powinno przenieść użytkownika do ekranu startowego.

---

<sup>4</sup> Tylko dlatego, że jesteś zarówno programistą, jak i użytkownikiem ☺.

## Więcej o łańcuchach tekstowych

---

Znakomicie! Właśnie udało nam się przebrnąć przez całą masę trudnego materiału. Możemy sobie pogratulować! Odpocznijmy trochę od poznawania nowych struktur danych i powróćmy do pracy ze strukturą, z którą mieliśmy już do czynienia — skromnym łańcuchem tekstowym. Wbrew swojej prostocie łańcuchy tekstowe są stosowane wszędzie. Wiele programów jest tworzonych niemal wyłącznie po to, aby pobierały i przetwarzaly teksty. Często będziesz wczytywał łańcuchy, żeby prezentować je użytkownikom, ale także będzie się zdarzać, że zechcesz wydobyć z nich jakąś treść. Być może w celu zaimplementowania funkcji szukającej będziesz musiał mieć możliwość odnajdowania jakiejś wartości w tekście, a może przyjdzie Ci wczytywać zbiory tabelarycznych danych rozdzielanych przecinkami, implementować listy najlepszych wyników w grach albo utworzyć interfejs w tekstowej grze przygodowej. Jedna z najpopularniejszych aplikacji, z jakich korzystasz każdego dnia, przeglądarka internetowa, jest w znacznej mierze rozbudowanym procesorem łańcuchów tekstowych, przetwarzającym strony HTML. Każdy z tych problemów wymaga przeprowadzania bardziej zaawansowanych operacji niż tylko wczytywania i wyświetlania łańcuchów tekstowych traktowanych jako całość.

Łańcuchy tekstowe mogą być dość długie i przechowywać w pamięci komputera wiele znaków, tak więc w celu tworzenia programów zachowujących maksymalną wydajność — nawet podczas przekazywania łańcuchów tekstowych do funkcji — możemy skorzystać z zalet niektórych poznanych już przez nas funkcjonalności, w szczególności z referencji. W rozdziale tym przedstawię rozmaite operacje, z których można korzystać podczas pracy z tekstami, a także wyjaśnię, jak zagwarantować odpowiednią szybkość działania programu w czasie ich przetwarzania. W zadaniach praktycznych otrzymasz szansę napisania paru ciekawych programów operujących na tekstach, dzięki którym poznasz możliwości związane z przetwarzaniem łańcuchów tekstowych.

### Wczytywanie łańcuchów tekstowych

Czasami podczas wczytywania łańcuchów tekstowych do programu chciałbyś pobrać jednocześnie cały wiersz znaków zamiast pojedynczego słowa, jak to się dzieje w przypadku korzystania ze spacji w roli separatora.

Istnieje specjalna funkcja, `getline`, wczytująca od razu cały wiersz. Pobiera ona „strumień wejściowy” i odczytuje z tego strumienia wiersz znaków. Przykładem strumienia wejściowego może być metoda `cin`, z której zwykle korzystasz w celu wczytywania pojedynczych słów (wyjaśnię Ci pewną tajemnicę, o której jeszcze nie wspomniałem — otóż metoda ta jest tak naprawdę obiektem, podobnie jak łańcuch tekstowy albo wektor, i jej typem jest strumień wejściowy, a `cin>>` to metoda wczytująca dane; wyjaśnianie tego wszystkiego w pierwszym rozdziale nie wydawało mi się dobrym pomysłem).

Oto prosty przykład demonstrujący wczytywanie pojedynczego wiersza wprowadzonego przez użytkownika:

### Przykładowy kod 41.: *getline.cpp*

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string wiersz;
    cout << "Proszę wpisać wiersz tekstu: ";
    getline( cin, wiersz, '\n' );
    cout << "Wiersz, który wpisałeś, to: " << '\n' << wiersz;
}
```

Program ten wczytuje do łańcucha *wiersz* sekwencję znaków aż do chwili, w której zostanie napotkany znak nowego wiersza, innymi słowy, gdy użytkownik naciśnie klawisz *Enter*.

Sam znak nowego wiersza zostanie pominięty — wczytany łańcuch będzie zawierać wszystkie wprowadzone znaki, aż do znaku nowego wiersza. Jeśli zechcesz, aby w danym łańcuchu znalazł się znak nowego wiersza, będziesz musiał go tam wstawić samodzielnie. Symbolem oznaczającym koniec wczytywania może być dowolny znak; nie musi to być znak nowego wiersza (taki znak nazywany jest **separatorem** lub delimiterem, ponieważ separuje on znaki wczytywane od niewczytywanych). Aby funkcja *getline* zwróciła swój wynik, użytkownik nadal będzie musiał nacisnąć klawisz *Enter*, ale wczytany tekst będzie zawierać wyłącznie znaki znajdujące się przed separatorem.

Spójrzmy na przykład demonstrujący, jak wczytywać tekst w formacie CSV, czyli z wartościami rozdzielonymi przecinkami. Dane w formacie CSV wyglądają następująco:

Stefan, Janecki, Aleja Szparagowa 40, Sopot, pomorskie, Polska

Każdy z przecinków ogranicza jedną z sekcji danych. Format ten przypomina nieco arkusz kalkulacyjny, z tym że dane nie są porozmieszczone w kolumnach arkusza, lecz rozdzielone przecinkami. Napiszmy program, który potrafi wczytywać wprowadzone przez użytkownika dane w formacie CSV i który przechowuje zestawienie uczestników gry komputerowej w następującym formacie:

*<imie gracza>,<nazwisko gracza>,<klasa gracza>*

Kiedy w dalszej części tej książki zapoznasz się z rozdziałem poświęconym plikowym operacjom wejścia-wyjścia, będziesz mógł wprowadzić w swoim programie kilka zmian, które umożliwią wczytywanie plików CSV z dysku, ale na razie pozostanmy przy odczytywaniu danych wprowadzanych przez użytkownika. Program zakończy działanie, kiedy wprowadzone imię gracza będzie puste.

### Przykładowy kod 42.: *csv.cpp*

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
```

```

while ( 1 )
{
    string imie;
    getline( cin, imie, ',' );

    if ( imie.size() == 0 )
    {
        break;
    }
    string nazwisko;
    getline( cin, nazwisko, ',' );

    string klasa_gracza;
    getline( cin, klasa_gracza, '\n' );
    cout << imie << " " << nazwisko << " ma klasę " << klasa_gracza << endl;
}
}

```

Zwróć uwagę na użycie z łańcuchem tekstowym metody `size`, co pozwala wykryć, czy mamy do czynienia z łańcuchem pustym. Jest to jedna z wielu metod, które są dostępne podczas pracy z łańcuchami tekstowymi.

## Długość łańcucha i dostęp do jego elementów

Aby ustalić długość łańcucha tekstuowego, można skorzystać z funkcji `length` albo `size`, z którą właśnie miałeś do czynienia. Funkcje te stanowią część klasy `string`, a każda z nich zwraca liczbę znaków łańcucha tekstuowego:

```

string moj_lancuch1 = "osiemnaście znaków";
int dlugosc = moj_lancuch1.length(); // Albo .size();

```

Miedzy metodami `size` a `length` nie ma różnic — możesz korzystać z tej, która Ci bardziej odpowiada<sup>1</sup>.

Łańcuchy tekstowe mogą być indeksowane liczbowo, tak samo jak tablice. Można na przykład iterować po znakach łańcucha, odczytując je na podstawie indeksu, tak jakby znajdowały się w tablicy. Taka możliwość jest przydatna, gdy chcesz pracować na poszczególnych elementach łańcucha, na przykład w celu odszukania określonego znaku, takiego jak przecinek.

Zwróć uwagę, że zastosowanie funkcji `length` albo `size` jest tu bardzo ważne, dzięki czemu nie nastąpi próba wyjścia poza koniec łańcucha. Podobnie jak w przypadku wykroczenia poza koniec tablicy, wyjście poza łańcuch tekstowy byłoby niebezpieczne.

Oto krótki przykład pokazujący przechodzenie w pętli przez łańcuch tekstowy w celu jego wyświetlenia:

```

for( int i = 0; i < moj_lancuch.length(); i++ )
{
    cout << moj_lancuch[ i ];
}

```

---

<sup>1</sup> Powód, dla którego istnieją obie metody, jest taki, że metoda `size` używana jest ze wszystkimi obiektami kontenerowymi STL, w związku z czym udostępniono ją we względu na zachowanie spójności. Dla większości programistów bardziej naturalne wydaje się jednak korzystanie z metody `length`.

# Wyszukiwanie i podłańcuchy

Klasa `string` udostępnia proste mechanizmy przeszukiwania oraz wydobywania podłańcuchów przy pomocy metod `find`, `rfind` oraz `substr`. Metoda `find` pobiera podłańcuch oraz pozycję w łańcuchu wyjściowym i odnajduje pierwsze wystąpienie podłańcucha, począwszy od podanej pozycji. Wynikiem działania tej metody jest indeks pierwszego wystąpienia szukanego podłańcucha albo specjalna wartość całkowita `string::npos` oznaczająca, że dany podłańcuch nie został odnaleziony.

Poniższy przykładowy program szuka wszystkich wystąpień podłańcucha "kot" w danym tekście i podaje łączną liczbę tych wystąpień:

## Przykładowy kod 43.: szukaj.cpp

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string tekst;
    int i = 0;
    int liczba_kotow = 0;
    cout << "Proszę wprowadzić wiersz z tekstem: ";
    getline( cin, tekst, '\n' );
    for ( i = tekst.find( "kot", 0 ); i != string::npos; i = tekst.find( "kot", i ) )
    {
        liczba_kotow++;
        i++; // Przejdź poza ostatnie wykryte wystąpienie słowa, aby uniknąć
              // ponownego wyszukania tego samego łańcucha
    }
    cout << "Słowo kot występuje " << liczba_kotow << " razy w tekście " <<
        ↵"" << tekst << "";
}
```

Jeśli chcesz odnaleźć wystąpienia podłańcucha, począwszy od końca łańcucha tekstu, w dokładnie taki sam sposób możesz skorzystać z funkcji `rfind`, z tym że przeszukiwanie będzie odbywać się w kierunku przeciwnym (łańcuchy nadal porównywane będą od strony lewej do prawej, co znaczy, że funkcja `rfind` szukająca słowa "kot" nie będzie porównywać go ze słowem "tok").

Funkcja `substr` tworzy nowy łańcuch tekstowy o podanej długości, zawierający wycinek danego łańcucha, począwszy od zadanej pozycji:

```
// Przykładowy prototyp
string substr (int pozycja, int dlugosc);
```

Aby na przykład wydzielić pierwsze dziesięć znaków łańcucha tekstu, można napisać:

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string moj_lancuch = "abcdefghijklmno";
    string pierwsze_dziesiec_liter = moj_lancuch.substr( 0, 10 );
```

```

cout << "Pierwsze dziesięć liter alfabetu to: "
    << pierwsze_dziesiec_liter;
}

```

## Przekazywanie łańcucha przez referencję

Łańcuchy tekstowe mogą być dość duże i zawierać wiele danych. Rzecz jasna nie wszystkie łańcuchy będą wielkie, ale zazwyczaj dobrą praktyką jest pobieranie parametrów łańcuchowych przez referencję:

```
void drukujLancuch (string& lancuch);
```

Przypomnij, że parametr referencyjny przypomina trochę wskaźnik; zamiast kopii zmiennej łańcuchowej przekazywana jest referencja do oryginalnej zmiennej:

```
string lancuch_do_wyswietlenia = "w tym łańcuchu znajduje się tylko jedno x";
drukujLancuch(lancuch_do_wyswietlenia );
```

W powyższym przykładzie funkcja drukujLancuch nie skopiuje zmiennej lancuch\_do\_wyswietlenia, tylko pobierze jej adres. Z parametru lancuch można korzystać tak jak z oryginalnej zmiennej.

Istnieją jednak minusy przekazywania zmiennych przez referencję. Pamiętaj, że w takim przypadku funkcja pobiera adres oryginalnej zmiennej, w związku z czym może ją modyfikować. Prawdopodobnie pisząc daną funkcję po raz pierwszy, nie zastosujesz takiego rozwiązania przypadkowo, ale kiedy powrócisz do niej podczas konserwacji programu (na przykład w celu dodania nowej funkcjonalności), możesz zapomnieć o tym, że przekazywana zmienna nie powinna być modyfikowana i — ojej! — zmienić ją. Ktoś wywołujący Twoją funkcję będzie zszokowany odkryciem, że jego dane uległy zmianie!

C++ udostępnia mechanizm zapobiegający wprowadzaniu przypadkowych zmian w parametrach referencyjnych — w funkcji można określić, że referencja ma być stała, do czego służy specjalne słowo kluczowe `const`. Referencji stałej nie można modyfikować, chociaż można ją odczytywać.

```
void drukuj_lancuch (const string& lancuch)
{
    cout << lancuch; // Dozwolone — łańcuch nie jest modyfikowany
    lancuch = "abc"; // Niedozwolone!
}
```

Za każdym razem, kiedy dodajesz do funkcji parametr referencyjny, zastanów się, czy funkcja ta powinna modyfikować referencję. Jeśli nie chcesz modyfikować parametru, oznacz go jako `const`, dzięki czemu funkcja go nie zmieni (i nie będzie mogła tego zrobić). Użycie słowa kluczowego `const` jednoznacznie określa, że parametr nie może być modyfikowany.

Użycie `const` nie ogranicza się wyłącznie do referencji. W ten sam sposób można oznaczać pamięć, do której odwołuje się wskaźnik. W tym przypadku można napisać mniej więcej taki kod:

```
void drukuj_wsk (const int* w_wart)
{
    if ( w_wart == NULL ) // Dobrze — pamięć, na którą wskazuje w_wart,
                           // nie jest modyfikowana
    {
        return;
    }
    cout << *w_wart; // Dobrze — pamięć może być odczytywana
```

```
*w_wart = 20; // Źle — pamięć, na którą wskazuje w_wart, jest modyfikowana  
w_wart = NULL; // Dobrze — pamięć nie jest modyfikowana, tylko sam wskaźnik  
}
```

Zauważ, że kompilator jest bardzo mądry i wie, czy Twój kod przypisuje wartość do pamięci, do której odwołuje się wskaźnik. Aby stwierdzić, co dzieje się z referencją, kompilator nie ogranicza się wyłącznie do sprawdzenia, czy wskaźnik podlega dereferencji. Modyfikowanie samego wskaźnika jest jak najbardziej dopuszczalne, ponieważ jego wartość jest kopiwana — zmiana wartości wskaźnika `w_wart` nie ma wpływu na zmienną przekazywaną do funkcji.

Z modyfikatora `const` można korzystać w szerszym zakresie w celu zagwarantowania, że określona zmienna nigdy nie zostanie poddana modyfikacjom. Jeśli spróbujesz zmienić wartość takiej zmiennej, kompilator ostrzeże Cię, że zrobisz coś, czego nie powinieneś. Jeśli jakąś zmienną zadeklarujesz jako `const`, powinieneś od razu nadać jej jakąś wartość (ponieważ nie będziesz mógł jej zmienić później).

```
const int x = 4; // Dobrze — podczas tworzenia zmiennej można przypisać do niej wartość  
x = 4; // Źle — zmiennej x nie można modyfikować
```

Używanie modyfikatora `const` przy każdej możliwej okazji świadczy o dobrym stylu programowania. Oznaczenie zmiennej jako `const` ułatwia czytanie reszty kodu, gdyż wiesz, że nikt nie będzie mógł tej zmiennej modyfikować. Kiedy więc zobaczyś przypisanie do tej zmiennej, będziesz mieć pewność, że wartość ta nie ulegnie późniejszej zmianie. Nie będziesz musiał śledzić w kodzie, czy przybiera ona jakieś inne wartości; będziesz mógł za to skupić się na pozostały (nie-`const`) zmiennych i sprawdzać, czy są one modyfikowane. Użycie słowa kluczowego `const` gwarantuje także, że wartość zmiennej nie ulegnie przypadkowej zmianie, wpływając na działanie kodu, w którym przyjęto, że zmienna ta będzie mieć taką samą wartość jak na początku.

Jeśli na przykład kod prosi użytkownika o podanie imienia i nazwiska, po czym tworzy nowy łańcuch tekstowy zawierający pełne dane użytkownika, zmienna zawierająca ten łańcuch może zostać zadeklarowana jako `const`, ponieważ nie powinna ona podlegać zmianom.

## Szerzenie się `const`

Modyfikator `const` przypomina wirusa. Kiedy jakaś zmienna zostanie zdefiniowana jako `const`, nie będzie jej można przekazać przez referencję do metody, która przyjmuje referencje nie-oznaczone jako stałe. Zmiennej takiej nie można też przekazać poprzez wskaźnik do metody, która pobiera wskaźniki nieokreślone jako `const`, ponieważ metoda ta mogłaby próbować zmienić wartość tej zmiennej za pośrednictwem wskaźnika. `const X*` stanowi inny typ niż `X*`, a `const X&` (deklaracja referencji do `X`) jest innym typem niż `X&`. Można przekształcić `X*` w `const X*` albo `X&` w `const X&`, ale podobny zabieg w drugą stronę jest niemożliwy. Jeśli na przykład napiszesz następującą metodę, nie będziesz mógł jej skompilować:

```
void drukuj_wart_nieconst (int& w_wart)  
{  
    cout << w_wart;  
}  
  
const int x = 10;  
  
drukuj_wart_nieconst( x ); // Nie skompiluje się; nie można przekazać  
// stałej całkowitej do funkcji przyjmującej niestałą referencję
```

Ograniczenie to dotyczy wyłącznie referencji i wskaźników, gdy oryginalna wartość jest współdzielona. Kiedy zmienna podlega kopiowaniu, jak to jest w przypadku przekazywania parametru przez wartość, nie ma potrzeby, aby parametr funkcji był stały:

```
void drukuj_wart_nieconst (int wart)
{
    cout << wart;
}

const int x = 10;

drukuj_wart_nieconst( x ); // Prawidłowo; x jest kopowane, tak więc nie ma
// znaczenia, że wartość nie jest stała, ponieważ jest ona lokalna w funkcji
```

W rezultacie, gdy tylko jedna ze zmiennych zostanie zdefiniowana jako const, może okazać się, że inne zmienne także muszą być oznaczone w taki sposób, co w szczególności dotyczy wskaźnikowych i referencyjnych parametrów funkcji.

Użycie zmiennych zadeklarowanych jako const może sprawiać trudności podczas pracy z biblioteką albo zestawem metod pomocniczych, w których w ogóle nie użyto takich zmiennych. Z drugiej jednak strony powinieneś używać modyfikatora const, kiedy tworzysz bibliotekę albo własne metody pomocnicze, dzięki czemu w kodzie wywołującym te metody będzie można korzystać z zalet wynikających ze stosowania tego rodzaju zmiennych.

Biblioteka standardowa C++ została napisana z uwzględnieniem zmiennych zadeklarowanych jako const, przez co bezpiecznie można korzystać z takich zmiennych we własnym kodzie oraz używać ich w połączeniu z tą biblioteką.

W dalszej części tej książki będę korzystał z modyfikatora const, gdy tylko rozwiązanie takie będzie właściwe.

Należy wiedzieć, że możliwe jest zadeklarowanie zmiennej jako const wewnątrz pętli, nawet jeśli będzie ona resetowana w każdej iteracji:

```
for ( int i = 0; i < 10; i++ )
{
    const i_do_kwadratu = i * i;
    cout << i_do_kwadratu;
}
```

Zmienną `i_do_kwadratu` można zadeklarować jako const, chociaż jej wartość jest nadawana na nowo w każdym przebiegu pętli. Jest to możliwe, ponieważ zasięg tej zmiennej ogranicza się wyłącznie do ciała pętli. Z punktu widzenia kompilatora zmienna `i_do_kwadratu` jest tworzona na nowo w każdej iteracji pętli.

## Const i STL

W poprzednim rozdziale, poświęconym STL, przyjrzaliśmy się funkcji, która potrafiła wyświetlać mapę. Być może zauważłeś, że mapa była przekazywana przez wartość, co znaczy, iż w celu przekazania mapy do funkcji `wyswietlMape` konieczne było jej kopiowanie. Oto jeszcze raz ta funkcja:

```
void wyswietlMape (map<string, string> mapa_do_wyswietlenia) // Mapa jest kopowana!
{
    for ( map<string, string>::iterator itr = mapa_do_wyswietlenia.begin(),
        koniec = mapa_do_wyswietlenia.end();
```

```
    itr != koniec;
    ++itr )
{
    cout << itr->first << " --> " << itr->second << endl;
}
}
```

Aby uniknąć kopiowania mapy, lepiej byłoby skorzystać z referencji. Jeszcze lepiej byłoby zadeklarować tę referencję jako const, żeby było jasne, że funkcja wyłącznie wyświetla mapę i w żaden sposób jej nie modyfikuje.

```
void wyswietlMape (const map<string, string>& mapa_do_wyswietlenia)
{
    for ( map<string, string>::iterator itr = mapa_do_wyswietlenia.begin(),
        ↪koniec = mapa_do_wyswietlenia.end();
        itr != koniec;
        ++itr )
    {
        cout << itr->first << " --> " << itr->second << endl;
    }
}
```

Jeśli postąpisz w taki sposób, kompilator zasypie Cię komunikatami o błędach! Problem polega na tym, że deklarując mapę jako const, oznajmiasz, że nikt nie powinien modyfikować elementów mapy, chociaż mogą to robić iteratory. W celu zmiany mojego adresu w swojej książce adresowej mógłbyś na przykład napisać:

```
if ( itr->first == "Alex Allain" )
{
    itr->second = "webmaster2@cplusplus.com"
}
```

Na szczęście biblioteka STL jest przyjazna dla modyfikatora const i wszystkie kontenery STL zawierają drugi, specjalny rodzaj iteratora, którym jest `const_iterator`. Można z niego korzystać tak jak ze zwykłego iteratora, z tym że nie pozwala on na modyfikowanie iterowanego kontenera przez nadpisanie iteratora:

```
void wyswietlMape (const map<string, string>& mapa_do_wyswietlenia)
{
    for ( map<string, string>::const_iterator itr = mapa_do_wyswietlenia.begin(),
        ↪koniec = mapa_do_wyswietlenia.end();
        itr != koniec;
        ++itr )
    {
        cout << itr->first << " --> " << itr->second << endl;
    }
}
```

Jeśli chcesz iterować po kontenerze zdefiniowanym jako const, zawsze powinieneś korzystać z iteratora `const_iterator`. Dobrą praktyką jest także stosowanie go, jeśli zależy Ci wyłącznie na odczytywaniu danych i nie musisz kontenera modyfikować.

## Sprawdź się

1. Który z poniższych zapisów prezentuje poprawny kod?

- A. `const int& x;`
- B. `const int x = 3; int *p_int = & x;`

- C. const int x = 12; const int \*p\_int = & x;  
 D. int x = 3; const int y = x; int& z = y;
- 2.** Która z poniższych sygnatur funkcji umożliwia skompilowanie kodu const int x = 3;  
 fun( x );?
- A. void fun (int x);  
 B. void fun (int& x);  
 C. void fun (const int& x);  
 D. Prawidłowe są odpowiedzi A i C.
- 3.** Jaki jest najlepszy sposób na stwierdzenie, czy szukanie łańcucha tekstuowego powiodło się?
- A. Porównać zwróconą pozycję z 0.  
 B. Porównać zwróconą pozycję z -1.  
 C. Porównać zwróconą pozycję ze string::npos.  
 D. Sprawdzić, czy zwrócona pozycja jest większa od długości łańcucha tekstuowego.
- 4.** W jaki sposób można utworzyć iterator dla stałego kontenera STL?
- A. Zadeklarować iterator jako const.  
 B. Zamiast iteratora należy skorzystać z indeksów w celu przejścia kontenera w pętli.  
 C. Za pomocą zapisu const\_iterator.  
 D. Zadeklarować typy szablonowe jako const.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

*Jeśli tylko będzie to właściwe, w każdym z zadań praktycznych zastosuj zmienne i referencje zadeklarowane jako const. Znaczy to, że jeśli będziesz pisać funkcję przyjmującą łańcuch tekstuowy, z dużym prawdopodobieństwem zechcesz przekazać go za pomocą referencji stałej.*

- 1.** Napisz program wczytujący dwa łańcuchy tekstuowe: „igla” i „stog siana”, oraz liczący, ile razy igła występuje w stogu siana.
- 2.** Napisz program pozwalający użytkownikowi wpisywać dane tabelaryczne w formacie podobnym do pliku CSV, z tym że zamiast przyjmować separatory w postaci przecinków, program powinien spróbować samodzielnie wykryć użyte separatory. Najpierw użytkownik powinien wprowadzić wiersze z danymi. Następnie program powinien poszukać możliwego separatora, wyszukując znaki niebędące cyframi, literami ani spacjami. Po odnalezieniu takich znaków we wszystkich wierszach program powinien wyświetlić je na ekranie i zapytać użytkownika, który z nich jest separatorem. Jeśli na przykład zostaną wprowadzone następujące dane:

Alex Allain, webmaster@cprogramming.com  
 Jan Kowalski, jan@nigdzie.com

program powinien poprosić użytkownika o dokonanie wyboru spośród przecinka, znaku małpy oraz kropki.

- 3.** Napisz program, który wczytuje wpisany przez użytkownika tekst w formacie HTML (bez obaw, w dalszej części książki opiszę, jak wczytywać pliki). Program powinien obsługiwać następujące znaczniki HTML: <html>, <head>, <body>, <b>, <i> oraz <a>. Każdy znacznik HTML występuje w wersji otwierającej, np. <html>, oraz zamkającej,

która jest poprzedzona ukośnikiem: </html>. Między znacznikami znajduje się tekst obsługiwany przez dany rodzaj znacznika, np. <b>tekst, który należy pogrubić </b>, albo <i>tekst, który należy pochylić </i>. Znaczniki <head></head> kontrolują tekst zawierający metadane, natomiast znaczniki <body></body> otaczają tekst do wyświetlania. Znaczniki <a> są używane do tworzenia hiperłączy, w których adres URL ma następujący format: <a href=URL>napis</a>.

Gdy Twój program wczyta tekst w formacie HTML, powinien pominąć znaczniki <html>. Tekst z sekcji <head> powinien zostać usunięty, dzięki czemu nie zostanie wyświetlony. Następnie program powinien pokazać na ekranie cały tekst zawarty w treści głównej, zmieniając go w taki sposób, aby fragmenty znajdujące się między znacznikami <b> i </b> zostały otoczone gwiazdkami (\*), fragmenty między znacznikami <i> oraz </i> zostały otoczone znakami podkreślenia (\_), natomiast tekst ze znaczników <a href=URL>napis</a> został wyświetlony jako tekst łącza (linkurl).

# ■ ■ ■ R O Z D Z I A Ł 2 0

## Debugowanie w Code::Blocks

---

Poznałeś już wiele przydatnych technik programistycznych, ale namierzanie błędów w złożonych programach nadal może być trudne. Na szczęście istnieje narzędzie służące w tym pomocą. Jest to **debugger**. Debugger umożliwia sprawdzanie stanu programu podczas jego działania, ułatwiając tym samym zrozumienie, co tak naprawdę się w nim dzieje. Początkujący programiści często odkładają na później naukę stosowania tego narzędzia, ponieważ korzystanie z niego wydaje się im uciążliwe albo zbędne. To fakt, aby korzystać z debugera, trzeba go poznać, ale ignorowanie go to jak dostrzeganie samych tylko drzew tam, gdzie rośnie wielki las. Debuggery pozwalają zaoszczędzić mnóstwo czasu, a w porównaniu z wcześniejszym pełzaniem korzystanie z nich przypomina chodzenie. Będzie Ci potrzebna praktyka i na początku będziesz się potykać, ale gdy już zapragniesz debugger do pracy, będziesz mknąć do przodu niczym wiatr.

W rozdziale tym omówię debugger środowiska Code::Blocks, ponieważ jeśli pracujesz w systemie Windows i przeprowadziłeś konfigurację Code::Blocks zgodnie z wcześniejszymi wskazówkami, powinien on być już u Ciebie zainstalowany. Istnieje wiele różnych debuggerów, ale większość pojęć zawiązanych z ich użyciem jest wspólna. Zamieściłem tu wiele zrzutów ekranowych, dzięki czemu nawet jeśli nie korzystasz z Windows, będziesz mógł przeczytać ten rozdział i zobaczyć, jak wygląda debugger. Twój środowisko programistyczne niemal na pewno jest wyposażone we własny debugger<sup>1</sup>.

W rozdziale tym będę korzystał z programów zawierających błędy w celu zaprezentowania rzeczywistego procesu debugowania. Jeśli tylko zechcesz prześledzić opisywane przeze mnie przypadki, będziesz mógł utworzyć w Code::Blocks (albo w innym środowisku, w którym pracujesz) odrębny projekt dla każdego z tych przykładów.

Pierwszy program powinien dla podanej kwoty naliczać odsetki na podstawie procenta składanego. Niestety, zawiera on jakiś błąd, wskutek czego wyświetla niewłaściwą kwotę.

### **Przykładowy kod 44.: debugowanie1.cpp**

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

double obliczOdsetki (double podstawa, double oprocentowanie, int lata)
{
```

<sup>1</sup> Jeśli korzystasz z Linuksa, będziesz mógł użyć GDB. Jeżeli pracujesz z Visual Studio albo Visual Studio Express, będziesz mieć do dyspozycji dołączone do nich bardzo dobre debuggery. Istnieją także debuggery samodzielne, z których możesz korzystać, ale ich omówienie wykracza poza zakres tej książki. Należy do nich na przykład WinDbg, który stanowi część Debugging Tools for Windows Microsoftu: <http://www.microsoft.com/whdc/devtools/debugging/default.mspx>. Xcode firmy Apple również udostępnia debugger.

```
double koncowy_mnoznik;
for ( int i = 0; i < lata; i++ )
{
    koncowy_mnoznik *= (1 + oprocentowanie);
}
return podstawa * koncowy_mnoznik;
}

int main ()
{
    double podstawa;
    double oprocentowanie;
    int lata;
    cout << "Podaj podstawę: ";
    cin >> podstawa;
    cout << "Podaj oprocentowanie: ";
    cin >> oprocentowanie;
    cout << "Podaj liczbę lat: ";
    cin >> lata;
    cout << "Po " << lata << " latach będziesz mieć " << obliczOsetki( podstawa,
        oprocentowanie, lata ) << " złotych" << endl;
}
```

Oto przykładowy rezultat działania tego programu:

```
Podaj podstawę: 100
Podaj oprocentowanie: .1
Podaj liczbę lat: 1
Po 1 latach będziesz mieć 1.40619e-306 złotych
```

Niedobrze!  $1.40619e-306$  złotych to z pewnością nieprawidłowa kwota. Najwidoczniej mamy w programie jakiś błąd. Spróbujmy uruchomić kod w debuggerze, aby stwierdzić, skąd biorą się nasze problemy.

## Zaczynamy

Najpierw, aby ułatwić sobie czynności debugowania, upewnijmy się, że środowisko Code::Blocks jest poprawnie skonfigurowane. W tym celu musimy utworzyć tak zwane **symbole debugowania**. Pomagają one debuggerowi zorientować się, który wiersz kodu jest akurat wykonywany, dzięki czemu będziemy wiedzieć, gdzie w programie się znajdujemy. Aby określić symbole debugowania, z menu *Project* wybierz polecenie *Build options*. Powinno wyświetlić się poniższe okno dialogowe (zobacz pierwszy rysunek na kolejnej stronie).

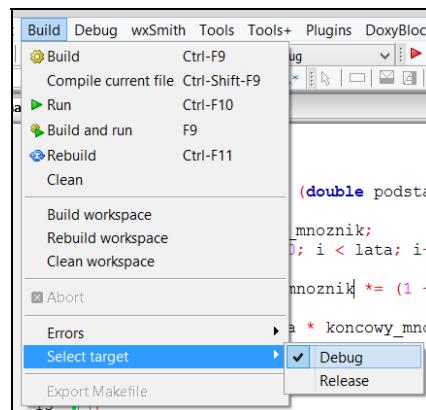
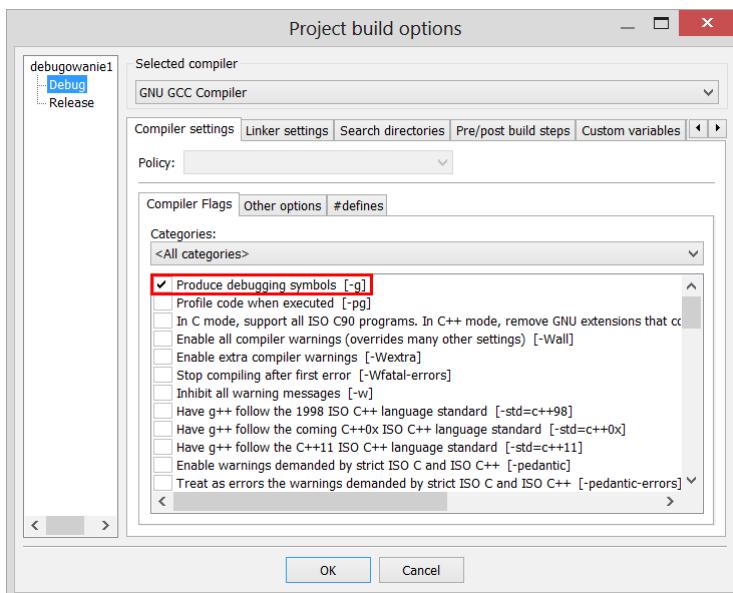
Upewnij się, czy dla pozycji *Debug* zaznaczona jest opcja *Produce debugging symbols [-g]*. Powinieneś także sprawdzić, czy jako cel komplikacji zaznaczono *Debug* (menu *Build>Select target/Debug*) (zobacz drugi rysunek na kolejnej stronie).

Dzięki temu zagwarantujesz, że Twój program będzie komplikowany przy użyciu symboli debugowania, które zostały skonfigurowane dla celu *Debug*.

Jeśli nie widzisz ani celu *Debug*, ani *Release*, zaznacz po prostu pole wyboru *Produce debugging symbols [-g]* dla swojego bieżącego celu komplikacji<sup>2</sup>. Upewnij się też, że opcja *Strip all symbols*

---

<sup>2</sup> Jeśli korzystasz z kompilatora g++, w celu utworzenia symboli komplikacji będziesz musiał w wierszu poleceń wpisać argument *-g*. Jeżeli używasz Xcode, on sam zatrudzi się o dołączenie odpowiednich symboli.



*from binary (minimizes size) [-s]* NIE jest zaznaczona (zazwyczaj cele komplikacji są definiowane podczas tworzenia projektu; najłatwiej zagwarantujesz, że poszczególne ustawienia będą poprawne, jeśli pozostawisz domyślne opcje proponowane przez Code::Blocks podczas konfigurowania projektu).

Kiedy już wszystko skonfigurujesz, będziesz mógł przystąpić do działania. Jeśli kompilowałeś swój program wcześniej, ale musiałeś zmienić jego konfigurację, będziesz musiał ponownie go skompilować. Kiedy już to zrobisz, możesz zabrać się za debugowanie!

## Wstrzymywanie działania programu

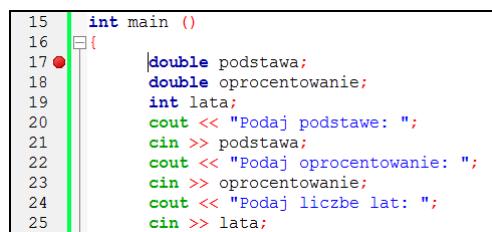
Jedną z zalet debuggera jest możliwość podejrzenia tego, co robi program — który fragment kodu jest wykonywany i jakie są wartości jego zmiennych. Aby zapoznać się z tymi informacjami, będziemy musieli **wstrzymać** działanie programu. W tym celu ustawimy gdzieś w kodzie

punkt wstrzymania i uruchomimy nasz program pod kontrolą debugera. Debugger zacznie wykonywać program, aż natrafi na linię z punktem wstrzymania. W tym miejscu będziesz mógł rozejrzeć się po swoim kodzie albo przejść go wiersz po wierszu i sprawdzić, w jaki sposób instrukcje w tych wierszach wpływają na Twoje zmienne.

Zdefiniujmy dość wczesny punkt wstrzymania, na samym początku funkcji `main`, żebyśmy mogli przesledzić wykonywanie się całego programu. W tym celu ustaw kursor w wierszu

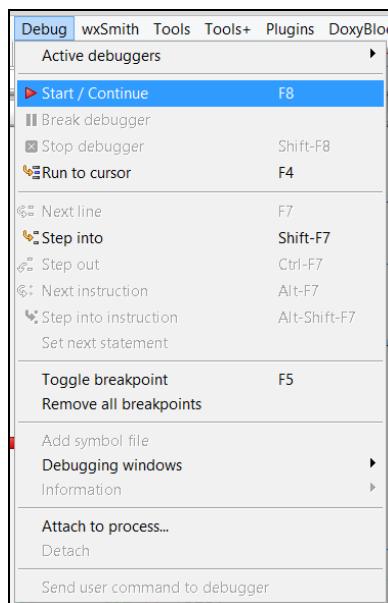
```
double podstawa;
```

i wybierz polecenie *Debug/Toggle breakpoint* (albo naciśnij *F5*). Na pasku bocznym, obok wybranego wiersza kodu, zostanie umieszczona mała czerwona kropka — oznacza ona, że w tym miejscu znajduje się punkt wstrzymania.



```
15 int main ()
16 {
17 ● double podstawa;
18     double oprocentowanie;
19     int lata;
20     cout << "Podaj podstawię: ";
21     cin >> podstawa;
22     cout << "Podaj oprocentowanie: ";
23     cin >> oprocentowanie;
24     cout << "Podaj liczbę lat: ";
25     cin >> lata;
```

Punkt ten można włączać i wyłączać, korzystając z polecenia *Toggle breakpoint*; można go także kliknąć. Teraz, kiedy masz już punkt wstrzymania, możesz uruchomić program! Wywołaj polecenie *Debug/Start* albo naciśnij klawisz *F8*.



Gdy tak postąpisz, program zacznie wykonywać się jak zwykle, aż dotrze do punktu wstrzymania. W naszym przypadku stanie się to praktycznie od razu, ponieważ punkt wstrzymania znajduje się w pierwszej linii programu.

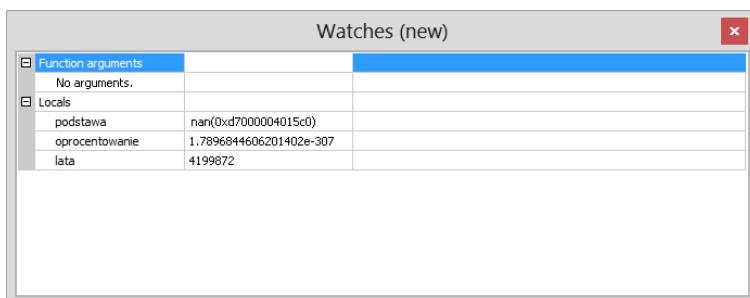
Teraz powinieneś zobaczyć, jak otwiera się okno debuggera, które będzie wyglądać mniej więcej tak (mogą zostać otwarte jeszcze inne okna, ale o nich opowieムjuż za chwilę):

```

15 int main ()
16 {
17     double podstawa;
18     double oprocentowanie;
19     int lata;
20     cout << "Podaj podstawę: ";
21     cin >> podstawa;
22     cout << "Podaj oprocentowanie: ";
23     cin >> oprocentowanie;
24     cout << "Podaj liczbę lat: ";
25     cin >> lata;
26
27     cout << "Po " << lata << " latach będziesz mieć " << obliczOdsetki( podstawa, oprocentowanie, lata ) << " złotych" << endl;
28 }
```

Należy zwrócić uwagę przede wszystkim na żółty trójkąt, widoczny pod czerwoną kropką. Trójkąt ten wskazuje linię kodu, która zostanie wykonana w następnej kolejności. Znajduje się on kilka wierszy pod naszą kropką. Trójkąt nie jest położony bezpośrednio pod nią, ponieważ tak naprawdę nie ma żadnego kodu maszynowego (tj. kodu wykonywanego przez procesor, który jest wynikiem komplikacji kodu w C++), związanego z deklaracjami zmiennych, tak więc nasz punkt wstrzymania, chociaż wydaje się znajdować w wierszu 17., tak naprawdę jest umieszczony w wierszu 20. (liczby widoczne z lewej strony kropki i trójkąta to numery wierszy).

Powinno także zostać otwarte okno *Watches* — będzie ono wyglądać mniej więcej tak (wartości, które zobaczyś, mogą być inne):



Jeśli go nie widzisz, rozejrzyj się za nim; może być schowane za innymi oknami debuggera.

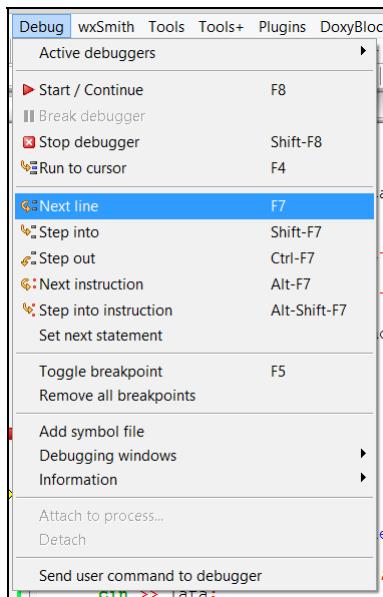
W oknie *Watch* rozwinąłem dwie pozycje — *Locals* oraz *Function Arguments*. Okno to pokazuje wszystkie aktualnie dostępne zmienne — zarówno zmienne lokalne, jak i argumenty funkcji — oraz ich wartości. Zauważ, że wartości wyglądają na pozabawione jakiegokolwiek sensu! Jest tak dlatego, że jeszcze ich nie zainicjalizowaliśmy; stanie się to dopiero w kilku następnych wierszach programu<sup>3</sup>.

W celu wykonania kilku następnych wierszy programu musimy poprosić debugger, aby przeszedł do następnej linii kodu. Przejście do kolejnego wiersza spowoduje wykonanie się bieżącej linii (oznaczonej żółtą strzałką). W Code::Blocks służy do tego polecenie *Next line*, czyli następny wiersz (zobacz rysunek na kolejnej stronie).

Zamiast z polecenia *Next line* można skorzystać ze skrótu klawiaturowego *F7*<sup>4</sup>.

<sup>3</sup> Pamiętaj, że zmienne nie są inicjalizowane podczas ich deklarowania.

<sup>4</sup> Był może zastanawiasz się, dlaczego istnieje zarówno polecenie *Next line*, jak i *Next instruction*. My zawsze będziemy korzystać z polecenia *Next line*. *Next instruction* jest używane podczas debugowania bez symboli, co wykracza poza zakres tej książki.



Kiedy już przejdziemy do następnego wiersza, program wywoła instrukcję cout i wyświetli na ekranie komunikat z prośbą o wprowadzenie wartości. Jeśli jednak spróbujesz wpisać jakąś kwotę, nie uda Ci się; program powróci do debuggera. Jeszcze raz naciśnij F7, aby wykonać następny wiersz kodu. Teraz program będzie czekać, aż użytkownik poda kwotę, ponieważ funkcja cin nie zwróciła jeszcze wartości — do tego potrzebne będą informacje uzyskane od użytkownika. Dla zachowania zgodności z naszym raportem dotyczącym błędów wpisz 100, po czym podaj wartości dwóch kolejnych zmiennych, które wprowadziliśmy za pierwszym razem: .1 dla oprocentowania i 1 dla liczby lat do obliczenia procenta składanego.

Teraz dotarliśmy do następującego wiersza:

```
cout << "Po " << lata << " latach będziesz mieć " << obliczOdsetki( podstawa,
    oprocentowanie, lata ) << " złotych" << endl;
```

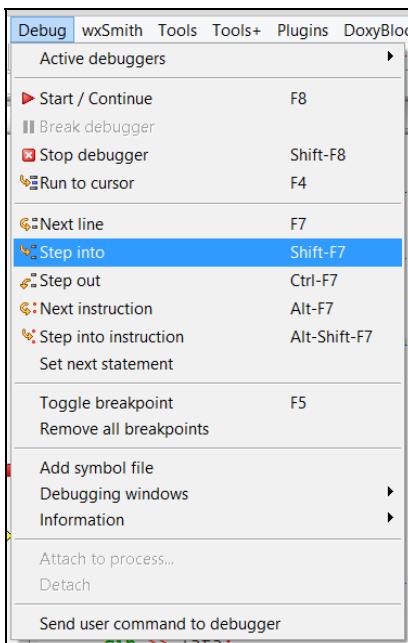
Sprawdźmy jeszcze raz, czy prawidłowo wprowadziliśmy dane. W tym celu można skorzystać z okna *Watch* i przyjrzeć się wartościami zmiennych lokalnych.

| Watches (new)      |                    |  |
|--------------------|--------------------|--|
|                    |                    |  |
| Function arguments |                    |  |
| Locals             |                    |  |
| podstawa           | 100                |  |
| oprocentowanie     | 0.1000000000000001 |  |
| lata               | 1                  |  |

Jak dotąd wszystko jest w porządku. Podstawa wynosi 100, oprocentowanie .1, a liczba lat 1. Co mówisz? Że oprocentowanie nie jest równe .1? Masz rację. Oprocentowanie w rzeczywistości wynosi .1000000000000001, ale ta mała jedynka na końcu to tylko zaburzenie wyni-

kające ze sposobu, w jaki reprezentowane są liczby zmiennoprzecinkowe. Pamiętaj, że liczby te nie są idealnie dokładne. Różnice wynikające z reprezentacji tych liczb są jednak na tyle małe, że w większości programów nie będą one mieć znaczenia<sup>5</sup>.

Teraz, kiedy już wiemy, że jak do tej pory wszystko jest w porządku, sprawdźmy, co się dzieje wewnętrz funkcji `oblicz0dsetki`. W tym celu należy wywołać kolejne polecenie debugera, którym jest *Step into*:



Polecenie *Step into* służy do wejścia do funkcji, która ma zostać wywołana z bieżącego wiersza — w przeciwieństwie do *Next line*, które powoduje wykonanie się całej funkcji i zwrócenie jej wyniku, jak to widzieliśmy w przypadku funkcji `cin`. Z polecenia *Step into* należy korzystać w celu debugowania wnętrza funkcji, co już za chwilę zrobimy.

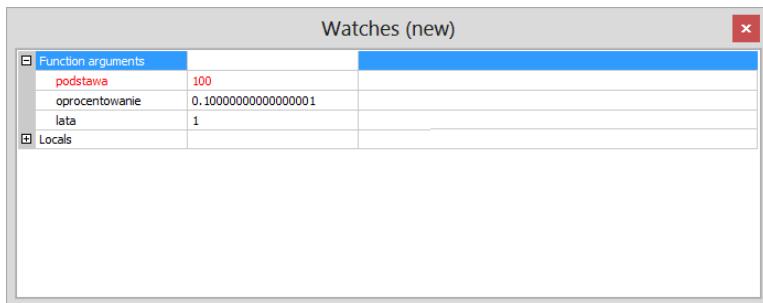
Wkroczmy zatem do funkcji `oblicz0dsetki`. Być może zastanawiasz się, czy w poniższym wierszu nie ma przypadkiem wielu wywołań funkcji.

```
cout << "Po " << lata << " latach będziesz mieć " << oblicz0dsetki( podstawa,
    >>oprocentowanie, lata ) << " złotych" << endl;
```

Co z tymi wszystkimi wywołaniami funkcji `cout`? Debugger Code::Blocks jest bardzo sprytny — nie wkroczy do funkcji pochodzących z biblioteki standardowej. Możemy po prostu przejść do naszego wiersza i od razu znajdziemy się w funkcji `oblicz0dsetki` z pominięciem funkcji, które nas nie interesują. Zróbcmy to teraz.

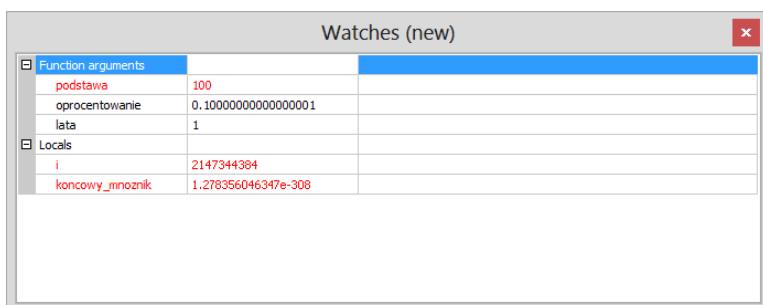
Teraz, gdy znajdujemy się wewnętrz funkcji `oblicz0dsetki`, pierwsze, co musimy zrobić, to sprawdzić, czy argumenty funkcji są prawidłowe — być może pomieszałyśmy kolejność argumentów. Rozszerzmy sekcję *Function Arguments* w oknie *Watch*:

<sup>5</sup> Prawdę jest jednak, że błędy zmiennoprzecinkowe mogą się nawarstwiać, co w przypadku niektórych aplikacji może wywołać poważne konsekwencje. Niemniej naszego programu to nie dotyczy.



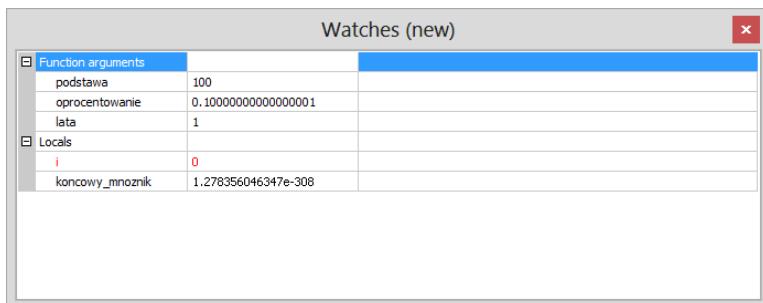
Wszystko wygląda dobrze!

Spójrzmy na zmienne lokalne:



Czy widzisz coś dziwnego? Zarówno zmienna *i*, jak i *koncowy\_mnoznik* wyglądają na zupełnie pozbawione sensu! Pamiętaj jednak, że kiedy poprzednim razem zaglądaliśmy do okna *Watch*, widzieliśmy bezsensowne wartości, ponieważ nie były one jeszcze zainicjalizowane. Wybierzmy polecenie *Next line* (*F7*), aby przeprowadzić inicjalizacje związane z naszą pętlą i zobaczyć, co się wtedy stanie.

Inicjalizacja pętli odbywa się w jednym tylko wierszu, tak więc możemy ponownie sprawdzić zmienne lokalne. Powinny wyglądać mniej więcej tak:



Z *i* jest już wszystko w porządku, ale co ze zmienną *koncowy\_mnoznik*? Nie wygląda na to, aby była poprawnie zainicjalizowana. Co więcej, już za chwilę w wierszu, w którym się znajdujemy, zostanie ona użyta:

```
koncowy_mnoznik *= (1 + oprocentowanie);
```

Wiersz ten nakazuje wykonać mnożenie koncowy\_mnoznik \* (1 + oprocentowanie) i przypisać otrzymany wynik do zmiennej koncowy\_mnoznik, ale jak widzimy, jej wartość jest wzięta z sufitu, w związku z czym mnożenie da w rezultacie wynik pozbawiony sensu.

Czy wiesz, jak naprawić taki błąd?

W wierszu, w którym jest deklarowana zmienna koncowy\_mnoznik, musimy ją zainicjalizować; w naszym przypadku powinniśmy przypisać jej wartość 1.

To tyle. Namierzyliśmy problem i znaleźliśmy rozwiązanie. Dziękujemy ci, debuggerze!

## Debugowanie awarii

Zajmijmy się innym rodzajem błędu, jakim jest awaria. Awarie często należą do tego typu usterek, które najbardziej przerażają początkujących programistów, ponieważ wydają się być ekstremalne. Wraz z upływem czasu awarie staną się jednak Twoim ulubionym rodzajem błędu do tropienia. To dlatego, że będziesz dokładnie wiedzieć, w którym miejscu pojawił się problem. W programie wystąpiła awaria, ponieważ znalazły się w nim nieprawidłowe dane, a Ty będziesz mógł go zatrzymać dokładnie w miejscu pojawienia się błędu, aby stwierdzić, co to były za dane i skąd one pochodzą.

Poniżej znajduje się prosty (ale zawierający błędy) program, który w liście powiązanej tworzy kilka węzłów, po czym wyświetla każdą wartość z listy.

### Przykładowy kod 45.: debugowanie2.cpp

```
#include <iostream>

using namespace std;

struct ListaPowiazana
{
    int wart;
    ListaPowiazana *nastepny;
};

void drukujListe (const ListaPowiazana *lst)
{
    if ( lst != NULL )
    {
        cout << lst->wart;
        cout << "\n";
        drukujListe( lst->nastepny );
    }
}

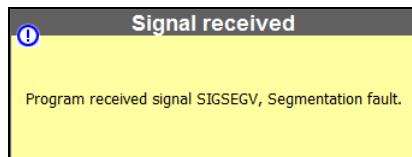
int main ()
{
    ListaPowiazana *lst;
    lst = new ListaPowiazana;
    lst->wart = 10;
    lst->nastepny = new ListaPowiazana;
    lst->nastepny->wart = 11;
    drukujListe( lst );

    return 0;
}
```

Program ten jednak nie zadziała, kiedy spróbujesz go uruchomić. Może wystąpi w nim awaria, a może wpadnie w nieskończoną pętlę. Z pewnością coś w nim nie gra!

Spróbujmy uruchomić go pod kontrolą debuggera i zobaczymy, czy to nam w czymś pomoże. Wybierz polecenie *Debug/Start* albo naciśnij klawisz *F8*.

Niemal natychmiast debugger wyświetli następujący komunikat:



Błąd segmentacji jest spowodowany użyciem wskaźników, które są nieprawidłowe, i zwykle oznacza, że program próbował przeprowadzić dereferencję wskaźnika NULL albo wskaźnika błędnego (czyli takiego, który został wcześniej zwolniony albo nigdy nie został zainicjalizowany). Możesz wyobrazić sobie, że program próbował uzyskać dostęp do segmentu pamięci, do którego nie miał prawa<sup>6</sup>.

W jaki sposób możemy ustalić, skąd wziął się zły wskaźnik? Otóż debugger zatrzymał się dokładnie w miejscu, w którym wystąpiła awaria. Aby znaleźć podejrzany wiersz kodu, poszukaj żółtej strzałki:

```
cout << lst->wart;
```

W linii tej znajduje się tylko jeden wskaźnik — `lst`. Sprawdźmy w oknie *Watch*, jaką ma on wartość. Jak widać, wynosi ona `0xbaadf00d`<sup>7</sup>! Dziwne, prawda? Jest to specjalna wartość używana przez kompilator w celu inicjalizowania pamięci podczas jej alokacji. Funkcjonalność ta jest stosowana wyłącznie podczas uruchamiania programów pod kontrolą debuggera, dlatego też będziesz mieć do czynienia z innym zachowaniem programu podczas jego debugowania niż wtedy, gdy uruchamiasz go poza debuggerem. Debugger jest pomocny dzięki zastosowaniu określonej wartości, o której wiadomo, że powoduje błąd segmentacji, gdy nastąpi próba jej odczytania — w ten sposób szybko dowiesz się, że użyty wskaźnik nie był zainicjalizowany<sup>8</sup>.

Teraz już wiemy, że wskaźnik `lst` nie został zainicjalizowany, ale dlaczego? Skorzystajmy z kolejnej funkcjonalności debuggera, którą jest **stos wywołań**. Stos wywołań pokazuje wszystkie funkcje, które są akurat wykonywane. Oto jak wygląda stos wywołań w oknie *Call stack*:

<sup>6</sup> W niektórych środowiskach używany jest termin *Access Violation* (naruszenie zasad dostępu), który ma to samo znaczenie.

<sup>7</sup> Jeśli jeszcze nie znasz takiego zapisu, powinieneś wiedzieć, że jest on stosowany do zapisu liczb szesnastkowych (heksadecymalnych) — liczb, których podstawa jest 16. Zwykle są one są poprzedzone prefiksem `0x`, a litery od `A` do `F` oznaczają liczby od 10 do 15. Szesnastkowe `0xA` jest zatem tą samą liczbą co dziesiątkowe 10.

<sup>8</sup> Alternatywne rozwiązywanie polega na użyciu wartości, która była zapisana w zmiennej, jaka wcześniej znajdowała się w miejscu, gdzie obecnie przechowywany jest wskaźnik. Ponieważ zawartość tego miejsca pamięci jest nieprzewidywalna i może nawet sprawiać wrażenie prawidłowej, program, który z niej korzysta, może zachowywać się dość dziwnie, a przyczyna takiego zachowania będzie trudna do wyśledzenia. Na przykład, zamiast ulec szybkiej awarii, program odczyta niewłaściwą pamięć i zepsuje się później, kiedy skorzysta z jej zawartości. Debugger stara się ułatwić nam życie, uspójniając zachowanie programów i gwarantując, że kod ulegnie awarii tak szybko, jak to tylko możliwe, dzięki czemu będziesz mógł znaleźć się jak najbliżej miejsca, w którym pojawił się problem.

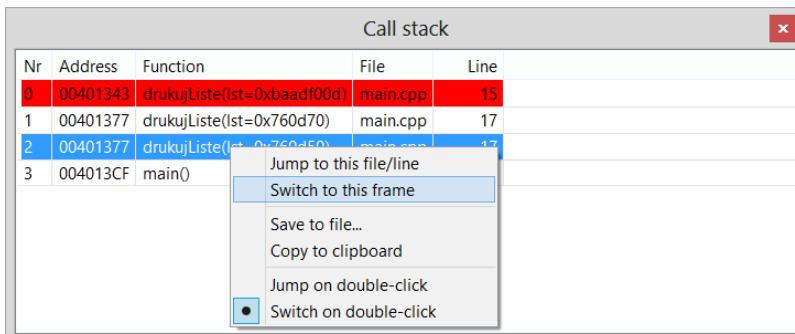
| Call stack |          |                             |          |      |
|------------|----------|-----------------------------|----------|------|
| Nr         | Address  | Function                    | File     | Line |
| 0          | 00401343 | drukujListe(lst=0xbaadf00d) | main.cpp | 15   |
| 1          | 00401377 | drukujListe(lst=0x570d70)   | main.cpp | 17   |
| 2          | 00401377 | drukujListe(lst=0x570d50)   | main.cpp | 17   |
| 3          | 004013CF | main()                      | main.cpp | 28   |

W oknie tym znajduje się kilka kolumn: *Nr* to po prostu numer, z którego możesz korzystać, odwołując się do każdej z ramek stosu. *Address* to adres danej funkcji<sup>9</sup>. Kolumna *Function* zawiera nazwę funkcji oraz jej argumenty (w rzeczy samej możesz stwierdzić, że *lst* jest równe `0xbaadf00d`, patrząc wyłącznie na stos wywołań). Poza tym znajdują się tu także kolumny z nazwą pliku oraz numerem wiersza, dzięki czemu będziesz mógł odszukać linię kodu, która była wykonywana.

Wykonywana akurat funkcja znajduje się na górze stosu wywołań; poniżej znajduje się funkcja, która ją wywołała itd. Na samym dole widoczna jest funkcja `main`, gdyż rozpoczyna ona program.

Widzimy, że zostały przeprowadzone trzy wywołania funkcji `drukujListe`, z których dwa pierwsze miały poprawne wartości wskaźników, natomiast w trzecim pojawiła się wartość `0xbaadf00d`. Jak pamiętasz, nasza funkcja `main` utworzyła dwa węzły listy. Dwa pierwsze wywołania funkcji `drukujListe` z pewnością skorzystały z tych węzłów, z kolei trzecie wywołanie użyło wskaźnika niezainicjalizowanego. Teraz wiemy, że jeszcze raz powinniśmy spojrzeć na kod, który inicjalizuje listę, i widzimy, że w ogóle nie nadajemy wartości `NULL` zmiennej następny węzła na końcu listy.

Chociaż udało nam się rozwiązać problem, może zdarzyć się, że będziesz potrzebował uzyskać więcej informacji na temat innych ramek stosu. W celu przyjrzenia się zmiennym lokalnym możesz zmienić kontekst debugera na dowolną inną ramkę stosu. W tym celu kliknij prawym przyciskiem myszy interesującą Cię ramkę stosu, po czym wybierz polecenie *Switch to this frame*.



Debugger przeniesie żółtą strzałkę na wywołanie funkcji, która jest wykonywana w tej ramce stosu. W celu analizowania zmiennych lokalnych, które stanowią część tej ramki stosu, będziesz mógł korzystać także z okna *Watches*.

<sup>9</sup> Adres funkcji może być przydatny, jeśli debugujesz na poziomie assemblera. W większości przypadków nie będziesz jednak z niego korzystać.

## Zaglądanie do zawieszzonego programu

Czasami nie masz do czynienia ze zwykłą awarią, ale widzisz, że program „tkwi w miejscu” — być może w nieskończonej pętli albo w oczekiwaniu na zakończenie jakiegoś powolnego wywołania systemowego. Jeśli znalazłeś się w takich okolicznościach, możesz uruchomić swój program pod kontrolą debuggera, poczekać na wystąpienie problemu, po czym włamać się do programu przy pomocy debuggera.

Aby zapoznać się z taką sytuacją, skorzystamy z jeszcze jednego przykładowego kodu:

### Przykładowy kod 46.: *debugowanie3.cpp*

```
#include <iostream>

using namespace std;

int main ()
{
    int silnia = 1;
    for ( int i = 0; i < 10; i++ )
    {
        silnia *= i;
    }
    int suma = 0;
    for ( int i = 0; i < 10; i++ )
    {
        suma += i;
    }
    // Silnia bez dwójkii
    int silnia_bez_dwojki = 1;
    for ( int i = 0; i < 10; i++ )
    {
        if ( i == 2 )
        {
            continue;
        }
        silnia_bez_dwojki *= i;
    }
    // Suma bez dwójkii
    int suma_bez_dwojki = 0;
    for ( int i = 0; i < 10; i++ )
    {
        if ( i = 2 )
        {
            continue;
        }
        suma_bez_dwojki += i;
    }
}
```

Program ten nigdy nie zakończy się, jeśli go uruchomisz — utknie w jakimś miejscu. Aby dowiedzieć się, gdzie to nastąpiło, uruchomimy go pod kontrolą debuggera, poczekamy, aż się zawiesi, po czym rozejrzymy się w nim.

Najpierw skompiluj program i uruchom go pod kontrolą debuggera (polecenie *Debug/Start* albo klawisz *F8*). Kiedy program zostanie uruchomiony, stwierdzisz, że nie może on się skończyć — gdzieś utknął, prawdopodobnie w jakimś rodzaju nieskończonej pętli. Poprośmy debuggera, aby włamał się do naszego działającego programu, i zobaczymy, co się stało. W tym

celu należy wywołać polecenie *Debug/Stop Debugger*, które umożliwia włamanie się do programu i rozejrzenie się w danym momencie jego wykonywania się (z polecenia tego można skorzystać także w celu zakończenia sesji debugera, jeśli program działa pod jego kontrolą).

Kiedy już zatrzymasz program, powinieneś spojrzeć na stos wywołań, który tym razem wygląda dość dziwnie, mniej więcej tak:

| Call stack |          |                           |                  |      |
|------------|----------|---------------------------|------------------|------|
| Nr         | Address  | Function                  | File             | Line |
| 0          | 7C90120F | ntdll!DbgUiConnectToDbg() | C:\WINDOWS\sy... |      |
| 1          | 7C951E40 | ntdll!KiIntSystemCall()   | C:\WINDOWS\sy... |      |
| 2          | 00000005 | ??()                      |                  |      |
| 3          | 00000004 | ??()                      |                  |      |
| 4          | 00000001 | ??()                      |                  |      |
| 5          | 004EF0D0 | ??()                      |                  |      |
| 6          | 00000000 | ??()                      |                  |      |

Nic tutaj nie jest naszym kodem! O co chodzi? To, co widzisz, jest wynikiem naszego włamania się do działającego programu. Czy zwróciłeś uwagę, że na wierzchu stosu znajduje się funkcja *ntdll!DbgUiConnectToDbg*? *Ntdll* jest biblioteką systemu Windows, natomiast wywoływana funkcja (*DbgUiConnectToDbg*) służy do włamywania się do uruchomionego procesu. Ale gdzie jest kod wykonywanego programu? Okazuje się, że w celu włamania się do procesu debugger utworzył kolejny wątek (wątek to sposób na jednocześnie wykonywanie kodu). Aby włamać się do procesu, debugger musi mieć możliwość uruchomienia pewnego kodu w czasie, gdy wykonywany jest nasz oryginalny kod. Debugger daje sobie z tym radę, tworząc nowy wątek, który dokonuje włamania do kodu. W poprzednich przykładach nie mieliśmy do czynienia z tym drugim wątkiem, ponieważ debugger dysponował wystarczającą kontrolą, aby włamywać się do kodu bez potrzeby tworzenia nowego wątku. W tym przypadku jednak nie chcieliśmy dostać się do programu w określonym wierszu jego kodu, tylko w pewnym momencie jego działania, aby sprawdzić, co wówczas robił. Teraz — żeby przejść do naszego kodu — musimy przełączyć się na właściwy wątek.

Aby mieć możliwość przełączania się między wątkami, należy otworzyć okno z wątkami (*Debug/Debugging windows/Running threads*) (zobacz pierwszy rysunek na kolejnej stronie).

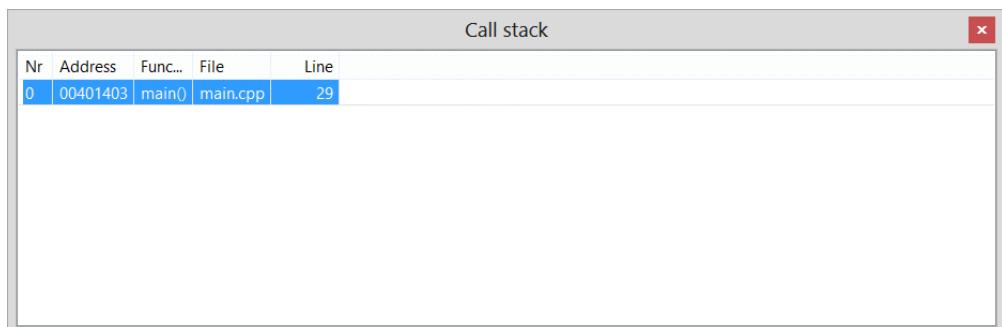
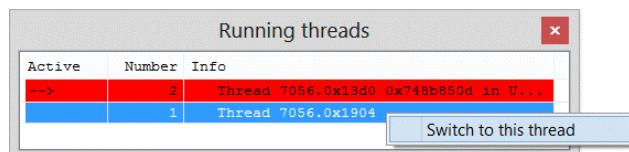
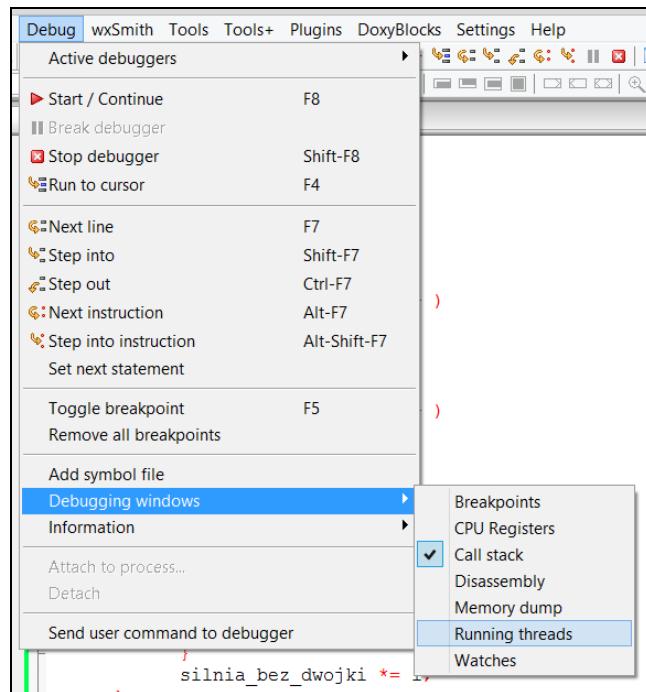
W oknie *Running threads* zobaczysz dwa wątki (zobacz drugi rysunek na kolejnej stronie).

Symbol --> w kolumnie *Active* wskazuje bieżący wątek. W tym przypadku jest to wątek użyty do włamania się do procesu. Chcielibyśmy przełączyć się na drugi wątek, aby zobaczyć informacje na jego temat. W tym celu kliknij ten wątek i wybierz polecenie *Switch to this thread* (zobacz trzeci rysunek na kolejnej stronie).

Teraz możemy powrócić do stosu wywołań i zapoznać się z o wiele czytelniejszymi informacjami (zobacz czwarty rysunek na kolejnej stronie).

To już jest nasz kod. Zobaczysz, że debugger umieścił żółtą strzałkę przy wierszu 29., wskazując następną linię do wykonania. To ten kod:

```
for ( int i = 0; i < 10; i++ )
{
    if ( i == 2 )
    {
        continue;
    }
    suma_bez_dwojki += i;
}
```



Ponieważ nasz program utknął w miejscu, a my znaleźliśmy się wewnątrz pętli, słuszne będzie przyjęcie hipotezy, że to właśnie ta pętla nie może się zakończyć. W jaki sposób możemy tego dowieść? Przejedźmy ten program krok po kroku (tzn. linia po linii).

Powinniśmy jednak zachować ostrożność. Jeśli po prostu klikniemy w debuggerze polecenie *Next line*, zostanie wykonany kod z wątku, który został użyty w celu włamania się do naszego kodu, gdyż jest to wątek, który właśnie się wykonuje. Zamiast przechodzić do następnego wiersza, musimy wstawić w naszym kodzie punkt wstrzymania i pozwolić, aby nasz program wykonywał się do momentu natrafienia na ten punkt<sup>10</sup>. Umieścmy punkt wstrzymania w wierszu z instrukcją *if*, po czym wybierzmy polecenie *Continue (Ctrl+F7)*. Kiedy kod dotrze do tego punktu, znajdziemy się we właściwym wątku i znowu będziemy mogli korzystać z opcji *Next line* w celu sprawdzania, co się dzieje w programie.

Zobaczysz, że za każdym razem natrafiamy na instrukcję *if ( i = 2 )*, a następnie powracamy na początek pętli.

Co tu się dzieje? Przyjrzyjmy się wartości zmiennej *i* w oknie *Locals*. Kiedy znajdujemy się w wierszu z pętlą, *i* jest równe 2. Po wykonaniu kodu pętli *i* przyjmuje wartość 3. Gdy wykonywana jest instrukcja *if*, *i* znowu powraca do wartości 2!

Wygląda na to, że ktoś nadaje zmiennej *i* wartość 2 — w tym przypadku musi to być instrukcja *if*. Istotnie! Mamy tu do czynienia z często spotykanym błędem użycia pojedynczego, zamiast podwójnego, znaku równości.

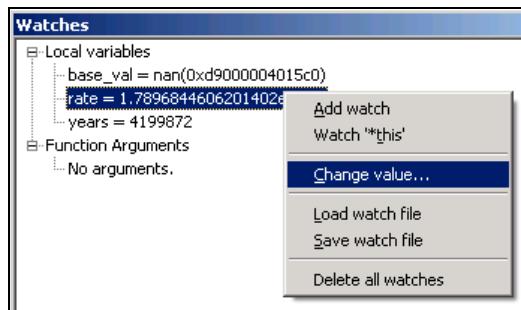
A tak przy okazji — być może zastanawiasz się, dlaczego program nigdy nie dochodzi do wiersza *continue*, tylko bezpośrednio z instrukcji *if* przeskakuje do pętli *for*. Jest to swojego rodzaju osobliwość w zachowaniu debuggera. Czasami bezpośrednie odniesienie kodu maszynowego do konkretnego wiersza w kodzie może być trudne. W tym przypadku debugger ma kłopot z odróżnieniem instrukcji *if ( x = 2 )* od instrukcji *continue*. Od czasu do czasu będziesz mieć do czynienia z takim zachowaniem debuggera, gdy wykonywany kod niezupełnie pokrywa się z Twoimi oczekiwaniami. Z pewnością podczas debugowania będziesz zwracać uwagę na tego typu przypadki.

## Modyfikowanie zmiennych

Czasami podczas debugowania zechcesz zmienić wartość jakiejś zmiennej, na przykład po to, aby zagwarantować, że jeśli zmienna ta będzie mieć określoną wartość, dalsza część kodu rzeczywiście zadziała. Można to zrobić w oknie *Watch* — jeśli klikniesz zmienną prawym przyciskiem myszy, będziesz mógł wybrać polecenie *Change value* i nadać wybranej zmiennej potrzebną Ci wartość (zobacz rysunek na kolejnej stronie).

Uważaj, aby w ten sposób nie nadawać wartości zmiennym przed ich inicjalizacją, gdyż w przeciwnym razie zostaną one nadpisane.

<sup>10</sup>Niektóre debuggery umożliwiają dokonanie wyboru, który wątek ma być wykonywany, poprzez „zamrażanie” wątków, jednak Code::Blocks nie dysponuje taką opcją.



## Podsumowanie

Środowisko Code::Blocks wyposażono w debugger, z którego możesz korzystać, aby szybko rozpocząć debugowanie. Jeśli pracujesz w systemie innym niż Windows, wiele (o ile nie wszystkie) opisanych tu koncepcji również będzie mieć zastosowanie, chociaż być może w zmienionej postaci. Podstawowa idea debugowania sprowadza się do lepszego zrozumienia stanu programu dzięki zastosowaniu takich rozwiązań jak punkty wstrzymania, przechodzenie kodu krok po kroku w celu dotarcia do właściwego miejsca w programie, po czym sprawdzenie, co się dzieje, poprzez poznanie stosu wywołań oraz wartości poszczególnych zmiennych.

## Zadania praktyczne

W przeciwieństwie do innych rozdziałów tym razem nie będę odpytywał Cię na temat debugowania ani zadawał kodu do napisania. Mam dla Ciebie kilka programów, które należy zdebugować — wszystkie działają nieprawidłowo. Dla każdego z nich powinieneś utworzyć projekt w Code::Blocks i zdebugować go. W niektórych programach może kryć się kilka błędów!

## Zadanie nr 1. Problem z wykładnikiem

### Przykładowy kod 47.: zadanie1.cpp

```
#include <iostream>

using namespace std;
int wykladnik (int podstawa, int wykl)
{
    int wartosc_narast;
    for ( int i = 0; i < wykl; i++ )
    {
        wartosc_narast *= podstawa;
    }
    return podstawa;
}

int main()
{
    int podstawa;
    int wykl;

    cout << "Podaj podstawę: ";
    cin >> podstawa;
    cout << "Podaj wykładnik: ";
    cin >> wykl;
    cout << "Wykładnik wynosi: " << wykladnik(podstawa, wykl) << endl;
}
```

```

    cin >> podstawa;
    cout << "Podaj wykładnik: ";
    cin >> wykl;
    wykladnik( wykl, podstawa );
}

```

## Zadanie nr 2. Problem z dodawaniem liczb

### Przykładowy kod 48.: zadanie2.cpp

```

#include <iostream>

using namespace std;

int sumujWartosci (int *wartosci, int n)
{
    int suma;
    for ( int i = 0; i <= n; i++ )
    {
        suma += wartosci[ i ];
    }
    return suma;
}

int main()
{
    int rozmiar;
    cout << "Podaj rozmiar: ";
    cin >> rozmiar;
    int *wartosci = new int[ rozmiar ];
    int i;
    while ( i < rozmiar )
    {
        cout << "Podaj wartość do dodania: ";
        cin >> wartosci[ ++i ];
    }
    cout << "Łączna suma wynosi: " << sumujWartosci( wartosci, rozmiar );
}

```

## Zadanie nr 3. Problem z ciągiem Fibonacciego<sup>11</sup>

### Przykładowy kod 49.: zadanie3.cpp

```

#include <iostream>

using namespace std;

int fibonacci (int n)
{
    if ( n == 0 )
    {
        return 1;
    }
}

```

<sup>11</sup> Jeśli nie znasz ciągu Fibonacciego, przydatne informacje o nim znajdziesz na stronie [http://pl.wikipedia.org/wiki/Ciąg\\_Fibonacciego](http://pl.wikipedia.org/wiki/Ciąg_Fibonacciego).

```
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
    }

int main()
{
    int n;
    cout << "Podaj liczbę, dla której należy obliczyć ciąg Fibonacciego: " << endl;
    cin >> n;
    cout << fibonacci( n );
}
```

## Zadanie nr 4. Problem z odczytywaniem i wyświetlaniem listy

### Przykładowy kod 50.: zadanie4.cpp

```
#include <iostream>

using namespace std;

struct Wezel
{
    int wart;
    Wezel *w_nastepny;
};

int main()
{
    int wart;
    Wezel *w_glowa;
    while ( 1 )
    {
        cout << "Podaj wartość, 0, aby powtórzyć: " << endl;
        cin >> wart;
        if ( wart = 0 )
        {
            break;
        }
        Wezel *w_tymczas = new Wezel;
        w_tymczas = w_glowa;
        w_tymczas->wart = wart;
        w_glowa = w_tymczas;
    }
    Wezel *w_itr = w_glowa;
    while ( w_itr != NULL )
    {
        cout << w_itr->wart << endl;
        w_itr = w_itr->w_nastepny;
        delete w_itr;
    }
}
```

# III

■ ■ ■ CZĘŚĆ III

## Tworzenie większych programów

---

**Uwaga!** Jeśli do tej pory tylko czytałeś tę książkę i nie rozwiązywałeś zadań praktycznych, ZATRZYMAJ SIĘ. Nie będziesz mógł docenić ani wykorzystać wiadomości zawartych w tej części, jeżeli chociaż trochę nie programowałeś. Niedługo zapoznasz się z jednymi z najważniejszych informacji, jakie zarówno w tej książce, ale będą one dla Ciebie niezrozumiałe, jeśli nie masz jeszcze za sobą żadnych doświadczeń natury praktycznej.

Wiele idei, o których pisałem do tej pory, umożliwiał Ci robienie nowych rzeczy. Teraz jednak nadeszła chwila, aby nie rozmawiać o robieniu kolejnych nowych rzeczy, ale o robieniu rzeczy większych. Na razie pisałeś niewielkie programy — domyślam się, że zwykle nie przekraczały one kilkuset wierszy. Gdy programy nie są zbyt duże, łatwo nad nimi zapanować, ale być może zacząłeś jużauważać, że praca z dłuższymi programami jest trudniejsza. Nawet jeśli jeszcze tak się nie dzieje, osiągniesz wreszcie punkt, w którym programy stają się za duże. Dla niektórych osób będzie to kilkaset wierszy, a dla innych kilka tysięcy, a może nawet jeszcze więcej, chociaż tak naprawdę nie ma to znaczenia. Dobra pamięć bez wątpienia jest niezlym atutem, ale nikt nie zrobi rzeczywiście interesujących rzeczy, posługując się wyłącznie własną pamięcią. Wszystkie programy stają się zbyt duże, aby je zrozumieć w całości. Chcesz napisać grę komputerową? Oprogramowanie naukowe? System operacyjny? Będą Ci potrzebne techniki umożliwiające nadawanie dużym programom odpowiedniej struktury i ułatwiające ich zrozumienie.

Na szczęście wielu programistów zetknęło się już z tego typu problemami i opracowali oni techniki upraszczające tworzenie dużych aplikacji. Reguły zarysowane w kilku kolejnych rozdziałach umożliwiają pisanie obszernych i zaawansowanych programów. Upraszczają one również projektowanie małych aplikacji.

Poznajmy zatem kilka koncepcji, do których będziemy powracać w trakcie nauki projektowania dużych programów. Zaczniemy od poradzenia sobie z kodem fizycznym — jak rozmieścić kod na dysku, aby nie był on jednym wielkim plikiem cpp. W dalszej kolejności omówię projekt logiczny programu — w jaki sposób pisać programy bez konieczności jednocięsnego pamiętania wszystkich szczegółów dotyczących jego funkcjonowania.



# 21

## ■ ■ ■ ROZDZIAŁ 21

# Rozbijanie programów na mniejsze części

W miarę jak Twoje programy zaczną się robić coraz większe, nie będziesz chciał, aby cały ich kod znajdował się w pojedynczym pliku źródłowym. Wprowadzanie zmian w programie stanie się niewygodne, a podczas szukania rozmaitych elementów w kodzie łatwo będzie można się pogubić. Kiedy Twoje programy zaczną liczyć po kilka tysięcy wierszy, z pewnością zechcesz podzielić je na więcej plików źródłowych<sup>1</sup>.

Korzystanie z więcej niż tylko jednego pliku źródłowego ułatwia orientowanie się, co gdzie się znajduje, ponieważ każdy z plików jest mniejszy i może zawierać kod związanego z określonym aspektem programu. Takie rozwiązańe upraszcza także projektowanie programu, gdyż każdy plik nagłówkowy będzie mieć specyficzny interfejs związany z kodem źródłowym, a inne pliki nie będą mogły korzystać z funkcji ani struktur danych, które nie zostały zdefiniowane w tych plikach. Mogłoby się wydawać, że takie podejście stwarza ograniczenia, jednak w rzeczywistości ułatwia ono odseparowanie implementacji każdego z podsystemów od funkcjonalności, które są udostępniane innym podsystemom.

## Proces komplikacji w języku C++

Zanim zaczniesz rozbijać swój kod na wiele plików, powinieneś lepiej zrozumieć podstawy procesu komplikacji kodu w języku C++.

Tak naprawdę słowo „komplikacja” nie odpowiada w pełni rzeczywistości. Komplikacja nie oznacza nawet tworzenia pliku wykonywalnego. Generowanie takiego pliku jest procesem wielo-etapowym; najważniejsze etapy to **przetwarzanie wstępne, komplikacja i konsolidacja**. Cały proces polegający na przejściu od plików z kodem źródłowym do pliku wykonywalnego najlepiej można określić słowem **budowanie**. Komplikacja jest tylko jednym z etapów procesu budowy, a nie całym procesem. Zwykle nie ma potrzeby wydawania odrębnych polecień w celu przeprowadzenia każdego z tych procesów; kompilator potrafi na przykład samodzielnie uruchomić preprocesor.

<sup>1</sup> Musiałem kiedyś pracować z plikiem, który liczył prawie 20 000 wierszy kodu i zajmował pół megabajta. Nikt nie chciał go nawet tknąć!

## Przetwarzanie wstępne

Pierwszym etapem w procesie budowy jest uruchomienie przez kompilator **preprocesora**. Celem preprocesora języka C jest wprowadzenie przed komplikacją zmian w tekście pliku. Preprocesor rozumie **dyrektywy preprocesora**, czyli polecenia zapisane bezpośrednio w pliku źródłowym, które jednak nie są przeznaczone dla kompilatora, ale właśnie dla preprocesora.

Wszystkie dyrektywy preprocesora zaczynają się symbolem krzyżyka (#). Sam kompilator nigdy tych dyrektyw nie zobaczy!

Na przykład instrukcja

```
#include <iostream>
```

mówi preprocesorowi, aby wczytał treść pliku `iostream` bezpośrednio do bieżącego pliku. Za każdym razem, kiedy dołączasz plik nagłówkowy, zostanie on dosłownie wklejony do Twojego pliku, jeszcze zanim zobaczy go kompilator, a sama dyrektywa `#include` zostanie usunięta.

Preprocesor rozwija również **makra**. Makro to łańcuch tekstowy, który jest zastępowany innym, zwykle bardziej złożonym łańcuchem. Makra umożliwiają przechowywanie stałych w jednym centralnym miejscu, dzięki czemu ich zmienianie jest łatwiejsze.

Możesz na przykład napisać

```
#define MOJE_IMIE "Alex"
```

po czym używać w pliku źródłowym nazwy `MOJE_IMIE` zamiast ciągu znaków "Alex".

```
cout << "Cześć " << MOJE_IMIE << '\n';
```

Kompilator zobaczy następującą instrukcję:

```
cout << "Cześć " << "Alex" << '\n';
```

Jeśli zechcesz zmienić używane w programie imię, zamiast przeprowadzać globalne szukanie i zamianianie w obrębie całego kodu, wystarczy, że poprawisz wiersz zawierający dyrektywę `#define`. Makra umożliwiają centralizowanie informacji, dzięki czemu ich aktualizacja jest łatwiejsza. Jeśli zechciałbyś nadać swojemu programowi numer wersji, aby odwoływać się do niego w kodzie, mógłbyś tego dokonać za pomocą makra:

```
#define WERSJA 4
// ...
cout << "Wersja " << WERSJA
```

Ponieważ preprocesor jest uruchamiany przed kompilatorem, można z niego także korzystać w celu usuwania kodu — czasami zdarza się, że chcesz mieć możliwość komplikowania pewnego fragmentu kodu tylko podczas debugowania. Można to zrealizować, mówiąc preprocesorowi, aby dołączył określony kod źródłowy tylko wtedy, gdy zdefiniowane jest makro. Jeśli kod ten będzie Ci potrzebny, zdefiniujesz makro, a jeśli już nie będziesz potrzebować kodu, skasujesz makro.

Możesz mieć na przykład związkany z debugowaniem kod, który wyświetla wartości pewnych zmiennych, ale nie chcesz, żeby zmienne te były pokazywane przy każdym uruchomieniu programu. Taki efekt możesz osiągnąć, dołączając warunkowo kod debugujący:

### Przykładowy kod 51.: `define.cpp`

```
#include <iostream>
```

```
#define DEBUG
```

```

using namespace std;

int main ()
{
    int x;
    int y;
    cout << "Podaj wartość x: ";
    cin >> x;
    cout << "Podaj wartość y: ";
    cin >> y;
    x *= y;

#ifndef DEBUG
    cout << "Zmienna x: " << x << '\n' << "Zmienna y: " << y;
#endif
    // Kod z użyciem zmiennych x i y
}

```

Jeśli zechcesz wyłączyć wyświetlanie zmiennych, wystarczy, że po prostu dodasz znaki komentarza przed dyrektywą `#define DEBUG`:

```
// #define DEBUG
```

Preprocesor realizuje także sprawdzanie, czy makro NIE jest zdefiniowane. Można na przykład uruchomić kod tylko wtedy, gdy DEBUG NIE zostało zdefiniowane, do czego służy dyrektywa `#ifndef`. Będziemy korzystać z tej techniki podczas omawiania pracy z wieloma plikami nagłówkowymi.

## Kompilacja

**Kompilacja** oznacza przekształcenie pliku z kodem źródłowym (plik `.cpp`) w **plik obiektowy** (plik `.o` albo `.obj`). Plik obiektowy zawiera program w postaci zrozumiałej dla procesora w komputerze, którą są **instrukcje w języku maszynowym**, odpowiadające poszczególnym funkcjom znajdującym się w pliku źródłowym. Każdy plik źródłowy jest **kompilowany oddzielnie**, co znaczy, że pliki obiektowe zawierają język maszynowy tylko dla pliku źródłowego, który został skompilowany. Jeśli na przykład skompilujesz (ale nie skonsolidujesz) trzy różne pliki, otrzymasz w rezultacie trzy pliki obiektowe o nazwach `<nazwapliku>.o` albo `<nazwapliku>.obj` (rozszerzenie będzie zależeć od Twojego kompilatora). Każdy z tych plików będzie zawierać kod źródłowy przetłumaczony na język maszynowy. Nie będziesz jednak mógł ich jeszcze uruchomić. Konieczne jest przekształcenie tych plików w pliki wykonywalne, z których będzie mógł korzystać system operacyjny. Do tego właśnie potrzebny jest konsolidator.

## Konsolidacja

**Konsolidacja** oznacza utworzenie pojedynczego pliku wykonywalnego (na przykład EXE albo DLL) na podstawie wielu plików obiektowych i bibliotek<sup>2</sup>. Konsolidator tworzy plik w formacie właściwym dla plików wykonywalnych i przenosi do niego zawartość poszczególnych plików obiektowych. Konsolidator radzi sobie także z plikami obiektowymi, które zawierają odwołania do funkcji zdefiniowanych poza plikiem źródłowym danego pliku obiektowego,

<sup>2</sup> Bądź na podstawie jednego pliku obiektowego, jeśli masz tylko jeden plik źródłowy. Konsolidacja zawsze ma miejsce, nawet w przypadku najprostszych, jednoplikowych programów.

na przykład funkcji będących częścią biblioteki standardowej C++. Kiedy odwołujesz się do standardowej biblioteki C++ (np. `cout << "Witaj!"`), korzystasz z funkcji, która nie została zdefiniowana w Twoim kodzie. Jej definicja znajduje się w odpowiedniku pliku obiektowego, tyle że nie Twojego, ale udostępnionego przez dostawcę kompilatora. Podczas komplikacji kompilator wie, że wywołanie funkcji jest poprawne, gdyż dołączyłeś plik nagłówkowy `iostream`. Ponieważ jednak funkcja ta nie stanowi części pliku `.cpp`, kompilator pozostawia w miejscu jej wywołania zaślepkę. Konsolidator przegląda plik obiektowy, dla każdej zaślepki odszukuje adres funkcji i zastępuje ją poprawnym adresem pochodząącym z jednego z plików obiektowych, które są konsolidowane.

Operacja ta czasami jest nazywana **wyrównywaniem** (ang. *fixup*). Kiedy dzielisz swój program na wiele plików źródłowych, korzystasz z tego, że konsolidator potrafi przeprowadzić takie wyrównanie dla wszystkich funkcji, które dokonują wywołań w innych plikach źródłowych. Jeśli konsolidator nigdzie nie może znaleźć definicji funkcji, wygeneruje błąd niezdefiniowanej funkcji. Nawet jeśli kompilator przepuści jakiś kod, nie znaczy to, że kod ten jest prawidłowy. Konsolidator jako pierwszy patrzy na program całościowo — w sposób, który umożliwia wykrywanie tego typu problemów.

## Dlaczego komplikacja i konsolidacja przebiegają oddzielnie?

Ponieważ nie wszystkie funkcje muszą być zdefiniowane w tym samym pliku obiektowym, istnieje możliwość komplikowania poszczególnych plików źródłowych i późniejszego ich konsolidowania. Jeśli na przykład wprowadzisz zmiany w pliku *CzestoZmieniany.cpp*, a plik *RzadkoZmieniany.cpp* pozostawisz bez zmian, nie trzeba będzie ponownie komplikować pliku obiektowego związanego z plikiem źródłowym *RzadkoZmieniany.cpp*. Pomijanie niepotrzebnych komplikacji może przyczynić się do zaoszczędzenia mnóstwa czasu przeznaczonego na budowanie. Im większa jest Twoja baza kodu, tym więcej czasu zyskasz<sup>3</sup>.

Aby móc w pełni korzystać z komplikacji warunkowej, będziesz potrzebować narzędzia pamiętającego, czy określony plik obiektowy jest **aktualny**, czyli czy odpowiadający mu plik źródłowy (albo jeden z plików nagłówkowych dołączanych przez ten plik) nie został zmieniony od czasu ostatniej komplikacji. Jeśli pracujesz w systemie Windows i korzystasz z Code::Blocks, środowisko to zajmie się tym w Twoim imieniu. Jeżeli pracujesz na Macu, Xcode automatycznie sprawdzi aktualność plików obiektowych, gdy dodasz nowe pliki w menu *File/New/New file....* W Linuksie możesz korzystać z narzędzia *make* (<http://www.gnu.org/software/make/make.html>), które znajduje się w większości dystrybucji \*nix<sup>4</sup>.

## Jak rozbić program na wiele plików?

W jaki zatem sposób można ustrukturyzować swój kod, aby skorzystać z zalet oddzielnej komplikacji? Zajmijmy się krok po kroku przykładowym prostym programem, *Oryg.cpp*, zawierającym nieco współdzielonego kodu, który chciałbyś teraz użyć w nowej aplikacji. Całość

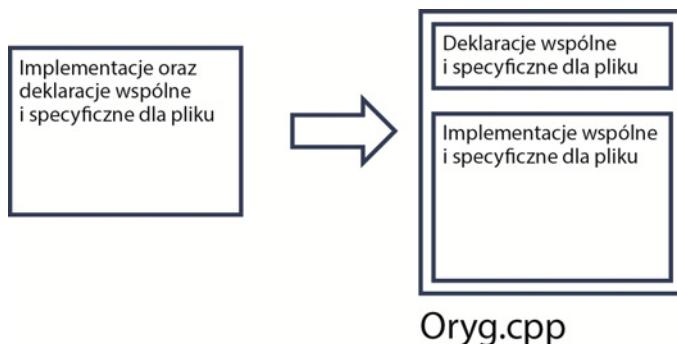
<sup>3</sup> Miałem do czynienia z bazami kodu, które godzinami budowały się od podstaw, a słyszałem o takich bazach, które budowały się przez kilka dni.

<sup>4</sup> Więcej informacji o plikach *makefile* znajdziesz pod adresem <http://www.cprogramming.com/tutorial/makefiles.html>.

procesu opiszę w bardzo usystematyzowany sposób, dzięki czemu będziesz mógł przyjrzeć się poszczególnym jego etapom. W praktyce jednak wiele z tych etapów będziesz mógł przeprowadzić jednocześnie.

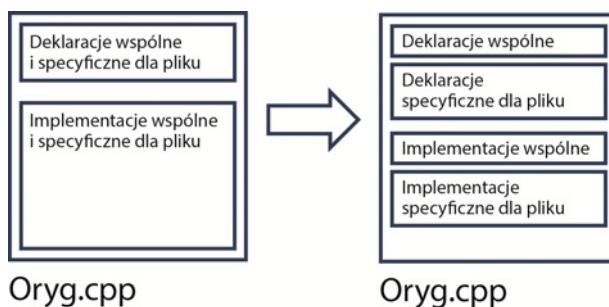
## Krok 1. Oddzielanie deklaracji od definicji

Jeżeli nie próbowałeś jeszcze rozdzielać kodu na wiele plików, prawdopodobnie nie masz odseparowanych deklaracji funkcji od ich definicji. Pierwszym krokiem będzie zatem zagwarantowanie, że wszystkie funkcje otrzymają swoje deklaracje i przeniesienie tych deklaracji na początek pliku. Proces ten można zilustrować następująco:



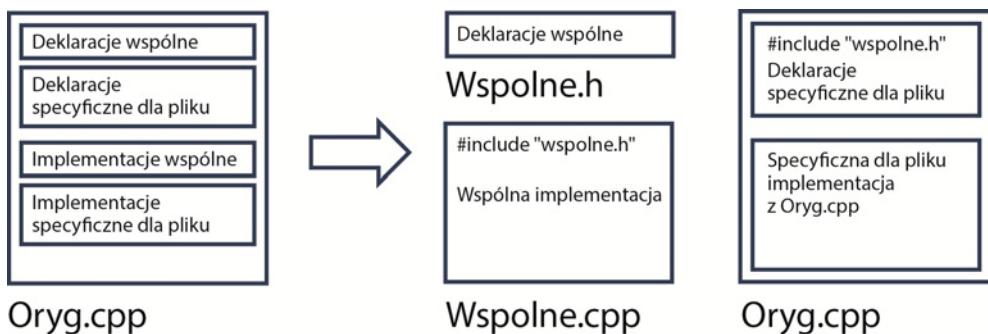
## Krok 2. Określenie, które funkcje powinny być wspólne

Teraz, kiedy odseparowałeś już deklaracje funkcji od ich definicji, możesz zastanowić się, które z nich są specyficzne dla tego pliku, a które powinny znaleźć się w pliku wspólnym.



## Krok 3. Przeniesienie wspólnych funkcji do nowych plików

Teraz możesz przenieść wspólne deklaracje do nowego pliku, *Wspolne.h*, a wspólną implementację do pliku *Wspolne.cpp* (zobacz rysunek na kolejnej stronie). Jednocześnie będziesz musiał dołączyć plik *Wspolne.h* z pliku *Oryg.cpp*. Nadal będziesz mógł wywoływać wspólne



funkcje, ponieważ wszystkie deklaracje znajdują się w pliku *Wspolne.h*. Potrzebna będzie jednak odpowiednia konfiguracja plików, dzięki czemu podczas budowy pliku *Oryg.cpp* nastąpi jego konsolidacja z plikiem *Wspolne.obj*, co szczegółowo omówię już za chwilę.

## Przykładowy program

Oto niewielki program zawierający ogólny kod z listą powiązaną. Tak się złożyło, że kod ten znalazł się w pliku *Oryg.cpp*. Popracujemy nad tym kodem i rozbijemy go na plik nagłówkowy oraz źródłowy, które będą nadawać się do wielokrotnego użycia.

### oryg.cpp

#### Przykładowy kod 52.: oryg.cpp

```
#include <iostream>

using namespace std;

struct Wezel
{
    Wezel *w_nastepny;
    int wartosc;
};

Wezel* dodajWezel (Wezel* w_lista, int wartosc)
{
    Wezel *w_nowy_wezel = new Wezel;
    w_nowy_wezel->wartosc = wartosc;
    w_nowy_wezel->w_nastepny = w_lista;

    return w_nowy_wezel;
}

void drukujListe (const Wezel* w_lista)
{
    const Wezel* w_biezacy_wezel = w_lista;
    while ( w_biezacy_wezel != NULL )
    {
        cout << w_biezacy_wezel->wartosc << endl;
        w_biezacy_wezel = w_biezacy_wezel->w_nastepny;
    }
}
```

```

}

int main ()
{
    Wezel *w_lista = NULL;
    for ( int i = 0; i < 10; ++i )
    {
        int wartosc;
        cout << "Podaj wartość węzła listy: ";
        cin >> wartosc;
        w_lista = dodajWezel( w_lista, wartosc );
    }
    drukujListe( w_lista );
}

```

Najpierw oddzielimy deklaracje od definicji. Ze względu na oszczędność miejsca pokażę same deklaracje; pozostała część pliku nie uległa zmianie.

## **oryg.cpp**

```

struct Wezel
{
    Wezel *w_nastepny;
    int wartosc;
};

Wezel* dodajWezel (Wezel* w_lista, int wartosc);
void drukujListe (const Wezel* w_lista);

```

Ponieważ w kodzie nie ma deklaracji specyficznych dla pliku, nie musimy ich odseparowywać. Od razu możemy umieścić wszystkie nasze deklaracje w nowym pliku nagłówkowym — *Wspolne.h* (w tym przypadku plik ten nazwiemy *listapowiazana.h*). Teraz pokażę te pliki w całości.

## **listapowiazana.h**

### **Przykładowy kod 53.: *listapowiazana.h***

```

struct Wezel
{
    Wezel *w_nastepny;
    int wartosc;
};

Wezel* dodajWezel (Wezel* w_lista, int wartosc);
void drukujListe (const Wezel* w_lista);

```

## **listapowiazana.cpp**

### **Przykładowy kod 54.: *listapowiazana.cpp***

```

#include <iostream>
#include "listapowiazana.h"

using namespace std;

```

```
Wezel* dodajWezel (Wezel* w_lista, int wartosc)
{
    Wezel *w_nowy_wezel = new Wezel;
    w_nowy_wezel->wartosc = wartosc;
    w_nowy_wezel->w_nastepny = w_lista;

    return w_nowy_wezel;
}

void drukujListe (const Wezel* w_lista)
{
    const Wezel* w_biezacy_wezel = w_lista;
    while ( w_biezacy_wezel != NULL )
    {
        cout << w_biezacy_wezel->wartosc << endl;
        w_biezacy_wezel = w_biezacy_wezel->w_nastepny;
    }
}
```

## oryg.cpp

### Przykładowy kod 55.: oryg\_nowy.cpp

```
#include <iostream>
#include "listapowiazana.h"

using namespace std;

int main ()
{
    Wezel *w_lista = NULL;
    for ( int i = 0; i < 10; ++i )
    {
        int wartosc;
        cout << "Podaj wartość węzła listy: ";
        cin >> wartosc;
        w_lista = dodajWezel( w_lista, wartosc );
    }
    drukujListe( w_lista );
}
```

Zauważ, że plik nagłówkowy nie powinien zawierać żadnych definicji funkcji. Gdybyśmy do pliku nagłówkowego dodali definicję funkcji, po czym dołączyli ten plik do kilku plików źródłowych, ta sama definicja pojawiłaby się kilkakrotnie podczas konsolidacji. Taka sytuacja wprowadziłaby w błąd i zezłościła konsolidator.

Musimy także zagwarantować, że deklaracje określonej funkcji nie wystąpią kilkakrotnie w tym samym pliku źródłowym. Istnieje możliwość, że plik *oryg.cpp* będzie zawierać więcej plików nagłówkowych i że jeden z tych plików także będzie dołączać plik *listapowiazana.h*:

## nowynaglowek.h

```
#include "listapowiazana.h"
// Dalszy kod
```

## oryg.cpp

```
#include "listapowiazana.h"
#include "nowynaglowek.h"
```

*/\* Dalsza część kodu w pliku oryg.cpp \*/*

Plik *oryg.cpp* dołącza plik nagłówkowy *listapowiazana.h* dwukrotnie — raz bezpośrednio i drugi raz pośrednio, poprzez dołączenie pliku *nowynaglowek.h*.

Poradzenie sobie z takim problemem wymaga użycia techniki znanej pod nazwą **strażnika include** (ang. *include guard*). W technice tej korzystamy z preprocesora w celu sprawdzenia, czy dany plik został już dołączony do kodu. Zasadniczy pomysł kryjący się za takim rozwiązaniem można wyrazić następująco:

```
if <jeszcze nie dołączliśmy danego pliku>
    <zaznaczmy, że go dołączliśmy>
    <dołączmy go>
```

Spokojnie możemy korzystać z powyższego wzorca, ponieważ nigdy nie powinniśmy znaleźć się w sytuacji, w której musielibyśmy kilkakrotnie dołączyć ten sam plik nagłówkowy.

Aby zaimplementować taki strażnik, musimy zastosować dyrektywę preprocesora `#ifndef`, z którą mieliśmy już do czynienia w tym rozdziale. Instrukcja ta oznacza: jeśli nie ma takiej definicji, dołącz blok kodu zakończony dyrektywą `#endif`.

```
#ifndef ORYG_H
// Zawartość nagłówka
#endif
```

Kod ten znaczy: jeśli nie ma definicji `ORYG_H`, dołącz poniższy kod, aż do dyrektywy `#endif`. Cała sztuczka polega na tym, że właśnie teraz możemy zdefiniować `ORYG_H`:

```
#ifndef ORYG_H
#define ORYG_H
// Zawartość nagłówka
#endif
```

Wyobraź sobie, co się stanie, kiedy ktoś dołączy ten plik dwa razy. Za pierwszym razem `ORYG_H` nie jest zdefiniowane, a zatem dyrektywa `#ifndef` dołączy pozostałą część pliku, razem z fragmentem definiującym `ORYG_H` (to fakt, `ORYG_H` jest definiowane jako puste, niemniej jest definiowane). Podczas kolejnego dołączenia pliku warunek `#ifndef` nie jest spełniony, w związku z czym nie zostanie dodany żaden kod.

Nie musisz wymyślać niepowtarzalnych nazw dla strażników `include`. Dobrym rozwiązaniem jest użycie nazwy pliku nagłówkowego z dopisany sufiksem `_H`. W ten sposób zagwarantujesz, że nazwy strażników nie będą się powtarzać i zmniejszysz prawdopodobieństwo wystąpienia konfliktu ze zdefiniowanymi przez kogoś innego wartościami lub strażnikami `include`<sup>5</sup>.

## Pozostałe zasady pracy z plikami nagłówkowymi

Nigdy nie dołączaj bezpośrednio pliku `.cpp`. Dołączenie takiego pliku wywoła tylko problemy, ponieważ kompilator skompiluje kopię każdej definicji funkcji z pliku `.cpp` do wszystkich plików obiektowych, a konsolidator zobaczy wielokrotne definicje tej samej funkcji. Coś takiego

<sup>5</sup> Jeśli zamierzasz udostępniać swój kod albo w szerokim zakresie korzystać z cudzego kodu, możesz także dodać w dyrektywie `define#` swoje imię albo nazwę firmy. W końcu ktoś inny także mógł utworzyć listę powiązaną.

nie powinno się wydarzyć. Nawet jeśli byś bardzo uważały, aby tak się nie stało, utraciłbyś korzyści związane z oszczędnością czasu, jakie daje oddzielna komplikacja.

Warto zwrócić szczególną uwagę na jeden z przypadków, kiedy ma zastosowanie reguła mówiąca, że należy mieć tylko jedną kopię danej funkcji: dowolna komplikacja, którą wykonujesz, powinna zawierać wyłącznie jeden plik źródłowy z funkcją `main`. Funkcja `main` jest punktem, od którego rozpoczyna się program, w związku z czym może istnieć tylko jedna jej wersja.

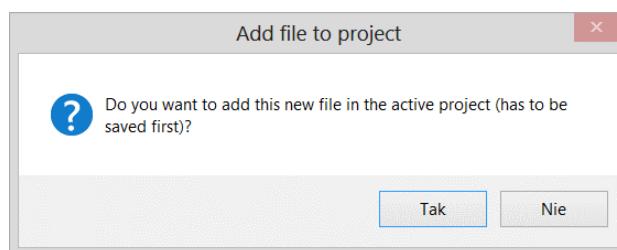
## Praca z wieloma plikami źródłowymi w środowisku programistycznym

Konfiguracja poprawnej konsolidacji wielu plików źródłowych zależy od Twojego środowiska programistycznego. Dla każdego ze środowisk pokażę, jak przeprowadzić taką konfigurację. Zaczynę od Code::Blocks.

### Code::Blocks

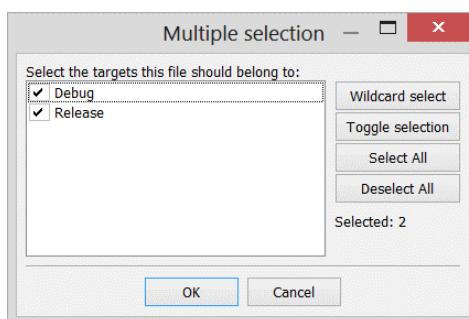
Aby w Code::Blocks dodać nowy plik źródłowy do bieżącego projektu, należy wybrać polecenie `File/New/Empty file`.

Zostaniesz zapytany, czy chcesz dodać nowy plik do bieżącego projektu.



Wybierz odpowiedź *Tak*.

Następnie będziesz musiał podać nazwę pliku. Po jej wpisaniu Code::Blocks zapyta Cię, jakie konfiguracje komplikacji będą potrzebne dla tego pliku. W przypadku plików źródłowych konfiguracje te określają, w jaki sposób Twój plik będzie dołączany na etapie konsolidacji.



Wybierz wszystkie dostępne opcje (zazwyczaj będą to *Debug* i *Release*). Chociaż raczej nigdy nie będziesz dołączać pliku nagłówkowego, możesz zaznaczyć te dwie opcje także dla takiego rodzaju pliku, ponieważ Code::Blocks jest na tyle mądry, że nie uwzględnii ich w opcjach konsolidacji.

Aby skorzystać z nowych plików, zwykle będziesz musiał dodać zarówno plik nagłówkowy, jak i źródłowy, po czym wprowadzić w pliku źródłowym zmiany, które omówiłem już wcześniej w tym rozdziale.

## g++

Jeśli używasz g++, nie będziesz musiał robić nic szczególnego, jak tylko z poziomu wiersza poleceń utworzyć pliki i nadać im nazwy. Jeśli na przykład masz pliki źródłowe *oryg.cpp*, *wspolne.cpp* i plik nagłówkowy *wspolne.h*, będziesz mógł skompilować oba pliki źródłowe za pomocą polecenia:

```
g++ oryg.cpp wspolne.cpp
```

Nie musisz podawać pliku nagłówkowego w wierszu polecen — zostanie on dołączony poprzez plik *.cpp*, w którym jest on potrzebny. Polecenie to spowoduje rekompilację wszystkich plików wymienionych w wierszu polecen. Jeśli chcesz skorzystać z zalet związanych z odrębną komplikacją, możesz skompilować każdy z tych plików oddzielnie, przy użyciu flagi *-c*:

```
g++ -c oryg.cpp  
g++ -c wspolne.cpp
```

Po czym je skonsolidować poleciением:

```
g++ oryg.o wspolne.o
```

lub po prostu:

```
g++ *.o
```

Jeśli tylko wiesz, że w bieżącym katalogu nie ma żadnych niepotrzebnych plików.

Ręczne sprawowanie kontroli nad oddzielną komplikacją jest dość żmudnym procesem. O wiele łatwiejsze jest przeprowadzanie komplikacji przy pomocy pliku **makefile**. Plik taki zawiera definicję procesu budowy i można w nim zakodować zależności między różnymi plikami źródłowymi, dzięki czemu w przypadku wprowadzenia zmian w jednym z plików źródłowych kompilator będzie wiedzieć, że należy zrekompilować pozostałe pliki, które zależą od zmodyfikowanego pliku.

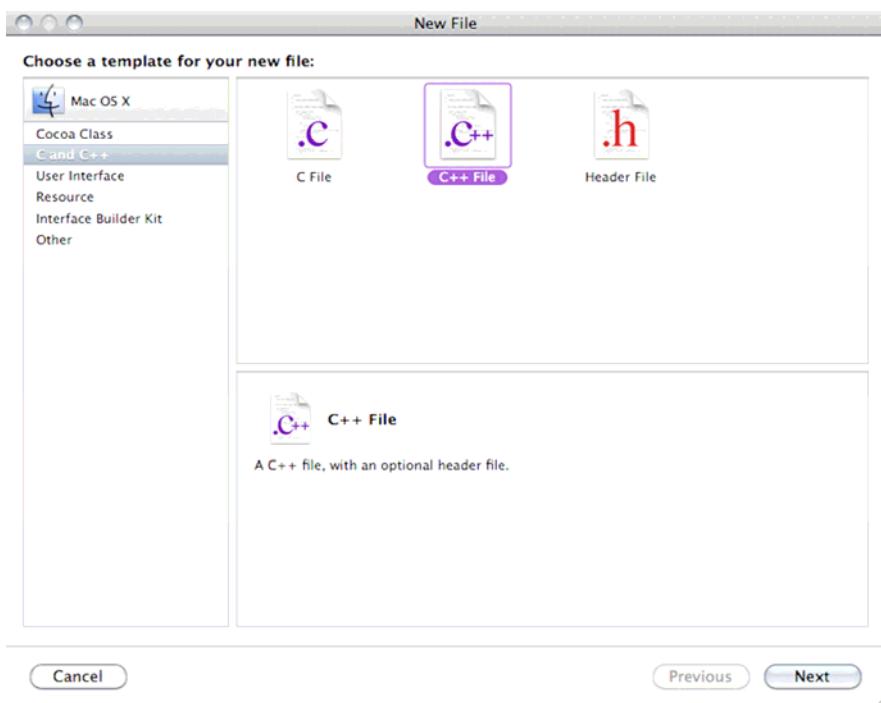
Omówienie plików makefile wykracza poza zakres tej książki.Więcej informacji na ten temat znajdziesz na stronie <http://www.cprogramming.com/tutorial/makefiles.html>. Jeśli jednak nie chcesz uczyć się o plikach makefile, możesz po prostu od razu kompilować swoje pliki C++ za pomocą polecenia:

```
g++ oryg.cpp wspolne.cpp
```

## Xcode

Aby w projekcie Xcode dodać nowy plik, wybierz polecenie *File/New File*. Jeśli chcesz mieć pewność, że nowe pliki będą widoczne w folderze *Sources* w drzewku z lewej strony okna, przed wybraniem polecenia *File/New File* wybierz katalog *Sources*, w którym znajduje się plik *main.cpp*. Nie jest to konieczne, ale dzięki temu Twoje pliki będą lepiej zorganizowane.

Po wybraniu polecenia *File/New* pojawi się kilka opcji związanych z typem pliku.



W sekcji z lewej strony wybierz *C and C++*, natomiast po prawej stronie wskaż *C++ file* (albo *Header file*, jeśli chcesz utworzyć plik nagłówkowy). Jeżeli chcesz dodać zarówno plik nagłówkowy, jak i plik implementacyjny cpp, wybierz opcję *C++ file*. Na następnym ekranie dostaniesz możliwość utworzenia pliku nagłówkowego. Kliknij przycisk *Next* (zobacz rysunek na kolejnej stronie).

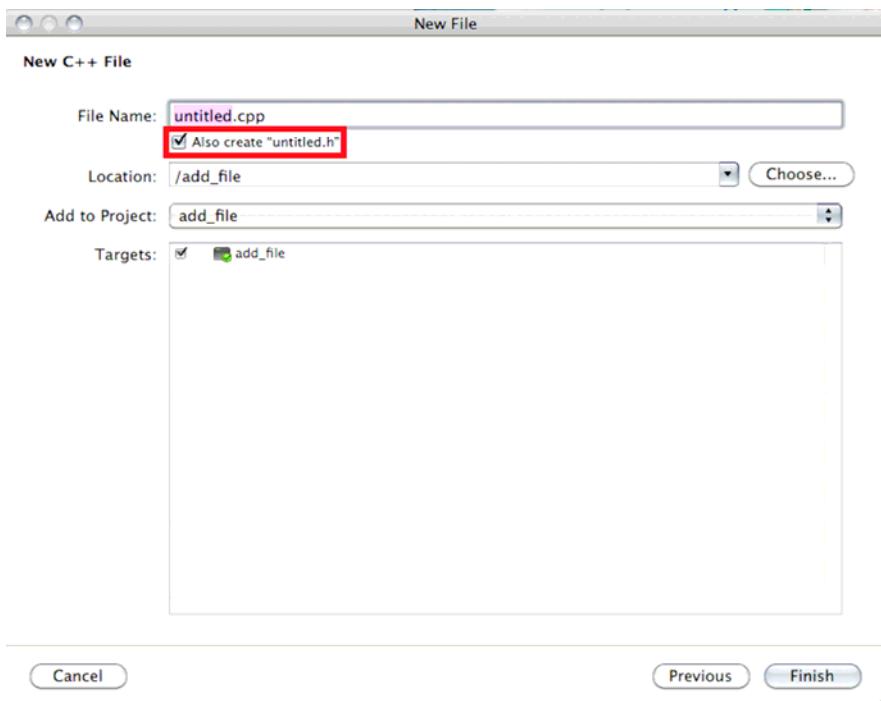
Podaj nazwę pliku oraz jego lokalizację, jeśli nie chcesz skorzystać z lokalizacji domyślnej. Możesz również zaakceptować ustawienia domyślne — w powyższym przykładzie dodaj plik bezpośrednio w katalogu *add\_file*, który jest skojarzony z projektem o nazwie *add\_file*.

Jeżeli wybierzesz tworzenie pliku C++, będziesz mógł utworzyć także plik nagłówkowy. Na powyższym rysunku otoczyłem ramką pole wyboru, które należy w takim przypadku zaznaczyć. Jeżeli wybierzesz tę opcję, plik nagłówkowy zostanie otworzony po kliknięciu przycisku *Finish*.

Xcode automatycznie skonfiguruje proces budowy w taki sposób, aby kompilować tworzone przez Ciebie pliki i konsolidować je z pozostałymi plikami.

## Sprawdź się

1. Który z poniższych etapów nie stanowi części procesu budowy oprogramowania w języku C++?
  - A. Konsolidacja.
  - B. Kompilacja.
  - C. Przetwarzanie wstępne.
  - D. Przetwarzanie końcowe.



- 2.** Kiedy wystąpi błąd związany z niezdefiniowaną funkcją?
- A. Na etapie konsolidacji.
  - B. Na etapie komplikacji.
  - C. Na początku działania programu.
  - D. Podczas wywołania funkcji.
- 3.** Co się może stać, gdy kilkakrotnie dołączysz plik nagłówkowy?
- A. Pojawią się błędy ostrzegające o powielonych deklaracjach.
  - B. Nic, pliki nagłówkowe zawsze są wczytywane tylko raz.
  - C. To zależy od implementacji pliku nagłówkowego.
  - D. Pliki nagłówkowe można dołączać jednorazowo tylko po jednym pliku źródłowym, tak więc nie stanowi to żadnego problemu.
- 4.** Jaka jest zaleta przeprowadzania komplikacji i konsolidacji na odrębnych etapach?
- A. Żadna. Takie rozwiązanie prowadzi do pomyłek i prawdopodobnie spowalnia cały proces, ponieważ trzeba uruchamiać jednocześnie kilka programów.
  - B. Takie rozwiązanie ułatwia diagnozowanie błędów, ponieważ wiesz, czy dany problem wykrył konsolidator czy kompilator.
  - C. Takie rozwiązanie umożliwia kompilowanie wyłącznie zmodyfikowanych plików, co skracą czas komplikacji i konsolidacji.
  - D. Takie rozwiązanie umożliwia kompilowanie wyłącznie zmodyfikowanych plików, co skracą czas komplikacji.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

1. Napisz program zawierający funkcje `dodaj`, `odejmij`, `pomnoz` i `podziel`. Każda z nich powinna przyjmować dwie liczby całkowite i zwracać wynik działania. Utwórz mały kalkulator korzystający z tych funkcji. Deklaracje funkcji umieść w pliku nagłówkowym, a ich kod źródłowy w pliku źródłowym.
2. W programie, który napisałeś w ramach zadania nr 1, przenieś wszystkie definicje funkcji do pliku źródłowego, który będzie niezależny od reszty kodu kalkulatora.
3. W programie z implementacją drzew binarnych, z którego korzystałeś w ramach ćwiczeń w rozdziale poświęconym drzewom binarnym, przenieś do pliku nagłówkowego wszystkie deklaracje funkcji oraz struktur. Umieść deklaracje struktur w jednym pliku, natomiast deklaracje funkcji w drugim pliku. Wszystkie implementacje przenieś do jednego pliku źródłowego. Napisz niewielki program umożliwiający korzystanie z podstawowych funkcjonalności drzewa binarnego.

# 22

## ■ ■ ■ ROZDZIAŁ 22

# Wprowadzenie do projektowania programów

---

Teraz, kiedy rozwiązałeś już problem fizycznego przechowywania kodu na dysku w taki sposób, który ułatwia edytowanie programów, gdy stają się one coraz większe, możemy skupić się na innym problemie — jak logicznie organizować kod, aby można go było łatwo poddawać edycji i z nim pracować. Zaczniemy od rozpatrzenia kilku z najczęściej spotykanych problemów, które pojawiają się wraz ze zwiększeniem się rozmiaru programów.

## Powielony kod

Chociaż przy okazji omawiania funkcji wspomniałem co nieco o problemie związanym z powielonym kodem, przyjrzyjmy się temu zagadnieniu jeszcze raz trochę dokładniej. Kiedy Twoje programy stają się coraz większe, pewne idee w nich zawarte zaczną się powtarzać. Na przykład w grze komputerowej potrzebny będzie kod, który kreśli na ekranie różne elementy graficzne (takie jak statek kosmiczny albo pocisk).

Zanim jednak narysujesz statek kosmiczny, potrzebna będzie najbardziej podstawowa możliwość związana z narysowaniem piksela. Piksela to pojedynczy punkt w określonym kolorze, umieszczony na ekranie przy użyciu dwuwymiarowych współrzędnych. W większości przypadków będziesz mógł znaleźć biblioteki graficzne realizujące tego typu zadania<sup>1</sup>.

Zapewne także chciałbyś, aby kod korzystający z pikseli (lub innych podstawowych elementów, takich jak linie albo okręgi, które udostępnia biblioteka) rysował także właściwe elementy gry — statki kosmiczne, pociski itd.

Takie rysowanie prawdopodobnie odbywało się w Twoim kodzie dość często; na pewno statek kosmiczny albo pocisk należy narysować od nowa, kiedy tylko zmieni on swoje miejsce. Gdybyś musiał umieszczać cały kod rysujący pocisk we wszystkich miejscach programu, w których należy go narysować, miałbyś bardzo dużo powielonego kodu.

Taka redundancja wprowadza do programu niepotrzebną złożoność, a złożoność znacznie utrudnia zrozumienie kodu. Z pewnością zamiast powtarzać dany proces w każdej części kodu, wolałbyś dysponować standardowymi sposobami wykonywania pewnych czynności, takich jak rysowanie statków kosmicznych albo pocisków. Dlaczego? Przypuśćmy, że chciałbyś coś

---

<sup>1</sup> W książce tej nie będziemy zajmować się grafiką, a więcej informacji na ten temat znajdziesz na stronie <http://www.cprogramming.com/graphics-programming.html>.

zmienić, na przykład kolor pocisku. Jeśli Twój kod wyświetla pocisk w dziesięciu różnych miejscach, musiałbyś zmodyfikować wszystkie te miejsca tylko po to, aby zmienić kolor. Co za męka!

Z każdym razem, kiedy chcesz wyświetlić pocisk, musisz jeszcze raz od początku wymyślać odpowiedni kod albo szukać przykładowych fragmentów realizujących to zadanie, po czym je kopiuować i wklejać, być może zmieniając przy tym nazwy niektórych zmiennych w celu uniknięcia konfliktów. W każdym z takich przypadków musisz zastanawiać się nad tym, jak narysować pocisk, zamiast po prostu wydać polecenie „narysuj pocisk”. Co więcej, kiedy powrócisz do swojego kodu i zaczniesz go czytać, będziesz musiał rozwiązać, co właściwie on robi. O wiele trudniej jest domyślić się, że polecenia

```
okrag(10, 10, 5 );
wypełnijOkrag(10, 10, CZERWONY );
```

rysują pocisk, niż wtedy gdy masz do czynienia z taką funkcją:

```
rysujPocisk( 10, 10 );
```

Funkcje nadają fragmentom kodu czytelne nazwy, dzięki czemu czytając program, możesz wiezieć, co tak naprawdę robi dany kod. Chociaż prawdopodobnie jeszcze tego nie doświadczyleś, w miarę tworzenia coraz większych programów będziesz poświęcać więcej czasu na czytanie kodu niż jego pisanie. Dobrze dobrane nazwy oraz przemyślane funkcje mogą zatem wywierać ogromny wpływ na komfort Twojej pracy.

## Założenia dotyczące przechowywania danych

Zaraza redundancji może zainfekować nie tylko Twoje algorytmy. Spójrzmy na kolejny przykład kodu, w którym ukrywa się redundancja. A gdybyś tak chciał napisać program szachowy, który bieżącą sytuację na szachownicy reprezentuje przy pomocy tablicy? Za każdym razem, gdy chcesz uzyskać dostęp do szachownicy, po prostu odwołujesz się do tablicy.

Aby zainicjalizować drugi wiersz zawierający wyłącznie białe piony, mógłbyś napisać:

```
enum BierkiSzachowe { BIALY_PION, BIALA_WIEZA /* i pozostałe bierki */ };
```

```
// ...mnóstwo kodu
```

```
for ( int i = 0; i < 8; i++ )
{
    szachownica[ i ][ 1 ] = BIALY_PION;
}
```

Później, jeśli zechcesz sprawdzić, jaka bierka znajduje się na danym polu, będziesz mógł sprawdzić wartość tablicy:

```
// ...mnóstwo kodu
if ( szachownica[ 0 ][ 0 ] == BIALY_PION )
{
    /* zrób coś */
}
```

Kiedy Twój program zacznie się rozrastać, w różnych jego miejscach pojawi się coraz więcej kodu korzystającego z szachownicy. Czy jest w tym coś złego? Przecież podczas odczytywania zawartości tablicy tak naprawdę niczego nie powtarzasz — jest to tylko jeden wiersz kodu, prawda? A jednak coś powtarzasz; wielokrotnie korzystasz z tej samej struktury danych. Używając w licznych miejscach programu tej samej struktury, przyjmujesz dość konkretne

założenie co do sposobu, w jaki reprezentowana jest szachownica. Nie powtarzasz algorytmu, ale za to powtarzasz założenie dotyczące reprezentacji danych. Pomyśl o tym następująco: fakt, iż tak się złożyło, że odczytanie szachownicy wymaga tylko jednego wiersza kodu, nie oznacza od razu, że zawsze tak będzie. Gdybyś zaimplementował szachownicę inaczej, mógłbyś potrzebować bardziej złożonej techniki umożliwiającej dostęp do szachownicy.

Zaawansowane programy szachowe nie korzystają w celu reprezentacji szachownicy z tablic, tylko ze złożonych plansz bitowych<sup>2</sup>. Każdorazowe odczytanie planszy bitowej wymaga więcej niż tylko jednej linii kodu. Gdybym pisał program szachowy, prawdopodobnie zacząłbym od tablicy, co pozwoliłoby mi skupić się na podstawowych algorytmach, zanim przystąpiłbym do optymalizacji kodu pod względem prędkości. Gdybym jednak w prosty sposób chciał zmienić reprezentację szachownicy, musiałbym ukryć tablicę. Jak mógłbym to zrobić?

Ostatnim razem, kiedy chcieliśmy coś ukryć, były to szczegółowe dotyczące rysowania pocisków. Zrealizowaliśmy nasz cel, tworząc funkcję, którą można wywoływać zamiast stosowania kodu rysującego pocisk na ekranie. Aby schować detale związane z reprezentacją szachownicy, także możemy zastosować funkcję. Zamiast bezpośrednio korzystać z tablicy, nasz kod mógłby wywoływać funkcję udostępniającą tablicę. Moglibyśmy na przykład napisać następującą funkcję pobierzBierke:

```
int pobierzBierke (int x, int y)
{
    return szachownica[ x ][ y ];
}
```

Zauważ, że funkcja ta przyjmuje dwa argumenty i zwraca wartość; dokładnie tak samo jak w przypadku odczytywania tablicy. Tak naprawdę nie zaoszczędzimy dzięki niej na pisaniu, gdyż potrzebne są dokładnie takie same dane wejściowe jak poprzednio — współrzędne *x* oraz *y*. Cała różnica polega na tym, że teraz sposób odczytywania tablicy został ukryty w jednej funkcji. Reszta programu może (i powinna) korzystać z tej funkcji w celu pobierania danych z tablicy. Jeśli później zdecydujesz się zmienić reprezentację tablicy, będziesz mógł zmodyfikować tylko tę jedną funkcję, a cała reszta kodu będzie po prostu działać<sup>3</sup>.

Rozwiążanie polegające na użyciu funkcji w celu ukrycia szczegółów czasami jest nazywane **abstrakcją funkcyjną**. Stosowanie abstrakcji funkcyjnej oznacza, że wszystkie powtarzalne operacje należy przenieść do funkcji — to funkcja powinna określać, jakie dane wejściowe pobiera oraz jakie dane wyjściowe przekazuje obiektom, które ją wywołują, i jednocześnie nic nie mówić im o tym, JAK jest zaimplementowana. To JAK może oznaczać zarówno zastosowany algorytm, jak i użyte struktury danych. Funkcja taka pozwala obiektom, które ją wywołują, korzystać z przewagi, jaką daje obietnica dotycząca działania jej interfejsu, bez konieczności posiadania przez nie wiedzy o tym, jak jest ona zaimplementowana.

Użycie funkcji w celu ukrycia danych i algorytmów ma wiele zalet:

1. Ułatwisz sobie życie w przyszłości. Zamiast być zmuszonym do pamiętania implementacji algorytmu, możesz po prostu skorzystać z napisanej wcześniej funkcji. Jeśli tylko ufasz, że działa ona prawidłowo dla wszystkich danych wejściowych, możesz zaufać także danym wyjściowym bez konieczności pamiętania, jak funkcja ta działa.

<sup>2</sup> <http://pclab.pl/art34801-5.html>

<sup>3</sup> Oczywiście przy założeniu, że dane z szachownicy będą konsekwentnie odczytywane przy użyciu tej funkcji. Prawdę pozostaje również to, że być może potrzebnych będzie kilka dodatkowych funkcji ustawiących bierki na planszy, niemniej lepiej jest zmodyfikować dwie funkcje niż kilkadziesiąt lub kilkaset.

2. Kiedy już ufasz, że funkcja „po prostu działa”, będziesz mógł przystąpić do rozwiązywania problemów, pisząc kod, który wielokrotnie z niej korzysta. Nie musisz przejmować się żadnymi szczegółami (na przykład tym, jak uzyskać dostęp do szachownicy), dzięki czemu będziesz mógł skupić się na problemach, które należy rozwiązać (na przykład na implementacji sztucznej inteligencji).
3. Jeśli mimo wszystko **odnajdziesz** błąd w implementacji logiki funkcji, przyjdzie Ci zmienić tylko jedną funkcję, bez konieczności modyfikowania licznych miejsc w całym kodzie programu.
4. Jeśli piszesz funkcje ukrywające struktury danych, dajesz sobie większą elastyczność w doborze sposobu przechowywania i reprezentacji danych. Możesz zacząć od mało efektywnych metod, które są proste do napisania, i później — gdy zajdzie taka potrzeba — zastąpić je wydajniejszymi implementacjami, bez konieczności modyfikowania jakichkolwiek innych elementów.

## Projekt i komentarze

Kiedy tworzysz dobrze zaprojektowane funkcje, powinieneś także je dokumentować. Poprawne udokumentowanie funkcji nie jest jednak tak proste, jak może się wydawać.

Dobre komentarze odpowiadają na pytania, które mógłby zadać czytelnik kodu. Takie komentarze, jak te, które wstawiłem w przykładach zamieszczonych w książce:

```
// Deklaracja zmiennej i jej inicjalizacja wartością 3  
int i = 3;
```

nie są tak naprawdę komentarzami, które powinieneś pisać! Tego typu komentarze mają na celu udzielanie odpowiedzi na pytania Czytelników, którzy dopiero rozpoczęli naukę programowania. W rzeczywistości osoby czytające Twoje programy będą znać język C++.

Co gorsza, wraz z upływem czasu komentarze dezaktualizują się, tak więc ktoś, kto czyta komentarz, nie tylko może zmarnować swój czas, ale także błędnie zrozumieć, co się dzieje w kodzie.

O wiele lepsze jest pisanie komentarzy odpowiadających na takie pytania jak: „Hej, co za dziwne rozwiązanie. Dlaczego zrobili tu coś takiego?” albo: „Jakie dopuszczalne wartości może przyjmować ta funkcja i co one znaczą?”. Oto przykład dokumentowania kodu w stylu, do którego powinieneś dążyć, tworząc własne funkcje:

```
/*  
 * Oblicz wartość ciągu Fibonacciego dla każdej dodatniej liczby całkowitej n.  
 * Jeśli wartość n jest mniejsza od 1, funkcja zwraca wartość 1  
 */  
int fibonacci (int n);
```

Zauważ, że w komentarzu do funkcji dokładnie opisano, co funkcja robi, jakie jej argumenty są prawidłowe i co się stanie, jeśli zostanie podany argument niepoprawny. Taki rodzaj dokumentacji sprawia, że użytkownik funkcji nie musi sprawdzać, jak została ona zaimplementowana — jest to bardzo dobre podejście!

Dobra dokumentacja programu niekoniecznie musi być obfita. Nie musisz komentować każdego wiersza kodu. Osobiście zawsze dołączam dokumentację do moich funkcji, które mają być używane poza jednym plikiem, a także dodaję komentarze za każdym razem, gdy kod jest skomplikowany albo wydaje się niezwykły.

Istnieje pewien zły nawyk wielu programistów wynikający z wypaczenia idei minimalnego komentowania — pisanie komentarzy jest czasem odkładane na sam koniec cyklu rozwoju oprogramowania. Kiedy kod zostanie już napisany, robi się zbyt późno, aby powrócić i umieścić przy nim sensowny komentarz. Wszystko, co można wówczas zrobić, sprowadza się do zamieszczania informacji, które tak czy inaczej można odczytać z samego kodu. Komentarze będą najbardziej przydatne, kiedy są zamieszczane podczas pisania kodu.

## Sprawdź się

- 1.** Jaka jest zaleta stosowania funkcji w porównaniu z bezpośrednim dostępem do danych?
  - A. Kompilator może optymalizować funkcję w celu zapewnienia szybszego dostępu do danych.
  - B. Funkcja może ukrywać swoją implementację przed obiektami ją wywołującymi, co upraszcza modyfikowanie tych obiektów.
  - C. Użycie funkcji to jedyny sposób na użycie tej samej struktury danych w wielu różnych plikach z kodem źródłowym.
  - D. Nie ma żadnych zalet.
- 2.** Kiedy kod należy przenieść do wspólnej funkcji?
  - A. Kiedy tylko zachodzi potrzeba jej wywołania.
  - B. Kiedy zacząłeś wywoływać ten sam kod w kilku różnych miejscach programu.
  - C. Kiedy kompilator zaczyna narzekać, że funkcje są zbyt duże, aby je skompilować.
  - D. Prawidłowe są odpowiedzi B i C.
- 3.** Dlaczego ukrywa się sposób reprezentacji struktury danych?
  - A. Aby łatwiej ją było zastąpić inną strukturą.
  - B. Aby kod, który korzysta ze struktury danych, był łatwiejszy do zrozumienia.
  - C. Aby prostsze było użycie struktury danych w nowych fragmentach kodu.
  - D. Wszystkie powyższe odpowiedzi są prawidłowe.

Odpowiedzi znajdują się na końcu książki.



# 23

## ROZDZIAŁ 23

### Ukrywanie reprezentacji struktur danych

---

Do tej pory miałeś do czynienia z ukrywaniem danych znajdujących się w zmiennych globalnych albo tablicach. Ukrywanie danych nie ogranicza się wyłącznie do tych kilku przykładów. Jeden z najczęściej spotykanych przypadków, gdy będzie Ci zależeć na schowaniu danych, ma miejsce podczas tworzenia struktury. Być może uznasz ten fakt za dziwny; w końcu struktura ma ściśle określony układ oraz zbiór wartości, które można w niej przechowywać. Jeśli spojrzyś na nią w taki właśnie sposób — jak na grupę pól — wówczas struktura rzeczywiście nie daje możliwości ukrycia szczegółów swojej implementacji (czyli tego, jakie zawiera pola i jaki mają one format). Możesz nawet zadać pytanie: „Czy jedynym celem struktury nie jest udostępnianie pewnego rodzaju danych? Dlaczego mielibyśmy ukrywać ich reprezentację?”. Okazuje się jednak, że o strukturach można myśleć inaczej; istnieją takie ich zastosowania, przy których wolałbyś właśnie tak postąpić.

W większości przypadków, gdy masz do czynienia ze zbiorem wzajemnie powiązanych informacji, o wiele ważniejsze jest to, co można z nimi zrobić, niż to, jak są one przechowywane. Jest to bardzo ważny punkt widzenia, który może zmienić sposób, w jaki myślisz. Powtórzę zatem, i to przy użyciu pogrubionej czcionki: **nie jest ważne, jak przechowujesz dane, ale co z nimi robisz.**

Ponieważ pogrubiona czcionka nie zawsze przyspiesza zrozumienie, weźmy pod uwagę prosty przykład — łańcuch tekstowy. Tak naprawdę nie ma znaczenia, jak przechowywane są łańcuchy tekstowe, o ile tylko nie implementujesz klasy `string`. W dowolnym kodzie pracującym z łańcuchami tekstowymi ważny jest sposób odczytywania długości łańcucha, odczytywanie i zapisywanie jego poszczególnych elementów albo wyświetlanie łańcuchów tekstowych na ekranie. W implementacji łańcucha tekstowego mogą być wykorzystane tablice znakowe i zmienna przechowująca jego długość, listy powiązane albo jeszcze inne funkcjonalności języka C++, o których jeszcze nie słyszałeś.

Najistotniejsze jest, że jako użytkownik typu `string` nie musisz wiedzieć, jak jest on zaimplementowany. Naprawdę ważne jest tylko to, co możesz z łańcuchami tekstowymi robić. Takich rzeczy może być mnóstwo, ale nawet C++ udostępnia mniej więcej 35 różnych funkcji związanych z łańcuchami, i w większości przypadków wielu z nich nie będziesz w ogóle potrzebować.

Często natomiast będzie Ci potrzebna możliwość tworzenia nowych typów danych bez konieczności ujawniania struktury użytej do ich implementacji. Na przykład podczas tworzenia łańcucha tekstowego nie musisz myśleć o buforze przechowującym poszczególne znaki. Wektory

STL oraz mapy działają na podobnej zasadzie — aby z nich korzystać, nie musisz wiedzieć, jak zostały zaimplementowane. Również dobrze ich implementacja mogłaby sprowadzać się do kar-mienia marchwią nadpobudliwych królików ze zdolnościami organizacyjnymi.

## Użycie funkcji w celu ukrycia układu struktury

W celu ukrycia pól struktury możesz utworzyć funkcje z nią związane. Wyobraź sobie na przykład niewielką strukturę reprezentującą szachownicę oraz pamiętającą, do kogo należy kolejny ruch (gracz biały albo czarny). Do przechowywania bierek oraz koloru gracza użyjemy wyliczeń:

```
enum BierkaSzachowa { PUSTE_POLE, BIALY_PION /* i pozostałe bierki */ };
enum KolorGracza { KG_BIALY, KG_CZARNY };

struct Szachownica
{
    BierkaSzachowa plansza[ 8 ][ 8 ];
    KolorGracza czy_j_ruch;
}
```

Möesz także napisać pozostałe funkcje obsługujące planszę i pobierające szachownicę jako parametr.

```
BierkaSzachowa pobierzBierke (const Szachownica *w_plansza, int x, int y)
{
    return w_plansza->plansza[ x ][ y ];
}

KolorGracza pobierzRuch (const Szachownica *w_plansza)
{
    return w_plansza->czy_j_ruch;
}

void wykonajRuch (Szachownica* w_plansza, int z_x, int z_y, int na_x, int na_y)
{
    // W rzeczywistości umieścilibyśmy tu jeszcze kod
    // weryfikujący dopuszczalność ruchu
    w_plansza->plansza[ na_x ][ na_y ] = w_plansza->plansza[ z_x ][ z_y ];
    w_plansza->plansza[ z_x ][ z_y ] = PUSTE_POLE;
}
```

Möesz z nich korzystać tak samo jak z wszystkich innych funkcji:

```
Szachownica b;
// Najpierw potrzebny jest kod inicjalizujący szachownicę.
// Teraz możemy z niej korzystać:
pobierzRuch( & b );
```

```
wykonajRuch( & b, 0, 0, 1, 0 ); // Przesuń bierkę z 0, 0 na 1, 0
```

Takie rozwiązanie jest całkiem dobre, a programiści piszący w C korzystają z niego od lat. Z drugiej jednak strony wszystkie te funkcje są skojarzone ze strukturą Szachownica tylko dlatego, że pobierają ją jako argument. Nigdzie nie jest jasno powiedziane, że „Funkcja ta powinna być uważana za nieodłączną część struktury”. Czy nie byłoby lepiej, gdyby struktura mogła zawierać nie tylko dane, ale także funkcje, które je przetwarzają?

W C++ wzięto sobie ten pomysł do serca i wbudowano go bezpośrednio w język. Aby można było z niego korzystać, w C++ wprowadzono ideę metod. **Metoda** jest funkcją zadeklarowaną

jako część struktury (miałeś już do czynienia z metodami w rozdziale poświęconym STL). W przeciwnieństwie do zwykłych funkcji, które nie są powiązane ze strukturami, metody z łatwością mogą operować na danych zapisanych w strukturach. Autor metody deklaruje ją jako część struktury, co bezpośrednio wiąże ją z tą strukturą. Kiedy metoda zostanie zadeklarowana jako część struktury, obiekt wywołujący tę metodę nie będzie musiał przekazywać tej struktury jako odrębnego argumentu! Takie wywołanie wymaga jednak specjalnej składni.

## Deklaracja metody i składnia wywołania

Zobaczmy, co uzyskamy, jeśli przekształcimy nasze funkcje w metody.

### Przykładowy kod 56.: metoda.cpp

```
enum BierkaSzachowa { PUSTE_POLE, BIALY_PION /* i pozostałe bierki */ };
enum KolorGracza { KG_BIALY, KG_CZARNY };
struct Szachownica
{
    BierkaSzachowa plansza[ 8 ][ 8 ];
    KolorGracza czyj_ruch;

    BierkaSzachowa pobierzBierke ( int x, int y )
    {
        return plansza[ x ][ y ];
    }

    KolorGracza pobierzRuch ()
    {
        return czyj_ruch;
    }

    void wykonajRuch ( int z_x, int z_y, int na_x, int na_y )
    {
        // W rzeczywistości umieścilibyśmy tu jeszcze kod
        // weryfikujący dopuszczalność ruchu
        plansza[ na_x ][ na_y ] = plansza[ z_x ][ z_y ];
        plansza[ z_x ][ z_y ] = PUSTE_POLE;
    }
};
```

Zwróć uwagę przede wszystkim na to, że metody zostały zadeklarowane wewnętrz struktury. Dzięki temu oczywiste jest, że powinny one być uważane za nieodłączną część tej struktury.

Co więcej, deklaracje metod nie przyjmują typu Szachownica jako odrębnego argumentu — wewnętrz metody ogólnie dostępne są wszystkie pola danej struktury. Zapis `plansza[ x ][ y ]` bezpośrednio udostępnia planszę w strukturze, dla której metoda została wywołana. Skąd jednak kod wie, z którą instancją struktury powinien pracować? Przecież możemy mieć więcej instancji struktury Szachownica.

Wywołanie metody wygląda następująco:

```
Szachownica b;
// Kod inicjalizujący plansze
b.pobierzRuch();
```

Wywołanie funkcji skojarzonej ze strukturą wygląda niemal dokładnie tak samo jak odczytanie pola tej struktury.

Wewnętrznie kompilator obsługuje szczegółowe związki z udostępnieniem danych w strukturze, dla której metoda ta została wywołana. Pojęciowo składnia <zmienna>.<metoda> stanowi skrócony zapis przekazujący zmienną <zmienna> do metody <metoda>. Teraz już wiesz, dlaczego składnia ta była nam potrzebna w rozdziale poświęconym STL — opisane tam funkcje działają według takich samych zasad jak metody.

## Przeniesienie definicji funkcji poza strukturę

Umieszczenie ciał wszystkich metod w strukturze może ją zabałaganić i utrudnić jej zrozumienie. Na szczęście istnieje możliwość rozbicia metod na deklaracje, które znajdują się wewnątrz struktury, oraz definicje umieszczone poza tą strukturą. Oto przykład takiego rozwiązania:

```
enum BierkaSzachowa { PUSTE_POLE, BIALY_PION /* i pozostałe bierki */ };
enum KolorGracza { KG_BIALY, KG_CZARNY };
struct Szachownica
{
    BierkaSzachowa plansza[ 8 ][ 8 ];
    KolorGracza czyj_ruch;

    // Deklaracje metod wewnątrz struktury
    BierkaSzachowa pobierzBierke (int x, int y);
    KolorGracza pobierzRuch ();
    void wykonajRuch (int z_x, int z_y, int na_x, int na_y);
};
```

Deklaracje metod znajdują się wewnątrz struktury i oprócz tego wyglądają jak zwykłe prototypy funkcji.

Definicje metod muszą jeszcze zostać w jakiś sposób powiązane ze swoją strukturą. Można to zrobić za pomocą specjalnej składni „zasięgowej”, która definiuje, że dana metoda należy do pewnej struktury. Składnia ta polega na zapisaniu nazwy metody w postaci <nazwa\_structury>::<nazwa metody>, a poza tym kod metody pozostaje taki sam:

```
BierkaSzachowa Szachownica::pobierzBierke (int x, int y)
{
    return plansza[ x ][ y ];
}

KolorGracza Szachownica::pobierzRuch ()
{
    return czyj_ruch;
}

void Szachownica::wykonajRuch (int z_x, int z_y, int na_x, int na_y)
{
    // W rzeczywistości umieścilibyśmy tu jeszcze kod
    // weryfikujący dopuszczalność ruchu
    plansza[ na_x ][ na_y ] = plansza[ z_x ][ z_y ];
    plansza[ z_x ][ z_y ] = PUSTE_POLE;
}
```

W dalszej części tej książki będę rozdzielać deklaracje od definicji w przypadku funkcji dłuższych niż kilka wierszy. Niektórzy praktycy zalecają, aby **nigdy** nie definiować metod wewnątrz struktur, gdyż w ten sposób ujawnia się więcej szczegółów o implementacji metod, niż jest to konieczne. Im więcej informacji na temat implementacji metody zdradzisz, tym większe

jest prawdopodobieństwo, że zamiast kodu polegającego na jej interfejsie, ktoś utworzy kod uzależniony właśnie od tych szczegółów. W dalszej części tej książki będę od czasu do czasu umieszczać definicje metod w obrębie klas w celu zaoszczędzenia miejsca.

## Sprawdź się

- 1.** Dlaczego warto korzystać z metod zamiast bezpośrednio z pól struktury?
  - A. Ponieważ metody są łatwiejsze do czytania.
  - B. Ponieważ metody są szybsze.
  - C. Nie należy tego robić. Zawsze należy korzystać bezpośrednio z pól.
  - D. Dzięki temu można modyfikować sposób reprezentacji danych.
- 2.** Który z poniższych zapisów definiuje metodę skojarzoną ze strukturą struct MojaStrukt  
(int funk(); );?
  - A. int funk() { return 1; }
  - B. MojaStrukt::int funk() { return 1; }
  - C. int MojaStrukt::funk(){ return 1; }
  - D. int MojaStrukt funk(){ return 1; }
- 3.** Dlaczego umieściłbyś definicję metody wewnętrz klasy?
  - A. Żeby użytkownicy klasy wiedzieli, jak ona działa.
  - B. Ponieważ takie rozwiązanie zawsze przyspiesza działanie kodu.
  - C. Nie należy tego robić! W ten sposób wyciekną wszystkie szczegółowe implementacje tej metody.
  - D. Nie należy tego robić, ponieważ spowolni to działanie programu.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

1. Napisz strukturę udostępniającą interfejs planszy do gry w kółko i krzyżyk. Zaimportuj grę w kółko i krzyżyk dla dwóch graczy z metodami umieszczonymi w tej strukturze. Postaraj się, aby podstawowe operacje, takie jak wykonanie ruchu przez gracza i sprawdzenie, czy któryś z uczestników wygrał, stanowiły część interfejsu napisanej przez Ciebie struktury.



# 24

## ■ ■ ■ ROZDZIAŁ 24

### Klasa

---

Kiedy Bjarne Stroustrup tworzył język C++, chciał wesprzeć ideę tworzenia struktur definiowanych raczej przez funkcje, które są w nich udostępnione, niż przez dane, które zostały użyte w celu ich implementacji. Mógł zrobić wszystko, na co tylko miał ochotę, rozszerzając istniejącą już koncepcję struktury, ale zamiast tego zdecydował się na utworzenie zupełnie nowego bytu, którym jest **klasa**.

Klasa przypomina strukturę, z tym że jest wzbogacona o możliwość określenia, które metody i dane należą do wewnętrznej implementacji klasy, a które z metod są przeznaczone dla jej użytkowników. Możesz uważać, że słowo „klasa” ma podobne znaczenie do słowa „kategoria”. Kiedy definiujesz klasę, tworzysz całkowicie nową kategorię lub rzecz. Klasa nie nasuwa już skojarzeń z danymi, które zostały ustrukturyzowane — jest ona definiowana przez metody, które udostępnia w ramach swojego interfejsu. Klasy mogą również nie pozwalać na przypadkowe korzystanie ze szczegółów ich implementacji.

Zgadza się, w C++ istnieje możliwość zabronienia metodom, które nie należą do danej klasy, korzystania z wewnętrznych danych tej klasy. Tak naprawdę, kiedy definiujesz klasę, nic nikomu nie zostanie domyślnie udostępnione, z wyjątkiem metod, które należą do tej klasy! Musisz jawnie zdecydować, co będzie publicznie dostępne. Możliwość zabronienia sięgania spoza klasy po dane pozwala kompilatorowi sprawdzać, czy programiści nie korzystają z danych, których nie powinni używać. Takie rozwiążanie jest niczym dar boży ułatwiający pielęgnację oprogramowania. Możesz zmieniać podstawowe elementy swojej klasy, na przykład sposób przechowywania szachownicy, nie obawiając się przy tym, że popsujesz kod istniejący poza tą klasą.

Nawet jeśli jesteś jedynym programistą pracującym nad projektem, gwarancja, że nikt nie „oszukuje” i nie zagląda do wnętrza metody, stanowi bardzo miłą okoliczność. W rzeczywistości jest to kolejny powód, dla którego metody są przydatne — jak już niedługo stwierdzisz, wyłącznie metody mają dostęp do „wewnętrznych” danych.

Od tej pory będę korzystał z klas, kiedy tylko zechcę ukryć sposób przechowywania danych, natomiast ze struktur, kiedy nie będę miał żadnych powodów do ukrywania ich implementacji. Możesz być zaskoczony, jak rzadko używa się struktur — ukrywanie danych jest bardzo przydatną możliwością. Ze starych dobrych struktur korzystam praktycznie tylko wtedy, gdy implementuję klasę i potrzebna mi jest jakaś pomocnicza struktura, w której będzie przechowywana część danych. Ponieważ struktura ta jest specyficzna wyłącznie dla tej klasy i nie udostępniam jej publicznie, zwykle nie potrzebuję robić z niej pełnowymiarowej klasy. Jak już powiedziałem, nie istnieje żadna żelazna reguła nakazująca tak robić, niemniej powszechnie przyjęto postępować w taki właśnie sposób.

# Ukrywanie sposobu przechowywania danych

Zabierzmy się za zgłębianie składni służącej do ukrywania danych przy pomocy klas. W jaki sposób wykorzystać klasę w celu ukrycia pewnych danych i jednocześnie udostępnić wszystkim wybrane metody? Otóż klasy pozwalają na określenie każdej metody i pola (często nazywanych **składowymi** klasą) jako publicznych lub prywatnych. Składowe publiczne są ogólnie dostępne, natomiast składowe prywatne są dostępne wyłącznie dla innych składowych danej klasy<sup>1</sup>.

Oto przykładowa deklaracja metody publicznej oraz pozostałych danych prywatnych:

## Przykładowy kod 57.: *klasa.cpp*

```
enum BierkaSzachowa { PUSTE_POLE, BIALY_PION /* i pozostałe bierki */ };
enum KolorGracza { KG_BIALY, KG_CZARNY };

class Szachownica
{
public:
    BierkaSzachowa pobierzBierke (int x, int y);
    KolorGracza pobierzRuch ();
    void wykonajRuch (int z_x, int z_y, int na_x, int na_y);

private:
    BierkaSzachowa _plansza[ 8 ][ 8 ];
    KolorGracza _czyj_ruch;
};

// Definicje metod pozostają dokładnie takie same!
BierkaSzachowa Szachownica::pobierzBierke (int x, int y)
{
    return _plansza[ x ][ y ];
}

KolorGracza Szachownica::pobierzRuch ()
{
    return _czyj_ruch;
}

void Szachownica::wykonajRuch (int z_x, int z_y, int na_x, int na_y)
{
    // W rzeczywistości umieścilibyśmy tu jeszcze kod
    // weryfikujący dopuszczalność ruchu
    _plansza[ na_x ][ na_y ] = _plansza[ z_x ][ z_y ];
    _plansza[ z_x ][ z_y ] = PUSTE_POLE;
}
```

Zwróci uwagę, że powyższa deklaracja klasy bardzo przypomina deklarację struktury z poprzedniego rozdziału, z jedną zasadniczą różnicą. Użyłem dwóch nowych słów kluczowych: **public** i **private**. Wszystkie składowe zadeklarowane po słowie kluczowym **public** są dostępne dla każdego, kto będzie z nich korzystać (w tym przypadku są to metody `pobierzBierke`, `pobierzRuch` i `wykonajRuch`). Wszystkie składowe w sekcji zaczynającej się od słowa **private** są dostępne

<sup>1</sup> Istnieje jeszcze trzeci rodzaj — składowe chronione, które omówię później.

wyłącznie dla metod zaimplementowanych jako część klasy Szachownica (są to tablica \_plansza i zmienna \_czyj\_ruch)<sup>2</sup>.

Składowe publiczne i prywatne możesz deklarować w dowolnej kolejności. Następująca deklaracja klasy upublicznia dokładnie takie same elementy:

```
class Szachownica
{
public:
    BierkaSzachowa pobierzBierke (int x, int y);

private:
    BierkaSzachowa _plansza[ 8 ][ 8 ];
    KolorGracza _czyj_ruch;

public:
    KolorGracza pobierzRuch ();
    void wykonajRuch (int z_x, int z_y, int na_x, int na_y);
};
```

Pisząc kod, zawsze rozpoczynam od sekcji publicznej, po której następuje sekcja prywatna. W ten sposób podkreślam fakt, że sekcja publiczna jest przeznaczona dla wszystkich użytkowników klasy (innych programistów), ponieważ jest ona pierwszą rzeczą, którą zobaczy użytkownik danej klasy<sup>3</sup>.

## Deklarowanie instancji klasy

Deklarowanie instancji klasy bardzo przypomina deklarowanie instancji struktury:

```
Szachownica b;
```

Wywołanie metody w klasie również wygląda identycznie:

```
b.pobierzRuch();
```

Istnieje jednak pewna drobna różnica w terminologii. Kiedy deklarujesz nową zmienną określonej klasy, zmienna ta jest na ogół nazywana **obiektem**. Słowo „obiekt” powinno przywoływać na myśl rzeczywiste obiekty w rodzaju kierownicy w samochodzie — coś, co ujawnia dość wąski interfejs i ukrywa sporą złożoność. Jeśli chcesz skierować samochód w lewo, kręcisz kierownicą w lewo i nie zastanawiasz się, jak działają poszczególne mechanizmy. Jedyne, co musisz zrobić, to obrócić kierownicę i być może przyspieszyć. Wszystkie szczegóły są ukryte za prostym interfejsem użytkownika. W języku C++ wszelkie szczegóły implementacji obiektu są schowane za zestawem publicznych wywołań funkcji. Funkcje te składają się na „interfejs użytkownika”. Kiedy już zdefiniujesz interfejs, Twoja klasa może go implementować w całkowicie dowolny sposób — wyłącznie od Ciebie zależy, jak będą reprezentowane dane i jak zaimplementujesz metody.

<sup>2</sup> Nazwę każdej składowej prywatnej poprzedziłem znakiem podkreślenia, aby ułatwić zidentyfikowanie danego elementu jako prywatnego, chociaż zabieg ten nie jest wymagany w C++. Początkowo nazwy takie mogą wydawać się brzydkie, ale w znacznym stopniu ułatwiają one czytanie kodu! Jeśli przyjmiesz tę konwencję, pamiętaj, aby po znaku podkreślenia nie stosować wielkich liter — taki prefiks może wywoływać konflikty w przypadku niektórych kompilatorów. Jeśli tylko podczas deklarowania prywatnych pól i metod po znaku podkreślenia będziesz umieszczać małe litery, wszystko będzie w porządku.

<sup>3</sup> Tym użytkownikami są, rzecz jasna, inni programiści, a nie końcowi użytkownicy programu. W wielu przypadkach sam będziesz przyszłym użytkownikiem swojej klasy.

## Odpowiedzialności klasy

Za każdym razem, gdy tworzysz klasę w C++, pomyśl o tym, jakby to było tworzenie nowego rodzaju zmiennej — nowego typu danych. Twój nowy typ jest dokładnie taki sam jak `int` albo `char`, z tym że ma o wiele większe możliwości. Z taką ideą miałeś już do czynienia. W C++ `string` jest klasą i w rzeczywistości stanowi ona nowy typ danych, z którego możesz korzystać. Koncepcja dostępu publicznego i prywatnego ma sens, kiedy możesz brać pod uwagę tworzenie nowego typu danych: chcesz wówczas udostępnić określona funkcjonalność oraz określony interfejs. Typ `string` umożliwia na przykład wyświetlanie samego siebie, pracę z podłańcuchami oraz poszczególnymi znakami oraz odczytywanie podstawowych atrybutów, takich jak długość łańcucha tekstowego. Naprawdę nie ma żadnego znaczenia, jak klasa ta została zaimplementowana.

Jeśli myślisz o tworzeniu klasy jak o definiowaniu nowego typu danych, sensowne jest, abyś w pierwszej kolejności określił, co ma być publiczne; co Twoja klasa powinna robić. Wszystkie elementy publiczne będą mogły zostać użyte przez każdego programistę, który będzie z tej klasy korzystać. Traktuj je jak interfejs; jak funkcję z interfejsem składającym się z argumentów i wartości zwrotnej. Każde takie rozwiązanie powinieneś starannie przemyśleć, ponieważ gdy zaczniesz już korzystać z interfejsu, jego zmiana będzie wymagać zmodyfikowania wszystkich jego użytkowników. Ponieważ metoda jest publiczna, może istnieć wiele elementów, które ją wywołują. Nie ma prostego sposobu na ograniczenie liczby niezbędnych zmian, które należy w nich dokonać. Nikt nie wymyśli zupełnie nowego sposobu na prowadzenie samochodów, ponieważ wszyscy musieliby od nowa nauczyć się jeździć! Nie ma natomiast żadnych przeszkód, aby zastosować nowy typ silnika, jak na przykład przy przejściu od korzystania wyłącznie z silników spalinowych do silników hybrydowych, ponieważ zmianie podlega implementacja, a nie interfejs.

Kiedy już wymyślisz podstawowy interfejs publiczny, powinieneś zacząć zastanawiać się nad tym, jak zaimplementujesz metody publiczne, z których będzie zbudowany ten interfejs. Wszystkie metody i pola, z których zechcesz skorzystać w celu zaimplementowania swoich metod publicznych i które nie muszą być publiczne, powinny być prywatne.

W przeciwieństwie do interfejsu publicznego prywatne metody i dane można łatwo modyfikować. Prywatne składowe klasy są dostępne wyłącznie dla metod tej klasy (zarówno publicznych, jak i prywatnych). Definiując szczegóły implementacji jako prywatne, dajesz sobie możliwość późniejszego wprowadzenia w niej zmian, jeśli kiedykolwiek zdecydujesz się na modyfikację funkcjonalności klasy (bardzo trudno uzyskać prawidłową implementację już za pierwszym razem). Pamiętaj, jak to było z samochodem hybrydowym!

Moja rada jest następująca: nigdy, ale to przenigdy nie definiuj pól danych jako publiczne i zawsze domyślnie definiuj metody jako prywatne, przenosząc je do publicznego interfejsu tylko wtedy, gdy jesteś przekonany, że określona metoda powinna się tam znaleźć. Przejście z prywatnego do publicznego jest łatwe, zmiana w drugą stronę będzie trudna — nie uda Ci się zagnieć dzina z powrotem do butelki. Jeśli potrzebujesz udostępnić określone pole, napisz metodę, która pobiera i zapisuje jego wartości (metody odczytujące dane często nazywane są **getterami**, natomiast metody zapisujące — **setterami**).

Zasada zakazująca upubliczniania pól danych może wydawać się bezwzględna. Czy nie będziesz musiał z jej powodu pisać całego mnóstwa setterów i getterów — funkcji takich jak `pobierz_Ruch`, które nie robią nic innego, jak tylko zwracają wartości prywatnych pól danych w rodzaju `_czyj_ruch?`

Tak, czasami będziesz musiał to robić. Mały koszt związany z pisaniem takich metod jest jednak całkowicie równoważony przez kłopoty, w których byś się znalazł, gdybyś zdał sobie sprawę, że w celu dodania nowej funkcjonalności wystarczyłaby zmiana trywialnego gettera. Możesz na przykład podjąć decyzję o zaprzestaniu przechowywania pewnej wartości w zmiennej i zamiast tego obliczać ją na podstawie innych zmiennych. Jeśli nie masz gettera i wszystkim pozwolasz, aby odczytywali tę zmienną z pola publicznego, znajdziesz się w tarapatach.

Z pewnością mógłbyś podać kilka przykładów pól, które bez obaw można przekształcić w publiczne. Radzę ci jednak, abyś tego nie robił. Co prawda zaoszczędzisz sobie nieco pisania, ale za to narazisz się na potencjalny ból głowy w przyszłości. Konsekwencją nietrafnego wybrania pól jest nieprawidłowy projekt, którego nie będziesz mógł łatwo zmienić.

## Co tak naprawdę znaczy private?

Tylko dlatego, że zadeklarujesz pewien element jako prywatny, nie dostaniesz automatycznie jakiejś gwarancji bezpieczeństwa. Prywatne pola klasy są przechowywane w pamięci komputera tak samo jak pola publiczne, zwykle tuż obok nich. Każdy kod może skorzystać ze wskaźnikowych sztuczek i odczytać zapisane w nich informacje. Ani system operacyjny, ani język programowania nie gwarantują bezpieczeństwa prywatnych danych w przypadku działania jakiegoś złośliwego programu. Zdefiniowanie danych jako prywatnych pozwala kompilatorowi zapobiegać przypadkowemu korzystaniu z tych danych, ale nie gwarantuje im pełnego bezpieczeństwa. Chociaż nie otrzymujesz takiej gwarancji, nadawanie danym statusu prywatności pozostaje przydatne.

A tak przy okazji, istnieje powszechnie stosowane określenie na użycie metod publicznych w celu ukrycia danych prywatnych — hermetyzacja. **Hermetyzacja** (zwana też enkapsulacją) oznacza ukrywanie implementacji, dzięki czemu użytkownicy klasy mogą pracować wyłącznie z określonym zbiorem metod, które stanowią interfejs tej klasy. Okreżenia takie jak „ukrywanie danych” albo „szczegóły implementacji” są bardziej sugestywne, ale to ze słowem „hermetyzacja” będziesz mieć częściej do czynienia. Teraz już wiesz, co ono oznacza.

## Podsumowanie

Klasa stanowi jeden z podstawowych klocków tworzących większość prawdziwych programów pisanych w C++. Klasy umożliwiają programistom tworzenie dużych projektów, które są proste do zrozumienia i z którymi łatwo pracować. Właśnie poznaleś pełną bardzo przydatną cechę klasy — możliwość ukrywania danych, a w kilku kolejnych rozdziałach przedstawię Ci wiele kolejnych funkcjonalności klas.

## Sprawdź się

1. Dlaczego miałybyś korzystać z danych prywatnych?

- A. Aby je zabezpieczyć przed hakerami.
- B. Aby inni programiści nie mogli się do nich dobrać.
- C. Aby było jasne, które dane powinny być użyte wyłącznie w celu implementacji klasy.
- D. Nie należy tego robić, bo pisanie programu stanie się trudniejsze.

- 2.** Czym różni się klasa od struktury?
- A. Niczym.
  - B. W klasie domyślnie wszystko jest publiczne.
  - C. W klasie domyślnie wszystko jest prywatne.
  - D. Klasa pozwala określić, które pola mają być publiczne, a które prywatne, natomiast w strukturze jest to niemożliwe.
- 3.** Co należy zrobić z polami danych w klasie?
- A. Zdefiniować je domyślnie jako publiczne.
  - B. Zdefiniować je domyślnie jako prywatne, ale przekształcić w publiczne, jeśli zajdzie taka potrzeba.
  - C. Nigdy nie definiować ich jako publiczne.
  - D. Klasy zazwyczaj nie zawierają danych, ale jeśli już tak się stanie, można z nimi robić, co się chce.
- 4.** W jaki sposób można zdecydować, czy metoda powinna być publiczna?
- A. Metoda nigdy nie powinna być publiczna.
  - B. Metoda zawsze powinna być publiczna.
  - C. Metoda powinna być publiczna, jeśli jest potrzebna do korzystania z głównych funkcjonalności klasy; w przeciwnym przypadku powinna być prywatna.
  - D. Metoda powinna być publiczna, jeśli jest prawdopodobne, że ktoś będzie chciał z niej korzystać.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Strukturę z zadania praktycznego z poprzedniego rozdziału, reprezentującą planszę do gry w kółko i krzyżyk, zaimplementuj jako klasę. Metody, które mogą być przydatne na zewnątrz klasy oznacz jako publiczne, natomiast wszystkie dane oraz funkcje pomocnicze zdefiniuj jako prywatne. Jaką część swojego kodu musiałeś zmienić?

# 25

## ■ ■ ■ ROZDZIAŁ 25

## Cykl życia klasy

---

Tworząc klasę, chciałbyś, aby była ona tak prosta w użyciu, jak to tylko możliwe. Istnieją trzy podstawowe operacje, które z dużym prawdopodobieństwem będzie realizować każda klasa:

1. Własna inicjalizacja.
2. Czyszczenie pamięci lub innych zasobów.
3. Kopiowanie siebie samej.

Wszystkie te trzy operacje są niezbędne w celu utworzenia dobrego typu danych. Weźmy za przykład klasę `string`. Zmienna łańcuchowa musi mieć możliwość inicjalizowania się, nawet jeśli w rezultacie powstanie pusty łańcuch. Nie powinna w tym celu korzystać z jakiegoś zewnętrznego kodu. Kiedy deklarujesz zmienną typu `string`, jest ona od razu gotowa do użycia. Co więcej, kiedy przestaniesz z niej korzystać, musi ona po sobie posprzątać, ponieważ typ łańcuchowy alokuje pamięć. Podczas używania klasy `string` nie musisz wywoływać metody sprzątającej — taki zabieg jest przeprowadzany automatycznie. I wreszcie musi istnieć możliwość kopiowania jednej zmiennej łańcuchowej do drugiej, tak samo jak można kopować między różnymi zmiennymi wartość całkowitą. Podsumowując powyższe rozważania, można stwierdzić, że te trzy funkcjonalności powinny być częścią każdej klasy, dzięki czemu poprawne z niej korzystanie będzie łatwe, natomiast korzystanie nieprawidłowe będzie utrudnione.

Zajmijmy się tymi funkcjonalnościami, począwszy od inicjalizowania obiektu, i zobaczymy, w jaki sposób język C++ ułatwia nam realizację tych zadań.

## Konstruowanie obiektu

Być może zauważyłeś, że interfejs klasy `Szachownica` (tj. publiczna część klasy), inicjalizujący planszę, nie zawierał kodu. Zaraz to naprawimy.

Kiedy deklarujesz zmienną pewnej klasy, musi być jakiś sposób na zainicjalizowanie zmiennej:

`Szachownica plansza;`

W C++ kod uruchamiany w momencie deklaracji obiektu jest nazywany **konstruktorem**. Konstruktor powinien skonfigurować obiekt w taki sposób, aby można było z niego korzystać bez dodatkowej inicjalizacji. Konstruktor może także przyjmować argumenty, co mogłeś już zobaczyć w przypadku deklarowania wektora o podanym rozmiarze:

```
vector<int> v( 10 );
```

Powyższa instrukcja wywołuje konstruktor z parametrem 10 — konstruktor wektora inicjalizuje nowy wektor, który od razu będzie mógł przechowywać dziesięć liczb całkowitych.

W celu utworzenia konstruktora należy po prostu zadeklarować funkcję o takiej samej nazwie, jaką ma klasa, z tym że bez podawania argumentów ani wartości zwrotnej (nawet void; zwracać wartości w ogóle nie należy uwzględniać).

### Przykładowy kod 58.: konstruktor.cpp

```
enum BierkaSzachowa { PUSTE_POLE, BIALY_PION /* i pozostałe bierki */ };
enum KolorGracza { KG_BIALY, KG_CZARNY };

class Szachownica
{
public:
    Szachownica(); // <-- Brak wartości zwrotnej!
    KolorGracza pobierzRuch();
    BierkaSzachowa pobierzBierke (int x, int y);
    void wykonajRuch (int z_x, int z_y, int na_x, int na_y);

private:
    BierkaSzachowa _plansza[ 8 ][ 8 ];
    KolorGracza _czyj_ruch;
};

Szachownica::Szachownica() // <-- Tutaj też nie ma wartości zwrotnej
{
    _czyj_ruch = KG_BIALY;
    // Zaczynamy od wyczyszczenia całej szachownicy i ustawienia na niej bierek
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            _plansza[ i ][ j ] = PUSTE_POLE;
        }
    }
    // Pozostały kod inicjalizujący plansze...
}
```

Nie będę zamieszczać wszystkich definicji metod, jeśli nie uległy one zmianom, będę natomiast pokazywać pełne deklaracje klas, abyś mógł zobaczyć, jak współpracują one z resztą kodu.

Zauważ, że konstruktor stanowi część publicznej sekcji klasy. Gdyby konstruktor klasy Szachownica nie był publiczny, niemożliwe byłoby tworzenie instancji obiektu. Dlaczego? Konstruktor musi być wywoływany każdorazowo podczas tworzenia obiektu, ale gdyby był prywatny, nikt spoza klasy nie mógłby go wywołać! Skoro jednak wszystkie obiekty muszą w celu swojej inicjalizacji wywoływać konstruktor, w sytuacji takiej w ogóle nie można by deklarować obiektów.

Konstruktor jest wywoływany w tym samym wierszu, w którym tworzony jest obiekt:

```
Szachownica plansza; // Wywołanie konstruktora klasy Szachownica
```

Można go też wywołać podczas alokowania pamięci:

```
Szachownica *plansza = new Szachownica; // Wywołuje konstruktor klasy Szachownica w ramach
// alokacji pamięci
```

Jeśli deklarujesz wiele obiektów:

```
Szachownica a;
Szachownica b;
```

konstruktory zostaną uruchomione w kolejności deklarowanych obiektów (najpierw a, potem b).

Tak samo jak w przypadku zwykłych funkcji, konstruktor może przyjmować dowolną liczbę argumentów, a jeśli obiekt można inicjalizować na różne sposoby, konstruktory można przeładowywać typami argumentów. Mógłbyś na przykład utworzyć drugi konstruktor klasy Szachownica, który pobiera rozmiar planszy:

```
class Szachownica
{
    Szachownica ();
    Szachownica (int rozmiar_planszy);
};
```

i zdefiniować funkcję działającą w taki sam sposób jak dowolna inna metoda tej klasy:

```
Szachownica::Szachownica (int rozmiar)
{
    // ...kod
}
```

Argumenty są przekazywane do konstruktora następująco:

```
Szachownica plansza ( 8 ); // 8 jest argumentem konstruktora Szachownica
```

Przy użyciu słowa kluczowego new przekazywanie argumentów wygląda tak samo jak podczas bezpośredniego wywoływania konstruktora:

```
Szachownica *w_plansza = new Szachownica( 8 );
```

Mam jeszcze małą uwagę na temat składni. Chociaż w celu przekazywania argumentów do konstruktora używa się nawiasów, nie można ich zastosować w czasie deklarowania obiektu za pomocą konstruktora bezargumentowego.

#### ZŁY KOD

```
Szachownica plansza();
```

Prawidłowy sposób zapisania powyższej instrukcji wygląda następująco:

```
Szachownica plansza;
```

Można jednak użyć nawiasów w przypadku korzystania ze słowa kluczowego new:

```
Szachownica *plansza = new Szachownica();
```

Dzieje się tak z powodu pewnej osobliwości związanej z przetwarzaniem składni języka C++ (szczegółowe wyjaśnienia byłyby jednak wyjątkowo zagmatwane). Unikaj po prostu stosowania nawiasów podczas deklarowania obiektu za pomocą konstruktora, który nie przyjmuje parametrów.

## Co się stanie, jeśli nie utworzysz konstruktora?

Jeśli nie utworzysz konstruktora, nasz przyjaciel C++ zrobi to za Ciebie. Konstruktor ten nie pobiera argumentów, ale inicjalizuje wszystkie pola klasy, wywołując ich domyślne konstruktory (chociaż nie zainicjalizuje typów podstawowych, takich jak int albo char, tak więc bądź ostrożny). Zwykle zalecam, abyś tworzył własne konstruktory, aby zagwarantować, że wszystko zostanie zainicjalizowane zgodnie z Twoimi potrzebami.

Kiedy już zadeklarujesz konstruktor swojej klasy, C++ nie będzie automatycznie generować w Twoim imieniu własnego konstruktora. Kompilator przyjmie, że wiesz, co robisz, i że sam chcesz utworzyć wszystkie konstruktory klasy. W szczególności, jeśli utworzysz konstruktor, który pobiera argumenty, Twój kod nie otrzyma domyślnego konstruktora, chyba że zadeklarujesz go celowo.

Taki zabieg może mieć zaskakujące konsekwencje. Jeśli Twój kod korzysta z wygenerowanego automatycznie konstruktora domyślnego, a Ty później napiszesz własny, niedomyślny konstruktor, który pobiera jeden lub kilka argumentów, kod bazujący na automatycznym konstruktorem domyślnym nie zostanie skompilowany. Będziesz musiał ręcznie dodać konstruktor domyślny, ponieważ Twój kod nie tworzy już za Ciebie żadnego konstruktora.

## Inicjalizacja składowych klasy

Każda składowa klasy musi zostać zainicjalizowana w konstruktorze. Wyobraź sobie, że składową naszej klasy Szachownica jest zmienna typu string:

```
class Szachownica
{
public:
    Szachownica ();

    string pobierzRuch ();
    BierkaSzachowa pobierzBierke (int x, int y);
    void wykonajRuch (int z_x, int z_y, int na_x, int na_y);

private:
    BierkaSzachowa _plansza[ 8 ][ 8 ];
    string _czyj_ruch;
};
```

Można oczywiście przypisać zmiennej `_czyj_ruch` wartość:

```
Szachownica::Szachownica ()
{
    _czyj_ruch = "biały";
```

Rzeczywisty kod wykonywany w tym miejscu może być jednak trochę zaskakujący. Najpierw, już na samym początku konstruktora `Szachownica`, zostanie wywołany konstruktor zmiennej `_czyj_ruch`. Jest to całkiem słuszne, ponieważ dzięki temu możesz w konstruktorze bezpiecznie korzystać z wszystkich pól swojej klasy. Jeśli by konstruktory tych składowych nie zostały wywołane, nie mógłbyś ich używać. Celem konstruktora jest udostępnienie obiektu!

Zamiast polegać na konstruktorze domyślnym, możesz przekazać argumenty konstruktorowi składowej klasy. Składnia jest w takim przypadku nieco dziwna, ale działa:

```
Szachownica::Szachownica ()
// Po dwukropku następuje lista zmiennych razem z argumentami dla konstruktora
: _czyj_ruch ( "biały" )
{
    // W tym momencie został wywołany konstruktor zmiennej _czyj_ruch,
    // a zmiana ta zawiera wartość "biały"
```

Taki sposób definiowania konstruktora nazywany jest **listą inicjalizacyjną**. Będziemy mieć jeszcze do czynienia z taką składnią podczas inicjalizowania składowych klasy. Składowe na liście inicjalizacyjnej są porozdzielane przecinkami. Jeśli na przykład do klasy Szachownica dodamy nową składową, która przechowuje liczbę dokonanych ruchów, moglibyśmy ją zainicjalizować na liście:

```
class Szachownica
{
public:
    Szachownica ();

    string pobierzRuch ();
    BierkaSzachowa pobierzBierke (int x, int y);
    void wykonajRuch (int z_x, int z_y, int na_x, int na_y);

private:
    BierkaSzachowa _plansza[ 8 ][ 8 ];
    string _czyj_ruch;
    int _licznik_ruchow;
};

Szachownica::Szachownica ()
// Po dwukropku następuje lista zmiennych razem z argumentami dla konstruktora
: _czyj_ruch ( "biały" )
, _licznik_ruchow( 0 )
{ }
```

## Użycie listy inicjalizacyjnej do pól stałych

Jeśli pole w swojej klasie zadeklarujesz jako const, musi ono zostać zainicjalizowane na liście:

```
class ListaStalych
{
public:
    ListaStalych (int wart);

private:
    const int _wart;
};

ListaStalych::ListaStalych (int wart)
: _wart( wart )
{ }
```

Nie można zainicjalizować pola typu const poprzez przypisanie do niego wartości, ponieważ pola te są już zdefiniowane jako niezmienne. Lista inicjalizacyjna jest jedynym miejscem, w którym nie są one jeszcze w pełni uformowane, dzięki czemu można tam bezpiecznie określać wartości obiektów stałych. Z tego samego powodu pole, które jest referencją, także musi być inicjalizowane na liście.

Spotkamy jeszcze jedno zastosowanie list inicjalizacyjnych, kiedy przejdziemy do omawiania dziedziczenia.

## Niszczenie obiektu

Podobnie jak w przypadku konstruktora inicjalizującego obiekt, czasami będzie Ci potrzebny kod, który posprząta po obiekcie już nieużywanym. Jeśli na przykład konstruktor alokuje pamięć (albo inny zasób), trzeba ją będzie zwrócić do systemu operacyjnego, gdy obiekt nie jest już potrzebny. Czyszczenie takie nazywane jest niszczeniem obiektu i odbywa się w specjalnej metodzie nazywanej **destruktorem**. Destruktory są wywoływanie, kiedy dany obiekt nie jest już potrzebny, na przykład wtedy, gdy skasujesz wskaźnik do tego obiektu.

Spójrzmy na przykład. Przypuśćmy, że masz klasę reprezentującą listę powiązaną. W celu zaimplementowania odpowiedniej klasy mógłbyś utworzyć pole, które przechowuje bieżącą głowę listy:

```
struct WezelListyPowiazanej
{
    int wart;
    WezelListyPowiazanej *w_nastepny;
};

class ListaPowiazana
{
public:
    ListaPowiazana () ; // Konstruktor
    void wstaw (int wart); // Dodanie węzła

private:
    WezelListyPowiazanej *_w_glowa;
};
```

Jak już to widzieliśmy, węzeł główny w liście powiązanej, tak samo jak inne elementy, wskazuje pamięć przydzieloną za pomocą instrukcji new. Oznacza to, że w pewnym momencie, kiedy skończymy już pracę z obiektem klasy ListaPowiazana, będziemy musieli po nim posprzątać. Właśnie do tego służy destruktor. Zobaczmy, w jaki sposób możemy dodać go do naszego typu. Destruktor, podobnie jak konstruktor, ma specjalną nazwę — jest to nazwa klasy poprzedzona tildą (~), i podobnie jak to jest w przypadku konstruktora, nie zwraca on wartości. W przeciwieństwie jednak do konstruktora, destruktor nigdy nie przyjmuje żadnych argumentów.

```
class ListaPowiazana
{
public:
    ListaPowiazana () ; // Konstruktor
    ~ListaPowiazana () ; // Destruktor — zwróć uwagę na tildę (~)

    void wstaw (int wart); // Dodanie węzła

private:
    WezelListyPowiazanej *_w_glowa;
};

ListaPowiazana::~ListaPowiazana ()
{
    WezelListyPowiazanej *w_itr = _w_glowa;
    while ( w_itr != NULL )
    {
```

```

        WezelListyPowiazanej *w_tymcz = w_itr->w_nastepny;
        delete w_itr;
        w_itr = w_tymcz;
    }
}

```

Kod destruktora jest podobny do kodu usuwającego wszystkie elementy listy powiązanej, który już widziałeś. Jedyna różnica polega na tym, że w klasie istnieje specjalna metoda służąca wyłącznie do przeprowadzania takiego oczyszczania. Poczekaj jeszcze chwilę! Czy nie byłoby bardziej sensowne, gdyby każdy węzeł sam sprzątał swoje dane? Czy to właśnie nie jest celem destruktora? A gdybyśmy napisali taki kod?

```

class WezelListyPowiazanej
{
public:
    ~WezelListyPowiazanej ();
    int wart;
    WezelListyPowiazanej *w_nastepny;
};

WezelListyPowiazanej::~WezelListyPowiazanej ()
{
    delete w_nastepny;
}

```

Możesz w to wierzyć lub nie, ale powyższy kod inicjuje ciąg rekurencyjnych wywołań funkcji. Wywołanie instrukcji `delete` uruchamia destruktor obiektu, na który wskazuje `w_nastepny` (albo nie robi nic, jeżeli `w_nastepny` ma wartość `NULL`). Destruktor z kolei wywołuje polecenie `delete` i uruchamia następny destruktor. Jaki jednak jest nasz przypadek bazowy? Co powstrzyma ten ciąg destrukcji? W pewnym momencie zmienna `w_nastepny` będzie mieć wartość `NULL` i wówczas wywołanie instrukcji `delete` nic nie zrobi. Taki jest zatem przypadek bazowy, został on tylko ukryty w wywołaniu polecenie `delete`. Kiedy będziemy już mieć destruktor dla klasy `WezelListyPowiazanej`, destruktor klasy `ListaPowiazana` będzie musiał go po prostu wywołać:

```

ListaPowiana::~ListaPowiazana ()
{
    delete _w_glowa;
}

```

Takie wywołanie polecenia `delete` rozpoczyna ciąg rekurencyjnych wywołań, które dojdą aż do końca listy.

Być może myślisz teraz, że jest to całkiem ładny wzorzec, tylko do czego jest nam potrzebny akurat destruktor? Czy nie moglibyśmy napisać własnej metody i nadać jej nazwę wedle własnego uznania? Oczywiście, że moglibyśmy tak zrobić. Destruktor ma jednak pewną przewagę nad takim rozwiązaniem — jest on wywoływany automatycznie, gdy dany obiekt nie jest już potrzebny.

Co w takim razie znaczy stwierdzenie, że „obiekt nie jest już potrzebny”? Otóż może on mieć trzy różne znaczenia:

- 1.** Wskaźnik do tego obiektu został usunięty.
- 2.** Obiekt wyszedł poza zakres.
- 3.** Obiekt należy do klasy, której destruktor został wywołany.

## Niszczanie podczas usuwania

Wywołanie polecenie delete jawnie określa moment wywołania destruktora, z czym zresztą miałeś już do czynienia:

```
ListaPowiazana *w_list = new ListaPowiazana;
delete w_list; // Metoda ~ListaPowiazana (destruktor) jest wywoływaną dla obiektu w_list
```

## Niszczanie przy wyjściu poza zakres

W drugim przypadku, tj. przy wyjściu poza zakres, niszczanie jest operacją domniemaną. Jeżeli obiekt deklarowany jest między nawiasami klamrowymi, jego zakres kończy się poza tymi nawiasami:

```
if ( 1 )
{
    ListaPowiazana lista;
} // W tym miejscu wywoływany jest destruktor obiektu lista
```

Nieco bardziej złożony przypadek ma miejsce wtedy, gdy obiekt jest deklarowany wewnątrz funkcji. Jeśli funkcja zawiera instrukcję return, destruktor zostanie wywołany podczas wychodzenia z tej funkcji. W takiej sytuacji wyobrażam sobie, że destruktory obiektów zadeklarowanych wewnątrz bloku kodu są wywoływane podczas wychodzenia z tego bloku w miejscu, w którym znajduje się zamykający nawias klamrowy. Wyjście z bloku może nastąpić po wykonaniu się ostatniej instrukcji w bloku lub po napotkaniu instrukcji return albo break:

```
void foo ()
{
    ListaPolaczona lista;

    // Jakiś kod...
    if ( /* Jakiś warunek */ )
    {
        return;
    }
} // W tym miejscu wywoływany jest destruktor
```

W powyższym przykładzie, chociaż polecenie return znajduje się wewnątrz instrukcji if, nadal wyobrażam sobie, że destruktor jest wywoływany po natrafieniu na ostatni nawias klamrowy. Najważniejsze jednak, abyś zapamiętała, że destruktor zostanie uruchomiony wtedy, kiedy obiekt znajdzie się poza zakresem — gdy nie będzie można się odwołać do tego obiektu bez wywołania błędu kompilatora.

Jeśli masz do czynienia z wieloma obiektami, dla których pod koniec bloku należy wywołać destruktory, będą one uruchamiane w odwrotnym porządku w stosunku do kolejności, w jakiej były konstruowane te obiekty. Na przykład w przypadku kodu

```
{
    ListaPowiazana a;
    ListaPowiazana b;
}
```

destruktor obiektu b zostanie uruchomiony przed destruktorem obiektu a.

## Niszczanie przez inny destruktor

Jeśli z kolei masz do czynienia z obiektem, który znajduje się wewnątrz innej klasy, destruktor tego obiektu zostanie wywołany po uruchomieniu destruktora tej klasy. Spójrz na przykład prostej klasy:

```
class NazwaOrazEmail
{
    /* Tutaj zwykle znajdują się jakieś metody */
private:
    string _nazwa;
    string _email;
};
```

W powyższym kodzie destruktor pól `_nazwa` oraz `_email` zostanie wywołany po zakończeniu działania destruktora obiektu klasy `NazwaOrazEmail`. Takie rozwiązanie jest bardzo wygodne, ponieważ nie musisz robić nic specjalnego w celu posprzątania po obiektach swojej klasy. Tak naprawdę musisz jedynie wyczyścić wskaźniki (albo inne zasoby, takie jak uchwyty do plików albo połączenia sieciowe), wywołując instrukcję `delete`.

A przy okazji — jeśli nie dodasz do swojej klasy destruktora, kompilator zapewni jego uruchomienie dla wszystkich obiektów, które stanowią część tej klasy.

Idea stosowania konstruktora w celu inicjalizowania klasy oraz destruktora po to, aby oczyścić pamięć lub inne zasoby należące do danej klasy ma swoją nazwę: **inicjowanie przy pozyskaniu zasobu** (ang. *resource allocation is initialization*, w skrócie RAII). Podstawowe znaczenie tego pojęcia w C++ jest takie, że w celu obsługi zasobów należy tworzyć klasy. Kiedy tworzone są klasy, całą inicjalizację powinien przeprowadzać konstruktor, natomiast za czyszczenie powinien odpowiadać destruktor. Od użytkowników klasy nie należy domagać się żadnych specjalnych czynności. Często stosowanie tej zasady prowadzi do powstawania takich klas jak `NazwaOrazEmail` z przykładu powyżej. Oba znajdujące się w niej łańcuchy tekstowe sprzątają same po sobie, a klasa `NazwaOrazEmail` nie potrzebuje swojego, ręcznie pisanej destruktora.

## Kopiowanie klas

Trzecim przystankiem w naszej wycieczce po związkach z klasami ideach będzie obsługa kopiowania instancji klas. W języku C++ powszechnie tworzy się nowe klasy w taki sposób, aby umożliwić kopiowanie ich instancji. Na przykład mógłbyś napisać:

```
ListaPowiazana lista_jeden;
ListaPowiazana lista_dwa;

lista_dwa = lista_jeden;
ListaPowiazana lista_trzy = lista_dwa;
```

W C++ istnieją dwie funkcje, które możesz zdefiniować, aby mieć gwarancję, że tego rodzaju operacje kopiowania będą przebiegać poprawnie. Jedna z nich jest realizowana za pomocą operatora przypisania, a druga nazywana jest konstruktorem kopującym. Zaczniemy od zapoznania się z operatorem przypisania, po czym porozmawiamy o konstruktorze kopującym.

Być może zastanawiasz się, czy te funkcje są faktycznie potrzebne. Czy kopiowanie nie powinno po prostu działać? Odpowiedź na to pytanie brzmi „tak”, czasami kopiowanie po prostu działa; język C++ udostępnia domyślne wersje konstruktora kopującego oraz operatora przypisania.

W niektórych przypadkach nie możesz jednak zdać się na domyślne wersje funkcji. Czasami kompilator nie jest wystarczająco mądry, aby wiedzieć, czego oczekujesz. Na przykład domyślna wersja konstruktora kopiącego oraz operatora przypisania zrealizuje coś, co jest nazywane **płytką kopią wskaźnika**. Z płytka kopią mamy do czynienia wtedy, gdy drugi wskaźnik wskazuje to samo miejsce pamięci co pierwszy wskaźnik. Takie rozwiązanie jest uważane za płytkie, ponieważ nie następuje kopianie żadnego ze wskazywanych miejsc pamięci, a tylko tworzona jest kopia samego wskaźnika. Czasami płytka kopia może wystarczyć, ale istnieją przypadki, kiedy sytuacja taka rodzi problemy.

Załóżmy na przykład, że mamy naszą klasę `ListaPowiazana` i że piszemy poniższy kod:

```
ListaPowiazana lista_jeden;
ListaPowiazana lista_dwa;

lista_jeden = lista_dwa;
```

Problem polega na tym, że domyślny operator przypisania generuje następujący kod:

```
lista_jeden._w_glowa = lista_dwa._w_glowa
```

Który możemy wyobrazić sobie następująco:



Teraz mamy dwa obiekty o tej samej wartości wskaźnika, a destruktor każdego z nich będzie próbował zwolnić pamięć związaną z danym wskaźnikiem.

Kiedy wywołany zostanie destruktor obiektu `lista_dwa`, usunie on wartość `lista_dwa._w_glowa`. (destruktor obiektu `lista_dwa` jest uruchamiany jako pierwszy, ponieważ destruktory są wywoływane w odwrotnej kolejności względem konstruktorów, a konstruktor obiektu `lista_dwa` został uruchomiony jako drugi). Jako następny zostanie wywołany destruktor obiektu `lista_jeden`, który usunie wartość `lista_jeden._w_glowa`. Problem jednak tkwi w tym, że wartość `lista_dwa._w_glowa` została już usunięta i jeśli spróbujesz skasować wskaźnik po raz drugi, w Twoim programie wystąpi awaria!

Najwyraźniej kiedy zostanie wywołany jeden z destruktorów, druga lista zostaje utracona! Dzięki operatorom przypisania można obsłużyć tego typu sytuacje. Zobaczmy, jak powinien wyglądać nasz kod.

## Operator przypisania

Operator przypisania jest wywoływanym podczas przypisywania obiektu do innego, istniejącego już wcześniej obiektu, jak na przykład w poniższym zapisie:

```
lista_dwa = lista_jeden
```

Implementacja operatora przypisania wymaga zastosowania nowej składni, która umożliwia zdefiniowanie operatora. Na szczęście nie jest ona zbyt skomplikowana:

```
ListaPowiazana& operator= (ListaPowiazana& lewa, const ListaPowiazana& prawa);
```

Taki zapis wygląda jak zwykła deklaracja funkcji, która przyjmuje dwa argumenty: nieoznaczoną jako `const` referencję do obiektu `ListaPowiazana` oraz oznaczoną jako `const` referencję do tego samego obiektu, i która zwraca referencję do obiektu `ListaPowiazana`. Jedynym dziwnym elementem jest nazwa tej funkcji: `operator=`. Oznacza ona, że nie definiujemy nowej funkcji, tylko określamy, jakie będzie znaczenie znaku równości, gdy zostanie on użyty w odniesieniu do klasy `ListaPowiazana`. Pierwszym argumentem tej funkcji jest lewa strona znaku równości, czyli element, do którego następuje przypisanie — nie może on być zatem stały. Drugim argumentem jest prawa strona znaku równości; to wartość przypisywana (powinna ona być stała, ponieważ nie ma żadnego powodu, aby ją modyfikować, chociaż nie istnieje bezwzględny wymóg oznaczania jej jako `const`):

```
lewa = prawa;
```

Powodem zwracania referencji do obiektu `ListaPowiazana` jest możliwość łączenia przypisań:

```
lista_powiazana = lewa = prawa;
```

Od tej pory, zamiast w większości przypadków deklarować autonomiczną funkcję `operator=`, można uczynić z niej funkcję składową klasy, dzięki czemu będzie ona mogła działać z prywatnymi polami klasy (w przeciwieństwie do samodzielnej funkcji, którą zdefiniowałem powyżej). Zobaczmy, jak to wygląda:

```
class ListaPowiazana
{
public:
    ListaPowiazana () ; // Konstruktor
    ~ListaPowiazana () ; // Destruktor — zwróć uwagę na tyłde
    ListaPowiazana& operator= (const ListaPowiazana& inna);

    void wstaw (int wart); // Dodanie węzła

private:
    WzelListyPowiazanej *_w_glowa;
};
```

Zwróci uwagę, że brakuje jednego argumentu. Jest tak dlatego, że wszystkie funkcje składowe klasy domyślnie przyjmują klasę jako argument. W tym przypadku metoda `operator=` jest używana, gdy lewą stroną przypisania jest klasa. Innymi słowy, w kodzie:

```
lewa = prawa;
```

funkcja `operator=` zostanie wywołana dla zmiennej `lewa`. To tak jakby napisać:

```
lewa.operator= ( prawa );
```

Po zakończeniu działania funkcji zmieniąca `lewa` będzie mieć taką samą wartość jak `prawa`. Teraz pomówmy o tym, jak dla naszej klasy `ListaPowiazana` powinniśmy napisać funkcję `operator=`:

```
ListaPowiazana& ListaPowiazana::operator= (const ListaPowiazana& inna)
{
    // Co tu wpisać?
}
```

Z wcześniejszego omówienia wiemy już, że samo kopowanie adresu wskaźnika nie jest wystarczającą dobrym rozwiążaniem. Zamiast tego chcielibyśmy przekopiować całą strukturę. Nasza logika jest następująca: najpierw zwolnimy naszą istniejącą listę (ponieważ nie jest już potrzebna), po czym przekopiujemy każdy węzeł listy, dzięki czemu uzyskamy dwie odrębne listy. Na koniec zwróciśmy kopię klasy, z którą pracujemy, ponieważ potrzebujemy wartości zwrotnej.

Ostatnia czynność wymaga nowej składni, gdyż musimy dysponować jakimś sposobem na odniesienie się do bieżącego obiektu. W języku C++ możemy użyć specjalnej zmiennej, nazywanej wskaźnikiem **this**. Wskaźnik **this** wskazuje instancję klasy. Jeśli na przykład napiszesz `lista_jeden.wstawElement( 2 );`, wówczas wewnątrz metody `wstawElement` będziesz mógł użyć słowa kluczowego `this`, co spowoduje wskazanie elementu `lista_jeden`. Ze wskaźnika `this` będziemy korzystać także w celu zagwarantowania w metodzie pewnej dozy bezpieczeństwa.

```
ListaPowiazana& ListaPowiazana::operator= (const ListaPowiazana& inna)
{
    // Sprawdzamy, czy nie przypisujemy obiektu samemu sobie — jeśli tak się
    // dzieje, możemy to zignorować. Zauważ, że w tym miejscu
    // korzystamy ze wskaźnika this, aby zagwarantować, że inna wartość
    // nie ma takiego samego adresu jak nasz obiekt
    if ( this == & inna )
    {
        // Zwróć obiekt this, aby zachować możliwość łączenia przypisań
        return *this;
    }
    // Przed kopiowaniem nowych wartości powinniśmy zwolnić zaalokowaną wcześniej pamięć,
    // ponieważ nie jest już ona używana
    delete _w_glowa;
    _w_glowa = NULL;

    WezelListyPowiazanej *_w_itr = inna._w_glowa;
    while ( _w_itr != NULL )
    {
        wstaw( _w_itr->wart );
    }
    return *this;
}
```

Kilka uwag na temat powyższej funkcji: zauważ, że najpierw sprawdzamy, czy nie następuje przypisanie zmiennej sobie samej. Zazwyczaj nie oczekujemy, że coś takiego się stanie, ale nie powinniśmy rezygnować z zagwarantowania, że takie przypisanie będzie bezpieczne. Następujący kod:

```
a = a;
```

powinien być stu procentowo bezpieczny i nie powodować żadnych zmian.

Następnie powinniśmy zwolnić pamięć skojarzoną ze starą listą, ponieważ nie jest ona już nam potrzebna. Usuwając zmienną `_w_glowa`, możemy pozbyć się całej listy, tak samo jakbyśmy się posłużyli destruktorem.

Na koniec musimy zapełnić listę właściwymi wartościami, co możemy wykonać, przechodząc w pętli przez starą listę i wstawiając do naszej listy pobrane z niej elementy. Voilà — mamy klasę, którą można kopować!

Na szczęście nie wszystkie klasy wymagają aż tak wyszukanego kopiowania. Jeśli żadna ze składowych klasy nie jest wskaźnikiem, prawdopodobnie operator przypisania w ogóle nie będzie potrzebny! Tak jest, życzliwy i rozważny język C++ udostępnia operator przypisania, który domyślnie skopiuje wszystkie elementy, wywołując **własny** operator przypisania (jeśli jest obiektem danej klasy) albo kopując je kawałek po kawałku (jeśli jest wskaźnikiem albo inną wartością). Jeśli więc nie masz w swojej klasie wskaźnika, w większości przypadków możesz zdać się na domyślny operator przypisania. Możesz kierować się ogólną zasadą, zgodnie z którą jeśli musisz napisać własny destruktor, prawdopodobnie powinieneś także napisać swój

operator przypisania. Jest tak dlatego, że jeśli dysponujesz destruktorem, to zapewne w celu zwalniania pamięci, a jeśli zwalniasz pamięć, musisz zagwarantować, że kopie klas otrzymują swoje kopie pamięci.

## Konstruktor kopiąjący

Jest jeszcze jeden przypadek do przemyślenia. Co zrobić, jeżeli chcesz skonstruować obiekt, który ma być taki sam, jak inny obiekt:

```
ListaPowiazana lista_jeden;
ListaPowiazana lista_dwa( lista_jeden );
```

Jest to szczególny przypadek użycia konstruktora — pobiera on obiekt takiego samego typu, jaki ma obiekt konstruowany. Taki konstruktor nosi nazwę **konstruktora kopiącego**. Konstruktor kopiąjący powinien sporządzić nowy obiekt, który będzie dokładną kopią oryginału. W tym przypadku obiekt `lista_dwa` powinien zostać zainicjalizowany w taki sposób, aby wyglądał identycznie z `lista_jeden`. Konstruktor kopiąjący przypomina w działaniu operator przypisania, z tym że zamiast działać na istniejącej klasie, zaczyna od klasy, która jest zupełnie niezainicjalizowana. Dzięki takiemu rozwiążaniu procesor nie traci czasu na konstruowanie klasy, tylko nadpisuje wartości. Konstruktor kopiąjący jest zwykle prosty do zaimplementowania i często bardzo przypomina operator przypisania. Oto jak wyglądałby on dla klasy `ListaPowiazana`:

```
class ListaPowiazana
{
public:
    ListaPowiazana () ; // Konstruktor
    ~ListaPowiazana () ; // Destruktor — zwróć uwagę na tyłkę
    ListaPowiazana& operator= (const ListaPowiazana& inna);
    ListaPowiazana (const ListaPowiazana& inna);

    void wstaw (int wart); // Dodanie węzła

private:
    ListaPowiazanaNode * _w_glowa;
};

ListaPowiazana::ListaPowiazana (const ListaPowiazana& inna)
    : _w_glowa( NULL ) // Zaczynamy od NULL, na wypadek gdyby lista inna była pusta
{
    // Zauważ, że kod ten jest bardzo podobny do kodu metody operator=.
    // W prawdziwym programie miałyby sens utworzenie
    // metody pomocniczej wykonującej to zadanie
    ListaPowiazanaNode *w_itr = inna._w_glowa;
    while ( w_itr != NULL )
    {
        wstaw( w_itr->wart );
    }
}
```

Widzisz? Proste jak drut.

Jeśli nie napiszesz swojego konstruktora kopiącego, kompilator udostępni konstruktor domyślny. Taki konstruktor zachowuje się jak domyślny operator przypisania, tj. dla każdego

obiektu klasy uruchamia konstruktor kopiujący i wykonuje zwykłe kopie wartości takich jak liczby całkowite albo wskaźniki. Jeśli potrzebny jest Ci własny operator przypisania, to w większości przypadków prawdopodobnie będziesz musiał napisać również konstruktor kopiujący.

Istnieje pewna rzecz, którą powinieneś wiedzieć o konstruktorze kopiującym i która czasami zaskakuje początkujących programistów (sam byłem zdziwiony, kiedy pierwszy raz się o tym dowiedziałem).

Spójrz na następujący kod:

```
ListaPowiazana lista_jeden;  
ListaPowiazana lista_dwa = lista_jeden;
```

Co takiego Twoim zdaniem się wydarzy? Czy zostanie wywołany operator przypisania? Nie. Okazuje się, że kompilator jest na tyle sprytny, iż potrafi rozpoznać, że obiekt `lista_dwa` jest inicjalizowany na podstawie obiektu `lista_jeden`, w związku z czym sam wywoła konstruktor kopiujący, unikając tym samym niepotrzebnej inicjalizacji obiektów. Czy to nie miło?

## Pełna lista metod generowanych przez kompilator

Poznałeś już wszystkie metody, które kompilator automatycznie wygeneruje w Twoim imieniu:

1. Domyślny konstruktor.
2. Domyślny destruktor.
3. Operator przypisania.
4. Konstruktor kopiujący.

W odniesieniu do każdej klasy, którą tworzysz, powinieneś zastanowić się, czy możesz przyjąć domyślne implementacje metod oferowane przez kompilator. W wielu sytuacjach będziesz mógł tak zrobić, ale podczas pracy ze wskaźnikami często będziesz wolał zadeklarować własny destruktor, operator przypisania oraz konstruktor kopiujący (na ogół, jeśli będzie Ci potrzebna jedna z tych metod, będziesz potrzebować ich wszystkich).

## Całkowite zapobieganie kopiowaniu

Czasami w ogóle nie będzie Ci potrzebna możliwość kopiowania obiektów. Czy nie byłoby korzystne móc powiedzieć: „Nie pozwól na kopiowanie tego obiektu”? W ten sposób mógłbyś uniknąć implementowania konstruktora kopiującego lub operatora przypisania **i jednocześnie** nie ryzykowałbyś, że kompilator sam wygeneruje niebezpieczne wersje metod.

Istnieją sytuacje, w których możliwość kopiowania obiektów jest po prostu niepożądana. Jeśli na przykład piszesz grę komputerową z klasą reprezentującą aktualny statek kosmiczny gracza, tak naprawdę nie chciałbyś mieć kopii tego statku. Chcesz dysponować tylko pojedynczym statkiem, który zawiera wszystkie informacje na temat jego użytkownika.

Możesz zapobiec kopiowaniu, *deklarując* konstruktor kopiujący i operator przypisania, ale ich nie *implementując*. Kiedy już zadeklarujesz metodę, kompilator nie wygeneruje już jej automatycznie. Jeśli spróbujesz z niej skorzystać, otrzymasz błąd podczas konsolidacji, ponieważ użyłeś niezdefiniowanej funkcji. Taki błąd może być trudny do zlokalizowania, ponieważ konsolidator nie poda numeru wiersza, w którym wystąpił problem. O wiele lepsze komunikaty

błędów otrzymasz, jeśli oznaczysz metody jako prywatne. Dzięki temu w większości przypadków błędy wystąpią na etapie komplikacji, objawiając się czytelniejszymi komunikatami. Zobaczmy, jak coś takiego zrobić:

```
class Gracz
{
public:
    Gracz ();
    ~Gracz ();

private:
    // Deklarując poniższe metody, ale ich nie definiując,
    // sprawiasz, że kompilator ich nie wygeneruje
    operator= (const Gracz& inny);
    Gracz (const Gracz& inny);

    InformacjaOGraczu *_w_gracz_info;
};

// Brak implementacji operatora przypisania i konstruktora kopiącego
```

Podsumujmy: zawsze powinieneś wybrać jedną z poniższych możliwości:

- 1.** Skorzystać zarówno z domyślnego konstruktora kopiącego, jak i operatora przypisania.
- 2.** Utworzyć własny konstruktor kopowania oraz operator przypisania.
- 3.** Oznaczyć konstruktor kopiący i operator przypisania jako prywatne i nie tworzyć ich implementacji.

Jeśli nie zrobisz nic, dzięki kompilatorowi uzyskasz opcję pierwszą. Często najłatwiej rozpoczęć od opcji trzeciej i później, w razie potrzeby, dodać operator przypisania oraz konstruktor kopowania.

## Sprawdź się

- 1.** Kiedy dla klasy należy napisać konstruktor?
  - A. Zawsze, bez konstruktora nie można korzystać z klasy.
  - B. Kiedy wystąpi potrzeba zainicjalizowania klasy wartościami, które nie są domyślne.
  - C. Nigdy, kompilator zawsze zapewnia konstruktor.
  - D. Tylko wtedy, gdy potrzebny jest także destruktor.
- 2.** Jaki związek istnieje między destruktorem a operatorem przypisania?
  - A. Nie ma związku.
  - B. Destruktor klasy jest wywoływany przed uruchomieniem operatora przypisania.
  - C. Operator przypisania musi określić, jaka pamięć ma zostać zniszczona przez destruktor.
  - D. Operator przypisania musi zagwarantować, że wywołanie destruktora zarówno klasy kopowanej, jak i nowej będzie bezpieczne.
- 3.** Kiedy należy użyć listy inicjalizacyjnej?
  - A. Kiedy konstruktory powinny być maksymalnie wydajne i aby uniknąć konstruowania pustych obiektów.
  - B. Kiedy inicjalizowane są wartości stałe.
  - C. Kiedy dla pola klasy należy uruchomić niedomyślny konstruktor.
  - D. W każdym z powyższych przypadków.

**4.** Jaka funkcja zostanie uruchomiona w drugim wierszu poniższego kodu?

```
string str1;  
string str2 = str1;
```

- A. Konstruktor dla str2 oraz operator przypisania dla str1.
- B. Konstruktor dla str2 oraz operator przypisania dla str2.
- C. Konstruktor kopiący dla str2.
- D. Operator przypisania dla str2.

**5.** Jakie funkcje (i w jakiej kolejności) zostaną wywołane dla poniższego kodu?

```
{  
    string str1;  
    string str2;  
}
```

- A. Konstruktor dla str1 i konstruktor dla str2.
  - B. Destruktor dla str1 i konstruktor dla str2.
  - C. Konstruktor dla str1, konstruktor dla str2, destruktor dla str1 i destruktor dla str2.
  - D. Konstruktor dla str1, konstruktor dla str2, destruktor dla str2 i destruktor dla str1.
- 6.** Jeśli wiesz, że klasa ma konstruktor kopiący, który nie jest domyślny, jakie zdanie na temat operatora przypisania powinno być prawdziwe?
- A. Operator przypisania powinien być domyślny.
  - B. Operator przypisania nie powinien być domyślny.
  - C. Operator przypisania powinien być zadeklarowany, ale nie zaimplementowany.
  - D. Poprawne są odpowiedzi B i C.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

**1.** Zaimplementuj klasę `wektorInt`, zastępującą wektor, która operuje wyłącznie na liczbach całkowitych (nie musisz używać szablonów takich jak STL). Twój klasa powinna mieć następujący interfejs:

- Nieprzyjmujący argumentów konstruktor, który alokuje 32-elementowy wektor.
- Konstruktor przyjmujący jako argument rozmiar początkowy.
- Metoda `pobierz`, przyjmująca indeks i zwracająca wartość zapisaną pod tym indeksem.
- Metoda `ustaw`, przyjmująca indeks i wartość, która zapisuje tę wartość pod podanym indeksem.
- Metoda `dodaj`, dodająca element na końcu wektora i powiększającą wektor, jeśli będzie to potrzebne.
- Metoda `wstaw`, wstawiająca element na początku wektora.
- Konstruktor kopiący oraz operator przypisania.

Twoja klasa nie powinna powodować wycieków pamięci; każda przydzielona pamięć musi zostać zwrócona. Zastanów się dokładnie, w jaki sposób Twój klasa może zostać niewłaściwie użyta i jak obsłużyłbyś takie sytuacje. Co się stanie, jeśli użytkownik poda ujemny rozmiar początkowy? A co z próbą odczytania wartości spod ujemnego indeksu?

# 26

## ■ ■ ■ ROZDZIAŁ 26

## Dziedziczenie i polimorfizm

---

Do tej pory mówiłem o tym, jak na podstawie klas pisać pełne i użyteczne typy, co jest możliwe dzięki udostępnianiu czytelnych interfejsów oraz szerokiej pomocy w zakresie tworzenia, kopирования oraz usuwania obiektów. Rozwińmy teraz ideę interfejsu. Przypuśćmy, że masz samochód. Jest on już dość stary i zardzewiały. Niestety, żyjesz w świecie, w którym każdy producent samochodów stosuje inny system sterowania. Niektórzy z nich montują kierownice, inni dżojstiki, a jeszcze inni myszki. W niektórych samochodach jest pedał gazu, a w innych trzeba przeciągnąć pasek przewijania<sup>1</sup>. Taka sytuacja byłaby okropna. Za każdym razem, kiedy chciałbyś skorzystać z auta, musiałbyś od nowa uczyć się nim kierować. Ta sama sytuacja powtarzałaby się, gdybyś chciał pożyczyć samochód albo kupić nowy — znowu musiałbyś uczyć się nim kierować.

Na szczęście samochody są budowane zgodnie w pewnymi standardami. Za każdym razem, kiedy wsiadasz do auta, spotykasz ten sam interfejs — kierownicę, pedał gazu itd. Jedyna rzeczywista różnica polega na tym, że w niektórych samochodach znajduje się automatyczna skrzynia biegów, a w innych biegi należy przełączać ręcznie. Istnieją dwa interfejsy, w które może być wyposażone auto: ręczny i automatyczny.

Jeśli tylko wiesz, jak obsługiwać automatyczną skrzynię biegów, będziesz mógł prowadzić wszystkie samochody wyposażone w taki mechanizm. Kiedy jedziesz autem, szczegóły działania silnika nie są ważne. Istotne jest, że masz do czynienia z takimi samymi metodami realizującymi kierowanie, przyspieszanie i hamowanie jak we wszystkich innych samochodach.

Tylko co wspólnego ma to wszystko z C++? Otóż w języku tym można pisać kod, który oczekuje pewnego, dobrze zdefiniowanego interfejsu (w powyższej analogii Ty byłeś kodem, a mechanizm sterujący samochodem był interfejsem). Implementacja interfejsu (czyli samochodu) nie ma znaczenia. Jest zupełnie obojętnie, jaka konkretna implementacja interfejsu (wybranego przez Ciebie samochodu) zostanie użыта przez kod (przez Ciebie — kierowcę), ponieważ interfejs jest zaimplementowany w sposób zrozumiały przez kod. Ty, jako kierowca, możesz preferować pewne samochody, ale i tak będziesz potrafił kierować każdym z nich.

W jakiej sytuacji chciałbyś napisać kod mający takie właściwości? Pomyśl o grze komputerowej. W takiej grze może istnieć wiele różnych obiektów wyświetlanych na ekranie — pociski, statki kosmiczne, wrogowie. W głównej pętli gry należy dla każdej wyświetlonej ramki narysować od nowa wszystkie te obiekty na ich nowych pozycjach.

---

<sup>1</sup> Wiem, wiem; kierowanie samochodem za pomocą kółka przewijania prawdopodobnie powodowałoby masę wypadków.

Potrzebna jest Ci możliwość napisania kodu w następującej formie:

Wyczyść ekran.  
Przejdz w pętli przez listę wszystkich wyświetlanych obiektów.  
Narysuje każdy obiekt, który można wyświetlić.

W idealnej sytuacji lista zawierałaby wszystkie rodzaje możliwych do wyświetlenia obiektów. Wszystkie one muszą zawierać implementację jakiegoś wspólnego interfejsu, który umożliwi ich wyświetlanie. Niemniej chciałbyś także, aby pocisk, statek kosmiczny oraz wróg należały do różnych klas. Wszystkie te obiekty będą zawierać różne dane wewnętrzne (statek gracza musi mieć licznik trafień, wrogie statki kosmiczne powinny być wyposażone w sztuczną inteligencję, która będzie nimi sterować, a pociski muszą mieć zdefiniowany współczynnik zniszczenia, jakie mogą wywołać).

Wszystko to nie ma jednak znaczenia w pętli, która rysuje obiekty na ekranie. Jedynym ważnym aspektem jest fakt, że każda z tych różnych klas wspiera interfejs umożliwiający ich wyświetlanie. Potrzebny jest nam pakiet klas wyposażonych w ten sam interfejs, który jednak w każdej klasie jest zrealizowany inaczej.

W jaki sposób można to osiągnąć? Najpierw zdefiniujmy po prostu, co rozumiemy pod pojęciem „rysownalny”:

```
class Rysownalny
{
public:
    void rysuj ();
};
```

Powyższa prosta klasa *Rysownalny* definiuje tylko jedną metodę — *rysuj*. Metoda ta rysuje bieżący obiekt. Czyż nie byłoby wspaniale, gdybyśmy mogli utworzyć wektor *vector<Rysownalny\*>*, a następnie przechowywać wszystko, co implementuje metodę *rysuj* w tym wektorze<sup>2</sup>? Gdybyśmy mogli w ten sposób postąpić, mielibyśmy możliwość napisania kodu, który rysuje wszystko na ekranie, przechodząc po prostu przez poszczególne elementy wektora i wywołując metodę *rysuj*.

Wszyscy, którzy używają obiektów przechowywanych w tym wektorze, mogliby korzystać wyłącznie z metod wchodzących w skład interfejsu *Rysownalny*, i to właśnie takie rozwiązanie jest nam potrzebne!

I wiesz co? C++ wszystko to umożliwia! Zobaczmy, jak możemy zrealizować nasze zadanie.

## Dziedziczenie w C++

Najpierw zapoznajmy się z nowym terminem, którym jest **dziedziczenie**. Dziedziczenie oznacza, że pewna klasa otrzymuje cechy innej klasy. W naszym przypadku cechą, która zostanie odziedziczona, jest klasa *Rysownalny*, a w szczególności metoda *rysuj*. Klasa, która dziedziczy cechy innej klasy, nazywana jest **podklassą**. Z kolei klasa dziedziczona to **nadklasa**<sup>3</sup>. Nadklasa często definiuje metodę (lub metody) interfejsu, które mogą mieć różne implementacje w poszczególnych podklasach. W naszym przykładzie nadklasą jest klasa *Rysownalny*. Każdy

<sup>2</sup> Być może zastanawiasz się, dlaczego umieściłem wskaźnik w wektorze. Otóż dlatego, że w celu uzyskania zachowania, które poznasz już za chwilę, potrzebny nam będzie wskaźnik.

<sup>3</sup> Czasami nadklasa nazywana jest klasą macierzystą lub nadrzędną, natomiast podklaśa klasą potomną lub podrzędną. W książce tej najczęściej będę używał terminów *nadklasa* i *podklaśa*.

możliwy do wyświetlenia obiekt w grze będzie jej podklasą. Każda podklasa odziedziczy cechę związaną z posiadanym metodą rysuj, dzięki czemu kod korzystający z klasy Rysowalny będzie wiedział, że metoda ta jest dla niego dostępna. Każda podklasa zaimplementuje własną wersję metody rysuj; w rzeczy samej podklasy *muszą* zaimplementować własne wersje tej metody, co zagwarantuje, że wszystkie podklasy będą dysponować poprawną wersją metody rysuj.

No dobra, czy ta koncepcja jest już dla Ciebie jasna? Przejedźmy wobec tego do składni:

```
class Statek : public Rysowalny
{
}
```

Zapis : public Rysowalny oznacza, że klasa Statek dziedziczy po klasie Rysowalny. W kodzie tym podklasa Statek dziedziczy wszystkie metody publiczne oraz publiczne dane z nadklasy Rysowalny. Na razie dziedziczenie dotyczy metody rysuj, która w rzeczywistości tworzy całą metodę. Jeśli napiszesz:

```
Statek s;
s.rysuj();
```

powyższe wywołanie metody rysuj spowoduje wywołanie implementacji metody rysuj zapisanej w klasie Rysowalny. W tym przypadku niezupełnie o to nam chodzi, ponieważ klasa Statek powinna mieć własną metodę rysującą, zamiast korzystać z wersji metody, która stanowi część interfejsu klasy Rysowalny.

Aby można było zrealizować takie rozwiązanie, klasa Rysowalny powinna pozwalać na przesłanianie swojej metody rysuj w podklasach. W tym celu należy oznaczyć taką metodę jako **wirtualną**. Metoda wirtualna to metoda będąca częścią nadklasy, którą można przesłaniać w poszczególnych podklasach.

```
class Rysowalny
{
public:
    virtual void rysuj ();
};
```

W wielu przypadkach w ogóle nie będziesz chciał, aby nadklasa udostępniała jakąś implementację, i zamiast tego zdecydujesz, że jej podklasy powinny dysponować własnymi implementacjami danej metody (ponieważ na przykład nie będzie istnieć dobry „domyślny” sposób rysowania obiektu). Taki zamysł możesz zrealizować, deklarując klasę jako **czysto wirtualną**, co wygląda następująco (zwróć uwagę na symbol = 0):

```
class Rysowalny
{
public:
    virtual void rysuj () = 0;
};
```

Taka składnia, widziana po raz pierwszy, zdecydowanie wygląda dziwnie! Zapis ten ma jednak swoją logikę. Nadanie metodzie wartości 0 to sposób na wskazanie, że metoda ta nie istnieje. Kiedy klasa zawiera metodę czysto wirtualną, jej podklasy muszą tę metodę zaimplementować. W tym celu podklasa powinna zadeklarować daną metodę jeszcze raz, tym razem bez symbolu = 0, co znaczy, że klasa ta udostępni rzeczywistą implementację tej metody:

```
class Statek : public Rysowalny
{
public:
    virtual rysuj ();
};
```

Teraz naszą metodę można zdefiniować tak samo jak każdą inną zwykłą metodę:

```
Statek::rysuj ()  
{  
    /* Kod realizujący rysowanie */  
}
```

Możesz zapytać, po co w ogóle jest nam potrzebna taka nadklasa jak `Rysowalny`, skoro metoda `rysuj` nie będzie mieć żadnej implementacji. Rzec w tym, że potrzebujemy nadklasy w celu zdefiniowania interfejsu, który zostanie zaimplementowany we wszystkich podklasach. Wówczas będziemy mogli napisać kod spodziewający się interfejsu klasy `Rysowalny` bez potrzeby znajomości, jaki dokładnie rodzaj klasy został użyty. Niektóre języki programowania umożliwiają przekazywanie dowolnych obiektów do dowolnych funkcji i jeśli tylko obiekt implementuje metody użyte w danej funkcji, wszystko będzie działać. Język C++ wymaga jednak, aby funkcja jawnie określała interfejsy swoich argumentów. Bez interfejsu klasy `Rysowalny` nie moglibyśmy nawet umieścić naszych klas w tym samym wektorze — nie byłoby żadnej „wspólnej” cechy, która pozwoliłaby nam zidentyfikować, co powinno się w tym wektorze znaleźć. Zobaczmy, jak wygląda kod korzystający z naszego wektora i rysujący wszystkie obiekty:

```
vector<Rysowalny*> rysowalne;  
// Zapisz Statek w wektorze, tworząc nowy wskaźnik do obiektu Statek  
rysowalne.push_back( new Statek() );  
  
for ( vector<int>::iterator itr = rysowalne.begin(), koniec = rysowalne.end();  
      itr != koniec; ++itr )  
{  
    // Pamiętaj, że jeśli mamy wskaźnik do obiektu, w celu  
    // wywoływania metod musimy stosować składnię z symbolem ->  
    (*itr)->rysuj(); // Wywołanie metody Statek::rysuj  
}
```

Do wektora możemy dodawać różne rodzaje obiektów klasy `Rysowalny` (założymy, że mamy klasę `Wrog`, która także dziedziczy po klasie `Rysowalny`):

```
rysowalne.push_back( new Statek() );  
rysowalne.push_back( new Wrog() );
```

Wszystko to będzie działać. Dla statków będziemy wywoływać metodę `Statek::rysuj`, a dla wrogów metodę `Wrog::rysuj`.

A tak przy okazji — bardzo ważne jest, abyśmy zamiast wektora `vector<Rysowalny>` mieli wektor `vector<Rysowalny*>`. Bardzo ważne jest użycie wskaźnika; bez niego żadna z opisanych tu funkcji by nie działała.

Aby zrozumieć, dlaczego tak jest, wyobraź sobie na chwilę, że napisaliśmy kod, który odwołuje się do obiektu bez pośrednictwa wskaźnika:

```
vector<Rysowalny> rysowalne;
```

W pamięci będziemy mieć trzy różne obiekty klasy `Rysowalny`, każdy o tej samej wielkości:

```
[ Rysowalny 1 ][ Rysowalny 2 ][ Rysowalny 3 ]
```

Jeśli wektor nie korzysta ze wskaźnika, powinien przechowywać całe obiekty. Nasze obiekty niekoniecznie muszą mieć takie same wielkości. Obiekty klasy `Statek` i `Wrog` mogą mieć różne pola i oba mogą być mniejsze od klasy `Rysowalny`. W takiej sytuacji kod nie będzie poprawnie działać.

Z drugiej jednak strony wskaźniki zawsze mają taką samą wielkość<sup>4</sup>. Możemy napisać:

[Wskaźnik do Rysowalny] [Wskaźnik do Rysowalny] [Wskaźnik do Rysowalny]

Jeśli będziemy mieć wskaźnik do obiektu Statek, zajmie on dokładnie tyle samo pamięci co wskaźnik do obiektu Rysowalny. Z tego powodu piszemy:

```
vector<Rysowalny*> rysowalne;
```

Teraz możemy już wstawić do wektora taki rodzaj wskaźnika, jaki tylko zechcemy, o ile wskaźnik ten odnosi się do klasy dziedziczącej po Rysowalny. Każdy z tych obiektów zostanie w pętli narysowany na ekranie za pomocą metody rysuj danej podklasy (technicznie do wektora pasuje dowolny wskaźnik, co jednak nie oznacza automatycznie, że skoro coś pasuje, to chcielibyśmy to mieć w naszym wektorze. Celem tego wektora jest przechowywanie listy obiektów, które można wyświetlić na ekranie. Umieszczenie w nim czegoś, czego nie da się narysować, narobiłoby nam kłopotów).

Zapamiętaj, że jeśli chcesz, aby klasa dziedziczyła interfejs po nadklasie, należy tę klasę przekazywać poprzez wskaźnik.

Teraz, kiedy poznaliśmy już najważniejsze szczegóły dotyczące naszego przykładu, cofnijmy się o krok i przyjrzyjmy się temu, co zrobiliśmy:

1. Najpierw zdefiniowaliśmy interfejs Rysowalny, który mogą dziedziczyć podklasy.
2. Dowolna funkcja może przyjąć obiekt klasy Rysowalny i dowolny kod może pracować z jej interfejsem. Kod ten może wywoływać metodę rysuj zaimplementowaną w określonym obiekcie, który jest wskazywany przez wskaźnik.
3. Dzięki temu istniejący kod może korzystać z nowych obiektów, o ile implementują one interfejs klasy Rysowalny. Do naszej menażerii możemy dodawać nowe elementy — ikony oznaczające ulepszenia albo dodatkowe życia, obrazy teł i cokolwiek tylko zechcemy, a kod, który je przetwarza, nie będzie musiał nic o nich wiedzieć, z wyjątkiem tego, że należą one do klasy Rysowalne.

Wszystko to sprowadza się do wielokrotnego użycia kodu. Wynika ono z możliwości współpracy istniejącego kodu z nowo utworzonymi klasami. Nowe klasy można pisać w taki sposób, aby współdziałały z istniejącym kodem (takim jak pętla, która rysuje każdy element gry) bez potrzeby modyfikowania go po to, aby wiedział o nowych klasach (co prawda musimy dodać obiekty nowej klasy do naszego wektora Rysowalny, ale sama pętla nie ulega zmianom).

Zachowanie takie określone jest nazwą **polimorfizm**. *Poli* oznacza wiele, a *morf* pochodzi od słowa *forma*, co zestawione razem znaczy *wiele form*. Innymi słowy, każda klasa implementująca określony interfejs stanowi jedną formę, a ponieważ kod, który został napisany w celu korzystania z tego interfejsu, potrafi obsługiwać wiele różnych klas, może on radzić sobie z wieloma formami tego interfejsu, tak samo jak osoba potrafiąca prowadzić może jeździć autem zasilanym benzyną, samochodem hybrydowym lub pojazdem elektrycznym.

---

<sup>4</sup> Jak na nasze potrzeby, stwierdzenie to jest dość bliskie prawdy. W niektórych komputerach mogą istnieć różne wskaźniki do różnych typów danych, ale nie będziemy się tym teraz przejmować. Jeśli jesteś zainteresowany tym tematem, więcej informacji o takich sytuacjach znajdziesz na stronie <http://stackoverflow.com/questions/1241205/are-all-data-pointers-of-the-same-size-in-one-platform>. W języku polskim informację na temat różnych rozmiarów wskaźników można znaleźć na przykład na stronie [http://pl.wikibooks.org/wiki/C/Wska%C5%82niki\\_-wi%C4%85cej](http://pl.wikibooks.org/wiki/C/Wska%C5%82niki_-wi%C4%85cej).

## Pozostałe zastosowania oraz nieprawidłowe użycia dziedziczenia

Polimorfizm bazuje na dziedziczeniu, a z dziedziczenia można korzystać nie tylko w celu powielania interfejsu. Jak już wcześniej dawałem do zrozumienia, możliwe jest użycie dziedziczenia w celu przejęcia implementacji funkcji.

Gdyby na przykład interfejs klasy `Rysowny` miał kolejną metodę niewirtualną, byłaby ona dziedziczona przez wszystkie obiekty implementujące tę klasę. Niektórzy programiści uważają, że dziedziczenie sprowadza się do ponownego korzystania z kodu dzięki przejmowaniu metod (co zapobiega konieczności pisania metod dla każdej podklasy). Jest to jednak raczej ograniczona forma wielokrotnego użycia. Z pewnością możesz zaoszczędzić sobie trudu, dokonując pełnego dziedziczenia implementacji metody, tylko że w takim przypadku stanieś przed sporym wyzwaniem: jak zagwarantować, że implementacja tej metody będzie poprawna we wszystkich podklasach? Rozwiążanie tego problemu będzie wymagać starannego przemyślenia, które elementy zawsze pozostają prawidłowe.

Zobaczmy, dlaczego jest to trudne. Wyobraź sobie, że masz klasy `Gracz` i `Statek`, z których obie implementują interfejs `Rysowny` i obie mają metodę `pobierzNazwe`. Możesz podjąć decyzję o dodaniu metody `pobierzNazwe` do klasy `Rysowny`, dzięki czemu klasy `Gracz` i `Statek` mogłyby dzielić taką samą implementację tej metody.

```
class Rysowny
{
public:
    string pobierzNazwe ();
    virtual void rysuj () = 0;
};
```

Ponieważ metoda `pobierzNazwe` nie jest wirtualna, wszystkie podklasy odziedziczą jej implementację. Co się stanie, jeśli zadecydujesz o dodaniu nowej klasy, takiej jak `Pocisk`, której obiekty także mają być rysownalne? Czy każdy pocisk rzeczywiście musi mieć własną nazwę? Oczywiście, że nie! Może się wydawać, że istnienie w klasie `Pocisk` zbędnej metody `pobierzNazwe` nie będzie wielkim problemem — przecież jedna zła metoda nie oznacza od razu końca świata. Problem jednak polega na tym, że wielokrotne powtórzenie takiego zabiegu doprowadzi do powstania zagmatwanych i skomplikowanych hierarchii klas, w których zadania interfejsu nie będą do końca zrozumiałe.

## Dziedziczenie, konstruowanie obiektów oraz ich niszczenie

Kiedy następuje dziedziczenie po nadklasie, konstruktor podklasy wywołuje konstruktor nadklasy, tak samo jak wywołuje on konstruktory dla wszystkich pól klasy.

Spójrzmy na następujący kod:

### Przykładowy kod 59.: konstruktor.cpp

```
#include <iostream>

using namespace std;

class Foo // Foo to powszechnie używana w programowaniu nazwa zastępcza
```

```

{
public:
    Foo () { cout << "Konstruktor klasy Foo" << endl; }
};

class Bar : public Foo
{
public:
    Bar () { cout << "Konstruktor klasy Bar" << endl; }
};

int main ()
{
    // Uroczy słonik ;
    Bar bar;
}

```

Kiedy inicjalizowany jest obiekt bar, najpierw uruchomiony zostanie konstruktor klasy Foo, a następnie konstruktor klasy Bar. Wynikiem działania kodu są komunikaty:

```

Konstruktor klasy Foo
Konstruktor klasy Bar

```

Wcześniej wywołanie konstruktora nadklasy umożliwia zainicjalizowanie wszystkich pól nadklasy przed ich użyciem przez konstruktor podklasy. Taki zabieg gwarantuje, że podkla- sa będzie mogła korzystać z pól nadklasy, wiedząc, że są one już zainicjalizowane.

Wszystko to automatycznie realizuje kompilator — nie musisz nic robić, aby został wywo- lany konstruktor nadklasy. Podobnie, po zakończeniu działania destruktora podklasy zosta- nie automatycznie uruchomiony destruktor nadklasy. Oto przykład kodu demonstrującego kolejność wywoływanego destruktorów:

### **Przykładowy kod 60.: destruktor.cpp**

```

#include <iostream>

using namespace std;

class Foo // Foo to powszechnie używana w programowaniu nazwa zastępcza
{
public:
    Foo () { cout << "Konstruktor klasy Foo" << endl; }
    ~Foo () { cout << "Destruktor klasy Foo" << endl; }
};

class Bar : public Foo
{
public:
    Bar () { cout << "Konstruktor klasy Bar" << endl; }
    ~Bar () { cout << "Destruktor klasy Bar" << endl; }
};

int main ()
{
    // Uroczy słonik ;
    Bar bar;
}

```

W tym przypadku wynik działania programu wygląda następująco:

```
Konstruktor klasy Foo  
Konstruktor klasy Bar  
Destruktor klasy Bar  
Destruktor klasy Foo
```

Zwróc uwagę, że destruktory są wywoływane w odwrotnej kolejności względem konstruktorów. Dzięki temu destruktor klasy Bar może bezpiecznie użyć metod dziedziczonych po klasie Foo, gdyż dane, na których działają te metody, są nadal aktualne i zdatne do użycia. Taka sposób działania jest bardzo podobny do logiki kryjącej się za wywoływaniem konstruktora nadklasy przed konstruktorem podklasy.

W niektórych przypadkach wołabyś wywołać w nadklasie konstruktor, który nie jest domyślny. Umożliwia to lista inicjalizacyjna, na której należy umieścić nazwę nadklasy.

```
class NadklasaFoo  
{  
public:  
    NadklasaFoo (const string& wart);  
};  
  
class Foo : public NadklasaFoo  
{  
public:  
    Foo ()  
        : NadklasaFoo ( "arg" ) // Przykładowa lista inicjalizacyjna  
    {}  
};
```

Wywołanie konstruktora nadklasy powinno nastąpić przed definicją pól listy inicjalizacyjnej.

## Polimorfizm i dziedziczenie obiektów

Skomplikowaną sprawą jest niszczenie obiektu i sposób, w jaki ono przebiega za pośrednictwem interfejsu. Możesz mieć na przykład taki kod:

```
class Rysowalny  
{  
public:  
    virtual void rysuj () = 0;  
};  
  
class MojRysowalny : public Rysowalny  
{  
public:  
    virtual void rysuj ();  
    MojRysowalny ();  
    ~MojRysowalny ();  
  
private:  
    int *_moje_dane;  
};  
  
MojRysowalny::MojRysowalny ()  
{  
    _moje_dane = new int;  
}
```

```

MojRysowalny::~MojRysowalny ()
{
    delete _moje_dane;
}

void usunRysowalny (Rysowalny *rysowalny)
{
    delete rysowalny;
}

int main ()
{
    usunRysowalny( new MojRysowalny() );
}

```

Co się zatem dzieje w metodzie `usunRysowalny`? Na pewno pamiętasz, że destruktor jest wywoływany w trakcie użycia instrukcji `delete`, tak więc w wierszu

```
delete rysowalne;
```

następuje wywołanie tej funkcji dla obiektu. Tylko skąd kompilator będzie wiedzieć, jak znaleźć destruktor dla klasy `MojRysowalny`? Nie zna on dokładnego typu zmiennej `rysowalny`, wie tylko, że jest to `Rysowalny` — coś z metodą o nazwie `rysuj`. Wie, jak znaleźć destruktor skojarzony z klasą `Rysowalny`, ale nie potrafi znaleźć destruktora klasy `MojRysowalny`. Niestety, ponieważ klasa `MojRysowalny` alokuje w swoim konstruktorze pamięć, ważne jest uruchomienie jej destruktora w celu zwolnienia tej pamięci.

Być może zastanawiasz się, czy takich problemów nie powinna rozwiązywać funkcja wirtualna? Faktycznie, powinna! Wszystko, co musimy zrobić, to zadeklarować destruktor w klasie `Rysowalne` jako wirtualny, dzięki czemu kompilator będzie wiedzieć, że kiedy dla wskaźnika do obiektu klasy `Rysowalne` zostanie wywołana metoda `delete`, powinien on szukać przeładowanego destruktora.

```

class Rysowalny
{
public:
    virtual void rysuj ();
    virtual ~Rysowalny ();
};

class MojRysowalny : public Rysowalny
{
public:
    virtual void rysuj ();
    MojRysowalny ();
    virtual ~MojRysowalny ();

private:
    int *_moje_dane;
};

```

Po oznaczeniu destruktora w nadklasie jako wirtualnego, za każdym razem, kiedy interfejs klasy `Rysowalny` będzie zwalniany za pomocą metody `delete`, wywoływany będzie destruktor przeładowany.

Ogólną zasadą jest, że kiedy definiujesz jakąkolwiek metodę w nadklasie jako wirtualną, destruktor nadklasy także powinien być wirtualny. Gdy już oznaczysz pierwszą metodę jako

wirtualną, oznajmiasz, że do metod przyjmujących interfejs można przekazywać klasy. Metody te mogą robić, co tylko chcą, w tym usuwać obiekty, w związku z czym destruktor powinien być wirtualny, aby istniała pewność, że obiekt zostanie prawidłowo posprzątany.

## Problem przycinania

**Problem przycinania** jest kolejną kwestią, z której należy sobie zdawać sprawę podczas pracy z dziedziczeniem. Zjawisko to zachodzi wówczas, gdy Twój kod wygląda mniej więcej tak:

```
class Nadklasa
{};

class Podklasa : public Nadklasa
{
    int wart;
};

int main()
{
    Podklasa pod;
    Nadklasa nad = pod;
}
```

Pole `wart` klasy `Podklasa` nie zostanie skopiowane podczas operacji przypisania do obiektu nad! Niestety, zazwyczaj nie o to chodzi (choć C++ na coś takiego pozwala), ponieważ obiekt zostanie przekopiowany tylko częściowo. Tego typu kod może zadziałać, chociaż często może doprowadzić do awarii programu<sup>5</sup>.

Na szczęście istnieje sposób na to, aby kompilator poinformował Cię w przypadku wystąpienia takiej sytuacji. Otóż można zadeklarować konstruktor kopiący klasy `Nadklasa` jako prywatny i nie implementować go:

```
class Nadklasa
{
public:
    // Zwrócić uwagę, że ponieważ deklarujemy konstruktor kopiący,
    // musimy udostępnić własny konstruktor domyślny
    Nadklasa () {}

private:
    // Zabronione — nie będziemy definiować tej metody
    Nadklasa (const Nadklasa& inny);
};

class Podklasa : public Nadklasa
{
    int wart;
};

int main()
{
    Podklasa pod;
    Nadklasa nad = pod; // Teraz ten wiersz wywoła błąd kompilacji
}
```

---

<sup>5</sup> W szczególności, gdy klasa zawiera funkcje wirtualne, które oczekują, że pola podklasy będą wypełnione.

Ale co zrobić w sytuacji, w której jednak chciałbyś mieć konstruktor kopiący? Inny sposób na uniknięcie tego problemu polega na takim deklarowaniu nadklas, aby każda z nich miała przynajmniej jedną funkcję czysto wirtualną. Dzięki temu masz gwarancję, że jeśli napiszesz:

Nadklaśa nad;

kod taki nie skompiluje się, ponieważ za pomocą funkcji czysto wirtualnych nie można tworzyć obiektów. Z drugiej jednak strony nadal możesz napisać:

Nadklaśa \*nad = & pod;

Dzięki czemu skorzystasz z zalet polimorfizmu bez natrafiania na problem przycinania.

## Dzielenie kodu z podklasami

Do tej pory mówiłem zarówno na temat publicznej, jak i prywatnej ochrony. Metody publiczne są dostępne dla każdego elementu spoza klasy, natomiast dane i metody prywatne są dostępne tylko dla innych metod i danych klasy.

Co jednak zrobić w sytuacji, w której chciałbyś, aby nadklaśa udostępniała metody, które mogą być wywoływanie przez podklasy, ale już nie przez klasy zewnętrzne? Jest to możliwe. Współdzielenie przez podklasy fragmentów kodu implementacyjnego jest spotykane dość często.

Wyobraź sobie na przykład, że masz metodę, która poprzez czyszczenie pewnego obszaru ekranu pomaga w wyświetlaniu obiektów. Nazwiemy tę metodę `czyscObszar`:

```
class Rysowny
{
public:
    virtual void rysuj ();
    virtual ~Rysowny ();
    void czyscObszar (int x1, int y1, int x2, int y2);
};
```

W tym przypadku celem korzystania z dziedziczenia nie jest przekazanie interfejsu, tylko umożliwienie podklasom uzyskania dostępu do wspólnego kodu implementacyjnego. Takie zastosowanie dziedziczenia jest prawidłowe, gdyż każda z podklas musi korzystać z metody `czyscObszar`, lub przynajmniej powinna mieć taką możliwość. Nie chcemy, aby była ona częścią publicznego interfejsu klasy, a tylko stanowiła szczegół implementacji tworzonej hierarchii klas.

Jak jednak możemy zapobiec sytuacji, w której metoda ta stałaaby się częścią interfejsu klasy? Jeśli oznaczmy ją jako publiczną, jak to pokazano powyżej, wszyscy będą mogli ją wywołać, a przecież tak naprawdę całego tego nie chcemy. Z drugiej jednak strony nie możemy zadeklarować tej metody jako prywatnej, ponieważ podklasy nie mają dostępu do prywatnych pól ani metod, a zablokowanie podklasom takiego dostępu całkowicie mija się z naszym celem!

## Dane chronione

Odpowiedzią jest skorzystanie z trzeciego i ostatniego już modyfikatora dostępu, którym jest `protected` (chronione). W przeciwieństwie do metod prywatnych, metody zadeklarowane w sekcji chronionej będą dostępne dla podklas, ale w odróżnieniu do metod publicznych, nie będą dostępne poza klasą. Składnia deklarowania metod chronionych jest taka sama jak w przypadku metod publicznych i prywatnych:

```
class Rysowalny
{
public:
    virtual void rysuj ();
    virtual ~Rysowalny ();
protected:
    void czyscObszar (int x1, int y1, int x2, int y2);
};
```

Teraz dostęp do metody `czyscObszar` będą mieć wyłącznie podklasy wywodzące się z klasy `Rysowalny`.

Metody chronione są używane dość często, chociaż nie zawsze zalecam ich stosowanie. Nie ma potrzeby, aby cała hierarchia klasy miała pełny dostęp do danych, z tego samego powodu, dla którego nie udostępniasz tych danych wszystkim wokół — po prostu chcesz pozostawić sobie możliwość ich późniejszego modyfikowania. Z metod chronionych powinieneś korzystać właśnie w celu udostępniania tych danych w podklasach.

## Dane obejmujące całą klasę

Wszystko, co do tej pory mogłeś zrobić z klasą, sprowadzało się do przechowywania danych w poszczególnych jej instancjach. W wielu przypadkach takie zastosowanie klas w zupełności wystarcza, ale czasami chciałbyś mieć możliwość przechowywania danych, które nie są specyficzne dla określonego obiektu, ale dla całej klasy jako takiej. Na przykład utworzyłeś klasę wymagającą, aby każdy z jej obiektów miał swój niepowtarzalny numer seryjny. W jaki jednak sposób przechowasz następny numer seryjny, który powinien zostać nadany? Na poziomie klasy powinieneś dysponować jakimś miejscem, w którym można taką informację zapisać, dzięki czemu w przypadku tworzenia nowego obiektu będziesz wiedzieć, jaki numer należy mu nadać. Dlaczego miałbyś robić coś takiego? Na przykład dlatego, że korzystanie z seryjnych numerów obiektów ułatwia ich identyfikowanie w dziennikach zdarzeń. Numer seryjny może być użyty w celu odszukania danego obiektu wśród licznych wierszy pliku dziennika zdarzeń.

Dane obejmujące całą klasę można tworzyć za pomocą składowych klasy, które są **statyczne**. W przeciwieństwie do zwykłych danych instancji, dane statyczne nie stanowią części żadnego z poszczególnych obiektów; są one dostępne we wszystkich obiektach klasy, a jeśli są publiczne, będą dostępne wszędzie. Zmienne statyczne są podobne do zmiennych globalnych, z tym że w celu uzyskania dostępu do zmiennej statycznej spoza klasy należy poprzedzić nazwę tej zmiennej nazwą klasy.

Zobaczmy, jak to zrobić. Oto klasa z deklaracją zmiennej statycznej:

```
class Wezel
{
public:
    static int numer_seryjny;
};

// Jesteśmy poza deklaracją klasy i dlatego musimy
// użyć prefiksu Wezel::
static int Wezel::numer_seryjny = 0;
```

Oprócz zmiennych statycznych mogą istnieć także statyczne metody. Są to metody stanowiące część klasy, ale z których można korzystać bez powoływanego obiektu tej klasy. Zajmijmy się możliwością nadawania każdemu węzłowi numeru seryjnego dzięki dodaniu prywatnej metody statycznej o nazwie `_pobierzNastepnyNumerSeryjny`.

```

class Wezel
{
public:
    Wezel ();

private:
    static int _pobierzNastepnyNumerSeryjny();

    // Statyczny — jedna kopia dla całej klasy
    static int _nastepny_numer_seryjny;

    // Niestatyczny — dostępny w każdym obiekcie, ale nie w metodach statycznych
    int _numer_seryjny;
};

// Jesteśmy poza deklaracją klasy i dlatego musimy
// użyć prefiksu Wezel:
static int Wezel::numer_seryjny = 0;

Wezel::Wezel ()
: _numer_seryjny( _pobierzNastepnyNumerSeryjny() )
{ }

int Wezel::_pobierzNastepnyNumerSeryjny()
{
    // Aby zwrócić wartość znajdująca się poprzednio w zmiennej,
    // korzystamy z przyrostkowej wersji operatora ++
    return _nastepny_numer_seryjny++;
}

```

Pamiętaj tylko, że kiedy korzystasz z metody statycznej, nie będzie ona mieć dostępu do pól charakterystycznych dla obiektu, chociaż stanowi część klasy. Ma ona dostęp jedynie do danych statycznych klasy. Do metody statycznej nie jest przekazywany wskaźnik this.

## W jaki sposób zaimplementowany jest polimorfizm?

Sposób, w jaki kompilator implementuje polimorfizm, stanowi zaawansowane zagadnienie, którego lektura zaprowadzi Cię głęboko w szczegóły tego obszaru języka C++. Zamieszczam ten podrozdział, ponieważ technika zastosowana w celu realizacji tego zagadnienia jest bardzo ciekawa i dlatego chciałbym Cię z nią zaznajomić. Nie jest istotne, abyś nauczył się tego wszystkiego już przy swoim pierwszym (ewentualnie drugim) kontakcie z polimorfizmem. Jeśli ciekawi Cię, jak zaimplementowana jest magia polimorfizmu, czytaj dalej; jeżeli jednak od lektury rozboli Cię głowa, nie wysilaj się. Zawsze będziesz mógł powrócić do tego podrozdziału, gdy staniesz przed potrzebą lepszego zrozumienia opisanych tu kwestii.

Główna idea polimorfizmu polega na tym, że gdy kompilator napotyka funkcję, która nie działa z konkretną podklassą, lecz z interfejsem, nie wie dokładnie, jaki kod maszynowy zostanie uruchomiony. Spójrzmy na przykład na poniższy kod:

```

vector<Rysowalny*> rysowalne;

void rysujWszystko ()
{
    for ( int i = 0; i < rysowalne.size(); i++ )
    {

```

```

        rysowalne[ i ]->rysuj();
    }
}

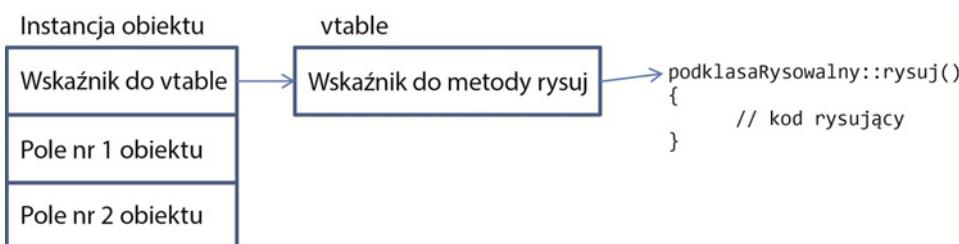
```

Instrukcja `rysowalne[ i ]->rysuj()` nie może zostać przełożona podczas komplikacji na wywołanie konkretnej funkcji, ponieważ metoda `rysuj` jest wirtualna. W zależności od tego, który obiekt został odziedziczony po klasie `Rysowalny`, może on zawierać rozmaite metody: rysującą pocisk, statek kosmiczny użytkownika, wrogi statek albo symbol ulepszenia broni.

Co więcej, metoda `rysuj` wszysktko uruchamia kod, o którym nic nie wie. Kod wywołujący metodę `rysuj` musi widzieć tylko interfejs klasy `Rysowalny`. Nie musi nic wiedzieć na temat metod implementujących tę klasę. Jak jednak można to zrealizować, jeśli metoda będzie wywoływana w podklasie klasy `Rysowalne`?

W obiekcie, jako ukryte pole, znajduje się lista metod wirtualnych. W tym przypadku lista ta zawiera tylko jedną pozycję z adresem metody `rysuj`. Każda metoda danego interfejsu ma przypisaną liczbę (metoda `rysuj` miałaby numer 0). Podczas wywoływanego metody wirtualnej liczba skojarzona z tą metodą jest traktowana jako indeks listy metod wirtualnych danego obiektu. Wywołanie metody wirtualnej jest kompilowane jako odszukanie tej metody na liście metod wirtualnych, po którym następuje uruchomienie odnalezionej metody. W powyższym kodzie wywołanie metody `rysuj` zostanie przekształcone na odszukanie metody nr 0 w tabeli metod, po którym nastąpi wywołanie kodu znajdującego się pod ustaloną w ten sposób adresem. Lista zawierająca metody wirtualne nazywana jest **tabelą metod wirtualnych** (w skrócie **vtable**, od **virtual table**).

A oto graficzne wyobrażenie tego procesu:



Ponieważ obiekt wyposażony jest we własną tabelę metod, z których może korzystać, kompilator podczas kompilowania różnych klas ma możliwość wprowadzania zmian w adresach zapisanych w tej tabeli w celu udostępnienia właściwej implementacji metody wirtualnej. Oczywiście nie musisz się tym zajmować osobiście; kompilator zrobi to wszystko za Ciebie. Kod korzystający z tabeli musi tylko znać dokładny indeks, pod którym jest zapisana każda z metod wirtualnych znajdujących się w tabeli.

Tabela metod wirtualnych zawiera wyłącznie metody zadeklarowane jako wirtualne. Metody niewirtualne nie potrzebują takiego mechanizmu, w związku z czym nie mają swoich wpisów w tabeli. Jeśli napiszesz klasę bez żadnych metod wirtualnych, klasa ta nie będzie zawierać tabeli metod wirtualnych.

Wywołanie metody wirtualnej jest równoznaczne z uruchomieniem kodu, który odczytuje tabelę metod wirtualnych i odszukuje metodę na podstawie jej indeksu. Napisanie

```
rysowalne[ i ]-.rysuj();
```

jest traktowane przez kompilator jak wydanie następujących poleceń:

1. Pobierz wskaźnik przechowywany w wektorze `rysowalne[ i ]`.
2. Na podstawie tego wskaźnika znajdź adres tabeli wirtualnej z grupą metod skojarzonych z interfejsem klasy `Rysowalny` (w tym przypadku istnieje tylko jedna taka metoda).
3. W tabeli z funkcjami znajdź funkcję o danej nazwie (czyli `rysuj`). Tabela ta jest zbiorem adresów odpowiadających położeniu w pamięci każdej funkcji.
4. Wywołaj funkcję z przypisanymi jej argumentami.

Krok 2. zwykle nie jest realizowany przy użyciu rzeczywistej nazwy funkcji, ale poprzez zamianę przez kompilator nazwy funkcji na jej indeks w tabeli. Dzięki temu w czasie działania programu wywołanie funkcji wirtualnej jest niewiarygodnie szybkie — różnica w wydajności między wywołaniem funkcji wirtualnej a wywołaniem zwykłej funkcji jest bardzo mała.

Kod generowany przez kompilator mógłbyś wyobrazić sobie następująco (oczywiście wymyśliłem składnię ze słowem kluczowym `wywołaj`):

```
wywołaj rysowalne[ i ]->vtable[ 0 ];
```

Z drugiej jednak strony istnieje pewna poważna wada związana z użyciem funkcji wirtualnych. Otóż obiekt musi zawierać po jednej tabeli metod wirtualnych na każdy odziedziczony interfejs. Oznacza to, że każdy interfejs wirtualny powiększa rozmiary danego obiektu o kilka bajtów. W praktyce można się tym przejmować tylko w przypadku kodu, który generuje dużo obiektów z małą liczbą zmiennych składowych.

## Sprawdź się

1. Kiedy uruchamiany jest destruktor nadklasy?
  - A. Tylko wtedy, gdy obiekt jest niszczyony przez wywołanie instrukcji `delete` ze wskaźnikiem nadklasy.
  - B. Przed wywołaniem destruktora podklasy.
  - C. Po wywołaniu destruktora podklasy.
  - D. Razem z wywołaniem destruktora podklasy.
2. Mając do czynienia z poniższą hierarchią klas, co musiałbyś zrobić w konstruktorze klasy `Kot`?
 

```
class Ssak {
public:
    Ssak (const string& nazwa_gatunku);
};

class Kot : public Ssak
{
public:
    Kot();
};
```

- A. Nic szczególnego.
- B. Skorzystać z listy inicjalizacyjnej w celu wywołania konstruktora klasy `Ssak` z argumentem `kot`.
- C. Wywołać konstruktor klasy `Ssak` z poziomu konstruktora klasy `Kot` z argumentem `kot`.
- D. Usunąć konstruktor klasy `Kot` i skorzystać z konstruktora domyślnego, który rozwiąże ten problem za Ciebie.

**3.** Co jest nie tak z poniższą definicją klasy?

```
class Nazywalny
{
    virtual string pobierzNazwe();
}
```

- A. Nie deklaruje ona metody pobierzNazwe jako publicznej.
  - B. Nie ma wirtualnego destruktora.
  - C. Nie zawiera implementacji metody pobierzNazwe, ale przy tym nie deklaruje jej jako czysto wirtualną.
  - D. Wszystkie odpowiedzi są prawidłowe.
- 4.** Kiedy deklarujesz metodę wirtualną w klasie interfejsowej, co powinna zrobić funkcja, aby mogła korzystać z metody interfejsu w celu wywołania metody w podklasie?
- A. Pobrać interfejs jako wskaźnik (albo referencję).
  - B. Nic, może po prostu skopiować obiekt.
  - C. Aby wywołać metodę, powinna znać nazwę podklasy.
  - D. O co chodzi? Co to jest metoda wirtualna?
- 5.** W jaki sposób dziedziczenie polepsza wielokrotne użycie kodu?
- A. Pozwalając, aby kod podklasy dziedziczył metody z nadklasy.
  - B. Pozwalając, aby nadklasa implementowała metody wirtualne dla swojej podklasy.
  - C. Pozwalając na pisanie kodu oczekującego interfejsu zamiast konkretnej klasy, co umożliwia nowym klasom implementowanie tego interfejsu i korzystanie z istniejącego kodu.
  - D. Pozwalając, aby nowe klasy dziedziczyły cechy konkretnej klasy, które można wykorzystać w metodach wirtualnych.
- 6.** Które z poniższych zdań poprawnie opisuje poziomy dostępności klas?
- A. Podklasa ma dostęp wyłącznie do publicznych metod i danych swojej klasy nadrzędnej.
  - B. Podklasa ma dostęp do prywatnych metod i danych swojej klasy nadrzędnej.
  - C. Podklasa ma dostęp wyłącznie do chronionych metod i danych swojej klasy nadrzędnej.
  - D. Podklasa ma dostęp do chronionych i publicznych metod oraz danych swojej klasy nadrzędnej.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1.** Zaimplementuj funkcję sortuj pobierającą wektor wskaźników do klasy interfejsowej Porownywalne, która definiuje metodę porownaj(Porownywalne& inny) i zwraca 0, kiedy obiekty są takie same, 1, kiedy porównywany obiekt jest większy od obiektu inny, oraz -1, gdy obiekt jest mniejszy od obiektu inny. Napisz klasę implementującą ten interfejs, utwórz kilka jej instancji i posortuj je. Jeśli nie wiesz, jaką klasę utworzyć, spróbuj zmierzyć się z klasą Tabellawynikow, która zawiera nazwy oraz punktację i sortuje dane w taki sposób, że najwyższe punktacje znajdują się na początku. Jeśli dwie punktacje są takie same, sortowanie powinno odbywać się według nazw.

2. Utwórz kolejną implementację swojej funkcji sortującej, tym razem przyjmującą interfejs o nazwie Komparator z metodą porownaj (const string& lewy, const string& prawy), która działa zgodnie z zasadami opisanymi w poprzednim zadaniu: zwraca 0, jeśli obie wartości są równe, 1, jeśli lewy > prawy, i -1, jeżeli lewy < prawy. Napisz dwie różne klasy dokonujące porównań: pierwszą, która przeprowadza porównania z pominięciem wielkości liter, i drugą, która sortuje w odwrotnej kolejności alfabetycznej.
3. Zaimplementuj metodę logującą, która jako argument przyjmuje klasę interfejsową KonwertowaIneNaString. Klasa ta zawiera metodę naString, która konwertuje obiekt na jego reprezentację łańcuchową. Metoda logująca powinna wyświetlać datę i czas oraz sam obiekt. Informacje na temat odczytywania daty możesz znaleźć na przykład na stronie <http://cpp0x.pl/kursy/Kurs-C++/Biblioteka-time-h/321>. Także i w tym przypadku zwrócić uwagę na to, jak możemy wielokrotnie korzystać z metody logującej, implementując po prostu interfejs.



# 27

## ■ ■ ■ ROZDZIAŁ 27

## Przestrzenie nazw

---

Tworząc coraz więcej klas, możesz zacząć się zastanawiać, czy ktoś już nie napisał kodu, który robi to samo, i czy możesz z niego korzystać. Czasami odpowiedź na takie pytanie będzie twierdząca. Wiele podstawowych algorytmów i struktur danych, takich jak listy powiązane albo drzewa binarne, zaimplementowano już w solidnej i nadającej się do wielokrotnego użycia postaci, z której możesz korzystać. Jeśli jednak korzystasz z kodu, który utworzył ktoś inny, powinieneś uważać, aby nie doprowadzić do konfliktu nazw.

Możesz na przykład napisać klasę `ListaPowiazana`, w której będzie zapisana implementacja listy powiązanej. Możliwe przy tym jest, że w kodzie, z którego korzystasz, znajduje się klasa o takiej samej nazwie, ale innej implementacji. W takim przypadku ktoś powinien się wycofać, gdyż nie mogą istnieć dwie klasy o takiej samej nazwie.

W celu uniknięcia tego typu konfliktów należy poszerzyć podstawową nazwę danego typu, tworząc **przestrzeń nazw**. Na przykład ja mógłbym umieścić swoją klasę z listą powiązaną o nazwie `ListaPowiazana` w przestrzeni nazw `com::cprogramming`, dzięki czemu pełna nazwa mojego typu wyglądałaby następująco: `com::cprogramming::ListaPowiazana`. Stosowanie przestrzeni nazw w znacznym stopniu zmniejsza ryzyko wystąpienia konfliktu nazw. Użyty operator `::` jest tym samym operatorem, z którego korzystaliśmy w celu uzyskania dostępu do statycznych metod klasy lub podczas deklarowania metody, z tym że zamiast zapewniać dostęp do elementów klasy, tym razem umożliwia korzystanie z elementów danej przestrzeni nazw.

Być może zastanawiasz się, dlaczego w kodzie standardowej biblioteki nie skorzystano z przestrzeni nazw, skoro stanowią one takie świetne rozwiązanie. Czy przestrzenie nazw nie wymuszają tylko pisania dłuższego kodu, nie dając nic w zamian?

Okazuje się, że miałeś już do czynienia z przestrzeniami nazw. Na początku każdego programu zamieszczony jest wiersz:

```
using namespace std;
```

Zapis ten zwalnia z obowiązku stosowania pełnych nazw podczas odwoływania się do takich obiektów jak `cin` albo `cout`. Gdybyśmy nie zamieścili powyższej instrukcji, musielibyśmy pisać `std::cin` albo `std::cout` za każdym razem, gdybyśmy chcieli z tych obiektów skorzystać. Technika ta dobrze się sprawdza, o ile tylko nie musisz korzystać z przestrzeni nazw w celu uniknięcia konfliktów w danym pliku, udostępniając wygodny skrót, który możesz stosować, jeśli wiesz, że kolizje nazw nie wystąpią. Jeśli tego typu konflikty pojawią się, wszystko, co trzeba zrobić, sprowadza się do pominięcia deklaracji `using namespace` i podawania pełnych nazw w pliku z kodem.

Zobaczmy, jak takie rozwiązanie działa w przypadku kodu z wcześniejszego przykładu. Jeśli mam dwie różne klasy o nazwie `ListaPowiazana`, w większości moich plików zamieszczę na

początku instrukcję `using namespace com::cprogramming`. Gdybym miał plik, który wywołuje konflikt nazw, zmieniłbym go w taki sposób, aby odnosił się do mojej klasy `ListaPowiazana` poprzez zapis `com::cprogramming::ListaPowiazana`. Nie musiałbym wszędzie zmieniać mojego kodu, a tylko pliki, w których korzystam z obu rodzajów klasy `ListaPowiazana`. W plikach tych stosowałbym pełne nazwy i usunąłbym instrukcję `using namespace com::cprogramming`.

Oto przykład na zadeklarowanie fragmentu kodu jako części należącej do przestrzeni nazw. W tym przypadku będzie to tylko jedna zmienna:

```
namespace cprogramming
{
    int x;
} //<-- Zwróć uwagę na brak średnika
```

Teraz w celu odwołania się do zmiennej `x` będę musiał użyć zapisu `::x` albo napisać:

```
using namespace cprogramming;
```

Jeśli tak zrobię, w plikach zawierających instrukcję `using namespace cprogramming` będę mógł pisać po prostu `x`.

Istnieje możliwość zagnieżdżania przestrzeni nazw, tj. umieszczania jednej przestrzeni wewnętrznej drugiej. Tego typu rozwiązywanie może być potrzebne, jeśli pracujesz w dużej firmie z dużą liczbą różnych jednostek, z których każda niezależnie od innych pracuje nad oprogramowaniem. W takich przypadkach dla zewnętrznej przestrzeni nazw mógłbyś stosować nazwę firmy, natomiast w każdej z jednostek korzystać z zewnętrznej przestrzeni nazw.

Oto prosty przykład deklaracji zagnieżdżonej przestrzeni nazw `com::cprogramming`:

```
namespace com {
    namespace cprogramming {
        {
            int x;
        }
    }
}
```

Teraz pełną nazwą zmiennej `x` jest `com::cprogramming::x` (w przykładzie tym nie stosowałem wcięć kodu dla każdej przestrzeni; gdybyś miał wiele przestrzeni nazw i wcinał wiersze dla każdej z nich, wcięcia wymknęłyby Ci się spod kontroli!).

Aby uzyskać dostęp do elementów tej przestrzeni nazw, mógłbyś napisać:

```
using namespace com::cprogramming;
```

Przestrzenie nazw są „otwarte”, co znaczy, że kod do przestrzeni nazw można wprowadzać w wielu plikach. Jeśli na przykład utworzysz plik nagłówkowy zawierający klasę, po czym umieścisztę klasę w przestrzeni nazw:

```
namespace com {
    namespace cprogramming {
        {
            class MojaKlasa
            {
                public:
                    MojaKlasa ();
            };
        }
    }
}
```

wówczas w innym pliku źródłowym będziesz mógł napisać:

```
#include "MojaKlasa.h"
```

```
namespace com {
```

```
namespace cprogramming
{
    MojaKlasa::MojaKlasa ()
    {}
}
```

W obu plikach będzie można dodawać kod w obrębie przestrzeni nazw, dokładnie tak jak tego chciałeś.

## Kiedy stosować instrukcję using namespace

Zazwyczaj nie powinieneś umieszczać deklaracji `using` w plikach nagłówkowych, a tylko w plikach `.cpp`. Problem jednak polega na tym, że w każdym pliku korzystającym z nagłówka może wystąpić konflikt nazw. Wszystkie poszczególne pliki `.cpp` powinny kontrolować przestrzenie nazw, z których korzystają. Zwykle zalecam stosowanie pełnych nazw w plikach nagłówkowych i zamieszczanie deklaracji `using` wyłącznie w obrębie plików `.cpp`.

Istnieją pewne, dobrze znane odstępstwa od tej reguły. Sama biblioteka standardowa ją narusza, chociaż ma ku temu dobry powód.

Jeśli napiszesz

```
#include <iostream.h>
```

zamiast

```
#include <iostream>
```

nie będziesz mógł korzystać z deklaracji `using` w odniesieniu do przestrzeni nazw `std`. Okazuje się, że plik `iostream.h` to w zasadzie:

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

Służy to zachowaniu wstępnej zgodności z programami utworzonymi przed wprowadzeniem do języka przestrzeni nazw. Dzięki temu, jeśli masz taki program:

### Przykładowy kod 61.: *iostream\_h.cpp*

```
#include <iostream.h>
```

```
int main ()
{
    cout << "Hello world";
}
```

nadal będziesz mógł go skompilować po dodaniu przestrzeni nazw do biblioteki standardowej.

Zalecam, aby w nowym kodzie stosować nowszy plik nagłówkowy (bez rozszerzenia `.h`), dzięki czemu unikniesz takiego zaśmiecania przestrzeni nazw. Umieszczenie zapisu `namespace std;` w każdym pliku nie kosztuje wiele, a zabieg ten zapewni Ci korzystanie z aktualnej wersji C++.

## Kiedy należy utworzyć przestrzeń nazw?

Najczęściej, jeśli pracujesz nad programem, który składa się zaledwie z kilku plików, tworzenie własnych przestrzeni nazw jest po prostu zbędne. Przestrzenie nazw są przewidziane do stosowania podczas tworzenia programów złożonych z dziesiątek lub setek plików rozmiieszczonych w licznych katalogach, gdy wystąpienie konfliktów nazw rzeczywiście jest prawdopodobne.

Szybkie, jednorazowe lub kilkuplikowe programy tak naprawdę nie wymagają własnych przestrzeni nazw. Sugeruję, abyś zaczął umieszczać swój kod w przestrzeni nazw, jeśli myślisz o wielokrotnym korzystaniu z niego w przyszłości lub gdy Twój program rozrasta się w takim stopniu, że musisz porozdzielać jego pliki pomiędzy różne katalogi. Kiedy Twój kod osiągnie taki stopień złożoności, powinieneś sięgnąć po wszystkie dostępne narzędzia, które umożliwiają Ci jego organizowanie.

Chociaż przestrzenie nazw z pewnością nie są najważniejszą funkcjonalnością C++, jaką poznaleś, są one bardzo przydatne podczas pracy z dużymi bazami kodu. Zrozumienie, do czego służą przestrzenie nazw i dlaczego korzystają z nich inni programiści, pozwoli Ci integrować cudzy kod z własnym.

## Sprawdź się

- 1. Kiedy powinieneś korzystać z dyrektywy `namespace`?**
  - A. We wszystkich plikach nagłówkowych, zaraz po `include`.
  - B. Nigdy, dyrektywa ta jest niebezpieczna.
  - C. Na początku każdego pliku cpp, gdy nie występuje konflikt przestrzeni nazw.
  - D. Bezpośrednio przed użyciem zmiennej z danej przestrzeni nazw.
- 2. Do czego potrzebne są przestrzenie nazw?**
  - A. Aby twórcy kompilatorów mieli ciekawą pracę.
  - B. Aby zapewnić lepszą hermetyzację w kodzie.
  - C. Aby zapobiec konfliktom nazw w dużych bazach kodu.
  - D. Aby ułatwić zrozumienie, do czego służy dana klasa.
- 3. Kiedy należy umieszczać kod w przestrzeni nazw?**
  - A. Zawsze.
  - B. Kiedy piszesz program, który jest na tyle duży, że składa się z więcej niż kilkudziesięciu plików.
  - C. Kiedy piszesz bibliotekę, która będzie udostępniana innym programistom.
  - D. Prawidłowe są odpowiedzi B i C.
- 4. Dlaczego nie należy umieszczać deklaracji `using namespace` w pliku nagłówkowym?**
  - A. Ponieważ jest to nieprawidłowe.
  - B. Nie ma powodu, aby tego nie robić. Deklaracja `using` jest ważna wyłącznie w pliku nagłówkowym.
  - C. Ponieważ wymusza to stosowanie deklaracji `using` na każdym, kto dołącza plik nagłówkowy, nawet wtedy, gdy jej zamieszczenie spowoduje powstanie konfliktów nazw.
  - D. Może to wywołać konflikt, jeśli wiele plików nagłówkowych zawiera deklaracje `using`.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- 1. Dodaj do przestrzeni nazw implementację wektora, którą utworzyłeś w ramach zadań praktycznych z rozdziału 25.**

# 28

## ■ ■ ■ ROZDZIAŁ 28

## Plikowe operacje wejścia-wyjścia

---

Dzięki plikom praca z komputerem zyskuje nową jakość. Bez plików wszystko, co robią komputery, istniałoby tylko do chwili wyłączenia maszyny lub zakończenia działania aplikacji. Język C++ udostępnia, rzecz jasna, możliwość odczytywania i zapisywania plików. Praca na plikach nazywana jest plikowymi operacjami **wejścia-wyjścia**, w skrócie we/wy (ang. I/O od *input/output*).

### Podstawy plikowych operacji wejścia-wyjścia

Odczytywanie i zapisywanie plików bardzo przypomina korzystanie z obiektów cout i cin. W przeciwieństwie jednak do cin i cout, które są zmiennymi globalnymi, w celu odczytywania i zapisywania plików musisz zdefiniować własne obiekty<sup>1</sup>, co znaczy, że powinieneś znać ich faktyczne typy.

Te dwa typy danych to ifstream i ofstream. Nazwy te są skrótami od angielskich określeń *input file stream* oraz *output file stream*, czyli, odpowiednio: wejściowy strumień plikowy i wyjściowy strumień plikowy. Strumień to po prostu zestaw danych, które można odczytywać i zapisywać. Zadaniem tych typów jest pobranie pliku i przekształcenie go w długi strumień danych, do których można uzyskać dostęp przypominający interakcję z użytkownikiem. Oba te typy wymagają pliku nagłówkowego fstream (nazwa fstream pochodzi od angielskich słów file stream, czyli strumień plikowy).

### Czytanie z plików

Najpierw zajmijmy się odczytywaniem informacji z plików. Aby odczytać plik, skorzystamy z typu ifstream. Instancję klasy ifstream można utworzyć z wykorzystaniem nazwy pliku, z którego chcemy czytać:

#### Przykładowy kod 62.: ifstream.cpp

```
#include <fstream>

using namespace std;
int main ()
{
    ifstream odczyt_pliku( "mojplik.txt" );
```

---

<sup>1</sup> Ze względu na wygodę czasami będę nazywał je funkcjami, chociaż tak naprawdę są to obiekty, dla których następuje wywoływanie metod.

Ten niewielki program podejmie próbę otwarcia pliku *mojplik.txt*. Plik ten będzie szukany w katalogu, w którym uruchomiony jest program (katalog taki nazywany jest **katalogiem roboczym** programu). Jeśli chcesz, możesz podać także pełną ścieżkę dostępu, na przykład *c:\mojplik.txt*.

Zauważ, iż napisałem, że program podejmie próbę otwarcia pliku. Plik ten może przecież nie istnieć. Aby przekonać się, czy plik został otworzony, możesz sprawdzić wynik tworzenia obiektu *ifstream*, wywołując metodę *is\_open*, która informuje, czy plik został pomyślnie otwarty<sup>2</sup>:

### Przykładowy kod 63.: *ifstream\_sprawdzanie\_bledu.cpp*

```
#include <iostream>
#include <fstream>

using namespace std;
int main ()
{
    ifstream odczyt_pliku( "mojplik.txt" );
    if ( ! odczyt_pliku.is_open() )
    {
        cout << "Nie mogę otworzyć pliku!" << '\n';
    }
}
```

Pracując z plikami, nie masz wyboru — musisz napisać kod obsługujący możliwe do wystąpienia błędy. Pliki mogą nie istnieć, mogą być uszkodzone lub zajęte przez inny proces albo system. W każdym z tych przypadków mogą nie powieść się określone operacje plikowe. Kiedy pracujesz z plikami, powinieneś być przygotowany na niepowodzenia — awarie dysku, uszkodzone pliki, utraty zasilania albo złe sektory na dysku. Wszystko to może doprowadzić do niepowodzenia operacji plikowych.

Kiedy plik zostanie już otwarty, można korzystać z obiektu *ifstream* tak samo jak z *cin*. Następujący kod odczytuje liczbę z pliku tekstowego:

### Przykładowy kod 64.: *odczyt\_pliku.cpp*

```
#include <iostream>
#include <fstream>

using namespace std;
int main ()
{
    ifstream odczyt_pliku( "mojplik.txt" );
    if ( ! odczyt_pliku.is_open() )
    {
        cout << "Nie mogę otworzyć pliku!" << '\n';
    }
    int liczba;
    odczyt_pliku >> liczba;
}
```

---

<sup>2</sup> Informacje na temat tych standardowych funkcji można znaleźć na przykład na stronie <http://en.cppreference.com/w/cpp> albo <http://www.cplusplus.com/reference/>.

Program ten będzie wczytywać z pliku cyfry do chwili, w której natrafi na spację bądź inny separator, tak samo jakby wczytywał znaki wprowadzane przez użytkownika. Jeśli na przykład plik *mojplik.txt* będzie zawierać następującą treść:

```
12 a b c
```

Po uruchomieniu programu zmienna `liczba` przyjmie wartość 12.

Ponieważ pracujemy z plikami, powinniśmy wiedzieć, czy nie wystąpił jakiś błąd. W języku C++ można sprawdzić, czy wartość została pomyślnie wczytana, sprawdzając wynik działania funkcji, która realizuje operację czytania. Możemy to wykonać następująco:

#### **Przykładowy kod 65.: sprawdzanie\_bledu\_odczytu.cpp**

```
#include <fstream>
#include <iostream>

using namespace std;
int main ()
{
    ifstream odczyt_pliku( "mojplik.txt" );
    if ( ! odczyt_pliku.is_open() )
    {
        cout << "Nie mogę otworzyć pliku!" << '\n';
    }
    int liczba;
    // W tym miejscu sprawdzamy, czy udało się wczytać liczbę
    if ( odczyt_pliku >> liczba )
    {
        cout << "Wczytana wartość to: " << liczba;
    }
}
```

Sprawdzając wynik wywołania funkcji `odczyt_pliku >> liczba`, dowiemy się o problemach związanych z odczytem dysku oraz formatem wczytywanych informacji. Czy pamiętasz, jak na początku tej książki pisałem o możliwości podania przez użytkownika litery, podczas gdy program oczekuje liczby? Takie rozwiązanie ustrzeże Cię przed tego typu problemem. Należy po prostu sprawdzić wynik zwracany przez procedurę wejścia. Jeśli wynikiem jej działania jest prawda, wszystko przebiegło prawidłowo i możemy zaufać wczytanym informacjom; jeżeli wynikiem jest fałsz, wówczas coś poszło nie tak i powinniśmy założyć, że wystąpił błąd.

## Formaty plików

Kiedy prosisz użytkownika o wprowadzenie danych, możesz go poinformować, czego oczekujesz, a gdy poda on niepoprawne informacje, możesz poprosić o ich poprawienie. W przypadku odczytywania danych z pliku nie masz takiego komfortu. Plik został już zapisany, prawdopodobnie nawet jeszcze przed powstaniem Twojego programu. W celu poprawnego wczytania danych powinieneś znać **format pliku**. Format pliku definiuje jego układ i wcale nie musi być skomplikowany. Założmy na przykład, że masz listę wyników osiągniętych w grze, którą chciałbyś zapisywać między kolejnymi uruchomieniami programu. Prosty format pliku mógłby sprowadzać się do dziesięciu wierszy, z których każdy zawiera jedną liczbę.

Przykładowa lista z wynikami może wyglądać następująco:

**Przykładowy plik 1.: wyniki.txt**

```
1000  
987  
864  
766  
744  
500  
453  
321  
201  
98  
5
```

Moglibyś napisać następujący program wczytujący listę z wynikami:

**Przykładowy kod 66.: wyniki.cpp**

```
#include <fstream>  
#include <iostream>  
#include <vector>  
  
using namespace std;  
int main ()  
{  
    ifstream odczyt_pliku ( "wyniki.txt" );  
    if ( ! odczyt_pliku.is_open() )  
    {  
        cout << "Nie mogę otworzyć pliku!" << '\n';  
    }  
    vector<int> wyniki;  
    for ( int i = 0; i < 10; i++ )  
    {  
        int wynik;  
        odczyt_pliku >> wynik;  
        wyniki.push_back( wynik );  
    }  
}
```

Powyższy kod jest dość prosty — otwiera on plik i odczytuje po kolej poszczególne wyniki. Tak naprawdę nie zakłada on nawet, że będą one rozdzielone znakami nowego wiersza; równie dobrze mogą to być spacje. Takie działanie programu nie wynika jednak z formatu pliku, lecz z jego implementacji. Inne programy działające z tym formatem mogą nie być aż tak tolerancyjne wobec danych, których oczekują. Dobra zasada dotycząca pracy z formatami plików jest nazywana prawem Postela. Oto jego brzmienie: „Bądź tolerancyjny względem tego, co przyjmujesz, i konserwatywny względem tego, co wysydasz”. Innymi słowy, kod tworzący plik powinien bardzo starannie przestrzegać specyfikacji, ale kod je wczytujący powinien być odporny na drobne błędy popełniane przez nieco gorzej napisane programy. W naszym przykładowym programie tolerujemy w roli separatorów zarówno spacje, jak i znaki nowego wiersza.

## Koniec pliku

Wcześniejszys kod został napisany z myślą o ściśle zdefiniowanym formacie pliku i nawet nie próbuje obsługiwać jakichkolwiek błędów. Jeśli na przykład w pliku będzie mniej niż dziesięć wyników, kod nie przestanie wczytywać pliku, nawet jeśli dotrze do jego końca. Możemy mieć

mniej wyników niż dziesięć, jeśli na przykład gra została uruchomiona dopiero dwukrotnie. Często w celu odniesienia się do sytuacji, w której osiągnięty został koniec pliku, używany jest skrót EOF (ang. *End Of File*).

Możemy sprawić, że nasz kod będzie bardziej niezawodny (tolerancyjny względem akceptowalnych danych), obsługując takie przypadki jak ten, gdy w pliku znajduje się mniej pozycji. W tej sytuacji ponownie można dać sobie radę, sprawdzając wynik działania metody użytej do odczytania pliku.

### Przykładowy kod 67.: wyniki\_eof.cpp

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;
int main ()
{
    ifstream odczyt_pliku ( "wyniki.txt" );
    if ( ! odczyt_pliku.is_open() )
    {
        cout << "Nie mogę otworzyć pliku!" << '\n';
    }
    vector<int> wyniki;
    for ( int i = 0; i < 10; i++ )
    {
        int wynik;
        if ( !( odczyt_pliku >> wynik ) )
        {
            break;
        }
        wyniki.push_back( wynik );
    }
}
```

Jeżeli kod natrafi na plik z liczbą wyników mniejszą niż dziesięć, zakończy ich wczytywanie, gdy tylko dotrze do końca pliku. Dzięki zastosowaniu wektora zamiast tablicy o stałym rozmiarze, takie krótkie pliki można obsłużyć z łatwością. Wektor będzie przechowywać dokładnie tyle wyników, ile wczytano, i nic ponad to. Gdybyśmy chcieli osiągnąć taki sam rezultat przy użyciu tablicy, musielibyśmy na bieżąco śledzić, ile wyników jest już w niej zapisanych. Nie moglibyśmy założyć, że jest już ona całkowicie wypełniona.

Czasami podczas pracy z plikiem chciałbyś wczytywać z niego wszystkie dane, aż dotrzesz do jego końca. W takich przypadkach potrzebny będzie sposób na odróżnienie niepowodzenia spowodowanego osiągnięciem końca pliku od niepowodzenia, którego przyczyną jest błąd w pliku. Metoda eof podpowie Ci, czy osiągnięty został koniec pliku. Możesz napisać pętlę, która wczytuje wszystkie dane, sprawdzając za każdym razem wynik odczytu, aż do wystąpienia błędu. W takiej sytuacji należy sprawdzić, czy metoda eof zwróciła prawdę. Jeśli tak, znajdziemy się na końcu pliku; jeśli nie, wystąpił inny błąd. Błędy te można zweryfikować, wywołując metodę fail, która zwraca prawdę, jeśli błędne są dane wejściowe, oraz fałsz, jeżeli wystąpił problem z odczytem z urządzenia. Po osiągnięciu końca pliku należy wywołać metodę clear, aby możliwe było przeprowadzanie innych operacji plikowych. Już wkrótce, w kolejnym podrоздziale, na przykładzie programu dopisującego nowe wyniki do tabeli pokażę, jak korzystać z tych metod.

Istnieje jeszcze jedna ważna różnica między odczytywaniem danych z pliku a interakcją z użytkownikiem. Co by się stało, gdybyśmy zmienili listę wyników, dołączając do liczby zdobytych punktów nazwę gracza? Musielibyśmy wczytywać zarówno nazwy zawodników, jak i osiągnięte przez nich wyniki. W celu obsłużenia nowej sytuacji konieczna byłaby modyfikacja naszego kodu. Starsze wersje programu nie byłyby w stanie odczytywać plików w nowym formacie, co przywróciłoby Cię o spory ból głowy, gdybyś miał wielu użytkowników i zechciał zaktualizować u nich format pliku! Istnieją techniki, z których możesz skorzystać w celu umożliwienia **przyszłych aktualizacji** formatu, polegające na dodawaniu opcjonalnych pól lub takim pisaniu programów, aby starsze wersje pomijały nowe elementy formatu. Rozwiązań te wykraczają jednak poza tematykę tej książki. Na razie wystarczy, jeśli będziesz wiedzieć, że zdefiniowanie formatu pliku stanowi pod pewnymi względami większe wyzwanie niż opracowanie podstawowego interfejsu programu.

## Zapisywanie plików

Typ, który jest nam potrzebny podczas zapisywania plików, to `ofstream`. Bardzo przypomina on typ `ifstream`, z tym że zamiast korzystać z niego w sposób podobny do `cin`, należy go używać tak jak `cout`.

Spójrzmy na prosty program, który w pliku `wyniki.txt` zapisuje wartości od 10 do 1 (już nie-długo sprawimy, że kod ten wygeneruje coś, co bardziej będzie przypominać listę najlepszych wyników).

### Przykładowy kod 68.: `ofstream.cpp`

```
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;
int main ()
{
    ofstream zapis_pliku ( "wyniki.txt" );
    if ( ! zapis_pliku.is_open() )
    {
        cout << "Nie mogę otworzyć pliku!" << '\n';
        return 0;
    }

    // Ponieważ nie mamy rzeczywistych wyników, zapiszemy liczby od 10 do 1

    for ( int i = 0; i < 10; i++ )
    {
        zapis_pliku << 10 - i << '\n';
    }
}
```

W kodzie tym nie musimy przejmować się osiągnięciem końca pliku. Kiedy informacje są zapisywane w pliku i zostanie osiągnięty jego koniec, obiekt `ofstream` rozszerzy ten plik. Taka operacja nazywana jest **dołączaniem do pliku**.

## Tworzenie nowych plików

Jeśli podczas korzystania z obiektu `ofstream` plik nie istnieje, zostanie on utworzony. Jeżeli natomiast plik już istnieje, zostanie on nadpisany. Prawdopodobnie nie będziesz mieć nic przeciwko nadpisywaniu pliku, jeśli zachowujesz na dysku listę wyników osiągniętych w grze, ponieważ w takim przypadku zapisujesz po prostu wszystkie dane. Jeśli natomiast na bieżąco zapisujesz dziennik systemowy, na przykład z informacjami o dacie i czasie uruchomienia Twojego programu, z pewnością nie chciałbyś go za każdym razem nadpisywać.

Konstruktor klasy `ofstream` przyjmuje drugi argument, który określa sposób obsługi pliku:

|                          |                                                                                                       |
|--------------------------|-------------------------------------------------------------------------------------------------------|
| <code>ios::app</code>    | Dołączanie do pliku; po każdym zapisie pozycja w pliku zostanie przeniesiona na jego koniec.          |
| <code>ios::ate</code>    | Bieżącą pozycją w pliku będzie jego koniec.                                                           |
| <code>ios::trunc</code>  | Usunięcie całej zawartości pliku.                                                                     |
| <code>ios::out</code>    | Umożliwia zapisywanie do pliku.                                                                       |
| <code>ios::binary</code> | Umożliwia przeprowadzanie operacji binarnych na strumieniu (dostępny także podczas czytania z pliku). |

Jeśli chciałbyś wybrać jednocześnie kilka opcji, na przykład otwarcie pliku i wykonywanie binarnych operacji we/wy (o czym już niedługo napiszę), możesz rozdzielić wybrane przez siebie opcje symbolem potoku (`|`)<sup>3</sup>:

```
ofstream jakis_plik ( "test.txt", ios::app | ios::binary );
```

Powyzszy kod spowoduje otwarcie pliku bez niszczenia jego obecnej zawartości, umożliwiając dopisywanie na jego końcu danych binarnych.

## Pozycja pliku

Kiedy program zapisuje lub odczytuje plik, kod dokonujący operacji we/wy musi wiedzieć, w którym miejscu pliku dana operacja nastąpi. Możesz wyobrazić sobie na ekranie komputera kursor wskazujący, gdzie pojawi się następny znak, który wpiszesz.

W przypadku podstawowych operacji nie musisz przejmować się pozycją. Możesz po prostu pozwolić, aby Twój kod odczytywał z pliku kolejne dane, jakie tam się znajdują, lub zapisywał w nim jakiekolwiek następne informacje, które mają być zachowane. Istnieje również możliwość zmiany pozycji w pliku bez dokonywania odczytu. Taka operacja często jest potrzebna podczas pracy z plikami przechowującymi złożone dane, takimi jak ZIP albo PDF, lub wówczas, gdy masz do czynienia z plikami obszernymi, gdy odczytanie wszystkich bajtów byłoby czasochłonne lub niemożliwe (na przykład jeśli implementujesz bazę danych).

Tak naprawdę w pliku można wyróżnić dwie różne pozycje. Jedna z nich to pozycja, spod której nastąpi odczyt danych, a druga to pozycja, pod którą program zapisze dane. Bieżącą pozycję można odczytać za pomocą metod `tellg` oraz `tellp`. Udostępniają one, odpowiednio: aktualną pozycję do odczytu (g jak `get`, czyli `pobierz`) oraz zapisu (p jak `put`, czyli `zapisz`) danych.

<sup>3</sup> Znak potoku jest operatorem bitowym (bitowym *lub*). Każda z opcji argumentu `ios::` zmienia wartość jednego z bitów na prawdę. Za pomocą operatora bitowego *lub* można łączyć poszczególne opcje. Więcej informacji na temat operatorów bitowych można znaleźć na przykład na stronie <http://guidecpp.cal.pl/cplus/operators-bits>.

Istnieje również możliwość zmiany aktualnego miejsca w pliku, tj. przeniesienia bieżącej pozycji za pomocą instrukcji `seekp` i `seekg`. Ich nazwy pochodzą od angielskiego słowa *seek*, które w tym przypadku oznacza przemieszczanie się po pliku, polegające na przenoszeniu pozycji do odczytu lub zapisu w nowe miejsce. Obie wymienione metody pobierają dwa parametry — odległość do przejścia oraz punkt wyjścia do przeprowadzenia operacji. Odległość do przejścia jest wyrażana w bajtach, natomiast punktem wyjścia jest bieżąca pozycja albo początek lub koniec pliku. Po dokonaniu przejścia możliwe będzie odczytywanie (lub zapisywanie) danych, począwszy od nowej pozycji w pliku. Zmiana jednej z pozycji nie ma wpływu na pozycję drugą.

Oto trzy opcje dotyczące zmiany pozycji w pliku:

|                            |                                                        |
|----------------------------|--------------------------------------------------------|
| <code>ios_base::beg</code> | Czytanie i zapisywanie, począwszy od początku pliku.   |
| <code>ios_base::cur</code> | Czytanie i zapisywanie, począwszy od bieżącej pozycji. |
| <code>ios_base::end</code> | Czytanie i zapisywanie, począwszy od końca pliku.      |

Aby na przykład przejść na początek pliku przed zapisaniem w tym miejscu danych, można napisać:

```
zapis_pliku.seekp( 0, ios_base::beg );
```

Wartość zwracana przez metody `tellp` i `tellg` jest specjalnym typem zmiennej o nazwie `streampos`, zdefiniowanym w bibliotece standardowej. Podlega on konwersji z oraz na liczby całkowite, chociaż korzystając ze `streampos` można ściśle wyrazić konkretny typ. Liczby całkowite mogą być stosowane praktycznie wszędzie, natomiast cel stosowania typu `streampos` jest ściśle określony. Służy on do przechowywania pozycji w pliku i przechodzenia do tych pozycji. Korzystanie w kodzie z właściwego typu zmiennej pozwala lepiej zdefiniować, do czego służy ta zmienna.

```
streampos pozycja = odczyt_pliku.tellg();
```

W niektórych przypadkach nie będzie potrzeby zmieniania pozycji w pliku i zdecydowanie wystarczy odczytanie lub zapisanie pliku od samego początku aż do końca. Wiele formatów plikowych jest jednak zoptymalizowanych pod względem dodawania do pliku nowych danych. Dodawanie danych na końcu pliku przebiega o wiele szybciej niż wstawianie danych w środku pliku. Problem ze wstawianiem danych w środku polega na tym, że wszystko, co znajduje się za punktem wstawienia, musi zostać przeniesione, dokładnie tak samo jak podczas wstawiania nowego elementu w środku tablicy<sup>4</sup>.

Zmodyfikujmy nasz wcześniejszy program służący do rejestrowania wyników w taki sposób, aby można było dodawać do pliku nowe najlepsze wyniki uzyskane w grze. Aby zadanie było ciekawsze, nowe wartości powinny być wstawiane na właściwych pozycjach.

W tym celu będziemy musieli zarówno odczytywać, jak i zapisywać plik, w związku z czym skorzystamy z klasy `fstream`, która udostępnia obie te możliwości. Możesz ją traktować jak połączone ze sobą klasy `ofstream` oraz `ifstream`. Najpierw pobierzemy od użytkownika nowy wynik. Następnie będziemy wiersz po wierszu wczytywać plik, aż natrafimy na punktację, która jest niższa od wartości podanej przez użytkownika. W tym właśnie miejscu będziemy musieli wstawić nowy wynik. Zapamiętamy tę pozycję, a wszystkie pozostałe wiersze wczytamy do

<sup>4</sup> Istnieje tu pewien szczególny przypadek — jeśli nadpisujesz istniejące dane nowymi danymi o dokładnie takiej samej długości, nie musisz niczego przenosić, a cała operacja jest tak szybka jak dodawanie danych na końcu pliku.

wektora, po czym powrócimy do zapamiętanego miejsca. Zapiszemy tam nową punktację, a następnie przywrócimy pozostałe wyniki z wektora, zastępując wiersze, które istnieją już w pliku.

Ponieważ użyjemy klasy `fstream`, będziemy mogli korzystać z wszelkich zalet związanych z możliwością zarówno odczytywania, jak i zapisywania plików, z tym że najpierw musimy jawnie poinformować konstruktor, że plik ma zostać otwarty na potrzeby obu tych operacji. W tym celu zastosujemy flagi `ios::in | ios::out`. Przed uruchomieniem programu będziesz musiał utworzyć plik z punktacją, gdyż nie wygeneruje on za Ciebie pustego pliku.

### **Przykładowy kod 69.: pozycja\_w\_pliku.cpp**

```
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;

int main ()
{
    fstream plik ( "wyniki.txt", ios::in | ios::out );
    if ( ! plik.is_open() )
    {
        cout << "Nie mogę otworzyć pliku!" << '\n';
        return 0;
    }
    int nowy_wynik;
    cout << "Podaj nowy wynik: ";
    cin >> nowy_wynik;

    // Poniższa pętla przeszukuje plik, aż natrafi na wartość,
    // która jest mniejsza od bieżącego wyniku; to właśnie w tym
    // miejscu, tuż przed znalezioną wartością, będziemy chcieli
    // wstawić nowy wynik. Aby zagwarantować, że nasza pozycja
    // jest dobra, w zmiennej poprz_poz_wyniku przechowujemy
    // pozycję poprzedzającą bieżącą punktację
    streampos poprz_poz_wyniku = plik.tellg();
    int biez_wynik;
    while ( plik >> biez_wynik )
    {
        if ( biez_wynik < nowy_wynik )
        {
            break;
        }
        poprz_poz_wyniku = plik.tellg();
    }

    // Jeśli fail zwróci prawdę, a my nie znajdujemy się na końcu pliku,
    // to znaczy, że dane były błędne
    if ( plik.fail() && ! plik.eof() )
    {
        cout << "Niepoprawny odczyt wyniku--wychodzę z programu";
        return 0;
    }
    // Bez wywołania clear nie moglibyśmy zapisywać do pliku,
    // gdybyśmy natrafili na eof
    plik.clear();
```

```
// Wracamy do miejsca znajdującego się tuż przed odczytem ostatniego
// wyniku, abyśmy mogli wczytać wszystkie wyniki, które są mniejsze
// od wyniku podanego przez użytkownika, i przenieść je w pliku
// o jedno miejsce dalej
plik.seekg( poprz_poz_wyniku );

// Teraz wczytamy wszystkie wyniki, począwszy od wyniku,
// który wczytaliśmy poprzednio
vector<int> punktacja;
while ( plik >> biez_wynik )
{
    punktacja.push_back( biez_wynik );
}
// W tej pętli spodziewamy się końca pliku, ponieważ chcemy
// odczytać wszystkie zapisane wyniki
if ( ! plik.eof() )
{
    cout << "Niepoprawny odczyt wyniku--wychodzę z programu";
    return 0;
}
// Ponieważ dotarliśmy do końca pliku, znowu musimy wywołać clear,
// abyśmy mogli w tym pliku zapisywać
plik.clear();

// Wracamy na pozycję, w której chcemy wstawić nasz wynik
plik.seekp( poprz_poz_wyniku );
// Jeśli nie zapisujemy na początku pliku, będziemy musieli
// wstawić nowy wiersz. Robimy to dlatego, że wczytywanie liczby
// zatrzymuje się przy pierwszym białym znaku, tak więc pozycja,
// na której się znajdująmy przed przystąpieniem do zapisywania,
// to koniec liczby, a nie początek następnego wiersza
if ( poprz_poz_wyniku != std::streampos(0) )
{
    plik << endl;
}
// Zapisz nasz nowy wynik
plik << nowy_wynik << endl;
// Przechodzimy w pętli przez resztę wektora punktacja, zapisując
// wszystkie pozostałe wyniki
for ( vector<int>::iterator itr = punktacja.begin(); itr != punktacja.end(); ++itr )
{
    plik << *itr << endl;
}
```

## Pobieranie argumentów z wiersza poleceń

Pisząc programy operujące na plikach, często chciałbyś, aby użytkownicy jako argument mogli podawać w wierszu poleceń nazwę pliku. Taki sposób obsługi programu często jest prostszy i ułatwia pisanie skryptów, które go wywołują. Zróbjmy sobie krótką przerwę od rozważań na tematy związane z czytaniem i zapisywaniem plików, abyśmy mogli zacząć wyposażać nasze programy w taką właśnie możliwość.

Argumenty w wierszu poleceń są podawane po nazwie programu i są do niego przekazywane za pośrednictwem systemu operacyjnego:

```
C:\moj_program\moj_program.exe arg1 arg2
```

Argumenty z wiersza poleceń są przekazywane bezpośrednio do funkcji `main`. Aby z nich skorzystać, należy podać pełną deklarację tej funkcji (wszystkie funkcje `main`, które do tej pory widziałeś, miały pustą listę argumentów). W rzeczywistości funkcja ta przyjmuje dwa argumenty: pierwszym z nich jest liczba argumentów w wierszu poleceń, a drugim pełna lista tych argumentów.

Pełna deklaracja funkcji `main` wygląda następująco:

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

Zmienna całkowita `argc` jest licznikiem argumentów. Przechowuje ona liczbę argumentów przekazanych do programu z wiersza poleceń, łącznie z nazwą programu. Być może zastanawiasz się, dlaczego nie musisz uwzględniać tych argumentów we wszystkich programach. Odpowiedź jest taka, że jeśli ich nie zamieścisz, kompilator po prostu zignoruje fakt ich braku.

Tablica wskaźników typu `char` jest listą wszystkich argumentów. Element `argv[ 0 ]` zawiera nazwę programu lub łańcuch pusty, jeśli nazwa nie jest dostępna. Kolejne elementy tablicy, o wartościach mniejszych od `argc`, zawierają argumenty z wiersza poleceń. Z każdego z nich można korzystać tak samo jak ze zmiennej łańcuchowej. Element `argv[ argc ]` jest wskaźnikiem `NULL`.

Spójrzmy na przykładowy program, który pobiera argument z wiersza poleceń. W tym przypadku będzie to nazwa pliku, którego zawartość zostanie wyświetlona na ekranie.

### **Przykładowy kod 70.: wyswietl\_plik.cpp**

```
#include <fstream>
#include <iostream>

using namespace std;

int main (int argc, char *argv[])
{
    // Aby program wykonał się poprawnie, zmienna argc powinna
    // być równa 2 (tj. dwa argumenty — nazwa programu i nazwa pliku)

    if ( argc != 2 )
    {
        // W celu wyświetlenia instrukcji użycia można skorzystać
        // z wartości argv[ 0 ] zawierającej nazwę programu
        cout << "Użycie: " << argv[ 0 ] << " <nazwa pliku>" << endl;
    }
    else
    {
        // Przyjmujemy, że argv[ 1 ] zawiera nazwę pliku do otwarcia
        ifstream plik( argv[ 1 ] );
        // Zawsze należy sprawdzić, czy otworzenie pliku powiodło się
        if ( ! plik.is_open() )
        {
            cout << "Nie mogłem otworzyć pliku " << argv[ 1 ] << endl;
            return 1;
        }
    }
}
```

```
char x;
// plik.get( x ) odczytuje do x następny znak z pliku
// i zwraca falsz, jeśli dotarliśmy do końca pliku
// lub jeśli wystąpił błąd
while ( plik.get( x ) )
{
    cout << x;
}
} // Plik jest zamknięty niejawnie przez swój destruktor
```

Powyższy program korzysta z pełnej deklaracji funkcji `main` w celu pobrania parametrów z wiersza poleceń. Najpierw sprawdza, czy użytkownik podał nazwę pliku. Następnie sprawdza, czy nazwa pliku jest prawidłowa, próbując go otworzyć. Jeśli plik istnieje, zostaje otwarty; jeżeli nie, program informuje użytkownika o błędzie. W przypadku gdy plik zostanie pomyślnie otworzony, program wyświetla na ekranie wszystkie znaki, które się w nim znajdują.

## Obsługa argumentów liczbowych

Jeśli chcesz pobrać z wiersza poleceń parametr i użyć go jako liczby, powinieneś wczytać go jako łańcuch tekstowy, po czym wywołać funkcję `atoi` (ang. *ASCII to Integer*, czyli ASCII na liczby całkowite). Funkcja ta pobiera ciąg `char*` i zwraca liczbę, która jest zapisana w tym ciągu znaków. Aby można było z niej korzystać, należy dołączyć plik nagłówkowy `cstdint.h`. Poniższy przykładowy program wczytuje argument z wiersza poleceń, konwertuje go na liczbę i wyświetla jej kwadrat:

### Przykładowy kod 71.: `atoi.cpp`

```
#include <cstdlib>
#include <iostream>

using namespace std;
int main (int argc, char *argv[])
{
    if ( argc != 2 )
    {
        // W celu wyświetlania instrukcji użycia można skorzystać
        // z wartości argv[ 0 ] zawierającej nazwę programu
        cout << "Użycie: " << argv[ 0 ] << " <liczba>" << endl;
    }
    else
    {
        int wart = atoi( argv[ 1 ] );
        cout << wart * wart;
    }
    return 0;
}
```

## Pliki binarne

Na razie dowiedziałeś się, jak pracować z plikami zawierającymi dane tekstowe. Zajmijmy się teraz **plikami binarnymi**, które często są używane ze względu na większą wydajność. Praca z plikami binarnymi wymaga innych technik programistycznych niż w przypadku plików tekstowych. Nie chciałbym zamacić Ci w głowie, ale powinieneś wiedzieć, że wszystkie pliki

w Twoim systemie są przechowywane w postaci binarnej. W wielu jednak przypadkach są one zapisywane w sposób umożliwiający ich odczytanie przez użytkowników. Na przykład pliki źródłowe C++ w całości składają się ze znaków, które można wczytać za pomocą prostego edytora tekstów. Taki rodzaj pliku, w którym każdy bajt jest nadającym się do przeczytania znakiem, nazywany jest **plikiem tekstowym**.

Nie wszystkie jednak pliki zawierają wyłącznie tekst. Niektóre z nich składają się z bajtów, które nie są drukowalnymi znakami. W takich plikach znajdują się nieprzetworzone dane binarne pochodzące z jednej lub wielu struktur danych zapisanych bezpośrednio na dysku.

Załóżmy na przykład, że masz do czynienia ze strukturą reprezentującą gracza:

```
struct gracz
{
    int wiek;
    int najlepszy_wynik;
    string nazwa;
};
```

Gdybyś chciał zapisać powyższą strukturę w pliku, miałbyś do wyboru dwie możliwości. Po pierwsze mógłbyś zapisać pola `wiek` oraz `najlepszy_wynik` w postaci tekstowej, dzięki czemu plik zawierający te dane można by otwierać w Notatniku. Taki plik mógłby wyglądać następująco:

```
19
120000
Tomek
```

Przy takiej reprezentacji danych potrzeba sześciu znaków do zapisania najlepszego wyniku. Jak już wiemy, każdy znak przechowywany jest w jednym bajcie, co oznacza, że zapisany w pliku wynik zajmie sześć bajtów. Wartość ta jednak jest liczbą całkowitą, a liczby takie zajmują zwykle tylko sześć bajtów (w systemach 32-bitowych). Czy zatem do zapisania wyniku nie powinny wystarczyć cztery bajty? Oczywiście, że tak! Jeśli w takim razie mamy zachować liczbę w czterech bajtach, nie możemy już w celu sprawdzenia najlepszego wyniku korzystać z edytora tekstopiowego. Dlaczego? Ponieważ przy zapisywaniu liczby 120000 w postaci łańcucha tekstopiowego jest ona kodowana w taki sposób, aby każdy znak był zapisany w postaci cyfry zajmującej jeden bajt. Jeżeli natomiast liczba jest bezpośrednio zachowywana w pliku, bajty w ogóle nie są kodowane w postaci znaków, w związku z czym będziemy mieć do czynienia z liczbą całkowitą składającą się z czterech bajtów. Kiedy taki plik zostanie otworzony w edytorze tekstopiowym, cztery znajdujące się w nim bajty zostaną potraktowane jak cztery znaki, tylko że nie będą mieć one żadnego związku z liczbą, którą zapisaliśmy! To, co zobaczymy na ekranie, będzie pozbawione sensu, ponieważ zawartość pliku została zakodowana w inny sposób.

Pliki w formacie binarnym zajmują mniej miejsca. Jak to zobaczyliśmy w naszym przykładzie, liczba 120000 przechowywana w formie znaków zabiera o połowę więcej miejsca, niż jeśli jest zapisana w postaci binarnej. Można wyobrazić sobie, że ma to ogromne znaczenie podczas przesyłania danych siecią albo gdy dysk twardy nie jest zbyt szybki ani pojemny. Z drugiej jednak strony pliki binarne są trudniejsze do przeglądania i zrozumienia — pliku binarnego nie da się po prostu otworzyć w edytorze tekstopiowym i sprawdzić, co też się w nim znajduje. Osoby pracujące nad formatami plików muszą dokonywać wyborów między formatami wydajnymi a formatami, które mogą być zrozumiane i modyfikowane przez człowieka. Bazujące na tekście języki znaczników, takie jak XML, są często stosowane w celu tworzenia formatów zrozumiałych dla ludzi, chociaż wymagających więcej miejsca.

Kiedy ważna jest oszczędność miejsca, a procesory są wystarczająco szybkie, możliwe jest stosowanie technik kompresji, takich jak ZIP, w celu ograniczenia zajmowanej objętości przy jednoczesnym zachowaniu tekstowych właściwości pliku po jego rozpakowaniu. Ponieważ wypakowanie pliku .zip jest bardzo łatwe, praca z nimi nie przysparza problemów, a są one zdecydowanie mniejsze niż niepoddane kompresji pliki tekstowe.

Pliki binarne są mimo wszystko powszechnie używane. Wiele istniejących formatów plików bazuje na binarności, a liczne pliki wręcz muszą być binarne — wszystkie formaty przechowujące obrazy, filmy albo dźwięk nie mają sensownego i zachowującego wierność tekstowego odpowiednika. Kiedy niezbędna jest maksymalna wydajność lub oszczędność miejsca, pliki binarne biją wszystkie inne na głowę. Na przykład Microsoft wprowadził w pakiecie Office 2007 nowe formaty plików, bazujące na języku XML umieszczonym wewnętrz plików ZIP. Dodał także format binarny plików Excela (.xlsm) dla użytkowników, którzy wymagają maksymalnej wydajności. Innymi słowy, pliki binarne pozostaną przy nas, a projektując nowy format pliku, powinieneś dokonać wyboru między łatwiejszą implementacją i reprezentacją plików bazujących na tekście a lepszą wydajnością i mniejszymi rozmiarami plików binarnych.

Możesz więc zapytać, jak właściwie pracować z plikami binarnymi?

## Praca z plikami binarnymi

Etap pierwszy polega na otwarciu pliku w trybie binarnym:

```
fstream plik( "test.bin", ios::binary );
```

Po otwarciu pliku można korzystać ze znanych już nam funkcji wejścia i wyjścia, z tym że będziemy musieli używać funkcji specyficznych dla pracy z danymi binarnymi. Poszczególne bajty będą zapisywane do pliku bezpośrednio z bloku pamięci. Metoda, z której skorzystamy, nazywa się `write` i jako argumenty przyjmuje wskaźnik do bloku pamięci oraz wielkość pamięci do zapisania. Typ wskaźnika to `char*`, ale Twoje dane nie muszą być znakami. Dlaczego w takim razie musimy bazować na łańcuchu tekstowym? W języku C++ praca z poszczególnymi bajtami sprawdza się do użycia zmiennych jednobajtowych, typu `char`, albo wskaźnika do serii bajtów, którym jest `char*`. Jeśli zechcesz zapisać w pliku nieprzetworzony ciąg bajtów, musisz udostępnić wskaźnik `char*`, który umożliwi umieszczenie tych bajtów w pliku. Aby zapisać w pliku liczbę całkowitą, powinniśmy potraktować ją jak serię bajtów, `char*`, i przekazać ten wskaźnik do metody, która przepisze odpowiednie bajty bezpośrednio z pamięci do pliku. Metoda `write` zapisze wszystkie znaki, bajt po bajcie, we właściwej kolejności. Przyjmijmy na przykład, że naszą liczbą jest 255. Jest ona reprezentowana w pamięci za pomocą bajtu `0xFF` (dziesiątknie 255). Liczba całkowita przechowująca bajt `0xFF` będzie wyglądać w pamięci komputera następująco:

0x000000FF

Albo bajt po bajcie:

00 00 00 FF

Aby zapisać liczbę całkowitą w pliku, będzie nam potrzebny sposób bezpośredniego odwołania się do tego ciągu bajtów. To właśnie dlatego skorzystamy ze wskaźnika `char*` — nie ze względu na możliwość reprezentowania w nim znaków ASCII, ale dlatego, że dzięki niemu możemy pracować na bajtach.

Potrzebny nam będzie także sposób na poinformowanie kompilatora, że powinien traktować nasze dane tak, jakby były tablicą znakową.

## Konwersja na typ char\*

Jak w takim razie możemy powiedzieć kompilatorowi, żeby traktował zmienną jak wskaźnik do znaku, a nie jak wskaźnik do jego faktycznego typu? Nakazanie kompilatorowi, aby traktował zmienną, jakby była innego typu, nazywane jest **rzutowaniem typu**. Wydać polecenie rzutowania to tak, jakby powiedzieć: „Naprawdę wiem, co robię. Rzeczywiście chcę, aby ta zmienna została użyta w taki sposób”. Chcemy potraktować zmienną jak ciąg pojedynczych bajtów, a zatem musimy skorzystać z rzutowania w celu zmuszenia kompilatora, aby udostępnił nam jej poszczególne bajty.

Dwa podstawowe rodzaje rzutowania to `static_cast` i `reinterpret_cast`. Pierwszy z nich jest używany w celu dokonywania konwersji między zbliżonymi typami, na przykład gdy kompilator powinien potraktować typ `double` jak `integer`, aby można było obciążić jego część dziesiętną, np. `static_cast<int>( 3.4 )`. Typ, na który jest dokonywane rzutowanie, znajduje się w nawiasach ostrokatnych po nazwie rodzaju rzutowania.

W naszym przypadku chcemy jednak całkowicie zignorować obecny typ i skłonić kompilator, aby zinterpretował serię bajtów jako należącą do zupełnie innego typu. W tym celu będzie nam potrzebne rzutowanie `reinterpret_cast`. Aby na przykład potraktować tablicę liczb całkowitych jako tablicę znakową, możemy napisać:

```
int x[ 10 ];
reinterpret_cast<char*>( x );
```

A tak przy okazji — praca z danymi binarnymi to jeden z niewielu obszarów, w których stosowanie rzutowania `reinterpret_cast` jest dobrym pomysłem. Natrafienie na `reinterpret_cast` powinno obudzić Twoją czujność! W ten sposób można zmuszać kompilator do robienia rzeczy, których normalnie by nie zrobił. W rezultacie kod, w którym zastosowano tego rodzaju rzutowanie, nie będzie sprawdzany tak samo starannie, jak to jest w przypadku reszty kodu. W tym konkretnym przypadku próbujemy sięgnąć do pamięci, która istotnie jest ciągiem bajtów — właśnie takie rozwiązanie jest nam potrzebne. Jeśli jednak nie musisz, nie stosuj rzutowania `reinterpret_cast`.

## Przykład binarnych operacji we/wy

Wreszcie możemy przejść do zademonstrowania binarnych operacji wejścia i wyjścia! Poniższy kod wypełnia tablicę, a następnie zapisuje ją w pliku. W tym celu korzysta z metody `write`, z którą mieliśmy już wcześniej do czynienia. Przyjmuje ona dwa argumenty; pierwszym jest źródło danych typu `char*`, a drugim rozmiar tego źródła. W tym przypadku źródłem jest nasza tablica, a wielkością źródła rozmiar tablicy w bajtach.

```
int liczby[ 10 ];

for ( int i = 0; i < 10; i++ )
{
    liczby[ i ] = i;
}
plik.write( reinterpret_cast<char*>( liczby ), sizeof( liczby ) );
```

Zaczynamy od utworzenia tablicy liczb całkowitych, która po rzutowaniu na `char*` będzie traktowana jak tablica bajtów i zostanie zapisana bezpośrednio na dysku. Kiedy później te bajty wczytamy, zostaną one umieszczone w pamięci dokładnie w takiej samej kolejności, a my będziemy mogli dokonać ich rzutowania na liczby całkowite w celu uzyskania wartości, od których zaczynaliśmy.

Zwróć uwagę, że rozmiar pliku do zapisania jest udostępniany za pomocą operatora `sizeof`. Polecenie to jest bardzo przydatne do odczytywania rozmiaru określonej zmiennej. W tym przypadku zwraca onołączną liczbę bajtów, które tworzą tablicę liczby.

Podczas stosowania polecenia `sizeof` w odniesieniu do wskaźnika powinieneś jednak zachować ostrożność. Jeśli przekażesz do niego wskaźnik, otrzymasz wielkość wskaźnika, a nie pamięci, którą on wskazuje. Powyższy kod działa poprawnie, ponieważ liczby jest tablicą, a nie wskaźnikiem, a `sizeof` zna całkowity rozmiar tablicy. Jeśli masz do czynienia ze zmienną wskaźnikową, np. `int *w_liczba`, jej rozmiar wyniesie zwykle cztery bajty, ponieważ tyle właśnie potrzeba do zapamiętania adresu. Jeśli potrzebujesz znać rozmiar wskazywanego elementu, możesz napisać `sizeof( *w_liczba )`. W tym przypadku wynik działania tego polecenia będzie taki sam jak instrukcji `sizeof( int )`. Jeśli wskaźnik wskazuje tablicę (gdybyś napisał `int *w_liczba = new int[ dlugosc ]`), mógłbyś uzyskać łączny rozmiar tablicy, pisząc `sizeof( *w_liczba ) * dlugosc`.

Z metody `write` można także korzystać w celu zapisania struktury bezpośrednio w pliku. Przyjmijmy na przykład, że masz następującą strukturę:

```
struct KartotekaGracza
{
    int wiek;
    int wynik;
}
```

Możesz wówczas utworzyć instancję struktury `KartotekaGracza` i zapisać ją w pliku:

```
KartotekaGracza kart;
kart.wiek = 10;
kart.wynik = 890;

plik.write(reinterpret_cast<char*>( & kart ), sizeof( kart ) );
```

Zwróci uwagę, że w tym przypadku w celu przekazania wskaźnika do struktury pobieramy adres struktury `kart`.

## Przechowywanie klas w pliku

A gdybyśmy tak zechcieli dodać do naszej struktury typ, który nie jest podstawowy? Na przykład co by się stało, gdybyśmy w strukturze umieścili łańcuch tekstowy?

```
struct KartotekaGracza
{
    int wiek;
    int wynik;
    string nazwa;
}
```

W tym przypadku po prostu dodaliśmy do struktury nazwę gracza w postaci łańcucha tekstu-wego. Co jednak by się stało, gdybyśmy teraz zechcieli zapisać naszą strukturę do pliku i dotarliśmy do tego łańcucha? Informacja znajdująca się w łańcuchu zostałaby zapisana, ale prawdopodobnie nie byłby to łańcuch w swojej postaci.

Typ łańcuchowy jest implementowany jako wskaźnik do łańcucha (z pewnymi dodatkowymi informacjami, takimi jak długość łańcucha). Podczas zapisywania do pliku struktury w postaci danych binarnych metoda `write` zachowa wszystko to, co znajduje się w łańcuchu, tj. wskaźnik i długość. Wskaźnik ten ma jednak sens tylko wtedy, gdy działa program! Rzeczywista wartość

wskaźnika — adres pamięci — przestanie być przydatna po wyjściu z programu, ponieważ pod adresem tym nie będzie się już nic znajdować. Gdy następnym razem nasza struktura zostanie wczytana do pamięci, dostaniemy wskaźnik wskazujący pamięć, która nie jest poprawnie zaalokowana, albo wskazujący dane, które nie mają nic wspólnego z zapisanym w niej łańcuchem.

Zamiast na ślepo zapisywać bezpośrednio do pliku naszą strukturę, potrzebny nam będzie trwały, dobrze zdefiniowany format reprezentujący dane binarne. W naszym formacie będziemy zapisywać znaki łańcucha oraz jego długość (długość łańcucha będzie potrzebna ze względów, które wyjawię już za chwilę). Zobaczmy, jak to wygląda:

```
KartotekaGracza kart;
kart.wiek = 11;
kart.wynik = 200;
kart.nazwa = "Jan";

fstream plik( "kartoteki.bin", ios::trunc | ios::binary | ios::in | ios::out );

plik.write( reinterpret_cast<char*>( & kart.wiek ), sizeof( kart.wiek ) );
plik.write( reinterpret_cast<char*>( & kart.wynik ), sizeof( kart.wynik ) );
int dlug = kart.nazwa.length();
plik.write( reinterpret_cast<char*>( & dlug ), sizeof( dlug ) );
plik.write( kart.nazwa.c_str(), dlug + 1 ); // + 1 na znak końca NULL
```

Przede wszystkim zwróć uwagę na użycie metody `c_str` w celu uzyskania wskaźnika do łańcucha znaków w pamięci, w przeciwieństwie do obiektu typu `string`, który nie ma zagwarantowanego miejsca w pamięci. Jeśli Twoim łańcuchem znaków jest „abc”, wówczas wywołanie metody `c_str` zwróci adres sekwencji tych znaków. Łańcuch tekstowy jest zakończony znakiem o wartości 0, który nazywany jest znakiem końca łańcucha<sup>5</sup>. Łańcuchy tekstowe w takim formacie zwane są łańcuchami w stylu języka C, ponieważ w języku tym był to jedyny szeroko stosowany format łańcuchowy.

Zapisywanie danych łańcuchowych w pliku binarnym jest jak najbardziej poprawne. Nawet gdy zapisujemy w nim znaki, nadal mamy do czynienia z danymi binarnymi — tak się tylko składa, że są one czytelne także dla człowieka.

Również najzupełniej prawidłowe jest, że nie zapisujemy pełnej struktury, z jaką pracujemy. Istotne jest, że mamy możliwość przekształcenia formatu zapisanego na dysku w obiekt w pamięci komputera. Nie musimy bezpośrednio przenosić do pliku bajtów z pamięci. Format plikowy stanowi pewną reprezentację danych, a struktura — inną. Oba formaty przechowują te same dane, a format struktury w pamięci komputera nie musi być taki sam jak format danych zapisanych w pliku.

<sup>5</sup> Czasami znak końca jest zapisywany jako \0. Taki zapis jest jak najbardziej poprawny. Różnica między 0 a \0 polega na tym, że w przypadku zapisu \0 wyjściowym typem zmiennej jest char, natomiast w przypadku zapisu bez ukośnika mamy do czynienia z liczbą całkowitą przekonwertowaną na char. Dla naszych celów oba zapisy będą mieliśmy traktować jako prawidłowe.

## Czytanie z pliku

Aby wczytać dane z pliku binarnego, należy skorzystać z metody `read`. Jej argumenty są niemal takie same jak argumenty metody `write` — miejsce do umieszczenia danych oraz ilość danych do odczytania<sup>6</sup>. W celu odczytania z pliku liczby całkowitej można skorzystać z następującego kodu:

```
int x = 3;  
plik.read( reinterpret_cast<char*>( & x ), sizeof( x ) );
```

Podczas pracy z plikami będzie Ci potrzebny sposób na odczytywanie i zapisywanie wszystkich rodzajów struktur, które zechcesz przechowywać w plikach. Zobaczmy, jak można wczytać z pliku strukturę KartotekaGracza. Najpierw weźmiemy się za łatwą część: zresetowanie pozycji w pliku i wczytanie pól wiek oraz wynik, które zostały zapisane na dysku bezpośrednio bez zmieniania ich formatu:

```
plik.seekg( 0, ios::beg );  
  
KartotekaGracza kartot;  
if ( ! plik.read( reinterpret_cast<char*>( & kartot.wiek ), sizeof( kartot.wiek ) ) )  
{  
    // Obsługa błędu  
}  
if ( ! plik.read( reinterpret_cast<char*>( & kartot.wynik ), sizeof( kartot.wynik ) ) )  
{  
    // Obsługa błędu  
}
```

A co z odczytaniem łańcucha? Nie możemy po prostu wczytać w miejsce naszej zmiennej łańcuchowej wskaźnika `char*` z pliku. Format typu `string` w pamięci jest inny niż na dysku. Musimy wczytać `char*`, a następnie utworzyć nowy łańcuch.

Teraz już wiesz, dlaczego musielibyśmy zapisać długość łańcucha — ponieważ musimy wiedzieć, ile miejsca należy przeznaczyć na zmienną `char*`. Wczytamy zatem długość łańcucha, zaalokujemy dla niego pamięć, a następnie wczytamy w to miejsce łańcuch.

```
int dlugosc_str;  
  
if ( ! plik.read( reinterpret_cast<char*>( & dlugosc_str ), sizeof( dlugosc_str ) ) )  
{  
    // Obsługa błędu  
}  
// Test rozsądku mający na celu upewnienie się, że nie próbujemy zaalokować zbyt  
// dużo pamięci  
else if ( dlugosc_str > 0 && dlugosc_str < 10000 )  
{  
    char *w_str_buf = new char[ dlugosc_str ];  
    if ( ! plik.read( w_str_buf, dlugosc_str + 1 ) ) // + 1 na znak końca NULL  
    {  
        // Obsługa błędu  
    }  
    // Sprawdzenie, czy łańcuch jest zakończony wartością NULL
```

---

<sup>6</sup> Najważniejszą różnicą do odnotowania jest fakt, że wskaźnik przekazywany do metody `write` może być stały, co oznacza możliwość zapisywania do pliku obiektów typu `const`. W tym przypadku, przekazując obiekt stały, należy użyć instrukcji `reinterpret_cast<const char*>` (zwróć uwagę na użycie w rzutowaniu słowa kluczowego `const`).

```

    if ( w_str_buf[ dlugosc_str ] == 0 )
    {
        kartot.nazwa = string( w_str_buf );
    }
    delete[] w_str_buf;
}

cout << kartot.wiek << " " << kartot.wynik << " " << kartot.nazwa << endl;

```

Oto pełna wersja programu, z którą możesz eksperymentować:

### Przykładowy kod 72.: *binary.cpp*

```

#include <iostream>
#include <string>
#include <fstream>

using namespace std;

struct KartotekaGracza
{
    int wiek;
    int wynik;
    string nazwa;
};

int main ()
{
    KartotekaGracza kart;
    kart.wiek = 11;
    kart.wynik = 200;
    kart.nazwa = "Jan";

    fstream plik( "kartoteki.bin", ios::trunc | ios::binary | ios::in | ios::out );
    plik.write( reinterpret_cast<char*>( & kart.wiek ), sizeof( kart.wiek ) );
    plik.write( reinterpret_cast<char*>( & kart.wynik ), sizeof( kart.wynik ) );
    int dlugosc = kart.nazwa.length();
    plik.write( reinterpret_cast<char*>( & dlugosc ), sizeof( dlugosc ) );
    plik.write( kart.nazwa.c_str(), kart.nazwa.length() + 1 );

    KartotekaGracza kartot;

    plik.seekg( 0, ios::beg );
    if ( ! plik.read( reinterpret_cast<char*>( & kartot.wiek ),
    ↳sizeof( kartot.wiek ) ) )
    {
        cout << "Błąd odczytu z pliku" << endl;
        return 1;
    }
    if ( ! plik.read( reinterpret_cast<char*>( & kartot.wynik ),
    ↳sizeof( kartot.wynik ) ) )
    {
        cout << "Błąd odczytu z pliku" << endl;
        return 1;
    }

    int dlugosc_str;

```

```
if ( ! plik.read( reinterpret_cast<char*>( & dlugosc_str ),  
                   sizeof( dlugosc_str ) ) )  
{  
    cout << "Błąd odczytu z pliku" << endl;  
    return 1;  
}  
  
// Test rozsądku mający na celu upewnienie się, że nie próbujemy zaalokować zbyt  
// dużo pamięci  
if ( dlugosc_str > 0 && dlugosc_str < 10000 )  
{  
    char *w_str_buf = new char[ dlugosc_str + 1 ];  
    if ( ! plik.read( w_str_buf, dlugosc_str + 1 ) ) // + 1 na znak końca NULL  
    {  
        delete[] w_str_buf;  
        cout << "Błąd odczytu z pliku" << endl;  
        return 1;  
    }  
    // Sprawdzenie, czy łańcuch jest zakończony wartością NULL  
    if ( w_str_buf[ dlugosc_str ] == 0 )  
    {  
        kartot.nazwa = string( w_str_buf );  
    }  
    delete[] w_str_buf;  
}  
  
cout << kartot.wiek << " " << kartot.wynik << " " << kartot.nazwa << endl;  
}
```

Po uruchomieniu powyższego programu spróbuj otworzyć plik *kartoteki.bin* w Notatniku albo innym edytorem tekstowym. Powinno Ci się udało odczytać imię Jan, ponieważ jest ono zapisane w postaci znaków łańcucha tekstopowego, ale pozostałe dane w pliku będą wydawały się pozbawione sensu.

## Sprawdź się

1. Z którego typu można korzystać, aby czytać z pliku?
  - A. ifstream
  - B. ofstream
  - C. fstream
  - D. Prawdziwe są odpowiedzi A i C.
2. Które z poniższych stwierdzeń jest prawdziwe?
  - A. Pliki tekstowe zajmują mniej miejsca niż pliki binarne.
  - B. W plikach binarnych łatwiej znajdować błędy.
  - C. Pliki binarne zajmują mniej miejsca niż pliki tekstowe.
  - D. Pliki tekstowe są zbyt wolne, aby korzystać z nich w prawdziwych programach.
3. Dlaczego podczas zapisywania do pliku binarnego nie można przekazywać wskaźnika do obiektu typu *string*?
  - A. Do metody *write* zawsze należy przekazywać *char\**.
  - B. Obiekt typu *string* nie może być przechowywany w pamięci.

- C. Nie znamy układu obiektu typu `string` — może on zawierać wskaźniki, które zostałyby zachowane w pliku.
- D. Łańcuchy tekstowe są zbyt duże i muszą być zapisywane po kawałku.
- 4.** Które z poniższych stwierdzeń na temat formatu pliku jest prawdziwe?
- Formaty plików, tak samo jak wszystkie inne dane wejściowe, można łatwo zmieniać.
  - Zmiana formatu pliku wymaga przemyślenia, co się stanie, gdy stara wersja programu spróbuje odczytać nową wersję pliku.
  - Projektowanie formatu pliku wymaga przemyślenia, co się stanie, gdy nowa wersja programu spróbuje odczytać starą wersję pliku.
  - Prawdziwe są odpowiedzi B i C.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

- Ponownie zaimplementuj tekstową wersję programu wstawiającego punktację we właściwe miejsca pliku, ale tym razem zamiast tekstuowego formatu pliku użyj formatu binarnego. Skąd będziesz wiedzieć, że Twój program działa? Utwórz program, który wyświetla plik w postaci tekstowej.
- Zmodyfikuj analizator składni HTML, który zaimplementowałeś w rozdziale 19., „Więcej o łańcuchach tekstowych”, w taki sposób, aby mógł wczytywać dane z pliku dyskowego.
- Napisz prosty analizator składni XML. XML jest językiem formatującym, podobnym do języka HTML. Dokument XML ma drzewistą strukturę węzłów o postaci `<wezel>[dane]</wezel>`, gdzie [dane] są tekstem lub innym zagnieżdżonym węzłem. Węzły XML mogą mieć wiele atrybutów o postaci `<atrybut wezla="wartosc"></wezel>` (rzeczywista specyfikacja języka XML zawiera o wiele więcej szczegółów, ale wymagałyby one znacznie więcej pracy, aby je zaimplementować). Twój analizator powinien przyjmować klasę interfejsową z kilkoma metodami, które są wywoływane, gdy wydarzy się coś interesującego:
  - Gdy odczytywany jest węzeł, powinna zostać wywołana metoda `poczatekWezla` z nazwą tego węzła.
  - Gdy odczytywany jest atrybut, powinna zostać wywołana metoda `odczytAtrubutu`. Metoda ta powinna być wywoływana dla węzła, z którym skojarzony jest atrybut, bezpośrednio po uruchomieniu metody `poczatekWezla`.
  - Jeżeli węzeł zawiera tekst, wywołaj metodę `odczytTekstuWezla` z treścią w postaci łańcucha tekstuowego. Jeśli natrafisz na taką sytuację jak `<wezel>tekst<podwezel>tekst</podwezel>kolejny tekst</wezel>`, powinny nastąpić odrębne wywołania metody `odczytTekstuWezla` dla tekstu znajdującego się przed podwęzłem oraz tekstu występującego za podwęzłem.
  - Kiedy odczytany zostanie węzeł końcowy, wywołaj metodę `wuezKoncowy` z nazwą tego węzła.
  - Jako część znacznika węzła możesz traktować każdy ze znaków `< i >`. Jeśli autor dokumentu XML chce, aby znak `< albo >` pojawił się w tekście, powinien je zapisać jako `&lt;` lub `&gt;`; (od angielskich słów *less than* i *greater than*, czyli *większe od*

i mniejsze od). Ponieważ znaki & także muszą być modyfikowane, powinny one przyjmować postać &amp;;. W swoim kodzie nie musisz jednak dokonywać interpretacji znaków &lt;, &gt; ani &amp;;.

Oto dwa przykładowe dokumenty XML, które możesz wykorzystać jako testowe dane wejściowe dla swojego programu:

```
<ksiazka-adresowa>
<pozycja>
    <nazwa>Alex Allain</nazwa>
    <email>webmaster@cprogramming.com</email>
</pozycja>
<pozycja>
    <nazwa>Jan NN</nazwa>
    <email>jan@nn.com</email>
</pozycja>
</ksiazka-adresowa>
```

oraz

```
<html>
    <head>
        <title>Tytul dokumentu</title>
    </head>
    <body>Oto <a href="http://www.cprogramming.com">link</a> do fajnej
        strony.</body>
</html>
```

Aby przekonać się, czy Twój analizator składni działa prawidłowo, możesz napisać kod wyświetlający każdy z elementów pliku podczas jego przetwarzania, a także sprawdzić, czy otrzymuje on wszystkie potrzebne elementy. Możesz także wykonać następne zadanie praktyczne, w którym skorzystasz ze swojego analizatora.

4. Przepisz swój analizator składni HTML w taki sposób, aby zamiast z ręcznie kodowanych danych korzystał z Twojego analizatora XML. Dodaj do niego możliwość wyświetlania list. Analizator powinien wczytywać znaczniki `<ul>` oraz `<n1>` dla list nieuporządkowanych i uporządkowanych. Każdy element listy powinien znajdować się między znacznikami `<li>` oraz `</li>`. Układ strony dla kodu:

```
<ul>
    <li>pierwszy element</li>
    <li>drugi element</li>
</ul>
```

powinien wyglądać następująco:

```
* pierwszy element
* drugi element
```

Natomiast dla kodu:

```
<n1>
    <li>pierwszy element</li>
    <li>drugi element</li>
</n1>
```

powinien wyglądać następująco:

```
1. pierwszy element
2. drugi element
```

Nie zapomnij uruchomić numeracji od 1, jeśli pojawi się druga lista numerowana!

## Szablony w C++

---

Do tej pory musiałeś określać typy wszystkiego, z czym miałeś do czynienia w C++. Deklarujesz zmienną? Potrzebujesz jej typu. Deklarujesz funkcję? Potrzebne będą Ci typy wszystkich jej parametrów, wartości zwrotnej oraz jej zmiennych lokalnych.

Czasami jednak będziesz chciał napisać kod uniwersalny. Użyte typy nie będą mieć znaczenia, ponieważ logika programu będzie taka sama dla wszystkich typów. Miałeś już do czynienia z takim rodzajem kodu, z tym że został on napisany przez kogoś innego; jest to biblioteka STL. STL jest zbiorem struktur danych (a także algorytmów), które działają w sposób ogólny. Mogą one pobierać dowolny typ danych, który Ty, programista, określisz. Kiedy chcesz zapisać w wektorze jakieś elementy, podajesz ich typ, jaki będzie tam przechowywany. Nie jesteś ograniczony do pracy wyłącznie ze wstępnie zdefiniowanymi rodzajami danych. Autorzy STL napisali jedną implementację wektora, która może przechowywać wszystkie rodzaje danych.

W jaki sposób udało im się zrealizować taką wspaniałą funkcjonalność? Otóż skorzystali oni z cechy języka C++, którą są **szablony**. Szablony umożliwiają napisanie „szablonu” funkcji lub klasy bez konieczności uwzględniania wszystkich typów. Gdy potrzebne będzie wsparcie określonego typu, kompilator utworzy **instancję** wersji szablonu, w której uzupełnione będą wszystkie typy. Tak właśnie się dzieje, kiedy piszesz instrukcję `vector<int> wekt;` — kompilator wypełnia szablon `vector` typem `int`, tworząc nadającą się do użycia klasę.

Korzystanie z szablonów jest, jak to już widziałeś, całkiem proste. Rozdział ten w całości został poświęcony *tworzeniu* własnych szablonów funkcji oraz klas. Zaczniemy od przyjrzenia się szablonom funkcji.

## Szablony funkcji

Szablony doskonale nadają się do tworzenia funkcji ogólnych. Mógłbyś na przykład wziąć pod uwagę napisanie małej funkcji pomocniczej obliczającej powierzchnię trójkąta:

```
int powierzchniaTrojkata (int podstawa, int wysokosc)
{
    return podstawa * wysokosc * 0.5
}
```

Co by się stało, gdybyś spróbował obliczyć powierzchnię trójkąta o wysokości równej 0,5 i podstawie wynoszącej 0,5? Wartości te zostałyby obcięte do liczb całkowitych i funkcja zwróciłaby 0, chociaż powierzchnia trójkąta wcale nie jest równa 0.

Alternatywą jest napisanie kolejnej metody:

```
double powierzchniaTrojkataDouble (double podstawa, double wysokosc)
{
    return podstawa * wysokosc * 0.5
}
```

Powyższy kod wygląda dokładnie tak samo jak kod pierwszej funkcji, oprócz wiersza z deklaracją typów, gdzie zamiast typu `int` znajduje się typ `double`. Gdybyśmy chcieli zrealizować to samo z jeszcze jednym typem, być może jakimś niestandardowym typem numerycznym, musielibyśmy napisać trzecią implementację naszej funkcji.

W takich sytuacjach idealnie sprawdzają się szablony C++. Szablon umożliwia „wyciągnięcie” typów na zewnątrz. Gdy obiekt wywołujący funkcję poda typy, które mają zostać użyte, kompilator wygeneruje funkcję dla każdego z podanych typów.

Składnia deklaracji szablonu może wyglądać trochę zniechęcająco, ale rozbijemy ją na części, dzięki czemu nabierze większego sensu. Oto jak należy napisać naszą funkcję w postaci szablonu:

```
template <typename T>
T powierzchniaTrojkata (T podstawa, T wysokosc)
{
    return podstawa * wysokosc * 0.5
}
```

Najpierw za pomocą słowa kluczowego `template` deklarujemy, że funkcja jest szablonem. Następnie w nawiasach ostrokatnych wymieniamy parametry szablonu. Parametry te to wartości, które określi użytkownika szablonu (np. `int` w wyrażeniu `vector<int>`). Parametr szablonu powinien być typem, a nie wartością, w związku z tym używamy słowa kluczowego `typename`. Bezpośrednio po nim umieszczać nazwę parametru `T`. Wszystko to bardzo przypomina deklarowanie argumentów funkcji. Kiedy obiekt wywołujący funkcję poda jako parametr szablonu typ, każde odwołanie do `T` w szablonie zostanie potraktowane tak, jakby to był ten właśnie typ. Takie działanie również jest podobne do użycia argumentu funkcji w celu skorzystania z wartości przekazanej do tej funkcji.

Jeśli na przykład napiszemy:

```
powierzchniaTrojkata<double>(0.5, 0.5);
```

wszystkie miejsca, w których `T` pojawiło się w kodzie, zostaną zastąpione typem `double`. Stanie się tak, jakbyśmy napisali funkcję `powierzchniaTrojkataDouble`. Utworzony przez nas kod jest dosłownie szablonem, z którego skorzystał kompilator, aby utworzyć wyspecjalizowaną funkcję obsługującą typ `double`.

Innymi słowy, poniższy wiersz kodu:

```
template <typename T>
```

można odczytać następująco: „Definiowana funkcja (albo klasa) jest szablonem. Wewnątrz niego użyję literę `T`, która zastępować będzie nazwę typu — takiego jak `int`, `double` albo `char` — lub klasy. Kiedy ktoś zechce użyć tego szablonu, będzie musiał podać typ, który zastąpi literę `T`. W tym celu umieści go w nawiasach ostrokatnych (`<>`) znajdujących się przed nazwą funkcji lub klasy”.

## Inferencja typów

W niektórych przypadkach obiekt wywołujący szablon funkcji nie musi jawnie podawać parametrów szablonu. Kompilator na podstawie argumentów przekazanych do funkcji często potrafi określić, jakie powinny być wartości parametrów szablonu. Jeżeli na przykład napiszesz:

```
powierzchniaTrojkata(0.5, 0.5);
```

kompilator domyśli się, że T powinno przyjąć typ double. Dzieje się tak dlatego, że parametr szablonu T został użyty do zadeklarowania argumentów funkcji. Ponieważ kompilator zna typy argumentów, potrafi wywnioskować, jakim typem powinno być T.

Inferencja typów jest przeprowadzana za każdym razem, gdy parametr szablonu jest używany jako typ któregoś z argumentów funkcji.

## Kacze typowanie

Jest takie powiedzenie, że jeśli coś „wygląda jak kaczka, chodzi jak kaczka i kwacze jak kaczka, to musi to być kaczka”. Zdumiewające, ale powiedzenie to często jest stosowane w odniesieniu do szablonów języka C++. Zaraz wyjaśnię dlaczego.

Kiedy przekazujesz parametr szablonu, kompilator musi zdecydować, czy parametr ten jest w danym szablonie poprawny. Na przykład w szablonie `oblicz_równanie` wartości przekazywane do funkcji powinny mieć wsparcie dla takich operatorów arytmetycznych jak dodawanie i mnożenie:

```
return x * y * 4 * z + y * z + x;
```

Niekórych typów nie można jednak mnożyć. Liczby całkowite i podwójnej precyzji mogą być przez siebie mnożone, chociaż należą do różnych typów, ale co z wektorem `vector<int>?` Pomyśl mnożenia wektorów jest absurdalny; takie mnożenie nie ma sensu, a klasa `vector` tego działania nie obsługuje.

Jeśli do funkcji `oblicz_równanie` spróbujesz przekazać trzy wektory, kod z jej wywołaniem nie skompiluje się:

```
ZŁY KOD
int main ()
{
    vector<int> a, b, c;
    compute_equation( a, b, c );
}
```

Kompilator jest bardzo precyzyjny i wskaże Ci, których operacji nie obsługuje wektor `vector<int>:`

```
template_compile.cc: In function 'T oblicz_równanie(T, T, T) [with T = std::vector<int,
           &std::allocator<int> >]':
template_compile.cc:13: instantiated from here
template_compile.cc:5: error: no match for 'operator*' in 'y * z'
template_compile.cc:5: error: no match for 'operator*' in 'x * y'
```

Komunikat o błędzie jest długi, ale możemy go rozbić na części. Pierwszy wiersz zawiera informację, w którym szablonie funkcji wystąpił problem (`oblicz_równanie`). Drugi wiersz wskazuje linię kodu, w której próbujesz użyć tego szablonu. Zwykle jest to linia, którą powinieneś sprawdzić (przy okazji, zdanie `instantiated from here` znaczy po prostu „miejsce,

w którym próbowałeś korzystać z szablonu”). **Tworzenie instancji** (ang. *instantiate*) to używane przez programistów określenie oznaczające tworzenie czegoś; w tym przypadku próbowałeś utworzyć implementację szablonu `oblicz_rownanie` z parametrem `vector<int>`.

Kolejne dwa wiersze dokładnie opisują przyczynę niepowodzenia komplikacji. W tym przypadku jest to no `match for 'operator*' in 'x * y'`, co znaczy, że kompilator nie wiedział, jak pomnożyć `x` przez `y` (operator `*` nie jest zdefiniowany dla wektorów). Ponieważ obie zmienne są wektoram, można domyślić się, że wektory nie obsługują operacji mnożenia<sup>1</sup>.

Innymi słowy, wektor nie zachowuje się jak liczba — „nie wygląda jak liczba, nie chodzi jak liczba i nie kwacze jak liczba”. Kiedy użyty zostanie szablon funkcji, kompilator sprawdza, czy określony typ może działać w danym szablonie. Ważne wówczas jest tylko to, czy typ ten obsługuje wywoływanie metody i przeprowadzane operacje. Musi on jedynie „wyglądać” jak typ, który rzeczywiście działa.

Kacze typowanie znacznie różni się od sposobu działania funkcji polimorficznych. Funkcja polimorficzna pobiera wskaźnik do klasy interfejsowej i może wywoływać wyłącznie metody, które zostały w niej zdefiniowane. W przypadku szablonów nie ma żadnego wstępnie zdefiniowanego interfejsu, z którym określone parametry szablonu musiałyby być zgodne. Jeśli tylko parametr szablonu może zostać użyty w sposób gwarantujący spójność z działaniem funkcji, funkcja ta skompiluje się. Innymi słowy, jeżeli typ przekazany do szablonu „wygląda jak kaczka, chodzi jak kaczka i kwacze jak kaczka”, szablon potraktuje go jak kaczkę. Zwykle szablony nie oczekują jako parametru ptactwa wodnego, ale mam nadzieję, że już rozumiesz, dlaczego mówi się, że szablony korzystają z **kaczego typowania**. W takim przypadku ważne jest tylko to, żeby typ obsługiwał metody potrzebne do funkcjonowania szablonu.

## Szablony klas

Szablony klas często leżą w gestii autorów bibliotek, którzy chcą tworzyć takie klasy jak `vector` albo `map`. Niemniej zwykli programiści także mogą czerpać korzyści z możliwości tworzenia kodu, który będzie ogólny. Nie korzystaj z szablonów tylko dlatego, że masz taką możliwość, ale szukaj okazji do pozbycia się klas, które różnią się wyłącznie użytym typem. Prawdopodobnie o wiele częściej będziesz pisać szablony metod niż szablony klas, ale wygodnie jest wieǳieć, jak z nich korzystać, na wypadek gdybyś na przykład zechciał zaimplementować własną, niestandardową strukturę danych.

Deklaracja szablonu klasy bardzo przypomina deklarowanie szablonu funkcji.

Możemy na przykład napisać małą klasę opakowującą tablicę<sup>2</sup>:

```
template <typename T> class OpakowanieTablicy
{
private:
```

<sup>1</sup> Był może zastanawiasz się, dlaczego kompilator nie protestuje w przypadku dodawania wektorów. Oczywiście zaprotestowałby, gdyby dotarł do tej operacji. Kompilator zauważał jednak problem z mnożeniem i dał sobie spokój jeszcze przed osiągnięciem dodawania.

<sup>2</sup> W programowaniu termin **opakowanie** jest stosowany wtedy, gdy jedna funkcja wywołuje drugą funkcję w celu skorzystania z większości jej funkcjonalności, przy czym funkcja zewnętrzna wykonuje jeszcze dodatkowo niewielką pracę, taką jak logowanie albo sprawdzanie błędów. W naszym przypadku główną metodą jest metoda użyta w celu zaimplementowania metody zewnętrznej; mówi się, że metoda zewnętrzna opakowuje metodę główną.

```
T *_w_pam;
};
```

Tak samo jak w przypadku szablonu funkcji, zaczynamy od zadeklarowania za pomocą słowa kluczowego `template`, że mamy zamiar utworzyć szablon, po czym podajemy listę jego parametrów. W tym przypadku istnieje tylko jeden parametr — `T`.

Typu `T` możemy użyć w jakimkolwiek miejscu, które określiliby użytkownika, tak samo jak w przypadku szablonu funkcji.

Podczas definiowania funkcji w szablonie klasy należy przestrzegać składni szablonu. Założymy, że do szablonu `OpakowanieTablicy` chcemy dodać konstruktor:

```
template <typename T> class OpakowanieTablicy
{
public:
    OpakowanieTablicy (int rozmiar);
private:
    T *_w_pam;
};

// Teraz, aby zdefiniować konstruktor poza klasą, musimy rozpocząć
// od zaznaczenia, że funkcja jest szablonem
template <typename T>
OpakowanieTablicy<T>::OpakowanieTablicy (int rozmiar)
    : _w_pam( new T[ rozmiar ] )
{ }
```

Zaczynamy od podstawowego zapisu związanego z tworzeniem szablonu, ponownie deklarując parametr szablonu. Jedyna różnica polega na tym, że do nazwy klasy dołączony jest symbol szablonu (`OpakowanieTablicy<T>`), dzięki czemu wiadomo, iż nazwa ta odnosi się do szablonu klasy, a nie do szablonu funkcji z nieszablonowej klasy o nazwie `OpakowanieTablicy`.

W implementacji tej metody możemy użyć parametru szablonu jako symbolu zastępującego przekazywany do niej typ, tak samo jak w przypadku szablonów funkcji. W przeciwieństwie jednak do szablonów funkcji, obiekt wywołujący funkcję nie musi nawet przekazywać parametru do szablonu; zostanie on pobrany ze wstępnej deklaracji typu szablonu. Kiedy na przykład odczytujesz rozmiar wektora liczb całkowitych, nie musisz pisać `wekt.size<int>()` ani `wekt<int>.size();` wystarczy, że napiszesz `wekt.size()`.

## Wskazówki dotyczące pracy z szablonami

Często łatwiej jest najpierw napisać dla określonego typu klasę, a dopiero później przepisać ją w postaci szablonu. Możesz na przykład zadeklarować klasę przy użyciu liczb całkowitych, po czym na podstawie tej deklaracji utworzyć szablon ogólny. Tego typu podejście nie jest konieczne i nie musisz go stosować, jeśli dość dobrze sobie radzisz z szablonami. Jeżeli jednak piszesz swoje pierwsze szablony, metoda ta pozwoli Ci odseparować problemy związane ze składnią szablonów od kłopotów z implementacją algorytmu.

Spójrzmy na przykład na prostą klasę realizującą kalkulator, która na razie działa tylko na liczbach całkowitych:

```
class Kalk
{
public:
    Kalk ();
    int mnozenie (int x, int y);
```

```

        int dodawanie (int x, int y);
    };

Kalk::Kalk ()
{
}

int Kalk::mnozenie (int x, int y)
{
    return x * y;
}

int Kalk::dodawanie (int x, int y)
{
    return x + y;
}

```

Klasa ta całkiem dobrze radzi sobie z liczbami całkowitymi. Teraz możemy przekształcić ją w szablon, dzięki czemu będziemy mogli przystosować nasz kalkulator także do liczb niecałkowitych:

### **Przykładowy kod 73.: *kalk.cpp***

```

template <typename Typ>
class Kalk
{
public:
    Kalk ();
    Typ mnozenie (Typ x, Typ y);
    Typ dodawanie (Typ x, Typ y);
};

template <typename Typ> Kalk<Typ>::Kalk ()
{}

template <typename Typ> Typ Kalk<Typ>::mnozenie (Typ x, Typ y)
{
    return x * y;
}

template <typename Typ> Typ Kalk<Typ>::dodawanie (Typ x, Typ y)
{
    return x + y;
}

int main ()
{
    // Demonstracja deklaracji
    Kalk<int> c;
}

```

Aby przeprowadzić przekształcenie w szablon, musielibyśmy wprowadzić kilka zmian. Najpierw zadeklarowaliśmy, że istnieje szablon o nazwie Typ:

```
template <typename Typ>
```

Następnie musielibyśmy umieścić powyższą deklarację szablonu przed klasą i przed każdą definicją funkcji:

```
template <typename Typ> class Kalk
template <typename Typ> int Kalk::mnozenie (int x, int y)
```

Musieliśmy także zmodyfikować definicję każdej funkcji w taki sposób, aby pokazać, że jest ona częścią szablonu klasy:

```
template <typename Typ> int Kalk<Typ>::mnozenie (int x, int y)
```

Na koniec zastąpiliśmy wszystkie odwołania do `int` odwołaniami do `Typ`:

```
template <typename Typ> Typ Kalk<Typ>::mnozenie (Typ x, Typ y)
```

Kiedy już przyzwyczaisz się do stosowania szablonów, przekształcenie klasy zdefiniowanej pod konkretny typ w szablon klasy działający z wieloma różnymi typami stanie się prostą, mechaniczną transformacją<sup>3</sup>. Wraz z upływem czasu zaczniesz stosować składnię szablonów na tyle swobodnie, że będziesz mógł pisać szablony klas od podstaw, bez potrzeby tworzenia jakiegokolwiek pośredniego kodu.

## Szablony i pliki nagłówkowe

Do tej pory przyglądaliśmy się szablonom, które znajdowały się bezpośrednio w plikach `.cpp`. Co by się stało, gdybyśmy zechcieli umieścić deklarację szablonu w pliku nagłówkowym? Problem wiążący się z takim rozwiążaniem polega na tym, że kod korzystający z szablonu funkcji lub klasy musi mieć dostęp do pełnej definicji tego szablonu (a także do wszystkich funkcji składowych wywoływanych w szablonie klasy) przy każdym wywołaniu funkcji szablonu. Jest to działanie diametralnie różne od wywoływanego zwykłych funkcji, gdy wymagane jest, aby obiekt wywołujący funkcję znał tylko jej deklarację. Jeśli na przykład umieścisz klasę `Kalk` we własnym pliku nagłówkowym, musiałbyś uwzględnić w tym pliku także pełną definicję konstruktora oraz metodę dodawanie, zamiast zamieszczać je standardowo w pliku `.cpp`. Gdybyś tak nie postąpił, wywołanie metody `Kalk` by się nie powiodło.

Ta niefortunna właściwość szablonów wynika ze sposobu, w jaki szablony są komplilowane. Gdy kompilator przetwarza szablony, przeważnie je ignoruje i tylko wtedy, gdy użyjesz szablonu z pewnym konkretnym typem (jak na przykład przy wywołaniu `Kalk<int>`), wygeneruje kod dla szablonu korzystającego z tego typu (w tym przypadku `int`). Aby wygenerować taki kod, większość kompilatorów musi dysponować szablonem, który to umożliwia. W rezultacie do każdego pliku, w którym wykorzystano dany szablon, musisz dołączyć jego pełny kod. Co więcej, podczas komplilowania pliku zawierającego szablon możesz nie dowiedzieć się niczego o błędach składni w szablonie, które pojawią się dopiero wtedy, gdy ktoś po raz pierwszy go użyje.

Kiedy tworzysz szablon klasy, na ogólny najprostsze rozwiązanie polega na umieszczeniu wszystkich definicji szablonu w pliku nagłówkowym. Aby jednoznacznie wskazać, że plik taki zawiera szablon, możesz mu nadać rozszerzenie inne niż `.h`, na przykład `.hxx`.

## Podsumowanie informacji o szablonach

Szablony umożliwiają tworzenie ogólnego kodu — takiego, którego użycie nie będzie ograniczone wyłącznie do jednego typu, na przykład całkowitego, ale który będzie działać dla każdego typu. Szablony często są używane do implementowania bibliotek języka C++ (takich jak na przykład standardowa biblioteka szablonów). Prawdopodobnie przekonasz się, że nie

<sup>3</sup> Uważaj jednak, aby nie popaść w rutynę. Jeśli na przykład licznik pętli używa liczb całkowitych, nie powinieneś zmieniać jego typu.

będziesz musiał zbyt często pisać kodu dla szablonów, chociaż powinieneś zwracać uwagę na kody wykazujące identyczną strukturę i różniące się użyтыm typem. Może na przykład okazać się, że tworzysz kod przeglądający w pętli różne rodzaje wektorów i że za każdym razem przeprowadzasz na tych wektorach takie same operacje. W wielu przypadkach przekonasz się, że szablon jest Ci potrzebny wtedy, gdy pracujesz z jakimś typem, dla którego szablon już istnieje, jak na przykład w przypadku kontenerów STL.

Przykładowo możesz napisać funkcję dodającą wartości w wektorze i drugą funkcję, która łączy zapisane w wektorzełańcuchy tekstowe. Obie te funkcje mają taką samą podstawową implementację, polegającą na przechodzeniu przez wektor w pętli i użyciu operatora +, ale działają na różnych typach. Kiedy napotkasz taki kod, postępuj zgodnie z zasadą, aby się nie powtarzać. Jeśli masz zamiar napisać kod, który robi to samo na dwóch różnych typach, nie twórz dwóch odrębnych implementacji, tylko skorzystaj z szablonu.

## Interpretacja komunikatów o błędach w szablonach

Wadą korzystania z szablonów są trudne do zrozumienia komunikaty o błędach generowane przez większość kompilatorów, gdy szablon zostanie nieprawidłowo użyty. Może się tak stać, nawet jeśli nie napisałeś szablonu, na przykład podczas korzystania z STL. Jedna pomyłka może sprawić, że zostaniesz zasypany całymi stronami komunikatów o błędach. Komunikaty o błędach w użyciu szablonów są trudne do czytania, ponieważ parametry szablonów są w nich rozwijane do pełnych typów, co dotyczy nawet parametrów, z których zwykle nie korzystasz (są one udostępniane jako parametry domyślne).

Weźmy na przykład taką, niewinnie wyglądającą deklarację wektora:

```
vector<int, int> vec;
```

Rzec w tym, że do deklaracji tej wkradła się drobna pomyłka — powinien w niej występować tylko jeden parametr szablonu. Kiedy jednak spróbujesz ją skompilować, pojawi się zawrotna liczba błędów:

```
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h: In instantiation of 'std::_Vector_base<int, int>':
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:159: instantiated from 'std::vector<int, int>'
template_err.cc:6: instantiated from here
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:78: error: 'int' is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/
↳bits/stl_vector.h:95: error: 'int' is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:99: error: 'int' is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h: In instantiation of 'std::_Vector_base<int, int>::_Vector_impl':
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:123: instantiated from 'std::_Vector_base<int, int>'
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:159: instantiated from 'std::vector<int, int>'
template_err.cc:6: instantiated from here
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:82: error: 'int' is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:86: error: 'int' is not a class, struct, or union type
```

```

/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h: In instantiation of 'std::vector<int, int>':
template_err.cc:6: instantiated from here
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:161: error: 'int' is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:193: error: no members matching 'std:::_Vector_base<int,
↳int>::_M_get_Tp_allocator' in 'struct std:::_Vector_base<int, int>'
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h: In destructor
'std::vector<Tp, _Alloc>::~vector() [with _Tp = int, _Alloc = int]':
template_err.cc:6: instantiated from here
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:272: error: '_M_get_Tp_allocator' was not declared in this scope
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h: In constructor 'std:: Vector_base<Tp, _Alloc>:: Vector_base
↳(const _Alloc&) [with _Tp = int, _Alloc = int]':
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:203: instantiated from 'std::vector<Tp, _Alloc>::vector
↳(const _Alloc&) [with _Tp = int, _Alloc = int]'
template_err.cc:6: instantiated from here
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:107: error: no matching function for call to 'std:::_Vector_base<int,
↳int>::_Vector_impl::Vector_impl(const int&)'
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:82: note: candidates are: std:::_Vector_base<int,
↳int>::_Vector_impl::Vector_impl(const std:::_Vector_base<int, int>::Vector_impl&)
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h: In member function 'void std:::_Vector_base<Tp,
↳_Alloc>::_M_deallocate(_Tp*, size_t) [with _Tp = int, _Alloc = int]':
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:119: instantiated from 'std:::_Vector_base<Tp,
↳_Alloc>::~_Vector_base() [with _Tp = int, _Alloc = int]'
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:203: instantiated from 'std::vector<Tp, _Alloc>::vector(const
↳_Alloc&) [with _Tp = int, _Alloc = int]'
template_err.cc:6: instantiated from here
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
↳stl_vector.h:133: error: 'struct std:::_Vector_base<int, int>::_Vector_impl' has
↳no member named 'deallocate'

```

O co w tym wszystkim chodzi? Kto jest odpowiedzialny za wymyślenie takiego komunikatu? Problem jest następujący: wektor ma drugi parametr, który jest domyślny w szablonie. Zazwyczaj kompilator uzupełnia go sam. Jeśli jednak umieścisz drugi parametr typu int, kompilator spróbuje go użyć, chociaż parametr ten nie powinien być liczbą całkowitą. Właśnie taką informację dostajesz niemal na samym początku listy z komunikatami o błędach.

error: 'int' is not a class, struct, or union type

Kod szablonu usiłuje użyć parametru w sposób, który jest nieprawidłowy dla typu całkowitego. Jeżeli na przykład masz taki kod:

```

template <typename T>
class Foo
{
    Foo ()
    {
        T x;
    }
}

```

```

        x.wart = 1;
    }
};

};

```

wówczas T nie może być liczbą całkowitą, ponieważ zmienna x (która jest typu T) musiałaby mieć pole o nazwie wart, a przecież zmienne całkowite nie mają żadnych pól, a już z pewnością nie mają pola o nazwie wart.

Gdybyśmy napisali

```
Foo<int> a;
```

kod taki by się nie skompilował.

Ponownie kłania się kacze typowanie. Szablon nie dba o to, jaki dokładnie typ otrzyma, ale zależy mu, aby typ ten „pasował” do kodu. W tym przypadku liczba całkowita nie jest zgodna ze składnią x.wart, a zatem kompilator ją odrzuca.

Szablon vector ma podobne ograniczenia co do drugiego parametru. Musi to być typ, który obsługiwał więcej funkcjonalności, niż udostępnia liczba całkowita. Wszystkie te błędy to informacje o różnego rodzaju sytuacjach, w których typ int będzie niepoprawny w roli parametru szablonu!

Mając do czynienia z taką ścianą tekstu, najlepiej jak zwykle zacząć od samego początku komunikatu o błędzie i starać się naprawiać po jednym błędzie naraz. Wydzielę teraz tekst do miejsca, w którym znajduje się słowo error, czyli po polsku „błąd”.

```

/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
→stl_vector.h: In instantiation of 'std::vector_base<int, int>':
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
→stl_vector.h:159: instantiated from 'std::vector<int, int>'
template_err.cc:6: instantiated from here
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/
→stl_vector.h:78: error: 'int' is not a class, struct, or union type

```

O wiele lepiej. Zostało już tylko kilka wierszy — podobnie jak w komunikacie o błędzie z podrozdziału o kaczym typowaniu. Damy radę!

Przeanalizujmy ten prostszy komunikat. Zwróć uwagę, że w pierwszym wierszu znajduje się tekst In instantiation of std::vector\_base<int, int> (czyli podczas tworzenia instancji std::vector\_base<int, int>). Tworzenie instancji szablonu oznacza próbę skompilowania szablonu z określonym zestawem jego parametrów. Taki błąd wskazuje na to, że wystąpił problem podczas tworzenia szablonu z danymi parametrami (vector\_base jest klasą pomocniczą używaną przy implementacji wektora). Następny wiersz informuje, że nie udało się skompilować szablonu vector\_base ze względu na próbę utworzenia szablonu vector<int, int> oraz komunikuje, że błąd wystąpił w 6. linii pliku template\_err.cc. Plik ten to nasz kod, wiemy już zatem, który wiersz kodu spowodował błąd.

Odnalezienie wiersza kodu z błędem zawsze stanowi pierwszy krok do określenia, co poszło źle. Bardzo często można stwierdzić, co jest nie tak, patrząc po prostu na kod. Jeśli problem nie wydaje się oczywisty na pierwszy rzut oka, możemy kontynuować przeglądanie listy instancji, aż dotrzemy do właściwego komunikatu o błędzie: error: 'int' is not a class, struct, or union type. Dzięki temu dowiesz się, że kompilator oczekiwał, iż Twoje int będzie klasą albo strukturą, a nie typem wbudowanym, takim jak liczba całkowita. Wektory powinny mieć możliwość przechowywania dowolnego typu, tak więc komunikat ten sugeruje, że wystąpił

problem z parametrem szablonu przekazanym do wektora. W tym momencie powinieneś sprawdzić, jak jest zadeklarowany, dzięki czemu stwierdziłeś, że potrzebny jest tylko jeden parametr szablonu.

Teraz, kiedy mamy już diagnozę pierwszego problemu, nadszedł czas, aby go naprawić i zrekomplilować kod. Zazwyczaj będziesz mógł jednorazowo namierzyć przynajmniej kilka błędów komilacji, chociaż w przypadku szablonów pierwszy błąd najczęściej pociąga za sobą wszystkie pozostałe. Najlepszym wyjściem będzie rozwiązywanie jednego problemu i uniknięcie zastanawiania się nad błędami, które zostały już naprawione.

W naszym przykładzie, w którym mieliśmy do czynienia z więcej niż jedną stroną komunikatów o błędach, przyczyną każdego błędu w programie było dodanie w parametrach szablonu drugiego int.

## Sprawdź się

- 1. Kiedy należy korzystać z szablonów?**
  - A. Kiedy chcesz zaoszczędzić na czasie.
  - B. Kiedy chcesz mieć szybciej działający kod.
  - C. Kiedy dla różnych typów musisz wiele razy pisać ten sam kod.
  - D. Kiedy chcesz mieć pewność, że później będziesz mógł korzystać z tego samego kodu.
- 2. Kiedy w parametrze szablonu należy podać typ?**
  - A. Zawsze.
  - B. Tylko podczas deklarowania instancji szablonu klasy.
  - C. Tylko wtedy, gdy nie można przeprowadzić inferencji typu.
  - D. W przypadku szablonów funkcji — jeśli nie można przeprowadzić inferencji typu; w przypadku szablonów klas — zawsze.
- 3. Skąd kompilator wie, że dany parametr szablonu może być użyty z danym szablonem?**
  - A. Implementuje specyficzny interfejs języka C++.
  - B. Podczas deklarowania szablonu należy określić ograniczenia.
  - C. Próbuje użyć parametru szablonu; jeśli jego typ obsługuje wszystkie wymagane operacje, przyjmuje go.
  - D. Podczas deklarowania szablonu należy podać listę wszystkich poprawnych typów.
- 4. Czym różni się umieszczenie szablonu klasy w pliku nagłówkowym od umieszczenia w takim pliku zwykłej klasy?**
  - A. Niczym.
  - B. Zwykła klasa nie może mieć zdefiniowanej w pliku nagłówkowym ani jednej swojej metody.
  - C. Szablon klasy musi mieć zdefiniowane w pliku nagłówkowym wszystkie swoje metody.
  - D. Szablon klasy nie potrzebuje odpowiadającego mu pliku .cpp, natomiast klasa musi mieć taki plik.

**5.** Kiedy należy przekształcić funkcję w szablon funkcji?

- A. Na samym początku. Nigdy nie wiadomo, czy tej samej logiki nie trzeba będzie użyć w odniesieniu do różnych typów, w związku z czym zawsze lepiej jest tworzyć szablony metod.
- B. Tylko wtedy, gdy nie można rzutować na typy, których akurat wymaga funkcja.
- C. Jeśli właśnie napisałeś niemal taką samą logikę, jaką już masz w kodzie, z tym że dla typu, który ma właściwości podobne do typu użytego we wcześniej istniejącej funkcji.
- D. Zawsze wtedy, gdy dwie funkcje robią „prawie” to samo i można dostosować ich implementację za pomocą kilku dodatkowych parametrów logicznych.

**6.** Kiedy dowiesz się o większości błędów popełnionych w kodzie szablonu?

- A. Jak tylko skompilujesz szablon.
- B. Na etapie konsolidacji.
- C. Podczas uruchamiania programu.
- D. Podczas pierwszej komplikacji kodu tworzącego instancję szablonu.

Odpowiedzi znajdują się na końcu książki.

## Zadania praktyczne

1. Napisz funkcję, która pobiera wektor i sumuje wszystkie wartości w nim zapisane, bez względu na typ danych liczbowych, które znajdują się w wektorze.
2. Zmodyfikuj klasę zastępującą wektor, którą zaimplementowałeś w ramach zadań praktycznych z rozdziału 25., w taki sposób, aby przekształcić ją w szablon mogący przechowywać dowolny typ.
3. Napisz metodę szukającą, która pobiera wektor dowolnego typu oraz dowolnego typu wartość i która zwraca prawdę, jeśli podana wartość zostanie odnaleziona w wektorze, oraz fałsz w przypadku przeciwnym.
4. Zaimplementuj funkcję sortującą, która przyjmuje dowolnego typu wektor i sortuje zapisane w nim wartości w ich naturalnej kolejności (uporządkowanie elementów uzyskasz za pomocą operatora < albo >).

# IV

▪ ▪ ▪ CZĘŚĆ IV

## Zagadnienia rozmaite

---

Poznałeś już narzędzia potrzebne do pisania ciekawych i dużych programów. Kilka zagadnień niezbyt pasowało do układu tej książki, chociaż są one bardzo przydatne. Jednym z tych tematów jest czytelne formatowanie danych pobieranych od użytkownika i wyświetlanych na ekranie. Temat ten bardziej wiąże się z interfejsem użytkownika niż z logiką algorytmu, niemniej jest równie ważny. Bez komunikowania się z użytkownikiem Twój program nigdy nie będzie interesujący!

Tematy zamieszczone w tej części możesz poznawać w dowolnej kolejności, w zależności od tego, co Cię interesuje. Niektóre z rozdziałów możesz przeczytać przed innymi, zwłaszcza jeśli poruszona w nich tematyka stanowi część kursu albo zajęć, na które uczęszczasz.



# ■ ■ ■ R O Z D Z I A Ł 3 0

## Formatowanie danych wyjściowych za pomocą iomanip

Czytelne formatowanie danych wyjściowych jest wymogiem często formułowanym przez nieznośnych użytkowników oprogramowania (następne żądanie, które Ci postawią, będzie takie, aby Twój program działał!). W języku C++ dane wyjściowe prezentowane instrukcją cout można przejrzeście formatować, korzystając z funkcji udostępnionych w pliku nagłówkowym iomanip.

### Rozwiązywanie problemów związanych z odstępami

Najczęściej spotykanym problemem, który dotyczy formatowania, jest niewłaściwy dobór odstępów. W poprawnie sformatowanych danych wyjściowych odstępy są rozmieszczone w taki sposób, że wszystko wygląda po prostu dobrze. Nie ma kolumn ze zbyt długim ani zbyt krótkim tekstem, a informacje są prawidłowo wyrównane. Dowiedzmy się zatem, jak to osiągnąć!

### Określanie szerokości pola za pomocą instrukcji setw

Funkcja setw umożliwia określenie minimalnej szerokości następnej danej, która ma zostać wyświetlona za pomocą operatora wstawiania. Jeśli dane wyjściowe będą krótsze od minimalnej szerokości, zostaną one dopełnione spacjami. Jeżeli długość danych wyjściowych będzie większa od minimalnej szerokości, nie zostanie podjęte żadne działanie, co ważne — dane wyjściowe nie zostaną obcięte.

Sposób korzystania z funkcji setw jest trochę dziwny — należy ją wywołać i przekazać jej wartość instrukcji cout:

#### Przykładowy kod 74.: setw.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    cout << setw( 10 ) << "dwa" << "trzy" << "trzy";
}
```

Wynik działania tego programu wygląda następująco:

dwa trzytrzy

Jeśli wywołasz funkcję setw bez przekazania jej wartości do cout, nie przyniesie ona żadnych skutków. Jak widać na powyższym przykładzie, wywołanie funkcji setw ma wpływ wyłącznie na dane wyjściowe, które są wyświetlane bezpośrednio po niej.

Instrukcja setw domyślnie wyrównuje łańcuch tekstowy do prawej strony (jest on dopełniany od strony lewej). Innymi słowy, łańcuch jest poprzedzany znakami dopełniającymi. Sposób wyrównywania można określić, przekazując funkcji cout kierunek wyrównania: left albo right (tj. do lewej albo do prawej). Poniższy program wyrówna tekst do strony lewej, dzięki czemu wyświetlany napis będzie czytelniejszy.

### Przykładowy kod 75.: setw\_left.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    cout << setw( 10 ) << left << "dwa" << "trzy" << "trzy";
}
```

Wynik działania powyższego programu wygląda następująco:

dwa trzytrzy

Tym razem pierwszy element został wyrównany do lewej strony.

Funkcja setw pozwala określać szerokość kolumny podczas działania programu. Aby na przykład wyświetlić kilka kolumn z danymi, mógłbyś odszukać najszerzy łańcuch w każdej kolumnie, po czym dopełnić wszystkie jej pozycje w taki sposób, aby były one nieco szersze od tego łańcucha.

## Zmiana znaku dopełniającego

Może się zdarzyć, że nie będziesz chciał użyć spacji jako znaku dopełniającego. Aby zmienić znak dopełniający, należy użyć funkcji setfill. Jej wywołanie przebiega podobnie do wywołania funkcji setw — należy ją przekazać bezpośrednio w funkcji cout.

Jeśli weźmiemy nasz wcześniejszy przykład i w jego kodzie wstawimy funkcję setfill z argumentem w postaci kreski:

```
cout << setfill( '-' ) << setw( 10 ) << "dwa" << "trzy" << "trzy";
```

wynik jego działania będzie wyglądać następująco:

-----dwa trzytrzy

## Trwała zmiana ustawień

Istnieje także możliwość zmiany znaku dopełniającego na stałe. W tym celu należy skorzystać z metody składowej fill funkcji cout. Na przykład kod:

```
cout.fill( '-' );
cout << setw( 10 ) << "A" << setw( 10 ) << "B" << setw( 10 ) << "C" << endl;
```

wyświetli następujący wiersz:

-----A-----B-----C

Metoda `fill` zwraca wcześniejszy znak dopełniający, co umożliwia zachowanie go na później. Ta wartość zwrotna jest przydatna, jeśli chcesz uniknąć wielokrotnych wywołań funkcji `setfill`. Na przykład po uruchomieniu następującego kodu:

```
const char ost_znak_dop = cout.fill( '-' );
cout << setw( 10 ) << "A" << setw( 10 ) << "B" << setw( 10 ) << "C" << endl;
cout.fill( ost_znak_dop );
cout << setw( 10 ) << "D" << endl;
```

ostatni wiersz zostanie wydrukowany jako:

D

Wyrównanie dopełnianego tekstu można trwale zmienić, wywołując metodę składową `setf` funkcji `cout`. Za pomocą flag `ios_base::left` albo `ios_base::right` można przekazywać do funkcji `setf`<sup>1</sup> informacje dotyczące wyrównania do strony lewej albo prawej:

```
cout.setf( ios_base::left );
```

Podobnie jak w przypadku funkcji `fill`, funkcja `setf` zwraca poprzednią wartość, którą, jeśli zajdzie taka potrzeba, można zachować na przyszłość.

Spróbuj uzupełnić poprzedni przykład o instrukcję `setf`, aby zobaczyć, w jaki sposób wpływa ona na formatowanie danych.

## Korzystanie ze znajomości iomanip

Wykorzystajmy naszą znajomość omówionych metod i napiszmy kod, który wyświetla imiona i nazwiska w dwóch kolumnach, wyrównując je w przejrzysty sposób:

Jan	Kowal
Teresa	Malicka
Jeremi	Noboginski
Maria	Zuzia-Fiolet

Musimy prawidłowo określić szerokość kolumn. Powinna być ona nieco większa od najszerszego elementu w każdej kolumnie. W pętli możemy sprawdzić elementy kolumn i odszukać ich maksymalne szerokości, po czym podczas wyświetlania danych wywoływać funkcję `setw` z argumentem określającym największą szerokość kolumny (być może zmieniając przy okazji znak dopełniający). Spójrzmy na kod realizujący to założenie:

### Przykładowy kod 76.: wyrownanie\_kolumn.cpp

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <string>

using namespace std;

struct Osoba
{
    Osoba (
```

<sup>1</sup> Nazwa funkcji `setf` to skrót od angielskich słów *set flag*, czyli *ustaw flagę*.

```
        const string& imie,
        const string& nazwisko
    )
    : _imie( imie )
    , _nazwisko( nazwisko )
    {}

    string _imie;
    string _nazwisko;
};

int main ()
{
    vector<Osoba> ludzie;

    ludzie.push_back( Osoba( "Jan", "Kowal" ) );
    ludzie.push_back( Osoba( "Teresa", "Malicka" ) );
    ludzie.push_back( Osoba( "Jeremi", "Noboginski" ) );
    ludzie.push_back( Osoba( "Maria", "Zuzia-Fiolet" ) );
    int imie_max_szer = 0;
    int nazwisko_max_szer = 0;

    // Odszukaj maksymalne szerokości

    for ( vector<Osoba>::iterator iter = ludzie.begin();
          iter != ludzie.end();
          ++iter )
    {
        if ( iter->_imie.length() > imie_max_szer )
        {
            imie_max_szer = iter->_imie.length();
        }
        if ( iter->_nazwisko.length() > nazwisko_max_szer )
        {
            nazwisko_max_szer = iter->_nazwisko.length();
        }
    }

    // Wyświetl elementy wektora
    for ( vector<Osoba>::iterator iter = ludzie.begin();
          iter != ludzie.end();
          ++iter )
    {
        cout << setw( imie_max_szer ) << left << iter->_imie;
        cout << " ";
        cout << setw( nazwisko_max_szer ) << left << iter->_nazwisko;
        cout << endl;
    }
}
```

## Wyświetlanie liczb

Przejrzyste wyświetlanie danych wymaga czasami poprawnego formatowania liczb. Podczas drukowania liczb szesnastkowych warto poprzedzać je znakami 0x, aby przekazać informację o ich podstawie. Wyświetlane dane będą czytelniejsze także wtedy, gdy w odpowiedni sposób określisz prezentowaną po przecinku liczbę zer (na przykład 2, jeśli masz do czynienia z pieniędzmi).

## Określanie precyzji wyświetlanego liczb za pomocą instrukcji setprecision

Funkcja `setprecision` definiuje maksymalną liczbę cyfr pokazywaną podczas wyświetlania danych numerycznych. Podobnie jak to jest w przypadku funkcji `setw`, wartość zwrotną funkcji `setprecision` należy przekazać do strumienia wyjściowego. Korzystanie z tej funkcji pod wszelkimi względami przypomina użycie funkcji `setw`. Aby w liczbie 2.71828 zostały wyświetlane łącznie trzy cyfry, należy napisać:

```
std::cout << setprecision( 3 ) << 2.71828;
```

Funkcja `setprecision` poprawnie zaokrąglą liczby, w związku z czym na ekranie nie pojawi się 2.71, tylko 2.72. Gdybyś jednak zastosował ją z liczbą 2.713, na ekranie pokazałby się wynik 2.71.

W przeciwieństwie do większości innych poleceń umieszczanych w strumieniu, `setprecision` zmieni precyzję wyświetlanego liczb do chwili ponownego jej wywołania. Jeśli więc następująco zmienisz kod w powyższym przykładzie:

```
cout << setprecision( 3 ) << 2.71828 << endl;
cout << 1.412 << endl;
```

na ekranie zostanie wyświetcone:

```
2.72
1.41
```

Być może zastanawiasz się, co się stanie, kiedy spróbujesz wydrukować wartość zawierającą przed kropką dziesiątną więcej cyfr, niż to określono w funkcji `setprecision`. Odpowiedź zależy od tego, czy chcesz wyświetlić liczbę zmienoprzecinkową, czy całkowitą. Liczba całkowita zostanie wydrukowana w całości, natomiast liczba zmienoprzecinkowa zostanie pokazana w notacji naukowej zadaną liczbą cyfr, zatem kod:

```
cout << setprecision( 2 ) << 1234.0 << endl;
wyświetli:
```

```
1.2e3
```

Przy okazji przypominam, że `e3` jest równoważne z zapisem  $\times 10^3$ .

Natomiast

```
cout << setprecision( 2 ) << 1234 << endl;
spowoduje wyświetlenie:
```

```
1234
```

## A co z pieniędzmi?

Być może zauważłeś, że do tej pory nie natrafiliśmy na rzeczywiście dobry sposób umożliwiający wyświetlanie liczb reprezentujących kwoty pieniężne, gdy zwykle chcemy mieć do dyspozycji dwa miejsca dziesiętne i nie życzymy sobie żadnego zaokrąglania.

Szybka odpowiedź na nasze potrzeby jest taka, że prawdopodobnie w ogóle nie powinniśmy przechowywać pieniędzy jako typ `double`! Jest tak dlatego, że wartości `double` nie są idealnie dokładne i mogą się w nie wdawać błędy zaokrągleń, powodując obcinanie tu i tam pojedynczych groszy. W większości aplikacji o wiele lepszym rozwiązaniem będzie przechowywanie łącznej liczby groszy w zmiennej całkowitej. Kiedy zechcesz wyświetlić daną kwotę, w celu

uzyskania idealnej precyzji podzieliś liczbę groszy przez 100 (dzięki czemu otrzymasz złotówkę), po czym skorzystasz z dzielenia modulo, co umożliwi Ci otrzymanie liczby groszy. Obie wyliczone w ten sposób kwoty wyświetlisz oddzielnie.

```
int grosze = 1001; // 10.01 zł
cout << grosze / 100 << "." << setw(2) << setfill('0') << grosze % 100;
```

Sensowne byłoby rzecz jasna napisanie funkcji pomocniczej wykonującej powyższe obliczenia oraz utworzenie klasy przechowującej pieniądze, co umożliwiłoby ukrycie szczegółów związanych z zastosowanym formatem liczbowym.

## Wyświetlanie liczb o różnych podstawach

Podczas pisania programów bardzo często potrzebujesz wyświetlać liczby w formacie ósemkowym lub szesnastkowym. W tym celu możesz korzystać z funkcji setbase. Funkcja ta, wstawiona do strumienia, ustawia podstawę liczby na 8, 10 albo 16. Na przykład instrukcja

```
cout << "0x" << setbase( 16 ) << 32 << endl;
```

wyświetli na ekranie:

0x20

co jest liczbą 32 w zapisie szesnastkowym. Podczas wstawiania danych do strumienia dla wywołań setbase ( 10 ), setbase( 8 ) i setbase( 16 ) można używać skróconych zapisów, odpowiednio: dec, oct oraz hex.

W powyższym kodzie jawnie nakazaliśmy drukowanie prefiksu 0x. Można jednak również korzystać z funkcji setiosflags, która powoduje automatyczne wyświetlanie podstawy. Jeśli do cout przekażesz wynik wywołania setiosflags( ios\_base::showbase ), liczby dziesiętne będą wyświetlane standardowo, szesnastkowe będą poprzedzane prefiksem 0x, natomiast ósemkowe prefiksem 0. Zatem kod:

```
cout << setiosflags( ios_base::showbase ) << setbase( 16 ) << 32 << endl;
```

spowoduje wyświetlenie na ekranie:

0x20

Podobnie jak to jest w przypadku funkcji setprecision, zmiany wprowadzane za pomocą funkcji setiosflags są trwałe. Wyświetlanie prefiksów można wyłączyć za pomocą argumentu noshowbase.

Opisane w tym rozdziale narzędzia powinny umożliwić Ci wyświetlanie danych w sposób, który będzie o wiele przyjemniejszy dla oka.

# ■ ■ ■ ROZDZIAŁ 31

## Wyjątki i raportowanie błędów

---

W miarę tworzenia coraz większych programów coraz bardziej będziesz potrzebował eleganckiego sposobu na obsługę błędów zgłaszanych przez Twoje funkcje. Istnieją dwie klasyczne metody raportowania błędów: za pomocą kodów oraz przy użyciu wyjątków. Implementacja kodów błędów nie wymaga żadnych nowych funkcjonalności języka — sprowadza się do tego, że każda funkcja (która może zawiść) zwraca kod błędu (albo sukcesu), który informuje o wyniku jej działania. Przewaga tej techniki polega na tym, że jest ona względnie prosta do zrozumienia:

```
int zawodnaFunkcja () ;

const int wynik = zawodnaFunkcja();

if ( wynik != 0 )
{
    cout << "Wywołanie funkcji nie powiodło się: " << wynik;
}
```

Z drugiej jednak strony, wadą takiego sposobu obsługi błędów jest wymóg, aby każda funkcja zwracała kod błędu, nawet jeśli trzeba od niej uzyskać jeszcze inną wartość. W celu otrzymania od takiej funkcji wartości obliczanej należy skorzystać z referencji albo parametru wskaźnikowego:

```
int zawodnaFunkcja () ;

int wart_obl;

const int wynik = zawodnaFunkcja( wart_obl );

if ( wynik != 0 )
{
    cout << "Wywołanie funkcji nie powiodło się: " << wynik;
}
else
{
    // Użyj wart_obl, żeby coś zrobić
}
```

Chociaż tego typu podejście działa, kod traci swój naturalny przebieg, którego należałoby po nim oczekiwać.

Wyjątki natomiast są zupełnie nową funkcjonalnością języka. Wyjątki działają w taki sposób, że gdy funkcja chce zaraportować wystąpienie błędu, natychmiast zatrzymuje się i zgłasza wyjątek. Po jego zgłoszeniu program szuka funkcji obsługującej wyjątek, która dalej się nim zajmie.

Funkcjonowanie wyjątku można postrzegać następująco: funkcja bezzwłocznie kończy swoje działanie bez przekazywania wartości zwrotnej. Co więcej, zamiast wracać do obiektu, który ją wywołał, funkcja przejdzie do miejsca, w którym błąd może zostać obsłużony. Jeśli takiego miejsca nie ma, w programie nastąpi awaria spowodowana nieobsłużonym wyjątkiem. Jeśli natomiast takie miejsce istnieje, program będzie kontynuować działanie w tym właśnie miejscu. Dzięki temu można pisać kod zawierający jeden punkt, do którego będą „udawać się” wszystkie niepowodzenia, po to aby zostały tam obsłużone.

W celu określenia miejsca, do którego w przypadku porażki powinna przechodzić funkcja, należy użyć bloku try/catch:

```
try
{
    // Kod, który w przypadku niepowodzenia zgłasza wyjątek
}
catch ( ... )
{
    // Miejsce, w którym obsługiwany jest wyjątek
    // (tj. miejsce, do którego przechodzi funkcja)
}
```

Każda funkcja w bloku try może zgłosić wyjątek, który zostanie obsłużony w bloku catch. Może istnieć wiele różnych rodzajów wyjątków, każdy należący do innej klasy, co umożliwia pisanie wielu bloków catch, z których każdy obsługuje specyficzny rodzaj błędu. Jeśli utworzysz taki blok catch(...) jak w przykładzie powyżej, wówczas każdy wyjątek nieobsłużony w innym, bardziej wyspecjalizowanym bloku catch zostanie obsłużony w tym właśnie ogólnym bloku. Możesz go traktować jak blok catch-wszystko. Dany wyjątek zostanie obsłużony w pierwszym bloku catch, który sobie z nim poradzi, jeśli zatem chcesz mieć blok catch wyłącujący pozostałe wyjątki, powinieneś go umieścić na samym końcu, po wszystkich wcześniejszych blokach:

```
try
{
    // Kod, który w przypadku niepowodzenia zgłasza wyjątek
}
catch ( const WyjatekNieOdnalezionoPliku& w )
{
    // Obsługa błędu spowodowanego nieodnalezieniem pliku
}
catch ( const WyjatekBrakMiejscaNaDysku& w )
{
    // Obsługa błędu spowodowanego brakiem miejsca na dysku
}
catch ( ... )
{
    // Miejsce, w którym obsługiwane są wszystkie pozostałe wyjątki
    // (tj. miejsce, do którego w takich przypadkach przechodzi funkcja)
}
```

## Zwalnianie zasobów po wystąpieniu wyjątku

Jeśli wywołasz funkcję, która zgłasza wyjątek, nie będziesz musiał go wyłapywać — wyjątek sam wydostanie się z Twojej funkcji i być może znajdzie blok catch w funkcji wyższego poziomu. Takie działanie kodu jest jak najbardziej prawidłowe, jeśli tylko nie musisz niczego robić

w odpowiedzi na wystąpienie wyjątku. Tak naprawdę bardzo często *nie* będziesz musiał niczego robić, ponieważ przy zakończeniu działania funkcji spowodowanym wyjątkiem wywoływanie są destruktory wszystkich obiektów lokalnych. Oto przykład:

```
int wywołajZawodnaFunkcje ()
{
    const string wart( "abc" );
    // Wywołaj kod, który zgłasza wyjątek
    zawodnaFunkcja();
}

int main ()
{
    try
    {
        wywołajZawodnaFunkcje();
    }
    catch ( ... )
    {
        // Obsługa błędu
    }
}
```

Jeśli w powyższym kodzie zawodnaFunkcja zgłosi wyjątek, łańcuch wart utworzony w funkcji wywołajZawodnaFunkcje zostanie po zgłoszeniu wyjątku zniszczyony, zwalniając wszystkie zasoby zaalokowane w celu jego zapamiętania. Taki proces nazywany jest **odwijaniem stosu** — każda ramka stosu, która nie przechwyci wyjątku, jest oczyszczana (czy też odwijana) przez wywołanie destruktora każdego obiektu w tej ramce. Pamiętaj, że obiekt ma swój domyślny destruktor, który po nim posprząta, nawet jeśli dla tego obiektu nie zdefiniujesz jawnego destruktora.

## Ręczne czyszczenie zasobów w bloku catch

Czasami po zgłoszeniu wyjątku będziesz musiał ręcznie oczyścić niektóre zasoby. W większości przypadków powinieneś postarać się utworzyć obiekt czyszczący dany zasób, ale jeśli nie masz takiej możliwości, zawsze będziesz mógł złapać wyjątek, przeprowadzić czyszczenie, a następnie ponownie zgłosić wyjątek. Oto przykład:

```
int wywołajZawodnaFunkcje ()
{
    const int* wart = new int;
    // Wywołaj kod, który zgłasza wyjątek
    try
    {
        zawodnaFunkcja();
    }
    catch ( ... )
    {
        delete wart;
        // Zwróć uwagę na użycie funkcji throw — ponownie zgłaszamy wyjątek
        throw;
    }
    delete wart; // Zauważ, że także i tu musimy umieścić funkcję delete.
    // Blok catch nie wykona się, jeśli nie wystąpi wyjątek.
    // Jedyny sposób na zagwarantowanie, że kod zawsze zostanie
    // uruchomiony, polega na wywołaniu destruktora lokalnego obiektu
}
```

```
int main ()
{
    try
    {
        wywolajZawodnaFunkcje();
    }
    catch ( ... )
    {
        // Obsługa błędu
    }
}
```

## Zgłaszanie wyjątków

Do tej pory całkiem sporo dowiedziałeś się na temat wyłapywania i obsługiwanego wyjątków, ale jak utworzyć wyjątek i go zgłosić? Tworzenie klasy wyjątku nie jest niczym szczególnym — jest to całkiem zwykła klasa. Umieszczasz w niej wszystkie pola, które według Ciebie są ważne, oraz metody dostępowe odczytujące informacje na temat wyjątku. Typowa klasa wyjątku będzie mieć mniej więcej taki interfejs:

```
class Wyjatek
{
public:
    virtual ~Wyjatek () = 0;
    virtual int pobierzKodBledu () = 0;
    virtual string pobierzOpisBledu () = 0;
};
```

Każdy określony rodzaj błędu będzie dziedziczyć po klasie Wyjatek oraz implementować poniższe metody wirtualne:

```
class WyjatekNieOdnalezionoPliku : public Wyjatek
{
public:
    WyjatekNieOdnalezionoPliku (int kod_bledu, const string& szczegoly)
        : _kod_bledu( kod_bledu )
        , _szczegoly( szczegoly )
    {}

    virtual ~WyjatekNieOdnalezionoPliku ()
    {}

    virtual int pobierzKodBledu ()
    {
        return _kod_bledu;
    }

    virtual string pobierzOpisBledu ()
    {
        return _szczegoly;
    }

private:
    int _kod_bledu;
    string _szczegoly;
};
```

Teraz będziesz mógł zgłaszać wyjątki w taki sam sposób, jak byś konstruował instancję klasy:

```
throw WyjatekNieOdnalezionoPliku ( 1, "Nie odnaleziono pliku" );
```

Jedna z zalet dziedziczenia wszystkich wyjątków po wspólnej klasie bazowej polega na tym, że wyjątki mogą być łapanie przez nadklasę. Możesz na przykład napisać:

```
catch (const Wyjatek& w )
{
}
```

co spowoduje złapanie dowolnego błędu dziedziczącego po klasie `Wyjatek`. Stosowanie staranne zdefiniowanej hierarchii wyjątków umożliwia pisanie kodu, który w jednym bloku `catch` potrafi obsługiwać różne błędy. Przykładowo wszystkie błędy wejścia i wyjścia mogą dziedziczyć po klasie `WyjatkiWeWY`, umożliwiając tym samym obsłużenie każdego wyjątku we/wy jakby to był ten sam błąd, chociaż w pewnych przypadkach — gdy różne podklasy będą wymagać różnej obsługi, kod nadal będzie potrafił wyłapywać specyficzne podklasy klasy `WyjatkiWeWY`.

Standardową nadklassą wyjątków w bibliotece standardowej C++ jest `std::exception`. Mając taką klasę macierzystą, nie musisz definiować własnej hierarchii wyjątków. Jeśli jednak korzystasz z biblioteki standardowej, użycie `std::exception` jako wspólnej klasy bazowej będzie mieć sens, gdyż dzięki niej będziesz mógł wyłapywać w swoim programie wszystkie wyjątki, zarówno te ze standardowej biblioteki, jak i własne.

## Specyfikacja wyjątków

Możesz zatem zgłosić wyjątek, kiedy natrafisz na błąd, oraz złapać wyjątek, jeśli wiesz, że funkcja może zawieść. Tylko skąd w ogóle będziesz wiedzieć, że funkcja potrafi zgłosić błąd? W C++ służy do tego **specyfikacja wyjątków** określająca wyjątki, które mogą zostać zgłoszone przez funkcję. Specyfikacja ta jest listą wyjątków (także pustą), która znajduje się na końcu deklaracji oraz definicji funkcji.

W pliku nagłówkowym:

```
void mozeZawiesc () throw (WyjatekNieOdnalezionoPliku);
void nieMozeZawiesc () throw ();
```

W pliku `.cpp`:

```
void mozeZawiesc () throw (WyjatekNieOdnalezionoPliku)
{
    throw FileNotFoundException();
}

void nieMozeZawiesc () throw ()
```

Problem ze specyfikacjami wyjątków jest taki, że **nie są one sprawdzane podczas komilacji**, a tylko w czasie działania programu. Co gorsza, jeśli funkcja zgłosi nieoczekiwany wyjątek, program może po prostu nagle zakończyć działanie. Oznacza to, że tak naprawdę nie powinieneś oczekiwania, iż specyfikacje wyjątków będą wiarygodne, ale za to możesz spodziewać się, że doprowadzą one do awarii Twojego programu. Niektóre narzędzia, takie jak na przykład PC-Lint (<http://www.gimpel.com/html/pcl.htm>), zapewniają sprawdzanie wyjątków w czasie komilacji i radzą sobie z wieloma problemami powodowanymi przez specyfikacje wyjątków.

W nowym standardzie C++, C++11 postawiono pod pregierezem całą specyfikację wyjątków, co oznacza, że w przyszłości prawdopodobnie przestaną one być częścią tego języka<sup>1</sup>.

W rezultacie tego wszystkiego musisz zaufać, że autor funkcji prawidłowo udokumentował wyjątki, które może ona zgłaszać, a jeżeli sam tworzysz funkcję zgłaszającą wyjątek, powinieneś go rzetельnie opisać.

## Korzyści płynące z wyjątków

Dwie główne zalety wynikające z implementacji wyjątków polegają na uproszczeniu logiki odpowiedzialnej za obsługę błędów dzięki umieszczeniu jej w jednym bloku catch, przez co nie ma potrzeby przeprowadzania wielu testów warunkowych z kodem zwrotnym, oraz ulepszeniu raportowania błędów przez udostępnienie informacji, które niosą ze sobą więcej treści niż sam tylko kod błędu.

Pierwsza zaleta, jaką jest możliwość obsługiowania błędów w jednym bloku catch, umożliwia przekształcenie takiego kodu:

```
if ( funkcja1() == ERROR )
{
    // Obsługa błędu
}
if ( funkcja2() == ERROR )
{
    // Obsługa błędu
}
if ( funkcja3() == ERROR )
{
    // Obsługa błędu
}
```

w następujący:

```
try
{
    funkcja1();
    funkcja2();
    funkcja3();
}
catch ( const Wyjatek& w )
{
    // Obsługa błędu
}
```

Cały kod zajmujący się obsługą błędów znajduje się w jednym miejscu, a główny przypadek użycia jest bardzo łatwy do prześledzenia.

Druga zaleta, polegająca na udostępnianiu dodatkowych informacji w przypadku wystąpienia błędu, również jest bardzo pożyteczna. Mając do dyspozycji wyłącznie kod błędu, dostaniesz, no właśnie... kod błędu. Użycie wyjątków pozwala, aby każdy błąd udostępniał dodatkowe informacje na swój temat. Na przykład wyjątek NieOdnalezionoPliku mógłby podawać nazwę pliku.

---

<sup>1</sup> W specyfikacji pozostawiono możliwość określenia, że funkcja z całą pewnością nie zgłasza wyjątków, co w pewnych sytuacjach może polepszyć wydajność kodu.

# Nieprawidłowe użycie wyjątków

Chociaż wyjątki są wspaniałym narzędziem służącym do raportowania błędów, mogą być nadużywane ze względu na to, że dają możliwość natychmiastowego powracania z funkcji do obiektu na stosie, który ją wywołał. Zazwyczaj nie powinieneś korzystać z wyjątków w celu obsługiwania spodziewanych sytuacji, które nie są błędami. W teorii mógłbyś użyć wyjątku na przykład po to, aby przekazać wynik działania funkcji, zamiast zwracać jej wartość. Takie rozwiązywanie byłoby jednak zarówno znacznie wolniejsze (gdyż obsługa wyjątku wymaga czasu), jak i dezorientujące osobę czytającą kod. Jak już mogłeś zobaczyć wcześniej, stosowanie wyjątków w celu raportowania błędów upraszcza główną logikę funkcji, jeśli jednak zaczniesz używać wyjątków w celu realizowania głównej logiki, utracisz tę prostotę.

Zobaczmy, jak mógłbyś przepisać kod, który w celu obsługi głównych przypadków użycia korzysta z wyjątków, w taki sposób, aby się tych wyjątków pozbyć. W tym celu posłużymy się przykładowym fragmentem kodu parsera. Parser to program, który wczytuje dobrze zdefiniowany kod, taki jak HTML, po czym interpretuje jego strukturę. Bardzo często parsery są wyposażone w funkcje analizujące poszczególne elementy kodu. W przypadku kodu HTML mogą to być funkcje interpretujące na przykład łącza albo tabele.

Jedna z możliwości zaimplementowania parsera może polegać na udostępnianiu raportów informujących, czy funkcje analizujLacze oraz analizujTabele były w stanie zinterpretować fragment tekstu, i zgłoszeniu wyjątku, jeśli interpretacja się nie powiodła:

```
try
{
    analizujLacze();
    return;
}
catch ( const WyjatekAnalizy& w )
{
    // To nie jest łącze — wypróbuj następny typ
}
try
{
    analizujTabele();
    return;
}
catch ( const WyjatekAnalizy& w )
{
    // To nie jest tabela — wypróbuj następny typ
}
```

Problem polega na tym, że jeśli dany fragment tekstu nie jest łączem ani tabelą, nie mamy do czynienia z błędem — jest to zwykły fragment kodu. O wiele lepiej będzie napisać parser następująco:

```
if ( spodziewaneLacze() )
{
    analizujLacze();
}
else if ( spodziewanaTabela() )
{
    analizujTabele();
}
```

Ponieważ HTML zazwyczaj podpowiada za pomocą kilku liter, jaki będzie kolejny element na stronie, w prosty sposób możesz napisać metody sprawdzające, czy następną częścią dokumentu jest łącze, czy tabela, dzięki czemu zamiast skomplikowanej obsługi wyjątków wystarczą proste instrukcje if.

## Podsumowanie informacji o wyjątkach

Wyjątki udostępniają czytelny sposób na raportowanie błędów bez konieczności zaśmiecania kodu szczegółową logiką ich obsługi. Dzięki odwijaniu stosu i destruktorom czyszczącym po obiektach wyjątki umożliwiają pisanie kodu, który zamiast złożonej obsługi błędów realizuje główną logikę algorytmu.

Zgłaszcenie wyjątków ma wpływ na wydajność kodu, w związku z czym powinieneś z nich korzystać, gdy pojawi się błąd, i nie stosować ich w celu sprawowania kontroli nad wykonywaniem się algorytmu. Twój parser mógłby zgłaszać wyjątki, jeśli na przykład napotka znaki, które są nieprawidłowe. Nie powinien tego robić w sytuacji, gdy wczyta tekst dopuszczalny w danym formacie pliku. Dzięki temu jasne jest, które okoliczności rzeczywiście powodują błędy. Takie rozwiązanie gwarantuje również najlepszą wydajność kodu, gdyż wyjątki są obsługiwane tylko w tych rzadkich sytuacjach, kiedy pojawia się prawdziwy problem. Wyjątek niemal zawsze powoduje przerwanie biegu algorytmu, w związku z czym nie ma przeszkód, aby jego obsługa była wolniejsza niż zwykłe.

Podczas prac nad wieloma rzeczywistymi programami implementacja obsługi błędów zajmuje zasadniczą część czasu poświęconego na opracowanie danej aplikacji. W związku z tym, w miarę wykraczania poza omówione w tej książce podstawy, coraz częściej będziesz mieć do czynienia z tym zagadnieniem.

## Końcowe przemyślenia

---

Zdobyłeś już sporo wiedzy na temat C++, ale Twoja podróż nie dobiegła końca. Tak naprawdę znajdujesz się dopiero na początku trwającego całe życie procesu nauki programowania. Teraz masz do dyspozycji narzędzia służące do pisania wielu ciekawych i złożonych aplikacji. Następnym etapem jest przejście do tworzenia programów — budowania złożonych systemów i praktycznej implementacji algorytmów oraz struktur danych. Programowanie nie sprawdza się jednak wyłącznie do kwestii używanego języka; należy zadawać sobie pytania o to, jak projektować programy i algorytmy, jak opracowywać interfejsy użytkownika, z jakich bibliotek korzystać, jak organizować zespoły programistów, a nawet czym się zająć w pierwszej kolejności. Innymi słowy, istotną rolę odgrywa **inżynieria oprogramowania**. W książce tej rzecz jasna zaledwie muśnięto niektóre z tych zagadnień, ale są to tematy stanowiące odrębną całość, których nie można potraktować zbyt lekko.

Podobnie nauka języka obcego nie sprawdza się wyłącznie do poznania podstaw jego gramatyki oraz składni. Nie uda Ci się bezpośrednio przejść od zapoznania się z językiem angielskim do napisania w nim wspaniałej powieści — podobnie jak w przypadku C++ nie przejdziesz od pisania prostych programów do tworzenia systemów operacyjnych. Ważne jednak jest, że zdobyłeś podstawy umożliwiające poznawanie koncepcji oraz idei niezbędnych do podjęcia następnego kroku. Oto kilka sugestii na temat tego, czym mógłbyś się zająć w następnej kolejności<sup>1</sup>:

1. Czytaj książki o inżynierii oprogramowania oraz projektowaniu algorytmów. Książki takie jak *Perełki oprogramowania* Jona Bentleya (Helion, Gliwice 2011) w miły sposób wprowadzą Cię w niektóre aspekty programowania, niemające związku z konkretnym językiem, takie jak podstawowa analiza algorytmów, projektowanie i szacowanie.
2. Pisz programy. Rozpocznij od naśladowania istniejącego oprogramowania, pisz klony istniejących narzędzi, poznając potrzebne Ci w tym celu biblioteki. Zaangażuj się — poszukaj stażu albo rozpocznij pracę nad projektem otwartego źródła. Im więcej kodu napiszesz, tym więcej napiszesz złego kodu, ale tylko pisząc zły kod, nauczysz się w końcu pisać kod poprawny.
3. Czytaj o innych dyscyplinach, nie tylko o samym programowaniu. Czytaj o testowaniu oprogramowania, o zarządzaniu projektami i produktami, ucz się marketingu. Im więcej będziesz wiedzieć o całym procesie opracowywania oprogramowania, tym bardziej zwięksysz szanse na realizację swoich zawodowych ambicji — jeżeli chcesz zostać kompetentnym programistą, architektem albo szefem.

<sup>1</sup> Niektóre z tych sugestii zostały zainspirowane znakomitym esejem Petera Norviga *Teach Yourself Programming in Ten Years* (*Naucz się programować w dziesięć lat*), do przeczytania na stronie <http://norvig.com/21-days.html>.

4. Poszukaj innych programistów, pracuj z nimi i ucz się od nich. Jest to jedna z zalet podjęcia studiów na uczelni lub stażu.
5. Znajdź mentora, który ma już za sobą ścieżkę podobną do Twojej. Słowa zapisane na papierze nie dadzą odpowiedzi na pytania, których nie przewidział autor. Kontakt z kimś podobnym do Ciebie pozwoli Ci pokonać wiele przeszkód. Okaż mu szacunek, ale nie obawiaj się zadawać pytań ani powiedzieć, że czegoś nie wiesz. Lekkie zawiadomienie zawsze jest niezłą okazją do nauki!
6. Ciesz się programowaniem. Jeśli pisanie programów nie daje Ci radości, prawdopodobnie nie będziesz chciał programować zawodowo na pełnym etacie. Nie zajmuj się nudnymi zagadnieniami, które sprawiają, że masz dość pisania programów.

Właśnie zakończyłeś lekturę tej książki, ale także rozpoczęłeś swoją karierę. Powodzenia!

## Rozwiązywanie testu z rozdziału 2.

1. Jaka wartość jest zwracana do systemu operacyjnego po poprawnym zakończeniu działania programu?  
A. -1  
B. 1  
**C. 0**  
D. Programy nie zwracają wartości.
2. Jaka jest jedyna funkcja, którą musi zawierać każdy program napisany w C++?  
A. start ()  
B. system ()  
**C. main ()**  
D. program ()
3. W jaki sposób oznaczany jest początek i koniec bloku kodu?  
**A. { i }**  
B. -> i <-  
C. BEGIN i END  
D. ( i )
4. Jaki znak kończy większość wierszy kodu w C++?  
A. .  
**B. ;**  
C. :  
D. '
5. Który z poniższych zapisów stanowi poprawny komentarz?  
A. /\* Komentarz \*/  
B. \*\* Komentarz \*\*  
**C. /\* Komentarz \*/**  
D. { Komentarz }
6. Który plik nagłówkowy jest potrzebny, aby uzyskać dostęp do instrukcji cout?  
A. stream  
B. Żaden, instrukcja cout jest dostępna domyślnie.

**C. iostream**

D. using namespace std;

## Rozwiążanie testu z rozdziału 3.

1. Jakiego typu zmiennej powinieneś użyć, kiedy chcesz zapisać taką liczbę jak 3,1415?
  - A. int
  - B. char
  - C. double**
  - D. string
2. Który z poniższych zapisów przedstawia poprawny operator służący do porównywania dwóch zmiennych?
  - A. :=
  - B. =
  - C. equal
  - D. ==**
3. W jaki sposób można uzyskać dostęp do danych typu string?
  - A. Typ string jest wbudowany w język, tak więc nie trzeba nic robić.
  - B. Ponieważ typ string jest używany podczas operacji wejścia-wyjścia, należy dołączyć plik nagłówkowy iostream.
  - C. Należy dołączyć plik nagłówkowy string.**
  - D. C++ nie obsługuje łańcuchów tekstowych
4. Który z poniższych typów nie jest poprawnym typem zmiennej?
  - A. double
  - B. real**
  - C. int
  - D. char
5. W jaki sposób można wczytać cały wiersz wprowadzony przez użytkownika?
  - A. Za pomocą cin>>.
  - B. Za pomocą readln.
  - C. Za pomocą getline.**
  - D. Nie da się tego zrobić w prosty sposób.
6. Co zostanie wyświetlone na ekranie w wyniku wykonania następującej instrukcji w C++:  
cout << 1234/2000?
  - A. 0**
  - B. 0,617
  - C. W przybliżeniu 0,617, ale dokładny wynik nie może być zapisany w liczbie zmiennoprzecinkowej.
  - D. To zależy od typów znajdujących się z obu stron równania.
7. Dlaczego w C++ jest potrzebny typ char, skoro istnieją już typy całkowite?
  - A. Ponieważ znaki i liczby całkowite są zupełnie różnymi rodzajami danych; pierwsze z nich to litery, a drugie to liczby.
  - B. Ze względu na wsteczną zgodność z C.

- C. Aby łatwiej było wczytywać oraz wyświetlać znaki zamiast liczb, ponieważ znaki są w rzeczywistości przechowywane jako liczby.
- D. Ze względu na wsparcie wielojęzykowości, aby możliwa była obsługa takich języków jak chiński albo japoński, w których występuje wiele znaków.

## Rozwiązańie testu z rozdziału 4.

1. Które z poniższych wyrażeń są prawdziwe?
  - A. 1
  - B. 66
  - C. 0,1
  - D. -1
2. Który z poniższych operatorów jest w C++ boolowskim ORAZ?
  - A. &
  - B. &&**
  - C. |
  - D. ||
3. Jaką wartość ma wyrażenie ! ( true && ! ( false || true ) )?
  - A. Prawda.**
  - B. Fałsz.
4. Który z poniższych zapisów instrukcji if ma prawidłową składnię?
  - A. if wyrażenie
  - B. if { wyrażenie }
  - C. if ( wyrażenie )**
  - D. wyrażenie if

## Rozwiązańie testu z rozdziału 5.

1. Jaka będzie ostateczna wartość zmiennej x po uruchomieniu kodu x; for( x=0; x<10; x++ ) {}?
  - A. 10**
  - B. 9
  - C. 0
  - D. 1
2. Kiedy zostanie wykonany blok następujący po instrukcji while ( x<100 )?
  - A. Kiedy x będzie mniejsze od 100.
  - B. Kiedy x będzie większe od 100.**
  - C. Kiedy x będzie równe 100.
  - D. Kiedy zechce.
3. Która z poniższych instrukcji nie tworzy struktury pętli?
  - A. for
  - B. do-while

- C. while  
**D. repeat until**
- 4.** Ile razy na pewno wykona się pętla do-while?
- A. 0  
 B. Nieskończanie wiele razy.  
**C. 1**  
 D. To zależy.

## Rozwiązywanie testu z rozdziału 6.

- 1.** Który z poniższych zapisów nie tworzy poprawnego prototypu funkcji?
- A. int funk(char x, char y);  
**B. double funk(char x)**  
 C. void funk();  
 D. char x();
- 2.** Który z poniższych typów zwraca funkcja o prototypie int funk(char x, double v, float t);?
- A. char  
**B. int**  
 C. float  
 D. double
- 3.** Który z poniższych zapisów przedstawia poprawne wywołanie funkcji (przy założeniu, że funkcja ta istnieje)?
- A. funk;  
 B. funk x, y;  
**C. funk();**  
 D. int funk();
- 4.** Który z poniższych zapisów przedstawia kompletną funkcję?
- A. int funk();  
**B. int funk(int x) {return x=x+1;}**  
 C. void funk(int) {cout<<"Witaj"}  
 D. void funk(x) {cout<<"Witaj";}

## Rozwiązywanie testu z rozdziału 7.

- 1.** Co następuje po instrukcji case?
- A. :**  
 B. ;  
 C. -  
 D. Nowy wiersz.
- 2.** Co jest potrzebne, aby kod nie przebiegał przez kolejne bloki case?
- A. end;  
**B. break;**

- C. stop;
- D. Średnik.

**3.** Jakie słowo kluczowe obsługuje niespodziewane przypadki?

- A. all
- B. contingency
- C. default**
- D. other

**4.** Jaki będzie wynik wykonania poniższego kodu?

```
int x = 0;
switch( x )
{
    case 1: cout << "Jeden";
    case 0: cout << "Zero";
    case 2: cout << "Witaj Świecie";
}
```

- A. Jeden
- B. Zero
- C. Witaj Świecie
- D. ZeroWitaj Świecie**

## Rozwiązywanie testu z rozdziału 8.

**1.** Co się stanie, jeśli przed wywołaniem funkcji rand nie uruchomisz funkcji srand?

- A. Wywołanie funkcji rand nie powiedzie się.
- B. Funkcja rand zawsze będzie zwracać 0.
- C. Podczas każdego uruchomienia programu funkcja rand będzie zwracać tę samą sekwencję liczb.**
- D. Nic się nie stanie.

**2.** Dlaczego funkcję srand wywołujesz z bieżącym czasem?

- A. Aby zagwarantować, że program zawsze będzie działać tak samo.
- B. Aby przy każdym uruchomieniu programu wygenerować nowe liczby losowe.**
- C. Aby zagwarantować, że komputer wygeneruje rzeczywiste liczby losowe.
- D. To się robi automatycznie; funkcję srand należy wywołać tylko wtedy, gdy za każdym razem chcesz nadawać ziarnu tę samą wartość.

**3.** Jaki zakres wartości zwraca funkcja rand?

- A. Jaki chcemy.
- B. Od 0 do 1000.
- C. Od 0 do RAND\_MAX.**
- D. Od 1 do RAND\_MAX.

**4.** Jaki wynik zwróci wyrażenie  $11 \% 3$ ?

- A. 33
- B. 3
- C. 8
- D. 2**

- 5.** Kiedy i ile razy powinieneś wywołać funkcję `srand`?
- Za każdym razem, gdy potrzebujesz liczby losowej.
  - Nigdy, to tylko taki ozdobnik w Windows.
  - C. Raz, na początku programu.**
  - Sporadycznie — aby poprawić losowość wyników, kiedy funkcja `rand` była używana już przez jakiś czas.

## Rozwiążanie testu z rozdziału 10.

- Który z poniższych zapisów poprawnie definiuje tablicę?
  - E. `int jakas_tablica[ 10 ];`**
  - `F. int jakas_tablica;`
  - `G. jakas_tablica{ 10 };`
  - `H. array jakas_tablica[ 10 ];`
- Jaki jest indeks ostatniego elementu tablicy liczącej 29 elementów?
  - 29
  - B. 28**
  - 0
  - D. Zdefiniowany przez programistę.
- Który z poniższych zapisów definiuje tablicę dwuwymiarową?
  - `array jakas_tablica[ 20 ][ 20 ];`
  - B. `int jakas_tablica[ 20 ][ 20 ];`**
  - `C. int jakas_tablica[ 20, 20 ];`
  - `D. char jakas_tablica[ 20 ];`
- Który z poniższych zapisów poprawnie odczyta siódmy element tablicy `foo` liczącej 100 elementów?
  - A. `foo[ 6 ];`**
  - `B. foo[ 7 ];`
  - `C. foo( 7 );`
  - `D. foo;`
- Który z poniższych zapisów poprawnie deklaruje funkcję, której argumentem jest tablica dwuwymiarowa?
  - `A. int funk ( int x[][] );`
  - `B. int funk ( int x[ 10 ][] );`
  - `C. int funk ( int x[] );`
  - D. `int funk ( int x[][ 10 ] );`**

## Rozwiążanie testu z rozdziału 11.

- Który z poniższych zapisów umożliwia dostęp do zmiennej `b` w strukturze `b`?
  - `b->zmn;`
  - B. `b.zmn;`**
  - `C. b-zmn;`
  - `D. b>zmn;`

- 2.** Który z następujących zapisów poprawnie definiuje strukturę?
- A. struct { int a; }
  - B. struct struktura\_a {int a};
  - C. struct struktura\_a int a;
  - D. struct struktura\_a {int a;};**
- 3.** Który z następujących zapisów deklaruje zmienną struktury typu foo o nazwie moje\_foo?
- A. moje\_foo as struct foo;
  - B. foo moje\_foo;**
  - C. moje\_foo;
  - D. int moje\_foo;
- 4.** Jaka będzie ostateczna wartość, którą wyświetli poniższy kod?

```
#include <iostream>

using namespace std;

struct MojaStruktura
{
    int x;
};

void aktualizujStrukture (MojaStruktura moja_struktura)
{
    moja_struktura.x = 10;
}

int main ()
{
    MojaStruktura moja_struktura;
    moja_struktura.x = 5;
    aktualizujStrukture( moja_struktura );
    cout << moja_struktura.x << '\n';
}
```

- A. 5**
- B. 10
- C. Ten kod się nie skompiluje.

## Rozwiążanie testu z rozdziału 12.

- 1.** Która z poniższych sytuacji NIE jest dobrym powodem do zastosowania wskaźnika?
- A. Chcesz pozwolić funkcji, aby modyfikowała argument, który zostanie do niej przekazany.
  - B. Chcesz zaoszczędzić pamięć i uniknąć kopiowania dużej zmiennej.
  - C. Chcesz mieć możliwość uzyskania pamięci od systemu operacyjnego.
  - D. Chcesz mieć możliwość szybszego odczytywania zmiennych.**
- 2.** Co przechowuje wskaźnik?
- A. Nazwę innej zmiennej.
  - B. Wartość całkowitą.**

- C. Adres pamięci innej zmiennej.  
**D. Adres pamięci, ale niekoniecznie innej zmiennej.**
- 3.** Skąd możesz uzyskać więcej pamięci podczas działania programu?
- Nie można uzyskać więcej pamięci.
  - Ze stosu.
- C. Z dostępnej pamięci.**
- D. Poprzez zadeklarowanie kolejnej zmiennej.
- 4.** Co niekorzystnego może się wydarzyć podczas stosowania wskaźników?
- Możesz sięgnąć do pamięci, z której nie powinieneś korzystać, co doprowadzi do awarii programu.
  - Możesz sięgnąć pod niewłaściwy adres pamięci, co spowoduje zniszczenie danych.
  - Możesz zapomnieć o zwróceniu pamięci do systemu operacyjnego, przez co w programie skończy się pamięć.
- D. Każde zdarzenie z powyższych.**
- 5.** Skąd bierze się pamięć używana przez zwykłą zmienną zadeklarowaną w funkcji?
- Z dostępnej pamięci.
  - Ze stosu.
  - Zwykłe zmienne nie używają pamięci.
  - Z samego programu — to dlatego pliki EXE są tak duże!
- 6.** Co powinieneś zrobić po zaalokowaniu pamięci?
- Nic, ta pamięć już na zawsze będzie Twoja.
  - Zwrócić ją do systemu operacyjnego, kiedy nie będzie już potrzebna.
  - Nadać wskazywanej zmiennej wartość 0.
  - Zapisać we wskaźniku wartość 0.

## Rozwiążanie testu z rozdziału 13.

- 1.** Który z poniższych zapisów poprawnie definiuje wskaźnik?
- int x;
  - int &x;
  - wsk x;
  - int \*x;**
- 2.** Który z poniższych zapisów podaje adres pamięci zmiennej całkowitej a?
- \*a;
  - a;
  - &a;**
  - address( a );
- 3.** Który z poniższych zapisów podaje adres pamięci zmiennej wskazywanej przez wskaźnik w\_a?
- w\_a;
  - \*w\_a;
  - &w\_a;**
  - address( w\_a );

- 4.** Który z poniższych zapisów podaje wartość przechowywaną pod adresem wskazywanym przez wskaźnik `w_a`?
- A. `w_a`;
  - B. `wart( w_a )`;
  - C. `*w_a`;**
  - D. `&w_a`;
- 5.** Który z poniższych zapisów poprawnie definiuje referencję?
- A. `int *w_int;`
  - B. `int &moja_ref;`
  - C. `int &moja_ref = & moja_oryg_wart;`
  - D. `int &moja_ref = moja_oryg_wart;`**
- 6.** Która z poniższych sytuacji nie jest dobrą okazją do użycia referencji?
- A. Przechowywanie adresu, który został dynamicznie zaallokowany w wolnej pamięci.**
  - B. Uniknięcie kopiowania dużej wartości podczas przekazywania jej do funkcji.
  - C. Wymuszenie, aby parametr przekazywany do funkcji nigdy nie przyjmował wartości NULL.
  - D. Zezwolenie funkcji na dostęp do oryginalnej zmiennej, która jest do niej przekazywana, bez konieczności korzystania ze wskaźników

## Rozwiązywanie testu z rozdziału 14.

- 1.** Które z poniższych słów kluczowych jest właściwym poleceniem służącym do alokowania pamięci w C++?
- A. new**
  - B. `malloc`
  - C. `create`
  - D. `value`
- 2.** Które z poniższych słów kluczowych jest właściwym poleceniem służącym do zwalniania zaallokowanej pamięci w C++?<sup>2</sup>
- A. `free`
  - B. delete**
  - C. `clear`
  - D. `remove`
- 3.** Które z poniższych stwierdzeń jest prawdziwe?
- A. Tablice i wskaźniki są tym samym.
  - B. Tablic nie można przypisywać do wskaźników.
  - C. Wskaźniki można traktować jak tablice, chociaż nimi nie są.**
  - D. Ze wskaźników można korzystać tak samo jak z tablic, ale nie można ich alokować tak jak tablic.

---

<sup>2</sup> No dobrze — masz także rację, jeśli na dwa pierwsze pytania odpowiedziałeś `malloc` i `free`; funkcje te pochodzą z języka C, ale to oznacza, że nie przeczytałeś tego rozdziału!

- 4.** Jakie będą ostateczne wartości `x`, `w_int` oraz `w_w_int` po wykonaniu poniższego kodu? Zwróć uwagę, że kompilator nie przyjmie bezpośrednio tego kodu, ponieważ liczby całkowite i wskaźniki są różnych typów. Aby jednak stwierdzić, co dzieje się z takimi wielokrotnymi wskaźnikami, ćwiczenie to można wykonać na kartce papieru.

```
int x = 0;
int *w_int = & x;
int **w_w_int = & w_int;
*w_int = 12;
**w_w_int = 25;
w_int = 12;
*w_w_int = 3;
w_w_int = 27;
```

- A. `x = 0, w_w_int = 27, w_int = 12`
  - B. `x = 25, w_w_int = 27, w_int = 12`
  - C. `x = 25, w_w_int = 27, w_int = 3`**
  - D. `x = 3, w_w_int = 27, w_int = 12`
- 5.** W jaki sposób można zaznaczyć, że wskaźnik nie wskazuje ważnej wartości?
- A. Nadać mu wartość ujemną.
  - B. Nadać mu wartość NULL.**
  - C. Zwolnić pamięć skojarzoną z tym wskaźnikiem.
  - D. Nadać mu wartość false.

## Rozwiązanie testu z rozdziału 15.

- 1.** Na czym polega przewaga list powiązanych nad tablicami?
  - A. Każdy element w liście powiązanej zajmuje mniej miejsca.
  - B. Listy powiązane można dynamicznie powiększać w celu umieszczania w nich nowych elementów bez konieczności kopiowania elementów istniejących.**
  - C. W listach powiązanych można szybciej odszukać konkretny element niż w tablicach.
  - D. Elementami przechowywanymi w listach powiązanych mogą być struktury.
- 2.** Które z poniższych stwierdzeń jest prawdziwe?
  - A. Nie ma żadnych powodów, aby w ogóle korzystać z tablic.
  - B. Listy powiązane i tablice charakteryzują się taką samą wydajnością.
  - C. Zarówno listy powiązane, jak i tablice dzięki indeksom gwarantują stały czas dostępu do swoich elementów.
  - D. Dodanie nowego elementu w środku listy powiązanej jest szybsze niż dodanie nowego elementu w środku tablicy.**
- 3.** Kiedy zwykle będziesz używać list powiązanych?
  - A. Kiedy trzeba przechować tylko jeden element.
  - B. Kiedy liczba elementów, które należy przechować, jest znana podczas komplikacji.
  - C. Kiedy trzeba dynamicznie dodawać i usuwać elementy.**
  - D. Kiedy potrzebny jest błyskawiczny dostęp do dowolnego elementu posortowanej listy bez potrzeby przeprowadzania jakichkolwiek iteracji.

- 4.** Dlaczego można zadeklarować listę powiązaną zawierającą referencję do elementu typu tej listy (struct Wezel { Wezel\* w\_nastepny; };)?
- A. To nie jest dozwolone.
  - B. Ponieważ kompilator wie, że tak naprawdę elementy autoreferencyjne nie potrzebują pamięci.
  - C. Ponieważ typ jest wskaźnikiem, potrzebne jest tylko tyle miejsca, ile zajmuje jeden wskaźnik. Pamięć niezbędna do realizacji kolejnego węzła zostanie zaalokowana później.**
  - D. Takie rozwiązanie jest dopuszczalne, dopóki zmienna w\_nastepny nie wskazuje następnej struktury.
- 5.** Dlaczego ważne jest, aby na końcu listy powiązanej znajdowała się wartość NULL?
- A. Wartość ta wskazuje koniec listy i nie pozwala, aby kod uzyskał dostęp do niezainicjalizowanej pamięci.**
  - B. Wartość NULL przeciwdziała powstawaniu w liście serii odwołań cyklicznych.
  - C. W ten sposób wspomaga się debugowanie kodu — kiedy spróbujesz wyjść poza listę, w programie nastąpi awaria.
  - D. Jeśli nie zapiszemy wartości NULL, lista ze względu na autoreferencję będzie potrzebować nieskończonej pamięci.
- 6.** W czym podobne są tablice i listy powiązane?
- A. Zarówno jedne, jak i drugie umożliwiają szybkie dodawanie nowych elementów w środku swoich struktur.
  - B. Zarówno jedne, jak i drugie umożliwiają sekwencyjne zapisywanie oraz odczytywanie danych.**
  - C. Zarówno tablice, jak i listy powiązane mogą z łatwością zwiększać swoje rozmiary poprzez dodawanie do nich nowych elementów.
  - D. Zarówno jedne, jak i drugie gwarantują szybki dostęp do wszystkich swoich elementów.

## Rozwiążanie testu z rozdziału 16.

- 1.** Kiedy masz do czynienia z rekurencją ogonową?
- A. Kiedy wołasz swojego psa.
  - B. Kiedy funkcja wywołuje sama siebie.
  - C. Kiedy ostatnią czynnością, jaką wykonuje funkcja rekurencyjna przed powrotem, jest wywołanie siebie samej.**
  - D. Kiedy algorytm rekurencyjny można napisać w postaci pętli.
- 2.** Kiedy skorzystasz z rekurencji?
- A. Kiedy algorytmu nie można rozpisać w postaci pętli.
  - B. Kiedy naturalniejsze jest wyrażenie algorytmu pod postacią podproblemów niż w formie pętli.**
  - C. Nigdy, rekurencja jest zbyt trudna  $\Theta$ .
  - D. Podczas pracy z tablicami i listami łączonymi.

- 3.** Jakie elementy są wymagane w algorytmie rekurencyjnym?
- A. Przypadek bazowy i wywołanie rekurencyjne.
  - B. Przypadek bazowy i jakiś sposób na rozbicie problemu na jego mniejsze wersje.
  - C. Jakiś sposób na połączenie mniejszych wersji problemu.
  - D. Każde z powyższych.**
- 4.** Co może się wydarzyć, jeśli przypadek bazowy jest niekompletny?
- A. Algorytm może zbyt wcześnie zakończyć działanie.
  - B. Kompilator wykryje błąd i zgłosi swoje zastrzeżenia.
  - C. To nie stanowi problemu.
  - D. Może wystąpić przepelenie stosu.**

## Rozwiążanie testu z rozdziału 17.

- 1.** Jaka jest podstawowa zaleta drzew binarnych?
- A. Korzystają ze wskaźników.
  - B. Mogą przechowywać dowolne ilości danych.
  - C. Umożliwiają szybkie odszukiwanie informacji.**
  - D. Usuwanie z nich danych jest łatwe.
- 2.** Kiedy użyjesz raczej listy powiązanej niż drzewa binarnego?
- A. Kiedy musisz przechowywać dane w taki sposób, który umożliwia ich szybkie wyszukiwanie.
  - B. Kiedy musisz mieć możliwość odczytywania danych w kolejności posortowanej.
  - C. Kiedy musisz mieć możliwość szybkiego dodawania danych na początku lub końcu, ale nigdy w środku.**
  - D. Kiedy nie musisz zwalniać pamięci, z której korzystasz.
- 3.** Które z poniższych stwierdzeń jest prawdziwe?
- A. Kolejność, w jakiej dodajesz dane do drzewa binarnego, może zmienić jego strukturę.**
  - B. Aby zagwarantować najlepszą strukturę drzewa binarnego, należy wstawać w nie elementy w kolejności posortowanej.
  - C. Szukanie elementów w liście powiązanej będzie szybsze niż w przypadku drzewa binarnego, jeśli elementy drzewa binarnego są powstawane w dowolnej kolejności.
  - D. Drzewa binarne nigdy nie da się przekształcić w taki sposób, aby miało strukturę listy powiązanej.
- 4.** Które z poniższych zdań wyjaśnia, dlaczego szukanie węzłów w drzewie binarnym przebiega szybko?
- A. Wcale tak nie jest. Dwa wskaźniki oznaczają, że przeglądanie drzewa wymaga więcej pracy.
  - B. Przejście w dół o każdy poziom drzewa powoduje, że liczba węzłów, które pozostały do sprawdzenia, jest redukowana mniej więcej o połowę.**
  - C. Drzewa binarne tak naprawdę wcale nie są lepsze od list powiązanych.
  - D. Wywołania rekurencyjne w drzewach binarnych są szybsze niż pętle w listach powiązanych.

## Rozwiążanie testu z rozdziału 18.

- 1.** Kiedy wskazane jest użycie wektora?
  - A. Kiedy trzeba przechowywać związek zachodzący między kluczem a wartością.
  - B. Kiedy podczas wymieniania zbioru elementów potrzebna jest maksymalna wydajność.
  - C. Kiedy nie musisz przejmować się szczegółami aktualizowania swojej struktury danych.**
  - D. Tak jak garnitur w przypadku rozmowy o pracę, wektor jest zawsze odpowiedni.
- 2.** W jaki sposób można jednocześnie usunąć wszystkie elementy z mapy?
  - A. Zamienić element na łańcuch pusty.
  - B. Wywołać metodę erase.
  - C. Wywołać metodę empty.
  - D. Wywołać metodę clear.**
- 3.** Kiedy powinieneś implementować własne struktury danych?
  - A. Kiedy potrzebujesz czegoś naprawdę szybkiego.
  - B. Kiedy potrzebujesz czegoś solidnego.
  - C. Kiedy trzeba skorzystać z przewagi, jaką dają nieprzetworzone struktury danych, na przykład w przypadku tworzenia drzewa wyrażeń arytmetycznych.**
  - D. Tak naprawdę nie trzeba implementować własnych struktur danych, chyba że z jakichś powodów chcemy to zrobić.
- 4.** Który z poniższych zapisów poprawnie definiuje iterator, którego można użyć z wektorem całkowitym `vector<int>?`
  - A. `iterator<int> itr;`
  - B. `vector::iterator itr;`
  - C. `vector<int>::iterator itr;`**
  - D. `vector<int>::iterator<int> itr;`
- 5.** Który z poniższych zapisów udostępnia klucz elementu, przy którym w danej chwili znajduje się iterator na mapie?
  - A. `itr.first`
  - B. `itr->first`**
  - C. `itr->klucz`
  - D. `itr.klucz`
- 6.** Jak podczas korzystania z iteratora sprawdzisz, czy dotarłeś do ostatniej iteracji?
  - A. Porównam iterator z wartością `NULL`.
  - B. Porównam iterator z wartością wywołania metody `end()` dla kontenera, po którym iteruję.**
  - C. Porównam iterator z wartością `0`.
  - D. Porównam iterator z wartością wywołania metody `begin()` dla kontenera, po którym iteruję.

## Rozwiążanie testu z rozdziału 19.

- 1.** Który z poniższych zapisów prezentuje poprawny kod?
  - A. const int& x;
  - B. const int x = 3; int \*p\_int = & x;
  - C. const int x = 12; const int \*p\_int = & x;**
  - D. int x = 3; const int y = x; int& z = y;
- 2.** Która z poniższych sygnatur funkcji umożliwia skompilowanie kodu `const int x = 3; fun( x );?`
  - A. void fun (int x);
  - B. void fun (int& x);
  - C. void fun (const int& x);
  - D. Poprawne są odpowiedzi A i C.**
- 3.** Jaki jest najlepszy sposób na stwierdzenie, czy szukanie łańcucha tekstowego powiodło się?
  - A. Porównać zwróconą pozycję z 0.
  - B. Porównać zwróconą pozycję z -1.
  - C. Porównać zwróconą pozycję ze string::npos.**
  - D. Sprawdzić, czy zwrócona pozycja jest większa od długości łańcucha tekstowego.
- 4.** W jaki sposób można utworzyć iterator dla stałego kontenera STL?
  - A. Zadeklarować iterator jako const.
  - B. Zamiast iteratora należy skorzystać z indeksów w celu przejścia kontenera w pętli.
  - C. Za pomocą zapisu const\_iterator.**
  - D. Zadeklarować typy szablonowe jako const.

## Rozwiążanie testu z rozdziału 21.

- 1.** Który z poniższych etapów nie stanowi części procesu budowy oprogramowania w języku C++?
  - A. Konsolidacja.
  - B. Kompilacja.
  - C. Przetwarzanie wstępne.
  - D. Przetwarzanie końcowe.**
- 2.** Kiedy wystąpi błąd związany z niezdefiniowaną funkcją?
  - A. Na etapie konsolidacji.**
  - B. Na etapie komplikacji.
  - C. Na początku działania programu.
  - D. Podczas wywołania funkcji.
- 3.** Co się może stać, gdy kilkakrotnie dołączysz plik nagłówkowy?
  - A. Pojawią się błędy ostrzegające o powielonych deklaracjach.**
  - B. Nic, pliki nagłówkowe zawsze są wczytywane tylko raz.
  - C. To zależy od implementacji pliku nagłówkowego.
  - D. Pliki nagłówkowe można dołączać jednorazowo tylko po jednym pliku źródłowym, tak więc nie stanowi to żadnego problemu.

- 4.** Jaka jest zaleta przeprowadzania komplikacji i konsolidacji na odrębnych etapach?
- A. Żadna. Takie rozwiązanie prowadzi do pomyłek i prawdopodobnie spowalnia cały proces, ponieważ trzeba uruchamiać jednocześnie kilka programów.
  - B. Takie rozwiązanie ułatwia diagnozowanie błędów, ponieważ wiesz, czy dany problem wykrył konsolidator czy kompilator.
  - C. Takie rozwiązanie umożliwia komplikowanie wyłącznie zmodyfikowanych plików, co skraca czas komplikacji i konsolidacji.
  - D. Takie rozwiązanie umożliwia komplikowanie wyłącznie zmodyfikowanych plików, co skraca czas komplikacji.**

## Rozwiązanie testu z rozdziału 22.

- 1.** Jaka jest zaleta stosowania funkcji w porównaniu z bezpośrednim dostępem do danych?
- A. Kompilator może optymalizować funkcję w celu zapewnienia szybszego dostępu do danych.
  - B. Funkcja może ukrywać swoją implementację przed obiektami ją wywołującymi, co upraszcza modyfikowanie tych obiektów.**
  - C. Użycie funkcji to jedyny sposób na użycie tej samej struktury danych w wielu różnych plikach z kodem źródłowym.
  - D. Nie ma żadnych zalet.
- 2.** Kiedy kod należy przenieść do wspólnej funkcji?
- A. Kiedy tylko zachodzi potrzeba jej wywołania.
  - B. Kiedy zacząłeś wywoływać ten sam kod w kilku różnych miejscach programu.**
  - C. Kiedy kompilator zaczyna narzekać, że funkcje są zbyt duże, aby je skompilować.
  - D. Prawidłowe są odpowiedzi B i C.
- 3.** Dlaczego ukrywa się sposób reprezentacji struktury danych?
- A. Aby łatwiej ją było zastąpić inną strukturą.
  - B. Aby kod, który korzysta ze struktury danych, był łatwiejszy do zrozumienia.
  - C. Aby prosto było użycie struktury danych w nowych fragmentach kodu.
  - D. Wszystkie powyższe odpowiedzi są prawidłowe.**

## Rozwiązanie testu z rozdziału 23.

- 1.** Dlaczego warto korzystać z metod zamiast bezpośrednio z pól struktury?
- A. Ponieważ metody są łatwiejsze do czytania.
  - B. Ponieważ metody są szybsze.
  - C. Nie należy tego robić. Zawsze należy korzystać bezpośrednio z pól.
  - D. Dzięki temu można modyfikować sposób reprezentacji danych.**
- 2.** Który z poniższych zapisów definiuje metodę skojarzoną ze strukturą struct MojaStrukt  
(int funk(); );?
- A. int funk() { return 1; }
  - B. MojaStrukt::int funk() { return 1; }

- C. int MojaStrukt::funk(){ return 1; }**
- D. int MojaStrukt funk(){ return 1; }
- 3.** Dlaczego umieściłbyś definicję metody wewnątrz klasy?
- Żeby użytkownicy klasy wiedzieli, jak ona działa.
  - Ponieważ takie rozwiązanie zawsze przyspiesza działanie kodu.
  - Nie należy tego robić! W ten sposób wyciekną wszystkie szczegóły implementacji tej metody.**
  - Nie należy tego robić, ponieważ spowolni to działanie programu.

## Rozwiązań testu z rozdziału 24.

- 1.** Dlaczego miałbyś korzystać z danych prywatnych?
- Aby je zabezpieczyć przed hakerami.
  - Aby inni programiści nie mogli się do nich dobrać.
  - Aby było jasne, które dane powinny być użyte wyłącznie w celu implementacji klasy.**
  - Nie należy tego robić, bo pisanie programu stanie się trudniejsze.
- 2.** Czym różni się klasa od struktury?
- Niczym.
  - W klasie domyślnie wszystko jest publiczne.
  - W klasie domyślnie wszystko jest prywatne.**
  - Klasa pozwala określić, które pola mają być publiczne, a które prywatne, natomiast w strukturze jest to niemożliwe.
- 3.** Co należy zrobić z polami danych w klasie?
- Zdefiniować je domyślnie jako publiczne.
  - Zdefiniować je domyślnie jako prywatne, ale przekształcić w publiczne, jeśli zajdzie taka potrzeba.
  - Nigdy nie definiować ich jako publiczne.**
  - Klasy zazwyczaj nie zawierają danych, ale jeśli już tak się stanie, można z nimi robić, co się chce.
- 4.** W jaki sposób można zdecydować, czy metoda powinna być publiczna?
- Metoda nigdy nie powinna być publiczna.
  - Metoda zawsze powinna być publiczna.
  - Metoda powinna być publiczna, jeśli jest potrzebna do korzystania z głównych funkcjonalności klasy; w przeciwnym przypadku powinna być prywatna.**
  - Metoda powinna być publiczna, jeśli jest prawdopodobne, że ktoś będzie chciał z niej korzystać.

## Rozwiązań testu z rozdziału 25.

- 1.** Kiedy dla klasy należy napisać konstruktor?
- Zawsze, bez konstruktora nie można korzystać z klasy.
  - Kiedy wystąpi potrzeba zainicjalizowania klasy wartościami, które nie są domyślne.**

- C. Nigdy, kompilator zawsze zapewnia konstruktor.
- D. Tylko wtedy, gdy potrzebny jest także destruktor.

**2.** Jaki związek istnieje między destruktorem a operatorem przypisania?

- A. Nie ma związku.
- B. Destruktor klasy jest wywoływany przed uruchomieniem operatora przypisania.
- C. Operator przypisania musi określić, jaka pamięć ma zostać zniszczona przez destruktor.
- D. Operator przypisania musi zagwarantować, że wywołanie destruktora zarówno klasy kopiowanej, jak i nowej będzie bezpieczne.**

**3.** Kiedy należy użyć listy inicjalizacyjnej?

- A. Kiedy konstruktory powinny być maksymalnie wydajne i aby uniknąć konstruowania pustych obiektów.
- B. Kiedy inicjalizowane są wartości stałe.
- C. Kiedy dla pola klasy należy uruchomić niedomyślny konstruktor.
- D. W każdym z powyższych przypadków.**

**4.** Jaka funkcja zostanie uruchomiona w drugim wierszu poniższego kodu?

```
string str1;  
string str2 = str1;
```

- A. Konstruktor dla str2 oraz operator przypisania dla str1.
- B. Konstruktor dla str2 oraz operator przypisania dla str2.
- C. Konstruktor kopiący dla str2.**
- D. Operator przypisania dla str2.

**5.** Jakie funkcje (i w jakiej kolejności) zostaną wywołane dla poniższego kodu?

```
{  
    string str1;  
    string str2;  
}
```

- A. Konstruktor dla str1 i konstruktor dla str2.
- B. Destruktor dla str1 i konstruktor dla str2.
- C. Konstruktor dla str1, konstruktor dla str2, destruktor dla str1 i destruktor dla str2.
- D. Konstruktor dla str1, konstruktor dla str2, destruktor dla str2 i destruktor dla str1.**

**1.** Jeśli wiesz, że klasa ma konstruktor kopiący, który nie jest domyślny, jakie zdanie na temat operatora przypisania powinno być prawdziwe?

- A. Operator przypisania powinien być domyślny.
- B. Operator przypisania nie powinien być domyślny.
- C. Operator przypisania powinien być zadeklarowany, ale nie zaimplementowany.
- D. Poprawne są odpowiedzi B i C.**

(Poza tym powinien być prywatny, dzięki czemu kompilator wcześniej wyłąpie problem.)

# Rozwiążanie testu z rozdziału 26.

- 1.** Kiedy uruchamiany jest destruktor nadklasy?
- Tylko wtedy, gdy obiekt jest niszczony przez wywołanie instrukcji `delete` ze wskaźnikiem nadklasy.
  - Przed wywołaniem destruktora podklasy.
  - C. Po wywołaniu destruktora podklasy.**
  - Razem z wywołaniem destruktora podklasy.

- 2.** Mając do czynienia z poniższą hierarchią klas, co musiałbyś zrobić w konstruktorze klasy Kot?

```
class Ssak {
public:
    Ssak (const string& nazwa_gatunku);
};

class Kot : public Ssak
{
public:
    Kot();
};
```

- Nic szczególnego.
- B. Skorzystać z listy inicjalizacyjnej w celu wywołania konstruktora klasy Ssak z argumentem kot.**
- Wywołać konstruktor klasy Ssak z poziomu konstruktora klasy Kot z argumentem kot.
- Usunąć konstruktor klasy Kot i skorzystać z konstruktora domyślnego, który rozwiąże ten problem za Ciebie.

- 3.** Co jest nie tak z poniższą definicją klasy?

```
class Nazywalny
{
    virtual string pobierzNazwe();
}
```

- Nie deklaruje ona metody `pobierzNazwe` jako publicznej.
  - Nie ma wirtualnego destruktora.
  - Nie zawiera implementacji metody `pobierzNazwe`, ale przy tym nie deklaruje jej jako czysto wirtualną.
  - D. Wszystkie odpowiedzi są prawidłowe.**
- 4.** Kiedy deklarujesz metodę wirtualną w klasie interfejsowej, co powinna zrobić funkcja, aby mogła korzystać z metody interfejsu w celu wywołania metody w podklasie?
- A. Pobrać interfejs jako wskaźnik (albo referencję).**
  - Nic, może po prostu skopiować obiekt.
  - Aby wywołać metodę, powinna znać nazwę podklasy.
  - O co chodzi? Co to jest metoda wirtualna?
- 5.** W jaki sposób dziedziczenie polepsza wielokrotne użycie kodu?
- Pozwalając, aby kod podklasy dziedziczył metody z nadklasy.
  - Pozwalając, aby nadklaś implementowała metody wirtualne dla swojej podklasy.

- C. Pozwalając na pisanie kodu oczekującego interfejsu zamiast konkretnej klasy, co umożliwia nowym klasom implementowanie tego interfejsu i korzystanie z istniejącego kodu.
- D. Pozwalając, aby nowe klasy dziedziczyły cechy konkretnej klasy, które można wykorzystać w metodach wirtualnych.
6. Które z poniższych zdań poprawnie opisuje poziomy dostępności klas?
- A. Podklasa ma dostęp wyłącznie do publicznych metod i danych swojej klasy nadrzęennej.
  - B. Podklasa ma dostęp do prywatnych metod i danych swojej klasy nadrzęennej.
  - C. Podklasa ma dostęp wyłącznie do chronionych metod i danych swojej klasy nadrzęennej.
  - D. Podklasa ma dostęp do chronionych i publicznych metod oraz danych swojej klasy nadrzęennej.**

## Rozwiążanie testu z rozdziału 27.

1. Kiedy powinieneś korzystać z dyrektywy namespace?
  - A. We wszystkich plikach nagłówkowych, zaraz po include.
  - B. Nigdy, dyrektywa ta jest niebezpieczna.
  - C. Na początku każdego pliku cpp, gdy nie występuje konflikt przestrzeni nazw.**
  - D. Bezpośrednio przed użyciem zmiennej z danej przestrzeni nazw.
2. Do czego potrzebne są przestrzenie nazw?
  - A. Aby twórcy kompilatorów mieli ciekawą pracę.
  - B. Aby zapewnić lepszą hermetyzację w kodzie.
  - C. Aby zapobiec konfliktom nazw w dużych bazach kodu.**
  - D. Aby ułatwić zrozumienie, do czego służy dana klasa.
3. Kiedy należy umieszczać kod w przestrzeni nazw?
  - A. Zawsze.
  - B. Kiedy piszesz program, który jest na tyle duży, że składa się z więcej niż kilkudziesięciu plików.
  - C. Kiedy piszesz bibliotekę, która będzie udostępniana innym programistom.
  - D. Prawidłowe są odpowiedzi B i C.**
4. Dlaczego nie należy umieszczać deklaracji using namespace w pliku nagłówkowym?
  - A. Ponieważ jest to nieprawidłowe.
  - B. Nie ma powodu, aby tego nie robić. Deklaracja using jest ważna wyłącznie w pliku nagłówkowym.
  - C. Ponieważ wymusza to stosowanie deklaracji using na każdym, kto dołącza plik nagłówkowy, nawet wtedy, gdy jej zamieszczenie spowoduje powstanie konfliktów nazw.**
  - D. Może to wywołać konflikt, jeśli wiele plików nagłówkowych zawiera deklaracje using.

## Rozwiązanie testu z rozdziału 28.

- 1.** Z którego typu można korzystać, aby czytać z pliku?
  - A. ifstream
  - B. ofstream
  - C. fstream
  - D. Prawdziwe są odpowiedzi A i C.**
- 2.** Które z poniższych stwierdzeń jest prawdziwe?
  - A. Pliki tekstowe zajmują mniej miejsca niż pliki binarne.
  - B. W plikach binarnych łatwiej znajdować błędy.
  - C. Pliki binarne zajmują mniej miejsca niż pliki tekstowe.**
  - D. Pliki tekstowe są zbyt wolne, aby korzystać z nich w prawdziwych programach.
- 3.** Dlaczego podczas zapisywania do pliku binarnego nie można przekazywać wskaźnika do obiektu typu string?
  - A. Do metody write zawsze należy przekazywać char\*.
  - B. Obiekt typu string nie może być przechowywany w pamięci.
  - C. Nie znamy układu obiektu typu string — może on zawierać wskaźniki, które zostały zachowane w pliku.**
  - D. Łąncuchy tekstowe są zbyt duże i muszą być zapisywane po kawałku.
- 4.** Które z poniższych stwierdzeń na temat formatu pliku jest prawdziwe?
  - A. Formaty plików, tak samo jak wszystkie inne dane wejściowe, można łatwo zmieniać.
  - B. Zmiana formatu pliku wymaga przemyślenia, co się stanie, gdy stara wersja programu spróbuje odczytać nową wersję pliku.
  - C. Projektowanie formatu pliku wymaga przemyślenia, co się stanie, gdy nowa wersja programu spróbuje odczytać starą wersję pliku.
  - D. Prawdziwe są odpowiedzi B i C**

## Rozwiązanie testu z rozdziału 29.

- 1.** Kiedy należy korzystać z szablonów?
  - A. Kiedy chcesz zaoszczędzić na czasie.
  - B. Kiedy chcesz mieć szybciej działający kod.
  - C. Kiedy dla różnych typów musisz wiele razy pisać ten sam kod.**
  - D. Kiedy chcesz mieć pewność, że później będziesz mógł korzystać z tego samego kodu.
- 2.** Kiedy w parametrze szablonu należy podać typ?
  - A. Zawsze.
  - B. Tylko podczas deklarowania instancji szablonu klasy.
  - C. Tylko wtedy, gdy nie można przeprowadzić inferencji typu.
  - D. W przypadku szablonów funkcji — jeśli nie można przeprowadzić inferencji typu; w przypadku szablonów klas — zawsze.**

- 3.** Skąd kompilator wie, że dany parametr szablonu może być użyty z danym szablonem?
- A. Implementuje specyficzny interfejs języka C++.
  - B. Podczas deklarowania szablonu należy określić ograniczenia.
  - C. Próbuje użyć parametru szablonu; jeśli jego typ obsługuje wszystkie wymagane operacje, przyjmuje go.**
  - D. Podczas deklarowania szablonu należy podać listę wszystkich poprawnych typów.
- 4.** Czym różni się umieszczenie szablonu klasy w pliku nagłówkowym od umieszczenia w takim pliku zwykłej klasy?
- A. Niczym.
  - B. Zwykła klasa nie może mieć zdefiniowanej w pliku nagłówkowym ani jednej swojej metody.
  - C. Szablon klasy musi mieć zdefiniowane w pliku nagłówkowym wszystkie swoje metody.**
  - D. Szablon klasy nie potrzebuje odpowiadającego mu pliku .cpp, natomiast klasa musi mieć taki plik.
- 5.** Kiedy należy przekształcić funkcję w szablon funkcji?
- A. Na samym początku. Nigdy nie wiadomo, czy tej samej logiki nie trzeba będzie użyć w odniesieniu do różnych typów, w związku z czym zawsze lepiej jest tworzyć szablony metod.
  - B. Tylko wtedy, gdy nie można rzutować na typy, których akurat wymaga funkcja.
  - C. Jeśli właśnie napisałś niemal taką samą logikę, jaką już masz w kodzie, z tym że dla typu, który ma właściwości podobne do typu użytego we wcześniej istniejącej funkcji.**
  - D. Zawsze wtedy, gdy dwie funkcje robią „prawie” to samo i można dostosować ich implementację za pomocą kilku dodatkowych parametrów logicznych.
- 6.** Kiedy dowiesz się o większości błędów popełnionych w kodzie szablonu?
- A. Jak tylko skompilujesz szablon.
  - B. Na etapie konsolidacji.
  - C. Podczas uruchamiania programu.
  - D. Podczas pierwszej komplikacji kodu tworzącego instancję szablonu.**

# Skorowidz

---

## A

abstrakcja funkcyjna, 267–268  
Access Violation, 240  
adres pamięci, 132, 155  
aktualizacja zmiennej, 71  
algorytm, 103, 107–109  
    mieszający, 206  
alokacja pamięci, 135  
argumenty  
    funkcji, 41, 83  
    w wierszu poleceń, 331  
arytmetyka wskaźników, 156  
automatyczne wcinanie tekstu, 17

## B

bajt, 55  
bit, 55  
błędy  
    kompilacji, 42  
    kompilatora, 22  
    podczas deklarowania zmiennych, 51–53

## C

C++ a C, 15–16  
ciało instrukcji, 59  
Code::Blocks, 17, 23  
cykl życia klasy, 283  
czysty tekst, 17

## D

dane  
    wejściowe, 47  
    wielowymiarowe, 115

debugger, 231  
debugowanie, 246  
    awarii, 239–241  
    modyfikowanie zmiennych, 245  
przepełnienia stosu, 186  
w Code::Blocks  
    wlamanie się do działającego programu, 243  
    wstrzymywanie działania programu, 233–239  
    zawieszenie programu, 242–245  
definicja metody, 274  
definiowanie funkcji, 86  
deklarowanie  
    funkcji, 87  
    instancji klasy, 279  
    klasy, 278  
    metody, 273  
    wskaźnika, 139–140  
    zmiennych, 47  
dekrementacja, 50  
delimiter, 222  
dereferencja wskaźnika, 141  
destruktor, 288–291  
    czyszczący po obiektach, 365  
dodłączanie do pliku, 326  
drzewo  
    binarne, 192–193  
    zastosowanie, 206  
    puste, 193  
    zrównoważone, 192  
dynamiczna alokacja pamięci, 151  
dyrektywy preprocesora, 252  
dziedziczenie, 300–306  
    a konstruowanie obiektów i ich niszczenie, 304–306  
dzielenie  
kodu z podklasami, 309  
programu na wiele plików, 254–259

**E**

edytor, 17  
endl, 41  
enkapsulacja, 281  
EOF, 325

**F**

fixup, 254  
format pliku, 323  
formatowanie danych wyjściowych, 357–362  
    za pomocą iomanip, 357–362

funkcja, 39, 81  
    atoi, 332  
    getline, 54–55, 221–222  
    main, 39  
    rand, 98  
    setf, 359  
    setfill, 358  
    setiosflags, 362  
    setprecision, 361  
    setw, 357  
    srand, 98, 100

**G**

g++, 33  
getter, 280  
głowa listy, 168

**H**

hermetyzacja, 281

**I**

implementacja  
    drzew binarnych, 194  
    polimorfizmu, 311–313  
include guard, 259  
indeks tablicy, 114  
inferencja typów, 345  
inicjalizacja  
    składowych klasy, 286–287  
    zmiennej, 52  
inicjowanie przy pozyskaniu zasobu, 291  
inkrementacja, 50  
input file stream, 321

instancja wersji szablonu, 343  
instrukcja

    break, 73–74  
    cin, 48  
    cin.get (), 41, 48  
    cin.ignore (), 48  
    continue, 74  
    cout, 41  
    delete, 289–290  
    else, 62  
    else-if, 62–63  
    include, 40  
    new, 151, 153  
    switch case, 91–94  
    using, 319  
    using namespace std, 40, 317, 319  
    warunkowa if, 59–60

interfejs użytkownika, 279  
iterator, 215–217  
iterowanie, 169

**J**

język programowania, 15

**K**

kacze typowanie, 345–346  
katalog roboczy programu, 322  
klasa, 277–281, 283  
    czysto wirtualna, 301  
    fstream, 328  
    string, 53, 223  
kod źródłowy, 16  
komentarze, 42, 268  
komentowanie kodu, 268  
komentowanie programów, 42  
komplikacja, 251, 253  
komplilator, 17  
komunikaty o błędach w szablonach, 350–353  
konfiguracja środowiska programistycznego  
    Linux, 33–38  
    Macintosh, 23–33  
    Windows, 17–28  
konsolidacja, 253–254  
konstruktor, 283–285  
    klasy ofstream, 327  
    kopiący, 295–296

konstruowanie obiektu, 283–285  
 kontrakt funkcji, 180  
 kontrolowanie przebiegu pętli, 73–74  
 kopiowanie klas, 291–292  
 korzystanie ze zmiennych, 48  
 koszt tworzenia drzew i map, 207

**L**

liczby  
 całkowite, 57  
 losowe w C++, 98–100  
 a błędy, 100  
 pseudolosowe, 97  
 zmienoprzecinkowe, 56  
 licznik pętli, 71  
 lista  
 inicjalizacyjna, 287  
 metod generowanych przez kompilator, 296  
 powiązana, 166, 170, 179  
 a tablica, 171–173  
 lukier składniowy, 156

**Ł**

łańcuchy tekstowe, 53–55, 221  
 wczytywanie, 221–223

**M**

makro, 252  
 mapa, 206–207, 218  
 mapa (STL), 214–215  
 metoda, 213, 272–275  
 chroniona, 310  
 cin, 221  
 clear, 325  
 eof, 325  
 fail, 325  
 fill, 359  
 find, 224  
 find (STL), 217  
 is\_open, 322  
 length, 223  
 prywatna, 309  
 publiczna, 309  
 push\_back, 214  
 read, 338  
 rfind, 224

seekg, 328  
 seekp, 328  
 size, 223  
 statyczna, 310–311  
 substr, 224  
 tellg, 327  
 tellp, 327  
 w STL, 215  
 wirtualna, 301, 312  
 write, 334, 336  
 modyfikatory dostępu, 278, 309

**N**

nadklasa, 300  
 nano, 35–38  
 nawiasy klamrowe, 40  
 nazwy zmiennych, 53  
 nazywanie funkcji, 89  
 niszczanie  
 drzewa, 197–199  
 obiektu, 288–291

**O**

obiekt, 279  
 obiekt cout, 41  
 obsługa argumentów liczbowych, 332  
 odczytywanie informacji z plików, 321–326  
 oddzielna kompilacja, 254  
 odpowiedzialności klasy, 280  
 odwijanie stosu, 365  
 ojciec w drzewie binarnym, 192  
 operacja dzielenia modulo, 99  
 operator  
 adresu, 140  
 boolowski, 64  
 dekrementacji, 50  
 inkrementacji, 50  
 logiczny, 64  
 kolejność wykonywania działań, 66–67  
 LUB, 65  
 łączenie wyrażeń, 66  
 negacja, 64  
 ORAZ, 65  
 odejmowania, mnożenia i dzielenia, 50  
 porównania, 49  
 kolejność wykonywania działań, 67

operator  
    przypisania, 49, 292–295  
    relacyjny, 61  
    umożliwiający dodanie dowolnej wartości  
        do zmiennej, 50  
    wstawiania, 41  
organizacja pamięci, 134  
otrzymywanie adresu zmiennej  
    za pomocą wskaźnika, 140  
output file stream, 321

## P

pamięć  
    dostępna, 134  
    komputera, 132–133  
pętla, 69  
    do-while, 72–73, 77  
    for, 70–72, 76  
    nieskończona, 69  
    wewnętrzna, 75  
    while, 69–70, 76  
    zagnieżdzona, 75–76  
    zewnętrzna, 75  
 piksel, 265  
plik  
    binarny, 332–340  
    nagłówkowy, 258–259, 321  
    obiektowy, 253  
    tekstowy, 333  
    wykonywalny, 16  
plikowe operacje wejścia-wyjścia, 321  
płytką kopią wskaźnika, 292  
pobieranie argumentów z wiersza polecień, 330–332  
poddzewa, 192  
podklasa, 300  
polimorfizm, 303, 311–313  
    a dziedziczenia obiektów, 306–308  
porównywanie łańcuchów tekstowych, 63–64  
pozycja pliku, 327–330  
praca z wieloma plikami źródłowymi  
    Code::Blocks, 260  
    g++, 261  
    Xcode, 261–262  
    w środowisku programistycznym, 260  
preprocesor, 252–253  
private, 278, 281  
programowanie, 16  
    rozwiązywanie problemów, 103–109

projektowanie  
    od dołu do góry, 107  
    od góry do dołu, 107  
protected, 309  
prototyp funkcji, 86–88  
przeciążanie funkcji, 89  
przeglądanie listy powiązanej, 169–170  
przekazywanie łańcucha przez referencję, 225–226  
przekazywanie  
    struktur, 127–129  
    tablic do funkcji, 116–117  
    wektorów do metod, 213  
przeładowywianie funkcji, 89  
przepelenie stosu, 186–187  
przestrzeń nazw, 317–320  
przesunięcie, 114  
przeszukiwanie drzewa, 196–197  
przetwarzanie wstępne, 252–253  
przycinanie, 308–309  
przypadek bazowy funkcji, 178  
public, 278

## R

RAII, 291  
ramka stosu, 183–184  
raportowanie błędów, 363–370  
referencja, 146–147  
    a dynamiczna alokacja pamięci, 152  
rekurencja, 177, 188  
    a pętle, 181–183, 188  
    a struktury danych, 179–181  
ogonowa, 182  
wzajemna, 187  
zalety i wady, 185  
rekursja, 177  
rozróżnienie wielkości liter, 52  
rzutowanie  
    reinterpret\_cast, 335  
    static\_cast, 335  
    typu zmiennej, 335

## S

separator, 222  
setter, 280  
składnia  
    C++, 39–43  
    funkcji, 81–82

struktury, 125–127  
 tablic, 113–114  
 wektora, 212  
 wskaźników, 139–140

## składowa

klasy, 278  
 prywatna, 278–279  
 publiczna, 278–279  
 skrócone wartościowanie, 65  
 słowo kluczowe

- const, 225–228
- delete, 151
- enum, 94
- false, 61
- new, 151, 285
- sizeof, 156
- template, 344
- true, 61
- typename, 344

## sortowanie

przez wstawianie, 122  
 tablic, 118–122  
 specyfikacja wyjątków, 367  
 sprawdzanie, czy wartość znajduje się w mapie, 217  
 standardowa biblioteka szablonów, Patrz STL  
 sterta, 134  
 STL, 211, 218–219, 343  
 stos, 134, 183–185  
 strażnik include, 259  
 struktura, 125, 146  
 struktura danych, 163
 

- a wskaźniki, 165

 strumień, 321  
 symbole debugowania, 232  
 synowie w drzewie binarnym, 192  
 szablony, 343, 347–350
 

- funkcji, 343–346
- klas, 346–347

**S**

średnik, 40

**T**

tabela  
 ASCII, 56  
 metod wirtualnych, 312

tablica, 113, 163
 

- a listy powiązane, 171–173
- a pamięć, 136
- a pętla, 115–116
- dwuwymiarowa, 115, 156
- wypadnięcie poza ostatni element tablicy, 118
- zastosowania, 114–115

 tworzenie
 

- instancji, 346
- listy powiązanej, 166–169
- nowych plików, 327

 typ danych
 

- ifstream, 321
- ofstream, 321, 326

 typ wyliczeniowy, 94–95
 typy zmiennych, 47

**U**

ukrywanie sposobu przechowywania danych  
 przy pomocy klas, 278  
 usuwanie węzła z drzewa, 199–206  
 użycie
 

- listy inicjalizacyjnej do pól stałych, 287
- wskaźnika, 140–143

**V**

vtable, 312

**W**

wartość NULL, 143, 152  
 warunek pętli, 71  
 wektor (STL), 212–214  
 węzły drzewa binarnego, 192  
 właściciel pamięci, 134  
 wskaźnik, 131–133, 135–136, 139–143
 

- a funkcje, 144–146
- a referencja, 147
- a tablica, 152–155
- do wskaźników, 157–160
- do wskaźników i tablic dwuwymiarowych, 159
- this, 294

 wycieki pamięci, 134  
 wyjątki, 363–370  
 wykładnik, 56  
 wykomentowanie kodu, 43

wyrażenie, 60  
fałszywe, 61  
prawdziwe, 61  
wyrównywanie, 254  
wyróżnianie składni, 17

## X

Xcode, 23  
Xcode 4, 28

## Z

zagłędzanie przestrzeni nazw, 318  
zakres zmiennej, 83  
zapisywanie plików, 326  
zapobieganie przed kopiowaniem, 296–297  
zasięg zmiennej, 83

zastosowanie funkcji, 88  
zgłaszanie wyjątków, 366–367  
ziarno, 97  
zintegrowane środowisko programistyczne, 17  
zmiana wartości zmiennych, 49  
zmienna, 47, 133  
globalna, 84–86  
lokalna, 83–84  
pętli, 71  
statyczna, 310–311  
typu bool, 61–62  
typu char, 47, 56  
typu double, 47, 56  
typu float, 55  
typu int, 47  
zwalnianie pamięci, 151



# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄZKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW  
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA  
**Helion SA**