

THINKING IN C++

EDYCJA POLSKA
BRUCE ECKEL

Helion ✓

Książka omawia zagadnienia związane z programowaniem, wyjaśnia, dlaczego są one problemami, oraz opisuje ujęcie stosowane w języku C++ przy ich rozwiązywaniu. Podręcznik poprowadzi cię krok po kroku od rozumienia C do etapu, w którym zbiór pojęć języka C++ stanie się twoim językiem ojczystym. Celem każdego rozdziału jest przybliżenie pojedynczego pojęcia lub niewielkiej grupy powiązanych ze sobą pojęć w sposób nie wymagający korzystania z żadnych dodatkowych terminów. Wprowadzenie do języka będzie odbywało się stopniowo, odzwierciedlając sposób, w jaki zazwyczaj przyswajasz sobie nowe informacje.

Tytuł oryginału: Thinking in C++

Tłumaczenie: Piotr Imiela

Authorized translation from the English language edition entitled THINKING IN C++: INTRODUCTION TO STANDARD C++, VOLUME ONE, 2nd edition by ECKEL, BRUCE, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright ©2000

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Polish language edition published by Wydawnictwo Helion.

Copyright ©2002

ISBN: 83-7197-709-3

Wydawnictwo HELION
ul. Chopina 6, 44-100 GLIWICE
tel. (prefiks-32) 231-22-19, (prefiks-32) 230-98-63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

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

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

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.

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

Printed in Poland.

Moim rodzicom, siostrze i bratu

Podziękowania

Dziękuję przede wszystkim tym, którzy za pośrednictwem Internetu przestali mi poprawki i sugestie. Byliście ogromnie pomocni w podniesieniu jakości książki — nie zdołałbym zrobić tego bez waszej pomocy. Szczególne podziękowania należą się Johnowi Cookowi.

Pomysły oraz wiedza zawarte w książce pochodzą wielu źródeł: od przyjaciół, takich jak Chuck Allison, Andrea Provaglio, Dan Saks, Scott Meyers, Charles Petzold i Michael Wilk, pionierów języka, jak Bjarne Stroustrup, Andrew Koenig i Rob Murray, członków komitetu standaryzacyjnego C++ Standards Committee, takich jak Nathan Myers (którego uwagi były szczególnie pomocne i życzliwe), Bill Ptauger, Reg Charney, Tom Peneilo, Tom Plum, Sam Druker i Uwe Steinmueller, od osób, które zabierają głos w ramach prowadzonej przez mnie grupy tematycznej C++ na Software Development Conference, a także od uczestników moich seminariów, zadających pytania, które musiałem usłyszeć, by uzczyńić materiał bardziej przejrzystym.

Ogromne podziękowania należą się Genowi Kiyooka, którego firma Digigami użyczyła mi swojego serwera internetowego.

Języka C++ nauczałem wraz z przyjacielem, Richardem Hale Shawem. Wskazówki i pomoc Richarda okazały się bardzo pomocne (podobnie jak Kim). Dziękuję również KoAnn Vikoren, Eriewi Faurotowi, Jennifer Jessup, Tarze Arrowood, Marcowi Pardiemu, Nicole Freeman, Barbarze Hanscome, Reginie Ridley, Alexowi Dunne'owi, atakże pozostałe obsadzie załodze MFI.

Szczególnie gorąco chciałem podziękować wszystkim moim nauczycielom i studentom (którzy również stali się moimi nauczycielami).

Chciałem również wyrazić głębokie uznanie i sympię za ich wysiłki moim ulubionym pisarzom: Johnowi Irvingowi, Nealowi Stephensonowi, Robertsonowi Daviesowi (będzie nam Ciębie brakować), Tomowi Robbinsowi, Williamowi Gibsonowi, Richardowi Bachowi, Carlosowi Castaniedzie oraz Gene'owi Wolfe.

Dziękuję Guido van Rossumowi za wymyślenie Pythona i bezinteresowne oddanie go światu. Swoim przyczynkiem wzbogaciłeś moje życie.

Dziękuję pracownikom Pretince Hali: Alanowi Apcie, Anie Terry, Scottowi Disanno, Toniemu Holmowi oraz mojej elektronicznej redaktorce Stephanie Enghsh. W dziale marketingu natomiast słowa wdzięczności należały Bryanowi Gambrelowi i Jennie Burger.

Sonda Donovan pomogła w produkcji płyty CD-ROM. Daniel Wil-Harris (oczywiście) przygotował projekt sitodruku, który znalazł się na płycie.

Wszystkim wspaniałym ludziom w Crested Butte dziękuję za uczynienie go magicznym miejscem, a szczególnie Alowi Smithowi (twórcy cudownego Camp4 Coffee Garden), moim sąsiadom Dave'owi i Erice, Marshy z księgarńi Heg's Place, Patowi i Johnowi z TeocaUi Tamale, Samowi z Bakery Cafe oraz Tiller — zajego pomoc w pracach nad dźwiękiem. A także wszystkim znakomitościom, które przesiadywały w Camp4, czyniąc moje poranki interesującymi.

Na liście moich przyjaciół (nie jest ona jeszcze zamknięta) znaleźli się: Zack Urlocker, Andrew Binstock, Neil Rubenking, Jiraig Brockschmidt, Steve Sinofsky, JD Hildebrandt, Brian McElhinney, Brinkley Barr, Larry O'Brien, Bill Gates z *Midnight Engineering Magazine*, Larry Constantine, Lucy Lockwood, Tom Keffer, Dan Puterman, Gene Wang, Dave Mayer, David Intersimone, Claire Sawyers, Włosi (Andrea Provaglio, Rossella Gioia, Laura Fallai, Marco i Leila Cantu, Corrado, Iisa i Christina Giustozzi), Chris i Laura Strandowie (oraz Parker), Almqistowie, Brad Jerbic, Marilyn Cvitanic, Mabry'owie, Haflingerowie, Pollockowie, Peter Vinci, Robbinsowie, Moelterowie, Dave Stoner, Laurie Adams, Cranstonowie, Larry Fogg, Mike i Karen Sequeira, Gary Entsminger i Allison Brody, Kevin, Sonda, i Ella Donovanowie, Chester i Shannon Andersenowie, Joe Lordi, Dave i Brenda Bartlettowie, Rentschlerowie, Lynn i Todd, oraz ich rodziny. No i, oczywiście, Mama i Tata.

Spis treści

Wstęp	13
Co nowego w drugim wydaniu?.....	13
Zawartość drugiego tomu książki.....	14
Skąd wziąć drugi tom książki?.....	14
Wymagania wstępne.....	14
Nauka języka C++.....	15
Cele.....	16
Zawartość rozdziałów.....	17
Ćwiczenia.....	21
Rozwiązania ćwiczeń.....	21
Kod źródłowy.....	21
Standardy języka.....	22
Obsługa języka.....	23
Błędy.....	23
Okładka.....	24
Rozdział 1. Wprowadzenie do obiektów	25
Postęp abstrakcji.....	26
Obiekt posiada interfejs.....	27
Ukryta implementacja.....	30
Wykorzystywanie istniejącej implementacji.....	31
Dziedziczenie — wykorzystywanie istniejącego interfejsu.....	32
Relacje typu „jest” i „jest podobny do”.....	35
Zastępowanie obiektów przy użyciu polimorfizmu.....	36
Tworzenie i niszczenie obiektów.....	40
Obsługa wyjątków — sposób traktowania błędów.....	41
Analiza i projektowanie.....	42
Etap 0. Przygotuj plan.....	44
Etap 1. Co tworzymy?.....	45
Etap 2. Jak to zrobimy?.....	49
Etap 3. Budujemy jądro.....	52
Etap 4. Iteracje przez przypadki użycia.....	53
Etap 5. Ewolucja.....	53
Planowanie się opłaca.....	55
Programowanie ekstremalne.....	55
Najpierw napisz testy.....	56
Programowanie w parach.....	57
Dlaczego C++ odnosi sukcesy?.....	58
Lepsze C.....	59
Zacząłeś się już uczyć.....	59

Efektywność.....	60
Systemy są łatwiejsze do opisania i do zrozumienia.....	60
Maksymalne wykorzystanie bibliotek.....	60
Wielokrotne wykorzystywanie kodu dzięki szablonom.....	61
Obsługa błędów.....	61
Programowanie na wielką skalę.....	61
Strategie przejścia.....	62
Wskazówki.....	62
Problemy z zarządzaniem.....	64
Podsumowanie.....	66
Rozdział 2. Tworzenie i używanie obiektów.....	67
Proces tłumaczenia języka.....	68
Interpretery.....	68
Kompilatory.....	68
Proces komplikacji.....	69
Narzędzia do rozłącznej komplikacji.....	71
Deklaracje i definicje.....	71
Łączenie.....	76
Używanie bibliotek.....	76
Twój pierwszy program w C++.....	78
Używanie klasy strumieni wejścia-wyjścia.....	78
Przestrzenie nazw.....	79
Podstawy struktury programu.....	80
„Witaj, świecie!”.....	81
Uruchamianie kompilatora.....	82
Więcej o strumieniach wejścia-wyjścia.....	82
Łączenie tablic znakowych.....	83
Odczytywanie wejścia.....	84
Wywoływanie innych programów.....	84
Wprowadzenie do łańcuchów.....	85
Odczytywanie i zapisywanie plików.....	86
Wprowadzenie do wektorów.....	88
Podsumowanie.....	92
Ćwiczenia.....	93
Rozdział 3. Język C w C++.....	95
Tworzenie funkcji.....	95
Wartości zwracane przez funkcje.....	97
Używanie bibliotek funkcji języka C.....	98
Tworzenie własnych bibliotek pomocą programu zarządzającego bibliotekami.....	99
Sterowanie wykonywaniem programu.....	99
Prawda i fałsz.....	99
if-else.....	100
while.....	101
do-while.....	101
for.....	102
Słowa kluczowe break i continue.....	103
switch.....	104
Użycie i nadużywanie instrukcji goto.....	105
Rekurencja.....	106
Wprowadzenie do operatorów.....	107
Priorytety.....	107
Automatyczna inkrementacja i dekrementacja.....	108

Wprowadzenie do typów danych.....	108
Podstawowe typy wbudowane.....	109
bool, true i false.....	110
Specyfikatory.....	111
Wprowadzenie do wskaźników.....	112
Modyfikacja obiektów zewnętrznych.....	115
Wprowadzenie do referencji.....	117
Wskaźniki i referencje jako modyfikatory.....	118
Zasięg.....	120
Definiowanie zmiennych „w locie”.....	120
Specyfikacja przydziału pamięci.....	122
Zmienne globalne.....	122
Zmienne lokalne.....	124
static.....	124
extern.....	126
Stałe.....	127
volatile.....	129
Operatory i ich używanie.....	129
Przypisanie.....	130
Operatory matematyczne.....	130
Operatory relacji.....	131
Operatory logiczne.....	131
Operatory bitowe.....	132
Operatory przesunięć.....	133
Operatory jednoargumentowe.....	135
Operator trójargumentowy.....	136
Operator przecinkowy.....	137
Najczęstsze pułapki związane z używaniem operatorów.....	137
Operatory rzutowania.....	138
Jawne rzutowanie w C++.....	139
sizeof — samotny operator.....	143
Słowo kluczowe asm.....	143
Operatory dosłowne.....	144
Tworzenie typów złożonych.....	144
Nadawanie typom nowych nazw za pomocą typedef.....	144
Łączenie zmiennych w struktury.....	145
Zwiększanie przejrzystości programów za pomocą wyliczeń.....	148
Oszczędzanie pamięci za pomocą unii.....	150
Tablice.....	151
Wskazówki dotyczące uruchamiania programów.....	159
Znaczniki uruchomieniowe.....	160
Przekształcanie zmiennych i wyrażeń w łańcuchy.....	162
Makroinstrukcja assert() języka C.....	162
Adresy funkcji.....	163
Definicja wskaźnika do funkcji.....	163
Skomplikowane deklaracje i definicje.....	164
Wykorzystywanie wskaźników do funkcji.....	165
Tablice wskaźników do funkcji.....	166
Make — zarządzanie rozłączną komplikacją.....	167
Działanie programu make.....	168
Pliki makefile używane w książce.....	171
Przykładowy plik makefile.....	171
Podsumowanie.....	173
Ćwiczenia.....	173

Rozdział 4. Abstrakcja danych.....	179
Miniaturowa biblioteka w stylu C.....	180
Dynamiczny przydział pamięci.....	183
Błędne założenia.....	186
Na czym polega problem?.....	188
Podstawowy obiekt.....	188
Czym są obiekty?.....	194
Tworzenie abstrakcyjnych typów danych.....	195
Szczegóły dotyczące obiektów.....	196
Zasady używania plików nagłówkowych.....	197
Znaczenie plików nagłówkowych.....	198
Problem wielokrotnych deklaracji.....	199
Dyrektywy preprocesora <code>#define</code> , <code>#ifdef</code> i <code>#endif</code>	200
Standard plików nagłówkowych.....	201
Przestrzenie nazw w plikach nagłówkowych.....	202
Wykorzystywanie plików nagłówkowych w projektach.....	202
Zagnieżdżone struktury.....	202
Zasięg globalny.....	206
Podsumowanie.....	206
Ćwiczenia.....	207
Rozdział 5. Ukrywanie implementacji.....	211
Określanie ograniczeń.....	211
Kontrola dostępu w C++.....	212
Specyfikator <code>protected</code>	214
Przyjaciele.....	214
Zagnieżdżeni przyjaciele.....	216
Czy jest to „czyste”?.....	218
Struktura pamięci obiektów.....	219
Klasy.....	219
Modyfikacja programu Stash, wykorzystująca kontrolę dostępu.....	222
Modyfikacja stosu, wykorzystująca kontrolę dostępu.....	223
Klasy-uchwyty.....	223
Ukrywanie implementacji.....	224
Ograniczanie powtórnych komplikacji.....	224
Podsumowanie.....	226
Ćwiczenia.....	227
Rozdział 6. Inicjalizacjai końcowe porządki.....	229
Konstruktor gwarantuje inicjalizację.....	230
Destruktor gwarantuje sprzątanie.....	232
Eliminacja bloku definicji.....	233
Pętle <code>for</code>	235
Przydzielanie pamięci.....	236
Klasa Stash z konstruktorami i destruktoremami.....	237
Klasa Stack z konstruktorami i destruktoremami.....	240
Inicjalizacja agregatowa.....	242
Konstruktor domyślne.....	245
Podsumowanie.....	246
Ćwiczenia.....	246
Rozdział 7. Przeciążanie nazw funkcji i argumenty domyślne.....	249
Dalsze uzupełnienia nazw.....	250
Przeciążanie na podstawie zwracanych wartości.....	251
Łączenie bezpieczne dla typów.....	252

Przykładowe przeciążenie.....	253
Unie.....	255
Argumenty domyślne.....	258
Argumenty-wypełniacze.....	259
Przeciążaniekontra argumenty domyślne.....	260
Podsumowanie.....	264
Ćwiczenia.....	265
Rozdział 8. Stałe.....	267
Podstawianie wartości.....	267
Stałe w plikach nagłówkowych.....	268
Bezpieczeństwo stałych.....	269
Agregaty.....	270
Różnice w stosunku do języka C.....	271
Wskaźniki.....	272
Wskaźniki do stałych.....	272
Stałe wskaźniki.....	273
Przypisanie a kontrola typów.....	274
Argumenty funkcji i zwracane wartości.....	275
Przekazywanie stałej przez wartość.....	275
Zwracanie stałej przez wartość.....	276
Przekazywanie i zwracanie adresów.....	279
Klasy.....	282
Stałe w klasach.....	282
Stałe o wartościach określonych podczas komplikacji, zawarte w klasach.....	285
Stałe obiekty i funkcje składowe.....	287
volatile.....	292
Podsumowanie.....	293
Ćwiczenia.....	294
Rozdział 9. Funkcje inline.....	297
Pułapki preprocesora.....	298
Makroinstrukcje a dostęp.....	300
Funkcje inline.....	301
Funkcje inline wewnętrz klas.....	302
Funkcje udostępniające.....	303
Klasy Stash i Stack z funkcjami inline.....	308
Funkcje inline a kompilator.....	311
Ograniczenia.....	312
Odwołania do przodu.....	313
Działania ukryte w konstruktorach i destruktorkach.....	313
Walka z bałaganem.....	314
Dodatkowe cechy preprocesora.....	315
Sklejanie symboli.....	316
Udoskonalona kontrola błędów.....	316
Podsumowanie.....	319
Ćwiczenia.....	320
Rozdział 10. Zarządzanie nazwami.....	323
Statyczne elementy języka C.....	323
Zmienne statyczne znajdujące się wewnątrz funkcji.....	324
Sterowanie łączeniem.....	328
Inne specyfikatory klas pamięci.....	330
Przestrzenie nazw.....	330
Tworzenie przestrzeni nazw.....	330
Używanie przestrzeni nazw.....	332
Wykorzystywanie przestrzeni nazw.....	336

Statyczne składowe w C++.....	337
Definiowanie pamięciadla statycznych danych składowych.....	337
Klasy zagnieżdżone i klasy lokalne.....	341
Statyczne funkcje składowe.....	342
Zależności przy inicjalizacjiobiektów statycznych.....	344
Jak można temu zaradzić?.....	346
Specyfikacja zmiany sposobu łączenia.....	352
Podsumowanie.....	353
Ćwiczenia.....	353
Rozdział 11. Referencjei konstruktor kopiący.	359
Wskaźniki w C++.....	359
Referencje w C++.....	360
Wykorzystanie referencji w funkcjach.....	361
Wskazówki dotyczące przekazywania argumentów.....	363
Konstruktor kopiący.....	363
Przekazywanie i zwracanie przez wartość.....	364
Konstrukcja za pomocą konstruktora kopiącego.....	369
Domyślny konstruktor kopiący.....	374
Możliwości zastąpienia konstruktora kopiącego.....	376
Wskaźniki do składowych.....	378
Funkcje.....	380
Podsumowanie.....	382
Ćwiczenia.....	383
Rozdział 12. Przeciążanie operatorów.	387
Ostrzeżenie i wyjaśnienie.....	387
Składnia.....	388
Operatory, które można przeciążać.....	389
Operatory jednoargumentowe.....	390
Operatory dwuargumentowe.....	393
Argumenty i zwracane wartości.....	402
Nietypowe operatory.....	405
Operatory, których nie można przeciążać.....	412
Operatory niebędące składowymi.....	413
Podstawowe wskazówki.....	414
Przeciążanie operacji przypisania.....	415
Zachowanie się operatora =.....	416
Automatyczna konwersja typów.....	425
Konwersja za pomocą konstruktora.....	425
Operator konwersji.....	427
Przykład konwersji typów.....	429
Pułapki automatycznej konwersji typów.....	430
Podsumowanie.....	432
Ćwiczenia.....	432
Rozdział 13. Dynamiczne tworzenie obiektów.	437
Tworzenie obiektów.....	438
Obsługa sterty w języku C.....	439
Operator new.....	440
Operator delete.....	441
Prosty przykład.....	442
Narzut menedżera pamięci.....	442

Zmiany w prezentowanych wcześniej przykładach.....	443
Usuwanie wskaźnika void* jest prawdopodobnie błędem.....	443
Odpowiedzialność za sprzątanie wskaźników.....	445
Klasa Stash przechowująca wskaźniki.....	445
Operatory new i delete dla tablic.....	450
Upodabnianie wskaźnika do tablicy.....	451
Brak pamięci.....	451
Przeciążanie operatorów new i delete.....	452
Przeciążanie globalnych operatorów new i delete.....	453
Przeciążanie operatorów new i delete w obrębie klas.....	455
Przeciążanie operatorów new i deletew stosunku do tablic.....	458
Wywołania konstruktora.....	460
Operatory umieszczania new i delete.....	461
Podsumowanie.....	463
Ćwiczenia.....	463
Rozdział 14. Dziedziczeniei kompozycja	467
Składnia kompozycji.....	468
Składnia dziedziczenia.....	469
Lista inicjatorów konstruktora.....	471
Inicjalizacja obiektów składowych.....	471
Typy wbudowane znajdujące się na liście inicjatorów.....	472
Łączenie kompozycji i dziedziczenia.....	473
Kolejność wywoływaniakonstruktorów i destruktorów.....	474
Ukrywanie nazw.....	476
Funkcje, które nie są automatycznie dziedziczzone.....	480
Dziedziczenie a statyczne funkcje składowe.....	483
Wybór między kompozycją a dziedziczeniem.....	484
Tworzenie podtypów.....	485
Dziedziczenie prywatne.....	487
Specyfikator protected.....	488
Dziedziczenie chronione.....	489
Przeciążanie operatorów a dziedziczenie.....	490
Wielokrotne dziedziczenie.....	491
Programowanie przyrostowe.....	492
Rzutowanie w góre.....	492
Dlaczego „rzutowanie w góre”?.....	494
Rzutowanie w górę a konstruktor kopiący.....	494
Kompozycja czy dziedziczenie (po raz drugi).....	497
Rzutowanie w górę wskaźników i referencji.....	498
Kryzys.....	498
Podsumowanie.....	498
Ćwiczenia.....	499
Rozdział 15. Polimorfizmi funkcje wirtualne	503
Ewolucja programistów języka C++.....	504
Rzutowanie w góre.....	504
Problem.....	506
Wiązanie wywołania funkcji.....	506
Funkcje wirtualne.....	506
Rozszerzalność.....	508
W jaki sposób język C++ realizuje późne wiązanie?.....	510
Przechowywanie informacji o typie.....	511
Obraz funkcji wirtualnych.....	512

Rzut oka pod maskę.....	514
Instalacja wskaźnika wirtualnego.....	515
Obiekty są inne.....	516
Dlaczego funkcje wirtualne?.....	517
Abstrakcyjne klasy podstawowe i funkcje czysto wirtualne.....	518
Czysto wirtualne definicje.....	522
Dziedziczenie i tablica VTABLE.....	523
Okrajanie obiektów.....	525
Przeciążanie i zasłanianie.....	527
Zmiana typu zwracanej wartości.....	529
Funkcje wirtualne a konstruktory.....	530
Kolejność wywoływania konstruktorów.....	531
Wywoływanie funkcji wirtualnych wewnątrz konstruktorów.....	532
Destruktory i wirtualne destruktory.....	533
Czysto wirtualne destruktory.....	535
Wirtualne wywołania w destruktach.....	537
Tworzenie hierarchii bazującej na obiekcie.....	538
Przeciążanie operatorów.....	541
Rzutowanie w dół.....	543
Podsumowanie.....	546
Ćwiczenia.....	546
Rozdział 16. Wprowadzenie do szablonów.....	551
Kontenery.....	551
Potrzeba istnienia kontenerów.....	553
Podstawy szablonów.....	554
Rozwiążanie z wykorzystaniem szablonów.....	556
Składnia szablonów.....	558
Definicje funkcji niebędących funkcjami inline.....	559
Klasa IntStack jako szablon.....	560
Stałe w szablonach.....	562
Klasy Stack i Stash jako szablony.....	563
Kontener wskaźników Stash, wykorzystujący szablony.....	565
Przydzielanie i odbieranie prawa własności.....	570
Przechowywanie obiektów jako wartości.....	573
Wprowadzenie do iteratorów.....	575
Klasa Stack z iteratorami.....	582
Klasa PStash z iteratorami.....	585
Dlaczego iteratory?.....	590
Szablony funkcji.....	593
Podsumowanie.....	594
Ćwiczenia.....	594
Dodatek A Styl kodowania.....	599
Dodatek B Wskazówki dla programistów.....	609
Dodatek C Zalecana literatura.....	621
Język C.....	621
Ogólnie o języku C++.....	621
Książki, które napisałem.....	622
Głębia i mroczne zaułki.....	623
Analiza i projektowanie.....	623
Skorowidz.....	627

Wstęp

Język C++, podobnie jak inne języki używane przez ludzi, umożliwia wyrażanie pojęć. Jeżeli robi to skutecznie, jako środek wyrazu będzie znacznie łatwiejszy w użyciu i bardziej elastyczny niż inne dostępne środki — w miarę jak problemy staną się coraz większe i bardziej złożone.

Nie można postrzegać C++ jedynie jako zbioru właściwości — pewne cechy nie mają sensu w oderwaniu od pozostałych. Można używać sumy poszczególnych elementów z myślą o *projektie*, a nie o zwyczajnym kodowaniu. Aby tak pojmować C++, trzeba rozumieć problemy za pomocą języka C oraz samego programowania. Książka omawia zagadnienia związane z programowaniem, wyjaśnia, dlaczego są one problemami, oraz opisuje użycie stosowane w języku C++ przy ich rozwiązywaniu. A zatem zbiór cech, który opisuję w każdym rozdziale, będzie wynikał ze sposobu postrzegania rozwiązania określonego rodzaju problemów za pomocą języka. Mam nadzieję poprowadzić cię krok po kroku od rozumienia C do etapu, w którym zbiór pojęć języka C++ stanie się twoim językiem ojczystym.

Nadal będę przyjmował założenie, że chcesz zbudować w swoim umyśle model, który umożliwi Ci zrozumienie języka aż do samych jego podstaw — jeżeli napotkasz problem, będziesz mógł wykorzystać swój model, uzyskując rozwiązanie. Spróbuje przekazać Ci wiedzę, która umożliwiła mi „myślienie w C++”.

Co nowego w drugim wydaniu?

Książka powstała w wyniku gruntownej modyfikacji jej pierwszego wydania. Modyfikacja ta miała na celu odzwierciedlenie wszystkich zmian, wprowadzonych do języka C++ w wyniku zakończenia prac nad standardem C++, a także wynikła z tego, czego nauczyłem się od czasu pierwszej edycji. Cały tekst pierwszego wydania został przejrzany i napisany ponownie, co wiązało się często ze zmianą przedstawionych przykładów, dopisaniem nowych, a także dodaniem wielu nowych ćwiczeń. Istotna zmiana układu i porządku materiału miała na celu odzwierciedlenie dostępności lepszych narzędzi oraz była rezultatem mojej pogłębianej wiedzy na temat tego, w jaki

sposób ludzie uczą się C++. Dodałem rozdział, będący krótkim wprowadzeniem do pojęć języka C i najistotniejszych cech C++, umożliwiający zrozumienie pozostałej części książki czytelnikom nieznającym podstaw języka C.

A zatem krótka odpowiedź na pytanie „co nowego w drugim wydaniu?” brzmi: to, co nie jest w nim zupełnie nowe, zostało gruntownie przeredagowane, czasami w stopniu uniemożliwiającym rozpoznanie pierwotnych przykładów i towarzyszącego im materiału.

Zawartość drugiego tomu książki

Zakończenie prac nad standardem C++ spowodowało dodanie do standardowej biblioteki C++ pewnej liczby nowych i ważnych bibliotek, takich jak łańcuchy, kontenery i algorytmy, a także wprowadzenie złożonych konstrukcji związanych z szablonami. Te i inne trudniejsze tematy zostały przeniesione do drugiego tomu książki¹, zawierającego takie zagadnienia, jak: wielokrotne dziedziczenie, obsługa wyjątków, wzorce projektowe, a także przykłady dotyczące budowy i uruchamiania stabilnych systemów.

Skąd wziąć drugi tom książki?

Podobnie jak książka, którą trzymasz w ręce, jej drugi tom zatytułowany *Thinking in C++, Volume 2* można w całości pobrać z internetowej witryny, znajdującej się pod adresem <http://helion.pl/online/thinking/index.html> W witrynie tej można również znaleźć informacje dotyczące przewidywanego terminu druku drugiego tomu.

Witryna zawiera również kod źródłowy programów zawartych w obu książkach, łącznie z poprawkami oraz informacjami dotyczącymi kursów na CD-ROM-ach, oferowanych przez MindView, Inc., otwartych seminariów, szkoleń stacjonarnych, konsultacji, doradztwa oraz prezentacji.

Wymagania wstępne

W pierwszym wydaniu książki przyjąłem założenie, że znasz język C przynajmniej na poziomie umożliwiającym czytanie napisanych w tym języku programów. Moim podstawowym celem było uproszczenie tego, co uważałem za trudne: języka C++. W tym wydaniu dodałem rozdział stanowiący krótkie wprowadzenie do C, lecz nadal zakładam, że masz pewne doświadczenie w programowaniu. Czytając powieść uczysz się wielu nowych słów w sposób intuicyjny, odwołując się do kontekstu, w którym występują; podobnie możesz uzyskać wiele informacji dotyczących C.

Naukajęzyka C++

Rozpocząłem swoją drogę prze? C++ w tym samym miejscu, w którym, jak sądzę znajduje się wielu czytelników książki jako programista z bardzo poważnym ugruntowanym podejściem do programowania Co gorsza, moje przygotowanie i doświadczenie pochodziły z programowania systemów wbudowanych na poziomie sprzętowym, w którym C był często postrzegany jako język wysokiego poziomu, stanowiący nieefektywny i rozrzutny sposób upychania bitów Później odkryłem, że nie byłem nawet bardzo dobrym programistą C, ukrywając moją ignorancję dotyczącą struktur, funkcji `malloc()` i `free()`, `setjmp()` i `longjmp()`, a także innych „wyrafionowych” pojęć Unikałem dyskusji na te tematy zamiast skorzystać z okazji pozyksania nowej wiedzy

Kiedy rozpoczynałem swoją przygodę z językiem C++, jedyną przyzwoitą książką na ten temat był samozwanzcy „przewodnik eksperta” Bjarne Stroustrup², a zatem pozostało mi uproszczenie podstawowych pojęć na własną rękę. Rezultatem była moja pierwsza książka o C++³, w której przede wszystkim przelałem na papier własne doświadczenia. Była ona pomyślana jako poradnik, umożliwiający programistom równoczesną naukę C i C++. Oba wydania książki⁴ spotkały się z entuzjastycznym przyjęciem

Mniej więcej w tym samym czasie, gdy ukazało się *Using C++*, rozpoczęłem nauczanie tego języka w ramach seminariów i prezentacji Nauczanie C++ (a później języka Java) stało się moim zawodem. Poczytając od 1989 roku widywałem potakujące głowy, twarze bez wyrazu i zagadkowe miny słuchaczy na całym świecie. Kiedy rozpoczęłem prowadzenie szkoleń przeznaczonych dla mniejszych grup, cos odkryłem w trakcie tych zajęć. Nawet ci spośród uczestników, którzy uśmiechali się i potakowali, byli często zdezorientowani. Przez wiele lat zajmując się tematyką C++ i Javy w ramach Software Development Conference doszedłem do wniosku, że zarówno ja sam, jak i inni prelegenci próbujemy przekazać typowemu odbiorcy nadmierną ilość informacji w zbyt krótkim czasie. W rezultacie z powodu zróżnicowanego poziomu słuchaczy i sposobu, w jaki prezentowałem materiał, kończyłem prelekcje, me docierając do części auditorium. Być może chciałem osiągnąć zbyt wiele, ale ponieważ jestem przeciwnikiem tradycyjnego sposobu prowadzenia wykładów (dla większości słuchaczy, jak sądzę, sprzeciw tak wynika ze znudzenia), dążyłem do utrzymania jednakowego tempa dla wszystkich

Przez pewien czas tworzyłem wiele różnych prezentacji w krótkich odstępach czasu. W taki sposób skończyłem z nauką prowadzoną metodą eksperymentowania i nawrotów (ta technika sprawdza się również przy projektowaniu programów w C++) Ostatecznie opracowałem kurs, wykorzystując wszystko, czego nauczyłem się w wyniku moich doświadczeń związanych z nauczaniem. Każdy problem jest przedstawiony osobno, w formie łatwych do zrozumienia kroków, zas po każdej prezentacji następują ćwiczenia praktyczne (idealna metoda nauki). Więcej informacji na temat

² Bjarne Stroustrup *The C++ Programming Language* Addison Wesley 1986 (pierwsze wydanie)

³ *UsingC++*, Osborne/McGraw Hill, 1989

⁴ *UsingC++ and C++ Inside & Out*, Osborne/McGraw-Hill, 1993

prowadzonych przeze mnie otwartych seminariów znaleźć można na stronie www.BruceEckel.com — są tam również informacje na temat kursów, które zapisałem na CD-ROM-ach.

Materiał zawarty w pierwszym wydaniu książki, opracowany w ciągu ponad dwóch lat, został przetestowany na wiele sposobów, na licznych rozmaitych kursach. Opinie, które zebrałem w trakcie każdego kursu, pomogły mi dokonać poprawek w materiale, tak aby lepiej służył celom nauczania. Nie jest to jednak zapis treści kursu — na nielicznych stronach umieściłem tyle informacji, ile zdołałem, i uporządkowałem je w taki sposób, by umożliwić przejście do następnego tematu. Książka jest adresowana przede wszystkim do czytelnika samotnie zgłębiającego tajniki nowego języka programowania.

Cele

W trakcie pisania niniejszej książki przyświecały mi następujące cele:

1. Prezentacja materiału **metodą „małych kroków”**, dzięki którym czytelnik może łatwo zrozumieć każde pojęcie, zanim przejdzie do kolejnych zagadnień.
2. Posługiwianie się przykładami możliwie jak najprostszymi i najkrótszymi. Uniemożliwia to często rozpatrywanie rzeczywistych przykładów, ale zauważę, że poczynającą są na ogół bardziej zadowoleni, kiedy potrafią zrozumieć każdy szczegół przykładu niż gdy są pod wrażeniem zakresu problemu, który ów przykład ilustruje. Obowiązuje również ograniczenie co do wielkości kodu, który może być przyswojony w czasie lekcji. Z tego powodu otrzymuję czasami uwagi krytyczne dotyczące używania przeze mnie „dziecięcych przykładów”, ale jestem skłonny to zaakceptować w imię skuteczności metody nauczania.
3. Staranny dobór kolejności prezentacji poszczególnych właściwości, by czytelnik nie napotkał czegoś, czego wcześniej nie poznał. Oczywiście, nie zawsze jest to możliwe — w takich przypadkach zamieszczone zostanie krótkie wprowadzenie.
4. Przekazywanie tego, co uważam za istotne dla zrozumienia języka, nie zaś wszystkiego, co wiem najego temat. Wierzę w „hierarchię ważności informacji” oraz w to, że istnieją wiadomości, których 95 procent programistów nigdy nie będzie potrzebowało; wprowadzają one jedynie zamieszanie i wywołują wrażenie złożoności języka. Na przykład w języku C zapamiętanie tabeli priorytetów (nigdy nie udało mi się tego dokonać) pozwala na przemyślniejsze zapisanie kodu. Jeżeli jednak zastanowisz się nad tym, dojdziesz do wniosku, że sprawi to kłopot komuś, kto będzie ten kod czytał lub zajmował się jego pielęgnacją. Zapomnij więc o priorytetach i uzywaj w przypadku niejasności nawiasów. Ta sama uwaga dotyczy niektórych informacji ojęzyku C++, które — jak sądzę — są ważniejsze dla twórców kompilatorów niż dla programistów.

5. Przedstawienie poszczególnych partii książki w sposób na tyle zwięzły, by zachować zarówno rozsądny czas lektury, jak i odpowiednie przerwy pomiędzy ćwiczeniami. Pozwała to na aktywny i twórczy udział uczestników w zajęciach praktycznych, a także daje czytelnikowi poczucie lepszego zrozumienia materiału.
6. Dostarczenie czytelnikom solidnych podstawa, umożliwiających im zrozumienie zagadnień w stopniu pozwalającym na przejście do trudniejszych kursów oraz książek (w szczególności, do drugiego tomu książki).
7. Unikałem używania wersji języka C++ związanej z jakimś konkretnym producentem, ponieważ uważam, że w przypadku nauki języka szczególnie dotyczące implementacji nie są tak istotne, jak sam język. Zawartość większości dokumentacji, dostarczanych przez producentów i opisujących szczegóły dokonanych przez nich implementacji, jest w zupełności zadowalająca.

Zawartość rozdziałów

C++ jest językiem, którego nowe i różnorodne właściwości zostały zbudowane na fundamencie istniejącej już składni (z uwagi na to jest on nazywany *hybrydowym* językiem obiektowym). W miarę jak kolejne osoby uczestniczą w procesie uczenia się, zaczynamy rozumieć drogę przebytą przez programistów osiągających kolejne etapy poznawania cech języka C++. Ponieważ wydaje się ona naturalnym kierunkiem umysłu ukształtowanego proceduralnie, postanowiłem to zjawisko zrozumieć, a następnie podążyć tą samą drogą. Pragnąłem przyspieszyć ten proces za pomocą formułowania pytań nasuwających się w trakcie nauki, udzielania na nie odpowiedzi, a także odpowiadania na pytania słuchaczy, których sam uczyłem się języka.

Niniejszy kurs został przygotowany z myślą o ułatwieniu nauki C++. Uwagi słuchaczy pozwoliły mi zrozumieć, które partie materiału sprawiają trudności i wymagają dodatkowego naświetlenia. W trakcie prezentacji materiału przekonałem się, że w miejscach, w których ambitnie zawarłem opis zbyt wielu cech jednocześnie, wprowadzenie dużej liczby nowych pojęć wymaga ich wyjaśnienia, co prowadzi do szybkiego wzrostu poziomu frustracji słuchaczy. W rezultacie zadałem sobie trud jednoczesnego wprowadzania możliwie jak najmniejszej liczby pojęć — w idealnym przypadku tylko jednego istotnego pojęcia w każdym z rozdziałów.

A zatem celem każdego rozdziału jest przybliżenie pojedynczego pojęcia lub niewielkiej grupy powiązanych ze sobą pojęć w sposób niewymagający korzystania z żadnych dodatkowych terminów. Dzięki temu możliwe jest zrozumienie każdej partii materiału w kontekście dotychczas posiadanej wiedzy, jeszcze przed przystąpieniem do dalszej lektury. Aby to osiągnąć, pozostawiłem pewne elementy języka C dłużej niż zamierzałem. Wynikającą z tego korzyścią jest to, że nie będziesz poirytowany z powodu stosowania wszystkich cech języka C++, zanim zostaną one wyjaśnione. Dzięki temu wprowadzenie do języka będzie odbywało się stopniowo, odzwierciedlając sposób, w jaki zazwyczaj przyswajasz sobie nowe informacje.

Poniżej zamieszczono krótki opis rozdziałów zawartych w książce:

Rozdział 1.: Wprowadzenie do obiektów. Kiedy projekty stają się zbyt wielkie i złożone, by można było nimi w łatwy sposób zarządzać, dochodzi do „kryzysu oprogramowania”. Programiści oznajmują wówczas: „nie jesteśmy w stanie dokończyć realizowanych projektów, a jeżeli nawet potrafimy, to są one zbyt kosztowne!”. Wywołuje to lawinę reakcji, omawianych w tym rozdziale. Zostały w nim również opisane idee programowania obiektowego (ang. *OOP — object oriented programming*) oraz wskazówki dotyczące sposobów przewyciężenia kryzysu oprogramowania. W tym rozdziale zapoznasz się także z podstawowymi pojęciami cech programowania obiektowego, a także ze wstępem do procesów analizy i projektowania. Dowiesz się ponadto o korzyściach i obawach związanych z wykorzystaniem języka oraz o przesłankach przemawiających za przejściem do świata C++.

Rozdział 2.: Tworzenie i używanie obiektów. W rozdziale został omówiony proces budowy programów z wykorzystaniem kompilatorów i bibliotek. Zaprezentowano pierwszy program w języku C++ i opisano sposób tworzenia i komplikacji programów. Następnie przedstawione są niektóre z podstawowych bibliotek obiektów, dostępnych w standardzie C++. Zapoznawszy się z treścią tego rozdziału, będziesz już dobrze zorientowany w tym, czym jest pisanie programów w języku C++, wykorzystujące standardowe biblioteki obiektów.

Rozdział 3.: Język C w C++. Ten rozdział stanowi zwięzły przegląd tych elementów języka C, które są używane w C++, oraz podstawowych cech języka dostępnych wyłącznie w C++. Prezentuje również program „make”, używany powszechnie przy projektowaniu oprogramowania, a także wykorzystywany we wszystkich przykładach zawartych w książce (kody źródłowe przykładów w książce, dostępne pod adresem <ftp://ftp.helion.pl/przyklady/thicpp.zip>, zawierają pliki programu make dla każdego z rozdziałów). Rozdział 3. zakłada, że masz solidne podstawy w dziedzinie programowania w językach proceduralnych, takich jak Pascal, C lub nawet niektóre odmiany Basica (o ile napisałeś dostatecznie dużo kodu w tym języku — szczególnie funkcji).

Rozdział 4.: Abstrakcja danych. Większość własności języka C++ dotyczy możliwości tworzenia nowych typów danych. Nie tylko zapewnia to doskonałą organizację kodu, ale stanowi również podstawę bardziej zaawansowanych możliwości programowania obiektowego. Przekonasz się, w jaki sposób jest realizowana ta idea, poprzez umieszczanie funkcji wewnętrz struktur, szczegółowo pokazujące, jak to zrobić; dowiesz się także, jaki rodzaj kodu wówczas powstaje. Nauczysz się również najlepszego sposobu organizowania kodu w pliki nagłówkowe oraz pliki zawierające implementację.

Rozdział 5.: Ukrywanie implementacji. Używając słowa kluczowego **private**, można sprawić, by niektóre dane i funkcje zawarte w strukturze były niedostępne dla użytkownika nowo utworzonego typu. Oznacza to, że jest możliwe oddzielenie wewnętrznej implementacji od interfejsu widzianego przez programistę wykorzystującego kod, co pozwala na łatwą zmianę tej implementacji, bez konieczności modyfikacji programu klienta. Zostało wprowadzone słowo kluczowe **class**, będące eleganckim sposobem opisu nowego typu danych. Ponadto rozszerzono znaczenie słowa „obiekt” (**jest** to szczególny rodzaj zmiennej).

Rozdział 6.: Inicjalizacja i sprzątanie. Jedną z najczęstszych przyczyn błędów w C++ są niezainicjowane zmienne. *Konstruktor w C++* gwarantuje, że zmienne utworzonego przez ciebie typu danych („obiekty twojej klasy”) zostaną zawsze prawidłowo zainicjowane. Jeżeli obiekty te wymagają również, aby po nich „posprzątać”, można użyć w języku C++ *destruktora*, zapewniającego, że sprzątanie to zostanie zawsze wykonane.

Rozdział 7.: Przeciążanie nazw funkcji i argumenty domyślne. Język C++ ma za zadanie wspomaganie realizacji dużych, złożonych projektów. W czasie ich tworzenia może się zdarzyć dołączenie wielu bibliotek, używających tych samych nazw funkcji. Być może zechcesz również używać tej samej nazwy, nadając jej różne znaczenia w obrębie pojedynczej biblioteki. Język C++ ułatwia to za pomocą *przeciążania nazw funkcji*, pozwalającego na używanie tych samych nazw funkcji, pod warunkiem, że różnią się one między sobą listami argumentów. Domyślne argumenty pozwalają natomiast na wywoływanie tej samej funkcji na różne sposoby, automatycznie dostarczając domyślne wartości niektórych argumentów.

Rozdział 8.: Stałe. Rozdział opisuje słowa kluczowe **const** oraz **volatile**, posiadające w C++ dodatkowe znaczenie — szczególnie gdy są używane w obrębie klas. Dowiesz się, co oznacza zastosowanie słowa kluczowego **const** w definicji wskaźnika. W rozdziale pokazano również, jak zmienia się znaczenie słowa kluczowego **const** w zależności od tego, czy jest ono używane wewnętrz, czy na zewnątrz klas, a także w jaki sposób utworzyć wewnętrz klas stałe o wartościach określonych w czasie komplikacji.

Rozdział 9.: Funkcje inline. Makroinstrukcje preprocesora eliminują narzut związany z wywołaniem funkcji, ale pozbawiają również zysków wynikających z kontroli typów w C++. Funkcje inline łączą wszystkie korzyści makroinstrukcji preprocesora z korzyściami wynikającymi z rzeczywistego wywołania funkcji. W rozdziale opisano szczegółowo implementację oraz sposób używania funkcji inline.

Rozdział 10.: Zarządzanie nazwami. Tworzenie nazw należy do podstawowych czynności podczas programowania — w miarę rozrastania się projektu liczba używanych nazw może nadmiernie wzrosnąć. C++ pozwala na zachowanie doskonałej kontroli nad nazwami pod względem ich tworzenia, widoczności, umieszczenia w pamięci i łączenia. Rozdział prezentuje kontrolę nazw w C++ z wykorzystaniem dwóch technik. W pierwszej z nich do kontroli widoczności i łączenia jest używane słowo kluczowe **static**. Opisano także jego specjalne znaczenie związane z klasami. Znacznie bardziej użyteczną techniką kontroli nazw i ich globalnego zasięgu jest **przestrzeń nazw C++**, pozwalająca na rozbicie globalnej przestrzeni nazw na rozłączne regiony.

Rozdział 11.: Referencje i konstruktor kopiący. Wskaźniki w C++ działają tak, jak wskaźniki w C, z dodatkową korzyścią wynikającą z silniejszej kontroli typów w C++. Język C++ dostarcza kolejnej metody adresowania, zapożyczonej z Algol i Pascala — *referencji*, umożliwiającej obsługę adresów przez kompilator przy zachowaniu normalnego sposobu zapisu programu. W rozdziale tym zetkniesz się również z konstruktorem kopującym, zarządzającym sposobem, w jaki obiekty są przekazywane przez wartość do i z funkcji. Na koniec wyjaśniono zagadnienie używania wskaźników do składowych klas.

Rozdział 12.: Przeciążanie operatorów. Właściwość ta jest czasem nazywana „ukierkiem składniowym” – pozwala „osłodzić” składnię używania własnych typów poprzez możliwość stosowania zarówno operatorów, jak i wywołań funkcji. W rozdziale tym dowiesz się, że przeciążenie operatora jest innym rodzajem wywołania funkcji. Zapoznasz się z tworzeniem własnych operatorów i radzeniem sobie z mylącym czasami sposobem używania argumentów i zwracanymi typami. Nauczysz się także podejmowania decyzji, czy operator powinien być składową klasy, czy też funkcją zaprzyjaźnioną.

Rozdział 13.: Dynamiczne tworzenie obiektów. Iloma samolotami będzie zarządzał system kontroli lotów? Ile figur geometrycznych potrzebował będzie system CAD? W przypadku ogólnego problemu programistycznego nie jest znana liczba, czas istnienia oraz typ obiektów wymaganych przez działający program. W tym rozdziale dowiesz się, w jaki sposób operatory new i delete elegancko rozwiązują ten problem w języku C++, w bezpieczny sposób tworząc obiekty na stercie (ang. *heap*). Zobaczysz również, jak operatory new i delete mogą być przeciążane na różne sposoby, umożliwiające kontrolę nad sposobem przydzielania i zwalniania pamięci.

Rozdział 14.: Dziedziczenie i kompozycja. Abstrakcja danych umożliwia tworzenie od podstaw nowych typów, jednakże kompozycja i dziedziczenie pozwalały na tworzenie nowych typów na podstawie typów już istniejących. W przypadku kompozycji nowy typ jest tworzony z innych typów, jak z klocków, podczas gdy dziedziczenie pozwala na utworzenie bardziej wyspecjalizowanej wersji istniejącego typu. W rozdziale tym poznasz składnię, sposób redefiniowania funkcji, a także znaczenie konstrukcji i destrukcji dla dziedziczenia i kompozycji.

Rozdział 15.: Polimorfizm i funkcje wirtualne. Możesz stracić dziewięć miesięcy, by odkryć na własną rękę i zrozumieć znaczenie tego kamienia węgielnego programowania obiektowego. Na podstawie niewielkich, prostych przykładów zobaczyś, w jaki sposób można tworzyć rodzinę typów, używając dziedziczenia i manipulując obiektami należącymi do tej rodziny za pomocą ich wspólnej klasy podstawowej. Słowo kluczowe virtual pozwala na traktowanie wszystkich obiektów należących do tej rodziny w sposób ogólny, co oznacza, że większość kodu nie jest zależna od informacji dotyczącej jakiegoś konkretnego typu. Umożliwia to rozbudowę programów, czyniąc ich tworzenie oraz pielęgnację kodu łatwiejszymi i tańszymi.

Rozdział 16.: Wprowadzenie do szablonów. Dziedziczenie i kompozycja umożliwiają wielokrotne wykorzystywanie kodu obiektu, ale nie zaspokajają wszystkich potrzeb związanych z wielokrotnym użyciem kodu. Szablony umożliwiają powtórne wykorzystanie kodu źródłowego, pozwalając kompilatorowi na zastąpienie nazw typów występujących w ciele klasy lub funkcji. Umożliwia to używanie bibliotek *klas kontenerowych*, stanowiących ważne narzędzie w szybkim i niezawodnym projektowaniu programów obiektowych (standardowa biblioteka C++ zawiera ważną bibliotekę klas kontenerowych). Rozdział dostarcza gruntownych informacji dotyczących tego istotnego tematu.

Dodatkowe tematy (i trudniejsze przykłady) zawarto w drugim tomie książki, który można pobrać z witryny internetowej <http://helion.pl/online/thinking/index.html>.

Ćwiczenia

Odkryłem, że ćwiczenia są wyjątkowo użyteczne w czasie seminariów, pogłębiając rozumienie materiału przez ich uczestników, dlatego też znajdują się one na końcu każdego rozdziału. Liczba ćwiczeń została znacznie zwiększena w stosunku do pierwszego wydania książki.

W większości ćwiczenia sądostatecznie łatwe, by mogły być wykonane w rozsądny czasie — w sali wykładowej lub laboratorium, pod kontrolą osoby prowadzącej zajęcia, dzięki czemu może ona upewnić się, że wszyscy słuchacze przyswoili sobie materiał. Niektóre ćwiczenia są nieco bardziej skomplikowane, by przykuć uwagę również zaawansowanych uczestników zajęć. Wiele ćwiczeń przygotowano w taki sposób, by można je było szybko rozwiązać — służą one raczej sprawdzeniu i ugruntowaniu wiedzy niż prezentacji istotnych problemów (te ostatnie zapewne znajdziesz samodzielnie albo, co bardziej prawdopodobne, to one dopadną ciebie).

Rozwiązania ćwiczeń

Rozwiązania wybranych ćwiczeń zamieszczono w dokumencie elektronicznym *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą w witrynie www.BruceEckel.com.

Kod źródłowy

Kod źródłowy programów zawartych w książce jest oprogramowaniem bezpłatnie dostępnym, chronionym prawem autorskim. Można je pobrać pod adresem <ftp://ftp.helion.pl/przyklady/thicpp.zip>.⁵ Prawo autorskie uniemożliwia przedruk kodu w środkach masowego przekazu bez uzyskania na to zgody, ale daje prawo używania go na wiele innych sposobów (zob. informacje poniżej).

Kod jest dostępny w postaci spakowanego archiwum, możliwego do rozpakowania na dowolnej platformie sprzętowo-programowej, na której dostępny jest program użytkowy „zip” (dla większości platform jest on dostępny — poszukaj odpowiedniej wersji w Internecie, o ile nie jest ona już zainstalowana w twoim systemie). W katalogu, do którego rozpakujesz pliki, znajdziesz następującą informację o prawach autorskich:

```
//:! :Copyright.txt
Copyright (c) 2000, Bruce Eckel
Kod źródłowy pochodzący z książki "Thinking in C++"
Wszelkie prawa zastrzeżone, I WYJĄTKIEM dozwolonych poniżej:
Niniejszy plik może być bezpłatnie wykorzystywany do własnych
celów (osobistych lub komercyjnych), włączając w to modyfikację
```

⁵ Ze względu na przejrzystość tłumaczenia, komentarze w kodach źródłowych widoczne w książce Posiadają polskie znaki diakrytyczne, natomiast kody pobrane z serwera FTP — nie posiadają tych znaków — przyp. red.

oraz dystrybucję wyłącznie w postaci wykonywalnej. Udzielane jest pozwolenie na używanie tego pliku podczas zajęć dydaktycznych, włączając w to wykorzystanie go w materiałach prezentacyjnych, pod warunkiem wskazania książki "Thinking in C++" jako źródła jego pochodzenia. Poza zastosowaniami dydaktycznymi nie można kopiować ani rozpowszechniać niniejszego kodu. Wyłącznym miejscem dystrybucji jest witryna <http://www.BruceEckel.com> (i jej oficjalne witryny lustrzane), w której jest on dostępny bezpłatnie. Nie można usuwać zastrzeżenia prawa autorskiego ani niniejszej informacji. Nie wolno rozpowszechniać zmodyfikowanej wersji kodu źródłowego, zawartego w niniejszym pakiecie. Nie wolno publikować niniejszego pliku drukiem, bez wyraźnej zgody autora. Bruce Eckel nie bierze odpowiedzialności za to, że niniejsze oprogramowanie będzie przydatne do jakichkolwiek celów. Oprogramowanie jest dostarczane "takie, jakie jest", bez bezpośredniej ani pośredniej gwarancji jakiegokolwiek rodzaju, włączając w to gwarancję przydatności handlowej, przydatności do konkretnego zastosowania oraz nie naruszania prawa. Całkowite ryzyko związane z jakością i wydajnością oprogramowania ponosi jego użytkownik. Bruce Eckel oraz wydawca nie ponoszą odpowiedzialności za jakiekolwiek zniszczenia wywołane u użytkownika ani u jakiekolwiek strony trzeciej w wyniku używania lub dystrybucji niniejszego oprogramowania. W żadnym wypadku Bruce Eckel ani wydawca nie ponoszą odpowiedzialności za jakiekolwiek utracone przychody, zyski lub dane, a także bezpośrednie, pośrednie, szczególne, wynikowe, przypadkowe, lub wynikające z naruszenia prawa uszkodzenia, wywołane w jakikolwiek sposób, niezależnie od teorii odpowiedzialności wynikającej z używania lub niemożności używania oprogramowania, nawet jeżeli Bruce Eckel oraz wydawca zostali powiadomieni o możliwości wystąpienia takich uszkodzeń. W przypadku ujawnienia błędów zawartych w oprogramowaniu należy wziąć pod uwagę poniesienie kosztów niezbędnego serwisu, jego naprawy lub korekty. Każdy, kto uważa, że znalazł błąd, proszony jest o jego zgłoszenie za pomocą formularza dostępnego na stronie www.BruceEckel.com (tego samego formularza należy użyć do zgłoszenia błędów niezwiązanych z kodem).

Możesz używać kodu we własnych projektach oraz w dydaktyce, dopóki pozostaje w nim informacja o zastrzeżeniu prawa autorskiego.

Standardy języka

Kiedy odwołuję się do zgodności ze standardem ISO języka C, w całej książce na ogół piszę „C”. Tylko w przypadku konieczności rozróżnienia pomiędzy standardowym C i starszą (sprzed ustanowienia standardu) wersją języka, wyraźnie to zaznaczam.

W czasie pisania książki Komitet Standardyzacyjny C++ zakończył pracę nad językiem. A zatem używam określenia *standardowe C++* w celu odwołania się do ustanionego standardu języka. Jeżeli zaś odwołuję się do C++, czytelnik powinien założyć, że mam na myśli „zwykłe C++”.

Istnieje pewna niejednoznaczność związana z obecną nazwą komitetu standaryzacyjnego C++ oraz samą nazwą standardu. Steve Clamage, przewodniczący komitetu, wyjaśnił to następująco:

Istnieją dwa komitety standaryzacyjne C++: Komitet J16 NCITS (poprzednio X3) oraz komitet ISO JTC1/SC22/WG14 ANSI powołał NCITS po to, by utworzyć komitet techniczny, mający na celu zaprojektowanie amerykańskich standardów narodowych.

J16 został powołany w 1989 r. w celu utworzenia amerykańskiego standardu C++. Mniej więcej w 1991 r. w celu utworzenia standardu międzynarodowego został powołany WG14. Projekt J16 został przekształcony w projekt „typu I” (międzynarodowy) i podporządkowany działaniom standaryzacyjnym ISO.

Oba komitety spotkały się w tym samym czasie i w tym samym miejscu; głos J16 stanowił amerykański głos za W14. WG14 przekazał prace techniczne projektowi JJ6. Następnie WG14 przegłosował prace techniczne przeprowadzone przez JJ6.

Standard C++ został pierwotnie utworzony jako standard ISO. Później ANSI przegłosował (zgodnie z rekomendacją J16) przyjęcie standardu ISO C++ jako amerykańskiego standardu C++,

Tak więc „ISO” jest właściwym sposobem odwoływania się do standardu C++.

Obsługa języka

Używany przez ciebie kompilator może nie obsługiwać wszystkich właściwości języka, omawianych w niniejszej książce, zwłaszcza gdy nie posiadasz jego najnowszej wersji. Implementacja takiego języka, jak C++ jest herkulesowym zadaniem; możesz się zatem spodziewać, że ujawni się część tych właściwości niż wszystkie naraz. Jeżeli jednak spróbujesz wykonać jeden z zawartych w książce przykładów i otrzymasz mnóstwo błędów zgłoszonych przez kompilator, niekoniecznie oznacza to błąd zawarty w kodzie lub w kompilatorze. Cecha ta może być po prostu jeszcze niezaimplementowana w używanym przez ciebie kompilatorze.

Błędy

Niezależnie od tego, jakich sztuczek używa autor w celu znalezienia błędów, niektóre z nich są nieuniknione i często okazują się potem oczywiste dla kogoś czytającego książkę po raz pierwszy. Jeżeli odkryjesz coś, co uważasz za błąd, skorzystaj, proszę, z formularza dostępnego na stronie www.BruceEckel.com. Twoja pomoc będzie mile widziana.

Okładka

Na okładce pierwszego wydania książki widniał wizerunek mojej twarzy, ale pierwotnie chciałem, by okładka drugiego wydania była raczej dziełem artysty, podobnie jak okładka książki *Thinking in Java*. Z jakiegoś powodu C++ kojarzyło mi się z art déco, z jego nieskomplikowanymi krzywymi i zmatowionymi chromami. Miałem na myśli te plakaty statków i samolotów o wydłużonych kadłubach.

Mój przyjaciel, Daniel Will-Harris (www.Will-Harris.com), z którym spotkałem się po raz pierwszy w klasie chóru, w gimnazjum, zrobił karierę światowej sławy projektanta i pisarza. Wykonał właściwie wszystkie moje projekty, w tym okładkę pierwszego wydania tej książki. Podczas projektowania okładki Daniel, niezadowolony z postępów, które poczyniliśmy, pytał: „W jaki sposób łączy to ludzi z komputerami?”. Ugrzęźliśmy.

Pod wpływem impulsu, bez żadnego konkretnego zamysłu, Daniel poprosił mnie, abym położył twarz na skanerze. Za pomocą jednego ze swoich programów graficznych (ulubionego — Corela Xary) dokonał „automatycznego trasowania” zeskanowanego wizerunku mojej twarzy. Opisał to następująco: „Automatyczne trasowanie jest komputerowym sposobem przekształcenia obrazu w linie i krzywe, które go naprawdę przypominają”. Następnie bawił się tym obrazem dopóty, dopóki uzyskał coś w rodzaju topograficznej mapy mojej twarzy — ilustrację tego, w jaki sposób komputery mogłyby postrzegać ludzi.

Skopiowałem ten obraz na papierze akwarelowym (niektóre kolorowe kserokopiarki wykonują kopie na grubych materiałach) i zrobiłem mnóstwo prób, kolorując go akwarelami. Następnie wybraliśmy te, które wyglądały najlepiej; Daniel zeskanował je ponownie i ułożył na okładce, dodając tekst i inne elementy projektu. Wszystko to trwało kilka miesięcy, głównie z uwagi na czas, jaki zajęło mi malowanie akwareli. Ale jestem z tej okładki szczególnie zadowolony, ponieważ mam swój udział w przedstawionym na niej dziele sztuki, a ponadto praca nad nią zachęciła mnie do namalowania kolejnych akwareli (prawdą jest to, co się mówi o praktyce).

Rozdział 1.

Wprowadzenie do obiektów

Rewolucja komputerowa rozpoczęła się od rozwoju sprzętu. Pierwsze języki programowania były więc próbą naśladowania zachowań komputerów.

Komputery jednak są nie tyle maszynami, ile raczej „wzmacniaczami umysłu” oraz nowym środkiem ekspresji. W rezultacie, coraz mniej przypominają maszyny, upodabniając się do elementów naszych umysłów i przypominając inne środki wyrazu, takie jak literatura, malarstwo, rzeźba, animacja i twórczość filmowa. Programowanie obiektowe stanowi krok w kierunku używania komputera w charakterze środka ekspresji.

Rozdział ten wprowadzi cię w podstawowe pojęcia programowania obiektowego (ang. *OOP – object oriented programming*), włączając w to przegląd obiektowych metod projektowania. Przyjęto założenie, że masz doświadczenie w używaniu jakiegoś języka proceduralnego, którym niekoniecznie musi być C. Jeżeli uważasz, że zanim zaczniesz czytać tę książkę, potrzebujesz lepszego przygotowania w zakresie programowania oraz informacji na temat składni języka C, warto zapoznać się z kursem „Thinking in C++: Foundations for C++ and Java”, zamieszczonym na dołączonym do książki CD-ROM-ie (dostępny również pod adresem: <http://Milion.pl/online/thinking/index.html>).

Rozdział zawiera zarówno podstawowe informacje, jak i materiał uzupełniający. Wiele osób odczuwa dyskomfort, debiutując w dziedzinie programowania obiektowego, jeżeli nie zrozumie najpierw jego ogólnej idei. Dlatego też wprowadzono wiele pojęć, zapewniających gruntowny przegląd programowania obiektowego. Niektórzy jednak nie są w stanie uchwycić ogólnych idei, dopóki nie poznają szczegółów — mogą oni czuć się zagubieni, jeżeli nie dostaną do ręki fragmentu programu. Jeżeli należysz do tej drugiej grupy i jesteś gotów poznać szczegóły języka, możesz bez wahania opuścić niniejszy rozdział — nie przeszkodzi ci to w pisaniu programów ani w nauce języka. Jednakże zapewne będziesz chciał powrócić do niego, by uzupełnić swoją wiedzę; dowiesz się bowiem, dlaczego obiekty są ważne i jak ich używać podczas projektowania.

Postęp abstrakcji

Wszystkie języki programowania dostarczają pewnych abstrakcji. Można negować twierdzenie, że złożoność problemów, które można rozwiązać, jest bezpośrednio związana z rodzajem i jakością abstrakcji. Pod pojęciem „rodzaj” rozumiem: „to, co jest przedmiotem abstrakcji”. Język asemblera jest w niewielkim stopniu abstrakcją komputera. Wiele tak zwanych języków imperatywnych, które pojawiły się później (takich jak Fortran, BASIC i C), było abstrakcjami języka asemblera. Stanowiły one istotny postęp w stosunku do języka asemblera, jednak ich podstawowy poziom abstrakcji wymagał nadal myślenia w kategoriach struktury komputera, a nie struktury rozwiązywanego problemu. Programista musi określić związek pomiędzy modelem maszyny (w „przestrzeni rozwiązania”, będącej miejscem, w którym modelowany jest problem — takim jak komputer) i modelem rozwiązywanego problemu (w „przestrzeni problemu”, czyli miejscu, w którym ten problem występuje). Wysiłek konieczny do dokonania takiego odwzorowania, a także fakt, że nie jest on związany z językiem programowania, prowadzą do tworzenia programów trudnych do napisania i kosztownych w utrzymaniu, a ich ubocznym efektem jest powstanie całej dziedziny „metod programowania”.

Rozwiązaniem alternatywnym w stosunku do modelowania maszyny jest modelowanie rozwiązywanego problemu. Wczesne języki programowania, takie jak LISP i APL, preferowały pewne szczegółowe punkty widzenia świata („wszystkie problemy są w istocie listami” lub „wszystkie problemy są algorytmiczne”). PROLOG nadawał wszystkim problemom postać ciągów decyzji. Powstały języki przeznaczone do programowania z ograniczeniami (ang. *constraint-based programming*), a także do programowania wykorzystującego wyłącznie operacje na obiektach graficznych (te ostatnie okazały się zbyt ograniczone). Każde z powyższych ujęć stanowi dobre rozwiązanie klasy problemów, dla których zostało stworzone, lecz poza tą dziedziną okazuje się niewygodne.

Programowanie obiektowe idzie o krok dalej, dostarczając programistie narzędzi umożliwiających reprezentację elementów w przestrzeni problemu. Reprezentacja ta jest wystarczająco ogólna, by nie ograniczać programisty do żadnego określonego rodzaju problemów. Odwołujemy się do elementów w przestrzeni problemu oraz ich reprezentacji w przestrzeni rozwiązania jako do „obiektów” (oczywiście, potrzebne będą również obiekty niemające swych odpowiedników w przestrzeni problemu). Idea polega na umożliwieniu programowi dopasowania się do specyficznego dialekta problemu poprzez dodawanie nowych typów obiektów, dzięki czemu kod oznaczający rozwiązanie wyrażony jest słowami opisującymi problem. Jest to bardziej elastyczny i efektywniejszy poziom abstrakcji języka niż te, którymi dysponowaliśmy do tej pory. A zatem programowanie obiektowe umożliwia wyrażenie problemu we właściwych mu kategoriach, a nie w kategoriach opisujących komputer, na którym będzie on rozwiązywany. Istnieje jednak nadal pewien związek z komputerem. Każdy obiekt przypomina mały komputer — znajduje się w jakimś stanie, a także posiada zbiór operacji, o których wykonanie można go poprosić. Wydaje się to stanowić niezłą analogię do obiektów występujących w rzeczywistym świecie — wszystkie one posiadają pewne szczególne cechy oraz charakteryzują się właściwym sobie zachowaniem.

Niektórzy projektanci języków programowania doszli do wniosku, że samo programowanie obiektowe nie umożliwia łatwego rozwiązywania wszystkich problemów programistycznych i stali się orędownikami kombinacji różnych podejść, tworząc języki programowania *wieloparadygmatowego* (ang. *multiparadigm programming languages*)¹.

Alan Kay podsumował pięć podstawowych cech języka Smalltalk — pierwszego udanegojęzyka obiektowego, stanowiącego zarazem jeden z języków, w oparciu o które powstało C++. Cechy te reprezentują podejście czysto obiektowe:

- 1. Wszystko jest obiektem.** Obiekt należy ujmować jako szczególny rodzaj zmiennej — przechowuje on dane, ale można również w stosunku do niego „zgłosić żądanie”, prosząc obiekt o wykonanie operacji na sobie samym. Teoretycznie, każdy składnik pojęciowy rozwiązywanego problemu (psy, budynki, usługi itp.) może być reprezentowany w programie w postaci obiektu.
- 2. Program jest grupą obiektów, przekazującymi sobie wzajemnie informacje o tym, co należy zrobić, za pomocą komunikatów.** Zgłoszenie żądania w stosunku do obiektu odbywa się poprzez „wysłanie do niego komunikatu”. Komunikat taki może być traktowany jako żądanie wywołania funkcji należącej do tego obiektu.
- 3. Każdy obiekt posiada własną pamięć, złożoną z innych obiektów.** Innymi słowy, nowy rodzaj obiektu jest tworzony poprzez utworzenie pakietu złożonego z już istniejących obiektów. A zatem można powiększać złożoność programu, ukrywając ją równocześnie za prostotą obiektów.
- 4. Każdy obiekt posiada typ.** Mówiąc potocznie, każdy obiekt jest *egzemplarzem* jakiejś *klasy*, przy czym słowo „klasa” jest synonimem słowa „typ”. Najistotniejszą cechą odróżniającą od siebie klasy jest informacja o tym, jakie komunikaty mogą być do nich wysyłane.
- 5. Wszystkie obiekty określonego typu mogą odbierać te same komunikaty.** Jak przekonamy się później, jest to twierdzenie idące nieco zbyt daleko. Ponieważ obiekt typu „okrąg” jest zarazem obiektem typu „figura”, nie ulega wątpliwości, że odbiera on komunikaty dotyczące figur. Oznacza to, że można napisać program odwołujący się do figur, automatycznie obsługujący wszystko to, co odpowiada opisowi figury. Ta *zastępowność* (ang. *substitutability*) należy do najważniejszych koncepcji programowania obiektowego.

Obiekt posiada interfejs

Prawdopodobnie pierwszym, który rozpoczął systematyczne badania związane z pojęciem *typu*, był Arystoteles. Mówił on o „klasie ryb i klasie ptaków”. Pomyśl, by wszystkie obiekty, będąc unikatowymi, były zarazem elementami klasy obiektów, posiadających pewne wspólne cechy i sposoby zachowania, został bezpośrednio wykorzystany w pierwszym języku obiektowym — Simuli-67. Jej podstawowym słowem kluczowym było słowo *class*, umożliwiające utworzenie w programie nowego typu.

¹ Patrz — Timothy Budd: *Multiparadigm Programming in Leda*, Addison-Wesley, 1995.

Simula, jak na to wskazuje jej nazwa, powstała z myślą o przeprowadzaniu symulacji, takich jak klasyczny „problem kasjera bankowego”². W problemie tym występuje zbiór kasjerów, klientów, kont, transakcji i jednostek pieniężnych — czyli wiele „obiektów”. Obiekty, które są identyczne (z wyjątkiem ich aktualnego stanu w czasie wykonywania programu), zostały pogrupowane w „klasy obiektów” — stąd pochodzi właśnie słowo kluczowe **class**. Tworzenie abstrakcyjnych typów danych (klas) jest fundamentalną ideą programowania obiektowego. Abstrakcyjne typy danych funkcjonują niemal tak samo, jak typy wbudowane: można tworzyć zmienne tych typów (nazywane w żargonie programowania obiektowego *obiektem* lub *egzemplarzem*) oraz operować na tych zmiennych (co nazywane jest *wysyłaniem komunikatów* lub *żądań* — wysyłany jest komunikat, a obiekt określa, co powinien z nim zrobić). Składniki (elementy) każdej klasy mają pewne cechy wspólne — każde konto ma saldo, każdy kasjer przyjmuje wpłaty itp. Równocześnie każdy element posiada swój własny stan — każde konto ma inne saldo, a każdy kasjer nosi jakieś nazwisko. A zatem unikatowe jednostki programu komputerowego mogą reprezentować poszczególnych kasjerów, klientów, każde konto, transakcję itp. Jednostki takie są obiektami. Każdy obiekt należy do konkretnej klasy, określającej jego cechy i sposób działania.

A zatem mimo że podczas programowania obiektowego zajmujemy się tworzeniem nowych typów danych, praktycznie we wszystkich obiektowych językach programowania używamy słowa kluczowego „class”. Widząc siwo „typ”, powinniśmy zatem myśleć „klasa” i na odwrót³.

Ponieważ klasa opisuje zbiór obiektów posiadających takie same cechy (elementy danych) i działania (funkcjonalność), jest ona w rzeczywistości typem danych, podobnie jak np. liczba zmiennopozycyjna, mająca również właściwy sobie zbiór cech i możliwych działań. Różnica polega na tym, że programista definiuje klasę w taki sposób, by odpowiadała ona problemowi, nie jest więc zmuszony do używania istniejącego typu danych, zaprojektowanego w celu odzwierciedlenia jednostki pamięci komputera. Rozszerzamy zatem język programowania, dodając do niego typy danych odpowiadające naszym potrzebom. System programowania przyjmuje nowe klasy, traktujące w taki sam sposób i zapewniając im taką samą kontrolę, jak typom wbudowanym.

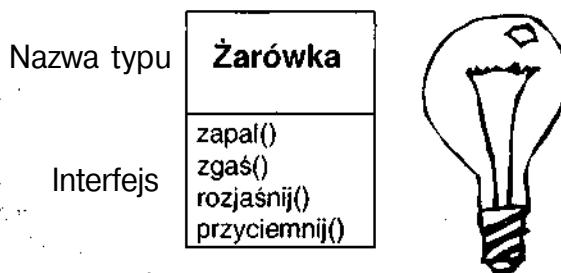
Obiektowe podejście do programowania nie ogranicza się do tworzenia symulacji. Niezależnie od tego, czy zgadzamy się z twierdzeniem, że każdy program jest symulacją projektowanego systemu, użycie technik obiektowych pozwala na łatwe zredukowanie dużego zbioru problemów do prostego rozwiązania.

Po utworzeniu klasy można z łatwością utworzyć dowolną liczbę obiektów tej klasy, a następnie operować na nich w taki sposób, jakby były rzeczywistymi elementami rozwiązywanego problemu. W istocie, jednym z zadań programowania obiektowego jest ustalenie wzajemnie jednoznacznego odwzorowania pomiędzy elementami przestrzeni problemu a elementami przestrzeni rozwiązania.

² Interesującą implementację tego problemu można znaleźć w drugim tomie książki, dostępnym pod adresem <http://helion.pl/online/thinking/index.html>.

³ Niektórzy czynią rozróżnienie, twierdząc, że typ określa interfejs, podczas gdy klasa jest szczególną implementacją tego interfejsu.

Jak jednak można zmusić obiekt do wykonania jakiegoś pożytecznego zadania? Musi istnieć sposób, w jaki można zażądać od obiektu, by np. wykonał transakcję, narysował coś na ekranie albo zmienił stan przełącznika. Ponadto każdy obiekt może spełniać tylko pewne określone żądania. Żądania, które można przekazywać obiekowi, są zdefiniowane poprzez jego *interfejs*. Tym, co określa interfejs, jest natomiast typ obiektu. Prostym tego przykładem może być reprezentacja żarówki⁴:



Żarówka zr;
zr.zapal();

Interfejs określa, *jakie* żądania mogą być kierowane do określonego obiektu. Jednakże gdzieś musi istnieć kod realizujący to żądanie. Wraz z ukrytymi danymi składa się on na *implementację*. Z punktu widzenia programowania proceduralnego nie jest to aż tak skomplikowane. Typ posiada funkcję związaną z każdym możliwym żądaniem i kiedy jest ono zgłoszane, funkcja ta jest wywoływana. Jest to zazwyczaj przedstawiane w taki sposób, że do obiektu „wysyłany jest komunikat” (zgłoszane jest żądanie), a obiekt podejmuje decyzję, co z nim zrobić (wykonuje kod).

W powyższym przykładzie nazwą typu (klasy) jest **Żarówka**, natomiast nazwą konkretnego obiektu typu **Żarówka** jest **zr**. Żądaniami, jakie można skierować do obiektu klasy **Żarówka**, jest jej zapalenie, zgaszenie, rozjaśnienie i przyciemnienie. Można utworzyć obiekt klasy **Żarówka**, określając jego nazwę (**zr**). Aby wysłać obiekowi komunikat, należy wpisać jego nazwę, łącząc ją z komunikatem żądania za pomocą kropki. Z punktu widzenia użytkownika predefiniowanej klasy jest to niemal wszystko, co dotyczy programowania z wykorzystaniem obiektów.

Przedstawiony powyżej diagram jest zgodny z notacją języka *UML* (ang. *Unified Modeling Language* – zunifikowany język modelowania). Każda klasa jest reprezentowana przez prostokąt; w jego górnej części znajduje się nazwa typu, w środkowej — dane składowe, które zamierzamy opisać, a w dolnej — funkcje składowe (funkcje należące do obiektu, odbierające wszelkie wysyłane do niego komunikaty). Diagramy projektowe UML przedstawiają często jedynie nazwę klasy oraz jej publiczne funkcje składowe, w związku z czym nie jest widoczna ich środkowa część. Jeżeli interesuje cię jedynie nazwa klasy, to można pominąć również dolną część diagramu.

⁴ Zawarte w niniejszym rozdziale

Ukryta implementacja

Pomocne jest dokonanie podziału na *twórców klas* (osoby tworzące nowe typy danych) oraz *klientów-programistów* (ang. *client programmers*) — „konsumentów klas”, używających tych typów danych w swoich aplikacjach. Celem klienta-programisty jest skompletowanie zestawu narzędzi, składającego się z klas, który umożliwi mu szybkie opracowanie aplikacji. Celem twórcy klasy jest natomiast utworzenie klasy w taki sposób, by ujawniała jedynie to, co jest niezbędne klientowi-programiście, zachowując resztę w ukryciu. Dlaczego? Dzięki temu, że klientowi-programiście nie wolno używać tego, co jest niewidoczne, twórca klasy ma możliwość dowolnego zmieniania niewidocznej części klasy, bez konieczności uwzględniania wpływu, jaki będzie to miało na kogokolwiek. Ukrytą częścią jest zazwyczaj wrażliwe wnętrze obiektu, które mogłoby zostać łatwo uszkodzone przez nieostrożnego, lub posiadającego zbyt małą wiedzę, klienta-programistę. A zatem ukrywanie implementacji zmniejsza liczbę błędów występujących w programach i odgrywa rolę nie do przecenienia.

W każdej relacji istotne jest określenie granic, respektowanych przez wszystkie zaangażowane w nią strony. Tworząc bibliotekę, ustanawiamy relację z klientem-programistą, będącym również programistą, lecz używającym naszej biblioteki do zbudowania własnej aplikacji lub utworzenia większej biblioteki.

Jeżeli wszystkie składowe klasy są dostępne dla wszystkich, klient-programista może wykonać dowolne działania i nie jest możliwe wyegzekwowanie przestrzegania jakichkolwiek reguł. Nawet w przypadku gdy nie chcemy, by programista bezpośrednio operował na niektórych składowych klasy, bez mechanizmu kontroli dostępu, nie jesteśmy w stanie temu zapobiec. Wszystko jest wystawione na widok publiczny.

A zatem pierwszym powodem wprowadzenia kontroli jest uniemożliwienie programistom dostępu do tego, czego nie powinni się imieć — elementów niezbędnych do wykonywania wewnętrznych operacji związanych z typem danych, lecz niebędących częścią interfejsu, potrzebnego użytkownikom do rozwiązywania ich szczególnych problemów. Jest to w rzeczywistości pomoc udzielana użytkownikom, ponieważ dzięki temu mogą łatwo odróżnić to, co jest dla nich istotne, od tego, co mogą pominąć.

Drugim powodem wprowadzenia kontroli dostępu jest umożliwienie projektantowi biblioteki zmiany wewnętrznych mechanizmów klasy, bez troszczenia się o to, jaki będzie to miało wpływ na klienta-programistę. Na przykład można ułatwić sobie pracę, implementując jakąś klasę w sposób uproszczony, a następnie dojść do wniosku, że konieczne jest jej zmodyfikowanie, tak by działała szybciej. Jeżeli jej interfejs oraz implementacja są od siebie wyraźnie oddzielone i chronione, można to łatwo zrobić, wymagając od użytkownika jedyne powtórnej konsolidacji programu.

Język C++ posiada trzy słowa kluczowe, wykorzystywane bezpośrednio do określenia ograniczeń w klasach: **public**, **private** i **protected**. Ich sposób użycia oraz znaczenie nie budzą wątpliwości — są one *specyfikatorami dostępu* (ang. *access specifiers*),

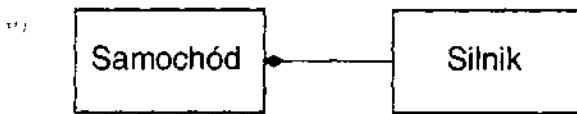
⁵ Jestem wdzięczny za ten termin mojemu przyjacielowi, Scottowi Mayersowi.

określającymi, kto może używać następujących po nich definicji. Specyfikator **public** oznacza, że są one dostępne dla wszystkich. Z kolei słowo kluczowe **private** oznacza, że definicje te mogą być używane wyłącznie przez ciebie (twórcę typu) jedynie wewnątrz funkcji składowych. Słowo **private** stanowi barierę pomiędzy tobą i klientem-programistą. Każdy, kto spróbuje odwołać się do prywatnej składowej klasy, otrzyma komunikat o błędzie już na etapie komplikacji. Znaczenie słowa **protected** jest zblisko do **private**, z wyjątkiem tego, że klasa dziedzicząca ma dostęp do składowych chronionych (występujących po specyfikatorze **protected**) klasy, nie ma natomiast dostępu do jej składowych prywatnych (**private**). Pojęcie dziedziczenia zostanie wkrótce wprowadzone.

Wykorzystywanie istniejącej implementacji

Kiedy klasa zostanie utworzona i przetestowana, powinna (w idealnym przypadku) stanowić użyteczny fragment kodu. Okazuje się, że możliwość jego ponownego wykorzystania nie jest tak łatwa do osiągnięcia, jak można by się tego spodziewać — dobry projekt wymaga bowiem doświadczenia i intuicji. Możliwość wielokrotnego wykorzystywania kodu jest jedną z największych korzyści zapewnianych przez obiektowe języki programowania.

Najprostszym sposobem ponownego wykorzystania klasy jest bezpośrednie użycie obiektu tej klasy. Można również umieścić ten obiekt wewnątrz nowej klasy, co nazываем „utworzeniem obiektu składowego”. Nowa klasa może być utworzona z dowolnej liczby innych obiektów dowolnych typów, w dowolnej kombinacji, niezbędnej do osiągnięcia wymaganej funkcjonalności tworzonej klasy. Ponieważ nowa klasa jest tworzona (komponowana) z klas już istniejących, proces ten nosi nazwę *kompozycji* (lub bardziej ogólnie — *agregacji*). Kompozycja jest często określana jako relacja typu „posiada”, rozumiana w taki sposób, jak występująca w zdaniu „samochód posiada silnik”.



Na powyższym diagramie języka UML kompozycja została oznaczona wypełnionym rombem, który symbolizuje istnienie pojedynczego samochodu. Zazwyczaj będzie używane prostsze oznaczenie połączenia — zwykła linia (bez rombu)⁶.

Kompozycja zapewnia duży stopień elastyczności. Obiekty składowe nowej klasy są zazwyczaj prywatne, co czyni je niedostępnymi dla klienta-programisty wykorzystującego tę klasę. Pozwala to na zmianę tych składowych, bez wpływu na istniejący

⁶ Jest to zazwyczaj wystarczająco precyzyjne w przypadku większości diagramów, na ogół nie jest również konieczne określanie, czy stosowana jest agregacja czy też kompozycja.

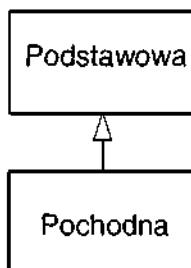
kod, napisany przez klienta. Można również zmieniać obiekty składowe w czasie wykonywania kodu, modyfikując dynamicznie zachowanie programu. Opisane w następnym podrozdziale dziedziczenie nie posiada już takiej elastyczności z uwagi na ograniczenia dotyczące klas tworzonych zajego pomocą, nakładane już w czasie komplikacji.

Ze względu na to, że dziedziczenie odgrywa tak ważną rolę w programowaniu obiektowym, jego znaczenie jest często podkreślane. Może to wywołać u niedoświadczonego programisty przekonanie, że dziedziczenie należy stosować w każdym przypadku. W rezultacie tworzone przez niego programy mogą być niespójne i nadmiernie skomplikowane. Podczas tworzenia nowych klas należy zawsze mieć przed o wszystkim na uwadze kompozycję, ponieważ jest ona prostsza i bardziej elastyczna. Takie podejście umożliwia Ci tworzenie bardziej przejrzystych programów. Gdy już nabierzesz pewnego doświadczenia, stanie się dla ciebie wystarczająco oczywiste, w jakich przypadkach należy używać dziedziczenia.

Dziedziczenie — wykorzystywanie istniejącego interfejsu

Idea obiektu jest sama w sobie wygodnym narzędziem. Pozwala ona na połączenie ze sobą danych oraz funkcji w *pojęcia*, umożliwiające reprezentację odpowiednich elementów przestrzeni problemu, bez konieczności używania dialekta właściwego wykorzystywanej maszyny. Pojęcia te są wyrażane w postaci podstawowych jednostek języka programowania, za pomocą słowa kluczowego class.

Wydaje się jednak, że szkoda byłoby, gdybyśmy zadali sobie trud utworzenia jakiejś klasy, a następnie byli zmuszeni do utworzenia zupełnie nowej, posiadającej być może podobne właściwości. Znacznie lepiej „sklonować” istniejącą klasę, a następnie dokonać, na utworzonej w ten sposób kopii, wszelkich niezbędnych rozszerzeń i modyfikacji. Uzyskujemy ten efekt właśnie dzięki *dziedziczeniu* (ang. *inheritance*). Jedyna różnica polega na tym, że w przypadku gdy oryginalna klasa (zwana *klasą podstawową*, *klasą bazową*, *nadklassą* lub *klasą nadzczną*) zostanie zmieniona, przekształcenia te zostaną również uwzględnione w zmodyfikowanym „klonie” (nazywanym *klasą pochodną*, *klasą potomną*, *podklassą* lub *klasą podrzędną*).

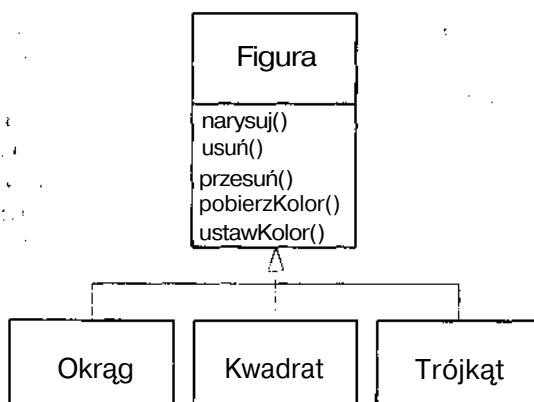


Strzałka na powyższym diagramie UML jest skierowana od klasy pochodnej do podstawowej. Jak przekonamy się później, może istnieć więcej niż jedna klasa pochodna.

Typ to nie tylko opis ograniczeń dotyczących pewnego zbioru obiektów — posiada on również powiązania z innymi typami. Dwa typy mogą mieć wspólne cechy i metody działania, lecz jeden z nich może zawierać więcej cech niż drugi; może również obsługiwać większą liczbę komunikatów (lub obsługiwać je w odmienny sposób). Dziedziczenie opisuje takie właśnie podobieństwo pomiędzy typami, posługując się pojęciami typów podstawowych oraz typów pochodnych. Typ podstawowy posiada wszystkie cechy i sposoby zachowania wspólne dla utworzonych na jego podstawie typów pochodnych. Tworzymy go po to, by reprezentował istotę naszych wyobrażeń dotyczących niektórych obiektów zawartych w systemie. Z typu podstawowego wyprowadzamy inne typy, prezentując różne sposoby, jakie istota ta może być urzeczywistniona.

Rozważmy przykład maszyny sortującej śmieci, przeznaczonej do recyklingu odpadów. Typem podstawowym jest odpad. Każdy odpad ma swoją wagę, wartość itp., a także może być pocięty, przetopiony lub rozmontowany. Z tego typu można wyprowadzić bardziej precyzyjnie zdefiniowane typy odpadków, posiadające dodatkowe cechy (butelka ma na przykład jakiś kolor) lub sposoby funkcjonowania (np. puszka aluminiowa może zostać zgnieciona, a puszka stalowa podlega oddziaływaniu pola magnetycznego). W dodatku mogą się różnie zachowywać (np. wartość papieru zależy od jego rodzaju oraz stanu). Dzięki wykorzystaniu dziedziczenia możemy budować hierarchię typów, opisującą rozwiązywanego problem za pomocą pojęć dotyczących występujących w nim typów.

Jako drugi rozpatrzymy klasyczny przykład figur geometrycznych, który mógłby znaleźć zastosowanie w komputerowym systemie wspomagającym projektowanie lub w symulacji gry. Typem podstawowym jest w tym przypadku „figura”. Każda figura posiada wielkość, kolor, położenie itd.; może ona być również narysowana, usunięta, przesunięta, pokolorowana itp. Z typu tego wyprowadzono (za pomocą dziedziczenia) typy poszczególnych figur: okręgu, kwadratu, trójkąta itd. — z których każdy może posiadać własne dodatkowe cechy i sposoby działania. Pewne figury mogą być na przykład odwrócone na drugą stronę. Niektóre działania mogą różnić się między sobą, np. w przypadku gdy chcemy policzyć pole powierzchni figury. Hierarchia typów wyraża zarówno podobieństwa, jak i różnice występujące wśród figur.



Sformułowanie rozwiązania za pomocą pojęć należących do języka problemu jest wyjątkowo korzystne, ponieważ nie wymaga tworzenia szeregu modeli pośrednich, umożliwiających przejście od opisu problemu do opisu rozwiązania. W przypadku

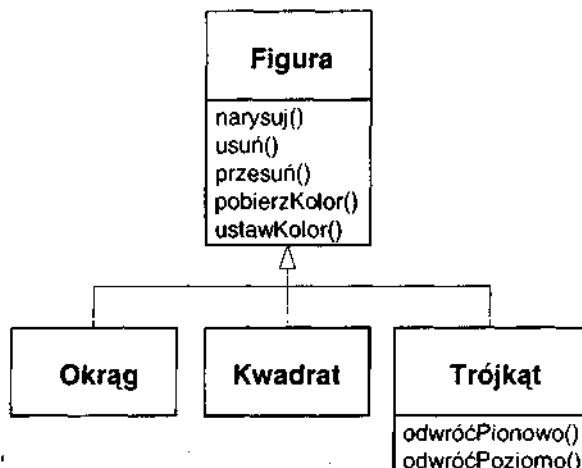
obiektów, hierarchia typów jest modelem podstawowym, co pozwala na bezpośrednie przejście od opisu systemu w rzeczywistym świecie do opisu systemu w postaci kodu.

Okazuje się, że jedną z trudności napotykanych przez osoby zajmujące się projektowaniem obiektowym jest zbyt łatwa droga prowadząca od początku do końca. Często zdarza się, że umysł, przyzwyczajony do poszukiwania skomplikowanych rozwiązań, jest początkowo tą prostotą zaskoczony.

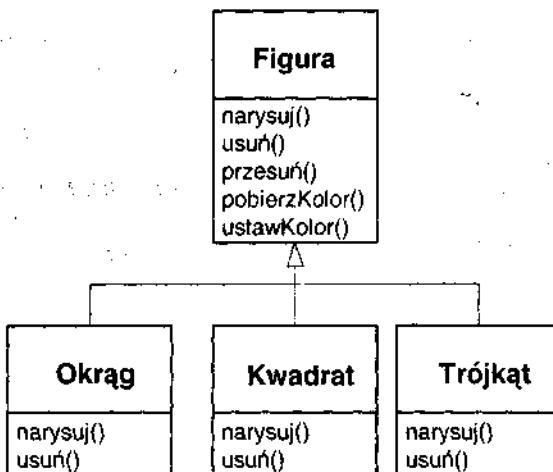
Poprzez dziedziczenie z istniejącego typu tworzymy nowy typ. Zawiera on nie tylko wszystkie składowe klasy podstawowej (mimo że jej składowe **prywatne** są ukryte i niedostępne), ale — co ważniejsze — powiela również interfejs. Oznacza to, że te wszystkie komunikaty, które możemy wysłać do obiektu klasy podstawowej, możemy również przekazać do obiektu klasy pochodnej. Ponieważ typ klasy poznajemy po komunikatach, które możemy do niej wysłać, oznacza to, że klasa pochodna *jest tego samego typu, co klasa podstawowa*. W opisany powyżej przykładzie oznacza to, że „okrąg jest figurą”. Ten rodzaj równoważności, uzyskiwanej poprzez dziedziczenie, jest jednym z kamieni milowych na drodze do zrozumienia znaczenia programowania obiektowego.

Ponieważ zarówno klasa podstawowa, jak i klasa pochodna posiadają taki sam interfejs, istnieje jakaś związana z nim implementacja. Oznacza to, że musi istnieć jakiś kod, wykonywany wówczas, gdy obiekt otrzymuje określony komunikat. Jeżeli utworzymy za pomocą dziedziczenia jakąś klasę i nie zrobimy niczego więcej, to metody klasy podstawowej przejdą w całości do klasy pochodnej. Oznacza to, że w takim przypadku obiekty klasy pochodnej mają nie tylko ten sam typ, co obiekty klasy podstawowej, ale również zachowują się tak samo, co nie jest szczególnie interesujące.

Istnieją dwa sposoby modyfikacji klasy pochodnej w stosunku do jej klasy podstawowej. Pierwszy jest dość prosty — można dodać do klasy pochodnej zupełnie nowe funkcje. Funkcje te nie są częścią interfejsu klasy podstawowej. Oznacza to, że klasa podstawowa po prostu nie wykonała wszystkiego tego, co było nam potrzebne, więc dodaliśmy nowe funkcje. Ten prosty i prymitywny sposób zastosowania dziedziczenia jest czasami najlepszym sposobem rozwiązywania problemu. Jednakże powinniśmy rozważyć dokładniej możliwość, że klasa podstawowa również potrzebuje dodanych przez nas funkcji. Taki właśnie proces iteracyjnego udoskonalania projektu odbywa się regularnie podczas programowania obiektowego.



Mimo że może się czasami wydawać, iż dziedziczenie pociąga za sobą dodawanie do interfejsu nowych funkcji, to nie zawsze jest to zgodne z prawdą. Drugim, i zarazem najważniejszym, sposobem modyfikacji klasy pochodnej jest *zmiana działania istniejącej funkcji klasy podstawowej*. Nosi ona nazwę *zasłaniania* (ang. *overriding*) funkcji.



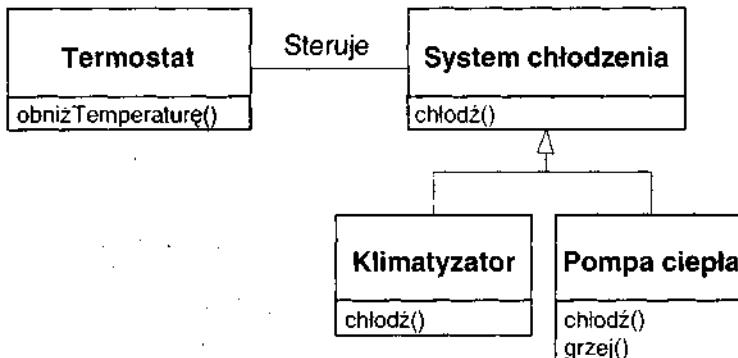
Aby zasłonić funkcję, należy po prostu utworzyć nową definicję tej funkcji w klasie pochodnej. W ten sposób oświadczasz: „używam w tym miejscu tej samej funkcji interfejsu, ale chcę, by w nowym typie robiła ona coś innego”.

Relacje typu „jest” i Jest podobny do”

A oto przedmiot dyskusji, która mogłaby dotyczyć dziedziczenia: czy nie powinno ono zasłaniać *jedynie* funkcji obecnych w klasie podstawowej (nie dodając nowych funkcji składowych, jeżeli nie występują one w klasie podstawowej)? Oznaczałoby to, że klasa pochodna jest *dokładnie* tego samego typu, co klasa podstawowa, ponieważ ma ona identyczny interfejs. W rezultacie można by całkowicie zastąpić obiekt klasy podstawowej obiektem klasy pochodnej. Można uznać taki przypadek za *czyste zastępowanie* — odwołujemy się do niego często jako do *zasady zastępowania* (ang. *substitution principle*). Jest to, w pewnym sensie, idealny sposób traktowania dziedziczenia. W takich przypadkach określamy często relację pomiędzy klasą pochodną i klasą podstawową jako relację typu *jest*, ponieważ możemy wówczas stwierdzić np. „*okrąg jest figurą*”. Sprawdzianem rozumienia dziedziczenia jest umiejętność opisania w logiczny sposób relacji typu „*jest*” pomiędzy klasami.

Zdarzają się jednak przypadki, w których konieczne jest dodanie do typu pochodnego nowych elementów interfejsu, powodując tym samym rozszerzenie tego interfejsu oraz utworzenie nowego typu. Typ podstawowy może być nadal zastąpiony typem pochodnym, ale zamiana ta nie jest już tak doskonała, ponieważ nowe funkcje nie są dostępne z poziomu typu podstawowego. Można tę sytuację opisać za pomocą relacji *jest podobny do* — nowy typ posiada interfejs starego typu, lecz zawiera również inne funkcje; nie możemy zatem powiedzieć, że typy te są dokładnie takie same. Jako przykład rozważmy klimatyzator. Założymy, że twój dom jest wyposażony w instalację

zawierającą wszystkie urządzenia niezbędne do sterowania chłodzeniem — czyli posiada umożliwiający to interfejs. Wyobraź sobie, że klimatyzator zepsuł się i zastępujesz go pompą ciepła — urządzeniem, które może zarówno ogrzewać, jak i chłodzić. Pompę ciepłą jest podobno klimatyzatora, ale zapewnia więcej możliwości. Ponieważ instalacja w twoim domu została zaprojektowana wyłącznie w celu sterowania chłodzeniem, może się ona komunikować jedynie z chłodzącą częścią nowego urządzenia. Interfejs nowego obiektu został rozbudowany, lecz istniejący system nie zna niczego poza interfejsem oryginalnym.



Oczywiście, po przyjrzeniu się projektowi stanie się jasne, że klasa podstawowa „system chłodzenia” nie jest dostatecznie ogólna i powinna zostać przemianowana na „system sterowania temperaturą” — w taki sposób, by obejmowała również ogrzewanie, co umożliwi funkcjonowanie zasady zastępowania. Jednakże powyższy diagram ilustruje to, co może zdarzyć się zarówno w czasie projektowania, jak i w rzeczywistym świecie.

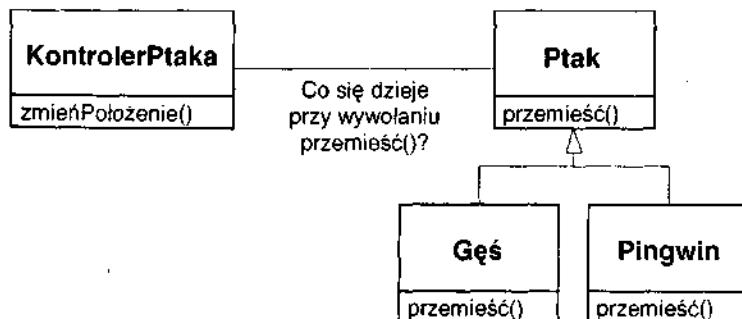
Po zapoznaniu się z zasadą zastępowania łatwo jest odnieść wrażenie, że takie podejście (czyste zastępowanie) jest jedynym sposobem postępowania oraz że projekt powinien być zbudowany w taki właśnie sposób. Przekonamy się jednak, że zdarzają się sytuacje, w których zachodzi konieczność dodania nowych funkcji do interfejsu klasy pochodnej. Po bliższym przyjrzeniu się oba powyższe przypadki powinny stać się dostatecznie oczywiste.

Zastępowanie obiektów przy użyciu polimorfizmu

Przy posługiwaniu się hierarchią typów często zdarza się, że chcemy traktować obiekt w taki sposób, jakby nie był on obiektem jakiegoś szczególnego typu, ale typu podstawowego. Pozwala to na napisanie kodu, który nie będzie zależny od konkretnych typów. W przykładzie dotyczącym figur funkcje operują na dowolnych figurach geometrycznych, niezależnie od tego, czy są one okręgami, kwadratami, trójkątami itd. Wszystkie figury mogą zostać narysowane, usunięte i przesunięte, więc funkcje te przesyłają po prostu komunikat do obiektu „figura”, nie uwzględniając tego, w jaki sposób obiekt ten komunikat obsłuży.

Na napisany w taki sposób kod nie ma wpływu dodawanie nowych typów, będące najczęstszym sposobem rozbudowy programu obiektowego w celu dostosowania go do obsługi nowych sytuacji. Na przykład możemy wprowadzić nowy podtyp figury — „pięciokąt” — nie modyfikując funkcji mających do czynienia jedynie z „ogólną” postacią figur. Możliwość łatwego rozszerzania programu poprzez wprowadzanie nowych typów jest istotna, ponieważ znacznie podnosi ona jakość projektów, obniżając zarazem koszt pielęgnacji oprogramowania.

Pojawia się jednak problem związany z próbą traktowania obiektów typu pochodnego jako obiektów ich typu podstawowego (okrągów jako figur, rowerów jako pojazdów, kormoranów jako ptaków itd.). Jeżeli funkcja zamierza polecić „ogólnej” figurze, by się narysowała, „ogólnemu” pojazdowi, by się przemieścił albo „ogólnemu” ptakowi, by się poruszył, to kompilator nie może w czasie komplikacji programu dokładnie wiedzieć, który fragment kodu zostanie wykonany. To właśnie stanowi istotę — kiedy wysyłany jest komunikat, programista *nie chce* wiedzieć, który fragment kodu zostanie wykonany. Funkcja rysowania może być bowiem zastosowana zarówno do okręgu, jak i kwadratu lub trójkąta, a obiekt wykona właściwy kod, w zależności od swojego konkretnego typu. Jeżeli nie musimy wiedzieć, który fragment kodu zostanie wykonany, to po dodaniu nowego podtypu wykonywany przez niego kod może być inny, bez konieczności dokonywania zmian w wywołaniu funkcji. Jak zatem działa kompilator, nie wiedząc dokładnie, który fragment kodu jest wykonywany? Na przykład na przedstawionym poniżej diagramie obiekt **KontrolerPtaka** pracuje po prostu z „ogólnymi” obiektami typu **Ptak**, nie znając ich typów. Jest to wygodne z punktu widzenia obiektu **KontrolerPtaka**, ponieważ nie potrzebuje on żadnego specjalnego kodu, który określałby jakiego dokładnie typu jest **Ptak**, z którym współpracuje, i w jaki sposób się ten **Ptak** zachowuje. Dlaczego zatem, gdy wywoływana jest funkcja **przemieść()**, pomimo nieznajomości konkretnego typu obiektu **Ptak**, podejmowane jest właściwe działanie (**Gęś** biegnie, leci lub płynie, **a Pingwin** biegnie lub płynie)?



Odpowiedź stanowi zasadniczy zwrot związany z programowaniem obiektowym — kompilator nie może wykonać wywołania funkcji w tradycyjny sposób. Wywołanie funkcji, wygenerowane przez kompilator języka niebędącego językiem obiektowym, powoduje tzw. *wczesne wiązanie* (ang. *early binding*). Pojęcie to niejest znane tym wszystkim, którzy traktują wywołania funkcji jedynie w tradycyjny sposób. Oznacza ono, że kompilator generuje wywołanie funkcji o pewnej nazwie, a program łączący (ang. *linker*) zamienia je w określony adres bezwzględny kodu, który ma zostać wywołany. W programowaniu obiektowym program nie może określić adresu kodu, dopóki nie zostanie uruchomiony, a zatem potrzebny jest jakiś inny mechanizm, umożliwiający wysłanie komunikatu do „ogólnego” obiektu.

Do rozwiązania tego problemu w językach obiektowych jest wykorzystana koncepcja *późnego wiązania* (ang. *late binding*). Kiedy do obiektu wysyłany jest komunikat, wywoływany kod nie jest określony aż do momentu uruchomienia programu. Kompilator gwarantuje, że funkcja istnieje, dokonując kontroli typowej argumentów oraz zwracanej wartości (języki, w których nie ma to miejsca, są nazywane językami o *słabej kontroli typów*), nie wie jednak, jaki dokładnie kod zostanie wykonany.

W celu przeprowadzenia późnego wiązania kompilator C++ wstawia — zamiast wywołania bezwzględnego — specjalny kod. Kod ten wyznacza adres ciała funkcji, wykorzystując informację zapisaną wewnątrz obiektu (proces ten został szczegółowo opisany w rozdziale 15.). A zatem każdy obiekt może zachowywać się różnie, zależnie od treści zawartego w nim specjalnego kodu. Po wysłaniu do obiektu komunikatu obiekt „domyśla się”, co należy z nim zrobić.

Aby określić, że funkcja ma posiadać elastyczność związaną z późnym wiązaniem, należy użyć słowa kluczowego **virtual** (wirtualny). Nie trzeba rozumieć działania funkcji wirtualnych, by ich używać, ale nie sposób bez tego programować obiektywne w C++. W języku C++ należy pamiętać o dodaniu słowa kluczowego **virtual**, ponieważ domyślnie funkcje składowe *nie są* dynamicznie wiązane. Funkcje wirtualne pozwalają na wyrażenie różnic wdziałaniu klas należących do jednej rodziny. Różnice te są przyczyną zachowania polimorficznego.

Rozważmy przypadek figur geometrycznych. Rodzina klas (z których wszystkie bazują na tym samym, jednolitym interfejsie) została przedstawiona w początkowej części rozdziału. Aby zademonstrować polimorfizm, napiszemy kod, który będzie ignorował szczególne związane z konkretnym typem, odwołując się wyłącznie do klasy podstawowej. Kod ten zostanie *oddzielony* od informacji dotyczącej konkretnego typu, dzięki czemu będzie łatwiejszy do napisania i do zrozumienia. Ponadto jeżeli za pomocą metody dziedziczenia do typu **Kształt** zostanie dodany nowy typ — np. **Sześciokąt** — to napisany kod będzie działał w nim również dobrze jak w przypadku już istniejących typów. Dzięki temu można *rozbudowywać* istniejący program.

Napiszemy w języku C++ funkcję (wkrótce dowiesz się, jak to zrobić):

```
void zróbCoś(Figura& f) {  
    f.usuń();  
    // ...  
    f.narysuj();  
}
```

Funkcja ta odwołuje się do dowolnego obiektu typu **Figura**, jest więc niezależna od konkretnego typu obiektu, który jest rysowany i usuwany (znak „&” oznacza „pobierz adres obiektu przekazanego funkcji **zróbCoś()**”, ale nie jest jeszcze konieczne rozumienie dokładnie wszystkich szczegółów). Jeżeli w *jakiejś* innej części programu zostanie użyta funkcja **zróbCoś()**:

```
Okrąg o;  
Trójkąt t;  
Linia l;  
zróbCoś(o);  
zróbCoś(t);  
zróbCoś(l);
```

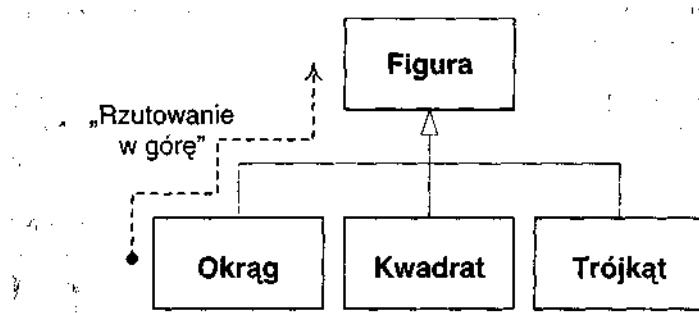
Wywołania funkcji **zróbCoś()** automatycznie działają poprawnie, niezależnie od konkretnego typu obiektu.

W istocie to zdumiewająca sztuczka. Weźmy pod uwagę wiersz:

```
zróbCoś(o);
```

W powyższym przypadku obiekt **Okrąg** został przekazany funkcji oczekującej obiektu typu **Figura**. Ponieważ **Okrąg** jest **Figurą**, to może być traktowany przez funkcję **zróbCoś()** jako obiekt typu **Figura**. Oznacza to, że **Okrąg** akceptuje wszystkie komunikaty, jakie funkcja **zróbCoś()** może wysyłać do obiektu **Figura**. Jest to zupełnie bezpieczne i całkowicie logiczne.

Proces traktowania klasy pochodnej w taki sposób, jakby był on typem podstawowym, nazywamy *rzutowaniem w górę* (ang. *upcasting*). Termin *rzutowanie* został użyty w znaczeniu „przenoszenia na jakąś płaszczyznę” (lub „dopasowania się do pewnej formy”), natomiast „w górę” pochodzi ze sposobu, w jaki zazwyczaj jest ułożony diagram dziedziczenia — z typem podstawowym na górze i klasami pochodnymi rozwijającymi się poniżej. A zatem rzutowanie w kierunku typu podstawowego przesuwa się w diagramie dziedziczenia do góry i stąd nosi nazwę rzutowania w górę.



Programy obiektowe zawierają jakieś rzutowania w górę, ponieważ jest to sposób, w jaki uniezależniamy się od precyzyjnej wiedzy na temat typu, z którym pracujemy. Przyjrzyjmy się kodowi funkcji **zróbCoś()**:

```
f.usuń();
// ...
f.narysuj();
```

Zauważmy, że powyższy kod nie zawiera poleceń: „jeżeli jesteś typu **Okrąg**, zrób to, a jeżeli jesteś typu **Kwadrat**, zrób tamto itd.”. W przypadku gdy piszemy kod sprawdzający wszystkie możliwe typy, którymi może być w danej chwili **Figura**, staje się on nieelegancki i musimy zmieniać go za każdym razem, gdy dodajemy nowy rodzaj obiektu **Figura**. W naszym przykładzie stwierdzamy natomiast: „jestes figurą i wiem, że potrafisz wykonać funkcje **usuń()** i **narysuj()**, więc zrób to, pamiętając o wszystkich szczegółach”.

Tym, co robi wrażenie w kodzie funkcji **zródCoś()**, jest fakt, że w jakiś sposób działa on właśnie tak, jak powinien. Wywołanie funkcji **rysuj()** w stosunku do okręgu powoduje wykonanie innego kodu niż wywołanie jej w stosunku do kwadratu czy linii.

... lecz gdy komunikat **rysuj()** jest wysyłany do anonimowego obiektu typu **Figura**, to podejmowane jest działanie właściwe dla tego rzeczywistego typu. To zdumiewające, ponieważ, jak wspomniano wcześniej, w czasie komplikacji kodu funkcji **zróbCoś()** kompilator języka C++ nie może wiedzieć dokładnie, z jakimi typami ma do czynienia. A zatem spodziewalibyśmy się wywołania funkcji **usuń()** i **narysuj()** typu **Figura**, a nie funkcji zdefiniowanych dla specyficznych typów: **Okrąg**, **Kwadrat** czy **Linia**. Jednakże dzięki polimorfizmowi dzieje się to, co powinno. Szczegółami zajęły się kompilator oraz system obsługujący wykonywanie programu — wystarczy tylko wiedzieć, że tak się dzieje i, co ważniejsze, umieć wykorzystać ten fakt podczas projektowania. Jeżeli funkcja składowa jest funkcją wirtualną (oznaczoną specyfikatorem **virtual**), to po wysłaniu obiektovi komunikatu wykona on właściwą czynność, nawet wówczas, gdy zastosowane będzie rzutowanie w góre.

Tworzenie i niszczenie obiektów

Z technicznego punktu widzenia do dziedziny programowania obiektowego należą: tworzenie abstrakcyjnych typów danych, dziedziczenie i polimorfizm. Występują w niej jednak również inne, nie mniej ważne kwestie. W niniejszym podrozdziale przedstawiono ich przegląd.

Szczególnie ważny jest sposób, w jaki obiekty są tworzone i niszczone. Gdzie przechowywane są dane obiektów i w jaki sposób kontrolowany jest czas ich życia? W różnych językach programowania przyjmuje się w takich przypadkach odmienne założenia. W języku C++, w którym najistotniejszą kwestią jest kontrola efektywności, programista ma możliwość wyboru. W celu osiągnięcia maksymalnej szybkości wykonania sposób przechowywania danych oraz czas życia mogą być określone w czasie pisania programu przez umieszczenie obiektów ria stosie lub w obszarze danych statycznych. Stos jest obszarem pamięci, używanym bezpośrednio przez mikroprocesor do przechowywania danych w czasie wykonywania programu. Zmienne znajdujące się na stosie nazywane są czasami zmiennymi *automatycznymi* lub *lokalnymi*. Obszar danych statycznych jest z kolei ustalonym fragmentem pamięci, przydzielonym przed rozpoczęciem pracy programu. Używanie stosu lub obszaru danych statycznych oznacza położenie nacisku na szybkość przydzielania i zwalniania pamięci, która może być pozytyczna w niektórych sytuacjach. Jednakże tracisz w ten sposób elastyczność, ponieważ musisz w czasie pisania programu znać dokładnie liczbę, czas życia oraz typ obiektów. Jeżeli próbujesz rozwiązać bardziej ogólny problem, taki jak np. projektowanie wspomagane komputerowo, program zarządzający magazynem lub kontrolą ruchu powietrznego, stanowi to nadmierne ograniczenie.

Drugim podejściem jest dynamiczne tworzenie obiektów w obszarze pamięci zwany *stertą* (ang. *heap*). W ujęciu tym liczba potrzebnych obiektów, ich „czas życia” oraz dokładne typy nie są znane aż do uruchomienia programu. Decyzje te są podejmowane spontanicznie, w czasie wykonywania programu. Jeżeli potrzebujesz nowego obiektu, to po prostu tworzysz go na stercie, używając do tego słowa kluczowego **new**. Kiedy zajmowana przez niego pamięć nie jest ci już potrzebna, musisz zwolnić ją za pomocą słowa kluczowego **delete**.

Ponieważ zarządzanie pamięcią odbywa się dynamicznie, w czasie wykonywania programu, czas potrzebny do przydzielenia pamięci na stercie jest znacznie większy niż w przypadku utworzenia obszaru pamięci na stosie (które jest często pojedynczą instrukcją mikroprocesora, polegającą na przesunięciu wskaźnika stosu w dół, oraz drugą, przemieszczającą go z powrotem w górę). W podejściu dynamicznym przyjmuje się, logiczne zazwyczaj, założenie, że obiekty mają tendencję do bycia złożonymi, a więc dodatkowy narzut, polegający na znalezieniu wolnej pamięci oraz jej zwolnieniu, nie będzie miał istotnego wpływu na tworzenie obiektów. Ponadto większa elastyczność jest niezbędna w przypadku rozwiązywania ogólnych problemów programistycznych.

Jednakże pozostaje jeszcze kwestia czasu życia obiektów. Jeżeli obiekt tworzony jest na stosie lub w obszarze danych statycznych, to kompilator określa, jak długo powinien on istnieć i może automatycznie go zniszczyć. Jeżeli jednak obiekt zostanie utworzony na stercie, kompilator nie posiada informacji na temat jego czasu życia. W języku C++ musimy określić za pomocą metod programowych, kiedy obiekt powinien zostać zniszczony, a następnie dokonać tego, używając słowa kluczowego **delete**. Alternatywnie, środowisko może posiadać mechanizm zwany *zbieraczem śmieci* (ang. *garbage collector*), automatycznie rozpoznającym nieużywane już obiekty, a następnie niszącym. Oczywiście, pisanie programów używających zbieracza śmieci jest znacznie wygodniejsze, wymaga jednak, by wszystkie tolerowały jego obecność oraz narzut związany z odśmiecaniem. Ponieważ nie jest to zgodne z wymaganiami projektowymi języka C++, zbieracz śmieci nie został do niego włączony, chociaż dostępne są wersje przeznaczone dla C++, dostarczane przez niezależne firmy.

Obsługa wyjątków — sposób traktowania błędów

Od początku istnienia języków programowania obsługa błędów należała do najtrudniejszych zagadnień. Ponieważ trudno jest zaprojektować dobry system obsługi błędów, wiele języków programowania po prostu ignoruje ten temat, przerzucając problem na projektantów bibliotek. Ci ostatni tworzą półśrodkie, sprawdzające się w wielu sytuacjach, ale łatwe do ominięcia — zazwyczaj można je po prostu zignorować. Zasadniczym problemem związanym z większością systemów obsługi błędów jest fakt, że polegają one na czujności programisty, wyrażającej się w postępowaniu zgodnym z ustaloną konwencją, która niejest wspierana przez język. Jeżeli programista nie zachowa czujności, co się często zdarza, gdy się spiesz, może łatwo o takim systemie zapomnieć.

Obsługa wyjątków (ang. *exception handling*) wiąże obsługę błędów bezpośrednio z językiem programowania, a czasami nawet z systemem operacyjnym. Wyjątek jest obiektem „zgłoszonym” (lub „wywrzuconym”) w miejscu, w którym wystąpił błąd. Następnie błąd ten może być „wyłapany” przez odpowiednią *procedurę obsługi wyjątku* (ang. *exception handler*), przeznaczoną do obsługi odpowiadającego mu rodzaju błędu. Funkcjonuje to w taki sposób, jak gdyby obsługa wyjątków była oddzielną, równoległą ścieżką wykonywania programu, która może być wykorzystana w razie

pojawienia się usterek. Ponieważ obsługa wyjątków wykorzystuje oddzielną ścieżkę wykonywania programu, nie musi ona ingerować w kod, odpowiadający jego normalnej pracy. Dzięki temu jest on łatwiejszy do napisania, nie musimy bowiem bez przerwy sprawdzać, czy nie wystąpiły jakieś błędy. Ponadto zgłoszony wyjątek nie przypomina kodu błędu zwróconego przez funkcję ani znacznika ustawionego przez nią w celu zasygnalizowania powstania błędu, które mogą być po prostu zignorowane. Wyjątku nie można zignorować — istnieje więc gwarancja, że w którymś miejscu zostanie on obsłużony. Wyjątki zapewniają wreszcie niezawodny sposób wychodzenia z kryzysów. Zamiast zakończyć działanie programu, jesteśmy często w stanie dokonać naprawy i przywrócić działanie programu, co umożliwia tworzenie znacznie bardziej niezawodnych systemów.

Warto nadmienić, że mechanizm obsługi wyjątków nie jest wyłączną cechą języków obiektowych, mimo że wyjątki są w nich zazwyczaj reprezentowane w postaci obiektów. Istniał on jeszcze przed powstaniem języków obiektowych.

W tym tomie książki obsługa wyjątków jest przedstawiona i używana jedynie w powierzchniowy sposób. Dokładny opis wyjątków znajduje się w tomie drugim (dostępnym w witrynie: <http://helion.pl/online/thinking/index.html>).

Analiza i projektowanie

Model obiektowy stanowi nowy, odmienny sposób myślenia o programowaniu i wiele osób ma początkowo kłopoty z rozpoczęciem pracy nad projektem obiektowym. Kiedy jednak przekonasz się, że niemal wszystko jest obiektem i nauczysz się myśleć w sposób bardziej „obiektowy”, będziesz mógł przystąpić do tworzenia „dobrych” projektów, wykorzystujących wszystkie zalety programowania obiektowego.

Metoda (zwana często *metodyką*) jest zbiorem procedur i heurystyk używanych do zmniejszenia złożoności problemu programistycznego. Wiele spośród metod programowania obiektowego zostało sformułowanych, zanim jeszcze powstało programowanie obiektowe. W niniejszym podrozdziale przedstawiono korzyści, które można osiągnąć, używając metodyki.

w dziedzinie metodyki dokonuje się wielu eksperymentów — szczególnie w programowaniu obiektowym — istotne więc jest zrozumienie, jaki problem próbuje rozwiązać dana metoda, zanim jeszcze zacznie się brać pod uwagę jej zastosowanie. Odnosi się to w szczególności do C++, gdyż sam język został opracowany z myślą o zmniejszeniu złożoności (w porównaniu z językiem C) związanej z wyrażaniem programu. Dzięki temu możliwe jest ograniczenie konieczności tworzenia coraz bardziej złożonych metodyk. Zamiast nich, o wiele prostsze metodyki mogą się okazać wystarczające do rozwiązania w języku C++ znacznie liczniejszej klasy problemów niż ta, z którą można sobie poradzić, używając prostych metodyk w językach proceduralnych.

Istotne jest również zrozumienie, że termin „metodyka” jest często używany nieco na wyrost. Wszystko, co czynisz obecnie, projektując i pisząc programy, jest metodą. Może to być Twoja własna metoda, której stosowania możesz nie być świadomym, lecz

jest to proces, przez który przechodzisz w czasie tworzenia. Jeżeli jest on efektywny, to wykorzystanie go w C++ może wymagać tylko nieznaczych poprawek. Jeżeli natomiast nie jesteś zadowolony z wydajności swojej pracy oraz sposobu, w jaki powstają Twoje programy, to możesz rozważyć przyjęcie jakiejś formalnej metody postępowania lub wybrać elementy, należące do wielu metod formalnych.

W trakcie procesu projektowania najważniejsze jest to, aby się nie zagubić. Można bowiem cel łatwo osiągnąć. Większość metod analizy i projektowania została opracowana z myślą o rozwiązywaniu najistotniejszych problemów. Należy pamiętać, że większość projektów nie należy do tej kategorii, można więc na ogół z powodzeniem przeprowadzić analizę i wykonać projekt, wykorzystując jedynie względnie mały podzbiór zaleceń związanych z daną metodą⁷. Jednak niektóre sposoby postępowania, niezależnie od tego, jak będą ograniczone, pozwolą Ci na pracę w znacznie lepszym stylu niż gdybyś po prostu rozpoczęł kodowanie.

Łatwo jest również utknąć, wpadając w pułapkę „analytycznego paraliżu”. Odnosisz wówczas wrażenie, że nie możesz posunąć się naprzód, ponieważ nie zostały ukończone wszystkie detale dotyczące bieżącego etapu. Pamiętaj, że bez względu na to, ile wykonaš analiz, są takie kwestie dotyczące systemu, które nie ujawnią się aż do rozpoczęcia projektowania i jeszcze liczniejsze takie, które pozostaną niewidoczne aż do rozpoczęcia kodowania albo nawet do chwili, gdy program będzie już ukończony i uruchomiony. Z tego powodu kluczową kwestią jest stosunkowo szybkie przejście etapu analizy i projektowania, a następnie implementacja testów zaproponowanego systemu.

Ten punkt jest warty podkreślenia. Z uwagi na doświadczenia językami proceduralnymi, godne pochwały jest to, że zespół będzie chciał postępować ostrożnie, rozumiejąc każdy najdrobniejszy szczegół, zanim przystąpi do projektowania i implementacji. Z pewnością podczas tworzenia systemu zarządzania bazą danych warto zrozumieć dokładnie potrzeby klienta. Jednakże zarządzanie bazą danych należy do klasy problemów dobrze postawionych i gruntownie poznanych — w wielu takich programach problemem do rozwiązywania jest struktura bazy danych. Klasa problemów programistycznych omawianych w bieżącym rozdziale należy do kategorii „nieprzewidywalne”⁸; w ich przypadku rozwiązywanie nie polega na prostym przekształceniu jakiegoś dobrze znanego rozwiązania. Zawiera natomiast jeden lub więcej „nieprzewidywalnych czynników” — elementów, które nie posiadają dobrze poznanych wcześniejszych rozwiązań i dla których niezbędne jest przeprowadzenie badań⁹. Próba gruntowej analizy nieprzewidywalnego problemu, dokonana przed przystąpieniem do projektowania i implementacji, kończy się analitycznym paraliżem z uwagi

⁷ Doskonałym tego przykładem jest książka Martina Fowlera *UML Distilled* (Addison-Wesley, 2000), sprowadzająca, niejednokrotnie przytaczając, metody języka UML, do dającego się ogarnąć podzbioru.

⁸ W oryginale występuje w tym miejscu wprowadzony przez autora termin „wild-card”, oznaczający rzecz, której własności są nieznane lub nieprzewidywalne — przyp. tłum.

⁹ Moja wskazówka dotycząca szacowania takich projektów jest następująca: „Jeżeli projekt zawiera więcej niż jeden element nieprzewidywalny, nie próbuj nawet planować, ile czasu zabierze jego wykonanie ani ile będzie ono kosztowało, dopóki nie stworzysz jednodziałającego prototypu. Występuje w nim zbyt wiele stopni swobody”.

na brak informacji pozwalających na rozwiązywanie tego rodzaju problemu w fazie analizy. Do jego rozwiązania konieczne jest powtarzanie całego cyklu, które z kolei wymaga podejmowania ryzykownych działań (co jest uzasadnione, bowiem próbujemy zrobić coś nowego i potencjalne profity mają większą wartość). Wydaje się, że ryzykowne jest „pośpieszne” przystąpienie do wstępnej implementacji, ale może ona zmniejszyć zagrożenie nieprzewidywalnego projektu, dając sposobność wczesnego przekonania się o tym, czy przyjęte założenia są możliwe do zrealizowania. Opracowywanie produktów jest zarządzaniem ryzykiem.

Często proponuje się, by „stworzyć coś z góry przeznaczonego do wyrzucenia”. W czasie programowania obiektowego można nadal usunąć część kodu, ale ponieważ jest on zamknięty wewnątrz klas, już za pierwszym podejściem powstanie z pewnością kilka użytecznych projektów klas i opracowanych zostanie parę wartościowych pomysłów, dotyczących projektu systemu, których nie trzeba będzie wyrzucać. Tak więc pierwsze, pobiczne ujęcie problemu nie tylko dostarcza istotnych informacji, przydatnych dla kolejnych cykli analizy, projektowania i implementacji, ale w rezultacie zostaje również utworzony kod, stanowiący dla nich podstawę.

Jak już wspomniano, nawet mimo wykorzystania metodyki zawierającej ogromną liczbę szczegółów, zaleczącą wykonanie wielu kroków oraz sporządzenie wielu dokumentów, nadal trudno jest określić, na czym należy poprzestać. Trzeba więc pamiętać o tym, czego próbujemy się dowiedzieć:

1. Z jakimi obiekttami mamy do czynienia (wjaki sposób podzielić projekt na części składowe)?
2. Jakie są ich interfejsy (jakie komunikaty trzeba wysyłać do każdego z obiektów)?

Nawet jeżeli utworzysz tylko obiekty oraz ich interfejsy, to i tak będziesz już mógł napisać program. Z rozmaitych powodów możesz potrzebować większej liczby opisów oraz dokumentów niż te, które zostały przedstawione, ale na pewno nie poradzisz sobie z mniejszą ich liczbą.

Proces można przeprowadzić w pięciu etapach, przy czym etap zerowy będzie stanowił tylko wstępную zgodę na pewien sposób organizacji pracy.

Etap 0. Przygotuj plan

Najpierw należy zdecydować, z jakich etapów będzie składała się praca. Wydaje się to oczywiste (podobnie jak *wszystko*, co zostało tu przedstawione), a mimo to często nie podejmuje się tej decyzji przed przystąpieniem do kodowania. Być może twój zamiar polega na tym, aby „przystąpić od razu do kodowania” (w przypadku dobrego poznania problemu jest to właściwe podejście). W porządku — ale przynajmniej przyjmij, że tak jest właśnie twój plan.

Na tym etapie możesz również dojść do wniosku, że niezbędna jest jakaś dodatkowa forma organizacji pracy, ale nie decyduj jeszcze o wszystkim. Jest zrozumiałe, że niektórzy programiści preferują pracę „na luzie”, w trybie nienarzucającym żadnych rozwiązań organizacyjnych — „będzie gotowe, jak skończę”. Przez pewien

czas może to się wydawać atrakcyjne, ale odkryłem, że wyznaczenie kilku „kamieni milowych” pomaga skupić się i zintensyfikować wysiłki związane z ich realizacją, zamiast tkwić w dążeniu do realizacji jedynego celu: „ukończenia projektu”. Ponadto powoduje to podział projektu na łatwiejsze fragmenty, dzięki czemu sprawia on wrażenie przystępniejszego (a poza tym kamienie milowe dają więcej okazji do świętowania).

Kiedy rozpoczynałem naukę budowy opowiadań (dzięki czemu napiszę kiedyś powieść), byłem przeciwny idei konstrukcji; pisząc po prostu przelewałem swoje myśli na papier. Później jednak zrozumiałem, że kiedy piszę na temat komputerów, konstrukcja jest wystarczająco przejrzysta, abym nie musiał się nad nią zastanawiać. Wciąż jednak organizuję swoją pracę, choć robię to jedynie podświadomie. Zatem jeżeli wydaje ci się, że twój plan polega jedynie na tym, aby rozpocząć kodowanie, nadal w pewien sposób pokonujesz kolejne etapy, zadając sobie pewne pytania i odpowiadając na nie.

Misja

Każdy tworzony system, niezależnie od tego, jak jest skomplikowany, mająkieś podstawowe zastosowanie, zadanie, służy zaspokojeniu jakiejś potrzeby. Abstrahując od interfejsu użytkownika, szczegółów sprzętowych i systemowych, zakodowanych algorytmów oraz problemów efektywności, odnajdziesz prostąraczejego istnienia. Podobnie jak ogólną koncepcję hollywoodzkiego filmu, możesz opisać ją w kilku zdaniach. Ten czysty opis stanowi punkt wyjścia.

Ogólna koncepcja jest bardzo ważna, bowiem nadaje ton twojemu projektowi — jest jego misją. Nie musisz od razu uchwycić całej idei (możesz realizować kolejne etapy projektu, zanim stanie się ona zupełnie oczywista), ale podejmuj próby ją zrozumienia, dopóki nie dotrzesz do sedna. Na przykład analizując system kontroli ruchu powietrznego możesz zacząć od ogólnej koncepcji, koncentrując się na konstruowanym systemie — „program wieże śledzi ruch samolotu”. Rozważ jednak, co się stanie, kiedy ograniczysz system do bardzo małego lotniska — prawdopodobnie to człowiek jest na nim kontrolerem ruchu powietrznego lub nie ma ono w ogóle żadnego kontrolera. Bardziej użyteczny model będzie nie tyle dotyczył tworzonego rozwiązania, ile opisywał sam problem: „samolot przylatuje, jest rozładowywany, naprawiany, ponownie ładowany i odlatuje”.

Etap 1. Co tworzymy?

W poprzedniej generacji metod projektowania programów (nazywanej *projektowaniem proceduralnym*) etap ten nazywano „tworzeniem analizy wymagań i specyfikacji systemu”. Łatwo się w tym pogubić — dokumenty o groźnie brzmiących nazwach, które mogły rozrastać się do rozmiarów dużych projektów, rządzących się własnymi prawami. Jednakże intencje były dobre. Analiza wymagań zakłada: „przygotuj listę wskazówek, dzięki którym dowiemy się, kiedy praca zostanie ukończona, a wymagania klienta zaspokojone”. Specyfikacja systemu głosi natomiast: „oto opis tego, co program będzie robił (nie *w jaki sposób*), aby sprostać postawionym wymaganiom”. Analiza wymagań jest w rzeczywistości umową pomiędzy

tobą a klientem (nawet jeżeli klient ten pracuje w twojej firmie albo jest on jakimś innym obiektem lub systemem). Specyfikacja systemu jest „perspektywicznym spojrzeniem na problem” oraz, w pewnym sensie, sposobem ustalenia, czy można go zrealizować i ile zajmie to czasu. Ponieważ obie te kwestie wymagają uzgodnień pomiędzy ludźmi (i zazwyczaj zmieniają się one w trakcie realizacji projektu), dla oszczędności czasu najlepiej zachować je w najprostszej możliwej formie – w idealnym przypadku w postaci list i prostych diagramów. Być może istnieją jakieś dodatkowe uwarunkowania, wymagające sporządzenia obszerniejszych dokumentów, ale dzięki utrzymaniu pierwotnej dokumentacji w krótkiej i zwięzłej postaci możliwe jest jej utworzenie w ciągu kilku sesji burz mózgów – przez prowadzącego spotkanie, tworzącego opracowanie na bieżąco. Nie tylko umożliwia to uwzględnienie wszystkich opinii, ale pozwala również nałączenie się do pracy wszystkich członków zespołu jednocześnie, a także sprzyja wzajemnemu zrozumieniu między nimi. I, co być może najważniejsze, pozwala rozpoczęć projekt z dużą entuzjazmem.

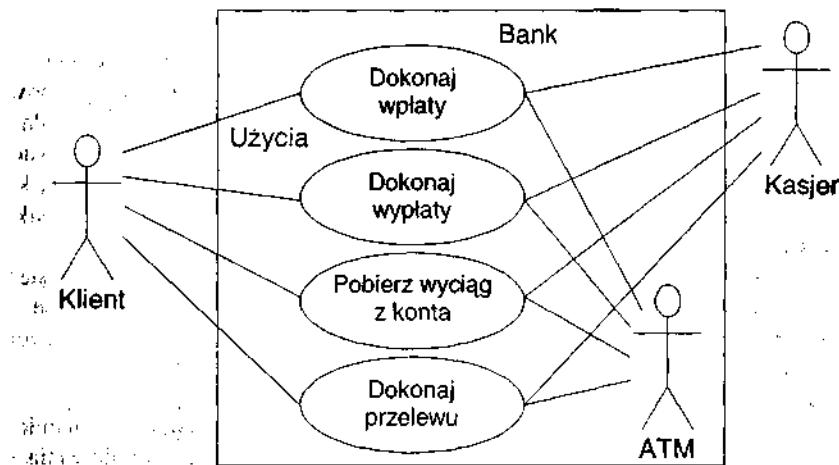
Konieczne jest skupienie uwagi na istocie tego, co próbujemy osiągnąć w bieżącym etapie: określeniu, jak powinien zadziałać system. Najbardziej przydatnym do tego narzędziem jest zbiór tzw. „przypadków użycia” (ang. *use cases*). Przypadki użycia identyfikują kluczowe cechy systemu, ujawniając niektóre z podstawowych klas, które zostaną użyte. Są one głównie opisowymi odpowiedziami na pytania w rodzaju¹⁰:

- 4 „Kto będzie używał systemu?”
- 4 „Co aktorzy mogą robić z systemem?”
- ◆ „W jaki sposób dany aktor wykonuje w systemie daną czynność?”
- ◆ „Jak inaczej mogłaby działać dana operacja, gdyby była wykonywana przez kogoś innego lub gdyby ten sam aktor miał odmienny cel (odkrywanie wariacji)?”
- ◆ „Jakie problemy mogą się pojawić podczas wykonywania w systemie danej operacji (odkrywanie wyjątków)?”

Jeżeli, na przykład, projektujesz bankomat, to przypadek użycia dla jakiegoś konkretnego aspektu funkcjonowania systemu może opisywać, co robi bankomat w każdej możliwej sytuacji. Każda z tych „sytuacji” jest nazywana *scenariuszem*, natomiast każdy przypadek użycia może być traktowany jako zbiór scenariuszy. Można ująć scenariusz jako pytanie rozpoczynające się słowami: „co zrobí system, jeżeli...?”. Na przykład, „co zrobí bankomat, jeżeli klient w ciągu ostatnich 24 godzin zdeponował czek, a środki znajdujące się jego rachunku, przed uwzględnieniem tego czeku, nie wystarczają do wypłacenia żądanej kwoty?”.

Diagramy przypadków użycia są celowo proste po to, by zapobiec przedwczesnemu ugrzęźnięciu w szczegółach implementacji systemu:

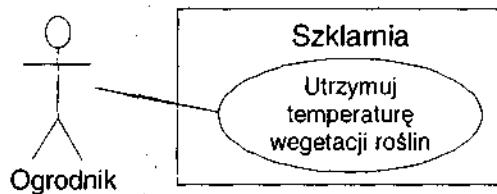
¹⁰ Dziękuję za pomoc Jamesowi H. Jarrettowi.



Każdy patykowaty ludzik reprezentuje „aktora”, którym jest na ogół człowiek albo jakiś inny niezależny agent (może być nim nawet inny system komputerowy — tak jak w przypadku ATM). Prostokąt określa zakres naszego systemu. Elipsy symbolizują przypadki użycia, będące opisami istotnych zadań, które można wykonać, używając systemu. Linie pomiędzy aktorami i przypadkami użycia reprezentują natomiast interakcje.

Nie ma znaczenia, w jaki sposób system jest naprawdę zaimplementowany, dopóki wydaje się taki z punktu widzenia użytkownika.

Przypadki użycia nie muszą być wyjątkowo skomplikowane, nawet gdy reprezentują one złożony system. Ich celem jest jedynie prezentacja systemu w taki sposób, jak wygląda on z punktu widzenia użytkownika. Na przykład:



Przypadki użycia tworzą specyfikacje wymagań, określając wszystkie możliwe interakcje użytkownika z systemem. Próbujemy zatem określić pełny zbiór przypadków użycia systemu, a gdy to już uczynimy, otrzymamy istotę tego, co system powinien robić. Korzyść wynikająca ze skupienia się na przypadkach użycia polega na tym, że zawsze sprowadzają nas one do istoty rzeczy, powstrzymując od zagłębiania się w kwestie nieistotne dla wykonania pracy. Oznacza to, że mając pełny zbiór przypadków użycia można opisać system i przejść do następnego etapu projektowania. Prawdopodobnie nie uda ci się jeszcze zrozumieć wszystkiego za pierwszym podejściem, ale nie ma to znaczenia. Wszystko wyjaśni się w swoim czasie — jeżeli na tym etapie chcesz uzyskać doskonałą specyfikację systemu, to z pewnością na tym utkniesz.

Jeżeli zdarzyło ci się utknąć, to możesz ruszyć z miejsca, posługując się metodą umożliwiającą zgrabne przybliżenie problemu: opisz system w kilku akapitach, a następnie przyjrzyj się rzecznikom i czasownikom. Rzeczniki mogą odpowiadać aktorom, kontekstom przypadków użycia (np. „hol wejściowy”) lub przedmiotom przetwarzanym w przypadkach użycia. Czasowniki mogą natomiast opisywać interakcje zachodzące pomiędzy aktorami i przypadkami użycia, a także określać kolejne kroki realizacji przypadków użycia. Zauważysz również, że obiekty i komunikaty są w fazie projektowania oznaczane za pomocą rzeczników i czasowników. Zwróć jednak uwagę na to, że przypadki użycia opisują interakcje pomiędzy podsystemami, a zatem technika „rzeczników i czasowników” może być używana jedynie w charakterze instrumentu myślowego — ze względu na to, że nie umożliwia ona tworzenia przypadków użycia”.

Granica pomiędzy przypadkiem użycia i aktorem może wskazywać na istnienie interfejsu użytkownika, ale nie definiuje ona tego interfejsu. Proces definiowania i tworzenia interfejsów użytkownika został opisany w książce Larry'ego Constantine'a i Lucy Lockwood: *Software for Use* (Addison Wesley Longman, 1999) oraz pod adresem: www.ForUse.com.

Chociaż jest to czarna magia, istotne jest wprowadzenie w tej fazie jakiegoś podstawowego harmonogramu. Wiesz już w przybliżeniu, co budujesz, a zatem jesteś prawdopodobnie w stanie określić, ile zajmie to czasu. Odgrywa w tym rolę wiele czynników. Jeżeli przewidujesz długi termin realizacji, to firma może zrezygnować z budowy systemu (i tym samym spożytkować swoje zasoby na jakiś rozsądniejszy cel). Może się również zdarzyć, że kierownik zdecydował już, ile powinien trwać projekt i będzie próbował wpływać na twoje oszacowanie. Jednak najlepiej jest mieć od początku rzetelnie przygotowany harmonogram i wcześniej uporać się z trudnymi decyzjami. Podejmowano wiele prób opracowania technik tworzenia dokładnych harmonogramów (podobnie jak technik przewidywania notowań giełdowych), ale prawdopodobnie najlepszym wyjściem jest poleganie na własnym doświadczeniu i intuicji. Zastanów się poważnie, ile naprawdę zajmie to czasu, a następnie podwój otrzymaną wartość i powiększ ją o 10 procent. Twoje początkowe przekonanie jest prawdopodobnie trafne — możesz otrzymać w tym czasie coś, co działa. Podwojenie czasu spowoduje jednak, że uzyskasz produkt przyzwoitej jakości, a dodatkowe 10 procent pozwoli na nadanie ostatniego szlifu i uporanie się ze szczegółami¹². Jakkolwiek zamierzasz to wyjaśnić i niezależnie od lamentów i prób manipulacji, które nastąpią po przedstawieniu takiego harmonogramu, po prostu wydaje się, że jest on zgodny z prawdą.

¹ Więcej informacji na temat przypadków użycia można znaleźć w książkach: Schneider, Winters: *Applying Use Cases*, Addison-Wesley, 1998 oraz Rosenberg: *Use Case Driven Object Modeling with UML* Addison-Wesley, 1999.

² Mój własny pogląd na ten temat uległ ostatnio zmianie. Podwojenie czasu i powiększenie go o 10 procent zapewni ci przybliżenie z rozsądnią dokładnością (zakładając, że w problemie nie występuje zbyt wiele czynników nieprzewidywalnych), ale nadal będziesz musiał dość intensywnie pracować, aby skończyć pracę w tym czasie. Jeżeli potrzebujesz czasu, by efekt tej pracy był elegancki, i chcesz się dobrze bawić podczaszej realizacji, to sądzę, że właściwym mnożnikiem czasu będzie raczej trzy albo cztery.

Etap 2. Jak to zrobimy?

Na tym etapie powinniśmy utworzyć projekt opisujący, jak wyglądają klasy i w jaki sposób będą one ze sobą współpracować. Doskonałym narzędziem, umożliwiającym określenie koniecznych klas oraz interakcji pomiędzy nimi, są karty CRC (ang. *Class-Responsibility-Collaboration – klasa-obowiązek-współpraca*). Jedną z zalet tej metody jest fakt, że jest ona nieskomplikowana technologicznie — zaczynamy od zbioru czystych kart o rozmiarach 3 na 5 cali (ok. 7,6x12,7 cm), a następnie zapisujemy na nich informacje. Każda karta reprezentuje pojedynczą klasę i zapisujemy na niej:

1. Nazwę klasy. Powinna być ona związana z istotą tego, co dana klasa robi — tak aby na pierwszy rzut oka wyglądało to logicznie.
2. „Obowiązki” klasy, czyli to, co powinna ona robić. Na ogół można je podsumować, wymieniając po prostu nazwy funkcji składowych klasy (ponieważ w dobrym projekcie nazwy te powinny być opisowe), ale nie wyklucza to sporządzenia dodatkowych notatek. Jeżeli musisz rozpocząć ten proces, spójrz na zagadnienie z punktu widzenia leniwego programisty: jakie obiekty powinny pojawić się w czarodziejski sposób, aby rozwiązać twój problem?
3. „Współpraca” klasy, czyli jej interakcja z innymi klasami. Użycie w tym miejscu szerokiego pojęcia „interakcji” jest celowe — może ona oznaczać zarówno łączenie klas albo po prostu istnienie jakiegoś innego obiektu, świadczącego usługi na rzecz obiektu danej klasy. Współpraca powinna również uwzględniać „obserwatorów” tej klasy. Na przykład jeżeli utworzymy klasę **Fajerwerk**, to kto ją będzie obserwował: **Chemik** czy **Widz**? Pierwszy z nich będzie chciał się dowiedzieć, jakich chemikaliów używa się do jego konstrukcji, drugi natomiast będzie zainteresowany kolorami i kształtami powstałymi podczas eksplozji.

Być może wydaje ci się, że karty powinny być większe, biorąc pod uwagę wszystkie informacje, które chciałbyś na nich umieścić. Są one jednak celowo małe, nie tylko po to, by zachować niewielkie rozmiary klas, ale również po to, by uniemożliwić ci zbyt wcześnie zagłębianie się w szczegóły. Jeżeli nie potrafisz zmieścić wszystkich niezbędnych informacji na temat klasy na małej kartce, oznacza to, że klasa ta jest zbyt złożona (opisałeś ją zbyt szczegółowo albo też powinieneś utworzyć więcej niż jedną klasę). Idealna klasa powinna być zrozumiała na pierwszy rzut oka. Ideą kart CRC jest pomoc w uzyskaniu pierwszego przybliżenia rozwiązania — umożliwiającego ogarnięcie całości projektu, a następnie jego dopracowanie.

Jedną z istotnych korzyści kart CRC jest komunikacja. Najlepiej uzyskuje się ją na bieżąco — w grupie, bez użycia komputerów. Każda osoba bierze na siebie odpowiedzialność za kilka klas (które początkowo nie posiadają nazw; nie ma również na ich temat żadnych innych informacji). Uruchamiamy symulację „na żywo”, znajdując za każdym razem rozwiązanie jednego scenariusza — podejmujemy decyzje, jakie komunikaty są wysyłane do poszczególnych obiektów w celu jego realizacji. W trakcie tego procesu poznajemy niezbędne klasy, wraz z ich zadaniami i współpracą pomiędzy nimi, oraz wypełniamy poszczególne karty. Po przejściu przez wszystkie przypadki użycia powinniśmy uzyskać pierwsze pełne przybliżenie projektu.

Pamiętam swoje najbardziej udane doświadczenia konsultingowe związane z tworzeniem wstępnego projektu, zanim rozpoczęłem używanie kart CRC. Stałem przed zespołem, który nigdy wcześniej nie realizował projektu obiektowego, rysując obiekty na tablicy. Rozmawialiśmy na temat tego, w jaki sposób powinny komunikować się ze sobą obiekty, wymazując niektóre z nich i zastępując je innymi. W rezultacie posługiwałem się „kartami CRC” na tablicy. Zespół (który wiedział, jakie zadania ma realizować system) w rzeczywistości tworzył projekt — był on w większym stopniu „własnością” zespołu niż zostało mu podarowane. Moja rola ograniczała się do kierowania tym procesem poprzez zadawanie właściwych pytań, testowanie założeń oraz wykorzystywanie uwag zespołu do ich modyfikacji. Prawdziwa zaleta tego podejścia polegała na tym, że zespół uczył się realizacji projektu obiektowego nie za pomocą przeglądania abstrakcyjnych przykładów, ale pracując nad projektem, który był dla niego wówczas najbardziej interesujący — czyli swoim własnym.

Kiedy rozpoczynasz pracę z zestawem kart CRC, możesz pokusić się o bardziej formalny opis swojego projektu, wykorzystując notację UML¹³. Nie musisz używać języka UML, ale może on być pomocny — szczególnie gdy zamierzasz powiesić na ścianie diagram, który każdy mógłby przemyśleć, co jest dobrym pomysłem. Rozwiązaniem alternatywnym w stosunku do UML są tekstowe opisy obiektów oraz ich interfejsów, albo — w zależności od używanego przez ciebie języka programowania — sam kod programu¹⁴.

Język UML pozwala również na stosowanie dodatkowej notacji w postaci diagramów, umożliwiających opis dynamicznego modelu systemu. Jest ona przydatna w przypadkach, w których przejścia pomiędzy stanami systemu lub podsystemu są na tyle znaczące, że konieczne jest przygotowanie dla nich odrębnych diagramów (na przykład w systemie sterującym). Może ponadto wystąpić konieczność opisania struktur danych w systemach lub podsystemach, w przypadku których dane są czynnikiem dominującym (na przykład w bazach danych).

Zakończysz drugi etap, kiedy opiszesz obiekty oraz ich interfejsy. A przynajmniej większość z nich — jest bowiem zazwyczaj kilka takich, o których zapomniano i które ujawnią się dopiero w trzecim etapie. Ale nie ma się czym przejmować — interesuje cię tylko to, by znaleźć w końcu wszystkie obiekty. Dobrze jest zrobić to na wczesnych etapach projektowania, ale programowanie obiektowe udostępnia konstrukcje, dzięki którym nie jest również problemem późniejsze ich odkrycie. W istocie projektowanie obiektów odbywa się zazwyczaj w pięciu etapach, w czasie całego procesu tworzenia programu.

Pięć etapów projektowania obiektów

Czas projektowania obiektu nie jest ograniczony do okresu, w którym pisany jest program. Projektowanie to przebiega natomiast w kolejnych etapach. Dzięki takiej perspektywie przestaniesz oczekiwania natychmiastowego osiągnięcia ideału — **uświadomyj sobie**, że zrozumienie tego, co robią obiekty i jak powinny one wyglądać, przychodzi

¹³ Początkującym polecam **wymienioną** już wcześniej pozycję *UML Distilled*.

¹⁴ Python (www.Python.org) jest często wykorzystywany w charakterze „wykonywalnego pseudokodu”.

dopiero po pewnym czasie. Taka obserwacja dotyczy również projektowania różnego rodzaju programów — wzorcowa postać określonego rodzaju programu powstaje w wyniku podejmowanych wielokrotnie prób rozwiązania problemu (wzorce projektowe zostały omówione w drugim tomie książki). Również obiekty mają swoje wzorce, ujawniające się poprzez ich zrozumienie i wielokrotne używanie.

- 1. Znajdowanie obiektów.** Ten etap następuje w czasie początkowej analizy programu. Obiekty mogą zostać znalezione dzięki poszukiwaniu: zewnętrznych czynników i ograniczeń, powtarzających się elementów systemu oraz najmniejszych jednostek pojęciowych. Niektóre z obiektów są oczywiste, jeżeli dysponujemy już biorem bibliotek klas. Podobieństwa między klasami, wskazujące na obecność klas podstawowych oraz dziedziczenia, mogą pojawić się wcześniej lub później w trakcie procesu projektowania.
- 2. Składanie obiektów.** W czasie tworzenia obiektu odkryjesz potrzebę dodania do niego składowych, niewidocznych jeszcze w chwilę jego znalezienia. Wewnętrzne wymagania obiektu mogą wymuszać wspomaganie go przez inne klasy.
- 3. Konstrukcja systemu.** Na tym etapie ponownie mogą się pojawić dodatkowe wymagania dotyczące obiektu. W miarę jak powiększa się Twoja wiedza, zmieniasz stopniowo swoje obiekty. Potrzeba komunikacji i wzajemnych powiązań z innymi obiektami występującymi w systemie może powodować zmianę wymagań Twojej klasy lub pociągać za sobą konieczność tworzenia nowych klas. Na przykład możesz odkryć potrzebę utworzenia klas pomocniczych, takich jak lista wiązana, zawierających niewiele informacji o stanie (lub w ogóle ich niezawierających) i wspomagających po prostu działanie innych klas.
- 4. Rozbudowa systemu.** W trakcie dodawania do systemu nowych funkcji możesz zauważyc, że Twój wcześniejszy projekt nie umożliwiał łatwej rozbudowy systemu. Bogatszy o tę wiedzę, możesz przebudować część systemu, na przykład dodając do niego nowe klasy lub hierarchię klas.
- 5. Wielokrotne wykorzystywanie obiektów.** Jest to prawdziwa próba wytrzymałościowa klasy. Jeżeli ktoś próbuje wykorzystać klasę w zupełnie nowych warunkach, to prawdopodobnie odkryje jakieś wady. W miarę jak będziesz modyfikował tę klasę, przystosowując ją do kolejnych programów, jej ogólne podstawy staną się coraz bardziej przejrzyste, aż w końcu uzyskasz typ naprawdę odpowiedni do wielokrotnego wykorzystania. Nie należy jednak spodziewać się, że większość obiektów wchodzących w skład projektu systemu będzie nadawała się do ponownego użytku — jest oczywiste, że wiele obiektów zostanie opracowanych w sposób specyficzny dla konkretnego systemu. Typy nadające się do wielokrotnego wykorzystania występują zazwyczaj, a ponadto po to, by można ich wielokrotnie używać, muszą one rozwiązywać bardziej ogólne problemy.

Wskazówki dotyczące projektowania obiektów

Przedstawione poniżej etapy zawierają wskazówki, które przydadzą Ci się w chwili, gdy zaczniesz się zastanawiać nad projektowaniem klas:

1. Niech konkretny problem spowoduje utworzenie klasy, a następnie pozwól jej rosnąć i dojrzewać, rozwiązujeając inne problemy.
2. Pamiętaj o tym, że odkrywanie niezbędnych klas (oraz ich interfejsów) stanowi głównączęść projektowania systemu. Gdybyś miał już te klasy, byłby to łatwy projekt.
3. Nie zmuszaj się do zrozumienia wszystkiego już na samym początku — ucz się w trakcie pracy. W końcu i tak to nastąpi.
4. Zaczni programować — stwórz coś działającego, dzięki czemu będziesz mógł potwierdzić poprawność swojego projektu lub wykazać jego błędnosć. Nie obawiaj się, że skończy się to utworzeniem popłatanego kodu, napisanego w stylu proceduralnym — klasy **dzielą bowiem** problem na części, pozwalając zapanować zarówno nad anarchią jak i entropią. Złe klasy nie psują dobrych klas.
5. Zawsze zachowuj prostotę. Lepsze są małe i proste obiekty, o oczywistym przeznaczeniu, niż wielkie i skomplikowane interfejsy. Kiedy przychodzi czas podejmowania decyzji, używaj brzytwy **Ockham'a** — rozważ wszystkie możliwości i wybierz tę, która jest najprostsza, ponieważ proste klasy są niemal w każdym przypadku najlepsze. Rozpocznij od małej i prostej klasy — jej interfejs będzie mógł rozwinąć, gdy już zrozumiesz ją lepiej, jednakże, w miarę upływu czasu, coraz trudniej będzie usunąć z klasy jakiekolwiek elementy.

Etap 3. Budujemy jądro

Jest to wstępne przejście ze zgrubnego projektu do kompilacji i wykonania głównej części kodu, który będzie można testować, dzięki czemu w szczególności potwierdzona lub zanegowana zostanie poprawność przyjętej architektury. Nie jest to proces jednoprzebiegowy, a raczej początek serii kroków, które, jak się przekonamy w czwartym etapie, umożliwiają iteracyjną budowę systemu.

Twoim celem jest znalezienie jądra architektury systemu, które musi zostać zaimplementowane w celu utworzenia działającego systemu, niezależnie od tego, jak bardzo jest on niekompletny na obecnym, początkowym etapie. Tworzysz szkielet, który będzie następnie — w trakcie kolejnych iteracji — stopniowo rozbudowywany. Przeprowadzisz pierwsze z wielu integracji i testów systemu, a także poinformujesz udziałowców, jak będzie wyglądał ich system i jak zaawansowane są prace nad nim. W idealnym przypadku ujawnisz również niektóre z istotnych zagrożeń projektu. Prawdopodobnie odkryjesz także modyfikacje oraz udoskonalenia, które mogą być wprowadzone do pierwotnej architektury systemu — kwestie, o których nie dowieǳiałbyś się, gdybyś go nie zaimplementował.

Elementem budowy systemu jest rzeczywiste potwierdzenie jego zgodności z analizą wymagań oraz specyfikacją (niezależnie od tego, w jakiej istnieją one postaci), uzyskane z wyniku testów. Upewnij się, że testy zweryfikują również wymagania i przypadki użycia. Gdy jądro twojego systemu będzie już stabilne, można przejść do następnego etapu i zwiększyć **nieco jego** funkcjonalność.

Etap 4. Iteracje przez przypadki użycia

Kiedy działa już zasadniczy szkielet systemu, dodanie do niego każdego zbioru cech jest samo w sobie niewielkim projektem. Zbiór cech dodawany jest do systemu w czasie *iteracji* – względnie krótkiego procesu twórczego.

Jak długo trwa iteracja? W idealnym przypadku każda iteracja zajmuje od jednego do trzech tygodni (czas ten może się różnić w zależności od języka implementacji). Po zakończeniu tego okresu uzyskujemy zintegrowany, przetestowany system, o funkcjonalności większej niż uprzednio. Jednakże najbardziej interesującą kwestią jest podstawa każdej iteracji – pojedynczy przypadek użycia. Każdy przypadek użycia jest pakietem powiązanych ze sobą funkcji, które wbudowuje się w system równocześnie, w trakcie pojedynczej iteracji. Pozwala to nie tylko wypracować lepszy pogląd na temat tego, jaki powinien być zakres przypadków użycia, ale stanowi również potwierdzenie słuszności ich idei – z uwagi na to, że nie tylko nie zostają one odrzucone, po zakończeniu etapów analizy i projektowania, ale stają się wręcz fundamentalną jednostką rozwoju w procesie tworzenia oprogramowania.

Iteracje kończą się po osiągnięciu docelowej funkcjonalności albo gdy nadajdzie narucony z góry ostateczny termin wykonania pracy, a klient może zadowolić się aktualną wersją systemu (pamiętaj, że oprogramowanie jest biznesem opartym na subskrypcji). Ponieważ proces ten jest iteracyjny, istnieje wiele sposobności do wydania produktu klientowi – nie jest to jakiś jedyny punkt końcowy. Projekty typu open-source funkcjonują wyłącznie w środowisku iteracyjnym, o silnym sprzężeniu zwrotnym, co jest z pewnością źródłem ich sukcesu.

Iteracyjny proces rozwoju oprogramowania jest korzystny z wielu powodów. Umożliwia wczesne ujawnienie i rozwiązanie istotnych zagrożeń projektu, klienci mają wiele okazji do zmiany swoich zamierzeń, większa jest satysfakcja programistów, a projektem można kierować z większą precyzją. Jednakże istotną dodatkową korzyścią jest sprzężenie zwrotne z udziałowcami, którzy widzą aktualny stan produktu, dokładnie wiedzą, jakim etapie znajduje się jego realizacja. Może to zmniejszyć lub zupełnie wyeliminować potrzebę organizowania jałowych spotkań, dotyczących stanu zaawansowania pracy, a także zwiększyć zaufanie i wsparcie ze strony udziałowców.

Etap 5. Ewolucja

Ten etap cyklu projektowego jest tradycyjnie określany mianem „pielęgnacji” (ang. *maintenance*) – słowem-wytrychem, który może oznaczać właściwie wszystko – począwszy od „doprowadzenia do działania zgodnie z pierwotnymi założeniami” przez „dodanie funkcji, o których klient zapomniał nadmienić” po bardziej tradycyjne „poprawianie ujawnionych błędów” oraz „dodawanie, w miarę potrzeby, nowych funkcji”. Tak wiele błędnych znaczeń przypisywano pojęciu „pielęgnacji”, że nie odaje ono trafnie istoty rzeczy – częściowo z tego powodu, że sugeruje, iż w rzeczywistości został napisany program i teraz trzeba jedynie „wymieniać w nim części, oliwić go i pilnować, aby nie zardzewiał”. Być może istnieje więc lepszy termin, określający to, z czym mamy w rzeczywistości do czynienia.

W tej książce posłużę się terminem *ewolucja*¹⁵. Oznacza on: „nie zrobisz od razu wszystkiego dobrze, więc pozostaw sobie możliwość, że gdy nabierzesz doświadczenia, powrócisz i wprowadzisz zmiany”. Być może będziesz musiał wprowadzić wiele zmian, w miarę uczenia się i głębszego zrozumienia problemu. Elegancja, którą osiągniesz aż do uzyskania dobrego rozwiązania, opłaci się zarówno w perspektywie krótkoterminowej, jak i w długoterminowej. W trakcie ewolucji twój program z dobrego zamienia się we wspaniałe, a zagadnienia, których nie rozumiałeś przy pierwszym podejściu, stają się jasne. Jest to również proces, w czasie którego twoje klasy mogą ewoluować z możliwych do wykorzystania w pojedynczym projekcie do zasobów nadających się do wielokrotnego użycia.

„Zrobienie tego dobrze” nie opisuje jedynie sytuacji, w której program działa zgodnie z wymaganiami oraz przypadkami użycia. Oznacza to również, że wewnętrzna struktura kodu jest logiczna, a jej poszczególne elementy wydają się dobrze do siebie pasować, nie zawiera ona niezgrabnych struktur, przeróżnych obiektów albo niezręcznych obnażonych fragmentów kodu. Musisz mieć również wrażenie, że struktura programu przetrwa zmiany, które niewątpliwie nastąpią w okresie jego istnienia, oraz że modyfikacje te będą mogły zostać wprowadzone bez trudności. Nie jest to łatwe zadanie. Musisz nie tylko zrozumieć, co tworzysz, ale również to, w jaki sposób program będzie się rozwijał (co nazywam *wektorem zmian*¹⁶). Na szczęście obiektowe języki programowania są szczególnie predysponowane do wspomagania tego rodzaju ciągłych modyfikacji — granice tworzone przez obiekty są tym, co powstrzymuje strukturę przed rozpadem. Umożliwiają one również wprowadzanie zmian — takich, które wydawałyby się drastyczne w programie proceduralnym — bez wywoływanego trzęsienia ziemi w twoim kodzie. W istocie wsparcie dla ewolucji jest być może najważniejszą zaletą programowania obiektowego.

W trakcie ewolucji tworzysz coś, co przynajmniej stanowi przybliżenie początkowego zamysłu. Następnie sprawdzasz jeszcze wszystko dokładnie, porównujesz z wymaganiami i wykrywasz wszelkie niedoskonałości. Następnie cofasz się i dokonujesz poprawek — ponownie projektując i implementując fragmenty programu, które nie działają dobrze¹⁷. Obecnie do rozwiązania problemu (lub jego części) możesz potrzebować wielu podejść, zanim wpadniesz na właściwe rozwiązanie (na ogół pomocne w tym przypadku jest zapoznanie się z *wzorcami projektowymi*, opisanymi w drugim tomie książki).

;

¹⁵ Przynajmniej jeden z aspektów ewolucji został opisany w książce Martina Flawera *Refactoring: improving the design of existing code* (Addison-Wesley, 1999). Upozdrawiam, że książka wykorzystuje przykłady napisane wyłącznie w języku Java.

¹⁶ Pojęcie to jest omawiane szczegółowo w rozdziale *Design Patterns*, znajdującym się w drugim tomie książki.

¹⁷ Jest to coś w rodzaju „szybkiego przygotowywania prototypów”, w przypadku którego miałybyś zamiar zbudować naprzecię wersję umożliwiającą poznanie systemu, a następnie wyrzucić przygotowany w ten sposób prototyp i zbudować go jak należy. Problem z szybkim przygotowywaniem prototypów polegał na tym, że zamiast wyrzucić prototyp, budowano na nim system. W połączeniu z brakiem struktury w programowaniu strukturalnym, prowadziło to do stworzenia niechlujnych systemów; których pielęgnacja była kosztowna.

Ewolucja zachodzi również **wtedy**, gdy budujesz system, sprawdzasz, czy spełnia on wymagania, a następnie odkrywasz, że nie spełnia on założeń. Gdy przeanalizujesz system w działaniu, zauważasz, że w istocie chciałeś rozwiązać zupełnie inny problem. Jeżeli myślisz, że ewolucja przebiegać będzie w taki właśnie sposób, to powinieneś zbudować swoją pierwszą wersję tak szybko, jak to możliwe, dzięki czemu przekonasz się, czy odpowiada oczekiwaniom.

Warto przede wszystkim zapamiętać, że domyślnie — z definicji, w rzeczywistości — jeżeli modyfikujesz klasę, to jej klasy podstawowe i klasy pochodne będą nadal działać. Nie musisz obawiać się modyfikacji (zwłaszcza gdy dysponujesz wbudowanym zbiorem testów pozwalających na sprawdzenie poprawności modyfikacji). Modyfikacja niekoniecznie musi oznaczać zepsucie programu, a ponadto wszelkie zmiany będą w rezultacie ograniczone do klas podanych i (lub) specyficznej współpracy pomiędzy zmienianymi klasami.

Planowanie się opłaca

Oczywiście, nie zbudowałbyś domu bez wielu starannie nakreślonych planów. Jeżeli budujesz taras lub budę dla psa, twoje plany nie będą tak szczegółowe, ale prawdopodobnie rozpoczniesz pracę od jakiegoś rodzaju szkicu, którym będziesz się kierował. Projektowanie oprogramowania popadało ze skrajnością w skrajność. Przez długi czas programowanie nie miało specjalnych ram organizacyjnych, ale wówczas zaczęły upadać duże projekty. W rezultacie skończyliśmy na metodykach, które miały groźną liczbę struktur i szczegółów, pomyślanych przede wszystkim z myślą o tych obszernych projektach. Te metodyki są zbyt odstręczające — można by odnieść wrażenie, że masz spędzić cały czas pisząc dokumentację i nie mając chwili na programowanie (często do tego dochodziło). Sugeruję pośrednią drogę; użyj podejścia, które odpowiada twoim potrzebom (i osobiście tobie). Niezależnie od tego, jak małym postanowisz go uczynić, *jakikolwiek* plan będzie stanowił znaczne udoskonalenie twojego projektu w stosunku do braku planu w ogóle. Pamiętaj, że według większości szacunków przeszło 50 procent projektów upada (niektóre oszacowania wskazują 70 procent)!

Postępując zgodnie z planem — najlepiej prostym i zwięzłym — i tworząc strukturę projektu przed rozpoczęciem kodowania, odkryjesz, że rzeczy łączą się ze sobą znacznie łatwiej niż gdy rzucisz się na głęboką wodę i zaczniesz pisanie kodu. Sprawi ci to również wielką satysfakcję. Z mojego doświadczenia wynika, że uzyskanie eleganckiego rozwiązania jest głęboko satysfakcjonujące na zupełnie nowej płaszczyźnie. Wydaje się bliższe sztuce niż technologii. I wreszcie elegancja zawsze się opłaca — nie jest tylko sztuką dla sztuki. Taki program jest nie tylko łatwiej zbudować i usunąć z niego błędy, ale również zrozumieć i pielęgnować, a to przynosi zawsze zyski.

Programowanie ekstremalne

Uczyłem się technik analizy i projektowania od czasu studiów, choć z przerwami. Koncepcja programowania ekstremalnego (ang. *extreme programming* — *XP*) jest

najbardziej radykalną i inspirującą ze znanych mi technik. Jej opis można znaleźć w książce Kenta Becka *Extreme Programming Explained* (Addison-Wesley, 2000) lub w Internecie — pod adresem www.xprogramming.com.

Programowanie ekstremalne jest zarówno filozofią dotyczącą sposobu pracy programisty, jak i zbiorem wskazówek, w jaki sposób należy wykonywać. Niektóre z tych wskazówek znajdują odzwierciedlenie w innych, wcześniejszych metodach, ale dwoma, moim zdaniem najistotniejszymi i najbardziej wyróżniającymi się, są: „najpierw napisz testy” oraz „programowanie w parach”. Beck, mimo że kładzie duży nacisk na cały proces, wskazuje, że przyjęcie tylko tych dwóch elementów spowoduje znaczne zwiększenie produktywności i niezawodności.

Najpierw napisz testy

Testowanie jest tradycyjnie zepchnięte na ostatni etap projektu, gdy „wszystko już działa, ale trzeba jeszcze to zrobić — tak dla pewności”. Wynika z tego jego niski priorytet, a osoby specjalizujące się w testowaniu są traktowane po macoszemu — często zsyia się ich gdzieś do piwnicy, z dala od „prawdziwych programistów”. Zespoły testerów odpłacają pięknym za nadobne, ubierając się na czarno i rechocząc z uciechy za każdym razem, gdy znajdą jakiś błąd (szczerze mówiąc, sam miałem to samo uczucie znajdując błędy w kompilatorach C++).

Programowanie ekstremalne rewolucjonizuje koncepcję testowania poprzez przyznanie mu równego (a nawet wyższego) priorytetu niż kodowaniu. W rzeczywistości, testy są pisane *zanim* jeszcze powstanie kod, który będzie testowany, i zostająą trwale do niego przypisane. Testy muszą być wykonane poprawnie ilekroć jest przeprowadzana integracja projektu (czasami częściej niż raz dziennie).

Rozpoczęcie pracy od pisania testów pociąga za sobą bardzo istotne skutki.

Po pierwsze — wymusza przejrzystą definicję interfejsu klasy. Często proponuję, by jako narzędzie pomocne przy próbie projektowania systemu „wyobrazić sobie doskonąłą klasę, rozwiązującą konkretny problem”. Strategia testowania w programowaniu ekstremalnym posuwa się jeszcze dalej — określa precyzyjnie, jak musi wyglądać klasa z punktu widzenia jej użytkownika i jak dokładnie ma ona działać. I robi to w przejrzysty sposób. Można napisać całe opowiadanie lub narysować dowolne diagramy, opisujące, w jaki sposób zachowuje się klasa i jak powinna ona wyglądać, ale nic nie jest tak rzeczywiste, jak zestaw testów. Te pierwsze są tylko listami życzeń, natomiast testy stanowią kontrakt uwierzytelniony przez kompilator i działający program. Trudno wyobrazić sobie bardziej konkretny opis klasy niż testy.

W czasie pisania testów jesteś zmuszony do gruntownego przemyślenia klasy, dzięki czemu często odkryjesz takie konieczne do jej działania funkcje, które można przeczytać podczas eksperymentów myślowych z diagramami UML, kartami CRC, przypadkami użycia itp.

Druga istotna korzyść, związana z tworzeniem testów na samym początku, wynika z ich uruchamiania po każdej komplikacji programu. Stanowią one uzupełnienie testów przeprowadzanych przez kompilator. Jeżeli spojrzesz z tej perspektywy na ewolucję

języków programowania, zauważysz, że istotne udoskonaleńa dotyczące technologii języków są w istocie związane z testowaniem. Asembler sprawdzał jedynie składnię, ale język C narucił pewne ograniczenia semantyczne, zapobiegające popełnianiu określonych typów błędów. Języki obiektowe nałożyły jeszcze większą liczbę ograniczeń semantycznych, które — jeśli się nad nimi zastanowić — są w istocie pewnymi rodzajami testów. Pytania w rodzaju — „Czy ten typ danych jest używany właściwie?”, „Czy ta funkcja jest wywoływana poprawnie?” — są testami, przeprowadzanymi przez kompilator lub system uruchomieniowy. Widzimy rezultaty wbudowania tych testów w język — ludzie są w stanie pisać bardziej złożone systemy i uruchamiać je znacznie szybciej, wkładając w to o wiele mniej wysiłku. Łamałem sobie głowę, dlaczego tak jest, ale teraz rozumiem, że dzieje się tak dzięki testom — jeśli popełniasz błąd, a siatka zabezpieczająca, utworzona przez wbudowane testy, informuje cię, że pojawił się problem i wskazuje miejsce, w którym wystąpił.

Jednakże wbudowane testy, możliwe do przeprowadzenia na podstawie projektu języka, dochodzą tylko do tego miejsca. W pewnym momencie musisz wkroczyć i do dać pozostałe testy, by — w połączeniu z kompilatorem i systemem uruchomieniowym — tworzyły one pełny zestaw, sprawdzający poprawność całego programu. I czy mając taki kompilator, zaglądający ci przez ramię, nie chciałbyś, aby testy te również pomagały ci od samego początku? To właśnie dlatego zaczynasz od pisania testów, a następnie uruchamiasz je automatycznie podczas każdej komplikacji systemu. Twoje testy stają się rozszerzeniem siatki zabezpieczającej, udostępnianej przez język.

W kwestii używania coraz bardziej efektywnych języków programowania odkryłem również i to, że jestem skłonny do przeprowadzania coraz śmielszych eksperymentów, ponieważ wiem, że język ostrzeże mnie przed stratą czasu na tropienie błędów. Schemat testowania programowania ekstremalnego wykonuje to samo w stosunku do całego projektu. Ponieważ wiesz, że napisane przez ciebie testy zawsze wychwycą wszystkie problemy, które sam wprowadzisz (zastanawiając się nad nimi, regularnie uzupełniasz je o nowe testy), możesz dokonywać wielkich zmian, kiedy tylko zajdzie taka potrzeba, nie martwiąc się przy tym o to, że pograżysz cały swój projekt w zupełnym chaosie. Jest to niewiarygodnie potężne narzędzie.

Programowanie w parach

Programowanie w parach koliduje ze skrajnym indywidualizmem, który wpajano nam od kołyski, poprzez szkołę (kiedy samodzielnie osiągaliśmy sukcesy lub odnosiliśmy porażki, a wspólna praca z sąsiadem była traktowana jako „oszustwo”) oraz media — szczególnie hollywoodzkie filmy, których bohater walczy zazwyczaj przeciwko bezmyślnemu konformizmowi¹⁸. Programiści również są uważani za wzorcowych indywidualistów — „kowboi kodowania”, jak mawia o nich Larry Constantine. Nawet zwolennicy programowania ekstremalnego, którzy sami toczą batalię przeciwko konwencjonalnemu myśleniu, prezentują pogląd, że kod powinien być pisany przez dwie osoby, pracujące przy tym samym komputerze. Sugerują także, że praca powinna odbywać się w pomieszczeniu, w którym znajduje się grupa stacji roboczych — bez

¹⁸ Mimo że jest to być może raczej amerykański punkt widzenia, hollywoodzkie opowieści docierają wszędzie.

ścianek działowych, tak bardzo lubianych przez projektantów wewnętrz. Beck twierdzi nawet, że pierwszą czynnością związaną z wdrożeniem programowania ekstremalnego jest wzięcie do ręki śrubokręta oraz klucza francuskiego i zdemontowanie wszystkiego, co zawadza¹⁹. Wymaga to jednak obecności szefa, który potrafi odeprzeć ataki działa administracyjnego.

Korzyścią wynikającą z programowania w parach jest to, że jedna z osób zajmuje się pisaniem kodu programu, podczas, gdy druga go analizuje. „Myśliciel” ma na uwadze obraz całości — nie tylko aktualnie rozwiązywany problem, ale również zalecenia programowania ekstremalnego. Kiedy dwie osoby pracują razem, jest mniej prawdopodobne, że jedna z nich złamie zasady, postanawiając na przykład: „nie chcę zaczynać od pisania testów”. Poza tym, gdy osoba pisząca kod utknie na czymś, może zmienić się miejscami z „myślicielem”. Jeżeli obaj nie będą sobie mogli z czymś poradzić, to ich uwagi zostaną być może przypadkowo usłyszane przez kogoś pracującego w tym samym pomieszczeniu, kto będzie potrafił im pomóc. Praca w parach nadaje czynnościom projektowym właściwy kierunek. I, co prawdopodobnie ważniejsze, czyni programowanie zajęciem znacznie bardziej towarzyskim i zabawnym.

Zacząłem wykorzystywać programowanie w parach w trakcie ćwiczeń — na jednym z moich seminariów — i wydaje mi się, że znacznie wzbogaciło to doświadczenia wszystkich jegouczestników.

Dlaczego C++ odnosi sukcesy?

Jednym z powodów sprawiających, że język C++ cieszy się takim powodzeniem jest fakt, że nie powstał on jedynie w celu przekształcenie języka C w język obiektowy (choć od tego się właśnie zaczęło), ale również rozwiązania wielu innych problemów, z którymi zmagajasię dziś projektanci — szczególnie ci, którzy zainwestowali już sporo w C. Języki obiektowe prezentowały tradycyjnie podejście, zgodnie z którym należy odrzucić całą swoją dotychczasową wiedzę i zacząć od zera — posługując się nowym zestawem pojęć i nową składnią, argumentując, że lepiej jest — w dłuższej perspektywie — pozbyć się balastu doświadczeń wyniesionych z języków proceduralnych. Niewykluczone, że na dłuższą metę jest to prawda. Jednakże w krótszej perspektywie wiele z tych doświadczeń okazuje się przydatne. Być może najbardziej wartościowymi elementami nie jest wcale fundament istniejącego kodu (który, używając odpowiednich narzędzi, można przetłumaczyć), lecz istniejący *fundament intelektualny*. Jeżeli czynnie programując w C, musisz odrzucić wszystko to, co wiesz o tym języku, po to, by przyswoić sobie nowy język. Stajesz się natychmiast znacznie mniej produktywny na okres wielu miesięcy — dopóki twój umysł nie dostosuje się do nowego schematu. Natomiast w przypadku, gdy możesz jako punkt wyjścia wykorzystać swoją obecną wiedzę na temat języka C, jesteś w stanie zarówno — posługując się

¹⁹Wtaczając to (w szczególności) system nagłaśniający. Pracowałem kiedyś w firmie, która nalegała na nagłaśnianie rozmów telefonicznych, przychodzących do każdego kierownika, co bez przerwy przeszkadzało nam w pracy (ale szefom nie przyszło do głowy wyłączenie czegoś tak ważnego, jak system nagłaśniający). W końcu, kiedy nikt nie patrzył, poprzecinałem przewody głośnikowe.

tą wiedzą — nadal wydajnie pracować, jak i równocześnie przechodzić do świata programowania obiektowego. Jako że każdy ma własny model programowania, przejście to jest już i tak wystarczająco chaotyczne — nawet bez dodatkowego obciążenia w postaci rozpoczęnia od nauki nowego modelu języka od podstaw. Tak więc, w największym skrócie, powodem, który sprawia, że C++ cieszy się powodzeniem, jest ekonomia — przejście do programowania obiektowego nadal kosztuje, ale C++ może okazać się mniej kosztowny²⁰.

Celem C++ jest zwiększenie produktywności. Można ją osiągnąć na wiele sposobów, ale język zaprojektowano w taki sposób, by pomagał ci i zarazem jak najmniej przeszkadzał nieuzasadnionymi regułami czy też koniecznością wykorzystywania określonego zbioru cech. Język C++ opracowano po to, by był użyteczny. Decyzje projektowe, które legły u podstaw C++, miały na celu udostępnienie programistom maksymalnych korzyści (przynajmniej z punktu widzenia języka C).

Lepsze C

Natychmiast zyskujesz — nawet jeżeli nadal piszesz kod w języku C — ponieważ C++ domyka wiele luk zawartych w języku C, a także zapewnia lepszą kontrolę typów oraz analizę dokonywaną w czasie komplikacji. Jesteś zmuszony do deklarowania funkcji, dzięki czemu kompilator może nadzorować ich używanie. Konieczność wykorzystywania preprocesora została praktycznie wyeliminowana w stosunku do podstawiania wartości i makr, co usunęło grupę trudnych do wykrycia błędów. C++ posiada mechanizm nazywany *referencjami*, umożliwiający wygodniejszą obsługę adresów argumentów funkcji oraz zwracanych wartości. Obsługa funkcji została udoskonalona dzięki *przeciążaniu funkcji*, pozwalającemu na używanie tej samej nazwy w stosunku do różnych funkcji. Cechą nazywaną *przestrzeniami nazw* poprawia również kontrolę nazw. Istnieje również szereg drobnych udoskonaleń, zwiększających bezpieczeństwo C.

Zacząłeś się już uczyć

Problemem z nauką nowego języka jest produktywność. *Żadna* firma nie może sobie pozwolić na gwałtowny spadek produktywności inżyniera zajmującego się programowaniem, tylko dlatego, że uczy się on nowegojęzyka. C++ jest rozszerzeniem języka C, a nie zupełnie nową składnią i modelem programowania. Pozwala to na dalsze tworzenie użytecznego kodu i jednocześnie stopniowe wprowadzanie nowych cech — w miarę ich poznawania i rozumienia. Być może jest to jeden z najważniejszych powodów sukcesu C++.

W dodatku — większość z całego kodu napisanego przez ciebie w języku C nadal działa w C++, lecz z uwagi na to, że kompilator C++ jest bardziej drobiazgowy, często ponowne skompilowanie programu w C++, ujawnia znajdujące się w nim ukryte błędy języka C.

²⁰ Powiedziałem „może”, ponieważ, z uwagi na złożoność C++, w rzeczywistości tańsze może okazać się przejście na język Java. Jednakże na decyzję, który język wybrać, wpływa wiele czynników i w książce zakładam, że wybrałeś C++.

Efektywność

Czasami dobrze jest zrezygnować z szybkości wykonania programu na rzecz produktywności programisty. Na przykład — model finansowy może być użyteczny jedynie przez krótki okres, dlatego też ważniejsze jest jego szybkie powstanie niż szybkie **działanie**. Jednakże większości aplikacji wymaga w jakimś stopniu efektywności, dlatego też C++ zawsze opowiada się po jej stronie. Ponieważ programiści piszący w języku C są na ogół bardzo na nią wyczuleni, stanowi to również dobry sposób na wytrącenie im z ręki argumentów, że język jest zbyt przeładowany i powolny. Wiele spośród właściwości C++ zaprojektowano w ten sposób, by umożliwić poprawę wydajności w przypadku, gdy generowany kod nie jest dostatecznie efektywny.

Dysponujesz nie tylko tą samą niskopoziomową kontrolą, jak w przypadku języka C (włączając w to możliwość bezpośredniego używania języka asemblera wewnątrz programu napisanego w C++), ale praktyka dowodzi, że szybkość obiektowego programu w C++ waha się w granicach $\pm 10\%$ w stosunku do programu napisanego w C, a częstojest mu jeszcze bliższa²¹. Projekt przygotowany pod kątem programu obiektowego może być w rzeczywistości bardziej efektywny niż jego odpowiednik w C.

Systemy są łatwiejsze do opisania i do zrozumienia

Klasy projektowane pod kątem konkretnego problemu na ogół lepiej go określają. Oznacza to, że pisząc kod, opisujesz swoje rozwiązanie, posługując się raczej pojęciami pochodząymi z przestrzeni problemu („wrzuć uszczelkę do kosza”) niż właściwymi komputerowi, stanowiącemu przestrzeń rozwiązania („ustaw bit procesora, oznaczający zamknięcie obwodu przekaźnika”).

Masz do czynienia z pojęciami wyższego poziomu, dzięki czemu możesz dokonać znacznie więcej w pojedynczym wierszu programu.

Inną korzyścią wynikającą z tej łatwości opisu jest pielęgnacja kodu, która (o ile wiezyć raportom) pochłania lwiączęść kosztów w okresie użytkowania programu. Jeżeli program jest przysłpniejszy, to jest on zarazem łatwiejszy w pielęgnacji. Dzięki temu można również obniżyć nakłady na utworzenie i pielęgnację dokumentacji.

Maksymalne wykorzystanie bibliotek

Najszybszym sposobem napisania programu jest użyciejuż napisanego kodu — biblioteki. Głównym celem języka C++ jest ułatwienie korzystania z bibliotek. Uzyskuje się to poprzez rzutowanie bibliotek na nowe typy danych (klasy), dzięki czemu dołączenie biblioteki oznacza dodanie do języka nowych typów. Ponieważ kompilator C++ czuwa nad sposobem użycia biblioteki — gwarantując właściwą inicjalizację i „sprzątanie”, a także zapewniając, że funkcje są poprawnie wywoływane — można skupić się na tym, do czego biblioteka ma służyć, zamiast zastanawiać się, w jaki sposób należy je używać.

²¹ Jednakże warto przejrzeć rubrykę Dana Saksa w *C/C++ User's Journal*, w której można znaleźć istotne rozważania dotyczące wydajności bibliotek C++.

Ponieważ używane nazwy mogą być przydzielone do poszczególnych fragmentów programu za pomocą przestrzeni nazw C++, można posłużyć się dowolną liczbą bibliotek, unikając popadania w znane z języka C kolizje nazw.

Wielokrotne wykorzystywanie kodu dzięki szablonom

Istnieje znacząca kategoria typów, które wymagają modyfikacji kodu źródłowego po to, by można ich było w efektywny sposób użyć powtórnie. *Szablony* w języku C++ dokonują automatycznej modyfikacji kodu źródłowego, co czyni je szczególnie efektywnym narzędziem, umożliwiającym wielokrotne wykorzystywanie kodu bibliotek. Typy projektowane za pomocą szablonów będą bez problemów działać z wieloma innymi typami. Szablony są szczególnie eleganckim rozwiązaniem z uwagi na to, że ukrywają przed klientem-programistą (użytkownikiem biblioteki) **złożoność**, związaną z takim sposobem wielokrotnego używania kodu.

Obsługa błędów

Obsługa błędów w C jest dobrze znanym i często ignorowanym problemem. W czasie tworzenia dużego i złożonego programu nie może przytrafić się nic gorszego, niż pojawienie się ukrytego gdzieś błędu, co do którego brakuje jakichkolwiek wskazówek, czym może on być spowodowany. *Obsługa wyjątków* w C++ (wprowadzona w tym, a opisana w pełni w drugim tomie książki — który można pobrać z witryny <http://helion.pl/online/thinking/index.html>) gwarantuje, że błąd zostanie zauważony, a w wyniku jego wystąpienia zostanie podjęte jakieś działanie.

Programowanie na wielką skalę

Wiele spośród tradycyjnych języków ma wbudowane ograniczenia, dotyczące wielkości i złożoności programów. Na przykład BASIC może doskonale nadawać się do sklecania szybkich rozwiązań dla pewnych klas problemów. Jednak w przypadku gdy program rozrośnie się powyżej kilku stron kodu lub wykroczy poza właściwą temu językowi dziedzinę problemów, programowanie zacznie przypominać płynwanie w coraz bardziej gęstniejącej cieczy. Język C również posiada takie ograniczenia. Na przykład gdy wielkość programu przekracza około 50 000 wierszy, problemem zaczynają być kolizje nazw — praktycznie kończą się nazwy zmiennych oraz funkcji. Innym, szczególnie uciążliwym, problemem są niewielkie luki w języku C — błędy ukryte w wielkim programie mogą być wyjątkowo trudne do znalezienia.

Nie ma wyraźnej granicy, określającej kiedy używany język zaczyna zawodzić, a nawet, gdyby **była**, to i tak byłaby zignorował. Nie powiesz przecież: „mój program w BASIC-u właśnie zrobił się zbyt duży — będę musiał przepisać go w C!”. Zamiast tego spróbujesz upchnąć do programu jeszcze kilka wierszy, aby dodać do niego kolejną nową funkcję. W ten sposób coraz realniejsze staje się widmo dodatkowych kosztów.

Język C++ zaprojektowano w ten sposób, by wspomagał on **programowanie na wielką skalę**, czyli usunął niewidoczne granice złożoności pomiędzy małymi a wielkimi

programami. Pisząc program narzędziowy typu „witaj, świecie!”, z pewnością nie musisz używać programowania obiektowego, szablonów, przestrzeni nazw ani obsługi wyjątków — ale możesz wykorzystać je wtedy, gdy będziesz ich potrzebował. Natomiast kompilator jest równie agresywny w wykrywaniu błędów — zarówno w stosunku do małego, jak i w wielkiego programu.

Strategie przejścia

Kiedy już zdecydujesz się na programowanie obiektowe, z pewnością zadasz pytanie: „W jaki sposób nakłonić mojego szefa (kolegów, wydział lub współpracowników) do używania obiektów?” Zastanów się, jak ty sam — niezależny programista — powinieneś rozpocząć naukę nowego języka oraz modelu programowania. Robiłeś to już wcześniej. Zaczyna się od nauki i przykładów. Następnie przychodzi czas na próbny projekt, pozwalający wyczuć podstawowe konstrukcje języka bez konieczności podejmowania zbyt skomplikowanych działań. Wreszcie nadchodzi czas na „rzeczywisty” projekt, który ma pełnić jakąś użyteczną rolę. Podczas jego realizacji kontynuujesz naukę — czytając, zadając pytania ekspertom i wymieniając się poradami zprzyjaciółmi. Jest to również metoda zalecana przezchodzący z C do C++ przez wielu doświadczonych programistów. Dokonanie takiego przejścia w całej firmie wiąże się, oczywiście, z pewnymi aspektami psychologii oddziaływań w obrębie grupy, ale na każdym kroku przydatna jest wiedza na temat tego, jak robiłaby to jedna osoba.

Wskazówki

Oto garść wskazówek, **wartych rozważenia przy przechodzeniu do programowania obiektowego oraz C++:**

1. Szkolenie

Pierwszym krokiem jest jakaś forma edukacji. Miej na uwadze inwestycje dokonane przez firmę w kod w czystym C i postaraj się nie pograżyć wszystkiego w chaosie na okres sześciu do ośmiu miesięcy, kiedy to wszyscy będą głowili się nad tym, jak działa wielokrotne dziedziczenie. Do szkolenia wybierz małą grupę, najlepiej złożoną z ludzi dociekliwych, dobrze ze sobą współpracujących i mogących wspierać się podczas nauki C++.

Czasami sugerowane jest podejście alternatywne, polegające na nauczaniu prowadzonym równocześnie na wszystkich szczeblach firmy, obejmującym zarówno kursy przeglądowe, przeznaczone dla menedżerów zajmujących się strategią, jak i szkolenia z zakresu projektowania i programowania, których uczestnikami są twórcy projektów. Jest to szczególnie korzystne w przypadku małych firm, dokonujących zasadniczego zwrotu w swoim sposobie działania, oraz na poziomie poszczególnych oddziałów większych firm. Ponieważ jednak wiąże się to z większymi wydatkami, można rozpoczęć od szkolenia na poziomie projektowym, zrealizować projekt pilotażowy (o ile to możliwe — z udziałem zewnętrznego doradcy), a następnie pozwolić, by uczestnicy projektu zostali nauczycielami pozostałych pracowników firmy.

2. Projekt niewielkiego ryzyka

Rozpocznij od projektu o niewielkim ryzyku i dopuść możliwość popełniania błędów. Kiedy już zyskasz pewne doświadczenie, będziesz mógł albo przydzielić członkom pierwotnego zespołu inne projekty, albo wykorzystać ich w charakterze personelu technicznego, wspierającego programowanie obiektowe. Pierwszy projekt może nie działać od razu poprawnie, dlatego też nie powiniem on mieć dla firmy kluczowego znaczenia. Winien być prosty, niezależny i pouczający, czyli wiązać się z utworzeniem klas, które przydadzą się innym pracującym w firmie programistom, kiedy rozpoczną naukę C++.

3. Bierz przykład z sukcesów

Zanim zaczniesz od zera, poszukaj przykładów dobrych projektów obiektowych. Istnieje duże prawdopodobieństwo, że ktoś już rozwiązał twój problem. Nawet jeżeli rozwiązanie niejest dokładnie tym, o co chodzi, możesz przypuszczalnie wykorzystać to, czego nauczyłeś się na temat abstrakcji, do zmodyfikowania istniejącego projektu — w taki sposób, by odpowiadał on twoim potrzebom. Jest to ogólna idea wzorców projektowych, opisanych w drugim tomie książki.

4. Używaj istniejących bibliotek klas

Podstawową motywacją ekonomiczną przejścia na C++ jest łatwość wykorzystania już istniejącego kodu w postaci bibliotek klas (w szczególności — standardowych bibliotek C++, opisanych szczegółowo w drugim tomie książki). Największe skrócenie cyklu projektowego nastapi wówczas, gdy nie trzeba będzie pisać niczego, oprócz **main()**, a tworzone i używane obiekty będą pochodziły z powszechnie dostępnych bibliotek. Jednakże niektórzy poczynającący programiści nie rozumieją tego — nie są świadomi istnienia bibliotek klas, albo, ulegając fascynacji językiem, pragną napisać klasy, które być może już istnieją. Największy sukces związany z programowaniem obiektowym i C++ możesz odnieść wtedy, gdy na wczesnym etapie przejścia zadasz sobie trud znalezienia i wykorzystania kodu napisanego przez innych.

5. Nie tłumacz istniejącego kodu na C++

Chociaż skompilowanie kodu napisanego w C za pomocą kompilatora C++ **zawyczaj** przynosi (niekiedy ogromne) korzyści, wynikające ze znalezienia błędów zawartych w starym kodzie, to tłumaczenie istniejącego i działającego kodu na C++ nie jest na ogół najlepszym sposobem wykorzystania czasu (**jeżeli** musisz przekształcić kod napisany w języku C w program obiektowy, wystarczy, że „opakujesz” go w klasy C++). Korzyści przyrastają stopniowo — szczególnie gdy planuje się powtórne wykorzystanie kodu. Jest jednak całkiem możliwe, że nie będziesz w stanie zauważyc spodziewanego zwiększenia wydajności w swoich pierwszych kilku projektach, chyba że będzie to zupełnie nowy projekt. Język C++ oraz programowanie obiektowe przynoszą najbardziej spektakularne rezultaty wówczas, gdy obejmują całość projektu — od pomysłu dojego realizacji.

Problemy z zarządzaniem

Jeżeli jesteś menedżerem, twoja praca polega na zdobywaniu środków dla zespołu, pokonywaniu przeszkód, które stoją na drodze do jego sukcesu i, ogólnie, na próbach zapewnienia najbardziej wydajnego i przyjaznego środowiska, dzięki któremu zespół będzie mógł dokonać wszystkich wymaganych od niego wyczynów. Przejście na C++ odpowiada wszystkim trzem powyższym kategoriom i wszystko byłoby wspaniale, gdyby nie kwestia kosztów. Mimo że przejście na C++ może być tańsze dla zespołu programującego w C (i prawdopodobnie w innych językach proceduralnych) — zależnie od twoich uwarunkowań²² — niż w przypadku alternatywnych języków obiektowych, nie odbywa się ono za darmo i istnieją przeszkody, na które powinieneś zwrócić uwagę.

Koszty początkowe

Koszt przejścia na C++ to więcej niż tylko zakup kompilatorów C++ (jeden z najlepszych kompilatorów, GNU C++, jest darmowy). Średnio- i długoterminowe koszty zostaną zminalizowane, jeżeli zainwestujesz w szkolenie (i prawdopodobnie nadzór nad pierwszym projektem), a także jeżeli zidentyfikujesz i zakupisz biblioteki klas rozwiązujeające twój problem, zamiast próbować zbudować je samodzielnie. Są to koszty, które muszą zostać uwzględnione w realistycznym planie. Istnieją ponadto ukryte koszty, związane z utratą wydajności w trakcie uczenia się nowego języka, a także, prawdopodobnie, nowego środowiska programowania. Szkolenie i doradztwo mogące z pewnością zminalizować, lecz członkowie zespołu muszą przezwyciężyć swoje problemy związane ze zrozumieniem nowej technologii. W czasie tego procesu będą oni popełniali więcej błędów (to zaleta, ponieważauważone błędy są najszybszą metodą uczenia się) i będą mniej wydajni. Jednak nawet wówczas, borykając się z różnymi problemami związanymi z programowaniem, mając właściwe klasy i odpowiednie środowisko programowania, można być bardziej produktywnym ucząc się C++ (nawet uwzględniając większą liczbę błędów i pisanie mniejszej liczby wierszy kodu dziennie) niż pozostając przy C.

Kwestie wydajności

Często zadawane jest pytanie: „czy programowanie obiektowe nie spowoduje, że moje programy staną się automatycznie znacznie większe i wolniejsze?”. Odpowiedź nie jest jednoznaczna. Większość tradycyjnych języków obiektowych była projektowana raczej z myślą o eksperymentowaniu i szybkim przygotowywaniu prototypów niż oszczędnym wykorzystaniem zasobów w czasie działania. Dlatego też, z praktycznego punktu widzenia, powodujące znaczne powiększenie programów oraz zmniejszenie szybkości ich pracy. Jednakże język C++ zaprojektowano z myślą o produkcji oprogramowania. Kiedy interesuje cię szybkie przygotowanie prototypu, możesz błyskawicznie połączyć ze sobą jego poszczególne elementy, pomijając zupełnie kwestie efektywności. Jeżeli używasz bibliotek dostarczanych przez niezależne firmy, to są one już zazwyczaj zoptymalizowane przez ich producentów — w każdym razie nie jest to problemem, gdy twoim celem jest szybkie opracowanie prototypu. Kiedy użyjesz systemu, o który ci chodziło, i okazuje się, że jest on wystarczająco mały i szybki,

²² Z uwagi na udoskonalenia związane z wydajnością, należy w tym miejscu również wziąć pod uwagę język Java.

przedsięwzięcie dobiera się do kresu. W przeciwnym przypadku rozpoczynasz optymalizację za pomocą narzędzi profilującego, szukając w pierwszej kolejności łatwych możliwości przyspieszenia programu, związanych z wykorzystaniem wbudowanych właściwości języka C++. Jeżeli to nie pomaga, potrzebujesz modyfikacji, których można dokonać w wewnętrznej implementacji klas, dzięki czemu nie ma konieczności zmiany jakiegokolwiek używającego ich kodu. Dopiero gdy wszystko zawiedzie, będziesz musiał zmienić projekt. Informacja, że wydajność jest tak istotna w tej części projektu jest wskazówką, że musi ona stanowić element pierwotnych założeń projektowych. Korzyścią wynikającą z szybkiego projektowania jest odkrycie tego faktu na wczesnym etapie realizacji systemu.

Jak już wspomniano, najczęściej podawaną wartością określającą różnice wielkości i szybkości działania programów napisanych w C i C++ jest $\pm 10\%$, a często programy te są siebie jeszcze bardziej zbliżone. Używając C++ zamiast C można nawet osiągnąć znaczne polepszenie wielkości i szybkości działania programu z uwagi na to, że projekt przygotowywany pod kątem C++ może być zupełnie odmienny od projektu tworzonego dla języka C.

Rezultat porównania wielkości i szybkości działania programów w językach C i C++ wynika raczej z praktyki i zapewne takim pozostanie. Niezależnie od liczby osób sugerujących realizację tego samego projektu w C i C++, prawdopodobnie żadna firma nie decyduje się na takie trwonne pieniężny, chyba że jest ona bardzo duża i zainteresowana tego rodzaju projektami badawczymi. Ale nawet w takim przypadku wydaje się, że istnieje lepszy sposób wydawania pieniężny. Niemal wszyscy programiści, którzy przeszli od C (lub jakiegoś innego języka proceduralnego) do C++ (albo innego języka obiektowego), znacznie zwiększyli swoją programistyczną wydajność — i jest to z pewnością najbardziej przekonujący argument, jaki można przytoczyć.

Typowe błędy projektowe

Kiedy zespół rozpoczyna przygodę z programowaniem obiektowym oraz C++, programiści popełniają zazwyczaj szereg typowych błędów projektowych. Dzieje się tak często z uwagi na zbyt małe sprzężenie zwrotne z ekspertami na etapie projektowania i implementacji wczesnych projektów — w firmie nie pracują jeszcze eksperci, a istnieje być może opór związany z angażowaniem zewnętrznych konsultantów. Łatwo jest przedwcześnie odnieść wrażenie, że już rozumie się programowanie obiektowe i zabrnąć w ślepą uliczkę. Problem oczywisty dla osoby doświadczanej może się stać dla nowicjusza przedmiotem wielkiej rozterki. Większość z tych bolesnych problemów można uniknąć, korzystając z usług w zakresie szkolenia i doradztwa, świadczonych przez doświadczonego zewnętrznego eksperta.

Z drugiej jednak strony fakt, że tak łatwo jest popełniać tego typu błędy projektowe, świadczy o głównej wadzie języka C++ — jego wstępnej zgodności z C (oczywiście, jest to również jego główna zaleta). Aby wywiązać się z zadania zdolności komplikacji kodu C, język musiał przyjąć pewne rozwiązania kompromisowe, czego wynikiem jest zawarta w nim pewna liczba „słabych punktów”. Większość z nich można naprawić w trakcie nauki języka. W książce, a także jej drugim tomie (oraz innych książkach, opisanych w dodatku C), próbuję przedstawić większość pułapek, z którymi prawdopodobnie będziesz miał do czynienia używając języka C++. Powinieneś mieć zawsze świadomość, że „siatka zabezpieczająca” jest jednak nieco dziurawa.

Podsumowanie

W niniejszym rozdziale podjąłem próbę przedstawienia szerokiego zakresu zagadnień dotyczących programowania obiektowego oraz C++. Wskazałem, co wyróżnia programowanie obiektowe, a w szczególności — co wyróżnia język C++, omówięlem koncepcje metodyk programowania obiektowego oraz, w końcowej części, problemy, z jakimi możesz mieć do czynienia, dokonując w swojej firmie przejścia do programowania obiektowego oraz C++.

Programowanie obiektowe i C++ mogą nie sprostać niektórym oczekiwaniom. Należy określić swoje potrzeby i podjąć decyzję, czy C++ optymalnie zaspokoi, ~~czy~~ tez lepiej będzie zdecydować się na inny system programowania (włączając w to system używany obecnie). Jeśli wiesz, że twoje potrzeby będą w dającej się przewidzieć przyszłości nietypowe i jeżeli twoje specyficzne uwarunkowania nie mogą być zaspokojone przez C++, to warto zainwestować w rozwiązania alternatywne²³. Nawet jeżeli w końcu zdecydujesz się na język C++, to przynajmniej dowiesz się, jakie małeś możliwości i wyrobisz sobie jasno określony pogląd, dlaczego wybrałeś właśnie ten kierunek.

Wiadomo, jak wygląda program proceduralny — składa się z definicji danych i wywołań funkcji. Aby poznać znaczenie takiego programu, musisz się trochę natrudzić, analizując wywołania funkcji i pojęcia niskiego poziomu po to, by utworzyć w swoim umyśle jego model. Jest to powód, dla którego w trakcie projektowania programu proceduralnego potrzebne są reprezentacje pośrednie — programy takie są same w sobie skomplikowane, ponieważ stosowane środki wyrazu są ukierunkowane raczej na komputer niż rozwiązywany problem.

Ponieważ C++ dodaje do języka C wiele nowych pojęć, oczywiste może ci się wydawać założenie, że funkcja **main()** w programie napisanym w C++ będzie znacznie bardziej skomplikowana niż w przypadku jego odpowiednika w języku C. Będziesz mile zaskoczony — dobrze napisany program w C++ jest na ogół znacznie prostszy i łatwiej go zrozumieć niż jego odpowiednik w języku C. Zobaczysz definicje obiektów reprezentujących koncepcje w przestrzeni problemu (a nie kwestie reprezentacji komputerowej) oraz komunikaty wysypane do tych obiektów, reprezentujące działania w tej przestrzeni. Jedną z atrakcyjnych cech programowania obiektowego jest to, że czytając kod dobrze zaprojektowanego programu można go łatwo zrozumieć. Zazwyczaj program ten jest znacznie krótszy, ponieważ wiele problemów można rozwiązać, wykorzystując kod istniejących bibliotek.

²³ W szczególności polecam przyjrzenie się językom: tova (<http://java.sun.com>) raz Python (<http://www.Python.org>).

Rozdział 2.

Tworzenie

i używanie obiektów

W rozdziale tym zostały wprowadzone pojęcia składni C++ oraz konstrukcji programu w stopniu umożliwiającym pisanie i uruchamianie prostych programów obiektowych. W kolejnym rozdziale znajduje się szczegółowy opis podstawowej składni C oraz C++.

Czytając ten rozdział poznasz podstawy programowania z użyciem obiektów w C++, a także odkryjesz niektóre z powodów, dla których język ten spotyka się z entuzjastycznym przyjęciem. Powinno to wystarczyć do przejścia do rozdziału 3., który może być nieco wyczerpujący, gdyż zawiera większość szczegółowych informacji na temat języka C.

Zdefiniowany przez użytkownika typ danych — czyli *klasa* — jest tym, co odróżnia C++ od tradycyjnych języków proceduralnych. Klasa jest nowym typem danych, utworzonym przez ciebie (lub kogoś innego) w celu rozwiązania określonego rodzaju problemów. Po utworzeniu klasy może jej używać każdy, nie znając szczegółów jej funkcjonowania ani nawet nie wiedząc, jak zbudowane są klasy. W rozdziale tym klasy traktowane są w taki sposób, jakby były jeszcze jednym wbudowanym typem danych, którego można używać w programach.

Klasy utworzone przez innych są na ogół połączone w biblioteki. W rozdziale przedstawiono wiele spośród bibliotek klas dostarczanych wraz z wszystkimi implementacjami C++. Szczególnie ważną biblioteką standardową jest biblioteka strumieni wejścia-wyjścia, która (między innymi) pozwala na odczytywanie danych z plików oraz klawiatury, a także zapisywanie ich w plikach oraz na ekranie. Poznasz również bardzo przydatną klasę **string** oraz kontener **vector**, pochodzące ze standardowej biblioteki C++. Kończąc lekturę tego rozdziału przekonasz się, jak łatwo jest używać predefiniowanych bibliotek klas.

Aby utworzyć swój pierwszy program, musisz najpierw poznać narzędzia używane do budowy aplikacji.

Proces tłumaczenia języka

Wszystkie języki komputerowe są tłumaczone z postaci, która jest zazwyczaj łatwa do zrozumienia przez człowieka (*kodu źródłowego*), do postaci wykonywanej przez komputer (*rozkazy komputera*). Translatory zaliczane są tradycyjnie do jednej z dwóch klas — *interpretorów* i *kompilatorów*.

Interprety

Interpreter przekłada kod źródłowy na czynności (mogą one składać się z grup rozkazów komputera), które natychmiast wykonuje. Przykładem popularnego języka interpretowanego jest BASIC. Tradycyjne interpretory BASIC-a tłumaczą i wykonują jednorazowo jeden wiersz programu, a następnie zapominają, że został on przetłumaczony. To czyni je wolnymi, ponieważ muszą one ponownie tłumaczyć każdy powtarzający się kod. Dla zwiększenia szybkości BASIC może być również komplikowany. Nowocześniejsze interpretory, takie jak na przykład interpretory języka Python, tłumaczą cały program na język pośredni, który jest następnie wykonywany przez znacznie szybszy interpreter¹.

Interprety mają wiele zalet. Przejście od pisania kodu do jego wykonania następuje niemal natychmiastowo, natomiast kod źródłowy jest dostępny przez cały czas, dzięki czemu w przypadku wystąpienia błędu interpreter może być znacznie bardziej precyzyjny. Często przytaczanymi zaletami interpretorów są: łatwość interakcji oraz szybkie projektowanie (ale już niekoniecznie wykonywanie) programów.

Języki interpretowane zawierają często szereg ograniczeń związanych z tworzeniem dużych projektów (Python wydaje się tu być wyjątkiem). Interpreter (albo jego zredukowana wersja) musi zawsze podczas wykonywania kodu pozostawać w pamięci. Ponadto wykorzystanie nawet najszybszych interpretorów może się wiązać z niemożliwymi do zaakceptowania ograniczeniami szybkości. Większość interpretorów wymaga przedstawienia do interpretacji od razu całego kodu źródłowego. Powoduje to nie tylko ograniczenia związane z pamięcią, ale może być również przyczyną bardziej złożonych błędów, gdy język nie dostarcza mechanizmów umożliwiających lokalizację oddziaływanego różnych fragmentów kodu.

Kompilatory

Kompilator tłumaczy kod źródłowy bezpośrednio na język asemblera lub rozkazy komputera. Ostatecznym produktem końcowym jest plik lub pliki zawierające język wewnętrzny komputera. Jest to złożony proces, który zazwyczaj odbywa się w kilku etapach. W przypadku kompilatorów droga od napisania kodu do jego uruchomienia jest znacznie dłuższa.

¹ Granica pomiędzy kompilatorami a interpretatorami zaczyna się nieco rozmywać — szczególnie w przypadku języka Python, posiadającego wiele cech oraz siłę języków komplikowanych, ale zarazem szybkość uruchamiania języków interpretowanych.

Programy generowane przez kompilator wymagają zazwyczaj — zależnie od pomyśliwości autorów — mniej lub więcej pamięci i uruchamiają się o wiele szybciej. Mimo że wielkość i szybkość programów są prawdopodobnie najczęściej przytaczanymi powodami używania kompilatorów, w wielu przypadkach nie są one przyczynami najważniejszymi. Niektóre języki (takie jak C) zostały zaprojektowane w sposób umożliwiający niezależną komplikację poszczególnych części programu. Części te są ostatecznie łączone w jeden program *wykonywalny* za pomocą narzędzia zwanego *programem łączącym* (ang. *linker*). Proces ten nosi nazwę *rozłącznej komplikacji*.

Rozłączna komplikacja przynosi wiele korzyści. Program, który kompilowany od razu w całości mógłby przekroczyć ograniczenia kompilatora albo środowiska komplikacji, może być skompilowany w oddzielnich częściach. Programy można tworzyć i testować partiami. Kiedy jakaś część programu już działa, może być zachowana i traktowana jako budulec dla jego kolejnych części. Zbiory przetestowanych i działających elementów można połączyć w *biblioteki* przeznaczone do użycia przez innych programistów. W trakcie tworzenia każdej części programu złożoność jego pozostałych części pozostaje ukryta. Wszystkie te cechy wspomagają tworzenie wielkich programów².

Stopniowo w kompilatorach znacznie udoskonalano mechanizmy ułatwiające usuwanie błędów. Wczesne kompilatory generowały jedynie język wewnętrzny komputera, a programista wstawiał do programu instrukcje print, by zorientować się, co się dzieje. Nie zawsze postępowanie to jest efektywne. Nowoczesne kompilatory mogą wstawać do wykonywalnego programu informacje dotyczące kodu źródłowego. Informacje te są wykorzystywane przez potężne *programy uruchomieniowe działające na poziomie kodu źródłowego* (ang. *source-level debuggers*) do prezentacji tego, co rzeczywiście wydarzyło się w programie — śledzącego przebieg w kodzie źródłowym.

Niektóre kompilatory radzą sobie z problemem szybkości komplikacji, dokonując *kompilacji w pamięci*. Większość kompilatorów pracuje z plikami — czytając i zapisując je na każdym etapie procesu komplikacji. Urządzenia wykonujące komplikację w pamięci przechowują kompilowany program w pamięci RAM. Dzięki temu uruchamianie niewielkich programów może się wydawać równie szybkie, jak w przypadku interpreterów.

Proces komplikacji

Aby programować w C i C++, trzeba zrozumieć poszczególne kroki, a także sposób działania narzędzi używanych w procesie komplikacji. Niektóre języki (w szczególności C i C++) rozpoczynają komplikację od uruchomienia *preprocesora* kodu źródłowego. Preprocesor jest prostym programem, zamieniającym wzorce napotkane w kodzie źródłowym na inne wzorce, zdefiniowane przez programistę (za pomocą *dyrektyw preprocesora*). Dyrektywy preprocesora są stosowane po to, aby oszczędzić sobie trudu pisania oraz zwiększyć czytelność kodu (w dalszej części książki dowiesz się, jak w projekcie C++ zapobiega się używaniu preprocesora w większości przypadków z uwagi na możliwość popełnienia trudno uchwytnych błędów). Przetworzony wstępnie kod jest zwykle zapisywany w pliku pośrednim.

² Python znowu jest tutaj wyjątkiem, umożliwia bowiem również rozłączną komplikację.

Kompilatory wykonują zazwyczaj swoją pracę w dwóch przebiegach. W pierwszym przebiegu dokonywana jest *analiza składniowa* (ang. *parsing*) przetworzonego wstępnie kodu. Kompilator rozbija kod źródłowy na małe jednostki i porządkuje go, używając struktury nazywanej *drzewem*. W wyrażeniu „A + B” elementy „A”, „+” i „B” są liśćmi drzewa składniowego.

Czasami pomiędzy pierwszym a drugim przebiegiem komplikacji używany jest *optymalizator globalny* (ang. *global optimizer*), umożliwiający uzyskanie krótszego i szybszego kodu.

W drugim przebiegu *generator kodu* przechodzi przez drzewo składniowe, tworząc dla poszczególnych węzłów drzewa kod w języku asemblera albo w języku wewnętrznym komputera. Jeżeli generator tworzy kod w języku asemblera, to w następnej kolejności musi zostać uruchomiony asembler. Końcowym rezultatem jest w obu przypadkach program wynikowy (plik, który ma na ogół rozszerzenie **.o** lub **.obj**). W drugim przebiegu używany jest czasem *optymalizator lokalny* (ang. *peephole optimizer*), poszukujący fragmentów kodu zawierających nadmiarowe instrukcje języka asemblera.

Program łączący (ang. *linker*) łączy ze sobą listę programów wynikowych w program wykonywalny, który może być załadowany i uruchomiony przez system operacyjny. Kiedy funkcja w jakimś programie wynikowym odwołuje się do funkcji lub zmiennej znajdującej się w innym programie wynikowym, odwołania te są rozwiązywane przez program łączący — sprawdza on, czy istnieją wszystkie zewnętrzne funkcje i dane, których istnienia zażądano w czasie komplikacji. Program łączący dodaje również specjalny kod, wykonujący pewne działania w czasie uruchamiania programu.

Program łączący, rozwiązując wszystkie odwołania, może przeszukiwać specjalne pliki nazywane *bibliotekami* (ang. *libraries*). Biblioteka składa się ze zbioru programów wynikowych, połączonych w pojedynczy plik. Biblioteki są tworzone i obsługiwane za pomocą *programów zarządzających bibliotekami* (ang. *librarians*).

Statyczna kontrola typów

W czasie pierwszego przebiegu kompilator przeprowadza *kontrolę typów*. Polega ona na sprawdzeniu poprawności użycia argumentów funkcji oraz zapobiega wielu rodzajom błędów programistycznych. Ponieważ kontrola typów odbywa się w czasie komplikacji, a nie w czasie wykonywania programu, jest nazywana *statyczną kontrolą typów*.

Niektóre języki obiektowe (szczególnie Java) dokonują pewnego rodzaju kontroli w czasie pracy programu (nazywa się to *dynamiczną kontrolą typów*). Dynamiczna kontrola typów jest — w połączeniu ze statyczną kontrolą typów — bardziej skuteczna niż sama tzw statyczna. Jednakże wiąże się ona z pewnym dodatkowym obciążeniem w czasie wykonania programu.

C++ wykorzystuje statyczną kontrolę typów, ponieważ język nie może założyć żadnego konkretnego wsparcia błędnych operacji w czasie wykonywania programu. Statyczna kontrola typów informuje programistę o niewłaściwym użyciu typów w czasie komplikacji, maksymalizując szybkość wykonania programu. W miarę poznawania C++ przekonasz się, że decyzje projektowe języka wspierały w większości ten sam rodzaj szybkiego, ukierunkowanego na produkcję programowania, z którego słynie język C.

Można wyłączyć staticzną kontrolę typów w języku C++. Można również samodzielnie przeprowadzić dynamiczną kontrolę typów — wystarczy tylko napisać odpowiedni kod.

Narzędzia do rozłącznej kompilacji

Rozłączna kompilacja jest szczególnie ważna w razie budowy dużych projektów. W przypadku języków C oraz C++ program może zostać utworzony w postaci małych, możliwych do ogarnięcia, niezależnie testowanych modułów. Najbardziej podstawowym narzędziem, pozwalającym na rozbicie programu na moduły, jest możliwość tworzenia nazwanych procedur lub podprogramów. W językach C i C++ podprogramy są nazywane *funkcjami*, stanowiącymi fragmenty kodu, który może być umieszczony w różnych plikach, dzięki czemu możliwa jest rozłączna kompilacja. Innymi słowy — funkcje stanowią elementarną jednostkę kodu, ponieważ nie jest możliwa sytuacja, w której różne fragmenty tej samej funkcji znajdowałyby się w różnych plikach. Funkcja musi być w całości umieszczona w pojedynczym pliku (choć pliki mogą zawierać więcej niż jedną funkcję).

W czasie wywoływania funkcji na ogół przekazywane są do niej pewne *argumenty* będące wartościami, z którymi ma ona pracować wówczas, gdy będzie wykonywana. Po zakończeniu funkcji przekazuje ona zazwyczaj *wartość zwracaną*, czyli wartość, która jest przesyłana z powrotem jako rezultat jej działania. Możliwe jest również napisanie funkcji, która nie pobiera argumentów ani nie zwraca żadnej wartości.

Podczas tworzenia programu składającego się z wielu plików funkcje w jednym pliku muszą mieć dostęp do funkcji i danych znajdujących się w innym pliku. W trakcie kompilacji pliku kompilator C lub C++ musi również wiedzieć o funkcjach i danych znajdujących się w innych plikach — w szczególności o tym, jakie są ich nazwy i poprawne użycie. Kompilator upewnia się, że funkcje oraz dane są używane poprawnie. Proces „informowania kompilatora” o nazwach zewnętrznych funkcji i danych oraz o tym, jak powinny one wyglądać, nazywany jest *deklaracją*. Po zadeklarowaniu funkcji lub zmiennej kompilator wie, w jaki sposób sprawdzić, czy jest ona właściwie używana.

Deklaracje i definicje

Ważne jest, by rozumieć różnicę pomiędzy *deklaracjami* i *definicjami*, ponieważ pojęcia te będą używane precyzyjnie w dalszej części książki. Na ogół wszystkie programy napisane w C i C++ wymagają deklaracji. Zanim napiszesz swój pierwszy program, musisz zapoznać się z poprawnym zapisem deklaracji.

Deklaracja przedstawia kompilatorowi nazwę — identyfikator. Komunikuje ona kompilatorowi: „ta funkcja lub zmienna istnieje w jakimś miejscu i powinna tak właśnie wyglądać”. Z kolei *definicja* nakazuje: „utwórz w tym miejscu zmienną” albo „utwórz tutaj tę funkcję”. Przydziela ona nazwie pamięć. Następuje to niezależnie od tego, czy wchodzi w grę zmienna czy funkcja — w każdym przypadku w miejscu

wystąpienia definicji kompilator przydziela pamięć. W przypadku zmiennej, kompilator określa, jaka jest jej wielkość i powoduje utworzenie w pamięci miejsca, przeznaczonego do przechowywania jej danych. W przypadku funkcji — kompilator tworzy kod, który w końcu również zajmuje miejsce w pamięci.

Zmienna lub funkcja mogą być zadeklarowane w wielu różnych miejscach, ale zarówno w C, jak i w C++ może wystąpić tylko jedna definicja (jest to czasami nazywane *regułą jednejdefinicji*). Kiedy program łączący scalą wszystkie programy wynikowe, to na ogół zgłasza błąd w sytuacji, gdy napotka więcej niż jedną definicję tej samej funkcji lub zmiennej. X

Definicja może być również deklaracją. Jeżeli kompilator nie napotkał wcześniej nazwy **x**, a ty wprowadzisz definicję `int x;`, kompilator potraktuje tę nazwę jako deklarację i równocześnie przydzieli zmiennej pamięć.

Składnia deklaracji funkcji

Deklaracja funkcji w C oraz C++ zawiera nazwę funkcji oraz typy przekazywanych jej argumentów i zwracanej wartości. Na przykład poniżej znajduje się deklaracja funkcji o nazwie **func1()**, która pobiera dwa całkowite argumenty (liczby całkowite są oznaczane w C i C++ za pomocą słowa kluczowego `int`) oraz zwraca wartość całkowitą:

```
int func1(int,int);
```

Pierwsze widoczne słowo kluczowe jest zwracaną wartością: `int`. Argumenty zawarte są w nawiasie, znajdująącym się po nazwie funkcji, w kolejności, w której są używane. Średnik sygnalizuje koniec instrukcji — w tym przypadku komunikuje kompilatorowi: „to już wszystko, definicja funkcji się skończyła!”.

Deklaracje języków C i C++ próbują naśladować sposób użycia obiektu. Na przykład — jeżeli a jest zmienną całkowitą, to powyższa funkcja może być użyta w następujący sposób:

```
a = func1(2,3);
```

Ponieważ **func1()** zwraca wartość całkowitą, kompilatory C i C++ sprawdzą sposób użycia **func1()**, by upewnić się, że a może przyjąć zwracaną wartość, a argumenty są podane właściwie.

Argumenty wymienione w deklaracjach funkcji mogą mieć nazwy. Kompilator je ignoruje, lecz mogą one być pomocne, przypominając użytkownikowi znaczenie parametrów. Na przykład możemy zadeklarować funkcję **func1()** w nieco innym stylu, mającym jednak to samo znaczenie:

```
int func1(int dlugosc, int szerokosc);
```

Pułapka

Istnieje zasadnicza różnica pomiędzy C i C++ w przypadku funkcji z pustymi listami argumentów. W języku C deklaracja:

```
int func2();
```

oznacza: „funkcję z dowolną liczbą argumentów dowolnego typu”. Uniemożliwia to kontrolę typów, dlatego też w C++ zapis taki oznacza „funkcję bez argumentów”.

Definicje funkcji

Definicja funkcji przypominacej deklarację, z tajednak różnicą, że zawiera ona treść (ciało) funkcji. Ciało funkcji jest zbiorem instrukcji, zamkniętym w nawiasie klamrowym. Nawiasy klamrowe symbolizują początek oraz koniec bloku kodu. Aby określić dla funkcji **func1()** definicję, będącą pustym ciałem (ciałem niezawierającym kodu), należy napisać:

```
int func1(int dlugosc, int szerokosc) { }
```

Zwrócić uwagę na to, że nawiasy klamrowe w definicji funkcji zastępują średnik. Ponieważ nawiasy klamrowe zawierają instrukcję lub grupę instrukcji, średnik nie jest potrzebny. Warto również zauważyć, że jeżeli chcesz używać argumentów w ciele funkcji, to muszą one posiadać nazwy (w powyższym przypadku, ponieważ nigdy nie są używane, mają charakter opcjonalny).

Składnia deklaracji zmiennej

Pojęciu „deklaracja zmiennej” przypisywano w przeszłości sprzeczne i mylące znaczenia, dlatego ważne jest zrozumienie właściwej definicji, co umożliwi poprawne odczytywanie kodu. Deklaracja zmiennej informuje kompilator o tym, jak ta zmienna wygląda. Oznacza ona: „wiem, że jeszcze nie spotkałeś tej nazwy, ale z pewnością ona gdzieś istnieje i jest zmienną typu X”.

Wraz z deklaracją funkcji podaje się jej typ (zwracaną wartość), nazwę, listę argumentów i kończy się ją średnikiem. To wystarcza kompilatorowi, by zrozumiał, że jest to deklaracja, i wiedział, jak powinna wyglądać ta funkcja. Można z tego wysnuć wniosek, że deklaracja zmiennej jest typem, po którym następuje nazwa. Na przykład:

```
int a;
```

mogłoby, zgodnie z powyższą logiką, stanowić deklarację zmiennej a będącej liczbą całkowitą. Występuje tu jednak konflikt – powyższy kod zawiera wystarczająco dużo informacji, aby kompilator zarezerwował pamięć dla zmiennej całkowitej o nazwie a, i do takiej właśnie sytuacji dochodzi. Aby rozwiązać ten dylemat, zarówno w przypadku C, jak i C++, konieczne jest słowo kluczowe oznaczające: „to tylko deklaracja, zmienna została zdefiniowana w innym miejscu”. Tym słowem kluczowym jest **extern**. Może ono oznaczać, że albo definicja znajduje się w innym pliku, albo występuje ona w jego dalszej części.

Deklaracja zmiennej bez jej definiowania oznacza użycie słowa kluczowego **extern** przed opisem tej zmiennej, tak jak poniżej:

```
extern int a;
```

Słowo kluczowe **extern** może być również zastosowane w stosunku do deklaracji funkcji. W przypadku funkcji **func1()** wygląda to następująco:

```
extern int func1(int dlugosc, int szerokosc);
```

Instrukcja ta jest równoważna poprzednim deklaracjom **func1()**. Ponieważ nie występuje tutaj ciało funkcji, kompilator musi traktować ją jako deklarację, a nie definicję funkcji. Dlatego też słowo kluczowe **extern** jest w przypadku deklaracji funkcji nadmiarowe i zarazem opcjonalne. Chyba nie najlepiej się stało, że twórcy języka C nie wymusili użycia słowa kluczowego **extern** w przypadku deklaracji funkcji — zapewniłoby to większą spójność i przejrzystość (ale wymagałoby więcej pisania, co zapewne wyjaśnia ich decyzję).

Poniżej przedstawiono nieco więcej przykładów deklaracji:

```
//: C02:Declare.cpp
// Przykłady deklaracji i definicji
extern int i; // Deklaracja bez definicji
extern float f(float); // Deklaracja funkcji
float b; // Deklaracja i definicja
float f(float a) { // Definicja
    return a + 1.0;
}

int i; // Definicja
int h(int x) { // Deklaracja i definicja
    return x + 1;
}

int main() {
    b = 1.0;
    i = 2;
    f(b);
    h(i);
} // /~/
```

W deklaracjach funkcji identyfikatory ich argumentów są opcjonalne. Natomiast w definicjach są one konieczne (identyfikatory argumentów wymagane są jedynie w C, a nie w C++).

Dołączanie nagłówków

Większość bibliotek zawiera pokaźną liczbę funkcji i zmiennych. Aby ułatwić pracę i zapewnić spójność podczas tworzenia zewnętrznych deklaracji tych elementów, języki C oraz C++ używają mechanizmu nazywanego *plikiem nagłówkowym* (ang. *header file*). Plik nagłówkowy jest plikiem zawierającym zewnętrze deklaracje biblioteki — tradycyjnie rozszerzeniem nazwy tego pliku jest „h”, tak jak w przypadku pliku **headerfile.h** (w starszych programach używa się innych rozszerzeń, takich jak **.hxx** lub **.hpp**, ale występują one rzadko).

Programista tworzący bibliotekę udostępnia plik nagłówkowy. Aby zadeklarować funkcje oraz zewnętrzne zmienne znajdujące się w bibliotece, użytkownik dołącza plik nagłówkowy. Aby to uczynić, należy użyć dyrektywy preprocesora **#include**. Nakazuje ona preprocesorowi, by otworzył wymieniony plik nagłówkowy i wstawił jego zawartość w miejscu, w którym znajduje się instrukcja **#include**. W dyrektywie **#include** nazwę pliku można podać na dwa sposoby — w nawiasie kątowym (**<>**) lub w cudzysłowie.

Podanie nazw plików w nawiasie kątowym, tak jak w przykładzie poniżej:

```
#include <header>
```

powoduje, że preprocesor poszukuje pliku w sposób zależny od implementacji, ale zazwyczaj wiąże się to z jakimś rodzajem „ścieżki wyszukiwania”, określonej w środowisku lub podanej kompilatorowi w wierszu polecenia. Mechanizm określania ścieżki wyszukiwania zależy od rodzaju komputera, systemu operacyjnego oraz implementacji C++ i może wymagać sprawdzenia.

Podanie nazwy pliku w cudzysłowie, jak w przykładzie:

```
#include "local.h"
```

informuje preprocesor, by poszukał pliku w (zgodnie ze specyfikacją) „sposób zdefiniowany w implementacji”. Oznacza to zazwyczaj poszukiwanie pliku w stosunku do bieżącego katalogu. Jeżeli plik nie zostanie znaleziony, to dyrektywa dołączenia jest przetwarzana ponownie w taki sposób, jak gdyby występował w niej nawias kątowy, a nie znaki cudzysłowu.

Aby dołączyć plik nagłówkowy „iostream”, należy napisać:

```
#include <iostream>
```

Preprocesor znajdzie plik nagłówkowy „iostream” (przeważnie w katalogu o nazwie „include”) i dołączy go.

Standardowy format dołączania plików w C++

W miarę rozwoju języka C++ różni producenci kompilatorów nadawali nazwom plików różne rozszerzenia. Ponadto rozmaite systemy operacyjne nakładały na nazwy plików odmienne ograniczenia, w szczególności dotyczące ich długości. Powodowało to problemy z przenośnością kodu źródłowego. Aby zniwelować te różnice, standard stosuje format dopuszczający nazwy plików dłuższe niż niesławne osiem znaków oraz eliminuje ich rozszerzenia. Na przykład zamiast dawnego stylu dołączania pliku **iostream.h**, który wyglądał następująco:

```
#include <iostream.h>
```

można obecnie pisać:

```
#include <iostream>
```

Translator może implementować instrukcję dołączania w sposób odpowiadający potrzebom określonego kompilatora i systemu operacyjnego, w razie potrzeby skracając nazwę i dodając do niej rozszerzenie. Oczywiście, jeżeli chcesz używać takiego stylu, zanim producent kompilatora zapewni dla niego wsparcie, to możesz również skopiać pliki nagłówkowe udostępnione przez producenta, usuwając z nich nazwy rozszerzenia.

Biblioteki odziedziczone z C są nadal dostępne pod nazwami z tradycyjnym rozszerzeniem „**h**”. Możesz ich jednak również używać za pomocą bardziej nowoczesnego stylu dołączania C++, poprzedzając ich nazwy literą „**c**”. A zatem dyrektywy:

```
#include <stdio.h>
#include <stdlib.h>
```

można zastąpić przez:

```
#include <cstdio>
#include <cstdlib>
```

I tak dalej, dla wszystkich standardowych plików nagłówkowych C. Pozwala to czytelnikowi na eleganckie **rozróżnienie**, czy używane są biblioteki C czy C++.

Rezultat zastosowania nowego formatu dołączania plików nie jest tożsamy z luźcikiem starego formatu — posługiwanie się rozszerzeniem „**h**” powoduje dołączenie dawnej wersji, pozbawionej szablonów, natomiast pominięcie „**h**” dołącza nową wersję, wykorzystującą szablony. Próba połączenia obu tych form w pojedynczym programie powoduje na ogół kłopoty.

Łączanie

Program łączący zestawia ze sobą programy wynikowe (często z rozszerzeniami nazw plików „**o**” lub „**obj**”), utworzone przez kompilator, czego wynikiem jest program wykonywalny, który system operacyjny może załadować i uruchomić. Jest to ostatni etap procesu komplikacji.

Cechy programów łączących zmieniają się w zależności od systemu. Na ogół, aby program łączący wykonał swe zadanie, wystarczy mu po prostu podać nazwy programów wynikowych oraz bibliotek, które mają być ze sobą połączone, a także nazwę pliku wykonywalnego. Niektóre systemy wymagają od użytkownika samodzielnego uruchomienia programu łączącego. W większości pakietów C++ program łączący jest wywoływany za pośrednictwem kompilatora C++. W wielu przypadkach program łączący jest uruchamiany w sposób niewidoczny dla użytkownika.

Niektóre ze starszych programów szukają programów wynikowych i bibliotek tylko jednokrotnie, przeszukując podaną listę plików od lewej do prawej strony. Oznacza to, że kolejność programów wynikowych oraz bibliotek może mieć znaczenie. Jeżeli napotkałeś jakiś tajemniczy problem, który nie ujawnił się aż do etapu łączenia, to jedną z możliwych przyczyn jest kolejność, w jakiej pliki zostały podane programowi łączącemu.

Używanie bibliotek

Po zapoznaniu się z podstawową terminologią jesteś w stanie zrozumieć, w jaki sposób używać bibliotek. Aby korzystać z biblioteki, należy:

1. Dołączyć plik nagłówkowy biblioteki.
2. Użyć funkcji i zmiennych, zawartych w bibliotece.
3. Dołączyć bibliotekę do programu wykonywalnego.

Te same kroki obowiązują również w przypadku, gdy programy wynikowe nie są scalone w bibliotekę. Dołączenie pliku nagłówkowego oraz połączenie programów wynikowych są podstawowymi etapami rozłącznej komplikacji, zarówno w C, jak i w C++.

Jak program łączący przeszukuje bibliotekę?

Kiedy odwołujesz się w języku C lub C++ do zewnętrznej funkcji lub zmiennej, program łączący, napotykając to odwołanie, może zadziałać dwojako. Jeżeli do tej pory nie spotkał definicji funkcji lub zmiennej, to dodaje jej identyfikator do listy „nieustalonych odwołań”. Jeżeli natomiast program łączący napotkał już definicję, odwołanie jest uznawane za ustalone.

Jeżeli program łączący nie może znaleźć definicji na liście programów wynikowych, przeszukuje biblioteki. Biblioteki są w jakiś sposób poindeksowane; program łączący nie musi więc przeglądać wszystkich programów wynikowych zawartych w bibliotece, lecz tylko jej indeks. Kiedy program łączący znajdzie w bibliotece definicję, wówczas cały program wynikowy (a nie tylko definicja funkcji) zostaje dołączony do programu wykonywalnego. Zwróć uwagę na to, że nie jest dołączana cała biblioteka, a tylko zawarty w niej program wynikowy, zawierający żądaną definicję (w przeciwnym przypadku programy byłyby niepotrzebnie duże). Jeżeli chcesz zmniejszyć wielkość programu wykonywalnego, to możesz rozważyć umieszczenie poszczególnych funkcji w oddzielnych plikach kodu — na etapie tworzenia własnych bibliotek. Wymaga to większych nakładów na edycję³, lecz może być pomocne dla użytkownika.

Ponieważ program łączący przeszukuje pliki w kolejności, w jakiej zostały one podane, możesz wykluczyć używanie funkcji znajdującej się w bibliotece, umieszczając na liście — przed wystąpieniem nazwy biblioteki — plik zawierający własną funkcję o tej samej nazwie. Zanim program łączący przeszuka bibliotekę, ustali on wszelkie odwołania do tej funkcji, wykorzystując swoją funkcję. Zostanie ona użyta w miejscu funkcji zawartej w bibliotece. Zwróć uwagę na to, że wystąpienie takiej sytuacji może być również błędem; chroniąc przed nią przestrzenie nazw C++.

Sekretne dodatki

Kiedy tworzony jest wykonywalny plik C lub C++, w niejawny sposób dołączane są do niego pewne elementy. Jednym z nich jest moduł startowy, zawierający procedury inicjalizujące, które muszą być wykonane za każdym razem, gdy uruchamiany jest program C lub C++. Procedury te przygotowują stos i inicjują w programie pewne zmienne.

Program łączący zawsze poszukuje w standardowej bibliotece skompilowanej wersji każdej „standardowej” funkcji, wywoływanej w programie. Ponieważ standardowa biblioteka jest zawsze przeszukiwana, można użyć jej dowolnego elementu, dołączając po prostu do swojego programu odpowiedni plik nagłówkowy — bez potrzeby informowania o konieczności przeszukiwania standardowej biblioteki. Na przykład funkcje strumieni wejścia-wyjścia znajdują się w standardowej bibliotece C++. Aby ich użyć, wystarczy dołączyć do swojego programu plik nagłówkowy **<iostream>**.

Jeżeli natomiast używasz jakiejś dodatkowej biblioteki, to musisz wyraźnie umieścić jej nazwę na liście plików, przekazywanych programowi łączącemu.

³

Polecałbym użycie Perla lub Pythona, w celu zautomatyzowania tego zadania — jako części procesu tworzenia bibliotek (patrz www.Perl.org lub www.Python.org).

Używanie bibliotek napisanych w czystym C

To, że piszesz kod w języku C++ nie oznacza wcale, że nie możesz używać funkcji bibliotecznych C. W rzeczywistości biblioteka C jest w całości włączona do standardej biblioteki C++. Przygotowując te funkcje wykonano ogromną pracę, dzięki czemu pozwalają one na zaoszczędzenie mnóstwa czasu.

W książce używane będą funkcje standardowej biblioteki C++ (a zatem również stan-dardowej biblioteki C), kiedy to będzie wygodne, ale, dla zapewnienia przenośności programów, stosowane będą jedyne *standardowe* funkcje biblioteczne. W niektórych przypadkach, kiedy należy użyć funkcji bibliotecznych *nieznajdujących się* w standardzie C++, dołożono wszelkich starań, by wykorzystane były funkcje zgodne z POSIX. POSIX jest standardem bazującym na wysiłkach standaryzacyjnych Uniksa, zawierającym funkcje wykraczające poza zakres biblioteki C++. Na ogół można spodziewać się obecności funkcji standardu POSIX na platformach uniksowych (w szczególności — w Linuksie), a często dostępne są one również w systemach DOS i Windows. Na przykład jeżeli używasz wielowątkowości, to korzystając z biblioteki wątków POSIX jesteś w lepszej sytuacji, ponieważ dzięki temu twój kod jest łatwiejszy do zrozumienia, przeniesienia i pielęgnacji (a biblioteka wątków POSIX będzie zazwyczaj wykorzystywała własności wielowątkowości systemu operacyjnego, o ile są one przez system udostępniane).

Twój pierwszy program w C++

Dysponujesz już niemal wszystkimi podstawowymi wiadomościami, niezbędnymi do utworzenia i skompilowania programu. Program będzie używał klas strumieni wejścia-wyjścia (ang. *iostream* — *input-output stream*), należących do standardu C++. Umożliwiają one odczytywanie i zapisywanie plików, a także „standardowego” wejścia i wyjścia (które są zwykle związane z konsolą, ale mogą być również przekierowane do plików lub innych urządzeń). W naszym prostym programie do wyświetlenia komunikatu na ekranie zostanie użyty obiekt będący strumieniem.

Używanie klasy strumieni wejścia-wyjścia

Aby zadeklarować funkcje oraz zewnętrzne dane zawarte w klasie strumieni wejścia-wyjścia, należy dołączyć plik nagłówkowy, używając instrukcji:

```
#include <iostream>
```

Pierwszy program używa pojęcia standardowego wyjścia, które oznacza „miejscie ogólnego przeznaczenia, służące do wysyłania informacji wyjściowych”. Później zostaną przedstawione przykłady, w których standardowe wyjście jest użyte w odmien-ny sposób, ale w tym wypadku informacje będą po prostu przekazywane na konsolę. Pakiet strumieni wejścia-wyjścia automatycznie definiuje zmienną (obiekt) o nazwie **cout**, który przyjmuje wszystkie dane skierowane na standardowe wyjście.

Do wyprowadzania danych na standardowe wyjście używany jest operator `<<`. Programujący w języku C znają ten operator jako „bitowy operator przesunięcia w lewo”, który zostanie opisany w następnym rozdziale. Wystarczy wiedzieć, że przesunięcie bitów w lewo nie ma nic wspólnego z wyjściem. Jednakże język C++ pozwala na *przeciążanie* (ang. *overloading*) operatorów. Przeciążając operator, nadaje się mu nowe znaczenie w przypadku, gdy jest on używany z obiektami określonego typu. W połączeniu z obiektami klasy strumieni wejścia-wyjścia operator `<<` oznacza „wyślij do”. Na przykład polecenie:

```
cout << "witaj!" ;
```

wysyła napis „witaj!” do obiektu o nazwie `cout` (co jest skrótem nazwy „console output”, oznaczającej po angielsku wyjście konsoli).

Przeciążanie operatorów opisano szczegółowo w rozdziale 12.

Przestrzenie nazw

Jak już wspomniano w rozdziale 1., jednym z głównych problemów spotykanych w języku C jest „wyczerpywanie się nazw” funkcji i identyfikatorów, gdy program osiąga pewną wielkość. Oczywiście, nazwy tak naprawdę wcale się nie kończą — jednak coraz bardziej staje się szybkie wymyślanie nowych. Co ważniejsze, gdy program osiąga pewną wielkość, jest na ogół dzielony na części, z których każda jest tworzona i utrzymywana przez inną osobę lub grupę. Ponieważ język C ma w rzeczywistości tylko jeden obszar, w którym funkcjonują wszystkie identyfikatory i nazwy funkcji, oznacza to, że programiści muszą zachować ostrożność, by przypadkowo nie użyć tych samych nazw w sytuacjach, w których mogą one ze sobą kolidować. Szybko staje się to uciążliwe, czasochłonne i — w ostatecznym rozrachunku — kosztowne.

Standardowe C++ posiada mechanizm zapobiegający takim kolizjom — słowo kluczowe `namespace` (od angielskich słów „name space” — przestrzeń nazw). Każdy zbiór definicji zawartych w bibliotece lub programie jest „opakowany” w przestrzeń nazw; w sytuacji, gdy jakaś inna definicja nosi tę samą nazwę, ale znajduje się w innej przestrzeni nazw, nie powoduje to kolizji.

Przestrzenie nazw są wygodnym i pomocnym narzędziem, ale ich obecność powoduje, że należy o nich wiedzieć przed podjęciem próby napisania jakikolwiek programu. Jeżeli dołączysz po prostu plik nagłówkowy i użyjesz funkcji lub obiektów pochodzących z tego pliku, to prawdopodobnie podczas próby komplikacji programu otrzymasz dziwnie wyglądające błędy, oznajmiające, że kompilator nie znalazł żadnych deklaracji obiektów, które dołączyłeś przecież w pliku nagłówkowym! Kiedy już zobaczyłeś ten komunikat kolejny raz, jego znaczenie stanie się dla ciebie oczywiste: „dołączyłeś plik nagłówkowy, ale wszystkie deklaracje znajdują się wewnątrz przestrzeni nazw, a ty nie zapowiedziałeś kompilatorowi, że zamierzasz używać deklaracji w tej właśnie przestrzeni nazw”.

Istnieje słowo kluczowe, dzięki któremu można powiedzieć: „chcę używać deklaracji i (lub) definicji, znajdujących się w tej przestrzeni nazw”. Tym właściwym słowem jest `using`. Wszystkie standardowe biblioteki C++ są zawarte w pojedynczej przestrzeni

nazw, którą jest **std** (skrót od „standard”). Ponieważ w książce używane są niemal wyłącznie standardowe biblioteki, niemal w każdym programie widnieje następująca dyrektywa **using**:

```
using namespace std;
```

Oznacza ona, że zamierzamy odsłonić wszystkie elementy zawarte w przestrzeni nazw **std**. Po tej instrukcji nie trzeba się obawiać o to, że jakiś szczególny składnik biblioteki znajduje się wewnętrz przestrzeni nazw. Dyrektywa **using** powoduje bowiem, iż wymieniona w niej przestrzeń nazw jest dostępna aż do końca pliku, w którym dyrektywa ta została umieszczona.

Odsłonięcie wszystkich elementów przestrzeni nazw po tym, jak ktoś zadał sobie trud ich ukrycia, może wydawać się szkodliwe i należy zachować ostrożność przed pochopnym ujawnianiem przestrzeni nazw (przekonasz się o tym w dalszej części książki). Jednakże dyrektywa **using** odsłania te nazwy jedynie w obrębie bieżącego pliku, jej działanie nie jest więc tak radykalne, jak mogłoby to się wydawać na pierwszy rzut oka (ale i tak poważnie się zastanów, zanim lekkomyślnie zrobisz to w pliku nagłówkowym).

Istnieje związek pomiędzy przestrzeniami nazw i sposobem, w jaki dołączane są pliki nagłówkowe. Przed standaryzacją nowoczesnego stylu dołączania plików nagłówkowych (bez końcowego „**h**”), jak w **<iostream>**) typowe było dołączanie plików nagłówkowych z rozszerzeniem „**h**” — jak np. **<iostream.h>**. Przestrzenie nazw również nie były wówczas elementem języka. Zapis dokonany w celu zapewnienia wstępnej zgodności z istniejącym kodem:

```
#include <iostream.h>
```

oznacza:

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

Jednakże w książce używany będzie standardowy format dołączania plików nagłówkowych (bez „**h**”), dlatego też konieczne będzie stosowanie dyrektywy **using**.

W rozdziale 10. temat przestrzeni nazw jest opisany znacznie bardziej gruntownie.

Podstawy struktury programu

Program w C lub w C++ jest zbiorem zmiennych, definicji funkcji oraz wywołań funkcji. Kiedy program rozpoczyna działanie, uruchamia kod inicjujący i wywołuje specjalną funkcję „**main()**”. W tym miejscu umieszczany jest główny kod programu.

Jak już wspomniano wcześniej, definicja funkcji składa się ze zwanego typu (który musi być określony w C++), nazwy funkcji, listy argumentów w nawiasie oraz kodu funkcji, zawartego w nawiasie klamrowym. Oto przykładowa definicja funkcji:

```
int function() {
    // Tu znajduje się kod funkcji (to jest komentarz)
}
```

Powyższa funkcja posiada pustą listę argumentów oraz ciało, zawierające jedynie komentarz.

W obrębie definicji funkcji może występować wiele nawiasów klamrowych, ale przy najmniej jeden z nich musi obejmować ciało funkcji. Ponieważ **main()** jest funkcją, to musi również stosować się do tych zasad. W C++ funkcja **main()** zawsze zwraca wartość typu **int**.

C oraz C++ są językami o swobodnej strukturze. Kompilator (z kilkoma wyjątkami) ignoruje znaki nowego wiersza i odstępu, musi więc w jaki sposób określić miejsca, w którym kończy się instrukcja. Instrukcje są oddzielone średnikami.

W języku C komentarze rozpoczynają się znakami /*, a kończą — parą znaków */. Mogą one zawierać znaki nowego wiersza. Język C++ używa komentarzy w stylu C, a ponadto ma dodatkowy typ komentarzy: //. Para znaków // rozpoczęta komentarzem, który kończy się znakiem nowego wiersza. W przypadku komentarzy jednowierszowych jest to wygodniejsze niż /* */. Komentarze takie są często używane w książce.

,Witaj, świecie!"

Przedstawiamy wreszcie pierwszy program:

```
//: C02:He110.cpp
// Przywitanie za pomocą C++
#include <iostream> // Deklaracje strumieni
using namespace std;

int main() {
    cout << "Witaj, świecie! Mam dzisiaj "
        << 8 << " urodziny!" << endl;
} ///-
```

Wiele argumentów jest przekazywanych obiektowi **cout** za pomocą operatorów „<<”. Drukuje on te argumenty w kolejności od lewej do prawej. Specjalna funkcja strumienia wejścia-wyjścia — **endl** — wyprowadza wiersz oraz znak nowego wiersza. Korzystając z klasy strumieni wejścia-wyjścia, można połączyć ze sobą ciąg argumentów, tak jak w powyższym przykładzie, co czyni tę klasę łatwą w użyciu.

W języku C tekst zawarty w **cudzysłowie** jest tradycyjnie nazywany „łańcuchem” (**ang. string**). Ponieważ jednak standardowa biblioteka C++ zawiera potężną klasę o nazwie **string**, umożliwiającą operacje na tekstach, w stosunku do tekstu zawartego w cudzysłowie będzie używany bardziej precyzyjny termin — **tablica znakowa**.

Kompilator tworzy miejsce w pamięci, przeznaczone do przechowywania tablicy znakowej, zapisując w nim kod ASCII każdego znaku. Kompilator automatycznie kończy tę tablicę dodatkową komórką pamięci, zawierającą wartość 0, oznaczającą koniec tablicy znakowej.

Wewnątrz tablicy znakowej można umieszczać znaki specjalne, używając do tego celu *sekwencji znaków specjalnych* (**ang. escape sequences**). Składają się one z lewego

ukośnika `\t` po którym następuje specjalny kod. Na przykład `\n` oznacza znak nowego wiersza. Instrukcja używanego przez ciebie kompilatora, lub inny dostępny poradnik na temat C, zawiera pełny zbiór sekwencji znaków specjalnych — należą do nich m.in.: `\t` (tabulacja), `\\"` (lewy ukośnik) oraz `\b` (znak cofania).

Zwróć uwagę na to, że instrukcja może przebiegać przez wiele wierszy oraz że kończy się znakiem średnika.

W powyższej instrukcji `cout` argumenty będące tablicami znakowymi są wymieszane ze stałymi liczbowymi. Ponieważ operator `<<`, używany z `cout`, jest przeciążony dla rozmaitych znaczeń, można wysyłać do `cout` argumenty różnych typów, a on „domyśla się, co zrobić z komunikatem”.

Czytając książkę zauważysz, że pierwszy wiersz każdego pliku jest komentarzem, rozpoczynającym się znakami początku komentarza (zazwyczaj `//`), po których następuje dwukropki. Natomiast ostatni wiersz kończy się komentarzem, po którym są umieszczone znaki `,/:~`. Jest to technika, której używam w celu łatwego wyciągania informacji z plików źródłowych (program, który to wykonuje, można znaleźć w drugim tomie książki, dostępnym w witrynie <http://helion.pl/online/thinking/index.html>). Pierwszy wiersz zawiera ponadto nazwę i położenie pliku, dzięki czemu można odwoływać się do niego w tekście oraz w innych plikach, a także łatwo odnaleźć go w kolejnych źródłowych programów zawartych w książce (można go pobrać z witryny <http://helion.pl/online/thinking/index.html>).

Uruchamianie kompilatora

Po pobraniu i rozpakowaniu plików źródłowych odszukaj program w podkatalogu CO2. Wywołaj kompilator, podając jako argument plik `Hello.cpp`. W przypadku prostych, jednoplikowych programów — takich jak przedstawiony — większość kompilatorów poprowadzi cię przez cały proces. Na przykład używając kompilatora GNU C++ (dostępny bezpłatnie w Internecie), należy napisać:

```
g++ Hello.cpp
```

Inne kompilatory będą miały podobną składnię — szczegółowych informacji poszukaj w dokumentacji używanego przez siebie kompilatora.

Więcej o strumieniach wejścia-wyjścia

Powyżej zostały przedstawione tylko najbardziej elementarne właściwości klasy strumieni wejścia-wyjścia. Wykorzystanie do formatowania wyjścia umożliwia takie operacje, jak formatowanie liczb dziesiętnych, ósemkowych i szesnastkowych. Poniżej znajduje się jeszcze jeden przykład użycia strumieni wejścia-wyjścia:

```
//: CO2:Stream2.cpp
// Dodatkowe własności strumieni wejścia-wyjścia
#include <iostream>
using namespace std;
```

```
int main() {
    // Określanie formatu za pomocą manipulatorów
    cout << "liczba dziesiętna "
        << dec << 15 << endl,
    cout << "osiemkowa: " << oct << 15 << endl,
    cout << "szesnastkowa: " << hex << 15 << endl,
    cout << "liczba zmiennopozycyjna: "
        << 3.14159 << endl,
    cout << "znak niedrukowalny (sterujący): "
        << char(27) << endl,
} //~
```

W przykładzie zaprezentowano klasę strumieni wejścia-wyjścia, drukującą liczby w systemie dziesiętnym, ósemkowym i szesnastkowym z wykorzystaniem *manipulatorów* strumieni wejścia-wyjścia (które same nie drukują niczego, ale zmieniają stan strumienia wyjściowego). Formatowanie liczb zmiennopozycyjnych jest określone automatycznie przez kompilator. Ponadto do obiektu będącego strumieniem można wysłać dowolny znak, wykorzystując *rzutowanie* (ang. *cast*) na typ **char** (**char** jest typem danych, umożliwiającym przechowywanie pojedynczego znaku). *Rzutowanie* to przypomina wywołanie funkcji **char()**, z argumentem będącym kodem ASCII znaku. W powyższym programie instrukcja `char(27)` wysyła do obiektu **cout** „znak sterujący”.

Łączenie tablic znakowych

Waczną właściwością preprocessora języka C jest *łączenie tablic znakowych*. Cechą ta jest wykorzystywana w niektórych przykładach zawartych w książce. Jeżeli dwie tablice znakowe przylegają do siebie i nie ma pomiędzy nimi żadnych znaków przestankowych, kompilator klei je ze sobą, tworząc pojedynczą tablicę znakową. Jest to szczególnie przydatne, gdy wydruki, zawierające kod programu, mają ograniczoną szerokość:

```
//: C02 Concat.cpp
// Łączenie tablic znakowych
#include <iostream>
using namespace std;

int main() {
    cout << "Ten tekst jest o wiele za długi "
        "aby umieścić go w jednym wierszu ale można "
        "go podzielić bez żadnych ubocznych skutków"
        "\ndopóki pomiędzy sąsiednimi tablicami "
        "znakowymi nie będzie żadnych znaków "
        "przestankowych \n".
} // -
```

Na pierwszy rzut oka powyższy kod wydaje się błędny, ponieważ nie każdy jego wiersz kończy się znajomym znakiem średnika. Pamiętaj jednak, że C i C++ są językami o swobodnej strukturze i chociaż na ogół na końcu każdego wiersza programu widnieje średnik, to w rzeczywistości wymagane jest, by średnikami kończyła się każda instrukcja. Ponadto jest jak najbardziej dopuszczalne, by obejmowała ona kilka wierszy.

Odczytywanie wejścia

Klasy strumieni wejścia-wyjścia zapewniają również możliwość odczytywania wejścia. Obiektem używanym do obsługi standardowego wejścia jest **cin** (skrót od ang. *console input* – wejście konsoli). Zwykle **cin** spodziewa się danych wejściowych pochodzących z konsoli, ale można przekierować do niego również inne źródła danych. Przykład przekierowania pokazano w dalszej części rozdziału.

Używanym z **cin** operatorem strumienia wejścia-wyjścia jest **>>**. Operator oczekuje na rodzaj danych wejściowych, odpowiadających jego argumentowi. Na przykład jeżeli zostanie mu podany argument całkowitoliczbowy, to będzie oczekwał na wprowadzenie z konsoli liczby całkowitej. Ilustruje to poniższy przykład:

```
//: C02:Numconv.cpp
// Zamiana liczby dziesiętnej na ósemkową i szesnastkową
#include<iostream>
using namespace std;

int main() {
    int number;
    cout < "Wprowadź liczbę dziesiętną: ";
    cin > number;
    cout < "wartość w zapisie ósemkowym = " 
        << oct << number << endl;
    cout < "wartosc w zapisie szesnastkowym = 0x"
        << hex << number << endl;
} //:-
```

Program ten zamienia liczby wprowadzone przez użytkownika na ich reprezentację ósemkową oraz szesnastkową.

Wywoływanie innych programów

Podczas gdy programy czytające standardowe wejście i zapisujące wyniki na standardowe wyjście są zazwyczaj używane w skryptach powłoki Uniksa lub w plikach wstawowych DOS-u, z programu w C lub C++ można wywołać dowolny inny program, używając funkcji **system()** będącej standardową funkcją C, zadeklarowaną w pliku nagłówkowym **<cstdlib>**:

```
//: C02:CallHello.cpp
// Wywołanie innego programu
#include <cstdlib> // Deklaracja funkcji "system()"
using namespace std;

int main() {
    system("Hello");
} //:-
```

Aby użyć funkcji **system()**, należy jej przekazać tablicę znakową o treści, która byłaby zwykle wpisana po znaku zachęty systemu operacyjnego. Może ona również zawierać argumenty wiersza poleceń; można ją także utworzyć w trakcie pracy programu (zamiast używać statycznej tablicy znaków, jak w powyższym przykładzie). Polecenie jest wykonywane, a sterowanie powraca do programu.

Program pokazuje, jak łatwo jest używać w języku C++ bibliotek, napisanych w czystym C — wystarczy jedynie dołączyć plik nagłówkowy i wywołać odpowiednią funkcję. Zgodność „w górę” pomiędzy C i C++ jest bardzo korzystna w sytuacji, gdy rozpoczynasz naukę języka C++, znając już C.

Wprowadzenie do łańcuchów

Mimo że tablice znakowe są całkiem użyteczne, to mają one pewne ograniczenia. Są one po prostu zbiorem znaków w pamięci, ale gdy zamierzasz wykonać jakiekolwiek działania, musisz uwzględnić wszystkie szczegóły. Na przykład wielkość tablicy znakowej jest ustalana w trakcie komplikacji. Jeżeli masz już tablicę znaków i chcesz dodać do niej kilka kolejnych znaków, to będziesz musiał się całkiem sporo nauczyć (włączając w to dynamiczne zarządzanie pamięcią, kopiowanie tablic znakowych oraz ich łączenie), zanim osiągniesz cel. Takimi właśnie kwestiami powinny zajmować się obiekty.

Klasa **string** (ang. *string* – ciąg znaków, łańcuch), należąca do standardu C++, została zaprojektowana dla wszystkich niskopoziomowych operacji na tablicach znakowych (a także po to, je ukrywać), którymi zajmowali się wcześniej programiści języka C. Od czasu powstania języka C z powodu tych operacji marnowano czas i popełniano błędy. Mimo że klasie **string** poświęcono cały rozdział w drugim tomie książki, jest ona na tyle ważna i tak bardzo usprawnia pracę, że zostanie wprowadzona w tym miejscu i będzie używana w wielu początkowych rozdziałach książki.

Aby używać łańcuchów, należy dołączyć plik nagłówkowy C++ **<string>**. Klasa **string** znajduje się w przestrzeni nazw **std**, konieczne jest więc zastosowanie dyrektywy **using**. Dzięki przeciążeniu operatorów składnia związana z używaniem łańcuchów jest dość intuicyjna:

```
//: C02>HelloStrings.cpp
// Podstawy standardowej klasy C++ string
#include <string>
#include <iostream>
using namespace std;

int main(){
    string s1, s2; // Puste łańcuchy
    string s3 = "Witaj, swiecie. "; // Inicjalizacja
    string s4("Mam dzisiaj"); // Również inicjalizacja
    s2 = "urodziny"; // Przypisanie łańcuchowi wartości
    s1 = s3 + " " + s4; // Łączenie łańcuchów
    s1 += " 8 "; // Dołączanie do łańcucha
    cout << s1 + s2 + "!" << endl;
} //:~
```

Pierwsze dwa łańcuchy **s1** i **s2** są początkowo puste, podczas gdy na przykładzie łańcuchów **s3** i **s4** pokazano dwa równoważne sposoby inicjalizacji obiektów klasy **string** za pomocą tablic znakowych (również łatwo można dokonać inicjalizacji obiektów klasy **string** za pomocą innych obiektów tej klasy).

Możesz przypisać wartość dowolnemu obiektowi klasy **string**, używając operatora „**=**”. Powoduje to automatyczną zmianę poprzedniej zawartości łańcucha na to, co znajduje się po prawej stronie. Do połączenia ze sobą łańcuchów używa się operatora „**+**”, który pozwala również na scalenie tablic znakowych z łańcuchami. Jeżeli chcesz dołączyć łańcuch lub tablicę znakową do innego łańcucha, to możesz użyć operatora „**+ =**”. Zwróć wreszcie uwagę na to, że strumienie wejścia-wyjścia zawsze „wiedzą”, co zrobić z łańcuchami, możesz więc po prostu wysłać łańcuch (lub wyrażenie, którego wynikiem jest łańcuch, jak np. `s1 + s2 + "!"`) bezpośrednio do **cout** — po to, by go wydrukować.

Odczytywanie i zapisywanie plików

W języku C proces otwierania plików i ich przetwarzania wymagał gruntownej wiedzy dotyczącej podstawy języka, niezbędnej do wykonywania złożonych operacji. Jednakże biblioteka strumieni wejścia-wyjścia C++ udostępnia prosty sposób operowania na plikach, dzięki czemu zagadnienie to można wprowadzić znacznie wcześniej niż w przypadku języka C.

W celu umożliwienia otwierania plików do odczytu i zapisu trzeba dołączyć plik **<fstream>**. Mimo że w takim przypadku automatycznie dołączony zostanie również plik **<iostream>**, to jeżeli zamierzamy używać obiektów **cin**, **cout** itd., na ogół roztroplne jest dołączyć jawnie plik **<iostream>**.

Aby otworzyć plik do odczytu, należy utworzyć obiekt **ifstream**, który zachowuje się podobnie do **cin**. W celu otwarcia pliku do zapisu trzeba natomiast utworzyć obiekt **ofstream**, który funkcjonuje podobnie do **cout**. Po otwarciu pliku można czytać z niego lub do niego zapisywać, tak jak w przypadku innych obiektów strumienia wejścia-wyjścia. To takie proste (i na tym, oczywiście, polega przełom).

Jedną z najbardziej przydatnych funkcji spośród zawartych w bibliotece strumieni wejścia-wyjścia jest **getline()**, pozwalająca na wczytanie pojedynczego wiersza (zakończonego znakiem nowego wiersza) do obiektu **string**⁴. Pierwszym argumentem jest obiekt **ifstream**, z którego odbywa się czytanie, natomiast drugim argumentem — obiekt **string**. Po zakończeniu wywołania funkcji obiekt **string** będzie zawierał wiersz.

Poniżej przedstawiono przykład prostego programu, kopiącego zawartość jednego pliku do drugiego:

```
//: C02:Scopy.cpp
// Kopiowanie jednego pliku do drugiego, po wierszu
#include <string>
#include <fstream>
using namespace std;
```

⁴ W rzeczywistości istnieje wiele wariantów funkcji **getline()**, co zostanie gruntownie omówione w rozdziale poświęconym strumieniom wejścia-wyjścia, znajdującym się w drugim tomie książki.

```

int main() {
    ifstream in("Scopy.cpp"); // Otwarcie do odczytu
    ofstream out("Scopy2.cpp"); // Otwarcie do zapisu
    string s;
    while(getline(in, s)) // Usuwa znak nowego wiersza
        out << s << "\n"; // ... musi dodać go z powrotem
} //:-/

```

Aby otworzyć pliki, wystarczy po prostu przekazać obiektom **ifstream** i **ofstream** nazwy plików, które mają być otwarte lub utworzone, jak w powyższym przykładzie.

Wprowadzono tu również nowe pojęcie, jakim jest pętla **while**. Chociaż zagadnienie to zostanie opisane szczegółowo w następnym rozdziale, warto wiedzieć, że jej podstawa działania polega na tym, że wyrażenie znajdujące się w nawiasie po instrukcji **while** steruje wykonywaniem następnej instrukcji (która może być również wiele instrukcji, zawartych w nawiasie klamrowym). Dopóki wyrażenie w nawiasie (w tym przypadku **getline(in, s)**) zwraca wynik o wartości „prawda” (ang. *true*), wykonywana jest instrukcja kontrolowana przez **while**. Okazuje się, że funkcja **getline()** zwraca wartość, która może być interpretowana jako „prawda”, wówczas gdy pomyślnie odczyta kolejny wiersz pliku, a „fałsz” (ang. *false*) po osiągnięciu końca pliku. Tak więc powyższa pętla **while** odczytuje kolejno wszystkie wiersze pliku wejściowego, przesyłając każdy z nich do pliku wyjściowego.

Funkcja **getline()** wczytuje znaki znajdujące się w każdym wierszu, aż do natkania znaku nowego wiersza (znak końcowy można zmienić, ale kwestia ta zostanie opisana w rozdziale poświęconym strumieniom wejścia-wyjścia, znajdującym się w drugim tomie książki). W każdym razie usuwa ona znak nowego wiersza i nie zapisuje go w docelowym obiekcie typu **string**. Tak więc jeżeli chcemy, aby skopiowany plik miał postać pliku źródłowego, musimy ponownie dopisać znaki nowego wiersza, jak przedstawiono w programie.

Innym interesującym przykładem jest kopiowanie całego pliku do pojedynczego obiektu typu **string**:

```

//: C02:FillString.cpp
// Wczytanie całego pliku do pojedynczego łańcucha
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ifstream in("FillString.cpp");
    string s, line;
    while(getline(in, line))
        s += line + "\n";
    cout << s;
} //:-/

```

Z uwagi na dynamiczną naturę typu **string** nie trzeba zaprzatać sobie uwagi tym, ile pamięci należy przydzielić łańcuchowi — wystarczy dodawać do niego dane, a łańcuch będzie powiększał się, by pomieścić wszystko to, co w nim umieszczono.

Jedną z korzyści, wynikających z umieszczenia w łańcuchu całego pliku, jest ta, że klasa **string** posiada wiele funkcji, służących do przeszukiwania i przetwarzania łańcuchów, co umożliwia modyfikację całego pliku jako pojedynczego łańcucha. Jednakże wiąże się to z pewnymi ograniczeniami. Na przykład często wygodnie jest traktować plik jako zbiór wierszy, a nie jak jeden wielki blok tekstu. Łatwiej dadasz numerację wierszy, jeżeli każdy jego wiersz będzie znajdował się w oddzielnym obiekcie typu **string**. Aby to uczynić, niezbędne jest inne podejście.

Wprowadzenie do wektorów

Za pomocą łańcuchów możemy wypełnić obiekt typu **string**, nie wiedząc, ile wymaga to pamięci. Problem z wczytywaniem poszczególnych wierszy pliku do oddzielnych obiektów typu **string** polega na tym, że na początku nie wiadomo, ile łańcuchów będzie do tego potrzebnych — okaże się to dopiero po przeczytaniu całego pliku. Aby rozwiązać ten problem, niezbędny jest jakiś kontener, który będzie się automatycznie powiększał, przechowując dowolną liczbę obiektów typu **string**.

Dlaczego właściwie należy się ograniczać do przechowywania łańcuchów? Okazuje się, że nieradko zdarzają się problemy polegające na tym, że w chwili pisania programu nie jest znana liczba jakichś obiektów. Ponadto nazwa „kontener” sugeruje, że byłby on bardziej użyteczny, gdyby mógł przechowywać w ogóle *dowolny rodzaj obiektów!* Na szczęście standardowa biblioteka C++ posiada gotowe rozwiązanie, którym są standardowe klasy kontenerowe. To jedna z lokomotyw standardu C++.

Kontenery i algorytmy, znajdujące się w standardowej bibliotece C++, są często mylone z STL. Standardowa biblioteka szablonów (ang. *Standard Template Library – STL*) jest nazwą użytą przez Alexa Stepanova (pracującego wówczas w firmie Hewlett-Packard) podczas prezentacji Komitetowi Standardyzacyjnemu C++ opracowanej przez siebie biblioteki. Odbyła się ona w ramach spotkania w San Diego (Kalifornia) wiosną 1994 roku. Nazwa ta przyjęła się i upowszechniła od czasu, gdy Hewlett-Packard zdecydował się udostępnić bibliotekę publicznie. W tym czasie komitet dołączył ją do standardowej biblioteki C++, dokonując w niej wielu zmian. Rozwój standardowej biblioteki szablonów kontynuowano w firmie Silicon Graphics (SGI – patrz <http://www.sgi.com/Technology/STL>). Biblioteka SGI STL różni się od standardowej biblioteki C++ wieloma szczegółami. A zatem, wbrew powszechnemu mniemaniu, standardowa biblioteka C++ nie „zawiera” STL. Może to być nieco mylące, ponieważ kontenery i algorytmy zawarte w standardowej bibliotece C++ mają te same korzenie (i zazwyczaj takie same nazwy) co SGI STL. W książce posłużę się terminami: „standardowa biblioteka C++”, „kontenery standardowej biblioteki” itp., unikając terminu „STL”.

Mimo że implementacja kontenerów oraz algorytmów w standardowej bibliotece C++ stosuje pewne zaawansowane pojęcia i jej pełny opis zajmuje dwa duże rozdziałы drugiego tomu książki, biblioteka ta może być skutecznie wykorzystana nawet przez osobę, która nie posiada gruntownej wiedzy na jej temat. Jest tak przydatna, że najbardziej podstawowy kontener — **vector** — został wprowadzony już

w bieżącym rozdziale i jest używany w dalszej części książki. Jak się przekonasz, możesz dokonać bardzo wiele, **używając** po prostu podstawowych właściwości klasy **vector** i nie zważając na jej wewnętrzną implementację (zapewne pamiętasz, że jest to ważna cecha programowania obiektowego). Kiedy dowiesz się nieco więcej na temat tego i innych kontenerów dzięki lekturze rozdziałów poświęconych bibliotece standardowej w drugim tomie książki, zrozumiesz zapewne, że w programach znajdujących się w początkowych rozdziałach książki nie użyto wektorów dokładnie w taki sposób, jak zrobiłby to doświadczony programista C++. Przekonasz się, że w większości przypadków prezentowany tutaj sposób ich użycia jest w zupełności wystarczający.

Klasa **vector** jest *szablonem*, co oznacza, że może ona być w efektywny sposób zastosowana do różnych typów. Możemy zatem utworzyć wektor kształtów, wektor kotów, wektor łańcuchów itp. Używając szablonu, można utworzyć „*klasę czegokolwiek*”. Aby zgłosić kompilatorowi, z jaką klasą będzie on miał do czynienia (w naszym przypadku — co będzie przechowywał wektor), należy umieścić nazwę żądanego typu w „nawiasie kątowym”, czyli w nawiasie złożonym ze znaków „<” i „>”. A zatem wektor łańcuchów (obiektów klasy **string**) powinien być zapisany jako **vector<string>**. W rezultacie otrzymasz specjalizowany wektor, przechowujący jedynie obiekty klasy **string**, i próba umieszczenia w nim innego elementu spowoduje zgłoszenie przez kompilator komunikatu o błędzie.

Ponieważ klasa **vector** reprezentuje pojęcie „kontenera”, musi istnieć jakiś sposób umieszczania w nim elementów i ich wydobywania. Aby dodać na końcu wektora zupełnie nowy element, należy użyć funkcji składowej **push_back()** (ponieważ jest to funkcja składowa, trzeba użyć „.”, aby wywołać ją dla konkretnego obiektu). Nazwa tej funkcji wydaje się nieco „wymyślona” — **push_back()** (ang. *push back* — doóż z tyłu) w przeciwieństwie do prostszej, jak np. „put” (ang. *put* — umieść) — czego powodem jest fakt, że istnieją również inne kontenery i funkcje składowe umożliwiające umieszczanie w nich nowych elementów. Na przykład funkcja składowa **insert()** (ang. *insert* — wstaw) pozwala na wstawienie czegoś do środka kontenera. Klasa **vector** również ją zawiera, ale zastosowanie tej funkcji jest nieco bardziej skomplikowane i dlatego wyjaśnimy je dopiero w drugim tomie książki. Istnieje również funkcja **push_front()** (ang. *push front* — doóż z przodu), nie należąca do klasy **vector**, która wstawia elementy na początek. Istnieje jeszcze o wiele więcej funkcji składowych klasy **vector** i znacznie liczniejsze kontenery znajdują się w standardowej bibliotece C++ — to niewiarygodne, ile można dokonać, wiedząc o kilku prostych właściwościach.

Tak więc można dodawać do wektora nowe elementy za pomocą funkcji **push_back()**, ale jakie później z niego wydobyć? Rozwiążanie jest przemyślniejsze i bardziej eleganckie — dzięki zastosowaniu przeciążenia operatora wektor przypomina tablice. Tablice (opisane dokładniej w następnym rozdziale) są typem danych, dostępnym praktycznie w każdym języku programowania, więc zapewne saci one znane. Tablice są *agregatami*, co oznacza, że składają się one z pewnej liczby połączonych ze sobą elementów. Cechą wyróżniającą tablice jest to, że ich elementy są tej samej wielkości i uporządkowano je w taki sposób, że następują kolejno po sobie. Co najważniejsze — elementy tablicy mogą zostać wybrane przez „indeksowanie”, a zatem można wskazać „element o numerze n” i zostanie on udostępniony, zazwyczaj szybko.

Mimo że istnieją pod tym względem wyjątki wśród języków programowania, indeksowanie uzyskuje się na ogół za pomocą nawiasów kwadratowych, tak więc jeżeli masz np. tablicę a i chcesz otrzymać jej piąty element, to zapisujesz a[4] (zwróć uwagę na to, że indeksowanie rozpoczyna się zawsze od zera).

Tak zwięzły i efektywny zapis indeksowania uzyskano w klasie **vector** dzięki przejęciu operatorów — w sposób podobny do tego, w jaki operatory „<<” i „>>” zostały wprowadzone do strumieni wejścia-wyjścia. Kolejny raz okazuje się, że nie trzeba wiedzieć, w jaki sposób zaimplementowano przeciążenia (temat ten przedstawiono w następnych rozdziałach), ale warto mieć świadomość, że kryją się za tym jakieś czary, sprawiające, że nawiasy kwadratowe działają razem z klasą **vector**.

Mając to na uwadze, możemy już zaprezentować program wykorzystujący wektory. Aby używać obiektów klasy **vector**, należy dołączyć plik nagłówkowy **<vector>**:

```
///: C02:Fillvector.cpp
// Kopiowanie całego pliku do wektora łańcuchów
#include<string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
    vector<string> v;
    ifstream in("Fillvector.cpp");
    string line;
    while(getline(in, line))
        v.push_back(line); // Dodanie wiersza na końcu
    // Dodanie numerów wierszy:
    for(int i = 0; i < v.size(); i++)
        cout << i << ": " << v[i] << endl;
} //
```

Większa część powyższego programu jest podobna do jego poprzedniej wersji — otwierany jest plik, a jego wiersze są kolejno wczytywane do łańcuchów (obiektów typu **string**). Te łańcuchy sajednak dopisywane na końcu wektora v. Po zakończeniu pętli **while** cały plik znajduje się w pamięci — wewnątrz obiektu v.

Następna instrukcja programu nosi nazwę pętli for. Jest ona podobna do pętli **while**, z wyjątkiem tego, że zapewnia ona pewną dodatkową kontrolę. Po instrukcji **for** następuje nawias, zawierający „wyrażenie sterujące”, podobnie jak w pętli **while**. Jednakże wyrażenie to składa się z trzech części: pierwszej — inicjalizującej, drugiej — sprawdzającej, czy należy już wyjść z pętli, i trzeciej — zmieniającej coś, na ogół po to, by przejść przez sekwencję elementów. W programie użyto pętli for w najczęściej spotykany sposób. Część inicjalizująca, int i = 0, tworzy całkowitą zmienną i, używaną w charakterze licznika pętli, i nadaje jej zerową wartość początkową. Część sprawdzająca oznacza, że aby pozostać w pętli, i powinno być mniejsze niż liczba elementów wektora v. Liczbę tę uzyskuje się za pomocą funkcji składowej **size()**, którą niejako „przemyciłem” do programu, ale ma ona dość oczywiste znaczenie (ang. **size** — wielkość, rozmiar). W ostatniej części wykorzystano skrót stosowany w językach C i C++ — operator „automatycznej inkrementacji”, zwiększający wartość

zmiennej i o jeden. W rezultacie zapis `i++` oznacza „pobierz wartość `i`, zwiększ ją o jeden i umieść rezultat z powrotem w zmiennej `i`”. Tak więc ostatecznym efektem działania pętli `for` jest utworzenie zmiennej `i`, a następnie zmiana jej wartości — od zera do liczby elementów wektora pomniejszonej o jeden. Dla każdej wartości i wykonywaną jest instrukcja `cout`, zapisująca wiersz złożony z wartości zmiennej `i` (przekształconej w cudowny sposób w tablicę znakową za pomocą instrukcji `cout`), dwukropka, spacji, wiersza pochodzącego z pliku i znaku nowego wiersza, wyprowadzanego przez `endl`. Po komplikacji i uruchomieniu programu przekonasz się, że efektem jego działania jest dopisanie w pliku numerów poszczególnych wierszy.

Z uwagi na sposób, w jaki operator `>>` działa ze strumieniami wejścia-wyjścia, można łatwo zmodyfikować powyższy program, tak aby zamiast na wiersze dzielił on plik wejściowy na słowa, oddzielone od siebie odstępami:

```
//: C02:GetWords.cpp
// Podział pliku na słowa oddzielone odstępami
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
    vector<string> words;
    ifstream in("GetWords.cpp");
    string word;
    while(in >> word)
        words.push_back(word);
    for(int i = 0; i < words.size(); i++)
        cout << words[i] << endl;
} ///-
```

Wyrażenie:

```
while(in >> word)
```

pobiera z wejścia za każdym razem jedno „słowo”, a gdy ma ono wartość „fałsz”, oznacza to, że osiągnięty został koniec pliku. Oddzielanie od siebie słów za pomocą odstępów jest dość toporne, ale pomocne w prostym przykładzie. W dalszej części książki zapoznasz się z bardziej wyrafinowanymi przykładami, pozwalającymi na podział danych wejściowych w dowolny sposób.

Aby pokazać, jak łatwo jest używać wektorów dowolnego typu, w poniższym przykładzie został utworzony wektor liczb całkowitych (`vector<int>`):

```
//: C02:Intvector.cpp
// Tworzenie wektora zawierającego liczby całkowite
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;
    for(int i = 0; i < 10; i++)
        v.push_back(i);
    for(int i = 0; i < v.size(); i++)
```

```
cout << v[1] << ". ";
cout << endl;
for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Przypisanie
for(int i = 0; i < v.size(); i++)
    cout << v[i] << ". ";
cout << endl;
} //
```

W celu utworzenia wektora przechowującego inny typ danych należy podać ten typ jako argument szablonu (w nawiasie kątowym). Szablony oraz dobrze zaprojektowane biblioteki szablonów mają w zamyśle ich twórców być łatwe w użytku.

Powyższy przykład ilustruje również inną ważną cechę klasy **vector**. W wyrażeniu:

```
v[i] = v[i] * 10;
```

użycie wektora nie jest ograniczone tylko do umieszczania w nim elementów i pobierania ich z powrotem. Można także dokonać *przypisania* (a więc i zmiany) do dowolnego elementu wektora, używając do tego celu również operatora indeksowania w postaci nawiasu kwadratowego. Oznacza to, że wektor jest elastycznym „notesem” ogólnego przeznaczenia, umożliwiającym pracę z grupami obiektów. Oczywiście skorzystamy z tego w następnych rozdziałach.

Podsumowanie

Celem rozdziału było pokazanie, jak łatwo może być programowanie obiektowe, *pod warunkiem*, że ktoś wykona za ciebie pracę, polegającą na zdefiniowaniu obiektów. W takim przypadku musisz dołączyć plik nagłówkowy, utworzyć obiekty, a następnie wysyłać do nich komunikaty. Jeżeli używane przez ciebie typy są efektywne i odpowiednio zaprojektowane, możesz oszczędzić sobie wiele pracy, a powstały program będzie również wydajny.

Przy okazji prezentacji łatwości programowania obiektowego z użyciem bibliotek klas przedstawiono również najbardziej podstawowe i użyteczne typy w standardowej bibliotece C++ — rodzinę strumieni wejścia-wyjścia (szczególnie tych czytających z konsoli oraz plików i zapisujących tam informacje), klasę **string** i szablon **vector**. Przekonałeś się już, jak łatwo można ich używać i zapewne możesz teraz wyobrazić sobie wiele projektów, które mógłbyś z ich pomocą zrealizować, a w rzeczywistości ich możliwości są jeszcze większe⁵. Mimo że w początkowych rozdziałach książki będziemy wykorzystywać jedynie ograniczoną funkcjonalność tych narzędzi, trzeba pamiętać, że i tak stanowią one duży krok naprzód w stosunku do prostoty nauki niskopoziomowego języka, jakim jest C (nauka niskopoziomowych cech C ma naprawdę pewien walor edukacyjny, lecz jest również czasochłonna). Ostatecznie znacznie zwiększasz swoją wydajność, jeżeli użyjesz obiektów do obsługi niskopoziomowych

⁵ Jeżeli jesteś szczególnie zainteresowany wszystkim, co można zrobić z tymi oraz innymi składnikami standardowej biblioteki, zajrzyj do drugiego tomu książki, dostępnego w witrynie www.BruceEckel.com.
<http://helion.pl/online/thinking/index.html>, atakże www.dinkynware.com.

zagadnień. Mimo wszystko głównym atutem programowania obiektowego jest ukrywanie szczegółów, umożliwiające pewien rozmach.

Jednakże poruszając się nawet na tak wysokim poziomie, do jakiego aspiruje programowanie obiektowe, nie można pominąć pewnych podstawowych zagadnień języka C. I właśnie one zostaną opisane w następnym rozdziale.

Ćwiczenia

Rozwiązania wybranych ćwiczeń można znaleźć w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, dostępnego za niewielką opłatą w witrynie <http://www.BruceEckel.com>.

1. Zmodyfikuj program **Hello.cpp** w taki sposób, by drukował twoje imię i wiek (lub rozmiar buta albo wiek twojego psa). Skompiluj i uruchom ten program.
2. Na podstawie programów **Stream2.cpp** oraz **Numconv.cpp** napisz program, który pyta o promień koła, a następnie drukuje jego powierzchnię.
Do wyznaczenia kwadratu promienia możesz użyć operatora „*”.
Nie próbuj wyprowadzać wartości jako ósemkowej lub szesnastkowej (działające wyłącznie z typami całkowitymi).
3. Napisz program, który otwiera plik i liczy zawarte w nim, oddzielone **odstępami** słowa.
4. Napisz program, który wyznacza liczbę wystąpień określonego słowa w pliku (użyj operatora „==” klasy **string** do znalezienia tego słowa).
5. Zmień program **Fillvector.cpp** w taki sposób, by drukował wiersze od końca — od ostatniego do pierwszego.
6. Zmień program **Fillvector.cpp** w taki sposób, by łączył wszystkie elementy znajdujące się w wektorze w pojedynczy łańcuch jedyne przed jego wydrukowaniem, ale nie próbuj dodawać numeracji wierszy.
7. Wyświetl **plik po jednym** wierszu, oczekując po każdym z nich naciśnięcia przez użytkownika klawisza „Enter”.
8. Utwórz wektor **vector<float>** i umieść w nim 25 liczb **zmiennopozycyjnych**, **używając** do tego pętli **for**. Wyświetl ten wektor.
9. Utwórz trzy obiekty **vector<float>** i wypełnij dwa pierwsze z nich w taki sposób, jak w poprzednim ćwiczeniu. Napisz pętlę **for**, która doda odpowiadające sobie elementy pierwszych dwóch wektorów, zapisując wynik w odpowiednim elemencie trzeciego wektora. Wyświetl zawartość wszystkich trzech wektorów.

Utwórz wektor **vector<float>** i umieść w nim 25 liczb, jak w poprzednich ćwiczeniach. Następnie podnieś do kwadratu każdą z liczb i umieść wynik z powrotem w tym samym miejscu wektora. Wyświetl zawartość wektora przed wykonaniem **mnożenia** i po nim.

Rozdział 3.

Język C w C++

Z uwagi na to, że język C++ w dużej mierze bazuje na języku C, trzeba najpierw poznać składnię C, aby programować w C++ — podobnie jak należy nabrać pewnej biegłości w algebrze, by zmierzyć się z analizą matematyczną.

Jeżeli nigdy wcześniej nie zetknąłeś się z językiem C, to w tym rozdziale znajdziesz podstawowe informacje dotyczące rodzaju języka C używanego w C++. Jeżeli znasz język C, opisany w pierwszym wydaniu książki Kernighana i Ritchiego (określany często jako K&R C), to w C++ oraz w standardowym C odnajdziesz zarówno nowe właściwości języka, jak i takie, które różnią się od już poznanych. Jeżeli natomiast znasz już standardowe C, warto przejrzeć ten rozdział, poszukując tematów związanych z C++. Zwróć uwagę na to, że niektóre z wprowadzonych w rozdziale podstawowych właściwości języka C++ są pokrewne właściwościom języka C. Część z nich ilustruje również zmianę w ujęciu pewnych kwestii w porównaniu z podejściem stosowanym w języku C. Bardziej zaawansowane właściwości języka C++ zostaną przedstawione w dalszych rozdziałach książki.

Rozdział ten stanowi dość krótki opis konstrukcji języka C i wprowadzenie do pewnych podstawowych konstrukcji C++. Przyjęto założenie, że masz już doświadczenie w programowaniu w jakimś innym języku.

Tworzenie funkcji

W „dawnym” C (przed powstaniem standardu języka) można było wywołać funkcję z dowolną liczbą argumentów dowolnego typu, nie wywołując sprzeciwu kompilatora. Wszystko wydawało się poprawne, dopóki nie został uruchomiony program. Pojawiały się wówczas zagadkowe wyniki (lub gorzej — program przerywał pracę) i nie docierały żadne wskazówki na temat tego, co może być ich powodem. Brak pomocy w przekazywaniu argumentów oraz enigmatyczne błędy są prawdopodobnie jednym z powodów, dla których język C nazywano „assemblerem wysokiego poziomu”. Programiści używający C przed wprowadzeniem standardu języka po prostu się do tego przyzwyczaili.

W standardowym C oraz C++ jest używane *prototypowanie funkcji* (ang. *function prototyping*). Powoduje ono konieczność opisania typów argumentów funkcji, zarówno w jej deklaracji, jak i w definicji. Opis ten stanowi właściwie „prototyp”. W czasie wywołania funkcji kompilator używa prototypu, by upewnić się, że zostały jej przekazane właściwe argumenty, a zwracana wartość jest poprawnie traktowana. Jeżeli programista pomyli się podczas wywołania funkcji, błąd ten zostanie wykryty przez kompilator.

Wiadomości o prototypowaniu funkcji znajdują się już w poprzednim rozdziale (mimo że nie zostało ono w taki sposób nazwane), ponieważ postać deklaracji funkcji w C++ wymaga właściwie odpowiedniego prototypowania. W prototypie funkcji lista argumentów zawiera typy argumentów, które muszą być przekazane funkcji, oraz (opcjonalnie w przypadku deklaracji) ich identyfikatory. Kolejność i typy argumentów muszą odpowiadać sobie w deklaracji, definicji i wywołaniu funkcji. Oto przykład prototypu funkcji zawartego w deklaracji:

```
int translate(float x, float y, float z);
```

Postać deklaracji zmiennych w prototype funkcji nie jest taka sama, jak zwykła definicja zmiennych. Oznacza to, że nie można w tym wypadku użyć zapisu: **float x, y, z.** Należy oddziennie określić typ *każdego* jej argumentu. W deklaracji funkcji dopuszczalna jest również następująca postać zapisu:

```
int translate(float, float, float);
```

Ponieważ kompilator sprawdza w wywołaniu funkcji jedynie typy jej argumentów, identyfikatory służą tylko do zwiększenia przejrzystości kodu w przypadku, gdy ktoś go czyta.

Nazwy argumentów są natomiast wymagane w definicji funkcji, ponieważ odwołujemy się do nich w obrębie funkcji:

```
int translate(float x, float y, float z) {  
    x = y = z;  
    // ...  
}
```

Okazuje się, że ta reguła obowiązuje tylko w stosunku do języka C. W C++ argumenty znajdujące się na liście argumentów definicji funkcji mogą pozostać anonimowe. Ponieważ nie mają one nazw, nie można oczywiście używać ich wewnętrz funkcji. Anonimowe argumenty zostały wprowadzone po to, by umożliwić programistom „rezerwację miejsca na liście argumentów”. Każdy, kto używa tej funkcji, musi wywoływać ją z odpowiednimi argumentami. Jednakże osoba, która utworzyła funkcję, może zastosować „zarezerwowany” argument w przyszłości, nie powodując konieczności modyfikacji kodu wywołującego funkcję. Możliwość zignorowania argumentu istnieje również w przypadku, gdy pozostawi się jego nazwę na liście argumentów. Powoduje to jednak zgłaszanie, podczas każdej komplikacji funkcji, irytującego ostrzeżenia, dotyczącego niewykorzystanego argumentu. Usunięcie nazwy argumentu wyeliminuje tę niedogodność.

W językach C oraz C++ są używane dwa różne sposoby deklaracji listy argumentów. Jeżeli funkcja posiada pustą listę argumentów, można zadeklarować ją w języku C++ jako **func()**. To informacja dla kompilatora, że funkcja ta nie przyjmuje w ogóle argumentów.

Należy zwrócić uwagę na to, że powyższy zapis oznacza pustą listę argumentów jedynie w języku C++. W języku C określa on natomiast „nieokreśloną liczbę argumentów” (co stanowi lukę w tym języku, ponieważ **wyłącza**, w takim przypadku, kontrolę ich typów). Zarówno w C, jak i w C++ **deklaracja func(void)** określa pustą listę argumentów, **Słowo kluczowe void** oznacza w tym przypadku „nic” (jak dowieś się w dalszej części rozdziału, w przypadku wskaźników może ono również oznaczać „brak typu”).

fany przypadek, dotyczący list argumentów, ma miejsce wtedy, gdy nie jest znana **liczba albo typy argumentów**, z którymi zostanie wywołana funkcja — w takiej sytuacji mamy do czynienia ze *zmienią listą argumentów*. Taka „nieokreślona lista argumentów” jest oznaczana wielokropkiem (...) - Definiowanie funkcji o zmiennej liście argumentów jest znacznie bardziej skomplikowane niż definiowanie zwyczajnej funkcji. Zmiennej listy argumentów można również używać w przypadku funkcji o ustalonej liczbie argumentów, jeżeli (z jakiegoś powodu) należy zablokować kontrolę ich typów, wynikającą z prototypowania funkcji. Dlatego właściwie powinno się ograniczyć stosowanie zmiennej listy argumentów do języka C i unikać używania jej w C++ (w którym to języku, jak się później przekonasz, są dostępne znacznie lepsze rozwiązania alternatywne). Opis obsługi zmiennych list argumentów można znaleźć w rozdziale poświęconym bibliotekom w dowolnym poradniku dotyczącym języka C.

Wartości zwracane przez funkcje

W języku C++ prototyp funkcji musi określać typ zwracanej przez nią wartości (w C pominięcie typu zwracanej wartości powoduje domyślne przyjęcie typu **int**). Specyfikacja zwracanego typu poprzedza nazwę funkcji. Aby określić, że funkcja nie zwraca żadnej wartości, należy użyć słowa kluczowego **void**. W przypadku próby zwrócenia przez funkcję wartości spowoduje to komunikat o błędzie. Poniżej znajduje się kilka kompletnych prototypów funkcji:

```
int f1(void); // Zwraca wartość int. nie przyjmuje żadnych argumentów
int f2(); // Podobnie, jak f1() w przypadku C++, ale nie w standardzie C!
float f3(float, int, char, double); // Zwraca wartość float
void f4(void); // Nie przyjmuje argumentów i nic nie zwraca
```

Aby przekazać z funkcji jakąś wartość, należy użyć instrukcji **return**. Powoduje ona **opuszczenie** funkcji i przejście do miejsca znajdującego się bezpośrednio po jej wywołaniu. Jeżeli instrukcja **return posiada jakiś argument**, staje się on wartością zwracaną przez funkcję. Jeżeli funkcja zwraca argument określonego typu, to wszystkie zawarte w niej instrukcje **return** muszą zwracać wartości tego właśnie typu. W **definicji funkcji** może znajdować się więcej niż jedna instrukcja **return**:

```
//: C03:Return.cpp
// Użycie instrukcji "return"
#include <iostream>
using namespace std;

char cfunc(int i) {
    if(i == 0)
        return 'a';
    if(i == 1)
        return 'g';
```

```
if(i == 5)
    return 'z';
return 'c';
}

int main() {
    cout << "wpisz liczbę całkowitą: ";
    int val;
    cin >> val;
    cout << cfunc(val) << endl;
} // :~
```

W funkcji **cfunc()** pierwsza instrukcja **if**, której wyrażenie sterujące osiągnie wartość **true** (prawda), spowoduje opuszczenie funkcji za pomocą instrukcji **return**. Zwróć uwagę na to, że deklaracja tej funkcji nie jest konieczna, ponieważ definicja funkcji występuje w programie przedniej użyciem w funkcji **main()**, dzięki czemu kompilator poznaje już funkcję, czytającą jej definicję.

Używanie bibliotek funkcji języka C

Wszystkie funkcje znajdujące się w twojej lokalnej bibliotece funkcji C są dostępne w czasie programowania w języku C++. Warto zapoznać się **dokładnie z biblioteką**, zanim zdefiniujesz własną funkcję — istnieje duża szansa, że ktoś rozwiązał już twój problem, poświęcając prawdopodobnie znacznie więcej czasu na przemyślenie odpowiedniej funkcji oraz jej uruchomienie.

Słowo przestrogi: wiele kompilatorów zawiera mnóstwo dodatkowych, niezwykle przydatnych funkcji, nie należących jednak do standardowej biblioteki C. Jeżeli masz pewność, że nigdy nie przeniesiesz aplikacji na inną platformę (która może być tego pewien?), nie wahaj się — użyj tych funkcji i ułatw sobie za'danie. Jeżeli jednak zależy ci na tym, by Twoja aplikacja była przenośna, to należy ograniczyć się do funkcji biblioteki standardowej. Jeżeli musisz wykonać działania specyficzne dla danej platformy, spróbuj umieścić je w oddzielnym fragmencie kodu, dzięki czemu, przy przenoszeniu programu na inną platformę, będzie go można łatwo zmienić. W C++ działania zależne od platformy można często zamknąć w obrębie klasy, co jest rozwiązaniem idealnym.

Zasada używania funkcji bibliotecznych jest następująca: najpierw odszukaj funkcję w podręczniku programowania (wiele podręczników zawiera zarówno spis funkcji, z podziałem na kategorie, jak i alfabetyczny). Opis funkcji powinien znajdować się w dziale prezentującym składnię kodu. W górnej części tego działu znajduje się zazwyczaj przynajmniej jeden wiersz, zawierający dyrektywę **#include**, który przedstawia nazwę pliku nagłówkowego z prototypem funkcji. Skopiuj ten wiersz do swojego programu, dzięki czemu funkcja będzie prawidłowo zadeklarowana. Teraz możesz wywołać funkcję — w taki sposób, jak przedstawia to dział podręcznika opisujący składnię. Jeżeli się pomyliłeś, kompilator **wykryje** to, porównując Twoje wywołanie z prototypem funkcji zawartym w pliku nagłówkowym, i powiadomi Cię o błędzie. Program łączący domyślnie przeszuka standardową bibliotekę, dzięki czemu jedynym działaniem, który musisz wykonać, jest dołączenie pliku nagłówkowego i wywołanie funkcji.

Tworzenie **własnych** bibliotek za pomocą programu zarządzającego bibliotekami

Możesz połączyć napisane przez siebie funkcje, tworząc bibliotekę. Większość pakietów programistycznych dostarczanych jest wraz z programem zarządzającym bibliotekami i grupami programów wynikowych. Każdy program zarządzający bibliotekami ma odrębny zbiór poleceń, ale generalna zasada jest następująca: aby utworzyć bibliotekę, utwórz plik nagłówkowy zawierający prototypy wszystkich funkcji zawartych w bibliotece. Plik ten należy umieścić w jakimkolwiek miejscu objętym ścieżką wyszukiwania preprocesora — albo w lokalnym katalogu (dzięki czemu będzie można go znaleźć za pomocą dyrektywy `#include "plik-nagłówkowy"`), albo w katalogu zawierającym dołączane pliki nagłówkowe (można go odszukać przy użyciu dyrektywy `#include <plik-nagłówkowy>`). Następnie przekaż wszystkie programy wynikowe programowi zarządzającemu biblioteką — wraz z nazwą wynikowej biblioteki (większość programów bibliotecznych wymaga typowego rozszerzenia nazwy **pliku**, takiego jak **.lib** lub **.a**). Umieść gotową bibliotekę w miejscu, w którym znajdują się inne biblioteki, dzięki czemu będzie ją mógł znaleźć program łączący. W czasie używania swojej biblioteki musisz dodać coś do wiersza poleceń, dzięki czemu program łączący będzie mógł odnaleźć w tej bibliotece wywoływanie przez ciebie funkcje. Szczegółowych informacji musisz poszukać w dostępnej dokumentacji, ponieważ różnią się one w poszczególnych systemach.

Sterowanie wykonywaniem programu

Rozdział obejmuje zawarte w C++ instrukcje sterujące wykonywaniem programu. Musisz poznać, zanim zaczniesz czytać i pisać programy w C lub w C++.

Język C++ wykorzystuje wszystkie instrukcje sterujące języka C. Obejmują one instrukcje **if-else**, **while**, **do-while**, **for** oraz instrukcję wyboru, nazywaną **switch**. C++ pozwala również na użycie niesławnej instrukcji **goto**, celowo w niniejszej książce pomijanej.

Prawda i fałsz

W celu określenia ścieżki wykonania programu wszystkie instrukcje warunkowe używają pojęcia prawdziwości lub fałszywości wyrażenia warunkowego. Przykładem wyrażenia warunkowego jest: `A == B`. Wykorzystuje ono operator warunkowy `==` do sprawdzenia, czy zmienna A jest równoważna zmiennej B. Wyrażenie ma wartość logiczną **true** (prawda) lub **false** (fałsz). Są one słowami kluczowymi tylko w C++ — w języku C wyrażenie jest „prawdziwe”, jeżeli ma niezerową wartość. Innymi operatorami warunkowymi są: `>`, `<`, `>=` itd. Wyrażenia warunkowe zostały opisane dokładniej w dalszej części rozdziału.

if-else

Instrukcja **if-else** może występować w dwóch postaciach — z **else** oraz bez niego. Wygląda ona następująco:

```
if(wyrażenie)
    instrukcja
```

lub:

```
if(wyrażenie)
    instrukcja
else
    instrukcja
```

„Wyrażenie” daje wartość **true** (prawda) lub **false** (fałsz). „Instrukcja” oznacza natomiast albo instrukcję **prostą**, zakończoną średnikiem, albo **złożoną**, **będącą** grupą instrukcji prostych, zamkniętych w nawiasie klamrowym. Słowo „instrukcja” oznacza zawsze instrukcję prostą albo złożoną. Zwróć uwagę na to, że instrukcją tą może być również inna instrukcja if, dzięki czemu mogłyby one połączone w łańcuch.

```
//: C03:Ifthen.cpp
// Demonstracja instrukcji warunkowych if i if-else
#include <iostream>
using namespace std;

int main() {
    int i;
    cout << "wpisz liczbę i nacisnij 'Enter'" << endl;
    cin >> i;
    if(i > 5)
        cout << "Jest większa niż 5" << endl;
    else
        if(i < 5)
            cout << "Jest mniejsza niż 5" << endl;
        else
            cout << "Jest równa 5" << endl;

    cout << "wpisz liczbę i nacisnij 'Enter'" << endl;
    cin >> i;
    if(i < 10)
        if(i > 5) // "if" jest po prostu kolejną instrukcją
            cout << "5 < i < 10" << endl;
        else
            cout << "i <= 5" << endl;
    else // Skojarzone z "if(i < 10)"
        cout << "i >= 10" << endl;
} III-
```

Zwyczajowo w instrukcji sterującej wykonaniem programu stosuje się wcięcia, dzięki czemu czytelnik może łatwo określić jego początek i koniec¹.

¹ Zwrót uwagi na to, że wszystkie konwencje wydają się kończyć na uznaniu, że należy stosować jakiś rodzaj wcięć. Walka pomiędzy różnymi sposobami formatowania kodu nie ma końca. Dodatek A zawiera opis stylu kodowania używanego w książce.

while

Instrukcje **while**, **do-while** oraz **for** sterują pętlami. Instrukcja jest powtarzana aż do momentu, gdy wyrażenie kontrolne zwróci wartość **false**. Postać instrukcji **while** jest następująca:

```
while(wyrażenie)
    instrukcja
```

Wartość wyrażenia jest obliczana jednokrotnie na początku pętli, a następnie przed każdym kolejnym powtórzeniem instrukcji.

W poniższym przykładzie program pozostaje wewnątrz pętli **while**, aż do wpisania tajnego numeru lub naciśnięcia Control-C.

```
//: C03:Guess.cpp
// Odgadywanie numeru (demonstracja instrukcji "while")
#include <iostream>
using namespace std;

int main() {
    int secret = 15;
    int guess = 0;
    // "!=" jest warunkiem "nierówne":
    while(guess != secret) { // Instrukcja złożona
        cout << "odgadnij numer: ";
        cin >> guess;
    }
    cout << "Zgadles!" << endl;
} //:~
```

Warunkowe wyrażenie instrukcji **while** nie jest ograniczone do prostego testu, jak w powyższym przykładzie — może być ono dowolnie skomplikowane, dopóki zwraca wartość **true** lub **false**. Możesz nawet zobaczyć kod, w którym pętla nie ma ciała, tylko sam średnik:

```
while(/* Zrob duzo w tym miejscu */)
    ;
```

W takim przypadku programista napisał wyrażenie warunkowe nie tylko w celu jego sprawdzenia, ale również po to, by wykonało ono jakąś pracę.

do-while

Postacią instrukcji **do-while** jest:

```
do
    instrukcja
while(wyrażenie);
```

Instrukcja **do-while** tym różni się od **while**, że jest zawsze wykonywana przynajmniej jednokrotnie, nawet jeżeli wartość wyrażenia jest już za pierwszym razem fałszywa.

W przypadku zwykłego **while**, jeżeli warunek za pierwszym razem jest fałszywy, instrukcja nie jest nigdy wykonywana.



Jeżeli pętla **do-while** zostanie użyta w programie **Guess.cpp**, to zmiennej **guess** nie trzeba będzie nadawać fikcyjnej wartości początkowej. Zostanie ona bowiem zainicjowana instrukcją **cin**, zanim jej wartość zostanie sprawdzona:

```
//: C03:Guess2.cpp
// Program odgadujący, wykorzystujący pętlę do-while
#include <iostream>
using namespace std;

int main() {
    int secret = 15;
    int guess; // Nie ma potrzeby inicjalizacji zmiennej
    do {
        cout << "odgadnij numer: ";
        cin >> guess; // Tu ma miejsce inicjalizacja
    } while(guess != secret);
    cout << "Zgadles!" << endl;
} //:~
```

Większość programistów unika instrukcji **do-while**, używając instrukcji **while**.

for

Pętla **for** wykonuje inicjalizację, poprzedzającą pierwszą iterację. Następnie dokonuje sprawdzenia warunku, a na końcu każdej iteracji wykonuje jakiś rodzaj „kroku”. Postać instrukcji **for** przedstawiono poniżej:

```
for(inicjalizacja; warunek; krok)
    instrukcja
```

Zarówno *inicjalizacja*, jak i *warunek* oraz *krok* mogą być puste. Kod inicjalizacji wykonyuje się na samym początku, tylko jeden raz. *Warunek* jest sprawdzany przed każdym przebiegiem pętli (jeżeli jego wartość na początku będzie fałszywa, instrukcja nigdy nie zostanie wykonana). Na końcu każdej pętli zostanie wykonany *krok*.

Pętle **for** są zazwyczaj używane do zadań „zliczania”:

```
//: C03:Charlist.cpp
// Wyświetlanie wszystkich znaków ASCII
// Demonstracja pętli "for"
#include <iostream>
using namespace std;

int main() {
    for(int i = 0; i < 128; i - i + 1)
        if (i != 26) // Oczyszczenie ekranu terminalu ANSI
            cout << "wartosc: " << i
            << " znak: "
            << char(i) // Konwersja typu
            << endl;
} //:~
```

Zmienna i została zdefiniowana w miejscu, w którym jest używana, a nie na początku bloku, oznaczonym klamrowym nawiasem otwierającym „{”. Stanowi to różnicę w stosunku do tradycyjnych języków proceduralnych (w tym C), wymagających deklaracji wszystkich zmiennych na początku bloku. Temat ten zostanie omówiony w dalszej części rozdziału.

Słowa kluczowe break i continue

Wewnątrz dowolnej konstrukcji tworzącej pętlę, takiej jak **while**, **do-while** lub **for**, można sterować jej przebiegiem — za pomocą instrukcji **break** i **continue**. Instrukcja **break** powoduje opuszczenie pętli, bez wykonywania pozostałych, zawartych w niej instrukcji. Natomiast instrukcja **continue** zatrzymuje wykonanie aktualnej iteracji, powodując powrót do początku pętli w celu rozpoczęcia nowej iteracji.

Przykładem użycia instrukcji **break** i **continue** jest program będący bardzo prostym systemem menu:

```
//: C03:Menu.cpp
// Prosty program menu, ilustrujący
// użycie instrukcji "break" i "continue"
#include <iostream>
using namespace std;

int main() {
    char c; // Do przechowywania odpowiedzi
    while(true) {
        cout << "GŁÓWNE MENU:" << endl;
        cout << "1: lewe, p: prawe, w: wyjście -> ";
        cin>>c;
        if(c == 'w')
            break; // Wyjście z "while(1)"
        if(c == 'l') {
            cout << "LEWE MENU:" << endl;
            cout << "wybierz a lub b: ";
            cin>>c;
            if(c == 'a') {
                cout << "wybrałeś 'a'" << endl;
                continue; // Powrót do głównego menu
            }
            if(c == 'b') {
                cout << "wybrałeś 'b'" << endl;
                continue; // Powrót do głównego menu
            }
        }
        else {
            cout << "nie wybrałeś a ani b!" << endl;
            continue; // Powrót do głównego menu
        }
    }
    if(c == 'p') {
        cout << "PRAWE MENU:" << endl;
        cout << "wybierz c lub d: ";
        cin>>c;
    }
}
```

```

if(c == 'c') {
    cout << "wybrales 'c'" << endl;
    continue; // Powrót do głównego menu
}
if(c == 'd') {
    cout << "wybrales 'd'" << endl;
    continue; // Powrót do głównego menu
}
else {
    cout << "nie wybrales c ani d!" 
        << endl;
    continue; // Powrót do głównego menu
}
cout << "musisz wpisać 1. p albo w!" << endl;
cout << "wyjdzie z menu..." << endl;
}   III-
}

```

Jeżeli użytkownik wybierze w **głównym** menu „w”, do wyjścia używane jest słowo kluczowe **break**; w przeciwnym razie program będzie działał bez końca. Po każdym z wyborów w **podmenu** następuje powrót na początek pętli **while** za pomocą słowa kluczowego **continue**.

Instrukcja **while(true)** jest równoważna poleceniu „wykonuj tę pętlę w nieskończoność”. Instrukcja **break** umożliwia przerwanie tej nieskończonej pętli po wpisaniu przez użytkownika „w”.

switch

Instrukcja **switch** wybiera jeden z fragmentów kodu na podstawie wartości wyrażenia całkowitego. Ma ona następującą postać:

```

switch(selektor) {
    case wartosc1 : instrukcja; break;
    case wartosc2 : instrukcja; break;
    case wartosc3 : instrukcja; break;
    case wartosc4 : instrukcja; break;
    case wartosc5 : instrukcja; break;
    ...
    default: instrukcja;
}

```

Selektor jest wyrażeniem dającym wartość całkowitą. Instrukcja **switch** porównuje wartość **selekторa** z każdą wartością **całkowitą**. Jeżeli znajdzie odpowiednią wartość, wykonana zostanie instrukcję (prostą lub złożoną). Jeżeli żadna wartość **całkowita** nie będzie odpowiadać wartości **selekторa**, zostanie wykonana instrukcja znajdująca się po słowie kluczowym **default**.

W powyższej definicji każda instrukcja po słowie kluczowym **case** kończy się instrukcją **break**, powodującą przejście wykonywania programu na koniec instrukcji **switch** (instrukcję tę kończy klamra, zamykająca nawias). Jest to typowy sposób tworzenia instrukcji **switch**, ale użycie instrukcji **break** jest opcjonalne. Jeżeli ona nie

występuje, program „przeskakuje” do instrukcji znajdującej się po następnym **słówie kluczowym case**. Oznacza to, że wykonywany jest kod kolejnych instrukcji case, aż do napotkania słowa **break**. Mimo że zazwyczaj takie działanie instrukcji jest **niepożądane**, to dla doświadczonego programisty może okazać się ono przydatne.

Instrukcja **switch** jest przejrzystym sposobem implementacji wielokierunkowej selekcji (czyli wyboru jednej z wielu ścieżek wykonywania programu), ale wymaga użycia selektora, który w czasie komplikacji programu ma wartość całkowitą. Na przykład łańcuch wykorzystany w charakterze selektora nie będzie działał z instrukcją **switch**. W przypadku selektora będącego łańcuchem należałoby **zamiast** tego użyć ciągu instrukcji if, porównując łańcuch w obrębie wyrażenia warunkowego.

Poprzedni program tworzący menu stanowi szczególnie wdzięczny przykład użycia instrukcji **switch**:

```
//: C03:Menu2.cpp
// Menu utworzone za pomocą instrukcji switch
#include <iostream>
using namespace std;

int main() {
    bool quit = false; // Znacznik wyjścia
    while(quit == false) {
        cout << "Wybierz a, b, c lub w, aby wyjsc: ";
        char response;
        cin >> response;
        switch(response) {
            case 'a' : cout << "wybrales 'a'" << endl;
                         break;
            case 'b' : cout << "wybrales 'b'" << endl;
                         break;
            case 'c' : cout << "wybrales 'c'" << endl;
                         break;
            case 'w' : cout << "wyjście z menu" << endl;
                         quit = true;
                         break;
            default   : cout << "Uzyj a,b,c lub w!" << endl;
                         break;
        }
    }
} //:-
```

Znacznik **quit** jest zmienną typu **bool** (skróć od ang. *Boolean* — boole’owski, logiczny), będącego typem spotykanym jedynie w C++. Przyjmuje on tylko wartości określone słowami kluczowymi **true** (prawda) lub **false** (fałsz). Wybranie „w” powoduje ustanie wartości **true** znacznika **quit**. Kiedy selektor pętli jest obliczany ponownie, warunek **quit == false** zwraca **false**, co powoduje, że wewnętrzne pętli niejest już wykonywane.

Używanie i nadużywanie instrukcji goto

Słowo kluczowe **goto** jest dostępne w C++, ponieważ istnieje ono w C. Używanie **goto** jest często odrzucane jako świadectwo złego stylu programowania, zresztą **zazwyczaj** słusznie. Ilekroć używasz **goto**, przyjrzyj się uważnie swojemu kodowi i sprawdź,

czy można napisać go w inny sposób. W wyjątkowych przypadkach zastosowanie **goto** może rozwiązać problem, z którym nie można sobie poradzić w inny sposób, ale należy dobrze przemyśleć ten wybór. Oto odpowiedni przykład:

```
//: C03:gotoKeyword.cpp
// Niesławna instrukcja goto jest dostępna w C++
#include <iostream>
using namespace std;

int main() {
    long val = 0;
    for(int i = 1; i < 1000; i++) {
        for(int j = 1; j < 100; j += 10) {
            val = i * j;
            if(val > 47000)
                goto bottom;
            // Instrukcja 'break' spowodowałaby jedynie
            // przejście do bardziej zewnętrznej instrukcji 'for'
        }
    }
    bottom: // Etykieta
    cout << val << endl;
} //:~
```

Alternatywne rozwiązanie polegałoby na ustaleniu znacznika logicznego, którego wartość byłaby sprawdzana w zewnętrznej pętli for, i na użyciu w wewnętrznej pętli instrukcji **break**. Jednakże w przypadku wielu poziomów pętli **for** lub **while** byłoby to niewygodne.

Rekurencja

Rekurencja jest interesującą i niekiedy użyteczną techniką programowania, polegającą na wywołaniu funkcji z jej wnętrza. Oczywiście, jeżeli jest to jedyne działanie, jakie ta funkcja wykonuje, to będzie ona wywoływaną aż do chwili, gdy skończy się pamięć. Musi zatem istnieć sposób umożliwiający wycofanie się z rekurencyjnych wywołań. W poniższym przykładzie uzyskuje się to, określając, że rekurencja kończy się po przekroczeniu przez argument **cat** znaku „Z”²:

```
//: C03:CatsInHats.cpp
// Prosty przykład rekurencji
#include <iostream>
using namespace std;

void removeHat(char cat) {
    for(char c - 'A'; c < cat; c++)
        cout << " ";
    if(cat <= 'Z') {
        cout << "kot " << cat << endl;
        removeHat(cat + 1); // Wywołanie rekurencyjne
    } else
        cout << "MIAAAU!!!" << endl ;
}
```

² Dziękuję Krisowi C. Matsonowi za sugestię wykorzystania tego przykładu.

```
int main() {
    removeHat('A');
} //:-~
```

A zatem dopóki cat jest mniejsze od „Z” w funkcji **removeHat()** jest ona wywoływana ze swojego wnętrza, czego wynikiem jest rekurencja. Podczas każdego wywołania funkcji **removeHat()** argument cat jest większy o jeden w stosunku do jego aktualnej wartości, co powoduje jego stałe zwiększenie.

Rekurencja jest często używana do rozwiązywania szczególnie złożonych problemów, ponieważ nie ogranicza ona „wielkości” rozwiązań — funkcja zagłębia się po prostu w rekurencję, dopóki nie zostanie znalezione rozwiązanie.

Wprowadzenie do operatorów

Operatory mogą być traktowane jako szczególny rodzaj funkcji (przekonasz się, że przeciążanie operatorów w C++ powoduje, że są one ujmowane właśnie w taki sposób). Operator działa na jednym lub kilku argumentach, a wynikiem jego działania jest zupełnie nowa wartość. Argumenty mają inną postać niż zwykłe wywołania funkcji, ale rezultat jest w obu przypadkach taki sam.

Jeżeli masz już jakieś doświadczenie programistyczne, to używane do tej pory operatory powinny być ci znajome. Pojęcia dodawania (+), odejmowania i zmiany znaku (-), mnożenia (*), dzielenia (/) i przypisania (=) mają zasadniczo to samo znaczenie w każdym języku programowania. Pełny zbiór operatorów został wymieniony w dalszej części rozdziału.

Priorytety

Priorytety operatorów określają, w jakiej kolejności są obliczane wyrażenia zawierające kilka różnych operatorów. W językach C oraz C++ występują specjalne reguły wyznaczające kolejność obliczeń. Najłatwiej zapamiętać, że mnożenie i dzielenie są wykonywane przed dodawaniem i odejmowaniem. Poza tym jeżeli masz problemy z interpretacją danego wyrażenia, to prawdopodobnie będzie ono również nieprzejrzyste dla każdej innej osoby czytającej twój kod. Należy zatem używać nawiasów, by wyraźnie określić kolejność wykonywania operacji. Na przykład wyrażenie:

$$A = X + Y - 2/2 + Z;$$

ma zupełnie inne znaczenie niż to samo wyrażenie, w którym następująco użyto nawiasów:

$$A = X + (Y - 2)/(2 + Z);$$

(spróbuj obliczyć wynik dla $X = 1$, $Y = 2$ i $Z = 3$).

Automatyczna inkrementacja i dekrementacja

Język C, a zatem również C++, jest pełen skrótów. Znacznie ułatwiają one wpisywanie kodu, ale czasami wyraźnie zmniejszają jego czytelność. Być może projektanci języka C uznali, że łatwiej zrozumieć skomplikowane fragmenty kodu, jeżeli nie trzeba błądzić wzrokiem po zbyt wielkich wydrukach.

Do najbardziej przydatnych skrótów należą operatory automatycznej **inkrementacji** i **dekrementacji**. Często używa się ich do zmiany wartości zmennych sterujących liczbą wykonania pętli.

Operatorem automatycznej dekrementacji jest „**—**”, co oznacza „zmniejszenie o jedną jednostkę”. Operatorem automatycznej inkrementacji jest „**++**” i oznacza „zwiększenie o jedną jednostkę”. Na przykład jeżeli A jest zmienną całkowitą, to wyrażenie **++A** jest równoważne zapisowi (A = A + 1). Operatory automatycznej inkrementacji i dekrementacji zwracają jako wynik wartość zmiennej. Jeżeli operator znajduje się przed zmienną (np. **++A**), to najpierw wykonywana jest operacja, a następnie wyznaczana zwracana wartość. Jeżeli natomiast operator znajduje się za zmienną (np. **A++**), to zwracana jest aktualna wartość zmiennej, a następnie wykonywana operacja. Na przykład:

```
//: C03:AutoIncrement.cpp
// Prezentacja użycia operatorów automatycznej
// inkrementacji i dekrementacji
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    int j = 0;
    cout << ++i << endl; // Preinkrementacja
    cout << j++ << endl; // Postinkrementacja
    cout << --i << endl; // Predkrementacja
    cout << j-- << endl; // Postdekrementacja
} //:-
```

Teraz już zapewne w pełni rozumiesz znaczenie nazwy „C++” — oznacza ona „krok naprzód w stosunku do C”.

Wprowadzenie do typów danych

Typy danych określają sposób użycia pamięci w pisany przez ciebie programie. Określając typ danych, przekazujesz kompilatorowi informację, jak utworzyć określony fragment pamięci, a także w jaki sposób wykonywać na nim operacje.

Rozróżniamy wbudowane i abstrakcyjne typy danych. Wbudowane typy danych to te, które wewnętrznie „rozumie” kompilator — są one w nim bezpośrednio zdefiniowane. W językach C i C++ występują niemal identyczne wbudowane typy danych. Inaczej jest z typami **zdefiniowanymi** przez użytkownika, będącymi klasami utworzonymi przez ciebie albo innego programistę. Często określa się je mianem abstrakcyjnych

typów danych. Kompilator zna wbudowane typy danych od momentu uruchomienia, natomiast „uczy się” obsługi abstrakcyjnych typów danych, czytając pliki nagłówkowe zawierające deklaracje klas (te zagadnienia zostały przedstawione w następnym rozdziale).

Podstawowe typy wbudowane

Specyfikacja standardu języka C, dotycząca wbudowanych typów danych (które dziedziczy C++), nie zawiera informacji o tym, z ilu bitów musi składać się każdy typ. Określa ona natomiast wymagania dotyczące minimalnej i maksymalnej wartości, które musi przechować każdy wbudowany typ danych. W przypadku gdy komputer pracuje w systemie dwójkowym, można bezpośrednio przełożyć tę maksymalną wartość na minimalną liczbę bitów, niezbędnych do jej przechowywania. Jeżeli jednak komputer używa do reprezentowania liczb na przykład liczb dziesiętnych kodowanych dwójkowo (ang. *binary-coded decimal – BCD*), to zajmowany obszar pamięci, **niezbędny** do przechowywania maksymalnych wartości każdego typu, będzie miał inną wielkość. Maksymalne i minimalne wartości, możliwe do wyrażenia w rozmaitych typach danych, zdefiniowano w systemowych plikach nagłówkowych **limits.h** i **float.h** (w dyrektywie `#include` języka C++ odwołania do nich umieścisz zazwyczaj w postaci `<climits>` i `<cfloat>`).

W językach C oraz C++ istnieją cztery podstawowe wbudowane typy danych, opisane w niniejszej książce dla komputerów działających w systemie dwójkowym. Typ **char** służy do przechowywania znaku i wykorzystuje przynajmniej 8 bitów (bajt) **pamięci**, chociaż może być również większy. Typ **int** przechowuje liczbę całkowitą i używa co najmniej dwóch bajtów pamięci. Typy **float** i **double** służą do przechowywania liczb **zmiennopozycyjnych**, zazwyczaj w formacie zmiennopozycyjnym IEEE. Typ **float** służy do przechowywania liczb zmiennopozycyjnych pojedynczej precyzji, a typ **double** — liczb zmiennopozycyjnych podwójnej precyzji.

Jak już wspomniano, zmienne **mogą być** definiowane w dowolnym miejscu ich zasięgu i można je równocześnie definiować i inicjalizować. Oto przykłady definicji zmiennych wykorzystujących cztery podstawowe typy danych:

```
//: C03:Basic.cpp
// Definiowanie czterech podstawowych
// typów danych C i C++

intmain() {
    // Definicja bez inicjalizacji:
    char protein;
    int carbohydrates;
    float fiber;
    double fat;
    // Równoczesna definicja i inicjalizacja:
    char pizza = 'A'. pop = 'Z';
    int dongdings = 100. twinkles = 150.
        heehos - 200;
    float chocolate = 3.14159;
    // Zapis wykładniczy:
    double fudge_ripple - 6e-4;
} //:-
```

W pierwszej części programu zdefiniowane są zmienne czterech podstawowych typów danych, bez ich inicjalizacji. Zgodnie ze standardem języka, wartość nieinicjalizowanej zmiennej jest nieokreślona (zazwyczaj oznacza to, że zawiera ona śmieci). W drugiej części programu zmienne są równocześnie zdefiniowane i zainicjalizowane (ilekroć to możliwe, najlepiej jest podawać wartość inicjalizującą zmiennej w miejscu jej definicji). Zwróć uwagę na zapis wykładniczy, użyty w stalej 6e-4, oznaczający „6 razy 10 podniesione do potęgi minus czwartej).

bool, true i false

Zanim do standardu C++ włączono typ **bool**, stosowano przeróżne techniki, aby zasymulować zachowanie zmiennych logicznych. Skutkowało to problemami z przenośnością i mogło stanowić źródło trudno uchwytnych błędów.

W standardowym C++ typ **bool** może posiadać dwa stany, opisywane wbudowanymi stałymi: **true** (któramożna przekształcić w wartość całkowitą równej jeden) oraz **false** (która można przekształcić w wartość całkowitą równą zero). Zarówno **bool**, jak i **true** oraz **false** są słowami kluczowymi. Ponadto do typu **bool** przystosowane zostały niektóre elementy języka:

Element	Wykorzystanie typu bool
<code>&& !</code>	Działają na argumentach typu bool , a wynikiem jest wartość typu bool .
<code>!_</code>	Wynikiem działania jest wartość typu bool .
<code>if, for, while, do</code>	Wartości wyrażeń warunkowych są przekształcone w wartości typu bool .
<code>? :</code>	Pierwszy argument jest zamieniany w wartość typu bool .

Ponieważ istnieje już mnóstwo programów, w których do reprezentowania znaczników jest używany typ **int**, kompilator zawsze przekształca typ **int** w typ **bool** (wartości niezerowe zamieniane są w **true**, a zero — w **false**). W idealnym przypadku kompilator powinien zgłosić ostrzeżenie, sugerujące wprowadzenie poprawek.

Praktyką należącą do kategorii „złego stylu programowania” jest używanie operatora `++` do nadawania znacznikowi wartości **true**. Jest to wprawdzie nadal dopuszczalne, ale *niewskazane*, co oznacza, że w bliższej lub dalszej przyszłości zostanie uznane za niedozwolone. Problem polega na tym, że w takim przypadku dokonywane jest niejawne przekształcenie typu **bool** w **int**, zwiększenie wartości o jeden (być może wykraczające poza normalny zakres wartości **bool**, czyli zero i jeden), a następnie ponownie niejawne przekształcenie do typu **bool**.

Eto typu **bool** są również, w razie potrzeby, automatycznie przekształcane wskaźniki (które zostaną wprowadzone w dalszej części rozdziału).

Specyfikatory

Specyfikatory zmieniają znaczenie podstawowych typów wbudowanych, znacznie powiększając ich zbiór. Istnieją cztery specyfikatory: **long**, **short**, **signed** i **unsigned**.

Specyfikatory **long** (długi) i **short** (krótki) zmieniają maksymalne i minimalne wartości, które mogą być przedstawione za pomocą typu. Zwykły typ int musi być co najmniej wielkości **short**. Hierarchia wielkości dla typów całkowitych jest następująca: **short int**, **int**, **long int**. Można sobie wyobrazić, że wielkość wszystkich tych typów będzie taka sama, dopóki będą one spełniać wymóg dotyczący wartości maksymalnych i minimalnych. Na przykład w komputerze, którego słowo jest 64-bitowe, **wszystkie** typy danych mogą mieć długość 64 bitów.

Hierarchia typów zmiennopozycyjnych jest następująca: **float**, **double** i **long double**. „**long float**” nie jest dopuszczalnym typem danych. Nie ma również „krótkich” (ang. **short**) liczb zmiennopozycyjnych.

Specyfikatory **signed** (ze znakiem) i **unsigned** (bez znaku) określają, w jaki sposób należy traktować bit znaku w typach całkowitych i znakowych. Liczba typu **unsigned** nie pamięta znaku, w związku z czym może wykorzystać jeden dodatkowy bit, dzięki któremu będzie w stanie przechowywać dwukrotnie większe liczby dodatnie niż liczba typu **signed**. Specyfikator **signed** jest przyjmowany domyślnie i jego użycie jest konieczne jedynie w przypadku typu **char**. Typ ten może być domyślnie (lecz nie musi) liczbą ze znakiem (signed). Zapis **signed char** wymusza wykorzystanie bitu znaku.

Zamieszczony poniżej przykładowy program wyświetla wyrażoną w bajtach wielkość poszczególnych typów danych, używając do tego operatora **sizeof**, który zostanie opisany w dalszej części rozdziału:

```
// C03:Specify.cpp
// Demonstracja użycia specyfikatorów
#include <iostream>
using namespace std;

int main() {
    char c;
    unsigned char cu;
    int i;
    unsigned int iu;
    short int is;
    short iis; // To samo. co short int
    unsigned short int isu;
    unsigned short iisu;
    long int i1;
    long iil; // To samo. co long int
    unsigned long int ilu;
    unsigned long iilu;
    float f;
    double d;
    long double ld;
    cout
```

```

<< "\n char= " << sizeof(c)
<< "\n unsigned char = " << sizeof(cu)
<< "\n int = " << sizeof(i)
<< "\n unsigned int = " << sizeof(iu)
<< "\n short = " << sizeof(is)
<< "\n unsigned short = " << sizeof(isu)
<< "\n long = " << sizeof(il)
<< "\n unsigned long = " << sizeof(ilu)
<< "\n float = " << sizeof(f)
<< "\n double = " << sizeof(d)
<< "\n long double = " << sizeof(lld)
<< endl;
} //:~
```

Zwróć uwagę na to, że wyniki uzyskane po uruchomieniu programu będą prawdopodobnie różne dla poszczególnych komputerów, systemów operacyjnych i kompilatorów, ponieważ (jak już wspomniano) jedynym wymaganiem jest, by każdy z tych typów był w stanie przechować wartości minimalne i maksymalne, określone w standardzie.

W przypadku modyfikacji liczby całkowitej za pomocą **specyfikatora short** lub **long** użycie słowa kluczowego **int** jest opcjonalne, o czym świadczy powyższy program.

Wprowadzenie do wskaźników

Uruchamiany program jest najpierw ładowany (przeważnie z dysku) do pamięci komputera. Tak więc wszystkie **elementy** programu są umieszczane w pamięci. Pamięć jest zorganizowana na ogół w postaci ciągu komórek, do których zazwyczaj odwołujemy się jako do **osmiobitowych bajtów**. W rzeczywistości jednak wielkość każdej z nich zależy od architektury komputera i jest nazywana **długością słowa**. Każde miejsce w pamięci można jednoznacznie odróżnić od pozostałych za pomocą **adresu**. W niniejszej książce założymy, że wszystkie komputery wykorzystują bajty, których adresy rozpoczynają się od zera i wzrastają aż do wielkości pamięci, w jaką wyposażony jest komputer.

Ponieważ program od chwili uruchomienia znajduje się w pamięci, każdy jego element ma określony adres. **Zacznijmy** od prostego programu:

```

//: C03:YourPetSl.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
    cout << "numer zwierzaka: " << pet << endl;
}

int main() {
    int i, j, k;
} //:~
```

W czasie wykonywania programu każdy jego element, nawet funkcja, zajmuje jakieś miejsce w pamięci. Okazuje się, że rodzaj obiektu i sposób jego zdefiniowania określają na ogół obszar pamięci, w którym będzie on umieszczony.

W językach C i C++ istnieje operator zwracający adres elementu. Operatorem tym jest „**&**”. Wystarczy poprzedzić znakiem „**&**” nazwę identyfikatora, a zwróci on jego adres. Można zmodyfikować program **YourPets1.cpp** w taki sposób, by drukował on adresy wszystkich swoich elementów:

```
//: C03:YourPets2.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
    cout << "numer zwierzaka: " << pet << endl;
}

int main() {
    int i, j, k;
    cout << "f(): " << (long)&f << endl;
    cout << "dog: " << (long)&dog << endl;
    cout << "cat: " << (long)&cat << endl;
    cout << "bird: " << (long)&bird << endl;
    cout << "fish: " << (long)&fish << endl;
    cout << "i: " << (long)&i << endl;
    cout << "j: " << (long)&j << endl;
    cout << "k: " << (long)&k << endl;
}   ///-
```

Zapis (**long**) oznacza *rzutowanie*. Oznacza on: „nie traktuj tego tak, jakby było zwykłym typem, lecz jako typ **long**”. Rzutowanie nie jest w tym przypadku konieczne, ale gdyby go nie było, adresy zostałyby wydrukowane w postaci szesnastkowej. Rzutowanie na typ **long** zwiększa zatem nieco ich czytelność.

Rezultat działania programu zależy od komputera, systemu operacyjnego i wielu innych czynników, ale i tak dostarczy ciekawych wskazówek. Uruchomienie go na moim komputerze dało następujące wyniki:

```
f(): 4198736
dog: 4323632
cat: 4323636
bird: 4323640
fish: 4323644
i: 6684160
j: 6684156
k: 6684152
```

Zmienne zdefiniowane wewnętrz funkcji **main()** znajdują się w innym obszarze pamięci niż zmienne zdefiniowane poza tą funkcją. Zrozumiesz, dlaczego tak się dzieje, kiedy dowiesz się więcej na temat języka. Wygląda również na to, że funkcja **f()** zajmuje odrębny obszar pamięci — kod jest w niej zazwyczaj oddzielony od danych.

Warto również wspomnieć, że kolejno definiowane po sobie zmienne wydają się umieszczone w ciągłym obszarze pamięci. Są one oddzielone od siebie liczbą bajtów, wymaganą przez ich typ danych. W powyższym przypadku jedynym typem danych jest int, a cat znajduje się cztery bajty dalej niż **dog**, **bird** — cztery bajty dalej niż **cat** itd. A zatem w tym komputerze typ **int** zajmuje cztery bajty.

Czy poza interesującym eksperymentem prezentującym organizację pamięci można jeszcze coś zrobić z uzyskanym adresem? Najważniejszą **czynnością**, którą można wykonać, jest zapisanie go w jakiejś zmiennej w celu późniejszego użycia. Języki C oraz C++ posiadają specjalny typ **zmiennych**, przechowujących adresy. Zmienne te są nazywane **wskaźnikami**.

Operatorem definiującym wskaźnik jest ten sam znak, który jest używany do oznaczania mnożenia — „*”. Kompilator „wie”, że nie jest to mnożenie, z kontekstu, w którym znak ten został użyty.

Definiując wskaźnik, musisz określić typ wskazywanej przez niego zmiennej. Należy zacząć od podania nazwy typu, a następnie, zamiast od razu podać identyfikator zmiennej, wstawić gwiazdkę pomiędzy nazwę typu i identyfikator zmiennej, co oznacza: „chwileczkę — to jest wskaźnik”. A zatem wskaźnik do zmiennej typu **int** jest następujący:

int* 1p: // ip wskazuje na zmienną typu int

Połączenie gwiazdki z nazwą typu wydaje się logiczne i czytelne, lecz bywa nieco mylące. Może powodować skłonność do traktowania „wskaźnika do liczby całkowitej” jako oddzielnego typu. Jednakże używając typu **int**, lub innego podstawowego typu danych, można zapisać:

int a, b, c;

natomiast za pomocą wskaźników *chciałbyś* zapewne napisać:

int* ipa, ipb, ipc;

Składnia języka C (oddziedziona również w C++) nie pozwala na taki logiczny zapis. W powyższej definicji tylko **ipa** jest **wskaźnikiem**, ipb i ipc są zaś zwykłymi liczbami całkowitymi (można by powiedzieć, że znak „*” jest bardziej związany z identyfikatorem zmiennej). A zatem najlepszy rezultat można osiągnąć umieszczając w każdym wierszu tylko jedną definicję. Dzięki temu uzyskuje się wyglądającą logicznie składnię, unikając wprowadzającego w błąd zapisu:

```
int* ipa;  
int* ipb;  
int* ipc;
```

Według generalnego zalecenia dotyczącego programowania w C++, nakazującego **inicjalizowanie** zmiennych w miejscu ich definicji, zapis **taki jest** lepszy. Na przykład powyższe zmienne nie zostały zainicjowane i **zawierają śmieci**. Znacznie lepiej jest napisać:

```
int a = 47;  
int* ipa = &a;
```

Teraz zmienność a oraz **ipa** zostały zainicjowane, a wskaźnik **ipa** zawiera adres zmiennej a.

Najprostszym działaniem, jakie można wykonać z zainicjowanym wskaźnikiem, jest zmiana wskazywanej przez niego wartości. Aby za pomocą wskaźnika uzyskać dostęp do zmiennej, należy dokonać operacji *wyłuskania* (ang. *dereference*), używając tego samego operatora, który posłużył do jego zdefiniowania, jak w poniższym przykładzie:

```
*ipa = 100;
```

Teraz zmienna a ma wartość 100, a nie 47.

Są to podstawy działania wskaźników — za ich pomocą można zapamiętać adres, używając go następnie do zmiany wartości wskazywanej zmiennej. Pozostaje jednak pytanie: po co modyfikować wartość jednej zmiennej, dokonując tego za pośredniczeniem drugiej?

Na potrzeby niniejszego wprowadzenia odpowiadamy na to pytanie, podając dwa rozległe obszary zastosowań wskaźników:

1. Zmiana „obiektów zewnętrznych” z wnętrza funkcji. Jest to, być może, najbardziej podstawowe wykorzystanie wskaźników i zostanie ono omówione poniżej.
2. Uzyskanie wielu innych przemyślnych technik programistycznych, opisanych w rozdziałach w dalszej części książki.

Modyfikacja obiektów zewnętrznych

W czasie przekazywania argumentu funkcji zwykle wewnątrz funkcji jest tworzona jego kopia. Działanie to nosi nazwę *przekazywania przez wartość* (ang. *pass by value*). Zostało ono zaprezentowane w poniższym programie:

```
//: C03:PassByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
    cout << "a = " << a << endl;
    a = 5;
    cout << "a = " << a << endl;
}

int main() {
    int x = 47;
    cout << "x = " << x << endl;
    f(x);
    cout << "x = " << x << endl;
} //:-
```

Zmienna a jest w funkcji **f()** *zmienną lokalną*, czyli istnieje tylko w czasie wywołania funkcji **f()**. Ponieważ jest to argument funkcji, wartość zmiennej a jest inicjowana za pomocą argumentu przekazywanego w trakcie wywołania funkcji. W funkcji **main()** tym argumentem jest zmienna x o wartości 47, zatem ta właśnie wartość jest kopiwana do zmiennej a w czasie wywołania funkcji **f()**.

Po uruchomieniu programu wyświetlane są następujące wyniki:

```
x = 47
a = 47
a = 5
x = 47
```

Początkowo wartość zmiennej x wynosi, oczywiście, 47. Podczas wywoływania funkcji **f()** jest przydzielany tymczasowy obszar pamięci, przechowujący wartość zmiennej a. Następnie zmienna a jest inicjowana za pomocą wartości zmiennej x, co potwierdza wydruk jej wartości. Można, oczywiście, zmienić wartość zmiennej a i pokazać, że uległa ona zmianie. Jednakże po zakończeniu funkcji **f()** tymczasowy obszar pamięci przydzielony zmiennej a przestaje istnieć i, jak widać, jedynym związkiem pomiędzy zmiennymi a i x było skopiowanie wartości zmiennej x do zmiennej a.

Z punktu widzenia wnętrza funkcji **f()** zmienna x jest *obiektem zewnętrznym* (termin używany przez autora). Modyfikacja zmiennej lokalnej nie powoduje, oczywiście, zmiany obiektu zewnętrznego, ponieważ znajdują się one w pamięci pod dwoma różnymi adresami. Jak jednak postąpić w sytuacji, gdy *chcemy* zmienić wartość obiektu zewnętrznego? W tym właśnie celu pomocne są wskaźniki. W pewnym sensie wskaźnik jest synonimem innej zmiennej. A zatem przekazując do funkcji wskaźnik, a nie zwykłą wartość, przesyłamy jej synonim zewnętrznego obiektu, co pozwala funkcji najego modyfikację, jak w poniższym przykładzie:

```
//: C03:PassAddress.cpp

#include <iostream>
using namespace std;

void f(int* p) {
    cout << "p = " << p << endl;
    cout << "*p = " << *p << endl;
    *p = 5;
    cout << "p = " << p << endl;
}

int main() {
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f(&x);
    cout << "x = " << x << endl;
} //:~
```

Teraz **f()** pobiera jako swój argument wskaźnik, dokonując, w czasie przypisania, operacji wyłuskania wskazywanej zmiennej, co powoduje modyfikację zewnętrznego obiektu x. Wyniki działania programu są następujące:

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

Warto zwrócić uwagę na to, że wartość zmiennej p jest taka sama, jak adres zmiennej x — czyli wskaźnik p faktycznie wskazuje na zmienną x. Jeżeli nie wydaje się to dość stateczne przekonujące, wystarczy zauważyc, że po przypisaniu wartości 5 zmiennej wyłuskanej ze wskaźnika p wartość x zostaje również zmieniona na 5.

A zatem przekazanie funkcji wskaźnika umożliwiającej modyfikację obiektu zewnętrznego. W dalszej części książki zaprezentowano jeszcze wiele innych sposobów użycia wskaźników, ale zmiana wartości obiektów zewnętrznych jest prawdopodobnie ich najbardziej podstawowym i najczęściej spotykanyem zastosowaniem.

Wprowadzenie do referencji

Zarówno w C, jak i w C++ wskaźniki działają podobnie, ale język C++ udostępnia dodatkowy sposób przekazywania funkcjom adresów, zwany *przekazywaniem przez referencję* (ang. *pass by reference*). Występuje on w wielu innych językach programowania, nie jest zatem wynalazkiem C++.

Na pierwszy rzut oka może się wydawać, że referencje są niepotrzebne i można napisać każdy program, wcale ich nie używając. Na ogół jest to prawda — z wyjątkiem kilku istotnych sytuacji, przedstawionych w dalszej części książki. Ogólna zasada działania referencji jest taka sama, jak wskaźników prezentowanych w poprzednim przykładzie — dzięki niej można przekazać funkcji adres argumentu. Różnica polega na tym, że wywołanie funkcji przyjmującej referencję jest, pod względem składni, bardziej przejrzyste niż wywołanie funkcji odbierającej wskaźniki (właściwie te różnice składniowe czynią referencje niezbędnymi w pewnych przypadkach). Jeżeli program **PassAddress.cpp** zostanie zmodyfikowany w taki sposób, by używał referencji, to będzie widoczna różnica w sposobie wywołania funkcji **f()** w obrębie funkcji **main()**:

```
//: C03:PassReference.cpp
#include <iostream>
using namespace std;

void f(int& r) {
    cout << "r = " << r << endl;
    cout << "&r = " << &r << endl;
    r = 5;
    cout << "r = " << r << endl;
}

int main() {
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f(x); // Wygląda, jak przekazanie przez wartość.
           // ale jest przekazaniem przez referencję
    cout << "x = " << x << endl;
} //:~
```

W celu przekazania do funkcji **f()** referencji należy — zamiast **int***, używanego do przekazania wskaźnika — podać najej liście argumentów **int&**. Odwołanie się do „**r**” (które zwracałoby adres, gdyby zmienna **r** była wskaźnikiem) wewnątrz funkcji **f()**

pozwala na uzyskanie wartości zmiennej, do której jest referencją. Jeżeli natomiast zmiennej r zostanie przypisana wartość, to także zostanie ona przypisana zmiennej, do której jest referencją. W rzeczywistości jedynym sposobem uzyskania adresu zawartego w zmiennej r jest użycie operatora „&”.

Zasadniczy efekt zastosowania referencji można wykryć wewnątrz funkcji **main()** – w składni wywołania funkcji **f()**, która ma postać **f(x)**. Mimo że wygląda to na zwykłe przekazanie argumentu przez wartość, w wyniku zastosowania referencji pobierany i przekazywany jest do funkcji adres zmiennej, a nie kopię jej wartości. Wyniki działania programu są następujące:

```
x = 47  
&x = 0065FEO0  
r = 47  
&r = 0065FEO0  
r = 5  
x = 5
```

A zatem przekazywanie argumentów przez referencję pozwala funkcji na modyfikację obiektu zewnętrznego w taki sam sposób, jak zastosowanie wskaźników (można również zauważyć, że referencja ukrywa fakt przekazywania adresu – zostanie to omówione w dalszej części książki). Tak więc, na potrzeby niniejszego uproszczonego wprowadzenia, można założyć, że referencje są odmiennym składniowo sposobem (czasami określa się je mianem „cukierka składniowego”) osiągnięcia tego samego celu, który uzyskuje się za pomocą wskaźników – pozwalały one funkcji na zmianę obiektów zewnętrznych.

Wskaźniki i referencje jako modyfikatory

Powyżej zostały przedstawione podstawowe typy danych **char**, **int**, **float** i **double** oraz specyfikatory **signed**, **unsigned**, **short** i **long**, które mogłyby być używane w niemal wszystkich kombinacjach wraz z podstawowymi typami danych. Teraz dodaliśmy wskaźniki i referencje, niezależne od podstawowych typów danych i specyfikatorów, dzięki czemu liczba możliwych kombinacji ulega potrojeniu:

```
//: C03:AllDefinitions.cpp  
// Wszystkie możliwe kombinacje podstawowych typów  
// danych, specyfikatorów, wskaźników i referencji  
#include <iostream>  
using namespace std;  
  
void f1(char c, int i, float f, double d);  
void f2(short int si, long int li, long double ld);  
void f3(unsigned char uc, unsigned int ui,  
       unsigned short int us1, unsigned long int uli);  
void f4(char* cp, int* ip, float* fp, double* dp);  
void f5(short int* sip, long int* lip,  
       long double* ldp);  
void f6(unsigned char* ucp, unsigned int* uip,  
       unsigned short int* usip,  
       unsigned long int* ulip);  
void f7(char& cr, int& ir, float& fr, double& dr);  
void f8(short int& sir, long int& lir,  
       long double& ldr);
```

```
void f9(unsigned char& ucr, unsigned int& uir,
        unsigned short int& usir,
        unsigned long int& ulir);

int main() {} //:-
```

Wskaźniki i referencje działają zarówno podczas przekazywaniu obiektów do, jak i na zewnątrz funkcji — przekonasz się o tym w kolejnych rozdziałach.

Istnieje jeszcze jeden typ, którego można użyć ze wskaźnikami — **void**. Określenie typu wskaźnika jako **void*** oznacza, że można do niego przypisać dowolny rodzaj adresu (podczas gdy w przypadku np. typu **int*** można przypisać do niego jedynie adres zmiennej typu **int**). Na przykład:

```
//: C03:VoidPointer.cpp
int main() {
    void* vp;
    char c;
    int i;
    float f;
    double d;
    // Do wskaźnika typu void* można
    // przypisać adres DOWOLNEGO typu:
    vp = &c;
    vp = &i;
    vp = &f;
    vp = &d;
} //:-
```

Po przypisaniu do wskaźnika typu **void*** tracona jest informacja o typie. Oznacza to, że przed użyciem wskaźnika konieczne jest jego rzutowanie na odpowiedni typ:

```
//: C03:CastFromVoidPointer.cpp
int main() {
    int i = 99;
    void* vp = &i;
    // Nie można dokonać wyłuskania ze wskaźnika void*:
    // *vp = 3; // Błąd komilacji
    // Przed wyłuskaniem, trzeba rzutować
    // z powrotem na typ int:
    *((int*)vp) = 3;
} //:-
```

Rzutowanie **(int*)vp** pobiera wskaźnik **void*** i informuje kompilator, by traktował go jako **int***, dzięki czemu można dokonać pomyślnie operacji wyłuskania. Nietrudno zauważać, że składnia tego wyrażenia jest nieelegancka, co gorsza — wskaźnik **void*** tworzy wyłom w systemie typów języka. Umożliwia on traktowanie jednego typu tak, jakby był on innym typem, a nawet temu sprzyja. W powyższym przykładzie typ **int** został potraktowany jako **int** — poprzez rzutowanie wskaźnika **vp** na typ **int***. Nic jednak nie stoi na przeszkodzie, aby rzutować go na **char*** albo **double***, co spowodowałoby zmianę innego obszaru pamięci niż przydzielonego zmiennej typu **int** i w konsekwencji prawdopodobnie załamanie wykonywania programu. Na ogół należy unikać wskaźników **void***, stosując je tylko w szczególnych sytuacjach, przedstawionych w dalszych partiach książki.

Nie można stosować referencji typu **void** — z powodów, które zostaną wyjaśnione w rozdziale 11.

Zasięg

Reguły zasięgu określają, gdzie zmienna jest dostępna, tworzona i niszczona (czyli gdzie kończy się jej zasięg). Zasięg zmiennej rozciąga się od miejsca, w którym została ona zdefiniowana, do pierwszego klamrowego nawiasu zamk傢cącego nawias rozpoczętyjący się w najbliższym miejscu poprzedzającym definicję zmiennej. Oznacza to, że zasięg jest określony przez „najbliższą” parę nawiasów klamrowych. Ilustruje to poniższy przykład:

```
//: C03:Scope.cpp
// Zasięg zmiennych
int main() {
    int scp1;
    // scp1 jest widoczna
    {
        // scp1 jest nadal widoczna
        // ....
        int scp2;
        // scp2 jest widoczna
        //....
        {
            // scp1 i scp2 są nadal widoczne
            //...
            int scp3;
            // scp1, scp2 i scp3 są widoczne
            //...
        } // <-- scp3 w tym miejscu przestała istnieć
        // scp3 nie jest dostępna
        // scp1 i scp2 są nadal widoczne
        //...
    } // <-- scp2 w tym miejscu przestała istnieć
    // scp3 i scp2 nie są w tym miejscu dostępne
    // scp1 jest nadal widoczna
    //...
} // <-- scp1 w tym miejscu przestała istnieć
//:-
```

Powyższy przykład pokazuje, kiedy zmienne są widoczne, a kiedy nie są one dostępne (tzn. znajdują się poza zasięgiem). Zmienne mogą być używane jedynie w obrębie swojego zasięgu. Zasięgi mogą być zagnieżdżone, o czym świadczą nawiasy klamrowe wewnętrz innich takich nawiasów. Zagnieżdżanie oznacza, że dostęp do zmiennej istnieje w zasięgu zawartym w aktualnym zasięgu. W powyższym przykładzie zmienna **scp1** jest dostępna wewnętrz wszystkich zasięgów, natomiast zmienna **scp3** — wyłącznie w najbardziej wewnętrzny zasięgu.

Definiowanie zmiennych „w locie”

Jak już wspomniano we wcześniejszej części rozdziału, pomiędzy C i C++ istnieje zasadnicza różnica w sposobie definiowania zmiennych. Oba języki wymagają, by zmienna została zdefiniowana przed użyciem, ale C (i wiele innych języków proceduralnych) wymusza definicję wszystkich zmiennych na początku zasięgu, dzięki czemu kompilator — tworząc blok — może przydzielić tym zmiennym pamięć.

W czasie czytania programu napisanego w języku C pierwszą rzeczą widoczną na początku zasięgu jest zazwyczaj blok definicji zmiennych. Deklarowanie wszystkich zmiennych na początku bloku wymaga od programisty tworzenia kodu w sposób wymuszony szczegółami implementacyjnymi języka. Przed napisaniem kodu zwykle nie wiadomo, jakie zmienne zostaną użyte. Powoduje to konieczność ciągłego przeskakiwania na początek bloku, co jest nie tylko niewygodne, ale może być również przyczyną błędów. Takie definicje zmiennych nie dostarczają na ogół żadnych informacji osobie czytającej program i są zazwyczaj mylące, ponieważ występują z dala od kontekstu, w którym zostały użyte.

Język C++ (ale nie C) pozwala na definicję zmiennych w dowolnym miejscu ich zasięgu, dzięki czemu można je definiować tuż przed użyciem. Ponadto w miejscu definicji zmiennej można dokonać jej inicjalizacji, co pozwala na uniknięcie niektórych rodzajów błędów. Taki sposób definiowania zmiennych powoduje, że program jest znacznie łatwiejszy do napisania, a także ogranicza błędy, spowodowane nieustannym przemieszczaniem się w obrębie zasięgu. Ułatwia to zrozumienie kodu, ponieważ definicje zmiennych są widoczne w takim kontekście, w jakim zmienne te są stosowane. Jest to szczególnie ważne w przypadku równoczesnej definicji i inicjalizacji zmiennej — znaczenie wartości inicjującej jest widoczne przy okazji użycia zainicjowanej zmiennej.

Zmienne można również definiować w obrębie wyrażeń sterujących pętli **for** i **while**, w warunku instrukcji **if** oraz w selektorze instrukcji **switch**. Poniżej zamieszczono przykłady definiowania zmiennych „w locie”:

```
//: C03:OnTheFly.cpp
// Definiowanie zmiennych "w locie"
#include <iostream>
using namespace std;

int main() {
    //..
    { // Początek nowego zasięgu
        int q = 0; // Język C wymaga definicji w tym miejscu
        //..
        // Definicja w miejscu użycia:
        for(int i = 0; i < 100; i++) {
            q++; // q pochodzi z szerszego zasięgu
            // Definicja na końcu zasięgu:
            int p = 12;
        }
        int p = 1; // Inne p
    } // Koniec zasięgu zawierającego q i zewnętrzne p
    cout << "Wpisz znaki:" << endl;
    while(char c = cin.get() != 'q') {
        cout << c << " to nie to" << endl;
        if(char x = c == 'a' || c == 'b')
            cout << "Wpisales a lub b" << endl;
        else
            cout << "Wpisales " << x << endl;
    }
    cout << "Wpisz A. B lub C" << endl;
    switch(int i = cin.get()) {
```

```

    case 'A': cout << "Bach" << endl; break;
    case 'B': cout << "Trach" << endl; break;
    case 'C': cout << "Bum" << endl; break;
    default: cout << "To nie jest A, B ani C!" << endl;
}
} //:~
```

Zmienna `p` jest zdefiniowana w obrębie najbardziej wewnętrznego zasięgu, tuż przed jego końcem — definicja ta jest więc zupełnie bezużyteczna (ale świadczy o tym, że zmienne rzeczywiście mogą być definiowane w dowolnym miejscu). Ta sama sytuacja ma miejsce w przypadku definicji zmiennej `p`, znajdującej się w bardziej zewnętrznym zasięgu.

Definicja zmiennej i w wyrażeniu kontrolnym pętli `for` jest przykładem możliwości zdefiniowania zmiennej *dokładnie* w miejscu, w którym jest ona potrzebna (można to zrobić tylko w C++). Zasięg zmiennej `i` jest zasięgiem wyrażenia kontrolowanego przez pętlę `for`, dzięki czemu można ponownie użyć tej zmiennej w następnej pętli `for`. Jest to wygodny i powszechnie stosowany styl programowania w języku C++ — i jest tradycyjną nazwą licznika pętli i nie ma potrzeby wymyślania dla niego nowych nazw.

Mimo że w prezentowanym przykładzie występują również zmienne definiowane w instrukcjach `while`, `if` oraz `switch`, to definicje takie nie są równie popularne, jak definicje znajdujące się w wyrażeniach instrukcji `for` — prawdopodobnie z uwagi na ograniczenia składniowe. Na przykład niemożliwe jest używanie w nich nawiasów — nie można zatem napisać:

```
while((char c = cin.get()) != 'q')
```

Dodatkowe nawiasy mogą wyglądać na użyteczne, ale z uwagi na brak możliwości ich użycia, rezultat wyrażenia różni się od spodziewanego. Przyczyną problemu jest to, że operator „`!=`” ma wyższy priorytet niż „`=`”, a zatem zmiennej znakowej `c` przypisywana jest wartość typu `bool`, przekształcona do typu `char`. Podczas jej wydruku na wielu terminalach pojawia się znak przedstawiający uśmiechniętą buzię.

Należy traktować możliwość definiowania zmiennych w instrukcjach `while`, `if` oraz `switch` jako wynikającą z konsekwencji, ale jedynym miejscem, w którym rzeczywiście stosuje się takie definicje (zresztą często), jest pętla `for`.

Specyfikacja przydziału pamięci

Podczas tworzenia zmiennej dostępnych jest wiele opcji określających czasie życia, sposób przydziału pamięci, a także to, w jaki sposób jest ona traktowana przez kompilator.

Zmienne globalne

Zmienne globalne są definiowane na zewnątrz ciał wszystkich funkcji i są dostępne dla wszystkich części programu (nawet znajdujących się w innych plikach). Na zmienne globalne nie ma wpływu zasięg i istnieją one przez cały czas (tj. czas życia

zmiennych globalnych dobiera się dopiero wraz z zakończeniem pracy programu). Jeżeli zmienna globalna, znajdująca się w jakimś pliku, została w innym pliku zadeklarowana przy użyciu słowa kluczowego **extern** (ang. *external* – zewnętrzny), to może ona być stosowana w pliku zawierającym tę deklarację. A oto przykład użycia zmiennych globalnych:

```
//: C03:Global.cpp
//{L} Global2
// Demonstracja zmiennych globalnych
#include <iostream>
using namespace std;

int globe;
void func();
int main() {
    globe = 12;
    cout << globe << endl;
    func(); // Modyfikuje zmienną globe
    cout << globe << endl;
} //:-
```

Poniżej przedstawiono treść pliku, który odwołuje się do zmiennej **globe**, zadeklarowanej jako zmienna zewnętrzna (**extern**):

```
//: C03:Global2.cpp {0}
// Dostęp do zewnętrznych zmiennych globalnych
extern int globe;
// (Odwołanie zostanie określone przez program łączący)
void func() {
    globe = 47;
} //:-
```

Pamięć jest przydzielana zmiennej **globe** na podstawie definicji, znajdującej się w pliku **Global.cpp**, i ta sama zmienna jest dostępna w kodzie zawartym w pliku **Global2.cpp**. Ponieważ kod zawarty w pliku **Global2.cpp** jest komplikowany niezależnie od kodu znajdującego się w pliku **Global.cpp**, kompilator musi być poinformowany, że zmienna ta istnieje w jakimś miejscu. Należy w tym celu użyć deklaracji:

```
extern int globe;
```

Po uruchomieniu programu jest oczywiste, że wywołanie funkcji **func()** rzeczywiście modyfikuje pojedynczy globalny egzemplarz zmiennej **globe**.

W pliku **Global.cpp** znajduje się specjalny, wymyślony przeze mnie, komentarz:

```
//{L} Global2
```

Informuje on, że do utworzenia programu konieczne jest dołączenie programu wynikowego o nazwie **Global2** (nie podano tu rozszerzenia nazwy pliku, ponieważ w przypadku programów wynikowych są one różne w różnych systemach). Komentarz znajdujący się w pierwszym wierszu pliku **Global2.cpp** zawiera inny specjalny znacznik **{O}**, który znaczy mniej więcej tyle: „nie próbuj na podstawie tego pliku tworzyć pliku wykonywalnego; jest on komplikowany po to, by mógł być włączony do jakiegoś innego pliku wykonywalnego”. Program **ExtractCode.cpp**, zawarty w drugim tomie książki (można go pobrać z witryny <http://helion.pl/online/thinking/index.html>),

odczytuje te znaczniki, tworząc odpowiedni plik **makefile**, dzięki czemu wszystko kompiluje się we właściwy sposób (informacje na temat plików **makefile** znajdują się na końcu niniejszego rozdziału).

Zmienne lokalne

Zmienne lokalne występują w obrębie jakiegoś zasięgu — są one „lokalne” w stosunku do funkcji. Często nazywa się je zmiennymi *automatycznymi*, ponieważ są one automatycznie tworzone przy wejściu do zasięgu, a następnie automatycznie usuwane po jego opuszczeniu. Użycie słowa kluczowego **auto** określa to wyraźnie, ale zmienne lokalne są zawsze domyślnie zmiennymi automatycznymi, nigdy więc nie ma potrzeby deklarowania czegokolwiek za pomocą specyfikatora **auto**.

Zmienne rejestrowe

Zmienna rejestrów jest rodzajem zmiennej lokalnej. Słowo kluczowe **register** nakazuje kompilatorowi, aby „dostęp do tej zmiennej był możliwie jak najszybszy”. Spód zwiększenia szybkości dostępu zależy od kompilatora, ale, jak wskazuje samo słowo kluczowe, polega on często na umieszczeniu zmiennej w rejestrze. Nie ma żadnych gwarancji, że zmienna zostanie ulokowana w rejestrze, ani nawet, że zwiększy się szybkość dostępu do niej. Jest to jedynie wskazówka dla kompilatora.

Używanie zmiennych rejestrowych podlega pewnym ograniczeniom. Mogą być one deklarowane jedynie w obrębie bloku (nie ma globalnych ani statycznych zmiennych rejestrowych). Zmiennych rejestrowych można jednak używać jako argumentów formalnych funkcji (tj. mogą one występować najwyżej liście argumentów).

Na ogół nie należy próbować wyręczać optymalizatora kompilatora, ponieważ prawdopodobnie sam dokona on lepszej optymalizacji. Dlatego też najlepiej jest unikać używania słowa kluczowego **register**.

static

Słowo kluczowe **static** (statyczny) ma wiele różnych znaczeń. Zazwyczaj zmienne zdefiniowane jako lokalne w stosunku do funkcji przestają istnieć, gdy kończy się jej zasięg. Ilekrótka funkcja jest wywoływana, zmiennym przydzielana jest pamięć, a ich wartości są ponownie inicjalizowane. Jeżeli potrzebna jest wartość, która będzie istniała przez cały okres pracy programu, można zdefiniować zmienną lokalną funkcji jako statyczną i nadać jej wartość początkową. Inicjalizacja ta jest dokonywana tylko podczas pierwszego wywołania funkcji, a dane zachowują swoje wartości pomiędzy wywołaniami funkcji. Dzięki temu funkcja może „pamiętać” pewne informacje pomiędzy swoimi wywołaniami.

Nasuwa się pytanie, czy nie można by użyć w takim przypadku zmiennych lokalnych. Urok zmiennych statycznych polega na tym, że nie są one dostępne poza zasięgiem funkcji, dzięki czemu ich wartości nie można przypadkowo zmienić. Pozwala to na ograniczenie zasięgu błędów.

Poniżej zamieszczono przykład użycia zmiennych statycznych:

```
//: C03:Static.cpp
// Użycie w funkcji zmiennych statycznych
#include <iostream>
using namespace std;

void func() {
    static int i = 0;
    cout << "i = " << ++i << endl;
}

int main() {
    for(int x = 0; x < 10; x++)
        func();
} ///-
```

Z każdym razem, gdy funkcja **func()** jest wywoływana w pętli **for**, drukuje ona inną wartość. Gdyby nie użyto słowa kluczowego **static**, wartość ta wynosiłaby zawsze „1”.

Drugie znaczenie słowa „statyczny” jest związane z jego pierwszym znaczeniem w zakresie „niewidoczności poza pewnym zasięgiem”. Kiedy słowo kluczowe **static** zostanie użyte w stosunku do nazwy funkcji lub zmiennej, znajdującej się na zewnątrz wszystkich funkcji, oznacza ono: „ta nazwa niejest dostępna poza bieżącym plikiem”. Nazwa funkcji lub zmiennej jest w takim przypadku lokalna w stosunku do pliku — ma ona *zasięg pliku*. Ilustruje to następujący przykład — skompilowanie i połączenie późniejszych dwóch plików wywoła zgłoszenie komunikatu o błędzie przez program łączący:

```
//: C03:FileStatic.cpp
// Demonstracja zasięgu pliku. Skompilowanie
// i połączenie pliku wraz z plikiem FileStatic2.cpp
// wywoła błąd programu łączącego

// Zasięg pliku oznacza dostępność wyłącznie
// w bieżącym pliku:
static int fs;

int main() {
    fs = 1;
} ///-
```

Mimo że zmienna **fs** została w następnym pliku określona jako zmienna zewnętrzna (**extern**), program łączący nie odnajdziejej, ponieważ w pliku **FileStatic.cpp** została ona zadeklarowana jako zmienna statyczna.

```
//: C03:FileStatic2.cpp {0}
// Próba odwołania do zmiennej fs
extern int fs;
void func() {
    fs = 100;
} ///:-
```

Specyfikator static może zostać również użyty w obrębie klasy. Zostanie to jednak wyjaśnione dopiero w dalszej części książki — po przedstawieniu zagadnień dotyczących tworzenia klas.

extern

Slowo kluczowe **extern** zostało już pokrótkie opisane i zaprezentowane. Informuje ono kompilator, że funkcja lub zmienna istnieje — nawet jeżeli kompilator nie napotkał jej do tej pory w aktualnie kompilowanym pliku. Ta zmienna lub funkcja może być zdefiniowana w innym pliku lub w dalszej części bieżącego pliku. Poniżej znajduje się przykład drugiego z wymienionych przypadków:

```
//: C03:Forward.cpp
// Wyprzedzające deklaracje funkcji i danych
#include <iostream>
using namespace std;

// Poniższe deklaracje nie dotyczą w rzeczywistości
// obiektów zewnętrznych, ale kompilator musi
// wiedzieć, że gdzieś one istnieją.
extern int i;
extern void func();
int main() {
    i = 0;
    func();
}
int i; // Definicja danych
void func() {
    i++;
    cout << i;
} //:~
```

Kiedy kompilator napotyka deklarację „**extern int i**”, dowiaduje się dzięki niej, że gdzieś musi się znajdować definicja zmiennej globalnej **i**. Kiedy dochodzi on do definicji zmiennej **i**, nie jest widoczna żadna inna deklaracja; „wie” więc, że znalazł to samo **i**, zadeklarowane we wcześniejszej części pliku. Gdyby zmienna **i** została zadeklarowana jako staticzna (**static**), oznaczałoby to informację dla kompilatora, że zmienna ta została zdefiniowana jako globalna (za pomocą **extern**). Jednakże ma ona zarazem zasięg ograniczony do pliku (przez użycie **static**), kompilator zgłosiłby zatem błąd.

Łączenie

Aby zrozumieć działanie programów napisanych w C oraz C++, trzeba poznać pojęcie **łączenia**. W wykonywanym programie identyfikator jest reprezentowany przez miejsce pamięci, przechowujące zmienną lub skompilowaną treść funkcji. Łączenie opisuje to miejsce z punktu widzenia programu łączącego. Istnieją dwa rodzaje łączenia: **łączenie wewnętrzne** i **łączenie zewnętrzne**.

Łączenie wewnętrzne oznacza, że miejsce w pamięci jest tworzone w celu reprezentowania identyfikatora wyłącznie na potrzeby kompilowanego pliku. Inne pliki mogą wykorzystywać tę samą nazwę identyfikatora do łączenia wewnętrznego lub jako nazwę zmiennej globalnej, nie powodując zgłoszania konfliktów przez program łączący — dla każdego identyfikatora jest bowiem tworzone odrębne miejsce w pamięci. Łączenie **wewnętrzne** jest w języku C i C++ określane za pomocą słowa kluczowego **static**.

Łączenie zewnętrzne oznacza, że w pamięci tworzone jest pojedyncze miejsce, reprezentujące identyfikator w stosunku do wszystkich komplikowanych plików. Miejsce to jest tworzone tylko jeden raz, a program łączący musi poradzić sobie z wszystkimi odnoszącymi się do niego odwołaniami. Zmienne globalne oraz nazwy funkcji są łączone zewnętrznie. Są one dostępne w innych plikach, po zadeklarowaniu ich za pomocą słowa kluczowego **extern**. Zmienne zdefiniowane poza ciałami funkcji (z wyjątkiem stałych w C++) oraz definicje funkcji są domyślnie łączone zewnętrznie. Można wymusić, by były one łączone wewnętrznie, używając słowa kluczowego **static**. Możajawnie określić, że dany identyfikator jest łączony zewnętrznie, definiując go za pomocą słowa kluczowego **extern**. W języku C nie jest konieczne używanie słowa kluczowego **extern** w celu definiowania zmiennych czy funkcji, ale może okazać się ono nieodzowne w przypadku stałych w C++.

Zmienne automatyczne (lokalne) istnieją tylko tymczasowo — na stosie, w czasie wywołania funkcji. Program łączący nie wie nic na temat zmiennych automatycznych, dlatego też *nie są one w ogóle łączone*.

State

W „dawnym” C (przed powstaniem standardu języka) do utworzenia stałej trzeba było użyć preprocesora:

```
#define PI 3.14159
```

Każde wystąpienie **PI** było zastępowane przez preprocesor wartością 3.14159 (nadal można używać tej metody — zarówno w C, jak i w C++).

W przypadku użycia do tworzenia stałych preprocesora, kontrola nad nimi znajduje się poza zasięgiem kompilatora. W stosunku do nazwy **PI** nie jest dokonywana żadna kontrola typów, nie sposób również określić adresu **PI** (nie można zatem przekazać wskaźnika ani referencji do **PI**). **PI** nie może być zmienną typu określonego przez użytkownika. Znaczenie **PI** rozciąga się od miejsca, w którym zostało ono zdefiniowane, do końca pliku — preprocesor nie rozróżnia bowiem zasięgów.

W języku C++ zostało wprowadzone pojęcie stałych nazwanych, działających tak samo, jak zmienne — z wyjątkiem tego, że ich wartości nie mogą być zmieniane. Modyfikator **const** informuje kompilator, że nazwa reprezentuje stałą. W charakterze stałej może być wykorzystany dowolny typ danych — zarówno wbudowany, jak i zdefiniowany przez użytkownika. Jeżeli podejmie się próbę modyfikacji wartości czegoś, co zostało zdefiniowane jako stała, spowoduje to zgłoszenie przez kompilator komunikatu o błędzie.

Typ stałej musi być określony, jak pokazano w przykładzie poniżej:

```
const int x = 10;
```

W standardowym C oraz C++ można używać stałych nazwanych na listach argumentów, nawet jeżeli odpowiadające im argumenty są wskaźnikami lub referencjami (oznacza to, że można określić adres stałej). Stałe mają swój zasięg, taki sam jak w przypadku zwyczajnych zmiennych, co pozwala na „ukrycie” stałej wewnątrz funkcji i zapewnia, że jej nazwa nie będzie miała wpływu na pozostałą część programu.

Stałe pochodzą z języka C++ i zostały one zaadaptowane na potrzeby standardu C w trochę odmienny sposób. W języku C kompilator traktuje stałe zupełnie tak samo jak zmienne, zaopatrzone w specjalny znaczek, oznaczający „nie zmieniaj mnie”. W przypadku definiowania stałych w języku C kompilator rezerwuje dla nich pamięć. Dlatego też, jeżeli zdefiniuje się w dwóch różnych plikach dwie stałe o tej samej nazwie (albo umieści definicję stałej w pliku nagłówkowym), program łączący zgłosi błąd wynikający z konfliktu nazw. Założenia dotyczące stosowania stałych w języku C różnią się nieco od założeń dotyczących ich wykorzystania w języku C++ (krótko mówiąc, w C++ wygodniej się ich używa).

Wartości stałych

W języku C++ stała musi zawsze posiadać inicjującą wartość (zasada ta nie obowiązuje w języku C). W przypadku typów wbudowanych wartości stałych mogą być wyrażone jako liczby dziesiętne, ósemkowe, szesnastkowe, zmiennopozycyjne (liczby dwójkowe nie zostały, niestety, uznane za ważne) albojako znaki.

W przypadku braku jakichkolwiek dodatkowych wskazówek kompilator zakłada, że wartość jest liczbą dziesiętną. Liczby 47, 0 i 1101 są więc traktowane jako liczby dziesiętne.

Stała rozpoczynająca się od 0 (zero) jest traktowana jako liczba ósemkowa (o podstawie 8). Liczby ósemkowe mogą się składać wyłącznie z cyfr od 0 do 7 — kompilator interpretuje pozostałe cyfry jako błędne. Przykładem prawidłowej liczby ósemkowej jest 017 (odpowiada ona w zapisie dziesiętnym liczbie 15).

Stała rozpoczynająca się od 0x jest traktowana jako liczba szesnastkowa (o podstawie 16). Liczby szesnastkowe składają się z cyfr od 0 do 9 oraz liter — od a do f (albo od A do F). Przykładem poprawnej liczby szesnastkowej jest 0x1fe (co odpowiada liczbie 510 w zapisie dziesiętnym).

Liczby zmiennopozycyjne mogą zawierać kropki dziesiętne³ oraz wykładniki potęgi (reprezentowane przez e, oznaczające „10 do potęgi”). Zarówno kropka dziesiętna, jak i e, są opcjonalne. Jeżeli do zmiennej zmiennopozycyjnej zostanie przypisana stała, kompilator dokona zamiany jej wartości na liczbę zmiennopozycyjną (proces ten jest jedną z postaci tzw. *niejawnej konwersji typów*). Dobrym zwyczajem jest jednak używanie kropki dziesiętnej albo litery e w celu przypomnienia osobie czytającej kod, że używana jest liczba zmiennopozycyjna — wskazówki takiej wymagają również niektóre ze starszych wersji kompilatorów.

Poprawnymi liczbami zmiennopozycyjnymi są: 1e4, 1.0001, 47.0, 0.0 i -1.159e-77. Aby wymusić typ liczby zmiennopozycyjnej, można do niej dopisać przyrostek. Przyrostki f oraz F oznaczają typ **float**, a L oraz 1 — typ **long double**. W każdym innym przypadku liczba jest typu **double**.

³ W językach C i C++ część ułamkowa liczby oddzielona jest kropką, a nie stosowanym w Polsce przecinkiem — przyp. tłum.

Stałe znakowe są znakami umieszczonymi pomiędzy znakami apostrofu, np. 'A', '0', ''. Zwróć uwagę na zasadniczą różnicę między znakiem '0' (kod ASCII 96) i wartością 0. Znaki specjalne są zapisywane w postaci „ukośnikowych sekwencji specjalnych”, rozpoczynających się znakiem lewego ukośnika: '\n' (nowy wiersz), '\t' (tabulacja), W (lewy ukośnik), V (powrót karetki), V" (cudzysłów), V (apostrof) itd. Stałe znakowe można również zapisywać w postaci ósemkowej: '\17' lub szesnastkowej: '\xff'.

volatile

Podczas gdy modyfikator **const** informuje kompilator: „ta wartość nigdy się nie zmienia” (co umożliwia mu dokonanie dodatkowej optymalizacji), modyfikator **volatile** (ulotny) komunikuje kompilatorowi: „nigdy nie wiadomo, kiedy zmienia się ta wartość”, powstrzymując go przed dokonywaniem jakichkolwiek optymalizacji, zakładających stabilność zmiennej. Tego słowa kluczowego należy używać podczas odczytu wartości znajdujących się poza kontrolą programu — np. rejestru elementu urządzenia komunikacyjnego. Zmienna oznaczona modyfikatorem **volatile** jest odczytywana zawsze wtedy, gdy potrzebna jest jej wartość, nawet jeżeli była już odczytywana w poprzednim wierszu programu.

Szczególny przypadek obszaru pamięci „znajdującego się poza kontrolą programu” występuje w programie wielowątkowym. Jeżeli obserwujesz jakiś znacznik, modyfikowany przez inny wątek lub proces programu, to znacznik ten powinien zostać oznaczony jako **volatile** — po to, by kompilator nie zakładał, że może zoptymalizować następujące po sobie odczytyjego wartości.

Zwróć uwagę na to, że użycie słowa kluczowego **volatile** może nie mieć żadnego skutku, jeśli kompilator nie dokonuje optymalizacji. Może natomiast uchronić cię przed poważnymi błędami, kiedy zaczniesz optymalizować swój kod (a kompilator rozpoczęcie poszukiwanie nadmiarowych odczytów wartości zmiennych).

Słowa kluczowe **const** i **volatile** zostaną opisane dokładniej w jednym z następnych rozdziałów.

Operatory i ich używanie

W rozdziale opisano wszystkie operatory występujące w C i C++.

Wszystkie operatory zwracają wartości na podstawie swoich argumentów. Wartość ta jest wyznaczana bez modyfikacji argumentów, z wyjątkiem operatorów: przypisania, inkrementacji i dekrementacji. Modyfikacja argumentu jest nazywana *skutkiem ubocznym* (ang. *side effect*). Najczęstsze zastosowanie operatorów modyfikujących swoje argumenty polega na wywołaniu skutku ubocznego, ale należy zapamiętać, że uzyskaną za ich pomocą wartość można wykorzystać tak samo, jak w przypadku operatorów pozbawionych skutków ubocznych.

Przypisanie

Przypisanie jest realizowane za pomocą operatora `=`. Oznacza ono: „prawą stronę (często nazywaną *p-wartością*) skopiuj na lewą stronę (nazywaną często *l-wartością*)”. *P-wartość* jest dowolną stałą, zmienną lub wyrażeniem zwracającym wartość, ale *l-wartość* musi być pojedynczą, nazwaną zmienną (czyli fizycznym miejscem, przeznaczonym do przechowywania danych). Na przykład można przypisać zmiennej stałą wartość (`A = 4;`), ale nie wolno przypisać niczego stałej wartości — nie może ona być *l-wartością* (błędny jest więc zapis: `4 = A;`).

Operatory matematyczne

Podstawowe operatory matematyczne są takie same, jak te dostępne w większości języków programowania: dodawanie (`+`), odejmowanie (`-`), dzielenie (`/`), mnożenie (`*`) oraz modulo (`%`, zwracający resztę z całkowitego dzielenia). Całkowite dzielenie obciną część ułamkową wyniku (nie jest on zaokrąglany). Operator modulo nie może być używany w stosunku do liczb zmiennopozycyjnych.

Języki C i C++ wykorzystują również skrócony zapis, umożliwiający równoczesne wykonanie danej operacji i przypisania. Jest on oznaczany jako operator, po którym następuje znak równości, i obowiązuje dla wszystkich operatorów dostępnych w języku (dla których taka operacja ma sens). Na przykład aby dodać 4 do zmiennej `x` i wynik tej operacji przypisać zmiennej `x`, należy napisać: `x += 4;`.

Poniższy przykład stanowi ilustrację sposobu używania operatorów matematycznych:

```
//: C03:Mathops.cpp
// Operatory matematyczne
#include <iostream>
using namespace std;

// Makroinstrukcja, wyświetlająca łańcuch i wartość
#define PRINT(STR, VAR) \
    cout << STR " " << VAR << endl

int main() {
    int i, j, k;
    float u, v, w; // Odnosi się również do liczb double
    cout << "Wprowadź liczbę całkowitą: ";
    cin >> j;
    cout << "Wprowadź jeszcze jedną liczbę całkowitą: ";
    cin >> k;
    PRINT("j", j); PRINT("k", k);
    i = j + k; PRINT("j + k", i);
    i = j - k; PRINT("j - k", i);
    i = k / j; PRINT("k / j", i);
    i = k * j; PRINT("k * j", i);
    i = k % j; PRINT("k % j", i);
    // Następne działają tylko z liczbami całkowitymi:
    j %= k; PRINT("j %= k", j);
    cout << "Wprowadź liczbę zmiennopozycyjną: ";
    cin >> v;
```

```

cout << "Wprowadź jeszcze jedną liczbę zmiennopozycyjną:";
cin >> w;
PRINT("v", v); PRINT("w", w);
u = v + w; PRINT("v + w", u);
u = v - w; PRINT("v - w", u);
u = v * w; PRINT("v * w", u);
u = v / w; PRINT("v / w", u);
// Następne działają z liczbami całkowitymi.
// znakami i liczbami double:
PRINT("u", u); PRINT("v", v);
u += v; PRINT("u += v", u);
u -= v; PRINT("u -= v", u);
u *= v; PRINT("u *= v", u);
u /= v; PRINT("u /= v", u);
} //:-

```

Oczywiście, *p-wartości* wszystkich przypisań, mogą być bardziej skomplikowane.

Wprowadzenie do makroinstrukcji preprocessora

Zwróć uwagę na użycie makroinstrukcji **PRINT()**, pozwalającej oszczędzić na pisaniu (i chroniącej przed popełnianiem pomyłek!). Makroinstrukcje preprocessora są tradycyjnie zapisywane wielkimi literami, dzięki czemu rzucają się w oczy. Mogą być one niebezpiecznym (ale także niezwykle użytecznym) narzędziem.

Argumenty znajdujące się w nawiasie po nazwie makroinstrukcji są zastępowane w całym kodzie, następującym po nawiasie zamykającym. Preprocesor usuwa nazwę **PRINT** i zastępuje ją tym kodem w każdym miejscu wywołania makroinstrukcji, co powoduje, że kompilator nie jest w stanie wygenerować żadnego komunikatu zawierającego nazwę makroinstrukcji. Nie przeprowadza również żadnej kontroli typów dotyczącej jej argumentów (to ostatnie może być *zaletą*, jak zobaczymy na przykładzie makroinstrukcji uruchomieniowych, przedstawionych na końcu rozdziału).

Operatory relacji

Operatory relacji określają związek pomiędzy wartościami argumentów. Zwracają one wartość logiczną (oznaczaną w języku C++ słowem kluczowym **bool**) **true** — jeżeli relacja jest prawdziwa — lub **false** — jeżeli jest ona fałszywa. Operatorami relacji są: mniejsze niż (<), większe niż (>), mniejsze lub równe (<=), większe lub równe (>=), równe (==) i nierówne (!=). Mogą one być używane z wbudowanymi typami danych — zarówno w C, jak i w C++. W języku C++ można przypisać im również specjalne definicje, umożliwiające ich stosowanie wraz z typami zdefiniowanymi przez użytkownika (zagadnienie to zostanie przedstawione w rozdziale 12., opisującym przeciążanie operatorów).

Operatory logiczne

Operatory *iloczynu logicznego* (**&&**) i *sumy logicznej* (**||**) zwracają wartości **true** lub **false** na podstawie logicznej relacji swych argumentów. Należy pamiętać, że w językach

C i C++ wyrażenie jest prawdziwe, jeżeli ma ono wartość niezerową, a fałszywe, kiedy jest one równe zeru. Podczas wydruku wartości typu **bool** zazwyczaj pojawiają się znaki '1' (dla **true**) i '0' (dla **false**).

W poniższym przykładzie zostały wykorzystane operatory relacji i operatory logiczne:

```
//: C03:Boolean.cpp
// Operatory relacji i operatory logiczne
#include <iostream>
using namespace std;

int main() {
    int i,j;
    cout << "Wprowadź liczbę całkowitą: ";
    cin >> i;
    cout << "Wprowadź jeszcze jedną liczbę całkowitą: ";
    cin >> j;
    cout << "i > j wynosi " << (i > j) << endl;
    cout << "i < j wynosi " << (i < j) << endl;
    cout << "i >= j wynosi " << (i >= j) << endl;
    cout << "i <= j wynosi " << (i <= j) << endl;
    cout << "i == j wynosi " << (i == j) << endl;
    cout << "i != j wynosi " << (i != j) << endl;
    cout << "i && j wynosi " << (i && j) << endl;
    cout << "i || j wynosi " << (i || j) << endl;
    cout << "(i < 10) && (j < 10) wynosi "
        << ((i < 10) && (j < 10)) << endl;
} //:-
```

W powyższym przykładzie można wymienić definicje typu zmiennych **int**, typu **float** lub **double**. Należy jednak wiedzieć o tym, że porównanie liczb zmiennopozycyjnej z zerem jest dokładne — liczba różniąca się od innej liczby w najmniejszej choćby częścią której „nierówna”. Liczba zmiennopozycyjna, której nawet najmniej znaczący bit jest niezerowy, jest natomiast traktowana jako „prawdziwa”.

Operatory bitowe

Operatory bitowe umożliwiają operacje na poszczególnych bitach liczby (ponieważ liczby zmiennopozycyjne wykorzystują specjalny wewnętrzny format zapisu informacji, operatory bitowe działają tylko z typami całkowitymi: **char**, **int** i **long**). Operatory bitowe zwracają wartości, wykonując operacje logiczne na odpowiadających sobie bitach argumentów.

Bitowy operator **koniunkcji** (**&**) zwraca bit o wartości jeden, gdy oba bity wejściowe mają wartość jeden, a zero — w przeciwnym przypadku. Bitowy operator **alternatywy** (**◊**) zwraca bit o wartości jeden, gdy dowolny z bitów wejściowych jest równy jeden, a bit o wartości zero — tylko **wówczas**, gdy oba bity wejściowe są równe zeru. Bitowy operator **różnicy symetrycznej** (**'**) zwraca wartość jeden, gdy dowolny z bitów wejściowych wynosi jeden, ale nie oba równocześnie. Bitowy operator **negacji** (**~**, nazywany również **uzupełnieniem do jedynki**) jest operatorem **jednoargumentowym** (wszystkie pozostałe operatory bitowe są operatormi **dwoargumentowymi**). Operator ten zwraca odwrotność bitu wejściowego — **jeden**, gdy wynosi on **zero**, a zero, gdy jest on **równy jeden**.

Operatory bitowe można zestawiać ze znakiem `=`, łącząc w ten sposób wykonanie operacji z przypisaniem — poprawnymi operacjami są więc: `&=`, `|=` i `^=` (ponieważ ~ jest **operatorem jednoargumentowym**, nie można łączyć go ze znakiem `=`).

Operatory przesunięć

Operatory przesunięć również działają na bitach. Operator przesunięcia w lewo (`<<`) zwraca wartość argumentu po jego lewej stronie, przesuniętą w lewo o liczbę bitów określoną przez argument po jego prawej stronie. Operator przesunięcia w prawo (`>>`) zwraca wartość argumentu znajdującego się po jego lewej stronie, przesuniętą w prawo o liczbę bitów określoną przez argument po jego prawej stronie. Jeżeli wartość argumentu po prawej stronie operatora jest większa niż liczba bitów argumentu pojego lewej stronie, wynik operacji jest nieokreślony. Jeżeli argument po lewej stronie operatora przesunięcia jest liczbą bez znaku, to przesunięcie w prawo jest przesunięciem logicznym (najstarsze bity są wypełniane zerami). Jeżeli natomiast argument po lewej stronie operatora jest liczbą ze znakiem, to przesunięcie w prawo może, ale wcale nie musi, być przesunięciem logicznym (czyli zachowanie operatora jest niezdefiniowane).

Operatory przesunięć mogą być połączone ze znakiem równości (`<<=` i `>>=`). W takim przypadku w wyniku operacji *l-wartość jest zastępowana l-wartością przesuniętą o liczbę bitów określoną przez p-wartość*.

Zaprezentowany poniżej przykład ilustruje użycie wszystkich operatorów działających na bitach. Na początku zdefiniowano funkcję ogólnego przeznaczenia, drukującą bajty w zapisie dwójkowym. Funkcja ta znajduje się w oddzielnym pliku, dzięki czemu można będzie można w łatwy sposób ponownie ją wykorzystać. Zadeklarowano ją w pliku nagłówkowym:

```
//: C03:printBinary.h
// Wyświetla bajt w zapisie dwójkowym
void printBinary(const unsigned char val);
///:-
```

A oto implementacja tej funkcji:

```
//: C03:printBinary.cpp {0}
#include <iostream>
void printBinary(const unsigned char val) {
    for(int i = 7; i >= 0; i--)
        if(val & (1 << i))
            std::cout << "1";
        else
            std::cout << "0";
}    ///-
```

Funkcja **printBinary()** przyjmuje jako argument pojedynczy bajt i wyświetla go, bit po bicie. Wyrażenie:

`(1 << i)`

zwraca **jedynkę** na każdej kolejnej pozycji bitu — w zapisie dwójkowym: 00000001, 00000010 itd. Jeżeli wyznaczy się **koniunkcję** tego bitu z wartością zmiennej `val` i uzyskany wynik będzie niezerowy, będzie to oznaczało obecność jedynki na odpowiadającej temu bitowi pozycji zmiennej `val`.

Ostatecznie funkcja została użyta w przykładzie, prezentującym operatory działające na bitach:

```
//: C03:Bitwise.cpp
//{L} printBinary
// Demonstracja operacji na bitach
#include "printBinary.h"
#include <iostream>
using namespace std;

// Makroinstrukcja, pozwalająca oszczędzić na pisaniu:
#define PR(STR, EXPR) \
    cout << STR; printBinary(EXPR); cout << endl;

int main() {
    unsigned int getval;
    unsigned char a, b;
    cout << "Wprowadź liczbę z zakresu od 0 do 255: ";
    cin >> getval; a = getval;
    PR("a w zapisie dwójkowym: ", a);
    cout << " Wprowadź liczbę z zakresu od 0 do 255: ";
    cin >> getval; b = getval;
    PR("b w zapisie dwójkowym: ", b);
    PR("a | b=".a | b);
    PR("a & b = ", a & b);
    PR("a ^ b = ", a ^ b);
    PR("~a = ", ~a);
    PR("~b = ", ~b);
    // Interesująca sekwencja bitów:
    unsigned char c = 0x5A;
    PR("c w zapisie dwójkowym: ", c);
    a |= c;
    PR("a |= c; a = ", a);
    b &= c;
    PR("b &= c; b = ", b);
    b ^= a;
    PR("b ^= a; b = ", b);
}
} //:~
```

Ponownie, w celu oszczędzenia sobie pisania, została użyta makroinstrukcja. Drukuje ona dowolny łańcuch, a następnie dwójkową reprezentację wyrażenia i znak nowego wiersza.

Zmienne w funkcji **main()** są typu **unsigned**, ponieważ w przypadku operacji na bitach znaki nie są na ogół potrzebne. Zmienna **getval** musi być zdefiniowana jako zmienna typu **int**, a nie **char** — z uwagi na to, że w przeciwnym przypadku instrukcja „**cin >>**” potraktowałaby pierwszą frejako znak. Podczas przypisywania zmiennym **a** i **b** wartości zmiennej **getval** jest ona przekształcana do pojedynczego bajtu (przez odcięcie pozostałych bitów).

Operatory **<<** i **>>** umożliwiają przesuwanie bitów, ale w przypadku gdy bity zostaną przesunięte poza granice liczby, są one tracone (często powiada się, że trafiają one do *zbiornika na bity* — miejsca, w którym znajdują się odrzucone bity, dzięki czemu mogą one być przypuszczalnie z powrotem **użyte**...). W czasie wykonywania operacji na bitach można również dokonać *obrotu*, co oznacza, że bity, które „wychodzą”

z jednego końca liczby są wstawiane na jej drugi koniec — tak jakby obracały się w kółko. Mimo że większość procesorów znajdujących się w komputerach posiada sprzętowe rozkazy obrotów (są one dostępne w języku asemblera), to języki C oraz C++ nie zapewniają bezpośredniego wsparcia dla instrukcji „obrotu”. Przypuszczałnie projektanci języka C uznali za usprawiedliwione pominięcie tej instrukcji (koncentrując się, jak twierdzili, na „języku minimalnym”), ponieważ można utworzyć własne polecenia, realizujące obroty. Poniżej zamieszczono przykładowe funkcje, dokonujące obrotów w lewo oraz w prawo:

```
// C03:Rotation.cpp {0}
// Wykonywanie obrotów w lewo i w prawo

unsigned char rol(unsigned char val) {
    int highbit;
    if(val & 0x80) // 0x80 jest najstarszym bitem
        highbit = 1;
    else
        highbit = 0;
    // Przesunięcie w lewo (najmłodszy bit
    // otrzymuje wartość 0):
    val <<= 1;
    // Wstawienie najstarszego bitu na najmłodszej pozycji:
    val |= highbit;
    return val;
}

unsigned char ror(unsigned char val) {
    int lowbit;
    if(val & 1) // Sprawdzenie wartości najmłodszego bitu
        lowbit = 1;
    else
        lowbit = 0;
    val >>= 1; // Przesunięcie w prawo o jedną pozycję
    // Wstawienie najmłodszego bitu na najstarszej pozycji:
    val |= (lowbit << 7);
    return val;
} // :~
```

Spróbuj użyć tych funkcji w programie **Bitwise.cpp**. Zwróć uwagę na to, że definicje (a przynajmniej deklaracje) funkcji **rol()** i **ror()** muszą być widoczne dla kompilatora w pliku **Bitwise.cpp**, zanim jeszcze funkcje te zostaną wykorzystane.

Stosowanie funkcji operujących na bitach jest na ogół wyjątkowo efektywne, ponieważ przekładają się one bezpośrednio na instrukcje asemblera. W niektórych przypadkach pojedyncza instrukcja języka C lub C++ generuje pojedynczy wiersz kodu w języku asemblera.

Operatory jednoargumentowe

Bitowy operator *negacji* nie jest jedynym operatorem działającym na pojedynczym argumentem. Jego odpowiednik, operator *negacji logicznej* (!), dla argumentu o wartości **true** zwraca wartość **false**. Jednoargumentowe operatory: minus (-) i plus (+) wyglądają tak samo, jak dwuargumentowe operatory odejmowania i dodawania — kompilator

określa, w jakim znaczeniu zostały one użyte, na podstawie tego, jak **zostało** zapisane wyrażenie. Na przykład instrukcja:

```
x = -a;
```

ma oczywiste znaczenie. Kompilator potrafi również określić znaczenie instrukcji:

```
x = a * -b;
```

ale **czytająca** jedna osoba może być nieco zdezorientowana, dlatego też bezpieczniej jest napisać:

```
x = a * (-b);
```

Jednoargumentowy operator minus zwraca ujemną wartość argumentu. **Jednoargumentowy** operator plus istnieje dla symetrii, chociaż w rzeczywistości nie wykonuje żadnych działań.

Operatory inkrementacji i dekrementacji (++ i --) zostały już wprowadzone we wcześniejszej części rozdziału. Są one jedynymi operatorami (z wyjątkiem operatorów zawierających przypisanie), które posiadają skutki uboczne. Operatory te zwiększały lub zmniejszały zmienną o jedną jednostkę, choć pojęcie „jednostki” ma różne znaczenia, w zależności od typu danych — szczególnie w przypadku wskaźników.

Ostatnimi operatorami jednoargumentowymi są w językach C i C++ operatory: adresu (&), wyłuskania (* i ->) oraz rzutowania, a w języku C++ — dodatkowo — **new** i **delete**. Operatory adresu i wyłuskania używane są ze wskaźnikami i zostały opisane w niniejszym rozdziale. Rzutowanie zostanie omówione w dalszej części rozdziału, natomiast operatory **new** i **delete** wprowadzono w rozdziale 4.

Operator trójargumentowy

Operator trójargumentowy **if-else** jest operatorem o niezwykłych właściwościach, ponieważ działa on na trzech argumentach. Jest on naprawdę operatorem, ponieważ — w przeciwieństwie do zwykłej instrukcji **if-else** — zwraca wartość. Składa się on z trzech wyrażeń: jeżeli pierwsze z nich (znajdujące się przed znakiem ?) daje wartość **true**, to wyznaczaną jest wartość wyrażenia następującego po ? i staje się ona zarówno wartością zwracaną przez cały operator. Jeżeli natomiast pierwsze wyrażenie daje wartość **false**, to wyznaczana jest wartość trzeciego wyrażenia (znajdującego się po znaku :) i to ona staje się wartością zwracaną przez operator.

Operator warunkowy może być używany z uwagi na swoje skutki uboczne lub zwracaną wartość. Poniższy przykład ilustruje oba te zastosowania:

```
a = --b ? b : (b = -99);
```

W przykładzie tym operator warunkowy **zwraca p-wartość**. Jeżeli rezultat dekrementacji zmiennej b jest **niezerowy**, to jest on przypisywany zmiennej a. Jeżeli natomiast zmienność b osiąga wartość **zerową**, to zarówno zmiennej a, jak i b jest przypisywana wartość -99. Zmienność b jest dekrementowana zawsze, ale wartość -99 przypisywana jest jej tylko wówczas, gdy w wyniku dekrementacji zmienność ta osiągnie wartość 0. Podobna instrukcja może być używana bez „**a =**” — wyłącznie z uwagi na swoje skutki uboczne:

```
--b ? b : (b = -99);
```

W powyższym **przykładzie** drugie wystąpienie zmiennej *b* jest nadmiarowe, gdyż wartość zwracana przez operator nie jest do niczego używana. Ponieważ jednak międy *? a* : wymagane jest wyrażenie, w tym przypadku może ono być po prostu stała, dzięki czemu program zadziała nieco szybciej.

Operator przecinkowy

Użycie przecinka nie jest ograniczone wyłącznie do oddzielenia od siebie nazw w definicji wielu zmiennych, jak w przykładzie poniżej:

```
int i, j, k;
```

Oczywiście, jest on również używany na listach argumentów funkcji. Jednakże przecinek może być także używany jako operator oddzielający od siebie wyrażenia — w takim przypadku zwraca on jedynie wartość ostatniego z nich. Wszystkie pozostałe wyrażenia, odseparowane od siebie przecinkami, są obliczane wyłącznie z uwagi na swoje skutki uboczne. W poniższym przykładzie inkrementowana jest lista zmiennych, a ostatnia z nich jest używana jako *op-wartość*:

```
II: C03:CommaOperator.cpp
#include <iostream>
using namespace std;
int main() {
    int a = 0, b = 1, c = 2, d = 3, e = 4;
    a - (b++, c++, d++, e++);
    cout << "a = " << a << endl;
    // Powyższy nawias jest konieczny. Bez niego
    // instrukcja byłaby wykonywana jako:
    (a = b++), c++, d++, e++;
    cout << "a = " << a << endl;
} III-
```

Najlepiej jest unikać używania przecinka inaczej niż jako separatora, ponieważ zazwyczaj nie postrzega się go jako operatora.

Najczęstsze pułapki związane z używaniem operatorów

Jak wynika z powyższego przykładu, jedną z pułapek związanych z używaniem operatorów jest próba pozbycia się nawiasów w przypadku choćby najmniejszej wątpliwości dotyczącej sposobu obliczania wyrażenia (kolejność wykonywania operacji można znaleźć w dowolnym podręczniku C).

Inny, wyjątkowo często spotykany błąd jest następujący:

```
//: C03:Pitfall.cpp
// Pomyłki w używaniu operatorów
```

```
int main() {
    int a = 1, b = 1;
    while(a == b) {
        // ...
    }
} // :~
```

Instrukcja `a = b` będzie zwracać wartość `true` zawsze wtedy, gdy `b` ma wartość niezerową. Zmiennej `a` jest przypisywana wartość zmiennej `b` i jest ona również zwracana przez operator `=`. Na ogół wewnątrz wyrażeń warunkowych zamierzamy użyć operatora porównania `==`, a nie przypisania. Dało się to we znaki niejednemu programiście (jednakże niektóre kompilatory sygnalizują takie problemy, co jest niezwykle pomocne).

Podobnym problemem jest użycie bitowych operatorów *koniunkcji* i *alternatywy* zamiast ich logicznych odpowiedników. Operatory *bitowej koniunkcji* i *alternatywy* są oznaczane pojedynczymi znakami (`&` i `|`), podczas gdy *logiczny iloczyn* i *suma* składają się z dwóch znaków (`&&` i `||`). Podobnie jak w przypadku `==` i `=`, łatwo zamiast dwóch znaków wpisać jeden. Użyteczną techniką mnemotechniczną może być spostrzeжение, że „bity są mniejsze, więc nie potrzebują aż tylu znaków w swoich operatorach”.

Operatory rzutowania

Termin *rzutowanie* używany jest tu w znaczeniu „przenoszenia na inną płaszczyznę”. Kompilator automatycznie przekształca jeden typ danych w inny, jeżeli jest to możliwe do zaakceptowania. Na przykład jeżeli zmiennej zmennopozycyjnej zostanie przypisana wartość całkowita, kompilator w niejawny sposób wywołuje funkcję (albo, co bardziej prawdopodobne — wstawia kod) przekształcającą typ `int` w `float`. Rzutowanie pozwala na uzyskanie takiej konwersji jawną albo wymuszenie jej w przypadkach, w których nie zostałaby ona wykonana.

Aby dokonać rzutowania, należy umieścić docelowy typ danych (łącznie ze wszystkimi modyfikatorami) w nawiasie, po lewej stronie rzutowanej wartości. Wartość ta może być *zmienną*, *stałą*, wynikiem wyrażenia albo wartością zwracaną przez funkcję. Oto przykład rzutowania:

```
//: C03:SimpleCast.cpp
int main() {
    int b = 200;
    unsigned long a = (unsigned long int)b;
}
```

Rzutowanie jest skutecznym narzędziem, ale może ono nastręczyć problemów, ponieważ w pewnych sytuacjach wymusza na kompilatorze traktowanie danych w taki sposób, jakby były one (na przykład) większe niż w rzeczywistości. Zajmą one zatem większy obszar pamięci — co z kolei może naruszyć inne dane. Sytuacje takie zdarzają się zazwyczaj podczas rzutowania wskaźników, a nie zwykłego rzutowania — takiego, jak przedstawione powyżej.

W języku C++ istnieje dodatkowa składnia operacji rzutowania, naśladowująca sposób wywołania funkcji. W zapisie tym nie docelowy typ danych, ale argument znajduje się w nawiasie — podobnie jak podczas wywołania funkcji.

```
//: C03:FunctionCallCast.cpp
int main() {
    float a = float(200);
    // Jest to równoważne:
    float b = (float)200;
} //:~
```

Oczywiście, w powyższym przykładzie **nie jest** naprawdę potrzebne rzutowanie — można napisać po prostu **200f** (w istocie kompilator to właśnie zrobi z powyższym wyrażeniem). Rzutowanie jest na ogół częściej używane w stosunku do zmiennych niż do funkcji.

Jawne rzutowanie w C++

Rzutowanie powinno być stosowane ostrożnie, ponieważ w rzeczywistości nakazuje ono kompilatorowi: „pomiń kontrolę typów — traktuj to jak zupełnie inny typ”. Oznacza to, że tworzysz lukę w systemie typów C++, powstrzymując kompilator przed poinformowaniem cię, że twoje działania dotyczące typów są niewłaściwe. Co gorsza, kompilator nie przeprowadza żadnych dodatkowych kontroli, które pozwoląby na wyłapanie błędów. Zaczynając używać rzutowań, narażasz się na wszelkiego rodzaju problemy. W rzeczywistości, każdy program, który zawiera wiele rzutowań, powinien być traktowany podejrzliwie, niezależnie od argumentów autora, że należało go napisać właśnie w ten sposób. Na ogół rzutowanie powinno być stosowane rzadko i ograniczone do rozwiązywania wyjątkowych problemów.

Kiedy stajesz oko w oko z programem pełnym błędów, dwiema głównymi podejrzanymi będą zapewne operacje rzutowania. W jaki jednak sposób odnajdziesz rzutowania, zapisane w stylu używanym w języku C? Są one zwykłymi nazwami typów ujętymi w nawiasy i gdy zaczniesz na nie „polować”, przekonasz się, że często niełatwo odróżnić je od reszty programu.

Standard języka C++ zawiera składnię jawnego rzutowania, umożliwiającą całkowitą rezygnację ze stylu używanego w C (oczywiście, nie można zabronić stosowania rzutowania w stylu C nie łamiąc zasad, ale autorzy kompilatorów mogliby bez trudu zaznaczyć użytkownikowi rzutowania realizowane w „starym” stylu). Składnia jawnego rzutowania umożliwia łatwe odnalezienie miejsc, w których je zastosowano, dzięki używanym przez nie nazwom:

static_cast	Rzutowania, które „dobrze się zachowują” i „stosunkowo dobrze się zachowują”, włączając w to operacje, które można przeprowadzić w ogóle bez użycia rzutowania (np. automatyczną konwersję typów).
const_cast	Rzutowania usuwające modyfikatory const i (lub) volatile .
reinterpret_cast	Rzutowanie z całkowitą zmianą znaczenia. Istotne jest, że bezpieczne wykorzystywanie takiego rzutowania wymaga ponownego rzutowania — z powrotem do pierwotnego typu. Typ, do którego dokonywane jest rzutowanie, jest zazwyczaj używany wyłącznie do operacji niskopoziomowych albo jakichś innych celów. Jest to najbardziej niebezpieczny rodzaj rzutowania.
dynamic_cast	Do bezpiecznego rzutowania „w dół” (ten rodzaj rzutowania zostanie opisany w rozdziale 5.).

Pierwsze trzy jawne rzutowania zostaną obszernie opisane w następnych podrozdziałach, natomiast ostatnie będzie zaprezentowane dopiero w rozdziale 15.

static_cast

Rzutowanie **static_cast** jest używane w przypadku wszelkich dobrze zdefiniowanych rzutowań. Obejmują one „bezpieczne” konwersje, które kompilator powinien dopuścić bez rzutowania, a także przekształcenia nieco mniej bezpieczne, ale dobrze określone. Rodzaje rzutowań, określonych przez **static_cast**, obejmują: typowe konwersje nie-wymagające rzutowania, przekształcenia zmniejszające rozmiar danych (z utratą informacji), wymuszenie rzutowania typu **void***, niejawne konwersje typów oraz statyczne poruszanie się w obrębie hierarchii klas (z uwagi na to, że klasy oraz ich dziedziczenie nie zostały do tej pory przedstawione, przypadek ten zostanie opisany w rozdziale 15.).

```
// C03:static_cast.cpp
void func(int) {}

int main() {
    int i = 0xffff; // Maksymalna dodatnia wartość = 32767
    long l;
    float f;
    // (1) Typowe konwersje bez użycia rzutowania:
    l = i;
    f = i;
    // Działają również:
    l = static_cast<long>(i);
    f = static_cast<float>(i);

    // (2) Konwersje zawężające:
    i = 1; // Możliwość utraty cyfr
    i = f; // Możliwość utraty informacji
    // Powiedzenie "wiem o tym" eliminuje ostrzeżenia:
    i = static_cast<int>(l);
    i = static_cast<int>(f);
    char c = static_cast<char>(i);

    // (3) Wymuszenie konwersji z typu void* :
    void* vp = &i;
    // Stary sposób powoduje niebezpieczną konwersję:
    float* fp = (float*)vp;
    // Nowy jest równie niebezpieczny:
    fp = static_cast<float*>(vp);

    // (4) Niejawne konwersje typów, dokonywane
    // zazwyczaj przez kompilator:
    double d = 0.0;
    int x = d; // Automatyczna konwersja typu
    x = static_cast<int>(d); // Bardziej jawną konwersją
    func(d); // Automatyczna konwersja typu
    func(static_cast<int>(d)); // Bardziej jawną konwersją
} //:-)
```

W sekcji (1) widoczne są sposoby konwersji stosowane w języku C — z zastosowaniem rzutowania oraz bez niego. Zmiana wartości **int** na **long** albo **float** nie jest problemem, ponieważ w tych przypadkach typy docelowe są zawsze w stanie przechować każdą wartość reprezentowaną przez **int**. Mimo że nie jest to konieczne, można użyć słowa kluczowego **static_cast**, sygnalizującego dokonane konwersje.

W sekcji (2) przedstawiono konwersje w przeciwnym kierunku. W ich przypadku istnieje możliwość utraty danych, ponieważ typ **int** nie jest tak „duży”, jak typy **long** czy **float** — nie jest bowiem w stanie przechowywać równie dużych liczb. Dlatego też są one nazywane *konwersjami zawężającymi*. Kompilator nadal jest w stanie je wykonać, ale najczęściej zgłosi w takich przypadkach ostrzeżenie. Można uniknąć wyświetlanego ostrzeżenia, wskazując za pomocą rzutowania, że właśnie taką operację zamierzało się wykonać.

W języku C++ (w odróżnieniu od C) przypisanie wartości typu **void*** nie jest możliwe bez użycia rzutowania. Jest ono niebezpieczne i w związku z tym wymaga, by programista zachował ostrożność. Użycie słowa kluczowego **static_cast** zapewnia, że gdy przyjdzie pora na szukanie błędów, miejsce, w którym dokonano rzutowania, będzie przynajmniej łatwiejsze do znalezienia niż w przypadku standardowego „starego” rzutowania.

Sekcja (4) programu prezentuje niejawne konwersje typów, dokonywane na ogół automatycznie przez kompilator. Ponieważ są one wykonywane automatycznie, nie wymagają rzutowania. Jednakże użycie **static_cast** umożliwia sygnalizację podejmowanych działań, ułatwiając ich zrozumienie lub późniejsze odszukanie.

const_cast

Jeżeli potrzebna jest konwersja z typu oznaczonego modyfikatorem **const** lub **volatile** do typu **nieposiadającego** takiego modyfikatora, należy użyć do tego celu słowa kluczowego **const_cast**. Są to *jedynie* rodzaje przekształceń, których można dokonać za pomocą rzutowania **const_cast** — jeżeli wykonywane są jakieś kolwiek inne konwersje, trzeba użyć do nich oddzielnego wyrażenia, bo w innym przypadku spowoduje to zgłoszenie błędu kompilacji.

```
//: C03:const_cast.cpp
int main() {
    const int i = 0;
    int* j = (int*)&i; // Postać niewskazana
    j = const_cast<int*>(&i); // Postać preferowana
    // Nie można wykonać równocześnie dodatkowego rzutowania:
    //! long* l = const_cast<long*>(&i); // Błąd
    volatile int k = 0;
    int* u = const_cast<int*>(&k);
} //:-
```

Podczas pobierania adresu obiektu typu **const** tworzony jest wskaźnik do stałej, którego nie można — bez rzutowania — przypisać do zmiennej niebędącej wskaźnikiem do stałej. Można to zrobić, używając „starego” stylu rzutowania, ale lepszym rozwiązaniem jest zastosowanie rzutowania **const_cast**. Ta sama uwaga odnosi się do modyfikatora **volatile**.

reinterpret_cast

Jest to najmniej bezpieczny mechanizm rzutowania, będący zarazem najbardziej prawdopodobną przyczyną błędów. Rzutowanie **reinterpret_cast** zakłada, że obiekt jest zbiorem bitów, który może być traktowany (dla jakichś niejasnych celów) tak, jakby był obiektem zupełnie innego typu. Jest to właśnie taka operacja niskopoziomowa, z jakich niechłubnie słynie język C. Właściwie zawsze istnieje konieczność ponownego rzutowania zmiennej do pierwotnego typu (albo przynajmniej traktowania jej tak, jakby była takiego typu), zanim wykona się **jakiekolwiek** inne działania.

```
//: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; };

void print(X* x) {
    for(int i = 0; i < sz; i++)
        cout << x->a[i] << ' ';
    cout << endl << "....." << endl;
}

int main() {
    X x;
    print(&x);
    int* xp = reinterpret_cast<int*>(&x);
    for(int* i = xp; i < xp + sz; i++)
        *i = 0;
    // Nie można w tym miejscu używać zmiennej xp tak.
    // jakby była typu X*, dopóki nie dokona się z powrotem
    // jej rzutowania;
    print(reinterpret_cast<X*>(xp));
    // W tym przykładzie można również użyć
    // oryginalnego identyfikatora:
    print(&x);
} ///:~
```

W powyższym, prostym przykładzie struktura X zawiera tablicę liczb całkowitych, ale gdy jest ona tworzona na stosie jako X x, każda ze znajdujących się w tej tablicy liczb zawiera śmieci (zostało to pokazane za pomocą użycia funkcji **print()**, wyświetlającej zawartość struktury). W celu zainicjowania tych wartości adres struktury jest rzutowany na wskaźnik do typu int, który następnie przechodzi przez całą tablicę, przypisując każdemu jej elementowi wartość 0. Zwróć uwagę na to, że górną granicą zmiennej i jest wyznaczana przez „dodanie” sz do xp — kompilator wie, że w rzeczywistości chodzi ci o sz wskazywanych pozycji powyżej xp i samodzielnie prowadza odpowiednie obliczenia na wskaźnikach.

Gdy stosuje się rzutowanie **reinterpret_cast**, uzyskuje się rezultat tak odmienny, że nie może on zostać użyty zgodnie z początkowym przeznaczeniem typu, dopóki nie dokona się jego rzutowania z powrotem na pierwotny typ. W powyższym przykładzie widoczne jest rzutowanie ponownie na typ X*, ale oczywiście, ponieważ nadal dysponujemy oryginalnym identyfikatorem, to możemy użyć go w tym przypadku. Jednak zmienna xp jest użyteczna wyłącznie jako int*, co stanowi rzeczywiście „**reinterpretację**” pierwotnego typu X.

Rzutowanie `reinterpret_cast` oznacza często niezalecany i (lub) nieprzenośny styl programowania, ale jest dostępne jeśli uważasz, że musisz go użyć.

sizeof – samotny operator

Operator `sizeof` jest jedynym operatorem w swoim rodzaju, ponieważ zaspokaja on niezwykłą potrzebę. Udziela informacji, dotyczącej wielkości pamięci przydzielonej jednostkom danych. Jak już wspomniano we wcześniejszej części rozdziału, `sizeof` informuje o liczbie bajtów, zajmowanych przez dowolną zmienną. Może on również podać wielkość typu danych (bez nazwy zmiennej):

```
//: C03:sizeof.cpp
#include <iostream>
using namespace std;
int main() {
    cout << "sizeof(double) = " << sizeof(double);
    cout << ". sizeof(char) = " << sizeof(char);
} //:~
```

Zgodnie z definicją, wielkość dowolnego typu znakowego `char` (`signed`, `unsigned` lub zwykłego) wynosi zawsze jeden, niezależnie od tego, czy pamięć zorganizowana jest oczywiście w taki sposób, że znaki przechowywane są w pojedynczych bajtach. Dla wszystkich pozostałych typów zwracaną wartością jest wielkość wyrażona w bajtach.

Warto zwrócić uwagę na to, że `sizeof` jest operatorem, a nie funkcją. Jeżeli zostanie on zastosowany do typu, należy posłużyć się widoczną powyżej notacją **nawiasową**, jednak w odniesieniu do zmiennych można go używać bez nawiasów:

```
//: C03:sizeofOperator.cpp
int main() {
    int x;
    int i = sizeof x;
} //:~
```

Operator `sizeof` może również podawać wielkość typów danych, zdefiniowanych przez użytkownika. Zostanie to wykorzystane w dalszej części książki.

Słowo kluczowe `asm`

Jest to proteza, umożliwiająca wstawienie do programu, napisanego w języku C++, kodu asemblerowego, obsługującego używany sprzęt. W programie asemblerowym istnieje często możliwość odwoływania się do zmiennych zdefiniowanych w C++. Ułatwia to komunikację z programem napisanym w C++, ograniczając zarazem konieczność używania języka asemblera do minimum — niezbędnego z uwagi na efektywność lub możliwość wykorzystania specjalnych instrukcji procesora. Szczegółowy opis składni używanej podczas pisania programu w asemblerze zależy od kompilatora; można go odnaleźć w dołączonej do niego dokumentacji.

Operatory dosłowne

Są to słowa kluczowe określające operacje bitowe i logiczne. Programistów, których klawiatury nie posiadają takich znaków, jak `&`, `|`, `^` itd., zmuszano w języku C do używania okropnych *trójznaków*, które były nie tylko irytujące przy wpisywaniu, ale również zupełnie niezrozumiałe w czasie czytania programu. W języku C++ usunięto tę niedogodność, wprowadzając dodatkowe słowa kluczowe:

słowo kluczowe	znaczenie
and	<code>&&</code> (iloczyn logiczny)
or	<code> </code> (suma logiczna)
not	<code>!</code> (negacja logiczna)
not_eq	<code>!=</code> (nierówność logiczna)
bitand	<code>&</code> (koniunkcja bitowa)
and_eq	<code>&=</code> (koniunkcja bitowa z przypisaniem)
bitor	<code> </code> (alternatywa bitowa)
or_eq	<code> =</code> (alternatywa bitowa z przypisaniem)
xor	<code>^</code> (bitowa różnica symetryczna)
xor_eq	<code>^=</code> (bitowa różnica symetryczna z przypisaniem)
compl	<code>~</code> (uzupełnienie do jedynki)

Jeżeli używany przez ciebie kompilator jest zgodny ze standardem C++, to będzie obsługiwał powyższe słowa kluczowe.

Tworzenie typów złożonych

Podstawowe typy danych oraz ich odmiany są niezbędne, ale raczej prymitywne. Języki C i C++ udostępniają narzędzia pozwalające na tworzenie na podstawie typów podstawowych bardziej skomplikowanych typów danych. Najważniejszym z nich jest struktura (`struct`), stanowiąca w języku C++ podstawę klasy (`class`). Jednakże najprostszym sposobem tworzenia bardziej złożonych typów jest przypisanie nazwie typu innej nazwy — za pomocą słowa kluczowego `typedef`.

Nadawanie typom nowych nazw za pomocą `typedef`

Nazwa słowa kluczowego `typedef` jest nieco myląca — sugeruje bowiem „definicję typu” (ang. *type definition*), podczas gdy bardziej stosownym, i oddającym istotę rzeczy, określeniem byłoby „nadanie nowej nazwy”. Jego składnia jest następująca:

`typedef opis-istniejącego-typu nowa-nazwa`

Słowa **kluczowego `typedef`** używa się często w sytuacjach, gdy typy danych stają się zbyt skomplikowane — po to, by uniknąć wpisywania długich nazw. Typowe użycie `typedef` ma następującą postać:

```
typedef unsigned long ulong;
```

A zatem gdy kompilator napotka w twoim programie typ **ulong**, będzie wiedział, że chodziło ci o **unsigned long**. Choć wydaje się, że to samo można łatwo osiągnąć za pomocą dyrektyw preprocessora, zdarzają się sytuacje, w których kompilator musi wiedzieć, że traktujesz nazwę w taki sposób, jakby była typem, i w związku z tym użycie **typedefjest** niezbędne.

Jednym z miejsc, w których wygodnie jest używać **typedef**, są typy wskaźnikowe. Jak już wspomniano, definicja:

```
int* x, y;
```

oznacza w rzeczywistości, że zmienna x jest typu **int***, a zmienna y — typu **int** (nie **int***). Oznacza to, że znak „*****” jest wiązany prawostronnie, a nie lewostronnie. Jeżeli jednak używa się słowa kluczowego **typedef**:

```
typedef int* IntPtr;  
IntPtr x, y;
```

to zarówno x, jak i y będą typu **int***.

Można twierdzić, że w przypadku prostych typów danych lepiej jest unikać stosowania **typedef**, dzięki czemu zachowana zostanie przejrzystość, a zatem również czytelność programu. Rzeczywiście, użycie wielu deklaracji **typedef** znacznie zmniejsza czytelność programu. Jednakże w języku C stosowanie **typedefest** szczególnie ważne w odniesieniu do struktur.

Łączenie zmiennych w struktury

Słowo kluczowe **struct** umożliwia połączenie grupy zmiennych w strukturę. Po utworzeniu struktury można zdefiniować wiele egzemplarzy zmiennych ustanowionego przez nią „nowego” typu danych. Na przykład:

```
//: C03:SimpleStruct.cpp  
struct Structure1 {  
    char c:  
    int i:  
    float f,  
    double d:  
};  
  
int main() {  
    struct Structure1 sl, s2;  
    sl.c = 'a': // Wybór elementu za pomocą ':'  
    sl.i - 1:  
    sl.f = 3.14;  
    sl.d = 0.00093;  
    s2.c = 'a';  
    s2.i = 1;  
    s2.f=3.14;  
    s2.d = 0.00093;  
} ///.~
```

Deklaracja **struct** musi kończyć się średnikiem. W funkcji **main()** utworzono dwa egzemplarze struktury **Structure1** — *s1* i *s2*. Każdy z nich posiada odrębne wersje elementów *c*, *i*, *f* i *d*. A więc *s1* i *s2* reprezentują dwie grupy zupełnie niezależnych od siebie zmiennych. Do wyboru poszczególnych elementów w obrębie *s1* i *s2* używany jest znak „.” — podobnie jak w przykładach stosowania obiektów klas w C++, zaprezentowanych w poprzednim rozdziale. Ponieważ klasy powstały ze struktur, więc wywodzi się z nich również ta składnia.

Zwraca uwagę niewygodny sposób używania struktury **Structure1** (jak się okaże, dotyczy to wyłącznie języka C, a nie C++). W języku C, podczas definiowania zmiennych, nie można napisać **Structure1** — trzeba natomiast dokonać zapisu: **struct Structure1**. Jest to przypadek, w którym w języku C szczególnie przydaje się deklaracja **typedef**:

```
//: C03:SimpleStruct2.cpp
// Użycie typedef w stosunku do struktury
typedef struct {
    char c;
    int i;
    float f;
    double d;
} Structure2;

int main() {
    Structure2 s1, s2;
    s1.c = 'a';
    s1.i = 1;
    s1.f = 3.14;
    s1.d = 0.00093;
    s2.c = 'a';
    s2.i = 1;
    s2.f = 3.14;
    s2.d = 0.00093;
} //:~
```

Używając w taki sposób deklaracji **typedef** (przynajmniej w C — w języku C++ spróbuj usunąć słowo **typedef**), można, podczas definiowania zmiennych *s1* i *s2*, udawać, że **Structure2** jest typem wbudowanym, podobnie jak **int** czy **float** (zawiera on tylko wyłącznie dane, czyli cechy, a nie zachowanie, co miałoby miejsce w C++ w przypadku prawdziwych obiektów). Nietrudno zauważyc, że na początku deklaracji pominięto identyfikator struktury, ponieważ jej celem było jedynie nadanie typowi nazwy. Może jednak zaistnieć potrzeba odwołania się do struktury z wnętrza jej definicji. W takich przypadkach można powtórzyć tę samą nazwę — zarówno jako nazwę struktury, jak i w deklaracji **typedef**:

```
//: C03:SelfReferential.cpp
// Struktura odwołująca się do siebie samej

typedef struct SelfReferential {
    int i;
    SelfReferential* sr; // Niesamowite, prawda?
} SelfReferential;
```

```
1int main() {  
2    SelfReferential srl, sr2;  
3    srl.sr = &sr2;  
4    sr2.sr = &srl;  
5    srl.i = 47;  
6    sr2.i = 1024;  
7} //:-
```

Łatwo zauważyc, że zmienne srl i sr2 wskazują na siebie nawzajem, a także każda z nich zawierają jakieś dane.

Nazwa zdefiniowana przez **struct** nie jest tą samą **nazwą**, która została utworzona przez **typedef**, ale zazwyczaj taki sposób jej użycia stanowi ułatwienie.

Wskaźniki i struktury

Wszystkimi strukturami, zawartymi w poprzednich przykładach, operuje się w taki sam sposób, jak obiektami. Jednak, podobnie jak w przypadku dowolnego obszaru pamięci, można również pobrać adres struktury (jak to pokazano w programie **SelfReferential.cpp**). Jak wskazano powyżej, do wyboru elementu obiektu, będącego strukturą, używa się znaku „.”. Jednak w przypadku wskaźnika do struktury, do wyboru elementu używa się operatora: „->”. Oto przykład:

```
//: C03:SimpleStruct3.cpp  
// Używanie wskaźników do struktur  
typedef struct Structure3 {  
    char c;  
    int i;  
    float f;  
    double d;  
} Structure3;  
  
int main() {  
    Structure3 sl, s2;  
    Structure3* sp = &sl;  
    sp->c = 'a';  
    sp->i = 1;  
    sp->f = 3.14;  
    sp->d = 0.00093;  
    sp = &s2; // Wskazuje na inną strukturę  
    sp->c = 'a';  
    sp->i = 1;  
    sp->f = 3.14;  
    sp->d = 0.00093;  
} //:-
```

W funkcji **main()** zmienna **sp**, będąca wskaźnikiem, wskazywała najpierw na strukturę **sl**, której składowe zostały zainicjowane za pomocą operatora „->” (tego samego operatora można użyć do odczytania ich wartości). Następnie **sp** wskazywała na strukturę **s2**, a jej składowe zostały zainicjowane w taki sam sposób, jak składowe **sl**. A zatem dodatkowa korzyść wynikająca z zastosowania wskaźników polega na tym, że ich wartości mogą być dynamicznie zmieniane, dzięki czemu wskazują one różne obiekty. Umożliwia to większą elastyczność w czasie pisania programu.

Oto wszystkie niezbędne informacje na temat struktur; dzięki lekturze dalszej części książki nabierzesz wprawy w ich używaniu (a szczególnie w stosowaniu ich bardziej efektywnych następców — klas).

Zwiększenie przejrzystości programów za pomocą wyliczeń

Wyliczeniowy typ danych umożliwia powiązanie nazw z liczbami, ułatwiając tym samym zrozumienie programu czytającej go osobie. Słowo kluczowe **enum** (pochodzące z języka C) automatycznie numeruje listę identyfikatorów, nadając im wartości 0, 1, 2 itd. Można również zadeklarować zmienne wyliczeniowe (które zawsze są reprezentowane przez liczby całkowite). Deklaracja typu wyliczeniowego przypomina deklarację struktury.

Wyliczeniowe typy danych są przydatne w przypadku zapamiętywania pewnego rodzaju cech:

```
//: C03:Enum.cpp

// Zapamiętywanie kształtów

enum ShapeType {
    circle,
    square,
    rectangle
}; // Musi kończyć się średnikiem, tak jak struct

int main() {
    ShapeType shape = circle;
    // Jakieś operacje...
    // A teraz działania zależne od kształtu:
    switch(shape) {
        case circle: /* obsługa okręgu */ break;
        case square: /* obsługa kwadratu */ break;
        case rectangle: /* obsługa prostokąta */ break;
    }
} //:-
```

Zmienna **shape** jest zmienną typu wyliczeniowego **ShapeType**, a jej wartość jest porównywana z wartościami znajdującymi się na liście elementów wyliczenia. Jednak ponieważ zmienna **shape** jest w rzeczywistości zmienną całkowitą, może ona przyjmować dowolne wartości typu **int** (w tym liczby ujemne). Z wartościami znajdującymi się na liście elementów wyliczenia mogą być również porównywane zmienne typu **int**.

Program wykorzystujący zaprezentowany w powyższym przykładzie sposób wyboru działania może sprawiać problemy. Język C++ umożliwia znacznie lepszy sposób wykonywania takich operacji, ale jego opis zostanie przedstawiony dopiero w dalszej części książki.

Jeżeli nie odpowiada ci sposób, w jaki kompilator przypisuje poszczególne wartości możesz zrobić to sam, tak jak w przykładzie poniżej:

```
enum ShapeType {  
    circle = 10, square = 20, rectangle = 50  
};
```

Jeżeli wartości zostaną nadane tylko niektórym nazwom, kompilator użyje w stosunku do pozostałych nazw kolejnych wartości całkowitych. Na przykład w przypadku deklaracji:

```
enum snap { crackle = 25, pop };  
kompilator nada elementowi pop wartość 26.
```

Nietrudno zauważyc, że program wykorzystujący typy wyliczeniowe jest znacznie bardziej przejrzysty. Jednak, w pewnej mierze, jest to jedynie próba (w języku C) osiągnięcia tego, co można uzyskać, wykorzystując klasy w języku C++. Dlatego też deklaracja **enum** jest w C++ używana rzadziej niż w języku C.

Kontrola typów w wyliczeniach

Wyliczenia są realizowane w języku C w dość prosty sposób — przypisują one wartości całkowite nazwom, nie dokonując przy tym żadnej kontroli typów. W języku C++, jak można się tego spodziewać na podstawie dotychczasowej wiedzy, pojęcie typu, podobnie zresztą jak wyliczeń, ma charakter podstawowy. Tworząc wyliczenie o określonej nazwie, w rzeczywistości generuje się nowy typ danych, tak samo jak w przypadku klas. Nazwa wyliczenia staje się słowem zastrzeżonym w obrębie bieżącej jednostki kompilacji.

Wyliczenia są w języku C++ objęte dokładniejszą kontrolą typów niż w C. Można to zauważyc na przykład w przypadku egzemplarza zmiennej **a**, będącej typu wyliczeniowego **color** (kolor). W języku C możliwy jest zapis **a++**, ale nie można zastosować go w C++. Wynika to z faktu, że inkrementacja zmiennej wyliczeniowej wymaga dokonania dwóch konwersji typów, z których jedną jest w języku C++ dopuszczalna, a druga — już nie. Wartość zmiennej wyliczeniowej jest najpierw rzutowana z typu **color** na typ **int**, następnie jest ona inkrementowana, a na koniec rzutowana z powrotem z typu **int** na typ **color**. W języku C++ to niedozwolone, ponieważ typ **color** jest zupełnie innym typem danych, niebędącym równoważnikiem typu **int**. Wydaje się to logicznie; skąd można bowiem, na przykład, wiedzieć, jaki kolor uzyska się w wyniku inkrementacji wartości **blue** (niebieski)? Jeżeli konieczna jest możliwość inkrementacji wartości typu **color**, to powinien on być klasą (posiadającą zdefiniowany operator inkrementacji), a nie wyliczeniem. Klasę taką można bowiem utworzyć w sposób zapewniający znacznie większe bezpieczeństwo. Podczas każdej próby niejawnej konwersji do typu wyliczeniowego kompilator będzie sygnalizował takie, rzeczywiście niebezpieczne, działanie.

Podobną, dodatkową kontrolę typów w języku C++ posiadają opisane poniżej unie.

Oszczędzanie pamięci za pomocą unii

Zdarza się czasami, że program obsługuje różne typy danych, używając do tego celu tych samych zmiennych. W takim przypadku istnieją dwa możliwe rozwiązania: utworzenie struktury, zawierającej te wszystkie typy danych, których przechowywanie może być potrzebne, albo wykorzystanie unii. Unia lokuje wszystkie swoje dane w tym samym miejscu — wyznacza wielkość pamięci niezbędną do przechowania największego umieszczonego w niej elementu. Wielkość ta staje się zarazem wielkością unii. Unie są używane w celu oszczędzania pamięci.

Każda wartość umieszczona w unii zajmuje miejsce rozpoczętym się zawsze na początku unii, ale obejmującym tylko niezbędną ilość pamięci. A zatem tworzy się w ten sposób „superzmienną”, zdolną do przechowywania dowolnej spośród zmiennych unii. Adresy wszystkich zmiennych, wchodzących w skład unii, są takie same (w przypadku klas lub struktur — różne).

Poniżej przedstawiono prosty przykład użycia unii. Przeprowadź eksperymenty z usuwaniem różnych elementów unii, obserwując, jaki będzie to miało wpływ na jej wielkość. Zwróć również uwagę na to, że nie ma sensu deklaracja więcej niż jednego egzemplarza każdego z typów danych, wchodzących w skład unii (chyba tylko po to, by odwoływać się do nich, używając różnych nazw).

```
//: C03:Un1on.cpp
// Wielkość unii i jej proste wykorzystanie
#include <iostream>
using namespace std;

union Packed { // Deklaracja podobna do klasy
    char i;
    short j;
    int k;
    long l;
    float f;
    double d;
    // Unia będzie miała wielkość typu double,
    // ponieważ jest to jej największy element
}; // Średnik kończy deklarację unii, podobnie jak struktury

int main(){
    cout << "sizeof(Packed) = "
        << sizeof(Packed) << endl;
    Packed x;
    x.i = 'c';
    cout << x.i << endl;
    x.d = 3.14159;
    cout << x.d << endl;
} //:-
```

Kompilator dokonuje odpowiedniego przypisania, zależnie od wybranej składowej unii.

Kompilator, po dokonaniu przypisania, pomija to, co dzieje się dalej z unią. W poniższym przykładzie można przypisać zmiennej x wartość zmennopozycyjną:

```
x.f = 2.222;
```

a następnie wyprowadzić je na wyjście w taki sposób, jakby była wartością całkowitą:

```
cout << x.1;
```

Spowoduje to pojawienie się na wyjściu śmieci.

Tablice

Tablice stanowią rodzaj typu złożonego, ponieważ pozwalają one połączyć ze sobą wiele zmiennych pod pojedynczą nazwą, w taki sposób, że będą one występować kolejno po sobie. Zapis:

```
int a[10];
```

tworzy miejsce w pamięci, przechowujące 10 zmiennych całkowitych, umieszczonych jedna za drugą, bez przypisywania odrębnej nazwy każdej z nich — wszystkie one używają natomiast pojedynczej nazwy a.

Aby uzyskać dostęp do *elementów tablicy*, należy zastosować tę samą składnię (wykorzystując nawiasy kwadratowe), która została użyta do zdefiniowania tablicy:

```
a[5] = 47;
```

Należy jednak pamiętać, że choć *rozmiarem* tablicy jest 10, indeksy jej elementów rozpoczynają się od zera (jest to czasami nazywane *indeksowaniem zerowym*). A zatem można odwoływać się tylko do elementów o numerach od 0 do 9, tak jak to pokazano w poniższym przykładzie:

```
//: C03:Arrayss.cpp
#include <iostream>
using namespace std;

int main() {
    int a[10];
    for(int i = 0; i < 10: i++) {
        a[i] - i * 10;
        cout << "a[" << i << "] = " << a[i] << endl;
    }
}
```

Dostęp do tablic jest wyjątkowo szybki. Jednakże nie ma tu żadnej „siatki zabezpieczającej”, chroniącej przed przekroczeniem indeksu końca tablicy — w takim przypadku wkracza się bowiem w obszar innych zmiennych. Inną niedogodnością jest wymóg zdefiniowania wielkości tablicy na etapie kompilacji — w przypadku konieczności zmiany rozmiaru tablicy w czasie pracy programu, nie można tego zrobić przy użyciu pokazanej powyżej składni (język C umożliwia dynamiczne tworzenie tablic, ale jest to rozwiązanie znacznie mniej eleganckie). Zaprezentowane w poprzednim rozdziale wektory, które można wykorzystać w C++, udostępniają obiekty podobne do tablic, automatycznie zmieniające swój rozmiar. Stanowią więc na ogół znacznie lepsze rozwiązanie w sytuacji, gdy wielkość tablicy nie może być z góry określona w czasie kompilacji.

Można tworzyć tablice dowolnego typu — nawet tablice struktur:

```
//: C03:StructArray.cpp
// Tablica struktur

typedef struct {
    int i, j, k;
} ThreePoint;

int main() {
    ThreePoint p[10];
    for(int i = 0; i < 10; i++) {
        p[i].i = i + 1;
        p[i].j = i + 2;
        p[i].k = i + 3;
    }
} //:~
```

Zwróć uwagę na to, że identyfikator *i*, będący składową struktury, jest niezależny od zmiennej *i*, stanowiącej licznik pętli **for**.

Aby przekonać się, że elementy tablicy są przechowywane w pamięci jeden za drugim, można wydrukować ich adresy, tak jak w przykładzie poniżej:

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std;

int main() {
    int a[10];
    cout << "sizeof(int) = " << sizeof(int) << endl;
    for(int i = 0; i < 10; i++)
        cout << "&a[" << i << "] = "
            << (long)&a[i] << endl;
} //:~
```

Po uruchomieniu programu można zauważyć, że każdy element tablicy znajduje się w odległości od poprzedniego elementu odpowiadającej wielkości liczby całkowitej (int). Oznacza to, że są one umieszczone po kolej w pamięci.

Wskaźniki i tablice

Identyfikatory tablic różnią się od identyfikatorów zwykłych zmiennych. Jednym z powodów jest to, że nie są one *l-wartościami* — nie można zatem przypisać im wartości. W istocie są one punktem zaczepienia składni, wykorzystującej nawiasy kwadratowe — podając nazwę tablicy bez nawiasów kwadratowych, uzyskuje się adres początku tablicy:

```
//: C03:ArrayIdentifier.cpp
#include <iostream>
using namespace std;

int main() {
    int a[10];
    cout << "a = " << a << endl;
    cout << "&a[0] = " << &a[0] << endl;
} //:~
```

Po uruchomieniu programu można zauważyc, że oba adresy (które zostaną wydrukowane szesnastkowo, nie występuje tu bowiem rzutowanie na typ **long**) są takie same.

A zatem identyfikator tablicy można traktować również jako wskaźnik dojej początku, dostępny wyłącznie do odczytu. I chociaż nie sposób zmienić wartości identyfikatora tablicy, by wskazywał jakieś inne miejsce, to *można* utworzyć inny wskaźnik, używając go do poruszania się w obrębie tablicy. W rzeczywistości notację wykorzystującą nawiasy kwadratowe można zastosować również w stosunku do zwykłych wskaźników:

```
//: C03:PointersAndBrackets.cpp
int main() {
    int a[10];
    int* ip = a;
    for(int i = 0; i < 10; i++)
        ip[i] = i * 10;
} //:-~
```

To, że nazwa tablicy zwraca adres jej początku, okazuje się dość istotne w przypadku przekazywania tablic funkcjom. Podczas deklaracji w charakterze argumentu funkcji tablicy jest deklarowany wskaźnik. A zatem, w poniższym przykładzie, listy argumentów funkcji **func1()** i **func2()** są w istocie takie same:

```
//: C03:ArrayArguments.cpp
#include <iostream>
#include <string>
using namespace std;

void func1(int a[], int size) {
    for(int i = 0; i < size; i++)
        a[i] = i * 1 - i;
}

void func2(int* a, int size) {
    for(int i = 0; i < size; i++)
        a[i] = i * i + i;
}

void print(int a[], string name, int size) {
    for(int i = 0; i < size; i++)
        cout << name << "[" << i << "] - "
            << a[i] << endl;
}

int main() {
    int a[5], b[5];
    // Tablice prawdopodobnie zawierają śmieci:
    print(a, "a", 5);
    print(b, "b", 5);
    // Inicjalizacja tablic:
    func1(a, 5);
    func1(b, 5);
    print(a, "a", 5);
    print(b, "b", 5);
    // Tablice są zawsze modyfikowane:
```

```
func2(a, 5);
func2(b, 5);
print(a. "a", 5);
print(b, "b". 5);
} //:~
```

Mimo że funkcje **func1()** i **func2()** różnią się deklaracjami swoich argumentów, sposób użycia argumentów w obrębie obu funkcji jest taki sam. W powyższym przykładzie ujawniły się jeszcze inne kwestie — tablice nie mogą być przekazywane przez wartość⁴, to znaczy nigdy nie jest tworzona lokalna kopia tablicy, która została przekazana funkcji. A zatem gdy podlega zmianie tablica, zawsze modyfikowany jest obiekt zewnętrzny. Jeżeli spodziewałeś się, że „zwykłe” argumenty powinny być przekazywane przez wartość, może to być na pierwszy rzut oka nieco mylące.

Zwróć uwagę na to, że funkcja **print()** używa nawiasów kwadratowych w stosunku do argumentów będących tablicami. Wprawdzie gdy przekazywane argumenty są tablicami, ujmowanie ich jako wskaźników jest praktycznie równoważne stosowaniu notacji wykorzystującej nawiasy kwadratowe, jednak używanie nawiasów informuje osobę czytającą program o tym, że argument jest traktowany jako tablica.

Warto również zauważyć, że w każdym przypadku funkcji przekazywany jest argument **size**. Przekazanie samego adresu tablicy nie jest wystarczającą informacją — funkcja musi zawsze znać wielkość tablicy — po to, aby nie przekroczyć jej zakresu.

Tablice, w tym tablice wskaźników, mogą być dowolnego typu. W rzeczywistości języki C i C++ posiadają specjalną listę argumentów funkcji **main()**, umożliwiającą przekazywanie programowi argumentów w wierszu poleceń. Lista ta ma następującą postać:

```
int main(int argc, char* argv[]) { // ... }
```

Pierwszy z argumentów jest liczbą elementów zawartych w tablicy, będącej drugim argumentem. Drugi argument jest zawsze tablicą elementów typu **char***, ponieważ argumenty podane w wierszu poleceń są przekazywane zawsze jako tablice znakowe (a, jak pamiętamy, tablice mogą być przekazywane wyłącznie w postaci wskaźników). Każda oddzielona odstępami grupa znaków znajdująca się w wierszu poleceń jest traktowana jako argument i zapisywana w oddzielnej tablicy. Poniższy program drukuje wszystkie argumenty, podane w wierszu poleceń przy jego uruchomieniu, przechodząc przez tablicę **argv**:

```
//: C03:CommandLineArgs.cpp
#include <iostream>
using namespace std;
```

⁴ Chyba że przyjmiesz dosłowną interpretację twierdzenia, że „wszystkie argumenty w językach C i C++ są przekazywane przez wartość, a »wartością« tablicy jest przekazywana przez jej identyfikator — czyli adres tablicy”. Z punktu widzenia języka asemblera może wydawać się to prawda, ale nie sądzę, by mogło ułatwić pracę z pojęciami wyższego poziomu. Dodanie w języku C++ referencji powoduje, że argument dotyczący „przekazywania wszystkiego przez wartość” wprowadza jeszcze większy bałagan — datego stopnia, że sądzę, iż lepiej jest myśleć o „przekazywaniu przez wartość”, jako przeciwieństwie „przekazywania adresu”.

```
int main(int argc, char* argv[]) {
    cout << "argc = " << argc << endl;
    for(int i = 0; i < argc; i++)
        cout << "argv[" << i << "] = "
            << argv[i] << endl;
} //:-
```

Jak zauważysz, **argv[0]** jest ścieżką dostępu i nazwą programu. Pozwala to programowi na uzyskanie informacji o sobie samym. Powoduje również dodanie jeszcze jednego elementu do tablicy argumentów, dlatego też typowym błędem przy pobieraniu argumentów wiersza poleceń jest użycie argumentu **argv[0]** w sytuacji, gdy potrzebny jest argument **argv[1]**.

Nie ma konieczności używania w funkcji **main()** identyfikatorów o nazwach **argc** oraz **argv** — ich nazwy wynikają tylko ze stosowanej konwencji (ale inni mogą być zdeorientowani, jeżeli zostaną wykorzystane odmienne nazwy). Istnieje również inny sposób zadeklarowania argumentu **argv**:

```
int main(int argc, char** argv) { // ...
```

Oba sposoby są równoważne, ale wersja używana w książce jest najbardziej intuicyjna i czytelna, ponieważ oznacza ona wprost: „jest to tablica wskaźników do znaków”.

Z wiersza poleceń można odczytać jedynie tablice znaków — jeżeli chcesz traktować argumenty w taki sposób, jakby były jakichś innych typów, musisz dokonać ich konwersji w programie. W standardowej bibliotece języka C istnieją funkcje, zadeklarowane w **<cstdlib>**, ułatwiające konwersję znaków na liczby. Najłatwiejszymi do użycia funkcjami są **atoi()**, **atol()** oraz **atof()**, dokonujące konwersji tablicy znaków ASCII odpowiednio do typów: **int**, **long** i zmiennopozycyjnego typu **double**. Poniżej znajduje się przykład z wykorzystaniem funkcji **atoi()** (dwie pozostałe funkcje wywołuje się w taki sam sposób):

```
//: C03:ArgsToInts.cpp
// Konwersja argumentów wiersza poleceń
// na liczby całkowite
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
    for(int i = 1; i < argc; i++)
        cout << atoi(argv[i]) << endl;
} //:-
```

Podeczas wywołania programu w wierszu poleceń można umieścić dowolną liczbę argumentów. Pętla **for** rozpoczyna się od wartości 1 — po to, by pominąć nazwę programu znajdująca się w **argv[0]**. Wpisanie w wierszu poleceń liczby zmiennopozycyjnej, zawierającej kropkę dziesiętną, spowoduje, że funkcja **atoi()** uwzględnii wyłącznie cyfry znajdujące się przed kropką. Jeżeli zostaną natomiast podane znaki niebędące cyframi, to funkcja **atoi()** zwróci wartość zero.

Format zmiennopozycyjny

Wprowadzona we wcześniejszej części rozdziału funkcja **printBinary()** jest wygodnym narzędziem, pozwalającym na penetrację wewnętrznej struktury różnych typów danych. Najbardziej interesującym z nich jest format zmiennopozycyjny, umożliwiający w językach C i C++ reprezentację zarówno bardzo dużych, jak i bardzo małych wartości, do czego wykorzystuje ograniczoną ilość pamięci. Mimo że nie sposób w niniejszym rozdziale przedstawić wszystkich szczegółów, warto wiedzieć, że bity w liczbach typów **float** i **double** są podzielone na trzy części: wykładnik, mantysę i bit znaku, a zatem przechowują one wartości wykorzystując notację wykładniczą. Poniższy program umożliwia zabawę, polegającą na drukowaniu dwójkowych wzorców liczb zmiennopozycyjnych. Pozwala ona na wyciągnięcie wniosków dotyczących formatu zmiennopozycyjnego, stosowanego przez kompilator (zazwyczaj jest to standard liczb zmiennopozycyjnych IEEE, ale używany przez ciebie kompilator może się do niego nie stosować):

```
//: C03:FloatingAsBinary.cpp
//{L} printBinary
//{T} 3.14159
#include "printBinary.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
    if(argc != 2) {
        cout << "Musisz podać liczbę" << endl;
        exit(1);
    }
    double d = atof(argv[1]);
    unsigned char* cp =
        reinterpret_cast<unsigned char*>(&d);
    for(int i = sizeof(double); i > 0 ; i -- 2) {
        printBinary(cp[i-1]);
        printBinary(cp[i]);
    }
}
```

III-

Najpierw program upewnia się, że został podany argument — sprawdzając wartość **argc**, która wynosi dwa, gdy podano jeden argument (jeśli nie podano w ogóle argumentów, wynosi ona jeden, ponieważ nazwa programu jest zawsze pierwszym elementem argv). Jeżeli wynik sprawdzenia jest niepomyślny, drukowany jest komunikat, a następnie wywoływana jest standardowa funkcja biblioteczna C, **exit()**, powodująca zakończenie programu.

Program pobiera argument z wiersza poleceń i za pomocą funkcji **atof()** przekształca znaki w liczbie typu **double**. Następnie liczba ta jest potraktowana jako tablica bajtów — jej adres jest rzutowany na typ **unsigned char***. Każdy z tych bajtów zostaje przekazany funkcji **printBinary()** w celu wydrukowania.

Przykład został przygotowany w taki sposób, by na moim komputerze bajty były drukowane poczynając od bitu znaku. Twój komputer może być inny, jest więc możliwe, że zechcesz zmienić kolejność wydruku. Warto również podkreślić, że format zmiennopozycyjny nie jest trywialny. Na przykład wykładnik oraz mantysa nie są zazwyczaj

wyrównane do granicy bajtów — rezerwuje się dla nich określoną liczbę bitów, które są następnie umieszczane w pamięci możliwie jak najbliżej. Aby naprawdę wiedzieć, co się dzieje, należy odgadnąć wielkość poszczególnych części liczby (bit znaku jest zawsze pojedynczym bitem, lecz wykładnik i manlysa mogą być różnych rozmiarów) i oddziennie wydrukować bity składające się na każdą z nich.

Arytmetyka wskaźników

Gdyby jedyną rzeczą możliwą do zrobienia ze wskaźnikiem wskazującym tablicę było traktowanie go w taki sposób, jakby był synonimem nazwy tej tablicy, wskaźniki do tablic nie zasługiwałyby na szczególną uwagę. Są one jednak bardziej elastyczne, ponieważ można zmienić ich wartość, tak by wskazywały jakieś inne miejsce (należy jednak pamiętać, że nie da się w taki sposób modyfikować identyfikatorów tablic).

Pojęcie *arytmetyki wskaźników* odnosi się do zastosowania wybranych operatorów matematycznych w stosunku do wskaźników. Arytmetyka wskaźników stanowi temat odrębny w stosunku do zwykłej arytmetyki, ponieważ aby wskaźniki zachowywały się prawidłowo, muszą podlegać pewnym ograniczeniom. Na przykład operatorem często używanym w stosunku do wskaźników jest `++`, który „zwiększa wskaźnik o jeden”. Naprawdę oznacza to, że wartość wskaźnika zostaje tak zmieniona, by przesunął się on do „następnej pozycji” (cokolwiek by to znaczyło). Oto przykład:

```
//: C03:PointerIncrement.cpp
#include <iostream>
using namespace std;

int main() {
    int i[10];
    double d[10];
    int* ip = i;
    double* dp = d;
    cout << "ip = " << (long)ip << endl;
    ip++;
    cout << "ip = " << (long)ip << endl;
    cout << "dp = " << (long)dp << endl;
    dp++;
    cout << "dp = " << (long)dp << endl;
} //:-
```

Program, uruchomiony na moim komputerze, daje następujące wyniki:

```
ip = 6684124
ip = 6684128
dp = 6684044
dp = 6684052
```

Mimo że operacja `++` przebiega tak samo zarówno w przypadku typu `int*`, jak i `double*`, można zauważyć, że wskazywany adres zmienił się tylko o 4 bajty dla wskaźnika typu `int*`, natomiast aż o 8 bajtów dla wskaźnika typu `double*`. Nie jest przypadkiem, że właśnie takie są rozmiary typów `int` i `double` w moim komputerze. I to jest właśnie sztuczka związana z arytmetyką wskaźników — kompilator wyznacza wielkość, o jaką powinien zmienić się wskaźnik tak, by wskazywał on następną pozycję tablicy (arytmetyka wskaźników ma znaczenie tylko w obrębie tablic). Działa to w taki sposób nawet w przypadku tablic struktur:

```
//: C03:PointerIncrement2.cpp
#include <iostream>
using namespace std;

typedef struct {
    char c;
    short s;
    int i;
    long l;
    float f;
    double d;
    long double ld;
} Primitives;

int main() {
    Primitives p[10];
    Primitives* pp = p;
    cout << "sizeof(Primitives) = "
        << sizeof(Primitives) << endl;
    cout << "pp - " << (long)pp << endl;
    pp++;
    cout << "pp = " << (long)pp << endl;
} //:~
```

Po uruchomieniu programu na moim komputerze uzyskałem następujące wyniki:

```
sizeof(Primitives) = 40
pp = 6683764
pp = 6683804
```

A zatem kompilator obsługuje odpowiednio wskaźniki do struktur (a także wskaźniki do klas i unii).

Arytmetyka wskaźników odnosi się również do operatorów `--`, `+` i `-`, lecz dwa ostatnie mają ograniczony zakres. Nie można dodać do siebie dwóch wskaźników, a w przypadku odjęcia od siebie wskaźników uzyskuje się w wyniku liczbę znajdujących się pomiędzy nimi elementów. Można jednak dodawać i odejmować od wskaźników wartości całkowite. Poniżej znajduje się przykład ilustrujący arytmetykę wskaźników:

```
//: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;

#define P(EX) cout << #EX << ":" << EX << endl;

int main() {
    int a[10];
    for(int i = 0; i < 10; i++)
        a[i] = i; // Przypisz wartości indeksów
    int* ip = a:
    P(*ip):
    P(*++ip):
    P(*(ip + 5)):
    int* ip2 = ip + 5;
    P(*ip2):
    P(*(ip2 - 4)):
    P(*--ip2):
    P(ip2 - ip); // Wynikiem jest liczba elementów
} //:~
```

Program rozpoczyna się jeszcze jedną makroinstrukcją, ale wykorzystuje ona cechę preprocessora nazywaną **łańcuchowaniem** (zaimplementowaną w postaci znaku „#”, poprzedzającego wyrażenie), która zmienia dowolne wyrażenie w łańcuch znaków. Jest to bardzo wygodne, pozwala bowiem na wydrukowanie wyrażenia, po którym następuje przecinek, a następnie wartość tego wyrażenia. Sposób wykorzystania tego wygodnego skrótu widoczny jest w funkcji **main()**.

Mimo że w stosunku do wskaźników poprawne są zarówno przedrostkowe, jak i przyrostkowe wersje operatorów ++ i --, w przykładzie użyto wyłącznie ich wersji przedrostkowych. W powyższych wyrażeniach operacje są bowiem wykonywane jeszcze przed wyłuskaniem wskaźników, dzięki czemu możliwa jest obserwacja efektu działania operatorów. Zwróci uwagę, że dodawane i odejmowane od wskaźników są wyłącznie wartości całkowite — kompilator nie pozwoli na dodawanie lub odejmowanie od siebie dwóch wskaźników.

A oto wyniki działania powyższego programu:

```
*ip: 0  
*++ip: 1  
*(ip + 5): 6  
*ip2: 6  
*(ip2 - 4): 2  
*-ip2: 5  
ip2 - ip: 4
```

We wszystkich przypadkach, dzięki arytmetyce wskaźników, wskaźnik jest zmieniany w taki sposób, by wskazywał „właściwego miejsce”, w zależności od rozmiaru elementów.

Nie przejmuj się, jeżeli arytmetyka wskaźników wydaje ci się na pierwszy rzut oka nieco przytłaczająca. Najczęściej jedynie tworzy się tablice i indeksuje je przy użyciu nawiasów kwadratowych, a najbardziej wyrafinowanymi operacjami wykonywanymi na wskaźnikach są ++ i --. Arytmetyka wskaźników jest zarezerwowana dla bardziej złożonych i wyrafinowanych programów — wiele spośród kontenerów, znajdujących się w standardowej bibliotece C++, ukrywa przed tobą większość ze swoich przemyślnie zrealizowanych szczegółów, dzięki czemu nie musisz się o nie martwić.

Wskazówki dotyczące Uruchamiania programów

Idealne środowisko programistyczne zawierałoby wspaniałego program uruchomieniowy (ang. *debugger*), dzięki któremu działanie programu stawałoby się zupełnie przejrzyste, co z kolei pozwoliłoby na szybkie znalezienie znajdujących się w nim błędów. Niestety, większość dostępnych programów uruchomieniowych ma jakieś słabe strony i wymaga umieszczenia w programie fragmentów kodu ułatwiających pracę programiście. Można również pracować w środowisku (na przykład takim, jak systemy wbudowane, w których latami nabywałem doświadczenia), w którym nie są dostępne

żadne programy uruchomieniowe, a możliwości komunikacji zwrotnej programu z użytkownikiem są nader ograniczone (sprowadzające się, na przykład, do wykorzystania jednowierszowego wyświetlacza LED). W takich przypadkach przydaje się pomysłowość, dotycząca sposobów wyszukiwania i wyświetlania informacji o przebiegu wykonania programu. W niniejszym rozdziale opisano niektóre przydatne w tym względzie techniki.

Znaczniki uruchomieniowe

Wprowadzenie kodu uruchomieniowego na stałe do programu może spowodować problemy. Zaczyna pojawiać się zbyt wiele informacji, co sprawia trudności z wyodrębnieniem z nich błędów. Kiedy wszystko wskazuje na to, że został już znaleziony błąd, rozpoczyna się wyrzucanie z programu kodu uruchomieniowego — po to, by później dojść do wniosku, że trzeba umieścić go tam z powrotem. Można rozwiązać te problemy, używając dwóch rodzajów znaczników — znaczników uruchomieniowych preprocesora oraz znaczników sprawdzanych w czasie pracy programu.

Znaczniki uruchomieniowe preprocesora

Wykorzystując dyrektywę preprocesora `#define` do zdefiniowania jednego lub większej liczby znaczników (najlepiej w pliku nagłówkowym), można następnie testować go za pomocą dyrektywy `#ifdef`, warunkowo włączając do programu kod uruchomieniowy. Kiedy uruchamianie programu wydaje się już zakończone, można po prostu unieważnić definicję znacznika dyrektywą `#undef`, co powoduje automatyczne usunięcie kodu uruchomieniowego (a zarazem zmniejszenie narzutu związanego z jego wykonywaniem oraz wielkości programu).

Warto określić nazwy znaczników uruchomieniowych jeszcze przed przystąpieniem do realizacji projektu, co pozwoli na zachowanie ich spójności. Znaczniki preprocesora sązwyczajowo zapisywane wielkimi literami, co odróżnia je od zmiennych. Często używaną nazwą znacznika jest po prostu **DEBUG** (ale należy uważać, by nie użyć nazwy **NDEBUG**, będącej nazwą zastrzeżoną w języku C). Kolejność użytych dyrektyw może być następująca:

```
#define DEBUG // Prawdopodobnie w pliku nagłówkowym
//...
#ifndef DEBUG // Sprawdzenie, czy zdefiniowano znacznik
/* kod uruchomieniowy */
#endif // DEBUG
```

Większość implementacji języków C i C++ pozwala również na definiowanie i unieważnianie znaczników z poziomu wiersza poleceń kompilatora, dzięki czemu można dokonać powtórnej komplikacji kodu i umieszczenia w nim informacji uruchomieniowej za pomocą jednego polecenia (najlepiej używając do tego **makefile** — narzędzia, które zostanie krótko opisane). Szczegóły na ten temat można znaleźć w dokumentacji kompilatora.

Znaczniki uruchomieniowe sprawdzanie w czasie pracy programu

W pewnych przypadkach wygodniej jest wyłączać i włączać znaczniki uruchomieniowe w czasie pracy programu, w szczególności ustawiając je w momencie uruchamiania w wierszu poleceń. Powtórna komplikacja programu tylko po to, aby wstawić do niego kod uruchomieniowy, jest natomiast nużąca.

Aby dynamicznie włączać i wyłączać kod uruchomieniowy, należy utworzyć znaczniki typu **bool**:

```
//: C03:DynamicDebugFlags.cpp
#include <iostream>
#include <string>
using namespace std;
// Znaczniki uruchomieniowe nie muszą
// być koniecznie globalne:
bool debug = false;

int main(int argc, char* argv[]) {
    for(int i = 0; i < argc; i++)
        if(string(argv[i]) == "--debug=on")
            debug = true;
    bool go = true;
    while(go) {
        if(debug) {
            // Kod uruchomieniowy
            cout << "Program uruchomieniowy włączony!" << endl;
        } else {
            cout << "Program uruchomieniowy wyłączony." << endl;
        }
        cout << "Zmień stan programu [włącz/wyłącz/wyjdź]: ";
        string reply;
        cin >> reply;
        if(reply == "włącz") debug = true; // Włącz go
        if(reply == "wyłącz") debug = false; // Wyłącz
        if(reply == "wyjdź") break; // Wyjście z pętli 'while'
    }
} //:-)
```

Program pozwala na włączanie i wyłączanie znacznika uruchomieniowego, aż do momentu, kiedy zostanie wpisane polecenie „wyjdź”, powodujące opuszczenie programu. Zwróć uwagę na to, że program wymaga wpisywania całych słów, a nie pojedynczych liter (możesz go jednak zmienić w taki sposób, by przyjmował litery). Do włączania kodu uruchomieniowego na początku programu można również użyć opcji podanej w wierszu poleceń — może ona wystąpić w dowolnym miejscu wiersza poleceń, ponieważ kod znajdujący się na początku funkcji **main()** przegląda wszystkie argumenty. Dzięki wyrażeniu:

```
string(argv[i])
```

sprawdzenie jej obecności jest dość łatwe do przeprowadzenia. Pobiera ono tablicę znakową **argv[i]** i tworzy łańcuch. Powyższy program poszukuje całego łańcucha **--debug=on**. Można również szukać łańcucha **--debug=**, a następnie sprawdzać, co znajduje się w jego dalszej części, udostępniając w ten sposób więcej opcji. Drugi tom książki (dostępny w witrynie <http://helion.pl/online/thinking/index.html>) zawiera rozdział poświęcony klasie **string**, należącej do standardu C++.

Mimo że znaczniki uruchomieniowe stanowią jeden z nielicznych przypadków, w których jest uzasadnione użycie zmiennych globalnych, to niekoniecznie musi tak być. Zwróć również uwagę na to, że zmienna **debug** została zapisana małymi literami, co przypomina osobie czytającej program, że niejest ona znacznikiem preprocesora.

Przekształcanie zmiennych i wyrażeń w teńcuchy

Podczas tworzenia kodu uruchomieniowego niewygodne jest wpisywanie wyrażeń drukujących, składających się z tablic znakowych (zawierających nazwy zmiennych), a następnie samych zmiennych. Na szczęście standard języka C zawiera operator *łańcuchowania* „#”, który wystąpił już we wcześniejszej części rozdziału. Umieszczenie znaku # przed argumentem makroinstrukcji preprocesora powoduje przekształcenie tego argumentu w tablicę znakową. Skojarzenie tego z faktem, że tablice znakowe, pomiędzy którymi nie występują znaki przestankowe, są łączone w pojedynczą tablicę znakową, pozwala na napisanie bardzo wygodnej makroinstrukcji, drukującej wartość zmiennej w trakcie uruchamiania programu:

```
#define PR(x) cout << #x " = " << x << "\n";
```

Wywołanie makroinstrukcji **PR(a)** w celu wydrukowania wartości zmiennej a przyniesie taki sam rezultat, jak kod:

```
cout << "a = " << a << "\n";
```

To samo dotyczy całych wyrażeń. Poniższy program wykorzystuje makroinstrukcję w celu utworzenia skrótu drukującego wyrażenie zamienione w tablicę znakową, a następnie wyznaczającego i drukującego wartość tego wyrażenia:

```
//: C03:StringizingExpressions.cpp
#include <iostream>
using namespace std;

#define P(A) cout << #A << ":" << (A) << endl;

int main() {
    int a = 1, b = 2, c = 3;
    P(a); P(b); P(c);
    P(a + b);
    P((c - a)/b);
} //:-
```

Nietrudno zauważyć, w jaki sposób techniki mogą szybko stać się niezastąpione, szczególnie gdy nie dysponujesz programem uruchomieniowym (lub musisz korzystać z wielu środowisk projektowych). Możesz użyć również dyrektywy **#ifdef**, dzięki której makroinstrukcja **P(A)** zostanie zdefiniowana jako „nic”, gdy zamierzasz usunąć kod uruchomieniowy.

Makroinstrukcja assert() języka C

W standardowym pliku nagłówkowym **<cassert>** znajduje się wygodna makroinstrukcja uruchomieniowa **assert()**. Używając makroinstrukcji **assert()**, należy podać

jej argument, będący wyrażeniem, które „deklaruje się jako prawdziwe”. Preprocesor generuje kod, weryfikujący prawdziwość wyrażenia. Jeżeli nie jest ono prawdziwe, program zatrzyma się, wytwarzając komunikat o błędzie, informujący o postaci wyrażenia oraz o tym, że niejest ono prawdziwe. Prezentuje to poniższy, prosty przykład:

```
//: C03:Assert.cpp
// Użycie makroinstrukcji uruchomieniowej assert()
#include <cassert> // Zawiera makroinstrukcje
using namespace std;

int main() {
    int i = 100;
    assert(i != 100); // Niepowodzenie
} //:-~
```

Makroinstrukcja została napisana w standardowym C, jest więc również dostępna w pliku nagłówkowym **assert.h**.

Po zakończeniu uruchamiania programu można usunąć kod tworzony przez makroinstrukcję, umieszczając w programie, przed włączeniem pliku **<cassert>**, wiersz:

```
#define NDEBUG
```

lub definiując znacznik **NDEBUG** w wierszu poleceń kompilatora. **NDEBUG** jest znacznikiem używanym w pliku **<cassert>** do zmiany sposobu, w jaki generowany jest kod makroinstrukcji.

W dalszej części książki zostaną przedstawione bardziej wyrafinowane rozwiązania, alternatywne w stosunku do makroinstrukcji **assert()**.

Adresy funkcji

Funkcja skompilowana i załadowana do komputera w celu wykonania zajmuje pewien obszar pamięci. Obszar ten, a więc również funkcja, posiada jakiś adres.

Język C nigdy nie był językiem uniemożliwiającym programiście podejmowania potencjalnie niebezpiecznych działań. Adresów zmiennych można używać za pomocą wskaźników, podobnie jak w przypadku adresów zmiennych. Deklaracja i sposób wykorzystywania wskaźników do funkcji wyglądają na pierwszy rzut oka trochę nieprzejrzyście, ale są one zgodne z konwencją stosowaną w innych konstrukcjach języka.

Definicja wskaźnika do funkcji

Aby zdefiniować wskaźnik do funkcji, która nie posiada argumentów i nie zwraca wartości, należy napisać:

```
void (*funcPtr)();
```

W przypadku tak skomplikowanej definicji najlepiej jest rozpocząć jej analizę od środka, a następnie przesuwać się stopniowo na zewnątrz. „Rozpoczęcie od środka” oznacza rozpoczęcie od nazwy zmiennej, która jest funcPtr. „Przesuwanie się na zewnątrz” to natomiast poszukanie najbliższego obiektu po prawej stronie (w tym przypadku nie ma tam niczego — poszukiwania kończą się bowiem szybko nawiasem zamykającym), a następnie po lewej stronie (gdzie znajduje się gwiazdka, oznaczająca wskaźnik), później znów po prawej (gdzie znajduje się pusta lista argumentów, określająca funkcję nie przyjmującą żadnych argumentów), a następnie znowu po lewej (słowo kluczowe void, oznaczające, że funkcja nie **zwraca żadnej** wartości). Tego typu przeglądanie deklaracji z lewa na prawo — tam i z powrotem — sprawdza się w większości przypadków.

Podsumujmy: „rozpoczynamy od środka” („**funcPtr** jest...”), przesuwamy się w prawo (nic tam nie ma — zatrzymujemy się na nawiasie zamykającym), przesuwamy się w lewo i napotykamy „*” (...wskaźnikiem do...”), przesuwamy się w prawo i znajdujemy pustą listę argumentów (...funkcji, która nie przyjmuje argumentów...”), przesuwamy się w lewo i napotykamy słowo void („funcPtr jest wskaźnikiem do funkcji, która nie przyjmuje argumentów i zwraca wartość void”).

Być może dziwisz się, dlaczego ***funcPrt** znajduje się w nawiasie. Gdyby go nie było, kompilator zobaczyłby tekst:

```
void *funcPtr();
```

Zamiast zmiennej, zadeklarowano by funkcję (zwracającą wartość **void***). Można sobie wyobrazić, że kompilator, próbując określić, jakiego rodzaju jest dana deklaracja lub definicja, przebywa taką samą drogą, jak opisana **powyżej**. Nawias jest mu potrzebny do tego, by „odbić się w przeciwnym kierunku”, i przesuwając się w lewo znaleźć „*” zamiast, poruszając się dalej w prawo, napotkać pustą listę argumentów.

Skomplikowane deklaracje i definicje

Zapoznawszy się ze składnią deklaracji w językach C i C++, można tworzyć bardziej skomplikowane definicje. Na przykład takie:

```
//: C03:ComplicatedDefinitions.cpp

/* 1. */ void * (*fp1)(int)[10];
/* 2. */ float (*(*fp2)(int,int,float))(int);
/* 3. */ typedef double (*(*(*fp3)())[10])();
      fp3 a;
/* 4. */ int (*(*f4())[10]))();

int main() {} //:~
```

Aby ją zrozumieć, przejdź przez każdą z nich, przesuwając się raz w prawo, raz w lewo. Pierwsza definicja głosi, że „**fp1**” jest wskaźnikiem do funkcji przyjmującej całkowity argument i zwracającej wskaźnik do tablicy, zawierającej 10 wskaźników do typu void”.

Według drugiej definicji: „**fp2** jest wskaźnikiem do funkcji, przyjmującej trzy argumenty (typów **int**, **int** i **float**) i zwracającej wskaźnik do funkcji, przyjmującej argument całkowity i zwracającej wartość typu **float**”.

Tworząc dużą liczbę definicji, można użyć deklaracji **typedef**. Trzeci przykład pokazuje, w jaki sposób użycie **typedef** może oszczędzić konieczności wpisywania za każdym razem skomplikowanych opisów. Oznacza on: „**fp3** jest wskaźnikiem do funkcji nieprzymierającej argumentów i zwracającej wskaźnik do tablicy 10 wskaźników do **funkcji**, które nie przyjmują argumentów i zwracają wartości typu **double**”. Następnie określa, że „zmienna **a** jest typu **fp3**”. Deklaracja **typedef** jest na ogół przydatna do tworzenia skomplikowanych opisów, na podstawie prostych.

Czwarty przykład nie jest definicją zmiennej, lecz deklaracją funkcji. Oznacza ona: „**f4** jest funkcją zwracającą wskaźnik do tablicy 10 wskaźników do funkcji, zwracających wartości całkowite”.

Rzadko (o ile w ogóle) są potrzebne równie skomplikowane deklaracje i definicje, jak te przedstawione powyżej. Jeżeli jednak wykonasz ćwiczenie, mające na celu ich zrozumienie, bez trudu poradzisz sobie z prostszymi przypadkami.

Wykorzystywanie wskaźników do funkcji

Po zdefiniowaniu wskaźnika do funkcji, jeszcze przed jego użyciem, trzeba przypisać mu adres. Podobnie jak adres tablicy **arr[10]** jest zwracany przez jej nazwę, podaną bez nawiasu kwadratowego (**arr**), adres funkcji **func()** uzyskuje się za pomocą nazwy funkcji, pozbawionej listy argumentów (**func**). Można również użyć w tym celu bardziej jednoznacznej składni: **&func()**. Aby wywołać funkcję, trzeba dokonać operacji wyłuskania wskaźnika, w taki sam sposób, w jaki został on zadeklarowany (należy pamiętać, że w językach C i C++ definicje sązawsze zbliżone do sposobu, w jaki używane są definiowane obiekty). Poniższy przykład przedstawia sposób zdefiniowania i późniejszego użycia wskaźnika do funkcji:

```
//: C03:PointerToFunction.cpp
// Definicja i wykorzystanie wskaźnika do funkcji
#include <iostream>
using namespace std;

void func() {
    cout << "func() wywołana..." << endl;
}

int main() {
    void (*fp)(); // Definicja wskaźnika do funkcji
    fp = func; // Inicjalizacja wskaźnika
    (*fp)(); // Wyłuskanie powoduje wywołanie funkcji
    void (*fp2)()= func; // Definicja i inicjalizacja
    (*fp2)();
} //
```

Po zdefiniowaniu **fp** jako wskaźnika do funkcji, za pomocą instrukcji **fp = func** (zwróć uwagę na to, że nazwa funkcji występuje tu bez listy argumentów) zostaje mu przypisany adres funkcji **func()**. Drugi przypadek prezentuje równoczesną definicję i inicjalizację.

Tablice wskaźników do funkcji

Jedną z najciekawszych konstrukcji, jakie można utworzyć, są tablice wskaźników do funkcji. Aby wybrać funkcję, wystarczy wybrać indeks tablicy i dokonać wyłuskania wskaźnika. Odpowiada to koncepcji *programu sterowanego tabelą* (ang. *table-driven code*) — zamiast instrukcji warunkowych lub instrukcji wyboru, na podstawie zmiennej stanu (lub kombinacji wielu zmiennych stanu) określa się funkcję, która ma zostać wykonana. Taki sposób projektowania może być przydatny, gdy często dodaje się lub usuwa funkcje zawarte w tabeli (albo zamierza tworzyć lub zmieniać taką tabelę dynamicznie).

Następny przykład tworzy kilka „ślepych” funkcji, używając do tego makroinstrukcji preprocesora, a następnie przygotowuje tablicę wskaźników do tych funkcji, wykorzystując automatyczną inicjalizację agregatową. A zatem można łatwo dodawać i usuwać funkcje znajdujące się w tabeli (czyli zmieniać tym samym funkcjonowanie programu), modyfikując tylko niewielkączęść kodu:

```
//: C03:FunctionTable.cpp
// Wykorzystanie tablicy wskaźników do funkcji
#include <iostream>
using namespace std;

// Makroinstrukcja, definiująca "ślepe" funkcje:
#define DF(N) void NO { \
    cout << "funkcja " #N " wywołana..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

void (*func_table[])()= { a, b, c, d, e, f, g };

int main() {
    while(1) {
        cout << "nacisnij klawisz od 'a' do 'g' " \
            "albo w, aby wyjść z programu" << endl;
        char c, cr;
        cin.get(c); cin.get(cr); // drugi dla CR
        if (c == 'w')
            break; // ... wyjście z while(1)
        if (c < 'a' || c > 'g')
            continue;
        (*func_table[c - 'a'])();
    }
} //:-
```

Być może zdajesz sobie teraz sprawę, jak bardzo przydatna może być ta technika do tworzenia niektórych rodzajów interpreterów lub programów przetwarzających listy.

Make — zarządzanie rozłączną kompilacją

W trakcie *rozłącznej kompilacji* (po podzieleniu kodu na pewną liczbę jednostek translacji) potrzebny jest jakiś sposób, umożliwiający automatyczną kompilację każdego pliku i informujący program łączący, by scalił wszystkie fragmenty — w tym odpowiednie biblioteki i kod inicjujący — w program wykonywalny. W przypadku większości kompilatorów można tego dokonać, używając pojedynczej instrukcji wiersza poleceń. Dla kompilatora GNU C++ można na przykład napisać:

```
g++ PlikZrodlowy1.cpp PlikZrodlowy 2.cpp
```

Problemem wynikającym z takiego podejścia jest to, że kompilator skompiluje najpierw oddzielnie każdy z plików, niezależnie od tego, czy *wymaga* on powtarnej kompilacji, czy też nie. Jeśli projekt składa się z wielu plików, konieczność ponownej kompilacji wszystkiego w sytuacji, gdy zmienił się tylko jeden z plików, może okazać się zbyt kosztowna.

Rozwiązaniem tego problemu, opracowanym w systemie Unix, ale dostępnym w takię czy innej postaci w każdym systemie, jest program o nazwie **make**. Zarządza on wszystkimi plikami wchodząymi w skład projektu, zgodnie z instrukcjami zawartymi w pliku tekstowym o nazwie **makefile**. Po dokonaniu zmian w niektórych plikach tworzących projekt i wpisaniu polecenia **make**, program ten wykorzystuje wskazówki zawarte w pliku **makefile**, porównując daty plików źródłowych z odpowiednimi plikami docelowymi. W przypadku gdy data pliku źródłowego jest późniejsza niż data pliku docelowego, wywołuje on dla tego pliku źródłowego kompilator. Program **make** ponownie kompiluje tylko te pliki źródłowe, które uległy zmianie, oraz pliki źródłowe, na które miały wpływ dokonane zmiany. Dzięki programowi **make** nie ma potrzeby powtarzanej kompilacji wszystkich plików wchodzących w skład projektu, ilekroć dokonane zostaną jakieś zmiany, ani też sprawdzania, czy wszystko zostało wykonane poprawnie. Plik **makefile** zawiera wszystkie polecenia niezbędne do złożenia projektu w całość. Jeżeli nauczysz się używania polecenia **make**, pozwoli ci to na zaoszczędzenie mnóstwa czasu i problemów. Zauważysz również, że program **make** jest zazwyczaj używany do instalacji nowych programów w komputerach pracujących pod kontrolą systemów operacyjnych Linux i Unix (choć ich pliki **makefile** są na ogół znacznie bardziej złożone niż te prezentowane w książce, a ponadto często w ramach procesu instalacji pliki te mogą zostać automatycznie utworzone dla danego komputera).

Ponieważ program **make** jest dostępny w takiej czy innej postaci dla każdego kompilatora C++ (a nawet gdy nie jest, to z każdym kompilatorem można użyć jego bezpłatnie dostępnych wersji), będzie on narzędziem używanym przeze mnie w całej książce. Producenci kompilatorów opracowali jednak również własne narzędzia, służące do tworzenia projektów. Narzędzia te pytają programistę o to, które pliki wchodzą w skład projektu, a następnie same określają wzajemne relacje pomiędzy nimi. Używają one plików podobnych do **makefile**, nazywanych na ogół *plikami projektowymi*, które są jednak nadzorowane przez środowisko programistyczne. Sposób konfiguracji i wykorzystywania plików projektów zależy od środowiska projektowego.

dla tego informacji o ich używaniu należy poszukać w odpowiedniej dokumentacji (choćż narzędzia służące do obsługi plików projektów, dostarczane przez producentów kompilatorów, są zazwyczaj tak proste, że można nauczyć się ich moją ulubioną metodą nauki — prób i błędów).

Pliki **makefile**, wykorzystywane w książce, powinny działać również wtedy, gdy używasz jakiegoś specyficznego narzędzia służącego do tworzenia projektów, dostarczonego przez producenta kompilatora.

Działanie programu make

Po wpisaniu polecenia **make** (albo innej nazwy, pod którą dostępne jest to narzędzie) program poszukuje w bieżącym katalogu pliku o nazwie **makefile**, który został utworzony przez autora projektu. Plik ten zawiera listę zależności pomiędzy plikami źródłowymi. Program **make** sprawdza daty modyfikacji plików. Jeżeli plik zależny posiada wcześniejszą datę niż plik, od którego on zależy, program **make** wykonuje *regułę*, podaną po tej zależności.

Wszystkie komentarze znajdujące się w plikach **makefile** rozpoczynają się znakiem `#` i rozciągają się aż do końca wiersza.

W prostym przypadku zawartość pliku **makefile** programu o nazwie „hello” może wyglądać następująco:

```
f Komentarz  
hello.exe: hello.cpp  
    mojkompilator hello.cpp
```

Oznacza to, że plik **hello.exe** (celowy) jest zależny od pliku **hello.cpp**. W przypadku gdy plik **hello.cpp** ma późniejszą datę modyfikacji niż plik **hello.exe**, program **make** wykonuje „regułę” **mojkompilator hello.cpp**. Plik **makefile** może zawierać liczne zależności i reguły. Wiele programów **make** wymaga, by wszystkie reguły rozpoczynały się tabulacją. W przypadku innych, odstępów są generalnie pomijane, można więc sformatować plik w taki sposób, by był on czytelny.

Reguły nie są ograniczone wyłącznie do wywołania kompilatora — program **make** może wywołać dowolny program. Tworząc grupy wzajemnie powiązanych zależności i reguł, można zmodyfikować pliki źródłowe, wpisać polecenie **make** i mieć pewność, że wszystkie zależne od nich pliki zostaną odpowiednio przebudowane.

Makrodefinicje

Pliki **makefile** mogą zawierać **makrodefinicje** (nie są to makroinstrukcje preprocesora, dostępne w C i C++). Makrodefinicje pozwalają na wygodną zamianę łańcuchów. Pliki **makefile** zawarte w książce używają makrodefinicji do wywołania kompilatora C++. Na przykład:

```
CPP = mojkompilator  
hello.exe: hello.cpp  
    $(CPP) hello.cpp
```

Znak = jest używany do zdefiniowania CPP jako makrodefinicji, natomiast znak \$ oraz następująca po nim para nawiasów powoduje jej rozwinięcie. W powyższym przypadku rozwinięcie oznacza, że wywołanie makrodefinicji \$(CPP) zostanie zastąpione łańcuchem **mojkompilator**. Chcąc zmienić wykorzystywany kompilator na inny, o nazwie **cpp**, należy po prostu zmodyfikować powyższą makrodefinicję, tak by miała ona postać:

```
CPP = cpp
```

Można również dopisać do tej makrodefinicji znaczniki kompilatora lub użyć w tym celu oddzielnych makrodefinicji.

Reguły przyrostkowe

Przekazywanie programowi **make** informacji o tym, w jaki sposób powinien on wywołać kompilator, oddziennie dla każdego pliku **cpp** zawartego w projekcie, jest niewygodne — szczególnie gdy wiadomo, że za każdym razem wywołanie to wygląda dokładnie tak samo. Ponieważ program **make** powstał po to, by oszczędzić na czasie, umożliwia on stosowanie skrótów podejmowanych działań pod warunkiem, że są one zależne od przyrostków nazw plików. Skróty te są nazywane *regułami przyrostkowymi*. Reguła przyrostkowa jest sposobem poinformowania programu **make** o tym, w jaki sposób przekształcić plik o określonym rozszerzeniu nazwy (np. **.cpp**) w plik o innym rozszerzeniu (np. **.obj** lub **.exe**). Po przekazaniu programowi **make** reguł, dotyczących tworzenia jednych rodzajów plików z innych, wystarczy podać informację dotyczącą zależności pomiędzy plikami. Kiedy program **make** znajdzie plik o dacie modyfikacji wcześniejszej niż pliku, od którego jest on zależny, wykorzysta tę regułę do utworzenia nowego pliku.

Reguła przyrostkowa informuje program **make**, że nie potrzebuje on szczegółowych reguł określających sposób tworzenia każdego pliku, ale może określić to na podstawie rozszerzenia nazwy pliku. W tym przypadku oznacza ona: „aby utworzyć plik o nazwie zakończonej na **exe** na podstawie pliku, którego nazwa kończy się na **cpp**, należy wywołać następujące polecenie”. Oto jak wygląda to dla powyższego przykładu:

```
CPP = mojkompilator
.SUFFIXES: .exe .cpp
.cpp.exe:
$(CPP) $<
```

Dyrektywa **.SUFFIXES** informuje program **make**, że powinien on zwrócić uwagę na pliki o podanych rozszerzeniach nazw, ponieważ w obrębie bieżącego pliku **makefile** mają one szczególne znaczenie. Następnie widoczna jest reguła przyrostkowa **.cpp.exe**, głosząca: „w taki sposób można przekształcić każdy plik o rozszerzeniu nazwy **cpp** w plik o rozszerzeniu **exe**” (w przypadku gdy plik **cpp** był modyfikowany później niż plik **exe**). Podobnie jak poprzednio używaną jest makrodefinicja \$(CPP), ale pojawia się również coś nowego: \$<. Ponieważ sekwencja ta rozpoczyna się znakiem „\$”, jest ona makrodefinicją, ale o specjalnym charakterze — wbudowaną makrodefinicją programu **make**. Może ona być używana wyłącznie w regułach przyrostkowych i oznacza: „coś, co było przyczyną zastosowania reguły” (nazywane również *elementem decydującym*), co w tym przypadku przekłada się na: „plik **cpp**, który musi zostać skompilowany”.

Po zdefiniowaniu reguł przyrostkowych można wydać polecenie w rodzaju „**make Union.exe**”, a zostanie zastosowana odpowiednia reguła przyrostkowa, mimo że w pliku **makefile** nie występuje nigdzie słowo „Union”.

Domyślne pliki wynikowe

Po makrodefinicjach i regułach przyrostkowych program **make** poszukuje w pliku pierwszego „pliku wynikowego” i tworzy go, chyba że określi się inaczej. W przypadku poniższego pliku **makefile**:

```
CPP = mojkompilator
.SUFFIXES: .exe .cpp
.cpp.exe:
    $(CPP) $<
target1.exe:
target2.exe:
```

wpisanie po polecenia „**make**” spowoduje utworzenie pliku **target1.exe** (przy użyciu domyślnej reguły przyrostkowej), ponieważ jest to pierwszy plik docelowy, napotkany przez program **make**. Aby utworzyć plik **target2.exe**, trzeba użyć polecenia „**make target2.exe**”. Jest to niewygodne, więc zazwyczaj tworzy się „ślepy” plik wynikowy, który zależy od wszystkich pozostałych plików wynikowych, jak w poniższym przykładzie:

```
CPP = mojkompilator
.SUFFIXES: .exe .cpp
.cpp.exe:
    $(CPP) $<
all: target1.exe target2.exe
```

W powyższym przykładzie „**all**” nie istnieje i nie ma również pliku o tej nazwie, więc po wpisaniu polecenia **make** program ten wykrywa plik „all” jako pierwszy plik wynikowy na liście (a więc zarazem domyślny plik wynikowy). Następnie zauważa, że plik „all” nie istnieje, więc najlepiej utworzyć go, sprawdzając wszystkie zależności. Analizuje więc plik **target1.exe** i (używając reguły przyrostkowej) sprawdza najpierw, czy plik **target1.cpp** istnieje, a następnie, czy był on modyfikowany później niż **target1.exe**. Jeżeli tak jest, program **make** wykonuje działanie zgodne z regułą przyrostkową (jeżeli określi się wyraźnie regułę dla konkretnego pliku wynikowego, to zostanie ona użyta zamiast reguły przyrostkowej). Następnie przechodzi on do kolejnego pliku, wymienionego na liście domyślnych plików wynikowych. Tak więc jeżeli utworzy się listę domyślnych plików wynikowych (określaną tradycyjnie „**all**”, ale można nadać jej dowolną nazwę), to można spowodować utworzenie wszystkich plików wykonywalnych wchodzących w skład projektu, wpisując polecenie „**make**”. Można ponadto utworzyć listy plików docelowych, niebędących plikami domyślnymi, wykonujące inne działania. Na przykład można je zorganizować w taki sposób, aby wpisanie polecenia „**make debug**” spowodowało przebudowanie wszystkich plików, z włączonym kodem uruchomieniowym.

Pliki makefile używane w książce

Wszystkie pliki przykładowych programów zostały automatycznie utworzone, na podstawie wydruków zawartych w tekstopisowej wersji książki, za pomocą programu **ExtractCode.cpp** (pochodzącego z jej drugiego tomu). Następnie zostały one umieszczone w podkatalogach o nazwach odpowiadających poszczególnym rozdziałom. Ponadto program **ExtractCode.cpp** tworzy w każdym podkatalogu szereg plików **makefile** (posiadających różne nazwy), dzięki którym można przejść do takiego podkatalogu i użyć polecenia **make -f mojkompilator.makefile**, zastępując „**mojkompilator**” nazwą używanego kompilatora (opcja „-f” określa natomiast, że wymieniony po niej plik ma zostać użyty jako **makefile**). Ostatecznie program **ExtractCode.cpp** tworzy „nadzędny” plik **makefile**, znajdujący się w głównym katalogu, do którego zostały rozpakowane pliki zawierające programy źródłowe. Ten „nadzędny” plik **makefile** powoduje przejście do każdego podkatalogu i wywołanie w nim programu **make** z odpowiednim plikiem **makefile**. W ten sposób można skompilować wszystkie programy zawarte w książce, używając do tego celu pojedynczego polecenia **make**, a proces ten zostanie przerwany, gdy kompilator nie będzie potrafił poradzić sobie z jakimś plikiem (kompilator zgodny z C++ powinien być w stanie skompilować wszystkie programy zawarte w książce). Ponieważ implementacje programu **make** różnią się w poszczególnych systemach, w plikach **makefile** zostały wykorzystane jedynie najbardziej podstawowe, najczęściej używane polecenia.

Przykładowy plik makefile

Jak już wspomniano, program **ExtractCode.cpp**, tworzący pliki programów na podstawie tekstu książki, automatycznie utworzył dla każdego rozdziału pliki **makefile**. Dlatego też pliki te nie zostały zamieszczone w książce (wszystkie pliki **makefile** zostały spakowane razem z programami źródłowymi i można je pobrać pod adresem: <ftp://ftp.helion.pl/przyklady/thicpp.zip>). Pozytywnie będzie jednak zobaczyć przykład pliku **makefile**. Zamieszczony poniżej plik stanowi skróconą wersję pliku, który został automatycznie utworzony dla bieżącego rozdziału przez program **ExtractCode.cpp**. W każdym podkatalogu znajduje się większa liczba plików **makefile** (mają one różne nazwy i należy wywołać je za pomocą polecenia „**make -f**”). Przedstawiony poniżej plik jest przeznaczony dla GNU C++:

```
CPP = g++
OFLAG = -o
.SUFFIXES : .o .cpp .c
.cpp.o :
    $(CPP) $(CPPFLAGS) -c $<
.C.o :
    $(CPP) $(CPPFLAGS) -c $<

all: \
    Return \
    Declare \
    Ifthen \
    Guess \
    Guess2
# Nie pokazano pozostałych plików, zawartych w rozdziale
```

```
Return: Return.o
$(CPP) $(OFLAG)Return Return.o

Declare: Declare.o
$(CPP) $(OFLAG)Declare Declare.o

Ifthen: Ifthen.o
$(CPP) $(OFLAG)Ifthen Ifthen.o

Guess: Guess.o
$(CPP) $(OFLAG)Guess Guess.o

Guess2: Guess2.o
$(CPP) $(OFLAG)Guess2 Guess2.o

Return.o: Return.cpp
Declare.o: Declare.cpp
Ifthen.o: Ifthen.cpp
Guess.o: Guess.cpp
Guess2.o: Guess2.cpp
```

Makrodefinicja CPP określa nazwę kompilatora. Aby użyć innego kompilatora, należy albo dokonać modyfikacji w pliku **makefile**, albo zmienić wartość **makrodefinicji** w wierszu polecień, jak w poniższym przykładzie:

```
make CPP=cpp
```

Warto jednak zwrócić uwagę na to, że program **ExtractCode.cpp** zawiera mechanizm umożliwiający automatyczne utworzenie plików **makefile** dla dodatkowych kompilatorów.

Druga **makrodefinicja**, **OFLAG**, jest opcją używaną do określenia nazwy pliku wyjściowego. Mimo że wiele kompilatorów automatycznie 'przyjmuje', że nazwa pliku wyjściowego posiada tę samą nazwę (z wyjątkiem rozszerzenia), co plik wejściowy, to niektóre kompilatory tego nie robią (na przykład kompilatory w systemach Linux i Unix, które domyślnie tworząplik o nazwie **a.out**).

W przedstawionym pliku można również zauważyc dwie reguły przyrostkowe — jedną dla plików **cpp**, a drugą dla plików **.c** (w przypadku konieczności kompilacji kodu źródłowego języka C). Domyślnym plikiem wynikowym jest **all**, a każdy ze związań z nim wierszy jest „kontynuowany” za pomocą lewego ukośnika, aż do pliku **Guess2**, ostatniego na liście, i dlatego znajdującego się w wierszu pozbawionym lewego ukośnika. W rozdiale zawarto znacznie więcej plików, ale z uwagi na zwięzłość w przykładzie wymieniono tylko kilka z nich.

Reguły przyrostkowe dotyczą tworzenia plików wynikowych (z rozszerzeniem **.o**) na podstawie plików **cpp**, ale na ogół trzeba jawnie określić reguły tworzenia plików wynikowych. W typowym przypadku pliki wykonywalne powstają bowiem w rezultacie połączenia ze sobą wielu różnych plików wynikowych i program **make** nie jest w stanie odgadnąć ich nazw. W powyższym przykładzie (w systemie Linux lub Unix) nie istnieje również standardowe rozszerzenie nazw plików wykonywalnych, więc reguły przyrostkowe nie będą w takim przypadku działać. Dlatego też w przykładowym pliku widoczne są jasno określone reguły tworzenia wszystkich plików wykonywalnych.

Przedstawiony powyżej plik **makefile** wykorzystuje w najbezpieczniejszy sposób możliwie jak najmniejszyszy zbiór poleceń programu **make** — używa jedynie jego podstawowych pojęć plików wynikowych i zależności oraz makrodefinicji. Takie podejście gwarantuje działanie z największą możliwą liczbą wersji polecenia make. W wyniku powstają co prawda na ogół większe pliki **makefile**, ale nie jest to problem, ponieważ są one automatycznie generowane przez program **ExtractCode.cpp**.

Istnieje wiele dodatkowych poleceń programu **make**, które nie są używane w książce; są również nowsze i przemyślniejsze wersje i odmiany tego programu, zawierające zaawansowane skróty, które pozwalają na oszczędzenie mnóstwa czasu. Dokumentacja dołączona do używanej przez ciebie wersji programu **make** może zawierać opis dodatkowych jego cech.Więcej informacji na temat programu możesz znaleźć w książce *Managing Projects with Make*, napisanej przez Orama i Talbotta (O'Reilly, 1993). Jeżeli natomiast producent stosowanego przez ciebie kompilatora w ogóle nie dostarcza programu **make**, albo używa jego niestandardowej wersji, możesz znaleźć jego wersje GNU przeznaczone dla dowolnej istniejącej platformy, poszukując w Internecie archiwów programów GNU (których jest wiele).

Podsumowanie

Rozdział stanowił dość bogaty przegląd wszystkich podstawowych cech składni języka C++, z których większość wywodzi się z języka C i występuje w obu tych językach (w rezultacie język C++ może poszczycić się wstępnią zgodnością z językiem C). Mimo że w przeglądzie zaprezentowano również pewne cechy języka C++, to został on przygotowany przede wszystkim z myślą o czytelnikach, którzy mają już pewne doświadczenie w programowaniu i potrzebują wprowadzenia obejmującego podstawy składni języków C i C++. Jeżeli programujesz już w C, to zapewne, oprócz cech języka C++, które są w większości dla ciebie nowe, znalazłeś tu niewiele nieznanych ci informacji dotyczących języka C. Jeżeli jednak zawartość rozdziału wydaje ci się zbyt przytłaczająca, to zapoznaj się z kursem *Thinking in C: Foundations for C++ and Java* (składającym się z wykładów, ćwiczeń i wskazówek dotyczących ich rozwiązań), dostępnym w witrynie www.BruceEckel.com.

Ćwiczenia

Rozwiązania wybranych ćwiczeń można znaleźć w dokumencie elektronicznym *The Thinking in C++ Annotated Solution Guide*, dostępnym za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Utwórz plik nagłówkowy (z rozszerzeniem .h). W pliku tym zadeklaruj grupę funkcji poprzez zmianę list argumentów i zwracanych przez nie wartości, wybierając je spośród typów: **void**, **char**, **int** oraz **float**. Następnie utwórz plik .cpp, do którego dołączany jest utworzony uprzednio plik nagłówkowy, i utwórz w nim definicje wszystkich tych funkcji. Każda z definicji powinna

drukować nazwę funkcji, listę argumentów oraz zwracaną wartość, dzięki czemu wiadomo, że została ona **wywołana**. Utwórz drugi plik **.cpp**, do którego dołączany będzie twój plik nagłówkowy. Ponadto będzie się w nim znajdowała definicja funkcji **int main()**, zawierająca wywołania wszystkich utworzonych uprzednio funkcji. Skompiluj i uruchom ten program.

2. Napisz program, wykorzystujący dwie zagnieździone pętle for i operator modułu (**%**) do znalezienia i wydrukowania liczb pierwszych (liczb całkowitych podzielnych jedynie przez 1 i same siebie).
3. Napisz program, wykorzystujący pętlę **while** do wczytywania słów ze standardowego wejścia (cin) do łańcucha (**string**). Będzie to „nieskończona” pętla **while**, którą przerwiesz (i opuścisz program) za pomocą instrukcji **break**. Przypisz każdemu przeczytanemu słowi wartość **całkowitą**, używając do tego sekwencji instrukcji if, a następnie zastosuj instrukcję **switch**, wykorzystującą tę wartość w charakterze selektora (zaproponowana kolejność **nie jest** przykładem dobrego stylu programowania — ma ona umożliwić ci ćwiczenie związane z przepływem sterowania). W każdym przypadku **case** wydrukuje coś sensownego. Musisz dokonać wyboru „**interesujących**” słów i ich znaczenia, a także zdecydować, jakie słowo będzie poleciением zakończenia programu. Przetestuj program, przekierowując jego standardowe wejście plik (**jeżeli chcesz oszczędzić sobie pisania**, plikiem tym może być plik źródłowy programu).
4. Zmodyfikuj program **Menu.cpp** tak, by zamiast instrukcji **if** używał on instrukcji **switch**.
5. Napisz program, który będzie obliczał dwa wyrażenia, zamieszczone w podrozdziale „**Priorytety**”.
6. Zmodyfikuj program **YourPets2.cpp** w taki sposób, by wykorzystywał wiele różnych typów danych (**char**, **int**, **float**, **double** i ich wariantów). Uruchom program i stwórz mapę zawartości jego pamięci. Jeżeli masz dostęp do różnych komputerów, systemów operacyjnych lub kompilatorów, wykonaj eksperyment w takiej liczbie ich kombinacji, w jakiej jesteś w stanie to zrobić.
7. Utwórz dwie **funkcje**, z których jedna będzie przyjmowała argument typu **string***, a druga typu **string&**. Każda z tych funkcji powinna modyfikować zewnętrzny obiekt **string** we właściwy sobie sposób. Utwórz i zainicjuj w funkcji **main()** obiekt typu **string**, wydrukuj jego wartość, a następnie **przekaz** go kolejno każdej z dwóch funkcji, drukując za każdym razem wynik ich działania.
8. Napisz program z zastosowaniem wszystkich trójznaków, aby przekonać się, czy używany przez ciebie kompilator je obsługuje.
9. Skompiluj i uruchom program **Static.cpp**. Usuń z programu słowo kluczowe **static**, skompiluj i uruchom go ponownie, a następnie wyjaśnij to, co się stało.
10. Spróbuj skompilować i dokonać **łączenia** plików **FileStatic.cpp** i **FileStatic2.cpp**. Co oznacza zgłoszony komunikat o błędzie?
11. Zmodyfikuj program **Boolean.cpp** tak, by zamiast z wartościami całkowitymi działał z wartościami typu **double**.

12. Zmodyfikuj programy **Boolean.cpp** i **Bitwise.cpp** w taki sposób, by używały operatorów dosłownych (jeżeli twój kompilator jest zgodny ze standardem C++, to będzie je obsługiwał).
13. Zmodyfikuj program **Bitwise.cpp** tak, by używał funkcji zawartych w programie **Rotation.cpp**. Upewnij się, że wyświetlasz jego wyniki w taki sposób, że widać, co dzieje się w czasie obrotów.
14. Zmodyfikuj program **Ifthen.cpp** w taki sposób, by używał on operatora trójargumentowego **if-else** (?:).
15. Utwórz strukturę przechowującą dwa obiekty typu string i jedną liczbę całkowitą. Utwórz egzemplarz tej struktury, zainicjuj wszystkie trzyjego wartości, a następnie wydrukuj. Pobierz adres obiektu i przypisz go wskaźnikowi do typu twojej struktury. Używając wskaźnika, zmień wszystkie trzy wartości obiektu i wydrukuj je.
16. Napisz program, wykorzystujący wyliczenie kolorów. Utwórz zmienną typu wyliczeniowego i wydrukuj liczby odpowiadające nazwom poszczególnych kolorów, używając do tego pętli for.
17. Poeksperymentuj z programem **Union.cpp**, usuwając różne elementy unii i obserwując, jaki ma to wpływ naję wielkość. Spróbuj dokonać przypisania do jednego z elementów unii (posiadającego jakiś typ), a następnie wydrukuj wartość innego elementu (o jakimś innym typie), wchodzącego w jej skład, i zobacz, co się stanie.
18. Napisz program definiujący dwie tablice liczb całkowitych, jedną obok drugiej. Dokonaj przypisania do elementu pierwszej tablicy, przekraczając wartość jej maksymalnego indeksu. Wydrukuj zawartość drugiej tablicy i zaobserwuj spowodowane przez to przypisanie zmiany. Następnie pomiędzy tymi dwoma tablicami umieść definicję zmiennej typu **char** i powtórz eksperyment. Aby ułatwić sobie kodowanie, możesz utworzyć funkcję drukującą tablice.
19. Zmodyfikuj program **ArrayAddresses.cpp** w taki sposób, by działał z typami danych **char**, **long int**, **float** i **double**.
20. Zastosuj sposób pokazany w programie **ArrayAddresses.cpp** do wydrukowania wielkości struktury i adresów elementów tablicy, zawartych w programie **StructArray.cpp**.
21. Utwórz tablicę obiektów typu **string** i przypisz kademuzej elementowi łańcuch. Wydrukuj zawartość tablicy, używając pętli for.
22. Na podstawie programu **ArgsToInts.cpp** napisz dwa nowe programy, używające odpowiednio funkcji **atol()** i **atof()**.
23. Zmodyfikuj program **PointerIncrement2.cpp** w taki sposób, by zamiast struktury używała unii.
24. Zmodyfikuj program **PointerArithmetic.cpp** w taki sposób, by działał z typami **long** i **long double**.

25. Zdefiniuj zmienną typu **float**. Pobierz jej adres, dokonaj jego rzutowania na typ **unsigned char***, a następnie przypisz go wskaźnikowi do typu **unsigned char**. Za pomocą tego wskaźnika i nawiasów kwadratowych dokonaj indeksowania w obrębie zmiennej **float** (przesuwając się od 0 do **sizeof(float)**), drukując „mapę” tej zmiennej przy użyciu opisanej w rozdziale funkcji **printBinary()**. Zmień wartość zmiennej typu **float** i przekonaj się, czy potrafisz wyjaśnić rezultat (zmienna **float** zawiera „zaszyfrowane” dane).
26. Zdefiniuj tablicę liczb całkowitych. Pobierz adres początku tej tablicy i użyj rzutowania **static_cast** w celu przekształcenia go do typu **void***. Napisz funkcję, która pobiera argument typu **void***, liczbę (oznaczającą liczbę bajtów) i wartość (określającą wartość, która powinna zostać przypisana każdemu bajtowi). Funkcja powinna przypisać każdemu bajtowi określona wartość z podanego zakresu. Wypróbuje działanie funkcji na zdefiniowanej uprzednio tablicy liczb całkowitych.
27. Utwórz stałą (**const**) tablicę liczb typu **double** i tablicę liczb typu **double**, opatrzoną modyfikatorem **volatile**. Przejdz przez każdą tablicę, używając rzutowań **const_cast** w celu przekształcenia każdego z ich elementów na typy pozabawione modyfikatorów **const** i **volatile** oraz nadając tym elementom wartości.
28. Napisz funkcję, pobierającą wskaźnik do tablicy liczb typu **double**, oraz wartość, określającą wielkość tablicy. Funkcja ta powinna wydrukować wszystkie elementy tablicy. Utwórz tablicę liczb typu **double**, zainicjuj każdy jej element wartością równą zero, a potem użyj swojej funkcji do wydrukowania zawartości tablicy. Następnie użyj rzutowania **reinterpret_cast** do przekształcenia adresu tablicy do typu **unsigned char*** i przypisz każdemu bajtowi takiej tablicy wartość 1 (wskaźówka: jest potrzebny operator **sizeof** do określenia liczby bajtów składających się na typ **double**). Następnie użyj napisanej wcześniej funkcji do wydrukowania zawartości tablicy. Jak myślisz, dlaczego jej elementy nie mają wartości 1.0?
29. (Trudne) Zmodyfikuj program **FloatingAsBinary.cpp** w taki sposób, by drukował on każdączęść liczb typu **double** w postaci oddzielnej grupy bitów. Aby to zrobić, należy zastąpić wywołanie funkcji **printBinary()** własnym, napisanym specjalnie w tym celu, kodem (który możesz opracować na podstawie funkcji **printBinary()**). Trzeba również odszukać i zrozumieć informacje dotyczące formatu **zmiennopozycyjnego** i kolejności bajtów, wykorzystywanych przez posiadany przez ciebie kompilator (to jest właśnie trudna część tego ćwiczenia).
30. Utwórz plik **makefile**, który nie tylko skompiluje pliki **YourPets1.cpp** i **YourPets2.cpp** (używając posiadanego przez ciebie kompilatora), ale również, w ramach działania związanego z domylnymi plikami wynikowymi, uruchomi oba te programy. Upewnij się, że zostały użyte reguły przyrostkowe.
31. Zmodyfikuj program **StringizingExpressions.cpp** w taki sposób, aby P(A) była zdefiniowana warunkowo, co pozwoli na automatyczne usunięcie kodu uruchomieniowego za pomocą znacznika, podanego w wierszu poleceń

kompilatora. Należy sprawdzić w dokumentacji używanego przez ciebie kompilatora, w jaki sposób definiuje się i unieważnia wartości preprocesora w wierszu poleceń kompilatora.

32. Zdefiniuj funkcję przyjmującą argument typu **double** i zwracającą wartość typu **int**. Utwórz i zainicjuj wskaźnik do tej funkcji, a następnie wywołaj ją, używając wskaźnika.
33. Zadeklaruj wskaźnik do funkcji przyjmującej argument typu **int** i zwracającej wskaźnik do funkcji, która przyjmuje argument typu **char** i zwraca wartość typu **float**.
34. Zmodyfikuj program **FunctionTable.cpp** w taki sposób, by każda funkcja zwracała łańcuch (zamiast drukować komunikat), którego wartość zostanie wydrukowana wewnątrz funkcji **main()**.
35. Utwórz plik **makefile** dla jednego z poprzednich ćwiczeń (dowolnie wybranego), który umożliwia wpisanie polecenia **make**, tworzącego gotowy program wykonywalny, oraz polecenia **make debug**, tworzącego program zawierający informacje uruchomieniowe.

Rozdział 4.

Abstrakcja danych

Język C++ jest narzędziem zwiększającym wydajność. Z jakiego innego powodu warto by podejmować wysiłek (**jest** to niewątpliwie trud, niezależnie od tego, jak łatwym próbujemy uczynić przejście) zmianyjęzyka, który już znasz i z powodzeniem wykorzystujesz, na taki, za którego pomocą osiągniesz przez pewien czas *mniejszą* wydajność, **dopóki biegłe go nie opanujesz?** Odpowiedź jest oczywista: wykorzystując nowe narzędzie, osiągniesz duże korzyści.

W terminologii stosowanej w programowaniu komputerów wydajność **oznacza**, że mniej liczna grupa osób może tworzyć większe i znacznie bardziej skomplikowane programy w krótszym czasie. Kiedy należy dokonać wyboru języka programowania, ważne sąż pewności również inne kwestie, takie jak efektywność (czy naturajęzyka nie spowoduje spowolnienia programu i rozdęcie jego kodu?), bezpieczeństwo (czy język zapewni, że program będzie zawsze wykonywał to, co zamierzono, i elegancko obsługuje błędy?) oraz pielęgnację (czy dzięki wykorzystaniu języka powstanie kod łatwy do zrozumienia, modyfikacji i rozszerzania?). Te z pewnością istotne czynniki zostaną przeanalizowane w niniejszej książce.

Zwykła wydajność oznacza, że program, którego napisanie zajmowało poprzednio trzem osobom tydzień, zabiera teraz jednej osobie kilka dni. Zagadnienie to dotyczy wielu aspektów ekonomii. Odczuwasz satysfakcję, ponieważ dokonałeś jakiegoś dzieła, twój klient (lub szef) jest zadowolony, gdyż produkty są tworzone szybciej, przez mniejszą liczbę ludzi, a klient również ma powody do radości, bowiem nabywa produkt taniej. Jedynym sposobem uzyskania dużego wzrostu wydajności jest wykorzystanie kodu napisanego przez innych. Oznacza to konieczność używania bibliotek.

Bibliotekę tworzą fragmenty kodu, napisane przez kogoś i połączone w całość. Najmniejsze pakiety składają się często z pliku o rozszerzeniu takim jak **lib** oraz jednego lub większej liczby plików **nagłówkowych**, informujących kompilator o zawartości biblioteki. Program łączący wie, w jaki sposób przeszukiwać plik biblioteki i wydobyć z niej odpowiedni, skompilowany kod. Jest to jednak tylko jedna z metod dostarczania bibliotek. Na platformach obejmujących wiele różnorodnych architektur, takich jak Linux czy Unix, często jedynym rozsądny sposobem dostarczania biblioteki jest dołączenie do niej kodu źródłowego, co pozwala na jej **przekonfigurowanie** i powtórną komplikację w nowym środowisku.

A zatem biblioteki są prawdopodobnie najważniejszym sposobem zwiększenia wydajności, a jednym z głównych celów projektowych języka C++ było ułatwienie korzystania z bibliotek. Wynika z tego, że istnieje przeszkoda utrudniająca wykorzystywanie bibliotek w języku C. Zrozumienie tego umożliwi ci pojęcie, jak został zaprojektowany język C++, a zatem również jak należy go używać.

Miniaturowa biblioteka w stylu C

Biblioteki powstają na ogół jako zbiory funkcji, ale jeżeli używasz niezależnych bibliotek języka C, to wiesz, że proces ten jest zazwyczaj bardziej skomplikowany, ponieważ życie nie składa się tylko z zachowań, działań i funkcji. Są również właściwości (niebieski, kilogramy, faktura, jasność), reprezentowane za pomocą danych. Kiedy zaczynasz zajmować się w języku C zbiorem właściwości, wygodne jest połączenie ich w strukturę, szczególnie gdy zamierzasz opisywać za ich pomocą więcej niż jeden element przestrzeni problemu. Dzięki temu możesz następnie utworzyć zmienną będącą egzemplarzem struktury, oddzielnie dla każdego elementu.

Tak więc większość bibliotek języka C zawiera zbiór struktur oraz zbiór działających na nich funkcji. Jako przykład takiego systemu rozważmy narzędzie programistyczne działające jak tablica, ale którego wielkość może być określona w czasie pracy programu, kiedy jest ono tworzone. Zostanie mu nadana nazwa **CStash**. Mimo że zostało ono napisane w C++, przypomina napisane w języku C.

```
//: C04:CLib.h
// Plik nagłówkowy biblioteki w stylu C
// Tablicopodobny twór. tworzony w czasie
// pracy programu
typedef struct CStashTag {
    int size;          // Wielkość każdego elementu
    int quantity;      // Liczba elementów pamięci
    int next;          // Następny pusty element
    // Dynamicznie przydzielana tablica bajtów:
    unsigned char* storage;
} CStash;

void initialize(CStash* s, int size);
void cleanup(CStash* s);
int add(CStash* s, const void* element);
void* fetch(CStash* s, int index);
int count(CStash* s);
void inflate(CStash* s, int increase);
//:-
```

Takie identyfikatory, jak **CstashTag**, są na ogół używane w strukturach, w których istnieje konieczność odwołania się do struktury z jej wnętrza. Na przykład podczas tworzenia *listy powiązanej* (każdy element takiej listy zawiera wskaźnik do następnego elementu) niezbędny jest wskaźnik do następnej zmiennej będącej **strukturą**, dlatego też potrzebny jest sposób określenia typu takiego wskaźnika wewnątrz ciała struktury. Również stosowanie deklaracji **typedef** w stosunku do struktur (przedstawione powyżej) jest niemal powszechnie w bibliotekach języka C. Dzięki temu można traktować struktury w taki sposób, jakby były nowymi typami, i definiować zmienne ich typów:

CStash A, B, C;

Wskaźnik do pamięci jest typu **unsigned char***. Najmniejszym fragmentem pamięci, obsługiwanym przez kompilator języka C, jest typ **unsigned char**, chociaż w przypadku niektórych komputerów może on być tej samej wielkości, co największy fragment. Zależy to od konkretnej implementacji, ale często ma on wielkość jednego bajtu. Może się wydawać, że skoro **CStash** jest projektowany w taki sposób, by przechowywał zmienne dowolnego typu, to bardziej odpowiedni byłby typ **void***. Jednakże celem nie jest w tym przypadku traktowanie pamięci jako bloku jakiegoś nieokreślonego typu, lecz jako bloku kolejnych bajtów.

Kod źródłowy pliku zawierającego implementację (którego możesz nie uzyskać w przypadku, gdy nabywasz komercyjną bibliotekę — zapewne otrzymasz jedynie skompilowane pliki o rozszerzeniach **obj**, **lib**, **dll** itp.) wygląda następująco:

```
//: C04:CLib.cpp {0}
// Implementacja przykładowej biblioteki w stylu C
// Deklaracje struktury i funkcji:
#include "CLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Liczba elementów dodawanych
// w przypadku powiększenia pamięci:
const int increment = 100;

void initialize(CStash* s, int sz) {
    s->size = sz;
    s->quantity = 0;
    s->storage = 0;
    s->next = 0;
}

int add(CStash* s, const void* element) {
    if(s->next >= s->quantity) //Czy wystarczy pamięci?
        inflate(s, increment);
    // Kopiowanie elementu do pamięci.
    // począwszy od następnego wolnego miejsca:
    int startBytes = s->next * s->size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < s->size; i++)
        s->storage[startBytes + i] = e[i];
    s->next++;
    return(s->next - 1); // Numer indeksu
}

void* fetch(CStash* s, int index) {
    // Kontrola zakresu indeksu
    assert(0 <= index);
    if(index >= s->next)
        return 0; // Oznaczenie końca
    // Tworzenie wskaźnika do żądanego elementu:
    return &(s->storage[index * s->size]);
}
```

```

int count(CStash* s) {
    return s->next; // Liczba elementów w CStash
}

void inflate(CStash* s, int increase) {
    assert(increase > 0);
    int newQuantity = s->quantity + increase;
    int newBytes = newQuantity * s->size;
    int oldBytes = s->quantity * s->size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = s->storage[i]; // Kopiowanie starego obszaru do nowego
    delete [] (s->storage); // Stary obszar pamięci
    s->storage = b; // Wskaźnik do nowego obszaru
    s->quantity = newQuantity;
}

void cleanup(CStash* s) {
    if(s->storage != 0) {
        cout << "zwalnianie pamięci" << endl;
        delete [] s->storage;
    }
} //:-)

```

Funkcja **initialize()** dokonuje niezbędnej konfiguracji struktury **CStash**, przypisując jej wewnętrznym zmiennym odpowiednie wartości. Pierwotnie wskaźnik **storage** jest ustawiany na zero, ponieważ nie została jeszcze przydzielona żadna pamięć.

Funkcja **add()** wstawia do **CStash** element, umieszczając go na następnej wolnej pozycji. Najpierw sprawdza, czy jest jeszcze dostępna pamięć. Jeżeli nie, powiększa obszar pamięci, używając do tego funkcji **inflate()**, która zostanie opisana później.

Ponieważ kompilator nie zna dokładnego typu przechowywanych zmiennych (wszystkie funkcje przyjmują wskaźniki typu **void***), nie można zwyczajnie dokonać przypisania, co z pewnością byłoby najwygodniejsze. Trzeba natomiast kopiować zmienną, bajt po bajcie. Najprostszą metodą dokonania tego kopiowania jest wykorzystanie indeksowania tablicy. Zazwyczaj w obszarze pamięci wskazywanym przez **storage** znajdują się już jakieś dane, o czym świadczy wartość zmiennej **next**. Aby rozpocząć od przesunięcia o właściwą liczbę bajtów, zmienna **next** jest mnożona przez wielkość każdego elementu (wyrażoną w bajtach) i daje w wyniku wartość **startBytes**. Następnie argument **element** jest rzutowany na typ **unsigned char***, dzięki czemu będzie się można do niego odwoływać bajt po bajcie, kopując go do wolnego obszaru pamięci, wskazywanej przez zmienną **storage**. Zwiększana jest zmienna **next**, co sprawia, że określa ona następny wolny fragment pamięci, a także zwracany jest „numer indeksu”, pod którym została zapisana wartość. Pozwoli to na jej późniejsze odczytanie, po podaniu tego indeksu funkcji **fetch()**.

Funkcja **fetch()** sprawdza, czy podany indeks nie wykracza poza dopuszczalne granice, a następnie zwraca adres żądanej zmiennej, wyznaczając go na podstawie argumentu **index**. Ponieważ argument **index** określa liczbę elementów, o jaką należy przesunąć się w obrębie **CStash**, musi być on pomnożony przez liczbę bajtów zajmowaną przez każdy element. W wyniku uzyskuje się liczbę będącą wielkością przesunięcia.

wyrażoną w bajtach. Kiedy przesunięcie to jest używane do tablicowego indeksowania wskaźnika **storage**, jego wynikiem nie jest adres, tylko bajt, znajdujący się pod tym adresem. W celu uzyskania adresu należy zatem użyć operatora adresu &.

Funkcja **count()** może się wydać doświadczonemu programiście nieco osobliwa. Wygląda na to, że zadano sobie wiele trudu po to, by wykonać coś, co prawdopodobnie można by zrobić znacznie łatwiej „ręcznie”. Jeśli na przykład dysponuje się strukturą typu **CStash**, o nazwie **intStash**, znacznie prostsze wydaje się określenie zawartej w niej liczby elementów za pomocą odwołania **intStash.next**, zamiast wywołania funkcji (które wiąże się z narzutem) w postaci **count(&intStash)**. Jeżeli jednak zamierza się zmienić wewnętrzną reprezentację struktury **CStash**, a zatem również sposób, w jaki wyznaczana jest liczba zawartych w niej elementów, to interfejs w postaci wywołania funkcji zapewnia jej niezbędną elastyczność. Niestety, większość programistów nie weźmie pod uwagę informacji o wyższości takiego podejścia do projektu biblioteki. Po przyjrzeniu się strukturze będą bezpośrednio odczytywać wartość zmiennej **next**, a być może nawet zmieniać ją bez twojego pozwolenia. Gdyby tylko istniał jakiś sposób, umożliwiający projektantowi biblioteki sprawowanie nad tym większej kontroli! Jak się później przekonamy, sposób taki istnieje.

Dynamiczny przydział pamięci

Nigdy nie wiadomo, ile pamięci będzie potrzebowała zmienna typu **CStash**, więc pamięć wskazywana przez zmienną **storage** jest przydzielana ze *sterty* (ang. *heap*). Sterta jest dużym blokiem pamięci, z którego przydzielane są w czasie pracy programu mniejsze fragmenty. Sterty używa się w przypadku, gdy w czasie pisania programu nie jest znana wielkość potrzebnej pamięci. Na przykład tylko w czasie pracy programu można określić, że potrzebna jest pamięć przechowująca 200, a nie 20 zmiennych **Samolot**. W standardzie C funkcjami służącymi do dynamicznego przydziału pamięci były: **malloc()**, **calloc()**, **realloc()** i **free()**. W języku C++ zastąpiono wywołania funkcji bibliotecznych bardziej wyrafinowanym (choć prostszym w użyciu) rozwiązaniem dotyczącym obsługi pamięci dynamicznej, włączonym do języka w postaci słów kluczowych **new** i **delete**.

Funkcja **inflate()** używa słowa kluczowego **new** do przydzielenia typowi **CStash** większego fragmentu pamięci. W takiej sytuacji zwiększymy jedynie przydzieloną pamięć, nigdy jej nie zmniejszając, a funkcja **assert()** zapewnia, że funkcji **inflate()** nigdy nie zostanie przekazana ujemna wartość parametru **increase**. Liczba elementów, które będą mogły być przechowywane w pamięci (po zakończeniu funkcji **inflate()**), jest wyznaczana jako **newQuantity**, a następnie mnożona przez liczbę bajtów przypadających na element. W rezultacie daje ona wartość **newBytes**, będącą liczbą przydzielanych bajtów. Aby było wiadomo, ile bajtów należy skopiować z poprzednio przydzielonego obszaru pamięci, na podstawie poprzedniej wartości zmiennej **quantity** wyznaczana jest wartość **oldBytes**.

Faktyczny przydział pamięci zachodzi w *wyrażeniu new*, które zawiera słowo klu-

czowe **new**:

Ogólna postać wyrażenia **new** jest następująca:

```
new typ;
```

gdzie **typ** opisuje typ zmiennej, która ma zostać przydzielona na stercie. W naszym przypadku potrzebna jest tablica elementów typu **unsigned char**, o wielkości **newBytes**, i dlatego tworzą one w przedstawionym programie **typ**. Można również dokonać przydziału czegoś tak prostego jak liczba **całkowita**, zapisując:

```
new int;
```

i chociaż przypadek taki zdarza się rzadko, to jest oczywiste, że postać wyrażenia jest spójna.

Wyrażenie **new** zwraca *wskaźnik* do obiektu dokładnie takiego typu, o jakiego przydzielenie został poproszony. A zatem, pisząc **new typ**, uzyskuje się w wyniku wskaźnik do **typu**. Jeżeli zapiszemy **new int**, otrzymamy wskaźnik do liczby całkowitej. Jeżeli potrzebna jest tablica znaków, uzyskany wskaźnik będzie wskazywał pierwszy element tej tablicy. Kompilator zagwarantuje, że wartość zwrócona przez wyrażenie **new** zostanie przypisana wskaźnikowi odpowiedniego typu.

Oczywiście, zawsze może się zdarzyć, że jeśli nie będzie już wolnej pamięci, żądanie jej przydziału zakończy się niepowodzeniem. Jak się przekonasz, język C++ posiada mechanizmy, uruchamiane w sytuacji, gdy operacja przydziału pamięci zakończy się niepowodzeniem.

Po przydzieleniu nowego obszaru pamięci muszą zostać skopiowane do niego dane znajdujące się w starym obszarze pamięci. Jest to dokonywane za pomocą indeksowania tablicowego — w pętli, bajt po bajcie. Po skopiowaniu danych stary obszar danych musi zostać zwolniony, dzięki czemu można go wykorzystać w innych partiach programu, kiedy będąone potrzebowaly przydzielenia pamięci. Słowo kluczowe **delete** jest przeciwieństwem słowa **new** i musi być ono użyt do zwolnienia każdego obszaru pamięci, przydzielonego za pomocą **new** (jeżeli zapomnisz o użyciu operatora **delete**, pamięć ta pozostanie niedostępna; jeśli zdarzy się więcej tzw. *wycieków pamięci*, w końcu zatrąkniesz). Ponadto w przypadku usuwania tablic stosowanajest specjalna składnia. Ma ona taką postać, jakby trzeba było przypomnieć kompilatorowi, że wskaźnik nie wskazuje pojedynczego obiektu, tylko tablicę — przed usuwanym wskaźnikiem umieszcza się pusty nawias kwadratowy:

```
delete []mojaTablica;
```

Po usunięciu starego obszaru pamięci wskaźnikowi **storage** można przypisać adres nowego obszaru; aktualizowana jest informacja o jego wielkości i funkcja **inflate()** kończy swą pracę.

Zwróci uwagę na to, że menedżer sterty jest dość prosty. Udostępnia on fragmenty pamięci, a następnie zabiera je z powrotem, po użyciu operatora **delete**. Nie wbudowano w niego żadnych mechanizmów umożliwiających *upakowanie sterty*, które dokonałyby jej kompresji, dzięki czemu na stercie byłyby dostępne większe fragmenty pamięci. Jeżeli program przydziela i zwalnia przez pewien czas pamięć na stercie, może to doprowadzić do jej *fragmentacji*. Polega ona na tym, że na stercie znajduje się jeszcze dużo wolnej pamięci, ale żaden jej fragment nie jest dostatecznie duży, by

można przydzielić żądaną w danej chwili wielkość. Program dokonujący upakowania sterty sprawia, że program się komplikuje, ponieważ przesuwa on przydzielone obszary pamięci, co powoduje, że wartości wskaźników przestają być aktualne. Niektóre środowiska operacyjne zawierają wbudowane mechanizmy pakowania sterty, ale wymagają użycia, zamiast wskaźników, specjalnych *uchwytów* pamięci (które mogą być czasowo przekształcane we wskaźniki, po zablokowaniu ich w pamięci w taki sposób, aby program pakujący stertę nie mógł ich przesunąć). Można również utworzyć własny system pakujący stertę, ale nie jest to łatwe zadanie.

Jeżeli jakaś zmienna tworzona jest w czasie komplikacji na stosie, to pamięć dla niej jest automatycznie przydzielana i zwalniana przez kompilator. Kompilator wie dokładnie, ile pamięci potrzeba, a także — dzięki zasięgowi — zna czas życia zmiennej. Jednak w przypadku dynamicznego przydziału pamięci, kompilator nie zna wielkości wymaganej pamięci ani okresu, w którym będzie ona potrzebna. Oznacza to, że pamięć nie jest automatycznie zwalniana. A zatem należy ją zwolnić za pomocą operatora **delete**, informującego menedżera sterty, że pamięć ta może zostać wykorzystana podczas następnego użycia operatora **new**. Logiczne jest, by w bibliotece zwalnianie pamięci odbywało się w obrębie funkcji **cleanup()** (ang. *cleanup* — sprzątanie), ponieważ dokonywane są w niej wszystkie końcowe operacje porządkowe.

W celu przetestowania biblioteki tworzone są dwie zmienne typu **CStash**. Pierwsza z nich służy do przechowywania liczb całkowitych, a druga — tablic 80-znakowych:

```
//: C04:CLibTest.cpp
//{L} CLib
// Test biblioteki w stylu C
#include "CLib.h"
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

int main() {
    // Definicje zmiennych znajdują się
    // na początku bloku, tak jak w języku C:
    CStash intStash, stringStash;
    int i;
    char* cp;
    ifstream in;
    string line;
    const int bufsize = 80;
    // Trzeba pamiętać o inicjalizacji zmiennych:
    initialize(&intStash, sizeof(int));
    for(i = 0; i < 100; i++)
        add(&intStash, &i);
    for(i = 0; i < count(&intStash); i++)
        cout << "fetch(&intStash, " << i << ") - "
            << *(int*)fetch(&intStash, i)
            << endl;
    // Przechowywanie 80-znakowych łańcuchów:
    initialize(&stringStash, sizeof(char)*bufsize);
    in.open("CLibTest.cpp");
    assert(in);
```

```

while(getline(in, line))
    add(&stringStash, line.c_str());
i = 0;
while((cp = (char*)fetch(&stringStash, i++))!=0)
    cout << "fetch(&stringStash, " << i << ") = "
        << cp << endl;
cleanup(&intStash);
cleanup(&stringStash);
} //:-

```

Zgodnie z wymaganiem języka C, wszystkie zmienne są tworzone na początku zasięgu funkcji **main()**. Oczywiście, należy pamiętać o późniejszej inicjalizacji zmiennych typu **CStash** za pomocą wywołania funkcji **initialize()**. Jednym z problemów związanych z bibliotekami języka C jest to, że trzeba przekazać jej użytkownikom informacje o tym, jak ważne są funkcje inicjalizujące i porządkujące. Jeżeli nie zostaną one wywołane, spowoduje to mnóstwo kłopotów. Niestety, użytkownicy nie zawsze zastanawiają się nad tym, czy inicjalizacja i sprzątanie są konieczne. Wiedzą, co *sami* chcą osiągnąć, i nie przejmują się zbytnio poradami w rodzaju: „Poczekaj, musisz najpierw koniecznie zrobić to!”. Niektórzy użytkownicy są nawet znani z tego, że inicjalizują elementy struktury na własną rękę. Z pewnością nie ma w języku C mechanizmu, który by temu zapobiegał. Jak się później przekonamy, mechanizm taki istnieje jednak w C++.

Zmienna **iniStash** jest wypełniana liczbami całkowitymi, a **stringStash** — tablicami znakowymi. Tablice znakowe powstają w wyniku otwarcia pliku źródłowego **CLib-Test.cpp** i wczytania znajdujących się w nim wierszy do zmiennej łańcuchowej **line**, a następnie utworzenia za pomocą funkcji składowej **c_str()** wskaźnika do znakowej reprezentacji tej zmiennej.

Po wypełnieniu **iniStash** i **stringStash** wyświetlana jest ich zawartość. Zawartość **intStash** jest drukowana za pomocą pętli **for**, wykorzystującej funkcję **count()** do określenia zakresu licznika pętli. Natomiast do wydruku **stringStash** służy instrukcja **while**, przerwana wtedy, gdy funkcja **fetch()** zwraca wartość zero, oznaczającą przekroczenie zakresu indeksu elementu.

Warto również zwrócić uwagę na dodatkowe rzutowanie w instrukcji:

```
cp = (char*)fetch(&stringStash, i++)
```

Zastosowano je z powodu dokładniejszej kontroli typów w języku C++, niepozwalającej na przypisanie wartości typu **void*** zmiennej żadnego innego typu (w języku C jest to dozwolone).

Błędne założenia

Jest jeszcze jedna, bardziej istotna kwestia, o której należy wspomnieć, zanim omówimy ogólne problemy związane z tworzeniem bibliotek języka C. Zwróć uwagę na to, że plik nagłówkowy **CLib.h** musi być dołączony do każdego pliku, który odwołuje się do pliku **CStash**, ponieważ kompilator *nie jest w stanie* dowiedzieć się, jaką jest postać struktury. Jednakże kompilator *może* domyślić się, jak wyglądają funkcje — wygląda to na zaletę, ale okazuje się **bynajmniej** na zaletę, ale okazuje się **bynajmniej** gorszą.

Mimo że należy zawsze deklarować funkcje, dołączając odpowiedni plik nagłówkowy, deklaracje funkcji nie są w języku C konieczne. W języku C (ale nie w C++) można wywołać funkcję, która nie została zadeklarowana. Dobry kompilator powinien zgłosić ostrzeżenie, informujące o tym, że funkcję należy najpierw zadeklarować, ale nie wynika to z wymagań standardu języka C. Jest to niebezpieczna praktyka, ponieważ kompilator C może założyć, że lista argumentów funkcji, wywołanej z argumentem będącym liczbą całkowitą, zawiera liczbę całkowitą, nawet jeżeli w rzeczywistości znajduje się tam liczba zmennopozycyjna. Jak się przekonasz, wywołuje to niekiedy bardzo trudne do wykrycia błędy.

W języku C każdy plik zawierający implementację (czyli o rozszerzeniu .c) nazywany jest *jednostką translacji*. Oznacza to, że kompilator jest uruchamiany oddzielnie dla każdej jednostki translacji i w czasie pracy zwraca uwagę jedynie na bieżącą jednostkę. Tak więc informacja dostarczona przez dołączenie pliku nagłówkowego jest bardzo istotna, ponieważ zależy od niej właściwe rozumienie przez kompilator reszty programu. Szczególnie ważne są deklaracje zawarte w pliku nagłówkowym, ponieważ gdy dołączony zostanie plik nagłówkowy, kompilator będzie wiedział dokładnie, jakie działania ma wykonywać. Na przykład gdy plik nagłówkowy będzie zawierał deklarację **void func(float)**, wówczas kompilator będzie wiedział, że jeżeli funkcja ta zostanie wywołana z argumentem typu całkowitego, to powinien w czasie przekazywania argumentu przekształcić typ **int** we **float** (jest to nazywane *promocją*). Bez tej deklaracji, kompilator języka C założyłby po prostu, że istnieje funkcja **func(int)**, nie dokonałby promocji, a funkcji **func()** zostałyby „po cichu” przekazane niewłaściwe dane.

Kompilator tworzy dla każdej jednostki translacji odrębny plik wynikowy, posiadający rozszerzenie .o, .obj lub podobne. Pliki wynikowe wraz z kodem, niezbędnym do uruchomienia programu, muszą zostać połączone w wykonywalny program za pomocą programu łączącego. W trakcie łączenia należy określić wszystkie zewnętrzne odwołania. Na przykład w pliku **CLibTest.cpp** są zadeklarowane (czyli kompilator wie, jaką mają postać) i używane takie funkcje, jak **initialize()** i **fetch()**, ale nie zostały one w tym pliku zdefiniowane. Są natomiast zdefiniowane gdzie indziej — w pliku **CLib.cpp**. A zatem odwołania do funkcji znajdujących się w pliku **CLib.cpp** są odwołaniami zewnętrznymi. Łącząc z sobą pliki wynikowe, program łączący musi określić aktualne adresy wszystkich nierostrzygniętych odwołań zewnętrznych. Adresy te są umieszczane w programie wykonywalnym, zastępując zewnętrzne odwołania.

Warto podkreślić, że w języku C zewnętrzne odwołania, których poszukuje program łączący, są po prostu nazwami funkcji, poprzedzonymi na ogół znakiem podkreślenia. Tak więc program łączący jedynie dopasowuje nazwę funkcji występującej w miejscu wywołana do jej ciała, znajdującego się w pliku wynikowym. Jeżeli przypadkowo wywoła się funkcję, która zostanie zinterpretowana przez kompilator jako funkcja **func(int)**, a w jakimś innym pliku wynikowym znajdzie się ciało funkcji **func(float)**, to program łączący wykryje w obu tych miejscach nazwę **_func** i uzna, że wszystko jest w porządku. W miejscu wywołania **func()** umieści na stosie wartość typu **int**, a ciało funkcji **func()** będzie oczekivało, że na stosie znajduje się wartość typu **float**. Jeżeli funkcja czyta jedynie tę wartość, nie zmieniając jej, nie spowoduje to zniszczenia stosu. W istocie wartość typu **float**, odczytana ze stosu, może się nawet wydawać poprawna. Jednakże sytuacja jest wówczas jeszcze gorsza, ponieważ błąd taki znacznie trudniej znaleźć.

Naczym polega problem?

Potrafimy się przystosować — nawet do sytuacji, do których być może *nie powinniśmy*. Styl, w jakim napisano bibliotekę **CStash**, był chlebem powszednim dla programujących w języku C, ale jeżeli przyjrzeć się mu bliżej, może się okazać, że jest on... niewygodny. Używając biblioteki **CStash**, trzeba przekazywać adres struktury każdej funkcji wchodzącej w skład biblioteki. Podczas czytania programu mechanizmy obsługujące bibliotekę mylą się z wywoływanymi funkcjami, co jest irytujące w przypadku, gdy próbuje się zrozumieć, jak to wszystko działa.

Jednak jedną z największych trudności, wiążących się z używaniem w języku C bibliotek, jest problem *kolizji nazw*. Język C posiada tylko jedną przestrzeń nazw funkcji, tj. kiedy program łączący poszukuje nazwy funkcji, posługuje się jedną główną listą. Ponadto podczas przetwarzania jednostki translacji kompilator może pracować tylko z jedną funkcją o podanej nazwie.

Założymy, że zamierzasz nabyć dwie biblioteki, dostarczane przez dwie różne firmy; każda z nich zawiera strukturę, którą trzeba zainicjować i po której trzeba posprzątać. Obie firmy uznały, że odpowiednimi nazwami dla tych czynności są **initialize()** i **cleanup()**. Co zrobi kompilator języka C, jeżeli dołączysz pliki nagłówkowe obu tych bibliotek do pojedynczej jednostki translacji? Na szczęście, kompilator zgłosi błąd, informując o tym, że występuje niezgodność typów dwóch różnych list argumentów zadeklarowanych funkcji. Nawet jeśli oba pliki nagłówkowe nie zostaną włączone do tej samej jednostki translacji, kłopoty będzie miał nadal program łączący. Dobry program łączący zauważy, że w takim przypadku występuje konflikt nazw. Jednakże niektóre programy łączące przyjmą pierwszą napotkaną nazwę funkcji, przeszukując listy plików wynikowych w kolejności podanej na liściełączonych modułów (działanie takie może być nawet traktowane jako zaleta z uwagi na możliwość zastąpienia funkcji bibliotecznej przygotowaną samodzielnie wersją).

W żadnym z tych przypadków nie sposób używać dwóch bibliotek języka C, zawierających funkcje o takich samych nazwach. Aby rozwiązać ten problem, producenci bibliotek często poprzedzają wszystkie nazwy funkcji unikatowymi ciągami znaków. W taki sposób funkcje **initialize()** i **cleanup()** mogą stać się funkcjami **CStash_initialize()** oraz **CStash_cleanup()**. Jest to działanie logiczne, ponieważ uzupełnia nazwę struktury, na której działa funkcja, nazwą tej funkcji.

Pora na pierwszy krok w kierunku tworzenia klas w języku C++. Nazwy zmiennych zawartych w strukturach nie kolidują z nazwami zmiennych globalnych. Dlaczego więc nie wykorzystać tego w przypadku nazw funkcji, które operują na tych właśnie strukturach? Innymi słowy, dlaczego nie uczynić funkcji składową struktury?

Podstawowy obiekt

To właśnie pierwszy krok. Funkcje w języku C++ mogą być umieszczone **wewnętrz struktur jako „funkcje składowe”**. Poniżej pokazano, jak wygląda to w przypadku konwersji zrealizowanej w języku C wersji struktury **CStash** na napisaną w C++ strukturę **Stash**:

```
//: C04:CppLib.h
// Biblioteka w stylu C. przeniesiona do C++

struct Stash {
    int size;           // Wielkość każdego elementu
    int quantity;      // Liczba elementów pamięci
    int next;          // Następny pusty element
    // Dynamicznie przydzielana tablica bajtów:
    unsigned char* storage;
    // Funkcje!
    void initialize(int size);
    void cleanup();
    int add(const void* element);
    void* fetch(int index);
    int count();
    void inflate(int increase);
}; //:-
```

Po pierwsze, należy zwrócić uwagę na to, że nie ma tu deklaracji **typedef**. Kompilator C++, zamiast wymagać stosowania deklaracji **typedef**, przekształca na potrzeby programu nazwę struktury w nazwę nowego typu (tak jak nazwami typów są **int**, **char**, **float** i **double**).

Wszystkie dane **składowe** struktury pozostały takie same, jak poprzednio, ale w jej ciele pojawiły się dodatkowo funkcje. Ponadto należy zwrócić uwagę na to, że z funkcji zawartych w bibliotece, napisanej w języku C, usunięto pierwszy argument. W języku C++ nie wymaga się od programisty przekazywania adresu struktury wszystkim operującym na niej funkcjom, lecz wykonuje to za niego po kryjomu kompilator. Obecnie jedynie argumenty przekazywane funkcjom związane są z tym, co te funkcje *robią*, a nie z mechanizmem ich działania.

Trzeba zdawać sobie sprawę z tego, że kod funkcji jest w rzeczywistości taki sam, jak w przypadku wersji biblioteki napisanej w C. Liczba argumentów jest identyczna (nawet jeżeli nie jest widoczny przekazywany funkcjom adres struktury, to nadal tam istnieje), a ponadto każda funkcja ma tylko jedno ciało. Dzieje się tak, ponieważ zapis:

Stash A, B, C;

nie oznacza wcale, że dla każdej zmiennej istnieją odrębne wersje funkcji **add()**.

A zatem generowany kod jest niemal identyczny z tym, który należałoby napisać dla wersji biblioteki przygotowanej w języku C. Co ciekawsze, zawiera on również „ozdobne nazwy”, które należałoby wykorzystać, tworząc **Stash_initialize()**, **Stash_cleanup()** itd. Kompilator robi praktycznie to samo, gdy nazwa funkcji znajduje się wewnątrz struktury. Tak więc funkcja **initialize()**, zawarta w strukturze **Stash**, nie będzie kolidowała z funkcją o nazwie **initialize()**, znajdującej się wewnątrz innej struktury, ani nawet z globalną funkcją o tej samej nazwie. W większości przypadków nie trzeba się przejmować uzupełnieniami nazw funkcji — używa się po prostu zwykłych nazw. Czasami jednak zachodzi konieczność określenia, że funkcja **initialize()** należy do struktury **Stash**, a nie do jakiejś innej. W szczególności podczas definiowania funkcji należy w pełni oznaczyć, **która** jest ona funkcją. Aby to uzyskać, stosuje się w języku C++ operator (::), nazywany *operatorem zasięgu* (z uwagi na to, że nazwy mogą obecnie należeć do różnych zasięgów — zasięgu globalnego lub

zawartego w strukturze). Na przykład chcąc wskazać funkcję **initialize()**, należącą do struktury Stash, należy napisać **Stash::initialize(int size)**. Sposób zastosowania operatora zasięgu ujawnia się w definicjach funkcji:

```
//: C04:CppLib.cpp {0}
// Biblioteka w języku C, przeniesiona do C++
// Deklaracje struktury i funkcji:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Liczba elementów dodawanych
// w przypadku powiększenia pamięci:
const int increment = 100;

void Stash::initialize(int sz) {
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}

int Stash::add(const void* element) {
    if(next >= quantity) //Czy wystarczy pamięci?
        inflate(increment);
    // Kopiowanie elementu do pamięci.
    // począwszy od następnego wolnego miejsca:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return(next - 1); // Numer indeksu
}

void* Stash::fetch(int index) {
    // Check index boundaries:
    assert(0 <= index);
    if(index >= next)
        return 0; // Oznaczenie końca
    // Tworzenie wskaźnika do żądanego elementu:
    return &(storage[index * size]);
}

int Stash::count() {
    return next; // Liczba elementów w Stash
}

void Stash::inflate(int increase) {
    assert(increase > 0);
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Kopiowanie starego obszaru do nowego
    delete []storage; // Stary obszar pamięci
```

```
storage = b; // Wskaźnik do nowego obszaru
quantity - newQuantity;
}

void Stash::cleanup() {
    if(storage != 0) {
        cout << "zwalnianie pamięci" << endl;
        delete []storage;
    }
} // :~
```

Istnieje szereg innych różnic pomiędzy C i C++. Po pierwsze, kompilator *wymaga* deklaracji zawartych w plikach nagłówkowych. W języku C++ nie można wywołać **niezadeklarowanej** uprzednio funkcji, ponieważ spowoduje to zgłoszenie błędu przez kompilator. Jest to istotna metoda zapewnienia spójności pomiędzy wywołaniem funkcji a jej definicją. Wymuszając deklarację funkcji przed jej wywołaniem, kompilator w praktyce zapewnia, że deklaracja ta zostanie dokonana przez dołączenie pliku nagłówkowego. Jeżeli w miejscu, w którym zdefiniowane są funkcje, dołączony zostanie ten sam plik nagłówkowy, kompilator upewni się, że zawarte w pliku nagłówkowym deklaracje są zgodne z definicjami funkcji. Oznacza to, że plik nagłówkowy staje się uwierzytelnioną składnicą deklaracji funkcji, gwarantując tym samym, że funkcje te będą wykorzystywane w jednolity sposób we wszystkich jednostkach translacji projektu.

Oczywiście, funkcje globalne mogą być nadal deklarowane „ręcznie” w każdym miejscu, w którym są one definiowane i wykorzystywane (*jest to na tyle niewygodne, że w rezultacie staje się niezwykle rzadkie*). Jednakże struktury muszą być zawsze zadeklarowane, zanim będą definiowane lub używane, a najwygodniejszym miejscem, w którym można umieścić definicję struktury, jest plik nagłówkowy — z wyjątkiem tych struktur, które sęcelowo ukryte w pliku.

Jak można zauważyc, wszystkie funkcje składowe wyglądają niemal tak samo jak wówczas, gdy były funkcjami języka C, z wyjątkiem wyszczególnienia zasięgu oraz faktu, że pierwszy parametr, występujący w wersji biblioteki napisanej w języku C, nie jest już przekazywany jawnie. Oczywiście, parametr ten nadal istnieje, ponieważ funkcje muszą działać na konkretnej zmiennej będącej strukturą. Należy jednak zwrócić uwagę na to, że w obrębie funkcji składowych pominięto również wybór elementów składowych struktury! A zatem zamiast **s->size = sz** piszemy **size = sz** — pominając irytujący przedrostek **s->**, który tak naprawdę nie wnosił żadnego nowego znaczenia. Kompilator języka C++ najwyraźniej nas w tym wyręcza. W rzeczywistości tworzy on „ukryty” pierwszy argument funkcji (zawierający adres struktury, który był uprzednio przekazywany jawnie) i stosuje selektor składowej przy każdym odwołaniu do zawartych w strukturze elementów danych. Oznacza to, że będąc wewnętrz funkcji składowej jakiejś struktury można odwoływać się do jej składowych (włączając w to składowe będące funkcjami) poprzez podanie ich nazw. Zanim kompilator **zacznie** poszukiwać nazwy wśród nazw globalnych, przeszuka najpierw lokalne nazwy struktury. Przekonasz się, że dzięki temu nie tylko łatwiej napiszesz programy, ale będą one również znacznie prostsze w czytaniu.

Co jednak zrobić w przypadku, gdy *chce się* użyć adresu struktury? W napisanej w **języku C** wersji biblioteki było to proste, ponieważ pierwszym argumentem każdej funkcji był wskaźnik **CStasch*** o nazwie *s*. W języku C++ okazuje się to jeszcze bardziej spójne. Istnieje specjalne słowo kluczowe **this**, zwracające adres struktury. Jest to odpowiednik argumentu **,s**, używanego w wersji biblioteki w języku C. Dlatego też można powrócić do stylu języka C, pisząc:

`this->size - Size;`

Kod wygenerowany przez kompilator jest dokładnie taki sam, nie ma więc potrzeby używania w takich przypadkach słowa kluczowego **this**. Można niekiedy natrafić na program, którego autor wszędzie stosuje konstrukcję **this->**, ale nie wnosi ona niczego nowego do treści programu i wskazuje często na brak doświadczenia programisty. Słowo kluczowe **this**, na ogół używane rzadko, jest dostępne w przypadku, gdy zachodzi potrzeba jego użycia (słowo **this** jest wykorzystywane w niektórych przykładach w dalszej części książki).

Została jeszcze jedna kwestia, o której trzeba wspomnieć. W języku C można przypisać wartość typu **void*** każdemu innemu typowi, jak w przykładzie poniżej:

```
int i = 10;  
void* vp = &i; // Możliwe zarówno w C, jak i w C++  
int* ip = vp; // Dopuszczalne tylko w C
```

i nie spowoduje to sprzeciwu kompilatora. Jednakże w języku C++ wyrażenie takie nie jest dopuszczalne. Dlaczego? Ponieważ język C nie cechuje drobiazgowości w sprawach typów, pozwala on na przypisanie wskaźnika nieokreślonego typu wskaźnikowi, którego typ został określony. W języku C++ nie jest to możliwe. Ponieważ pojęcie typu jest w języku C++ pojęciem kluczowym, kompilator zgłosi swój sprzeciw w przypadku naruszenia w jakikolwiek sposób informacji o typie. Było to zawsze ważne, lecz w języku C++ ma szczególne znaczenie, ponieważ struktury zawierają funkcje składowe. Gdyby w języku C++ można było bezkarnie zmieniać typy wskaźników do struktur, mogłoby się to nawet skończyć katastrofą — wywołaniem funkcji składowej **struktury**, której w ogóle w tej strukturze nie ma! Dlatego też, mimo że język C++ dopuszcza przypisanie dowolnego typu wskaźnika wskaźnikowi typu **void*** (był to pierwotny cel wprowadzenia typu **void*** o takim rozmiarze, by mógł przechować wskaźnik dowolnego typu), *nie pozwoli* on na przypisanie wskaźnika typu **void*** wskaźnikowi jakiegokolwiek innego typu. W takich przypadkach, dla zaszygnalizowania czytelnikowi oraz kompilatorowi, że naprawdę chce się traktować wskaźnik jako wskaźnik typu docelowego, trzeba zawsze używać rzutowania.

Wiąże się z tym pewna interesująca kwestia. Jednym z istotnych celów języka C++ jest możliwość komplikacji jak największej części istniejącego kodu języka C, co pozwala na łatwe przejście do nowego języka. Nie oznacza to jednak, że każda konstrukcja dostępna w języku C będzie automatycznie dopuszczalna w C++. Istnieje wiele takich działań, akceptowanych przez kompilator języka C, które są w istocie niebezpieczne i mogą być przyczyną błędów (zostaną opisane w dalszej części książki). W takich sytuacjach kompilator C++ zgłasza błędy i ostrzeżenia. Jest to w większym stopniu korzyścią niż utrudnieniem. Istnieje bowiem wiele sytuacji, w których na próżno próbuje *się* wytropić przyczynę błędu w programie napisanym w języku C, dopóki nie skompiluje się powtórnie programu w C++ i kompilator nie wskaże źródła

problemu! W języku C często okazuje się, że program można co prawda skompilować, ale w następnej kolejności trzeba doprowadzić do tego, by działał. Program, który kompliuje się poprawnie w C++, zazwyczaj również działa! Dzieje się tak, ponieważ język ten jest znacznie bardziej bezwzględny w stosunku do typów.

W programie testowym, prezentującym wykorzystanie napisanej w C++ wersji biblioteki **Stash**, można dostrzec wiele nowych elementów:

```
//: C04:CppLibTest.Cpp
//{L} CppLib
// Test biblioteki w C++
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
    Stash intStash;
    intStash.initialize(sizeof(int));
    for(int i = 0; i < 100; i++)
        intStash.add(&i);
    for(int j = 0; j < intStash.count(); j++)
        cout << "intStash.fetch(" << j << ") = "
            << *(int*)intStash.fetch(j)
            << endl;
    // Przechowywanie 80-znakowych łańcuchów:
    Stash stringStash;
    const int bufsize = 80;
    stringStash.initialize(sizeof(char) * bufsize);
    ifstream in("CppLibTest.cpp");
    assure(in, "CppLibTest.cpp");
    string line;
    while(getline(in, line))
        stringStash.add(line.c_str());
    int k = 0;
    char* cp;
    while((cp =(char*)stringStash.fetch(k++)) != 0)
        cout << "stringStash.fetch(" << k << ") = "
            << cp << endl;
    intStash.cleanup();
    stringStash.cleanup();
} //:-
```

Łatwo dostrzec, że wszystkie zmienne zostały zdefiniowane „w locie” (co opisano w poprzednim rozdziale). Oznacza to, że są one zdefiniowane w dowolnym miejscu zasięgu, a miejsce ich definicji nie jest ograniczone do jego początku, jak w przypadku języka C.

Kod programu jest dość podobny do kodu zawartego w pliku **CLibTest.cpp**, lecz w czasie wywoływanego funkcji używany jest operator selekcji składowej „.”, występujący po nazwie zmiennej. Składnię można uznać za wygodną, ponieważ naśladuje ona wybór zmiennej, stanowiącej składową struktury. Różnica polega na tym, że jest to funkcja, więc posiada ona listę argumentów.

Oczywiście, wywołanie generowane przez kompilator, w rzeczywistości przypomina bardziej wywołanie funkcji w wersji biblioteki, napisanej w C. A zatem, uwzględniając uzupełnienia nazwy funkcji i przekazanie parametru `this`, wywołanie funkcji `int Stash.initialize(sizeof(int), 100)` staje się czymś w rodzaju `Stash_initialize(&intStash, sizeof(int), 100)`. Aby poznać, jaki kryje się pod tym mechanizmem, pamiętaj, że `cstdlib` — oryginalny kompilator języka C++ firmy AT&T — tworzył w wyniku komplikacji kod w języku C, który był następnie kompilowany przez kompilator tego języka. Oznacza to, że kompilator `cstdlib` może być szybko przeniesiony na dowolny komputer posiadający kompilator języka C, przyczyniając się w ten sposób do szybkiego rozprzestrzeniania się technologii komplikacji C++. Ponieważ jednak kompilator C++ musi być w stanie generować program w języku C, istnieje jakiś sposób umożliwiający prezentację składni języka C++ w języku C (niektóre kompilatory nadal pozwalają na generację kodu w języku C).

Istnieje jeszcze jedna różnica w stosunku do kodu zawartego w `CLibTest.cpp` — jest nią dołączenie pliku nagłówkowego `require.h`. Jest to plik utworzony przez autora na potrzeby książki, dokonujący bardziej wyrafinowanej kontroli błędów niż ta realizowana przez funkcję `assert()`. Zawiera on szereg funkcji, między innymi zastosowaną tutaj funkcję `assure()`, używaną w stosunku do plików. Funkcja ta sprawdza, czy plik został pomyślnie otwarty i w przypadku niepowodzenia wyświetla na standardowym wyjściu komunikat, informujący, że pliku nie można otworzyć (dlatego też wymaga podania nazwy tego pliku jako drugiego argumentu), oraz powoduje zakończenie pracy programu. Funkcje zawarte w pliku nagłówkowym `require.h` będą używane w dalszej części książki, szczególnie do upewnienia się, że program został wywołany z odpowiednią liczbą argumentów oraz że używane przez niego pliki zostały poprawnie otwarte. Zastępują one powtarzający się i rytmującą kod, sprawdzający wystąpienie błędów, dostarczając ponadto naprawdę użytecznych komunikatów o błędach. Funkcje te zostaną szczegółowo opisane w dalszej części książki.

Czym są obiekty?

Po prezentacji wstępnego przykładu pora zrobić krok wstecz i przyjrzeć się pewnym zagadnieniom, związanym z terminologią. Włączenie funkcji do struktur jest istotą wkładu, wniesionego przez język C++ do języka C. Zarazem został wprowadzony zupełnie nowy sposób ujmowania struktur jako pojęć. W języku C struktury **stanowią jedynie** zbiorowiska danych — sposób upakowania, umożliwiający ich łączne traktowanie. Trudno jednak myśleć o nich inaczej niż jako o konwencji stosowanej przez programistów. Funkcje przetwarzające te struktury znajdują się wszędzie. Jednakże gdy razem z danymi spakowane są funkcje, struktura staje się nowym tworem, zdolnym do opisu zarówno cech (jak robią to struktury w języku C), jak i zachowań. Koncepcja obiektów — samodzielnych, ograniczonych jednostek, potrafiących pamiętać i działać, nasuwa się sama.

W języku C++ obiekty są po prostu zmiennymi, a ich najprostszą **definicją jest „obszar pamięci”** (co jest bardziej precyzyjną formą **stwierdzenia**, że „każdy obiekt musi mieć unikatowy identyfikator”, którym w przypadku C++ jest unikatowy adres pamięci). To miejsce, w którym mogą być przechowywane dane, z czego pośrednio wynika, że istnieją również operacje, które można na tych danych wykonać.

Niestety, w dziedzinie tych pojęć nie ma całkowitej zgodności wśród języków programowania, mimo że są one raczej powszechnie akceptowane. Można się spotkać z rozbieżnymi opiniami na temat tego, czym jest obiektowy język programowania, chociaż wydaje się to obecnie dość dobrze określone. Istnieją języki *bazujące na obiektach*, co oznacza, że posiadają one — podobnie jak język C++ — struktury zawierające funkcje, z czym zetknęliśmy się do tej pory. Jest to jednak jedynie wycinek perspektywy, obejmującej języki obiektowe i języki, które poprzestają na umieszczeniu funkcji w strukturach danych. Określone są jako języki bazujące na obiektach, a nie języki obiektowe.

Tworzenie abstrakcyjnych typów danych

Możliwość łączenia danych z funkcjami pozwala na utworzenie nowych typów danych. Często nazywa się ją *kapsułowaniem*¹ (ang. *encapsulation*). Istniejące typy danych mogą składać się z wielu połączonych ze sobą informacji. Na przykład liczba typu **float** zawiera wykładnik, mantysę oraz bit znaku. Można zażądać dodania jej do innej liczby zmiennopozycyjnej, liczby całkowitej itd. Posiada więc ona zarówno pewne dane, jak i określone zachowanie.

Definicja struktury **Stash** tworzy nowy typ danych. Można wykonywać na nim operacje **add()**, **fetch()** oraz **inflate()**. Egzemplarz tego typu tworzy się wpisując **Stash s**, podobnie jak pisząc **float f** tworzy się zmienną zmiennopozycyjną. Typ **Stash** posiada zarówno cechy, jak i zachowanie. Mimo że zachowuje się on jak prawdziwy, wbudowany typ danych, określamy go mianem *abstrakcyjnego typu danych*, prawdopodobnie z uwagi na to, że pozwala na przedstawienie w przestrzeni rozwiązania abstrakcyjnej postaci pojęcia, pochodzącego z przestrzeni problemu. Ponadto kompilator języka C++ traktuje go jak nowy typ danych i w gdy funkcja wymaga argumentu typu **Stash**, kompilator upewnia się, że funkcji została przekazana taka właśnie wartość. A zatem w stosunku do abstrakcyjnych typów danych (nazywanych czasami *typami zdefiniowanymi przez użytkownika*) kontrola typów przeprowadzana jest na takim samym poziomie, jak w przypadku typów wbudowanych.

Różnicą jest natomiast natychmiast widoczna w sposobie, w jaki wykonywane są operacje na obiektach. Zapis **obiekt.funkcjaSkładowa(listaArgumentów)** oznacza: „wywołaj funkcję składową obiektu”. Jednak w używanej potocznie terminologii obiektowej jest to również określone „wysyłaniem komunikatu do obiektu”. A zatem dla zmiennej, zdefiniowanej jako **Stash s**, instrukcja **s.add(&i)** oznacza „wysłanie do zmiennej s komunikatu, by wykonała na sobie operację **add()**”. Właściwie programowanie obiektowe można streszczyć w kilku słowach jako *wysyłanie komunikatów do obiektów*. I faktycznie — tworzymy grupę obiektów, a następnie wysyłamy do nich komunikaty. Cała sztuka polega oczywiście na tym, by określić, jakie są te obiekty i te komunikaty, ale kiedy po wykonaniu tego implementacja programu w języku C++ okazuje się zaskakująco łatwa.

Pojęcie to wywołuje spory. Niektórzy używają go w sposób przedstawiony w tym miejscu, podczas gdy inni stosują go w odniesieniu do *kontroli dostępu*, omawianej w następnym rozdziale.

Szczegóły dotyczące obiektów

W trakcie seminariów często pada pytanie: „jak duże są obiekty i jak one wyglądają?”. Odpowiedź brzmi: „są podobne do struktur języka C”. W istocie kod generowany przez kompilator języka C w przypadku struktur (bez dodatków właściwych językowi C++) wygląda zazwyczaj *dokładnie* tak samo, jak kod wygenerowany przez kompilator C++. Uspokaja to tych programistów języka C, którzy uzależniają swój kod od szczegółów dotyczących wielkości oraz rozmieszczenia danych w pamięci i z jakiegoś powodu, zamiast używać identyfikatorów, odwołują się bezpośrednio do poszczególnych bajtów tworzących strukturę (poleganie na określonej wielkości i rozmieszczeniu danych w pamięci prowadzi do tworzenia nieprzenośnych programów).

Wielkość struktury jest łączną wielkością wszystkich jej składowych. Czasami, gdy struktury umieszczane są w pamięci, dodawane są do nich dodatkowe bajty, pozwalające na odpowiednie rozmieszczenie dzielących ich granic, dzięki czemu możliwe jest uzyskanie większej szybkości wykonania programu. W rozdziale 15. pokażemy, jak w pewnych przypadkach do struktur dodawane są „ukryte” wskaźniki, ale tymczasem kwestia ta zostanie pominięta.

Wielkość struktury można określić za pomocą operatora **sizeof**. Oto krótki przykład:

```
//: C04:S1zeof.cpp
// Wielkości struktur
#include "CLib.h"
#include "CppLib.h"
#include <iostream>
using namespace std;

struct A {
    int i[100];
};

struct B {
    void f();
};

void B::f() {}

int main() {
    cout << "wielkosc struktury A = " << sizeof(A)
        << " bajtow" << endl;
    cout << "wielkosc struktury B = " << sizeof(B)
        << " bajtow" << endl;
    cout << "wielkosc CStash w C = "
        << sizeof(CStash) << " bajtow" << endl;
    cout << "wielkosc Stash w C++ = "
        << sizeof(Stash) << " bajtow" << endl;
} //:~
```

W przypadku mojego komputera (wyniki uzyskane na innych komputerach mogą być odmienne) pierwsza instrukcja wydrukowała wartość 200, ponieważ każda liczba całkowita zajmuje dwa bajty. Struktura B stanowi pewną anomalię, gdyż nie zawiera danych składowych. To niedopuszczalne w języku C, w przeciwieństwie do C++, w którym

jest potrzebna możliwość utworzenia struktury spełniającej jedyne zadanie — określenia zasięgu nazw funkcji. Może być jednak zaskoczeniem, że informacja wydrukowana przez drugą instrukcję nie jest zerem. We wczesnych wersjach języka wielkość tego typu struktur była zerowa, ale prowadziło to do kłopotliwych sytuacji, związanych z tworzeniem obiektów tego typu — posiadały one taki sam adres, jak obiekty utworzone bezpośrednio po nich i w związku z tym niczym się nie różniły. Jedną z podstawowych zasad dotyczących obiektów jest ta, że muszą one posiadać unikatowy adres. Obiekty niezawierające składowych będących danymi mają zawsze pewną minimalną, niezerową wielkość.

Dwie ostatnie instrukcje `sizeof` pokazują, że wielkość struktury w języku C++ jest taka samajak wielkość odpowiadającej jej wersji w C. Język C++ stara się nie dokładać żadnego niepotrzebnego narzutu.

Zasady używania plików nagłówkowych

Gdy tworzysz strukturę zawierającą dane składowe, tworzysz zarazem nowy typ danych. Zazwyczaj chcesz, by typ ten był dostępny zarówno dla ciebie, jak i dla innych. W dodatku zamierzasz oddzielić interfejs (deklaracje) od implementacji (definicji funkcji składowych), tak by można było zmienić implementację bez konieczności powtórnej kompilacji całego systemu. Osiagniesz to, umieszczając deklaracje dotyczące tworzonego typu w pliku nagłówkowym.

Kiedy zaczynałem uczyć się programowania w języku C, pliki nagłówkowe były dla mnie czymś tajemniczym. Autorzy wielu książek na temat C zdawali się nie przywiązywać do nich wagi, a kompilatory nie wymagały deklaracji funkcji. Pliki nagłówkowe wydawały się zatem w większości przypadków nieobowiązkowe — z wyjątkiem sytuacji, gdy deklarowane były struktury. W języku C++ używanie plików nagłówkowych staje się zupełnie oczywiste. Są one niezbędne do łatwego tworzenia programów i umieszcza się w nich ściśle określone informacje — deklaracje. Pliki nagłówkowe informują kompilator o zawartości bibliotek. Biblioteki można używać nawet wówczas, gdy posiada się jedynie plik nagłówkowy oraz plik będący programem wynikowym lub biblioteką — nie jest potrzebny do tego plik `cpp`, zawierający kod źródłowy. W plikach nagłówkowych przechowywana jest specyfikacja interfejsu.

Mimo że nie jest to wymuszone przez kompilator, najlepszym sposobem budowania w języku C dużych projektów jest wykorzystanie bibliotek — połączenie związanych ze sobą funkcji w jeden plik wynikowy lub bibliotekę oraz użycie pliku nagłówkowego, zawierającego wszystkie deklaracje funkcji. W języku C++ jest to *konieczne* — do biblioteki w C można powzruszać różne funkcje, ale abstrakcyjne typy danych języka C++ określają funkcje powiązane ze sobą możliwością wspólnego dostępu do danych zawartych w strukturze. Każda funkcja składowa musi zostać zadeklarowana w deklaracji struktury — nie można umieścić jej w żadnym innym miejscu. A zatem o ile używanie bibliotek funkcji było w języku C wspierane, o tyle jest już ono w języku C++ zinstytucjonalizowane.

Znaczenie plików nagłówkowych

W przypadku używania funkcji pochodzącej z biblioteki języka C daje możliwość pominięcia pliku nagłówkowego i zadeklarowania tej funkcji na własną rękę. W przeszłości wielu programistów tak robiło — po to, by przypieszyć choć trochę działanie kompilatora dzięki uniknięciu konieczności otwierania i dołączania pliku (nie ma to znaczenia w przypadku nowoczesnych kompilatorów). Poniższy przykład prezentuje deklarację funkcji `printf()` w języku C (pochodzącej z pliku `<stdio.h>`), będącą wynikiem skrajnego lenistwa:

```
printf(...);
```

Wielokropki określają *zmienną listę argumentów*², co oznacza: funkcja `printf()` posiada jakieś argumenty, z których każdy jest jakiegoś typu, ale należy to zignorować, akceptując w jej wywołaniu dowolne argumenty. Używając takiej deklaracji, zawieszasie wszelką kontrolę typów argumentów funkcji.

Taka praktyka może być źródłem trudnych do uchwycenia problemów. Jeżeli funkcje deklarowane są „ręcznie”, to w jednym z plików może zdarzyć się pomyłka. Ponieważ kompilator wykrywa w tym pliku jedynie wprowadzoną „ręcznie” deklarację, to może dostosować się do popełnionego błędu. Program zostanie co prawda połączony poprawnie, lecz użycie funkcji w tym właśnie pliku będzie błędne. Jest to trudny do znalezienia błąd, którego można łatwo uniknąć, używając pliku nagłówkowego.

Umieszczenie deklaracji wszystkich funkcji w pliku nagłówkowym, a następnie dołączenie tego pliku wszędzie tam, gdzie funkcje te są używane, a także w miejscu ich definicji, zapewnia stosowanie w całym systemie jednolitych deklaracji. Dołączenie pliku nagłówkowego do pliku, zawierającego definicje funkcji, gwarantuje również zgodność deklaracji z definicjami.

W przypadku deklaracji struktury w pliku nagłówkowym w języku C++ plik ten *musi* zostać dołączony wszędzie tam, gdzie struktura ta jest używana, oraz w miejscu, w którym zostały zdefiniowane funkcje składowe struktury. Kompilator języka C++ zgłosi błąd w przypadku próby wywołania zwykłej funkcji, a także wywołania lub definicji funkcji składowej, jeżeli nie zostały one uprzednio zadeklarowane. Wspierając właściwe stosowanie plików nagłówkowych, język zapewnia spójność w obrębie bibliotek, a także ogranicza liczbę błędów, wymuszając stosowanie w każdym miejscu jednolitego interfejsu.

Plik nagłówkowy jest kontraktem pomiędzy tobą a użytkownikiem biblioteki. Opisuje on twoje struktury danych, a także określa argumenty wywoływanych funkcji oraz wartości przez nie zwracane. Głosi on: „oto, co robi moja biblioteka”. Niektóre informacje zawarte w pliku nagłówkowym potrzebne są użytkownikowi opracowującemu własną aplikację, a wszystkie z nich niezbędne są kompilatorowi do wygenerowania poprawnego kodu. Użytkownik struktury dołącza po prostu plik nagłówkowy, tworzy obiekty (egzemplarze) tej struktury i dołącza do swojego programu odpowiedni moduł wynikowy lub bibliotekę (tj. skompilowany kod).

² Aby napisać definicję funkcji, używającej zmiennej liczby argumentów, trzeba użyć zbioru makroinstrukcji `varargs`, ale w języku C++ powinno się tego unikać. Szczegóły dotyczące makroinstrukcji `varargs` można znaleźć w dokumentacji kompilatora.

Kompilator wspomaga ten kontrakt, wymagając od ciebie zadeklarowania wszystkich struktur i funkcji, zanim zostaną one użyte oraz — w przypadku funkcji składowych — zanim zostaną zdefiniowane. Należy zatem umieścić deklarację w pliku nagłówkowym i dołączyć go do pliku, w którym zdefiniowane zostały funkcje składowe, a także do pliku (plikach), w których są one używane. Ponieważ w całym systemie dołączany jest ten sam plik nagłówkowy, opisujący swoją bibliotekę, kompilator może zapewnić spójność systemu i zapobiec błędem.

Aby poprawnie zorganizować swój kod i utworzyć odpowiednie pliki nagłówkowe, należy zwrócić uwagę na niektóre kwestie. Pierwszą z nich jest decyzja, jakie informacje należy umieścić w plikach nagłówkowych. Podstawowa reguła głosi, że powinny się w nich znajdować *wyłącznie deklaracje*, czyli informacje dla kompilatora; nie mogą one natomiast zawierać niczego, co przydzielałoby pamięć — generując kod lub tworząc zmienne. Jest to istotne, ponieważ pliki nagłówkowe są zazwyczaj dołączane do wielujednostek translacji, wchodzących w skład projektu i w przypadku gdy pamięć dla tego samego identyfikatora jest przydzielana w więcej niż jednym miejscu, powoduje to zgłoszenie przez program łączący błędu wielokrotnej definicji (jest to *reguła jednej definicji* języka C++ — każdą rzecz można zadeklarować do wolną liczbę razy, ale w programie może wystąpić tylko jedna jej definicja).

Od reguły tej istnieją jednak odstępstwa. Jeżeli zdefiniuje się w pliku nagłówkowym **zmienną**, której jest „statyczna w obrębie pliku” (jest widoczna tylko w jednym pliku), to w projekcie powstanie wiele jej egzemplarzy, ale podczas łączenia nie wystąpi kolizja nazw **zmiennych**³. Na ogół nie należy umieszczać w pliku nagłówkowym niczego, co powodowałoby dwuznaczność podczas łączenia.

Problem wielokrotnych deklaracji

Druga kwestia, związana z plikami nagłówkowymi, polega na tym, że w przypadku złożonego programu, po umieszczeniu deklaracji w pliku nagłówkowym, istnieje możliwość, że plik ten zostanie dołączony więcej niż jednokrotnie. Dobrym tego przykładem są strumienie wejścia-wyjścia. Do deklaracji struktury, realizującej funkcje wejścia-wyjścia, może być dołączony jeden lub większa liczba plików nagłówkowych, zawierających deklaracje strumieni. Jeżeli plik `cpp`, nad którym pracujesz, wykorzystuje większą liczbę struktur (dołączając pliki nagłówkowe każdej z nich), istnieje możliwość dołączenia pliku nagłówkowego `<iostream>` więcej niż jeden raz i tym samym niebezpieczeństwo powtórnej deklaracji strumieni.

Kompilator traktuje powtórne deklaracje struktur (dotyczy to zarówno struktur, jak i klas) jako błąd, ponieważ w przeciwnym przypadku pozwoliłoby to na używanie tej samej nazwy w odniesieniu do różnych typów. Aby zapobiec błędowi związanemu z wielokrotnym dołączaniem plików nagłówkowych, należy wbudować w nie pewną dozę inteligencji, używając do tego celu preprocesora (standardowe pliki nagłówkowe C++, takie jak `<iostream>`, posiadają już tę „inteligencję”).

 Jednak w standardzie C++, używanie elementów statycznych w obrębie pliku nie jest wskazane.

Zarówno język C, jak i C++ pozwalają na powtórную deklarację funkcji, pod warunkiem, że obie deklaracje są ze sobą zgodne; nie zezwala jednak wcale na powtórную deklarację struktur. W języku C++ zasada ta jest szczególnie ważna z uwagi na to, że gdyby kompilator pozwolił na powtórную deklarację struktury, to jeśli deklaracje te różniły się od siebie, nie wiedziałby, której z nich ma użyć.

Problem związany z powtórą deklaracją zyskuje jeszcze znaczenia w języku C++, ponieważ wszystkie typy danych (struktury zawierające funkcje) posiadają na ogół własne pliki nagłówkowe. W przypadku tworzenia nowego typu danych na podstawie typu już istniejącego konieczne jest dołączenie pliku nagłówkowego typu podstawowego do pliku nagłówkowego tworzonego typu. W każdym z tworzących projekt plików `cpp` możliwe jest dołączenie wielu plików, dołączających z kolei te same pliki nagłówkowe. Podczas pojedynczej komplikacji kompilator odczytuje wielokrotnie te same pliki nagłówkowe. Jeżeli temu się nie zapobiegnie, kompilator będzie napotykał powtórne deklaracje tej samej struktury, zgłaszając błędy. Aby rozwiązać ten problem, trzeba poszerzyć wiadomości na temat preprocesora.

Dyrektyny preprocesora `#define`, `#ifdef` i `#endif`

Dyrektyny preprocesora `#define` można użyć do utworzenia znaczników dostępnych w czasie komplikacji. Istnieją dwie możliwości: można po prostu poinformować preprocesor, że definiowany jest znacznik nieposiadający żadnej określonej wartości:

```
#define FLAG
```

lub nadać mu wartość (co jest w języku C typowym sposobem tworzenia stałych):

```
#define PI 3.14159
```

W każdym z powyższych przypadków preprocesor może sprawdzić, czy etykieta została zdefiniowana:

```
#ifdef FLAG
```

Wyrażenie to zwróci wartość **prawdziwą**, w wyniku czego kod następujący po dyrektywie `#ifdef` zostanie włączony do kodu przekazywanego kompilatorowi. Włączanie to kończy się w chwili napotkania przez preprocesor dyrektywy:

```
#endif
```

lub

```
#endif // FLAG
```

Niedozwolone jest dopisanie po dyrektywie `#endif` — w tym samym wierszu — **czegos**, co nie jest komentarzem, mimo że niektóre kompilatory mogły akceptować. Parę dyrektyw `#ifdef/#endif` mogą być wewnątrz siebie zagnieżdżane.

Przeciwieństwem `#define` jest `#undef`, w wyniku którego dyrektywa `#ifdef`, używająca tego samego identyfikatora, zwróci wartość negatywną. Dyrektywa `#undef` może również spowodować zaprzestanie używania przez kompilator makroinstrukcji. Przeciwieństwem

`#ifndef` jest `#ifndef`, zwracająca wartość pozytywną, gdy podana etykieta nie została zdefiniowana (jej właśnie będziemy używać w plikach nagłówkowych).

Są również inne użycieczne właściwości preprocesorajęzyka C. Ich pełny wykaz znajduje się w dokumentacji używanego kompilatora.

Standard plików nagłówkowych

W przypadku każdego pliku nagłówkowego zawierającego strukturę należy najpierw ustalić, czy plik ten nie został już dołączony do bieżącego pliku `cpp`. Aby to uczynić, trzeba sprawdzić wartość znacznika preprocesora. Jeżeli znacznik ten nie jest zdefiniowany, oznacza to, że bieżący plik nagłówkowy nie został jeszcze dołączony. Należy zatem zdefiniować znacznik (dzięki czemu struktura nie będzie mogła zostać powtórnie zadeklarowana) i zadeklarować strukturę. Jeżeli natomiast znacznik jest już zdefiniowany, oznacza to, że typ ten został już zadeklarowany, więc można po prostu pominąć deklarujący go kod. Plik nagłówkowy powinien więc wyglądać następująco:

```
#ifndef HEADER_FLAG  
#define HEADER_FLAG  
// Deklaracja typu...  
#endif // HEADER_FLAG
```

Gdy plik nagłówkowy jest dołączany po raz pierwszy, jego zawartość (włączając w to deklarację typu) zostanie dołączona przez preprocesor. We wszystkich następnych dołączeniach pliku — w obrębie tej samej jednostki komplikacji — deklaracja typu zostanie zignorowana. Nazwa `HEADER_FLAG` może być dowolną unikatową nazwą, ale niezawodnym standardem, który warto naśladować, jest zapisanie nazwy pliku wielkimi literami i zastąpienie występujących w niej kropek znakami podkreślenia (**jednakże** znajdujące się na początku nazwy znaki podkreślenia są zarezerwowane dla nazw systemowych). Ilustruje to poniższy przykład:

```
/: C04:Simple.h  
// Prosty nagłówek, zapobiegający powtarnej definicji  
#ifndef SIMPLE_H  
#define SIMPLE_H  
  
struct Simple {  
    int i,j,k;  
    initialize() { i = j = k = 0; }  
};  
#endif // SIMPLE_H ///:-
```

Mimo że tekst `SIMPLE_H`, znajdujący się po dyrektywie `#endif`, jest oznaczony jako komentarz i tym samym jest on ignorowany przez preprocesor, przydaje się do celów dokumentacyjnych.

Przedstawione powyżej dyrektywy preprocesora, uniemożliwiające wielokrotne dołączanie plików nagłówkowych, określone są często mianem *strażników dołączania*.

Przestrzenie nazw w plikach nagłówkowych

Nietrudno zauważyć, że niemal we wszystkich plikach, zawartych w książce, używa się naj częściej dyrektywy `using`, występującą zazwyczaj w postaci:

```
using namespace std;
```

Ponieważ `std` jest przestrzenią nazw, obejmującą całą standardową bibliotekę C++, taki sposób wykorzystania dyrektywy `using` pozwala na używanie bez kwalifikatorów nazw zawartych w standardowej bibliotece C++. Jednakże dyrektywa `using` nie występuje nigdy w plikach nagłówkowych (przynajmniej nie zewnętrz zasięgów). Dyrektywa `using` wyklucza bowiem ochronę danej przestrzeni nazw, a jej działanie rozciąga się do końca bieżącej jednostki komplilacji. Jeżeli dyrektywę tę umieści się poza jakimkolwiek zasięgiem, w pliku nagłówkowym, będzie to oznaczało, że utrata „ochrony przestrzeni nazw” nastąpi w każdym pliku, do którego zostanie dołączony ten plik nagłówkowy, a zatem często również w innych plikach nagłówkowych. Tak więc umieszczenie dyrektywy `using` w plikach nagłówkowych może bardzo łatwo doprowadzić do „wyłączenia” przestrzeni nazw praktycznie w każdym pliku, likwidując korzyści z nimi związane.

Krótko mówiąc — w plikach nagłówkowych nie należy umieszczać dyrektywy `using`.

Wykorzystywanie plików nagłówkowych w projektach

Podczas budowy projektów na ogół łączy się ze sobą wiele różnych typów (struktur danych wraz z towarzyszącymi im funkcjami). Zazwyczaj deklaracje każdego typu lub grupy powiązanych ze sobą typów znajdują się w oddzielnych plikach nagłówkowych, a następnie definiuje się funkcje tych typów w poszczególnych jednostkach translacji. Podczas używania typu trzeba dołączyć odpowiedni plik nagłówkowy, co zapewnia właściwe dokonanie deklaracji.

W książce występują przykłady przygotowane zgodnie z opisanymi powyżej wzorcami, ale częściej zdarza się, że prezentowane przykłady są bardzo małe. Wszystkie elementy — deklaracje, struktury, definicje funkcji oraz funkcja `main()` — mogą zatem znajdować się w pojedynczym pliku. Warto jednak pamiętać, że w praktyce korzystne jest używanie oddzielnych plików oraz plików nagłówkowych.

Zagnieżdżone struktury

Wygoda, wynikająca z usunięcia nazw danych funkcji z globalnej przestrzeni nazw, dotyczy również struktur. Można umieścić strukturę wewnętrz inniej struktury, utrzymując w ten sposób razem powiązane ze sobą elementy. Składnia takiej konstrukcji jest zgodna z oczekiwaniami, o czym świadczy przedstawiona poniżej struktura, stanowiąca implementację rozwijanego w dół stosu, zrealizowanego za pomocą prostej listy powiązanej, dzięki czemu „nigdy” nie wykracza on poza dostępna pamięć:

```
//: C04:Stack.h
// Zagnieżdżone struktury, tworzące listę powiązaną powiązanej
#ifndef STACK_H
#define STACK_H

struct Stack {
    struct Link {
        void* data;
        Link* next;
        void initialize(void* dat, Link* nxt);
    }* head;
    void initialize();
    void push(void* dat);
    void* peek();
    void* pop();
    void cleanup();
};
#endif // STACK_H ///:~
```

Zagnieżdżona struktura nosi nazwę **Link** i zawiera wskaźnik do następnej struktury **Link**, znajdującej się na liście, oraz wskaźnik do danych przechowywanych w strukturze. Jeżeli wartością wskaźnika **next** jest zero, oznacza to osiągnięcie końca listy.

Zwróć uwagę na to, że wskaźnik **head** został zdefiniowany bezpośrednio po deklaracji struktury **Link**, a nie w oddzielnej definicji o postaci **Link* head**. Jest to składnia wywodząca się z języka C, ale podkreśla ona znaczenie średnika, występującego po deklaracji struktury — średnik oznacza koniec listy oddzielonych przecinkami definicji zmiennych tego typu (zazwyczaj ta lista jest pusta).

Podobnie jak inne struktury prezentowane do tej pory, zagnieżdżona struktura posiada funkcję **initialize()**, zapewniającą odpowiednią inicjalizację. Struktura **Stack** posiada zarówno funkcje **initialize()** oraz **cleanup()**, jak i dwie funkcje. Funkcja **push()** pobiera wskaźnik do danych, które mają być przechowywane (zakłada ona, że dane zostały umieszczone na stercie); z kolei **pop()** zwraca wskaźnik danych (**data**) zawarty w elemencie znajdującym się na szczytce stosu i usuwa ten element ze stosu (po po-braniu elementu ze stosu użytkownik jest odpowiedzialny za usunięcie obiektu wskażanego przez **data**). Funkcja **peek()** również zwraca wskaźnik danych zawarty w elemencie znajdującym się na szczytce stosu, ale pozostawia ten element na stosie.

Poniżej przedstawiono definicje funkcji składowych:

```
//: C04:Stack.cpp {0}
// Lista powiązana z zagnieżdżaniem
#include "Stack.h"
#include "../require.h"
using namespace std;

void
Stack::Link::initialize(void* dat, Link* nxt) {
    data = dat;
    next = nxt;
}

void Stack::initialize() { head = 0; }
```

```
void Stack::push(void* dat) {
    Link* newLink = new Link;
    newLink->initialize(dat, head);
    head = newLink;
}

void* Stack::peek() {
    require(head != 0, "Stos jest pusty");
    return head->data;
}

void* Stack::pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}

void Stack::cleanup() {
    require(head == 0, "Stos nie jest pusty");
} //:-)
```

Pierwsza z definicji jest szczególnie interesująca, ponieważ prezentuje ona sposób, w jaki definiowana jest składowa zagnieździonej struktury. Używa się w tym celu dodatkowego poziomu zasięgu, określającego nazwę zawierającej go struktury. Funkcja **Stack::Link::initialize()** pobiera argumenty, przypisując swoim zmiennym ich wartości.

Funkcja **Stack::initialize()** ustawia zerową wartość wskaźnika **head**, dzięki czemu obiekt wie, że lista jest pusta.

Funkcja **Stack::push()** pobiera argument będący wskaźnikiem do zmiennej, która ma zostać zapamiętana, i umieszcza go na stosie. Najpierw wykorzystuje operator **new** do przydzielenia pamięci strukturze **Link**, którą ulokuje na szczytce stosu. Następnie wywołuje funkcję **initialize()** struktury **Link** w celu przypisania odpowiednich wartości składowym struktury **Link**. Zwróć uwagę na to, że wskaźnikowi **next** jest przypisywana aktualna wartość wskaźnika **head**, a następnie wskaźnikowi **head** przyporządkowany jest wskaźnik do nowo utworzonej struktury **Link**. W praktyce powoduje to umieszczenie struktury **Link** na szczytce stosu.

Funkcja **Stack::pop()** pobiera wskaźnik danych (**data**) znajdujących się na szczytce stosu, a następnie przesuwa wskaźnik **head** do następnej pozycji i usuwa stary szczyt stosu, zwracając ostatecznie pobrany wskaźnik danych. Gdy funkcja **pop()** usunie ostatni element stosu, wskaźnik **head** ponownie osiągnie wartość zerową, oznaczającą, że stos jest pusty.

Funkcja **Stack::cleanup()** w rzeczywistości niczego nie porządkuje. Określa ona natomiast „politykę firmy”, która sprowadza się do tego, że „klient-programista, używający tego stosu, jest odpowiedzialny za pobranie z niego wszystkich elementów oraz ich usunięcie”. Do zasygnalizowania, że w przypadku gdy stos nie jest pusty, wystąpił błąd programistyczny, używaną jest funkcja **require()**.

Dlaczego wszystkimi obiektami, które klient pozostawił na stosie, nie mógłby się zająć destruktor stosu? Problem polega na tym, że na stosie przechowywane są wskaźniki typu **void*** i, jak przekonamy się w rozdziale 13., użycie w stosunku do takiego wskaźnika operatora **delete** nie powoduje poprawnego usunięcia obiektu. Problem „którego jest odpowiedzialny za pamięć” nie jest nawet tak prosty, o czym dowiesz się dzięki lekturze następnych rozdziałów.

Poniżej znajduje się przykładowy program, testujący strukturę **Stack**:

```
//: C04:StackTest.cpp
//{L} Stack
//{T} StackTest.cpp
// Test zagnieżdzonej listy powiązanej
#include "Stack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // Argumentem jest nazwa pliku
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack textlines;
    textlines.initialize();
    string line;
    // Odczytanie pliku i zapamiętanie wierszy na stosie:
    while(getline(in, line))
        textlines.push(new string(line));
    // Pobranie wierszy ze stosu i wydrukowanie ich:
    string* s;
    while((s = (string*)textlines.pop()) != 0) {
        cout << *s << endl;
        delete s;
    }
    textlines.cleanup();
}///:-
```

Program jest to podobny do przedstawionego w poprzednim przykładzie, ale umieszcza wiersze odczytane z pliku (jako wskaźniki do łańcuchów) na stosie, a następnie pobiera je z niego, w rezultacie czego wiersze pliku są drukowane w odwrotnej kolejności. Warto zwrócić uwagę na to, że funkcja składowa **pop()** zwraca wskaźnik **void***, który przed wykorzystaniem musi być z powrotem rzutowany na typ **string***. W celu wydrukowania łańcucha dokonuje się wyłuskania wskaźnika.

Podczas wypełniania stosu **textlines** zawartość zmiennej **line** jest dla każdej operacji **push()** „kilonowana” za pomocą wyrażenia **new string(line)**. Wartość zwracana przez wyrażenie **new** jest wskaźnikiem do nowo utworzonego łańcucha, do którego kopowana jest wartość zmiennej **line**. Gdyby przekazywać funkcji **push** po prostu adres zmiennej **line**, doprowadziłoby to do wypełnienia stosu identycznymi adresami, wskazującymi na zmienną **line**. W dalszej części książki znajduje się więcej informacji na temat takiego „kilonowania”.

Nazwa pliku jest pobierana z wiersza poleceń. Do upewnienia się, że w wierszu poleceń podano odpowiednią liczbę argumentów, użyto funkcji `requireArgs()`, pochodzącej z pliku nagłówkowego `require.h`. Funkcja ta porównuje argument `argc` z liczbą oczekiwanych argumentów, drukując odpowiedni komunikat o błędzie i kończąc pracę programu w przypadku, gdy nie podano odpowiedniej ich liczby.

Zasięg globalny

Operator zasięgu pozwala na wyjście z sytuacji, w których domyślnie wybrana przez kompilator („najbliższa”) nazwa nie jest żądaną. Założmy, że mamy strukturę zawierającą identyfikator lokalny `a`, natomiast wewnątrz funkcji składowej zamierzamy użyć globalnego identyfikatora `a`. Kompilator powinien wybrać domyślnie lokalny identyfikator, należy więc określić to w jakiś inny sposób. Chcąc wskazać nazwę **globalną**, określając jej zasięg, należy użyć operatora zasięgu, przed którym nic się nie znajduje. Poniżej zamieszczono przykład prezentujący wybór zasięgu globalnego zarówno w przypadku zmiennej, jak i funkcji:

```
//: C04:Scoperes.cpp
// Określenie zasięgu globalnego
int a;
void f() {}

struct S {
    int a;
    void f();
};

void S::f() {
    ::f(); // Inaczej wywołanie byłoby rekurencyjne!
    ::a++; // Wybór zmiennej globalnej a
    a--; // Zmienna a w zasięgu struktury
}
int main() { S s; f(); } //:~
```

Gdyby w funkcji `S::f()` nie określono zasięgu, kompilator wybrałby domyślnie wersje `f()` i `a`, będące składowymi struktury.

Podsumowanie

W rozdziale przedstawiono zasadniczy „zwrot”, dokonany dzięki językowi C++ — możliwość umieszczania funkcji wewnątrz struktur. Takie nowe rodzaje struktur są nazywane *abstrakcyjnymi typami danych*, a zmienne utworzone za pomocą tych struktur — *obiektami* lub *egzemplarzami* tych typów. Wywoływanie funkcji składowych obiektów określa się mianem *wysyłania do nich komunikatów*. Wysyłanie komunikatów do obiektów jest podstawowym działaniem związanym z programowaniem obiektowym.

Mimo że połączenie danych i funkcji przynosi istotną korzyść z punktu widzenia organizacji kodu, ułatwia korzystanie z bibliotek, zapobiega kolizjom nazw, ukrywając je, to jednak można zrobić znacznie więcej, by uczynić programowanie w C++ bezpieczniejszym. W następnym rozdziale zostanie przedstawiony sposób pozwalający na takąochronę niektórych składowych struktury, by tylko jej autor mógł wykonywać na nich operacje. Wyznacza to wyraźną granicę pomiędzy tym, co może zmienić użytkownik danej struktury, i tym, co może zmodyfikować wyłącznie programista.

Ćwiczenia

Rozwiązań wybranych ćwiczeń można znaleźć w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

- ±. W standardowej bibliotece języka C funkcja **puts()** drukuje na konsoli tablicę znakową (dzięki czemu można napisać **puts("witaj")**). Napisz program w języku C, który wykorzystuje funkcję **puts()**, ale nie dołącza pliku nagłówkowego **<stdio.h>** ani nie deklaruje tej funkcji w żaden inny sposób. Skompiluj program za pomocą kompilatora języka C (niektóre kompilatory języka C++ pełnią równocześnie funkcję kompilatorów języka C — w takim przypadku musisz odszukać parametr wiersza poleceń, wymuszający kompilację języka C). Następnie skompiluj program za pomocą kompilatora C++ i zaobserwuj różnice.
2. Utwórz deklarację struktury posiadającej pojedynczą funkcję **składową**, a następnie utwórz definicję tej funkcji. Utwórz obiekt nowego typu danych i wywołaj funkcję składową.
3. Zmodyfikuj rozwiązanie poprzedniego ćwiczenia w taki sposób, by struktura została zadeklarowana w odpowiednio „strzeżonym” pliku nagłówkowym, z **definicją w jednym** pliku **cpp** i **funkcją main()** — w drugim.
4. Utwórz strukturę składającą się z pojedynczej składowej typu całkowitego oraz dwie funkcje globalne, z których każda przyjmuje wskaźnik do tej struktury. Pierwsza funkcja powinna posiadać drugi argument typu całkowitego i przypisywać jego wartość składowej całkowitej struktury. Druga funkcja powinna wyświetlać wartość tej składowej. Przetestuj obie funkcje.
5. Powtórz poprzednie ćwiczenie, zmieniając funkcje w taki sposób, aby stanowiły składowe struktury, i ponownie przetestuj działanie programu.
6. Za pomocą kluczowego **this** (odpowiadającego adresowi bieżącego obiektu) utwórz klasę, która będzie (nadmiarowo) dokonywała wyboru danych składowych i wywoływała funkcje składowe.
7. Zmodyfikuj strukturę **Stash** w taki sposób, aby przechowywała liczby typu **double**. Umieść na nim 25 wartości typu **double**, a następnie wydrukuj je na konsoli.

8. Powtórz poprzednie ćwiczenie, dokonując modyfikacji struktury **Stack**.
9. Za pomocą funkcji **printf()**, zawartej w pliku nagłówkowym **<stdio.h>**, utwórz plik zawierający funkcję **f()**, przyjmującą argument całkowity i drukującą jego wartość na konsoli. Użyj zapisu: **printf("%d\n", i)**, gdzie **i** jest drukowaną wartością całkowitą. Utwórz oddzielnego pliku, zawierającego funkcję **main()**, i zadeklaruj w nim funkcję **f()** jako funkcję przyjmującą argument typu **float**. Wywołaj funkcję **f()** z wnętrza funkcji **main()**. Spróbuj skompilować i dokonać łączenia programu za pomocą kompilatora C++ i sprawdź, co się stanie. Następnie skompiluj i dokonaj łączenia programu, używając do tego kompilatora języka C. Zobacz, co się stanie, gdy uruchomisz program. Wyjaśnij jego działanie.
10. Dowiedz się, w jaki sposób wygenerować na wyjściu kompilatorów C i C++ program w asemblerze. Napisz w języku C funkcję, awięzyku C++ — strukturę, zawierającą jedną funkcję składową. Dla każdego z tych plików wygeneruj program w języku asemblera. W utworzonych plikach poszukaj nazw utworzonych dla nazwy funkcji w języku C i funkcji składowej w C++, aby zobaczyć, jak nazwy te zostały uzupełnione przez kompilator.
11. Napisz program komplilujący warunkowo kod, zawarty w funkcji **main()**, w taki sposób, by w razie zdefiniowania określonego symbolu preprocesora drukowany był jakiś komunikat, a w przeciwnym przypadku — by drukowany był inny komunikat. Skompiluj ten kod, eksperymentując z dyrektywami **#define**, zawartymi w programie, a następnie dowiedz się, jak podaje się definicje preprocesora w wierszu poleceń kompilatora, i również przeprowadź eksperymenty.
12. Napisz program, używający makroinstrukcji **assert()** z argumentem, który jest zawsze fałszywy (zerowy), i zobacz, co się stanie, gdy go uruchomisz. Następnie skompiluj go, włączając do programu dyrektywę **#define NDEBUG**, uruchom go ponownie i zwróć uwagę na różnice.
13. Utwórz abstrakcyjny typ danych, reprezentujący kasetę wypożyczalni wideo. Zastanów się nad danymi oraz operacjami, których może wymagać typ **Video**, by działał prawidłowo w systemie zarządzania wypożyczalnią wideo. Dołącz funkcję składową **print()**, wyświetlającą informacje o zmiennej typu **Video**.
14. Utwórz obiekt **Stack**, przechowujący obiekty typu **Video**, pochodzące z poprzedniego ćwiczenia. Utwórz kilka obiektów typu **Video**, umieść je w obiekcie **Stack**, a następnie wydrukuj, używając funkcji **Video::print()**.
15. Napisz program, który drukuje na twoim komputerze wielkości wszystkich podstawowych typów danych, używając do tego operatora **sizeof**.
16. Zmodyfikuj strukturę **Stash** w taki sposób, by jako podstawowej struktury danych używała ona typu **vector<char>**.
17. Używając operatora **new**, dynamicznie przydziel obszary pamięci następujących typów: **int**, **long**, tablicę o 100 znakach i tablicę 100 wartości typu **float**. Wydrukuj adresy tych obszarów, a następnie zwolnij przydzieloną pamięć, używając operatora **delete**.

18. Napisz funkcję przyjmującą parametr typu **char***. Używając operatora **new**, przydziel dynamicznie pamięć tablicy znaków o takiej samej wielkości, jak tablica znakowa przekazana funkcji. Wykorzystując indeksowanie tablic, skopiuj znaki argumentu do dynamicznie przydzielonej tablicy (nie zapomnij o zerze, kończącym tablicę znakową) i zwróć wskaźnik do utworzonej w ten sposób kopii. Przetestuj napisaną funkcję w obrębie funkcji **main()**, przekazującej **statyczną tablicę** znaków znajdującej się w cudzysłowie, a zwróconą wartość przekaż z powrotem swojej funkcji. Wydrukuj oba łańcuchy i oba wskaźniki w taki sposób, aby było widać, że są to różne obszary pamięci. Zwolnij całą przydzieloną dynamicznie pamięć, używając do tego operatora **delete**.
19. Zaprezentuj przykład struktury zadeklarowanej wewnątrz innej struktury (czyli *struktury zagnieżdzionej*). W obu strukturach zadeklaruj dane składowe, a następnie zadeklaruj i zdefiniuj w nich funkcje składowe. Napisz funkcję **main()** testującą utworzone przez ciebie typy.
20. Napisz kod drukujący wielkości różnych struktur. Utwórz struktury zawierające wyłącznie dane składowe i takie, które składają się zarówno z danych, jak i z funkcji. Następnie utwórz strukturę, która w ogóle nie ma składowych. Wydrukuj wielkość wszystkich utworzonych struktur. Wyjaśnij wyniki uzyskane dla struktury nieposiadającej składowych.
21. Jak wiadomo, w przypadku struktur język C++ automatycznie podejmuje działanie równoważne użyciu słowa kluczowego **typedef**. Czyni tak również w przypadku wyliczeń i unii. Napisz niewielki program, który to demonstruje.
22. Utwórz stos (**Stack**) przechowujący elementy typu **Stash**. Każdy element typu **Stash** powinien zawierać pięć wierszy pliku wejściowego i należy go utworzyć za pomocą operatora **new**. Wczytaj plik, zapamiętując go na stosie, a następnie wydrukuj w pierwotnej postaci, pobierając dane ze stosu.
23. Zmodyfikuj poprzednie ćwiczenie, tworząc strukturę zawierającą stos obiektów typu **Stash**. Jej użytkownik powinien jedynie dodawać i pobierać wiersze, używając do tego funkcji składowych, ale wewnątrz struktury powinny być stosowane typy **Stack** i **Stash**.
24. Utwórz strukturę przechowującą liczbę całkowitą i wskaźnik do innego egzemplarza tej samej struktury. Napisz funkcję pobierającą adres jednej z takich struktur oraz liczbę całkowitą, oznaczającą długość listy, którą zamierzasz utworzyć. Funkcja ta będzie tworzyła cały łańcuch takich struktur (*listę powiązaną*), rozpoczynając od argumentu (*głowy listy*); każda z tych struktur będzie wskazywała na następną. Do ich tworzenia użyj operatora **new**, zapisując licznik (numer porządkowy obiektu) w składowej, będącej liczbą całkowitą. W ostatniej strukturze na liście przypisz wskaźnikowi następnej struktury wartość **zerową**, oznaczającą koniec listy. Napisz drugą funkcję, pobierającą głowę listy i przesuwającą się wzdłuż niej aż do końca. Funkcja ta drukuje dla każdego elementu zarówno wartość wskaźnika, jak i zapamiętaną w elemencie wartość całkowitą.
25. Powtórz poprzednie ćwiczenie, umieszczając wszystkie funkcje wewnątrz struktury (zamiast używania „oddzielnich” struktur i funkcji).

Rozdział 5.

Ukrywanie implementacji

Typowa biblioteka składa się w języku C ze struktury i kilku dołączonych funkcji, wykonujących na niej operacje. Zapoznaliśmy się ze sposobem, w jaki język C++ grupuje funkcje, związane ze sobą *pojęciowo*, i łączy je *literalnie*. Dokonuje tego umieszczając deklaracje funkcji w obrębie zasięgu struktury, zmieniając sposób, w jaki funkcje te są wywoływanew stosunku do tej struktury, eliminując przekazywanie adresu struktury jako pierwszego argumentu i dodając do programu nazwę nowego typu (dzięki czemu nie trzeba używać słowa kluczowego **typedef** w stosunku do identyfikatora struktury).

Wszystko to zapewnia większą wygodę — pozwala lepiej zorganizować kod i ułatwia zarówno jego napisanie, jak i przeczytanie. Warto jednak poruszyć jeszcze inne ważne zagadnienia, związane z łatwiejszym tworzeniem bibliotek w języku C++ — w szczególności są to kwestie, dotyczące bezpieczeństwa i kontroli. W niniejszym rozdziale zapoznamy się bliżej z kwestiami ograniczeń dotyczących struktur.

Określanie ograniczeń

W każdej relacji istotne jest określenie granic, respektowanych przez wszystkie zaangażowane w nią strony. Tworząc bibliotekę, ustanawiasz relację z *klientem-programistą*, używającym twojej biblioteki do zbudowania aplikacji lub utworzenia innej biblioteki.

W strukturach dostępnych w języku C, podobnie jak w większości elementów tego języka, nie obowiązują żadne reguły. Klienci-programiści mogą postąpić dowolnie ze strukturą i nie ma żadnego sposobu, by wymusić na nich jakiekolwiek szczególne zachowania. Na przykład mimo ważności funkcji o nazwach **initialize()** i **cleanup()** (wskazanej w poprzednim rozdziale), klient-programista może w ogóle ich nie wywołać (lepsze rozwiązywanie tej kwestii zaprezentujemy w następnym rozdziale). W języku C nie ma żadnego sposobu zapobieżenia temu, by klienci-programiści operowali bezpośrednio na niektórych składowych struktur. Wszystko ma charakterjawnny.

Istnieją dwa powody wprowadzenia kontroli dostępu do składowych struktur. Przede wszystkim programistom należy uniemożliwić stosowanie narzędzi niezbędnych do wykonywania wewnętrznych operacji związanych z typem danych, lecz niebędących częścią interfejsu potrzebnego klientom-programistom do rozwiązania ich własnych problemów. Jest to w rzeczywistości pomoc udzielana klientom-programistom, ponieważ dzięki temu mogąłtво odróżnić kwestie istotne od pozostałych.

Drugim powodem wprowadzenia kontroli dostępu jest umożliwienie projektantowi biblioteki zmiany wewnętrznych mechanizmów struktury z pominięciem wpływu na klienta-programistę. W przykładzie ze stosem, przedstawionym w poprzednim rozdziale, z uwagi na szybkość można by przydzielać pamięć dużymi porcjami zamiast tworzyć kolejny jej obszar, ilekroć dodawany jest nowy element. Jeżeli interfejs oraz implementacja są wyraźnie od siebie oddzielone i chronione, można tego dokonać, wymagając od klienta-programisty jedynie przeprowadzenia ponownego łączenia modułów wynikowych.

Kontrola dostępu w C++

Język C++ wprowadza trzy nowe słowa kluczowe, pozwalające na określenie granic w obrębie struktur: **public** (publiczny), **private** (prywatny) i **protected** (chroniony). Sposób ich użycia oraz znaczenie wydają się dość oczywiste. Są one *specyfikatorami dostępu* (ang. *access specifiers*), używanymi wyłącznie w deklaracjach struktur, zmieniającymi ograniczenia dla wszystkich następujących po nich definicji. Specyfikator dostępu zawsze musi kończyć się średnikiem.

Specyfikator **public** oznacza, że wszystkie następujące po nim deklaracje składowych są dostępne dla wszystkich. Składowe publiczne są takie same, jak zwykłe składowe struktur. Na przykład poniższe deklaracje struktur są identyczne:

```
//: C05:Public.cpp
// Specyfikator public przypomina zwykłe
// struktury języka C

struct A {
    int i;
    char j;
    float f;
    void func();
};

void A::func() {}

struct B {
public:
    int i;
    char j;
    float f;
    void func();
};
```

```
void B::func() {}

int main() {
    A a; B b;
    a.i = b.i = 1;
    a.j = b.j = 'c';
    a.f = b.f = 3.14159;
    a.func();
    b.func();
} //:-
```

Z kolei słowo kluczowe **private** oznacza, że do składowych struktury nie ma dostępu nikt, oprócz ciebie, twórcy typu, i to jedynie w obrębie funkcji składowych tego typu. Specyfikator **private** stanowi barierę pomiędzy tobą i klientem-programistą — każdy, kto spróbuje odwołać się do prywatnej składowej klasy, otrzyma komunikat o błędzie już na etapie komplikacji. W powyższej strukturze B mógłbyś chcieć na przykład ukryć część jej reprezentacji (tj. danych składowych), dzięki czemu byłyby one dostępne wyłącznie dla ciebie:

```
//: C05:Private.cpp
// Określanie granicy

struct B {
private:
    char j;
    float f;
public:
    int i;
    void func();
};

void B::func() {
    i = 0;
    j = '0';
    f = 0.0;
}

int main() {
    B b;
    b.i = 1;    // OK, składowa publiczna
    //! b.j = '1'; // Niedozwolone, składowa prywatna
    //! b.f = 1.0; // Niedozwolone, składowa prywatna
} //:-
```

Mimo że funkcja **func()** ma dostęp do każdej składowej struktury B (ponieważ jest ona również składową struktury B, więc automatycznie uzyskuje do tego prawo), to zwykła funkcja globalna, taka jak **main()**, nie posiada takich uprawnień. Oczywiście, do tych składowych nie mają dostępu również funkcje składowe innych struktur. Wyłącznie funkcje, które zostały wyraźnie wymienione w deklaracji struktury („kontrakt”), mają dostęp do prywatnych składowych struktury.

Nie istnieje określony porządek, w jakim powinny występować specyfikatory dostępu; mogą one również występować więcej niż jednokrotnie. Dotyczą one wszystkich zadeklarowanych po nich składowych, aż do napotkania następnego specyfikatora dostępu.

Specyfikator **protected**

Ostatnim specyfikatorem dostępu jest **protected**. Jego znaczenie jest zbliżone do specyfikatora **private**, z jednym wyjątkiem, który nie może zostać jeszcze teraz wyjaśniony — struktury „dziedziczące” (które nie posiadają dostępu do składowych prywatnych) mają zagwarantowany dostęp do składowych oznaczonych specyfikatorem **protected**. Zostanie to wyjaśnione w rozdziale 14., przy okazji wprowadzenia pojęcia dziedziczenia. Tymczasowo można przyjąć, że specyfikator **protected** działa podobnie do specyfikatora **private**.

Przyjaciele

Co zrobić w sytuacji, gdy chcemy jawnie udzielić pozwolenia na dostęp funkcji, niebędącej składowąbieżącej struktury? Uzyskuje się to, deklarując funkcję za pomocą słowa kluczowego **friend** (przyjaciel) *wewnątrz* deklaracji struktury. Ważne jest, by deklaracja **friend** występowała w obrębie deklaracji struktury, ponieważ programista (i kompilator), czytając deklarację struktury, musi poznać wszystkie zasady dotyczące wielkości i zachowania tego typu danych. A niezwykle ważną zasadę, obowiązującą w każdej relacji, stanowi odpowiedź na pytanie: „Kto ma dostęp do mojej prywatnej implementacji?”.

To sama struktura określa, który kod ma dostęp do jej składowych. Nie ma żadnego magicznego sposobu „włamania się” z zewnątrz, jeżeli nie jest się „przyjacielem” — nie można zadeklarować nowej klasy, twierdząc: „Cześć, jestem przyjacielem Boba!” i społudzając się, że zapewni to dostęp do prywatnych i chronionych składowych klasy Bob.

Wolno zadeklarować jako „przyjaciela” funkcję globalną; może nim być również składowa innej struktury albo nawet cała struktura zadeklarowana z użyciem słowa kluczowego **friend**. Poniżej zamieszczono przykład:

```
//: C05:Friend.cpp
// Słowo kluczowe friend daje specjalne
// prawa dostępu

// Deklaracja (niekompletna specyfikacja typu):
struct X;

struct Y {
    void f(X*);
};

struct X { // Definicja
private:
    int i;
public:
    void initialize();
    friend void g(X*, int); // Przyjaciel globalny
    friend void Y::f(X*); // Przyjaciel będący składową struktury
    friend struct Z; // Cała struktura jako przyjaciel
    friend void h();
};
```

```
void X::initialize() {
    i = 0;
}

void g(X* x, int i) {
    x->i = i;
}

void Y::f(X* x) {
    x->i = 47;
}

struct Z {
private:
    int j;
public:
    void initialize();
    void g(X* x);
}:

void Z::initialize() {
    J = 99;
}

void Z::g(X* x) {
    x->i += j;
}

void h() {
    X x;
    x.i = 100; // Bezpośrednia operacja na składowej
}

int main() {
    X x;
    Z z;
    z.g(&x);
} //:-
```

Struktura Y posiada funkcję składową f(), modyfikującą obiekt typu X. Jest to nieco zagadkowe, ponieważ kompilator języka C++ wymaga zadeklarowania każdej *rzeczy* przed odwołaniem się do niej. Należy więc zadeklarować strukturę Y, zanim jeszcze jej składowa Y::f(X*) będzie mogła zostać zadeklarowana jako przyjaciel struktury X. Jednakże, aby funkcja Y::f(X*) mogła zostać zadeklarowana, najpierw należy zadeklarować strukturę X!

Oto rozwiązanie. *Zwróć uwagę* na to, że funkcja Y::f(X*) pobiera *adres* obiektu X. Jest to istotne, ponieważ kompilator zawsze wie, w jaki sposób przekazać adres będący stałej wielkości, niezależnie od przekazywanego za jego pośrednictwem obiektu i nawet jeżeli nie posiada pełnej informacji dotyczącej jego wielkości. Jednakże w przypadku próby przekazania całego obiektu kompilator musi widzieć całą definicję struktury X, aby poznać jej wielkość i wiedzieć, w jaki sposób przekazać, zanim pozwoli na deklarację funkcji w rodzaju Y::g(X).

Przekazując adres struktury X, kompilator pozwala na utworzenie *niepełnej specyfikacji typu* X, umieszczonej przed deklaracją Y::f(X*). Uzyskuje się ją za pomocą deklaracji:

```
struct X;
```

Deklaracja ta informuje kompilator, że istnieje struktura o podanej nazwie, więc można odwołać się do niej, dopóki nie jest na jej temat potrzebna żadna dodatkowa wiedza, poza nazwą.

Potem funkcja Y::f(X*) w strukturze X może być już bez problemu zadeklarowana jako „przyjaciel”. W razie próby zadeklarowania ją, zanim kompilator napotka pełną specyfikację klasy Y, nastąpiłoby zgłoszenie błędu. Jest to cecha zapewniająca bezpieczeństwo i spójność, a także zapobiegająca błędom.

Zwrócić uwagę na dwie pozostałe funkcje zadeklarowane z użyciem słowa kluczowego **friend**. Pierwsza z nich deklaruje jako przyjaciela zwykłą funkcję globalną g(). Funkcja ta nie została jednak wcześniej zadeklarowana w zasięgu globalnym! Okazuje się, że taki sposób użycia deklaracji **friend** może zostać wykorzystany do równoczesnego zadeklarowania funkcji i nadania jej statusu przyjaciela. Dotyczy to również całych struktur. Deklaracja:

```
friend struct Z;
```

jest niepełną specyfikacją typu struktury Z, nadając równocześnie całej tej strukturze status przyjaciela.

Zagnieżdżeni przyjaciele

Utworzenie struktury zagnieżdżonej nie zapewnia jej automatycznie prawa dostępu do składowych prywatnych. Aby to osiągnąć, należy postąpić w szczególny sposób: najpierw zadeklarować (nie definiując) strukturę zagnieżdżoną, następnie zadeklarować ją, używając słowa kluczowego **friend**, a na koniec — zdefiniować strukturę. Definicja struktury musi być oddzielona od deklaracji **friend**, bo w przeciwnym przypadku kompilator nie uznałby jej za składową struktury. Poniżej zamieszczono przykład takiego zagnieżdżenia struktury:

```
//: C05:NestFriend.cpp
// Zagnieżdżeni "przyjaciele"
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;

struct Holder {
private:
    int a[sz];
public:
    void initialize();
    struct Pointer;
    friend Pointer;
    struct Pointer {
```

```
private:
    Holder* h;
    int* p;
public:
    void initialize(Holder* h);
    // Poruszanie się w obrębie tablicy:
    void next();
    void previous();
    void top();
    void end();
    // Dostęp do wartości:
    int read();
    void set(int i);
};

};

void Holder::initialize() {
    memset(a, 0, sz * sizeof(int));
}

void Holder::Pointer::initialize(Holder* rv) {
    h = rv;
    p = rv->a;
}

void Holder::Pointer::next() {
    if(p < &(h->a[sz - 1])) p++;
}

void Holder::Pointer::previous() {
    if(p > &(h->a[0])) p--;
}

void Holder::Pointer::top() {
    p = &(h->a[0]);
}

void Holder::Pointer::end() {
    p = &(h->a[sz - 1]);
}

int Holder::Pointer::read() {
    return *p;
}

void Holder::Pointer::set(int i) {
    *p = i;
}

int main() {
    Holder h;
    Holder::Pointer hp, hp2;
    int i;

    h.initialize();
    hp.initialize(&h);
    hp2.initialize(&h);
```

```

for(i = 0; i < sz; i++) {
    hp.set(i);
    hp.next();
}
hp.top();
hp2.end();
for(i = 0; i < sz; i++) {
    cout << "hp = " << hp.read()
        << ". hp2 = " << hp2.read() << endl;
    hp.next();
    hp2.previous();
}
} //:-_

```

Po zadeklarowaniu struktury **Pointer** deklaracja:

```
friend Pointer;
```

zapewnia jej dostęp do prywatnych składowych struktury **Holder**. Struktura **Holder** zawiera tablicę liczb całkowitych, do których dostęp jest możliwy właśnie dzięki strukturze **Pointer**. Ponieważ struktura **Pointer** jest ściśle związana ze strukturą **Holder**, rozsądne jest uczynienie z niej składowej struktury **Holder**. Ponieważ jednak **Pointer** stanowi oddzielną strukturę w stosunku do struktury **Holder**, można utworzyć w funkcji **main()** większą liczbę jej egzemplarzy, używając ich następnie do wyboru różnych fragmentów tablicy. **Pointer** jest strukturą z niezwykłym wskaźnikiem języka C, gwarantuje więc zawsze poprawne wskazania w obrębie struktury **Holder**.

Funkcja **memset()**, wchodząca w skład standardowej biblioteki języka C (zadeklarowana w **<cstring>**), została dla wygody wykorzystana w powyższym programie. Począwszy od określonego adresu (będącego pierwszym argumentem) wypełnia ona całapamięć odpowiednią wartością (podaną jako drugi argument), wypełniając kolejnych bajtów (n stanowi trzeci argument). Oczywiście, można przejść przez kolejne adresy pamięci, używając do tego pętli, ale funkcja **memset()** jest dostępna, starannie przetestowana (więc jest mało prawdopodobne, że powoduje błędy) i prawdopodobnie bardziej efektywna niż kod napisany samodzielnie.

Czy jest to „czyste”?

Definicja klasy udostępnia „dziennik nadzoru”, dzięki któremu analizując klasę można określić funkcje mające prawo do modyfikacji jej prywatnych elementów. Jeżeli funkcja została zadeklarowana z użyciem słowa kluczowego **friend**, oznacza to, że nie jest ona funkcją składową, ale mimo to chcemy dać jej prawo do modyfikacji prywatnych danych. Musi ona widzieć w definicji klasy, by wszyscy wiedzieli, że należy do funkcji uprzywilejowanych w taki właśnie sposób.

Język C++ jest hybrydowym językiem obiektowym, a nie językiem czysto obiektowym. Słowo kluczowe **friend** zostało do niego dodane w celu pominięcia problemów, które zdarzają się w praktyce. Można sformułować zarzut, że czyni to język mniej „czystym”. C++ został bowiem zaprojektowany w celu sprostania wymogowi użyteczności, a nie po to, by aspirował do miana abstrakcyjnego ideału.

Struktura pamięci obiektów

W rozdziale 4. napisano, że struktura przygotowana dla kompilatora języka C, a następnie skompilowana za pomocą kompilatora C++, nie powinna ulec zmianie. Odnoси się to przede wszystkim do układu pamięci obiektów tej struktury, to znaczy określenia, jak w pamięci przydzielonej obiekowi rozmieszczone są poszczególne zmienne, stanowiące jego składowe. Gdyby kompilator języka C++ zmieniał układ pamięci struktur języka C, to nie działałby żaden program napisany w C, który (co nie jest zalecane) wykorzystywałby informację o rozmieszczeniu w pamięci zmiennych tworzących strukturę.

Rozpoczęcie stosowania specyfikatorów dostępu zmienia jednak nieco postać *rzeczy*, przenosząc nas całkowicie w domenę języka C++. W obrębie określonego „bloku dostępu” (grupy deklaracji, ograniczonej specyfikatorami dostępu), gwarantowany jest zwarty układ zmiennych w pamięci, tak jak w języku C. Jednakże poszczególne bloki dostępu mogą nie występować w obiekcie w kolejności, w której zostały zadeklarowane. Mimo że kompilator *zazwyczaj* umieszcza te bloki w pamięci dokładnie w taki samej kolejności, w jakiej są one widoczne w programie, nie obowiązuje w tej kwestii żadna reguła. Niektóre architektury komputerów i (lub) środowiska systemów operacyjnych mogą bowiem udzielać jawnego wsparcia składowym prywatnym i chronionym, co może z kolei wymagać umieszczenia tych bloków w specjalnych obszarach pamięci. Specyfikacja językanie ma na celu ograniczenie możliwości wykorzystania tego typu korzyści.

Specyfikatory dostępu stanowią składniki struktur i nie wpływają na tworzone na ich podstawie obiekty. Wszelkie informacje dotyczące specyfikacji dostępu znikają, zanim jeszcze program zostanie uruchomiony — na ogół dzieje się to w czasie kompilacji. W działającym programie obiekty stają się „obszarami pamięci” i niczym więcej. Jeżeli naprawdę tego chcesz, możesz złamać wszelkie reguły, odwołując się bezpośrednio do pamięci, tak jak w języku C. Języka C++ nie zaprojektowano po to, by chronił cię przed popełnianiem głupstw. Stanowi on jedynie znacznie łatwiejsze i bardziej wartościowe rozwiązanie alternatywne.

Na ogół poleganie podczas pisania programu na czymkolwiek, co jest zależne od implementacji, nie jest dobrym pomysłem. Jeżeli musisz użyć czegoś, co zależy od implementacji, zamknij to w obrębie struktury, dzięki czemu zmiany związane z przenoszeniem programu będą skupione w jednym miejscu.

Klasy

Kontrola dostępu jest często określana mianem *ukrywania implementacji*. Umieszczenie funkcji w strukturach (często nazywane kapsułkowaniem¹) tworzy typy danych, posiadające zarówno cechy, jak i zachowanie. Jednakże kontrola dostępu wy-

¹ Jak już wspomniano, kapsułkowaniem jest również często nazywana kontrola dostępu.

znacza ograniczenia w obrębie tych typów danych, wynikające z dwóch istotnych powodów. Po pierwsze, określają one, co może, a czego nie może używać klient-programista. Można wbudować w strukturę wewnętrzne mechanizmy, nie martwając się o to, że klienci-programiści uznają te mechanizmy za część interfejsu, którego powinni używać.

Prowadzi to bezpośrednio do drugiego z powodów, którym jest oddzielenie interfejsu od implementacji. Jeżeli struktura jest używana w wielu programach, lecz klienci-programiści mogajedynie wysyłać komunikaty do jej publicznego interfejsu, to można w niej zmienić wszystko co jest prywatne, bez potrzeby zmiany kodu wykorzystujących ją programów.

Kapsułkowanie i kontrola dostępu, traktowane łącznie, tworzą coś więcej niż struktury dostępne w języku C. Dzięki nim wkraczamy do świata programowania obiektowego, w którym struktury opisują klasy obiektów w taki sposób, jakbyśmy opisywali klasę ryb albo klasę ptaków — każdy obiekt, należący do tych klas, będzie posiadał takie same cechy oraz rodzaje zachowań. Tym właśnie stała się deklaracja struktury — opisem, w jaki sposób wyglądają i funkcjonują wszystkie obiekty jej typu.

W pierwszym języku obiektowym, Simuli-67, słowo kluczowe **class** służyło do opisu nowych typów danych. Najwyraźniej zainspirowało to Stroustrupą do wyboru tego samego słowa kluczowego dla języka C++. Świadczy to o tym, że najważniejszą cechą całego języka jest tworzenie nowych typów danych, będące czymś więcej niż strukturami języka C zaopatrzonimi w funkcje. Z pewnością wydaje się to wystarczającym uzasadnieniem wprowadzenia nowego słowa kluczowego.

Jednakże sposób użycia słowa kluczowego **class** w języku C++ powoduje, że jest ono niemal niepotrzebne. Jest identyczne ze słowem kluczowym **struct** pod każdym względem, z wyjątkiem jednego: składowe klasy są domyślnie prywatne, a składowe struktury — domyślnie publiczne. Poniżej przedstawiono dwie struktury, dające takie same rezultaty:

```
//: C05:Class.cpp
// Podobieństwo struktur i klas

struct A {
private:
    int i, j, k;
public:
    int f();
    void g();
};

int A::f() {
    return i + j + k;
}

void A::g() {
    i = j = k = 0;
}

// Taki sam rezultat uzyskuje się za pomocą:
```

```
class B {
    int i, j, k;
public:
    int f();
    void g();
};

int B::f() {
    return i + j + k;
}

void B::g() {
    i = j = k = 0;
}

int main() {
    A a;
    B b;
    a.f(); a.g();
    b.f(); b.g();
} //:~
```

Klasa jest w języku C++ podstawowym pojęciem związanym z programowaniem obiektowym. Jest jednym ze słów kluczowych, które *nie zostały zaznaczone* w książce pogrubioną czcionką — byłoby to irytujące w przypadku słowa powtarzanego tak często jak „class”. Przejście do klas jest tak istotnym krokiem, że podejrzewam, iż Stroustrup miałby ochotę wyrzucić w ogóle słowo kluczowe **struct**. Przeszkodę stanowi jednak konieczność zachowania wstępnej zgodności języka C++ z językiem C.

Wiele osób preferuje styl tworzenia klas bliższy strukturom niż klasom. Nie przywiązuje wagi do „domyślnie prywatnego” zachowania klas, rozpoczynając deklaracje klas od ich elementów publicznych:

```
class X {
public:
    void funkcja_interfejsu();
private:
    void funkcja_prywatna();
    int wewnętrzna_reprezentacja;
};
```

Przemawia za tym argument, że czytelnikowi takiego kodu wydaje się bardziej logiczne czytanie najpierw interesujących go składowych, a następnie pominięcie wszystkiego, co zostało oznaczone jako prywatne. Faktycznie, wszystkie pozostałe składowe należy zadeklarować w obrębie klasy jedynie dla tego, że kompilator musi znać wielkości obiektów, by mógł przydzielić im we właściwy sposób pamięć. Istotna jest także możliwość zagwarantowania spójności klasy.

Jednak w przykładach występujących w książce składowe prywatne będą znajdowały się na początku deklaracji klasy, jak poniżej:

```
class X {
    void funkcja_prywatna();
    int wewnętrzna_reprezentacja;
public:
    void funkcja_interfejsu();
};
```

Niektórzy zadają sobie nawet trud uzupełniania swoich prywatnych nazw:

```
class Y {  
public:  
    void f();  
private:  
    int mX; // Uzupełniona nazwa  
};
```

Ponieważ zmienna **mX** jest już ukryta w zasięgu klasy **Y**, przedrostek **m** (od ang. *member* — członek, składowa) jest niepotrzebny. Jednak w projektach o wielu zmiennych globalnych (czego należy unikać, ale co w przypadku istniejących projektów jest czasami nieuniknione) ważną rolę odgrywa możliwość odróżnienia, które dane sądanymi globalnymi, a które — składowymi klasy.

Modyfikacja programu Stash, wykorzystująca kontrolę dostępu

Modyfikacja programu z rozdziału 4., dokonana w taki sposób, by używała on klas oraz kontroli dostępu, wydaje się racjonalna. *Zwróć uwagę* nato, w jaki sposób część interfejsu, przeznaczona dla klienta-programisty, została obecnie wyraźnie wyróżniona. Dzięki temu nie istnieje już możliwość, że przypadkowo będzie on wykonywał operacje na nieodpowiedniej części klasy:

```
//: C05:Stash.h  
// Zmieniony w celu wykorzystania kontroli dostępu  
#ifndef STASH_H  
#define STASH_H  
  
class Stash {  
    int size;      // Wielkość każdego elementu  
    int quantity; // Liczba elementów pamięci  
    int next;      // Następny pusty element  
    // Dynamicznie przydzielana tablica bajtów:  
    unsigned char* storage;  
    void inflate(int increase);  
public:  
    void initialize(int size);  
    void cleanup();  
    int add(void* element);  
    void* fetch(int index);  
    int count();  
};  
#endif // STASH_H //:/~
```

Funkcja **inflate()** została określona jako prywatna, ponieważ jest ona używana wyłącznie przez funkcję **add()**; stanowi zatem część wewnętrznego mechanizmu funkcjonowania klasy, a nie jej interfejsu. Oznacza to, że w przyszłości można będzie zmienić wewnętrzną implementację, używając innego systemu zarządzania pamięcią.

Powyższa zawartość pliku nagłówkowego jako jedyna — poza jego nazwą — uległa modyfikacji w powyższym przykładzie. Zarówno plik zawierający implementację, jak i plik testowy pozostały takie same.

Modyfikacja stosu, wykorzystująca kontrolę dostępu

W drugim przykładzie w klasę zostanie przekształcony program tworzący stos. Zagnieżdżona struktura danych jest obecnie strukturą prywatną, co wydaje się korzystne, ponieważ gwarantuje, że klient-programista nigdy nie będzie musiał się jej przyglądać ani nie uzależni on swojego programu od wewnętrznej reprezentacji klasy **Stack**:

```
//: C05:Stack2.h
// Zagnieżdżone struktury, tworzące listę powiązaną
#ifndef STACK2_H
#define STACK2_H

class Stack {
    struct Link {
        void* dąta;
        Link* next;
        void initialize(void* dat, Link* nxt);
    }* head;
public:
    void initialize();
    void push(void* dat);
    void* peek();
    void* pop();
    void cleanup();
};

#endif // STACK2_H ///:-
```

Podobnie jak poprzednio, implementacja nie uległa w tym przypadku zmianie, nie została więc w tym miejscu powtórnie przytoczona. Plik zawierający program testowy również się nie zmienił. Została jedynie zmodyfikowana moc, uzyskana dzięki interfejsowi klasy. Istotną korzyścią, wynikającą z kontroli dostępu, jest uniemożliwienie przekraczania granic podczas tworzenia programu. W rzeczywistości jedynie kompilator posiada informacje dotyczące poziomu zabezpieczeń poszczególnych składowych klasy. Nie istnieje żadna informacja umożliwiająca kontrolę dostępu, która byłaby dołączana do nazwy składowej klasy, a następnie przekazywana programowi łączącemu. Cała kontrola zabezpieczeń jest dokonywana przez kompilator i nie zostaje przerwana w czasie wykonywania programu.

Zwrócić uwagę na to, że interfejs prezentowany klientowi-programistie rzeczywiście odpowiada teraz rozwijanemu w dół stosowi. Jest on obecnie zaimplementowany w postaci powiązanej listy, lecz można to zmienić, nie modyfikując elementów wykorzystywanych przez klienta-programistę, a zatem (co ważniejsze) również ani jednego wiersza napisanego przez niego kodu.

Klasy-uchwyty

Kontrola dostępu w języku C++ pozwala na oddzielenie interfejsu od implementacji, jednak ukrycie implementacji jest tylko częściowe. Kompilator musi nadal widzieć deklaracje wszystkich elementów obiektu po to, by mógł poprawnie je tworzyć i odpowiednio

obsługiwać. Można wyobrazić sobie język programowania, który wymagałby określenia jedynie publicznego interfejsu obiektu, pozwalając na ukrycie jego prywatnej implementacji. Jednakże język C++ dokonuje kontroli typów statycznie (w czasie komplikacji), zawsze gdy jest to tylko możliwe. Oznacza to, że programista jest powiadamiany o błędach możliwie jak najszybciej, a także to, że program jest bardziej efektywny. Jednak dołączenie prywatnej implementacji pociąga za sobą dwa skutki — implementacja jest widoczna, nawet jeżeli nie ma do niej łatwego dostępu, a ponadto może ona wywoływać niepotrzebnie powtórkompilację programu.

Ukrywanie implementacji

W przypadku niektórych projektów nie wolno dopuścić do tego, by ich implementacja była widoczna dla klienta-programisty. Plik nagłówkowy biblioteki może zawierać informacje o znaczeniu strategicznym, których firma nie zamierza udostępniać konkurentom. Być może pracujesz nad systemem, w którym istotną kwestię stanowi bezpieczeństwo — na przykład algorytm szyfrowania — i nie chcesz umieszczać w pliku nagłówkowym informacji, które mogłyby ułatwić złamanie kodu. Albo zamierzasz umieścić swoją bibliotekę we „wrogim” środowisku, w którym programiści i tak będą odwoływać się do prywatnych składowych klasy — wykorzystując wskaźniki i rzutowanie. We wszystkich takich przypadkach lepiej skompilować rzeczywistą strukturę klasy wewnętrz pliku, zawierającego implementację, niż ujawniać ją w pliku nagłówkowym.

Ograniczanie powtórkompilacji

Menedżer projektu, dostępny w używanym przez ciebie środowisku programistycznym, spowoduje powtórkompilację każdego pliku, jeśli został on zmodyfikowany, lub jeżeli został zmieniony plik, od którego jest on zależny — czyli dołączony do niego plik nagłówkowy. Oznacza to, że ilekroć dokonywana jest zmiana dotycząca klasy (niezależnie od tego, czy dotyczy ona deklaracji jej publicznego interfejsu, czy też składowych prywatnych), jesteś zmuszony do powtórznej komplikacji wszystkich plików, do których dołączony jest plik nagłówkowy tej klasy. Często jest to określane mianem *problemu wrażliwej klasy podstawowej*. W przypadku wczesnych etapów realizacji dużych projektów może to być irytujące, ponieważ wewnętrzna implementacja podlega częstym zmianom — gdy projekt taki jest bardzo obszerny, czas potrzebny na komplikacje niekiedy uniemożliwia szybkie wprowadzanie w nim zmian.

Technika rozwiązująca ten problem nazywana jest czasami *klasami-uchwytami* (ang. *handle classes*) lub „kotem z Cheshire”² — wszystko, co dotyczy implementacji znika i pozostaje tylko pojedynczy wskaźnik — „uśmiech”. Wskaźnik odnosi się do struktury, której definicja znajduje się w pliku zawierającym implementację, wraz z wszystkimi definicjami funkcji składowych. Tak więc dopóki nie zmieni się interfejs, dopóty plik nagłówkowy pozostaje niezmieniony. Implementacja może być dowolnie zmieniana w każdej chwili, powodując jedynie konieczność ponownej komplikacji i powtórnego połączenia z projektem pliku zawierającego implementację.

² Nazwa ta jest przypisywana Johnowi Carolanowi, jednemu z pionierów programowania w C++ i, oczywiście, Lewisowi Carollowi. Można ją również postrzegać jako formę „pomostowego” wzorca projektowego, opisanego w drugim tomie książki.

Poniżej zamieszczono prosty przykład, demonstrujący wykorzystanie tej techniki. Plik nagłówkowy zawiera wyłącznie publiczny interfejs klasy oraz wskaźnik do (nie w pełni określonej) klasy:

```
//: C05:Handle.h
// Klasy-uchwyty
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
    struct Cheshire; // Tylko deklaracja klasy
    Cheshire* smile;
public:
    void initialize();
    void cleanup();
    int read();
    void change(int);
};

#endif // HANDLE_H ///:-
```

To wszystko, co widzi klient-programista. Wiersz:

```
struct Cheshire;
```

stanowi *niepełną specyfikację typu* albo *deklarację klasy* (*definicja* klasy zawierałaby jej ciało). Informuje ona kompilator, że **Cheshire** jest nazwą struktury, lecz nie dostarcza żadnych szczegółów na jej temat. Informacja wystarcza jedynie do utworzenia wskaźnika do tej struktury — nie można utworzyć obiektu, dopóki nie zostanie udostępnione jej ciało. W przypadku zastosowania tej techniki ciało to jest ukryte w pliku zawierającym implementację:

```
//: C05:Handle.cpp {0}
// Implementacja uchwytu
#include "Handle.h"
#include "../require.h"

// Definicja implementacji uchwytu:
struct Handle::Cheshire {
    int i;
};

void Handle::initialize() {
    smile = new Cheshire;
    smile->i = 0;
}

void Handle::cleanup() {
    delete smile;
}

int Handle::read() {
    return smile->i;
}

void Handle::change(int x) {
    smile->i = x;
} ///:-
```

Cheshire jest strukturą zagnieźdzoną, musi więc ona zostać zdefiniowana w zasięgu klasy:

```
struct Handle::Cheshire {
```

W funkcji **Handle::initialize()** strukturze **Cheshire** przydzielana jest pamięć, która jest później zwalniana przez funkcję **Handle::cleanup()**. Pamięć ta jest używana zamiast wszystkich elementów danych, które są zazwyczaj umieszczane w prywatnej części klasy. Po skompilowaniu pliku **Handle.cpp** definicja tej struktury zostaje ukryta w pliku wynikowym i nie jest ona dla nikogo widoczna. Jeżeli następuje zmiana elementów struktury **Cheshire**, to jedynym plikiem, który musi zostać powtórnie skompilowany, jest **Handle.cpp**, ponieważ plik nagłówkowy pozostanie niezmieniony.

Sposób użycia klasy **Handle** przypomina wykorzystywanie każdej innej klasy — należy dołączyć jej plik nagłówkowy, utworzyć obiekty i wysyłać do nich komunikaty:

```
//: C05:UseHandle.cpp
//{L} Handle
// Używanie klasy-uchwytu
#include "Handle.h"

int main() {
    Handle u;
    u.initialize();
    u.read();
    u.change(l);
    u.cleanup();
} //:-
```

Jedyną rzeczą, do której ma dostęp klient, jest publiczny interfejs klasy. Dopóki więc zmienia się jedynie jej implementacja, powyższy plik nie będzie nigdy wymagał powtarnej komplikacji. Tak więc, mimo że nie jest to doskonały sposób ukrycia implementacji, stanowi on w tej dziedzinie ogromny krok naprzód.

Podsumowanie

Kontrola dostępu w języku C++ zapewnia programistę ścisły nadzór nad utworzoną przez siebie klasą. Użytkownicy klasy widzą w przejrzysty sposób, czego mogą używać, a co powinni zignorować. Jeszcze ważniejsza jest możliwość gwarancji, że żaden klient-programista nie będzie uzależniony od jakiegokolwiek części kodu tworzącego wewnętrzną implementację klasy. Dzięki temu twórcy klasy mogą zmieniać wewnętrzną implementację, wiedząc, że żaden z klientów-programistów nie zostanie zmuszony do wprowadzania jakichkolwiek modyfikacji w swoim programie, ponieważ nie ma on dostępu do tej części klasy.

Mając możliwość zmiany wewnętrznej implementacji, można nie tylko udoskonalić swój projekt, ale również pozwolić sobie na popełnianie błędów. Bez względu na to, jak skrupulatnie zaplanuje się wszystko i wykona, i tak dojdzie do pomyłek. Wiedza, że popełnianie takich błędów jest stosunkowo bezpieczne, umożliwia eksperymentowanie, efektywniejszą naukę i szybsze zakończenie projektu.

Publiczny interfejs klasy jest tym, co widzi klient-programista, należy więc we właściwy sposób przemyśleć go w trakcie analizy i projektowania. Lecz nawet on pozostawia pewną możliwość wprowadzania zmian. Jeżeli postać interfejsu nie zostanie od razu gruntownie przemyślana, można *uzupełnić* go o nowe funkcje, pod warunkiem, że nie zostaną z niego usunięte te spośród funkcji, które zostały już użyte przez klientów-programistów.

Ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Utwórz klasę posiadającą dane składowe publiczne, prywatne oraz chronione. Utwórz obiekt tej klasy i zobacz, jakie komunikaty kompilatora uzyskasz, próbując odwołać się do wszystkich danych składowych klasy.
2. Utwórz strukturę o nazwie **Lib**, zawierającą trzy obiekty będące łańcuchami (**string**): **a**, **b** oraz **c**. W funkcji **main()** utwórz obiekt o nazwie **x** i przypisz wartości składowym **x.a**, **x.b** oraz **x.c**. Wydrukuj te wartości. Następnie zastąp składowe **a**, **b** i **c** tablicą, zdefiniowaną jako **string s[3]**. Zauważ, że w rezultacie dokonanej zmiany przestanie działać kod, zawarty w funkcji **main()**. Teraz utwórz klasę o nazwie **Libc**, zawierającą prywatne składowe, będące łańcuchami **a**, **b** i **c**, a także funkcje składowe **seta()**, **geta()**, **setb()**, **getb()**, **setc()** i **getc()**, umożliwiające ustawianie i pobieranie wartości składowych. W podobny sposób jak poprzednio napisz funkcję **main()**. Teraz zastąp prywatne składowe **a**, **b** i **c**, prywatną tablicą **string s[3]**. Zauważ, że mimo dokonanych zmian, kod zawarty w funkcji **main()** nie przestał działać poprawnie.
3. Utwórz klasę i globalną funkcję, będącą „przyjacielem”, operującą na prywatnych danych tej klasy.
4. Utwórz dwie klasy, tak by każda z nich posiadała funkcję składową, przyjmującą wskaźnik do obiektu drugiej klasy. W funkcji **main()** utwórz egzemplarze obu obiektów i wywołaj w każdym z nich wymienione wcześniej funkcje składowe.
5. Utwórz trzy klasy. Pierwsza z nich powinna zawierać dane prywatne, a także wskazać jako swoich „przyjaciół” całą drugą klasę oraz funkcję składową trzeciej klasy. Zademonstruj w funkcji **main()**, że wszystko działa poprawnie.
6. Utwórz klasę **Hen**. Umieść wewnątrz niej klasę **Nest**. Wewnątrz klasy **Nest** ulokuj klasę **Egg**. Każda z klas powinna posiadać funkcję składową **display()**. W funkcji **main()** utwórz obiekty każdej z klas i wywołaj dla każdego z nich funkcję **display()**.
7. Zmodyfikuj poprzednie ćwiczenie w taki sposób, aby klasy **Nest** i **Egg** zawierały dane prywatne. Określ „przyjaciół” tych klas, tak aby do ich danych prywatnych miały dostęp klasy, w których są one zagnieżdżone.

8. Utwórz klasę, której dane składowe zawarte będą w regionach: publicznym, prywatnym i chronionym. Dodaj do klasy funkcję składową `showMap()`, drukującą nazwy oraz adresy każdej z tych danych składowych. Jeżeli to możliwe, skompiluj i uruchom program, używając różnych kompilatorów, komputerów i systemów operacyjnych. Obserwuj, czy zmienia się układ danych składowych w pamięci.
9. Skopiuj pliki zawierające implementacje i testy programu **Stash**, zawartego w rozdziale 4., tak aby je skompilować i uruchomić razem z zawartym w bieżącym rozdziale plikiem **Stash.h**.
10. Zapamiętaj obiekty klasy **Hen**, utworzonej w 6. ćwiczeniu, używając do tego klasy **Stash**. Pobierz je ponownie, anastępnie wydrukuj (jeżeli jeszcze nie zostało to wykonane, należy utworzyć funkcję składową `Hen::print()`).
11. Skopiuj pliki zawierające implementacje i testy programu **Stack**, zawartego w rozdziale 4., tak aby je skompilować i uruchomić razem z zawartym w bieżącym rozdziale plikiem **Stack2.h**.
12. Zapamiętaj obiekty klasy **Hen**, utworzonej w 6. ćwiczeniu, używając do tego klasy **Stack**. Pobierz je z powrotem ze stosu, a następnie wydrukuj (jeżeli jeszcze nie zostało to wykonane, należy utworzyć funkcję składową `Hen::print()`).
13. Zmodyfikuj strukturę **Cheshire**, zawartą w pliku **Handle.cpp**, i sprawdź, czy twój menedżer powtórnie skompiluje i połączy jedynie ten plik, nie komplilując ponownie pliku **UseHandle.cpp**.
14. Utwórz klasę **StackOfInt** (stos przechowujący wartości całkowite) w klasie o nazwie **StackImp**. Użyj do tego celu techniki „kota z Cheshire”, ukrywającej niskopoziomowe struktury danych, stosowane do przechowywania elementów. Zaimportuj dwie wersje klasy **StackImp** — wykorzystującą tablicę liczb całkowitych o stałym rozmiarze i stosującą `typvector<int>`. Ustaw maksymalną wielkość stosu, by nie uwzględniać powiększania tablicy w pierwszym z tych przypadków. Zwróć uwagę na to, że opis klasy, zawarty w pliku **StackOfInt.h**, nie zmienia się podczas dokonywania zmian w klasie **StackImp**.

Rozdział 6.

Inicjalizacja

i końcowe porządkи

W rozdziale 4. osiągnęliśmy znaczący postęp w zakresie używania bibliotek, łącząc rozproszone komponenty typowej biblioteki języka C i zamykając je w strukturze (abstrakcyjnym typie danych, nazywanym *klasą*).

Dzięki temu nie tylko jest udostępniane jedyne, ujednolicone wejście do komponentu biblioteki, ale ukrywa się również nazwy poszczególnych funkcji w obrębie nazwy klasy. W rozdziale 5. zostało wprowadzone pojęcie kontroli dostępu (ukrywania implementacji). Umożliwia ona projektantowi klasy zdefiniowanie wyraźnych granic, określających, czym może się posługiwać **klient-programista**, a co znajduje się poza jego zasięgiem. Oznacza to, że wewnętrzne mechanizmy działania typu danych pozostają pod kontrolą projektanta klasy i ich postać jest zależna od jego woli, natomiast dla **klientów-programistów** jest zupełnie oczywiste, na które składowe klasy mogą i powinni oni zwrócić uwagę.

Kapsułkowanie wraz z kontrolą dostępu stanowią istotny krok na drodze do ułatwień w dziedzinie wykorzystywania bibliotek. Tworzone przez nie pojęcie „nowego typu danych” jest pod pewnymi względami lepsze niż wbudowane typy danych, dostępne w języku C. Kompilator języka C++ **może** obecnie zapewnić kontrolę typów w odniesieniu do takich „nowych typów danych”, gwarantując zarazem bezpieczeństwo podczas ich wykorzystywania.

W sprawach dotyczących bezpieczeństwa kompilator języka C++ zapewnia znacznie większe możliwości niż język C. W tym i w następnych rozdziałach zostaną przedstawione dodatkowe cechy języka C++, dzięki którym błędy zawarte w kodzie niemal samoczynnie się ujawniają — czasami nawet zanim jeszcze skompiluje się program, lecz najczęściej w postaci komunikatów o błędach i ostrzeżeń kompilatora. Dzięki temu już wkrótce przyzwyczaisz się do nieprawdopodobnego scenariusza, według którego program w języku C++, który się kompiluje, często od razu działa poprawnie.

Dwoma kwestiami związanymi z bezpieczeństwem są: inicjalizacja i „sprzątanie”. Przyczyną licznych błędów w języku C było to, że programista zapomniał o inicjalizacji

zmiennej lub o przeprowadzeniu „końcowych porządków”. Zdarza się to szczególnie w przypadku bibliotekęzyka C, gdy klient-programista nie wie, w jaki sposób zainicjalizować strukturę, albo nawet nie jest świadomego tego, że powinien to zrobić (biblioteki często nie zawierają funkcji inicjalizujących, co sprawia, że klient-programista jest zmuszony do samodzielnego inicjalizowania struktur). Końcowe porządkie stanowią szczególny problem, ponieważ programiści języka C są przyzwyczajeni do zapominania o zmiennych, które już nie istnieją. W związku z tym wszelkie sprzątanie, które mogłoby być konieczne w przypadku zawartych w bibliotekach struktur, jest często pomijane.

W języku C++ pojęcia inicjalizacji i sprzątania są niezbędnym elementem łatwego korzystania z bibliotek oraz sposobem eliminacji wielu trudno uchwytnych błędów, występujących wówczas, gdy klient-programista zapomni o wykonaniu tych działań. W rozdziale omówiono właściwości języka C++, pomocne w zapewnieniu odpowiedniej inicjalizacji i sprzątania.

Konstruktor gwarantuje inicjalizację

Klasy **Stash** i **Stack**, zdefiniowane w poprzednim rozdziale, posiadają funkcję **initialize()**, która, jak wskazuje na to jej nazwa, należy wywołać, zanim w jakikolwiek sposób zostanie użyty obiekt. Niestety, oznacza to, że właściwą inicjalizację musi w tym przypadku zapewnić klient-programista. Jednakże **klienci-programiści**, dając do rozwiązania swoich problemów za pomocą twojej fantastycznej biblioteki, mają tendencję do zapominania o takich drobiazgach, jak inicjalizacja. W języku C++ inicjalizacja jest zbyt istotna, by pozostawić ją klientowi-programistie. Projektant klasy może zapewnić inicjalizację każdego obiektu, dostarczając specjalną funkcję, nazywaną **konstruktorem**. Jeżeli klasa posiada konstruktor, kompilator automatycznie wywołuje go w miejscu, w którym tworzony jest obiekt, zanim jeszcze klient-programista będzie mógł podjąć jakiekolwiek działania z nim związane. Wywołanie konstruktora nie zależy w żaden sposób od woli klienta-programisty — jest dokonywane przez kompilator w miejscu, w którym definiowany jest obiekt.

Kolejnym problemem jest wybór nazwy tej funkcji. Wiążą się z tym dwie kwestie. Po pierwsze, każda nazwa może potencjalnie kolidować z nazwami, używanymi w charakterze składowych klas. Po drugie, skoro kompilator jest odpowiedzialny za wywołanie konstruktora, to zawsze musi wiedzieć, którą funkcję powinien wywołać. Rozwiązanie wybrane przez Stroustrupą wydaje się najprostsze i najbardziej logiczne — nazwa konstruktora jest taka sama, jak nazwa klasy. Logiczne jest, że taka funkcja zostanie wywołana automatycznie w czasie inicjalizacji.

Poniżej zamieszczono przykład prostej klasy posiadającej konstruktor:

```
class X {  
    int i;  
public:  
    X(); // Konstruktor  
};
```

Obecnie, gdy obiekt jest definiowany:

```
void f() {  
    X a;  
    // ...  
}
```

zachodzi to samo, co w przypadku gdyby zmienna a była liczbą całkowitą — obiekowi przydzielana jest pamięć. Lecz kiedy program dochodzi do *punktu sekwencyjnego* (miejscia wykonania), w którym zdefiniowana jest zmienna a, następuje automatyczne wywoływanie konstruktora. Oznacza to, że w miejscu definicji kompilator „po cichu” wstawia wywołanie funkcji **X::X()** dla obiektu a. Podobnie jak w przypadku każdej funkcji składowej, pierwszym (niewidocznym) argumentem konstruktora jest wskaźnik **this** — adres obiektu, dla którego został on wywoływany. Jednak w przypadku konstruktora wskaźnik **this** wskazuje na niezainicjowany blok pamięci, a jego prawidłowa inicjalizacja jest właśnie zadaniem konstruktora.

Podobnie jak w przypadku każdej innej funkcji, konstruktor może posiadać argumenty pełniące rozmaite funkcje: umożliwiające określenie, w jaki sposób tworzony jest obiekt, przekazujące wartości inicjujące itd. Argumenty konstruktora są sposobem gwarantującym, że wszystkie elementy obiektu są inicjowane odpowiednimi wartościami. Na przykład jeżeli klasa **Tree** (drzewo) zawiera konstruktor, pobierający pojedynczy argument całkowity, oznaczający wysokość drzewa, to jej obiekt musi zostać utworzony w następujący sposób:

```
Tree t(12); // 12-metrowe drzewo
```

Jeżeli **Tree(int)** jest jedynym konstruktorem, kompilator nie dopuści do utworzenia obiektu w żaden inny sposób (w następnym rozdziale zostaną przedstawione wielokrotne konstruktory, a także różne sposoby wywoływania konstraktorów).

Oto cała wiedza o konstruktorach — są to funkcje o specjalnych nazwach, wywoływanie automatycznie przez kompilator dla każdego obiektu, w miejscu jego tworzenia. Pomimo swojej prostoty, są one wyjątkowo cenne, pozwalają bowiem na wyeliminowanie dużej klasy problemów, powodując równocześnie, że kod jest łatwiejszy zarówno do napisania, jak i do przeczytania. Na przykład w przedstawionym powyżej fragmencie kodu nie sposób dostrzec jawnego wywołania żadnej funkcji **initialize()**, które byłoby oderwane pojęciowo od inicjalizacji. W języku C++ definicja i inicjalizacja są pojęciami połączonymi ze sobą — nie można używać jednego z nich w oderwaniu od drugiego.

Zarówno konstruktor, jak i destruktor są osobliwymi rodzajami funkcji — nie zwracają one żadnej wartości. Jest to czymś zupełnie innym niż zwracanie wartości typu **void**; w tym przypadku funkcja co prawda niczego nie zwraca, ale zawsze istnieje możliwość zmiany tej sytuacji. Konstruktory i destruktory nie zwracają niczego i nie można tego zmienić. Działania polegające na umieszczeniu obiektu w programie, a później jego usunięciu są czymś szczególnym, podobnie jak narodziny i śmierć, i kompilator zawsze sam wywołuje funkcje konstruktora i destruktora, by mieć pewność, że zostały one wykonane. Gdyby zwracały one jakąś wartość i można było ją dowolnie wybrać, to kompilator musiałby skądś wiedzieć, co ma zrobić z tą wartością; albo też **klient-programista** byłby zmuszony do jawnego wywoływania konstruktorów i destruktów, co z kolei wyeliminowałoby związane z nimi bezpieczeństwo.

Destruktor gwarantuje sprzątanie

Jako programista języka C pewnie często zastanawiałeś się nad tym, jak ważna jest inicjalizacja, mniej natomiast interesowało cię sprzątanie. W końcu, na czym miałyby polegać sprzątanie po liczbie całkowitej? — można o tym zapomnieć! Jednakże w przypadku bibliotek „porzucenie” niepotrzebnego już obiektu nie jest tak bezpieczne. Co się stanie, jeżeli zmienił on **stan jakiegoś** elementu sprzętowego, narysował coś na ekranie lub przydzielił pamięć na stercie? Jeżeli o tym zapomnisz, usuwany obiekt nigdy nie zostanie w pełni zamknięty. W języku C++ sprzątanie jest równie ważne, jak inicjalizacja, i dlatego właśnie jest ono gwarantowane przez destruktora.

Składnia destruktora jest podobna do składni konstruktora — w charakterze nazwy funkcji używaną jest nazwa klasy. Jednakże destruktory różni się od konstruktora tym, że jego nazwę poprzedza znak tyldy (~). W dodatku destruktory nie wymagają nigdy żadnych argumentów. Poniżej przedstawiono deklarację destruktora:

```
class Y {  
public:  
    ~Y();  
};
```

Destruktor jest wywoływany przez kompilator automatycznie, gdy kończy się zasięg **obiektu**. O ile miejsce wywołania konstruktora jest widoczne (jest nim miejsce definicji obiektu), o tyle jedynym świadectwem wywołania destruktora jest nawias klamrowy, zamkujący zasięg, w którym znajduje się obiekt. Mimo to destruktory jest **wywoływany**, nawet jeżeli używa się instrukcji **goto**, wychodząc z zasięgu (instrukcja **goto** istnieje nadal w języku C++ w celu zapewnienia wstępnej zgodności z językiem C, a także z tego powodu, że zdarzają się sytuacje, w których okazuje się ona przydatna). Należy pamiętać o tym, że instrukcja **dalekiego goto**, zaimplementowana w postaci funkcji **setjmp()** oraz **longjmp()**, zawartych w standardowej bibliotece funkcji języka C, nie powoduje wywołania destruktorek (tak głosi jej specyfikacja — nawet jeżeli używany przez ciebie kompilator wywołuje w takim przypadku destruktory, to poleganie na właściwości, która nie została uwzględniona w specyfikacji, oznacza, że napisany w taki sposób program jest nieprzenośny).

Poniżej zamieszczono przykładowy program, demonstrujący przedstawione do tej pory właściwości konstruktorów i destruktorek:

```
//: C06:Constructor1.cpp  
// Konstruktory i destruktory  
#include <iostream>  
using namespace std;  
  
class Tree {  
    int height;  
public:  
    Tree(int initialHeight); // Konstruktor  
    ~Tree(); // Destruktor  
    void grow(int years);  
    void printszie();
```

```

};

Tree::Tree(int initialHeight) {
    height = initialHeight;
}

Tree::~Tree() {
    cout << "wewnatrz destruktora drzewa" << endl;
    printsize();
}

void Tree::grow(int years) {
    height += years;
}

void Tree::printsizeC() {
    cout << "Wysokosc drzewa wynosi " << height << endl;
}

int main() {
    cout << "przed klamrowym nawiasem otwierajacym" << endl;
{
    Tree t(12);
    cout << "po utworzeniu drzewa" << endl;
    t.printsize();
    t.grow(4);
    cout << "przed klamrowym nawiasem zamykajacym" << endl;
}
    cout << "po klamrowym nawiasie zamykajacym" << endl;
} //:-
}

```

A oto wyniki działania powyższego programu:

```

przed klamrowym nawiasem otwierajacym
po utworzeniu drzewa
Wysokosc drzewa wynosi 12
przed klamrowym nawiasem zamykajacym
wewnatrz destruktora drzewa
Wysokosc drzewa wynosi 16
po klamrowym nawiasie zamykajacym

```

Destruktor jest zatem wywoływany automatycznie w miejscu klamrowego nawiasu, zamykającego **zasięg**, w którym zdefiniowano obiekt.

Eliminacja bloku definicji

W języku C wszystkie zmienne **musiały** być zawsze zdefiniowane na początku bloku, po klamrowym nawiasie otwierającym. Nie jest to jakieś wyjątkowe wymaganie wśród języków **programowania**; jako jego uzasadnienie często podaje się **argument**, że jest to „dobry styl programowania”. Autor niniejszej książki ma jednak co do tego wątpliwości. Jako programista zawsze uważałem, że jest niewygodne przeskakiwanie do początku bloku po to, by utworzyć jakąś nową zmienną. Zauważałem **również**, że kod programu jest bardziej czytelny, gdy definicje zmiennych znajdują się blisko **miejsca**, w którym są używane.

Być może są to argumenty stylistyczne. Jednak w języku C++ występuje poważny problem, związany z **definicją** wszystkich zmiennych na początku zasięgu. Jeżeli istnieje konstruktor, to musi on zostać wywołany podczas tworzenia obiektu. Jednak, jeżeli ten konstruktor przyjmie jeden lub więcej argumentów, służących do inicjalizacji, to skąd wiadomo, że informacja potrzebna do inicjalizacji będzie znana już na początku zasięgu? Na ogół nie będzie ona znana. Ponieważ w języku C nie występuje pojęcie prywatności, oddzielenie definicji i inicjalizacji nie stanowi problemu. Jednak język C++ gwarantuje, że tworzony obiekt jest równocześnie inicjalizowany. Dzięki temu w systemie nie ma nigdy żadnych niezainicjowanych obiektów. Język C nie zwraca na to uwagi — co więcej, *wspiera* praktykę tworzenia niezainicjowanych zmiennych, wymagając zdefiniowania ich na początku bloku, zanim jeszcze dostępna jest informacja niezbędna do ich **inicjalizacji**!

Generalnie język C++ nie pozwala na utworzenie obiektu, zanim nie zostaną przygotowane informacje, potrzebne konstruktorowi do jego inicjalizacji. Z uwagi na to język, który wymuszałby definicje zmiennych na początku zasięgu, nie byłby możliwy do zrealizowania. W rzeczywistości styl języka wydaje się zachęcać do definiowania obiektów w miejscu możliwe bliskim ich użycia. W C++ każda reguła dotycząca „obiektów” automatycznie odnosi się również do obiektów wbudowanych typów. Oznacza to, że zarówno obiekty dowolnego typu, jak i zmienne wbudowanych typów, mogą być definiowane w dowolnym miejscu zasięgu. Ponadto z definicjami zmiennych można poczekać — do miejsca, w którym będą dostępne inicjujące je wartości, dzięki czemu zawsze będzie można równocześnie definiować i inicjować:

```
//: C06:DefineInitialize.cpp
// Definiowanie zmiennych w dowolnych miejscach
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class G {
    int i;
public:
    G(int ii);
};

G::G(int ii) { i = ii; }

int main() {
    cout << "wartosc inicjujaca? ";
    int retval = 0;
    cin >> retval;
    require(retval != 0);
    int y = retval + 3;
    G g(y);
} // -
```

Wykonywany jest zatem pewien kod, po czym jest definiowana i inicjalizowana zmienna **retval**, następnie wykorzystywana do pobrania wartości wpisanej przez

¹ Język C99, zaktualizowana wersja standardu C, pozwala na definicje zmiennych w dowolnym miejscu zasięgu, podobnie jak C++.

użytkownika. W dalszej kolejności definiowane są zmienne y oraz g. Z drugiej strony język C nie pozwala na definiowanie zmiennych w jakimkolwiek innym miejscu niż na początku zasięgu.

Generalnie należy definiować zmienne możliwie jak najbliżej miejsca ich wykorzystania i zawsze inicjalizować je podczas definicji (w przypadku typów wbudowanych, których inicjalizacją jest opcjonalna, to jedynie sugestia stylistyczna). Kwestia ta wiąże się również z bezpieczeństwem. Ograniczając obszar, w którym zmienna jest dostępna w zasięgu, zmniejsza się prawdopodobieństwo jej nieprawidłowego użycia w jakimś innym miejscu zasięgu. Dodatkowo uzyskuje się większą czytelność kodu, ponieważ nie trzeba już przeskakiwać do początku zasięgu i z powrotem, by dowiedzieć się, jakiego typu jest zmienna.

Pętle for

W języku C++ często występuje licznik pętli for, zdefiniowany bezpośrednio w wyrażeniu pętli:

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

Powyższe wyrażenia są ważnymi przypadkami szczególnymi, wprawiającymi często w zakłopotanie początkujących programistów C++.

Zmienne i oraz j zostały zdefiniowane bezpośrednio w wyrażeniu pętli **for** (czego nie można wykonać w języku C), można więc ich użyć w pętli **for**. To bardzo wygodna konstrukcja, ponieważ kontekst sprawia, że rozwiewają się wszelkie wątpliwości dotyczące przeznaczenia zmiennych i i j; nie ma więc potrzeby używania tak niezgrabnych nazw, jak **i_licznik_petli**.

Możesz się jednak rozczarować, jeśli sądzisz, że czas życia zmiennych i i j wykracza poza zasięg pętli for —jest on ograniczony zasięgiem pętli².

W rozdziale 3. wspomniano, że instrukcje **while** i **switch** również pozwalają na definiowanie obiektów w obrębie swoich wyrażeń kontrolnych, chociaż takie ich zastosowanie wydaje się znacznie mniej istotne niż w przypadku pętli for.

Należy zwrócić uwagę na zmienne lokalne, które zasłaniają zmienne znajdujące się w bardziej zewnętrznym zasięgu. Na ogół użycie tych samych nazw dla zmiennej zagnieżdzonej i zmiennej globalnej w stosunku do jej zasięgu jest mylące i może być przyczyną błędów³.

² Wcześniejsza wersja projektu standardu języka C++ głosiła, że czas życia tej zmiennej rozciąga się do końca zasięgu, w którym zawarta jest pętla **for**. Niektóre kompilatory nadal działają w taki sposób, ale nie jest to poprawne. Dlatego też twój kod będzie przenośny tylko wówczas, gdy ograniczysz zasięg zmiennej do pętli **for**.

³ Język Java uznał to za na tyle zły pomysł, że oznacza taki kod jako błędny.

Autor książki zauważył, że niewielkie zasięgi są cechą dobrych projektów. Jeżeli kod pojedynczej funkcji zajmuje wiele stron, to być może zakres jej działania jest zbyt duży. Bardziej rozdrobnione funkcje są nie tylko bardziej użyteczne, ale łatwiej w nich również znaleźć błędy.

Prydzielanie pamięci

Zmienne mogą być obecnie definiowane w dowolnym miejscu zasięgu. Wydawałoby się więc, że również ich pamięć może nie być określona, aż do miejsca wystąpienia definicji. Obecnie jest jednak bardziej prawdopodobne, że kompilator zadziała zgodnie z praktyką stosowaną w języku C, przydzielając całą pamięć dla zasięgu w miejscu, w którym znajduje się kwadratowy nawias otwierający, rozpoczęty ten zasięg. Nie ma to żadnego znaczenia, ponieważ programista i tak nie ma dostępu do tego obszaru pamięci (znanego również pod nazwą obiektu), dopóki nie zostanie on zdefiniowany⁴. Mimo że pamięć jest przydzielana na początku bloku, wywołanie konstruktora następuje dopiero w punkcie sekwencyjnym, w którym definiowany jest obiekt, ponieważ jego identyfikator nie jest wcześniej dostępny. Kompilator sprawdza nawet, czy definicja obiektu (a zatem i wywołanie konstruktora) nie została umieszczona w miejscu, w którym sterowanie tylko warunkowo przechodzi przez punkt sekwencyjny, jak na przykład w instrukcji **switch** lub w takim, które może przeskoczyć instrukcja **goto**. Usunięcie komentarzy, w których umieszczone są instrukcje poniższego programu, spowoduje zgłoszenie ostrzeżenia lub błędu:

```
//: C06:Nojump.cpp
// Należy można przeskakiwać konstruktory

class X {
public:
    X();
};

X::X() {}

void f(int i) {
    if(i < 10) {
        //! goto jump1; // Błąd: goto pomija inicjalizację
    }
    X x1; // Wywołanie konstruktora
jump1:
    switch(i) {
        case 1 :
            X x2; // Wywołanie konstruktora
            break;
        //! case 2 : // Błąd: case pomija inicjalizację
        X x3; // Wywołanie konstruktora
        break;
    }
}

int main() {
    f(9);
    f(11);
}///:-
```

⁴ W porządku — można to zrobić, bawiąc się wskaźnikami, ale byłoby to niezwykle nierozsądne.

W powyższym kodzie zarówno instrukcja **goto**, **jak** i **switch** mogą potencjalnie przeskoczyć przez punkt sekwencyjny, w którym wywoływany jest konstruktor. Obiekt taki znajdowałby się w zasięgu, nawet jeżeli nie byłby wywołany jego konstruktor; kompilator zgłasza więc komunikat o błędzie. Ponownie gwarantuje to, że obiekty nie mogą być tworzone, jeżeli nie są one również inicjalizowane.

Całe omawiane tutaj przydzielanie pamięci odbywa się, oczywiście, na stosie. Pamięć jest przydzielana przez kompilator przez przesunięcie wskaźnika stosu „w dół” (to pojęcie względne, które może oznaczać zwiększenie lub zmniejszanie aktualnej wartości wskaźnika stosu, zależnie od rodzaju używanego komputera). Obiekty mogą być również tworzone na stercie za pomocą operatora **new**, co zostanie omówione dokładniej w rozdziale 13.

Klasa Stash z konstruktorami i destruktormi

W przykładach z poprzednich rozdziałów są przedstawione funkcje, które w oczywisty sposób odpowiadają konstruktorom i destruktorm: **initialize()** i **cleanup()**. Poniżej zaprezentowano plik nagłówkowy klasy **Stash**, używający konstruktorów i destruktorek:

```
//: C06:Stash2.h
// Z konstruktorami i destruktormi
#ifndef STASH2_H
#define STASH2_H

class Stash {
    int size;           // Wielkość każdego elementu
    int quantity;      // Liczba elementów pamięci
    int next;          // Następny pusty element
    // Dynamicznie przydzielana tablica bajtów:
    unsigned char* storage;
    void inflate(int increase);
public:
    Stash(int size);
    ~Stash();
    int add(void* element);
    void* fetch(int index);
    int count();
};

#endif // STASH2_H III-
```

Jedynymi definicjami funkcji składowych, które uległy zmianie, są **initialize()** i **cleanup()**, zastąpione przez konstruktor i destruktur.

```
//: C06:Stash2.cpp {0}
// Konstruktor i destruktory
#include "Stash2.h"
#include "../require.h"
#include <iostream>
```

```
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}

int Stash::add(void* element) {
    if(next >= quantity) // Czy wystarczy pamięci?
        inflate(increment);
    // Kopiowanie elementu do pamięci.
    // poczawszy od następnego wolnego miejsca:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return(next - 1); // Numer indeksu
}

void* Stash::fetch(int index) {
    require(0 <= index, "Stash::fetch indeks ma wartosc ujemna");
    if(index >= next)
        return 0; // Oznaczenie końca
    // Tworzenie wskaźnika do żądanego elementu:
    return &(storage[index * size]);
}

int Stash::count() {
    return next; // Liczba elementów w Stash
}

void Stash::inflate(int increase) {
    require(increase > 0,
           "Stash::inflate increase ma wartosc zerowa lub ujemna");
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Kopiowanie starego obszaru do nowego
    delete [] (storage); // Stary obszar pamięci
    storage = b; // Wskaźnik do nowego obszaru
    quantity = newQuantity;
}

Stash::~Stash() {
    if(storage != 0) {
        cout << "zwalnianie pamięci" << endl;
        delete [] storage;
    }
} //:-
```

Do śledzenia błędów programisty, zamiast funkcji **assert()**, zostały użyte funkcje zawarte w pliku **require.h**. Informacje wyświetlane w przypadku błędu przez funkcję **assert()** nie są tak użyteczne, jak informacje wyświetlane przez funkcje pochodzące z **require.h** (zostaną one zaprezentowane w dalszej części książki).

Ponieważ funkcja **inflate()** jest składową prywatną, jedynym przypadkiem, w którym wywołanie **require()** mogłoby zakończyć się niepowodzeniem, jest przypadkowe przekazanie funkcji **inflate()** niepoprawnej wartości przez inną funkcję składową klasy. Jeżeli masz pewność, że nie może się to zdarzyć, można rozważyć usunięcie wywołania **require()**. Należy jednak pamiętać, że dopóki treść klasy nie zostanie ustalona, istnieje zawsze możliwość dodania do niej jakiegoś nowego kodu, który stanie się przyczyną błędów. Koszt wywołania **require()** jest niewielki (a poza tym wywołanie to może być automatycznie usunięte za pomocą preprocesora), natomiast korzyści wynikające z uzyskania niezawodnego kodu — znaczne.

Zwróć uwagę na to, że definicje obiektów klasy **Stash**, widoczne w poniższym programie, występują bezpośrednio przed miejscami, w których obiekty te są potrzebne, a także na to, że inicjalizacja wydaje się elementem definicji, znajdującym się na liście argumentów konstruktora:

```
//: C06-Stash2Test.cpp
//{L} Stash2
// Konstruktory i destruktory
#include "Stash2.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
    Stash intStash(sizeof(int));
    for(int i = 0; i < 100; i++)
        intStash.add(&i);
    for(int j = 0; j < intStash.count(); j++)
        cout << "intStash.fetch(" << j << ") = "
            << *(int*)intStash.fetch(j)
            << endl;
    const int bufsize = 80;
    Stash stringStash(sizeof(char) * bufsize);
    ifstream in("Stash2Test.cpp");
    assure(in, "Stash2Test.cpp");
    string line;
    while(getline(in, line))
        stringStash.add((char*)line.c_str());
    int k = 0;
    char* cp;
    while((cp = (char*)stringStash.fetch(k++))!=0)
        cout << "stringStash.fetch(" << k << ") - "
            << cp<< endl;
} //:-
```

Zwróć również uwagę na to, w jaki sposób wyeliminowano wywołania funkcji `cleanup()`. Jednakże destruktory są nadal wywoływane automatycznie, gdy kończy się zasięg obiektów `iniStash` i `stringStash`.

Warto podnieść pewną kwestię dotyczącą klasy **Stash**. Autor książki zwrócił szczególną uwagę na to, by używać jedynie wbudowanych typów danych — to znaczy typów **nieposiadających** destruktorów. Podjęcie próby skopiowania do klasy **Stash** obiektów innych klas spowodowałoby lawinę wszelkich możliwych problemów i program nie działałby poprawnie. Standardowa biblioteka języka C++ obecnie poprawnie kopiuje obiekty zawarte w swoich kontenerach, ale jest to raczej proces kłopotliwy i skomplikowany. W zamieszczonym poniżej przykładzie wykorzystania klasy **Stack** zastosowano wskaźniki w celu uniknięcia tego problemu. W następnym rozdziale również klasa **Stash** zostanie zmodyfikowana w taki sposób, by używała wskaźników.

Klasa Stack z konstruktorami i destruktormi

Powtórna implementacja listy powiązanej (zawartej w klasie **Stack**), wykorzystująca konstruktory i destruktory, pokazuje, jak elegancko działają one z operatorami `new` i `delete`. Poniżej przedstawiono zmodyfikowany plik nagłówkowy:

```
//: C06:Stack3.h
// Z konstruktorami i destruktormi
#ifndef STACK3_H
#define STACK3_H

class Stack {
    struct Link {
        void* data;
        Link* next;
        Link(void* dat, Link* nxt);
        ~Link();
    }* head;
public:
    Stack();
    ~Stack();
    void push(void* dat);
    void* peek();
    void* pop();
};

#endif // STACK3_H ///:-
```

Klasa **Stack** posiada nie tylko konstruktor i destruktur, ale również zagnieżdżoną klasę **Link**:

```
//: C06:Stack3.cpp {0}
// Konstruktory i destruktory
#include "Stack3.h"
#include "../require.h"
using namespace std;
```

```
Stack::Link::Link(void*dat, Link*nxt) {
    data = dat;
    next = nxt;
}

Stack::Link::~Link() { }

Stack::Stack() { head = 0; }

void Stack::push(void* dat) {
    head = new Link(dat,head);
}

void* Stack::peek() {
    require(head != 0, "Stos jest pusty");
    return head->data;
}

void* Stack::pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}

Stack::~Stack() {
    require(head == 0, "Stos nie jest pusty");
} //:-)
```

Konstruktor **Link()::Link()** inicjalizuje wskaźniki **data** i **next**, więc wiersz:

```
head = new Link(dat,head);
```

zawarty w funkcji **Stack::push()**, nie tylko przydziela pamięć nowemu obiektovi klasy **Link** (wykorzystując dynamiczne tworzenie obiektów za pomocą słowa klu-czowego **new**, opisane w rozdziale 4.), ale również w elegancki sposób inicjalizuje jego wskaźniki.

Można by zadać pytanie, dlaczego destruktor struktury **Link** nie wykonuje żadnych działań — w szczególności, dlaczego nie usuwa za pomocą operatora **delete** obszaru pamięci wskazywanego przez **data**? Wynika to z dwóch problemów. W rozdziale 4., w którym został po raz pierwszy przedstawiony program **Stack**, wyjaśniono, że nie sposób poprawnie usunąć obszaru wskazywanego przez wskaźnik typu **void***, jeżeli wskazuje on obiekt (twierdzenie to zostało udowodnione w rozdziale 13.). Ponadto gdyby destruktor struktury **Link** usuwał obszar wskazywany **data**, to w efekcie funk-cja **pop()** wróciłaby wskaźnik do usuniętego obiektu, co z pewnością byłoby błędem. Jest to czasami nazywane problemem *prawa własności*: struktura **Link**, a więc również klasa **Stack**, przechowuje jedynie wskaźniki, ale nie odpowiadająca usunięcie wskazywanych przez nie obiektów. Trzeba zatem zwrócić szczególną uwagę na to, by wiedzieć, kto jest za to odpowiedzialny. Na przykład jeżeli nie pobierzesz za pomocą funkcji **pop()** wszystkich wskaźników znajdujących się na stosie i nie usu-niesz — za pomocą **delete** — wskazywanych przez nie obiektów, to nie zostaną one usunięte automatycznie przez destruktor klasy **Stack**. Może być to złożony problem,

prowadzący do „wyciekania” pamięci. Dlatego też różnica pomiędzy dobrze i źle działającym programem może sprowadzać się do wiedzy na temat tego, kto odpowiada za usunięcie obiektu. Z tego właśnie powodu funkcja `Stack::~Stack()` wyświetla komunikat o błędzie, jeśli obiekt klasy `Stack` nie jest w czasie destrukcji pusty.

Z uwagi na to, że tworzenie i usuwanie obiektów struktury `Link` jest ukryte wewnątrz klasy `Stack` — a więc stanowi część jej wewnętrznej implementacji — nie zdołasz dostrzec operacji w programie testowym. To jednak *ty* ponosisz odpowiedzialność za usunięcie obiektów, wskazywanych przez wartości zwarcane przez funkcję `pop()`:

```
//: C06:Stack3Test.cpp
//{L} Stack3
//{T} Stack3Test.cpp
// Konstruktory i destruktory
#include "Stack3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // Argumentem jest nazwa pliku
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack textlines;
    string line;
    // Odczytanie pliku i zapamiętanie wierszy na stosie:
    while(getline(in, line))
        textlines.push(new string(line));
    // Pobranie wierszy ze stosu i wydrukowanie ich:
    string* s;
    while((s = (string*)textlines.pop()) != 0) {
        cout << *s << endl;
        delete s;
    }
} //:-
```

W powyższym przykładzie wszystkie wiersze, znajdujące się na stosie `textlines`, zostały pobrane i usunięte. Jednak gdyby się tak nie stało, to wywołanie `require()` wyświetliłoby komunikat oznaczający, że nastąpił wyciek pamięci.

Iinicjalizacja agregatowa

Agregat (ang. *aggregate*) jest dokładnie tym, co to słowo oznacza — grupą połączonych ze sobą elementów. Definicja ta obejmuje agregaty mieszanych typów, takie jak np. struktury czy klasy. Tablice są agregatami elementów jednego typu.

Iinicjalizacja agregatów może stanowić źródło błędów i jest zajęciem nużącym. Znacznie bezpieczniejsza jest, dostępna w języku C++, *inicjalizacja agregatowa* (ang. *aggregate initialization*). Podczas tworzenia obiektu będącego agregatem należy jedynie dokonać przypisania, a inicjalizacja zostanie przeprowadzona przez kompilator.

W zależności od rodzaju agregatu, przypisanie to może mieć wiele postaci, ale zawsze elementy zawarte w przypisaniu muszą znajdować się w nawiasie klamrowym. W przypadku tablicy elementów wbudowanego typu jest to dość proste:

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

W razie podania większej liczby inicjatorów niż wynosi liczba elementów tablicy, kompilator zgłosi komunikat o błędzie. Co jednak się stanie w przypadku, gdy podanych inicjatorów będzie *zbyt mało*? Na przykład:

```
int b[6] = {0};
```

Wówczas kompilator używa pierwszego inicjatora do zainicjowania pierwszego elementu tablicy, a następnie wartości zerowej dla wszystkich elementów, dla których nie podano inicjatorów. Należy pamiętać, że działanie takie nie ma miejsca w przypadku, gdy tablica została zdefiniowana bez podania listy inicjatorów. Tak więc powyższe wyjaśnienie jest zwięzlym sposobem inicjalizacji wszystkich elementów tablicy wartością **zerową**, bez potrzeby używania do tego pętli for i bez możliwości popełnienia błędu, polegającego na niewłaściwym indeksowaniu elementów tablicy (w zależności od kompilatora, sposób ten może być również znacznie bardziej efektywny niż wykorzystanie pętli **for**).

Drugim skrótem, stosowanym w przypadku tablic, jest **automatyczne zliczanie** (ang. **automatic counting**), polegające na tym, że kompilator wyznacza rozmiar tablicy na podstawie liczby inicjatorów:

```
int c[] = { 1, 2, 3, 4 };
```

Jeżeli zdecydujesz się teraz na dodanie do tablicy jeszcze jednego elementu, to należy jedynie dopisać jeszcze jeden inicjator. Napisanie kodu w taki sposób, by dokonywane zmiany dotyczyły tylko jednego miejsca, zmniejsza prawdopodobieństwo popełnienia błędu w czasie jego modyfikacji. Jak jednak uzyskać informację na temat wielkości takiej tablicy? Umożliwia to sztuczka działająca w sposób **niezależny** od zmian wielkości tablicy, czyli zastosowanie wyrażenia **sizeof c / sizeof *c** (wielkość całej tablicy, podzielona przez wielkość jej pierwszego elementu)⁵:

```
for(int i = 0; i < sizeof c / sizeof *c; i++)
    c[i]++;
```

Struktury są również agregatami, a zatem mogą być inicjalizowane w podobny sposób. Ponieważ struktura, zdefiniowana w stylu języka C, posiada wszystkie składowe publiczne, można im bezpośrednio przypisać wartości:

```
struct X {
    int i;
    float f;
    char c;
};

X xl = { 1, 2.2, 'c' };
```

⁵ W drugim tomie książki (dostępnym **bezpłatnie** w witrynie www.BruceEckel.com) przedstawiony zostanie **bardziej** zwięzły sposób wyznaczenia wielkości tablicy za pomocą szablonów.

W przypadku tablicy takich obiektów można je zainicjować, używając dla każdego obiektu zagnieżdżonego nawiasu klamrowego:

```
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
```

W powyższym przykładzie trzeci element tablicy został zainicjowany wartością zerową.

Jeżeli którakolwiek składowa jest prywatna (co jest typowe w przypadku dobrze zaprojektowanych klas w języku C++) lub nawet gdy wszystkie dane składowe są publiczne, ale zdefiniowano **konstruktor**, inicjalizacja odbywa się w inny sposób. W powyższych przykładach inicjatory są przypisywane bezpośrednio elementom aggregatu, natomiast konstruktory są sposobem wymuszenia inicjalizacji obiektu za pośrednictwem „oficjalnego” interfejsu. W celu dokonania inicjalizacji muszą więc zostać wywołane konstruktory. A zatem w przypadku struktury o następującej postaci:

```
struct Y {
    float f;
    int i;
    Y(int a);
};
```

należy wskazać wywołania konstruktorów. Najlepiej zrobić to w jawnym sposób, jak w poniższym przykładzie:

```
Y y1[J = { Y(1), Y(2), Y(3) }];
```

Występują tu trzy obiekty i trzy wywołania konstruktorów. Jeśli zdefiniowany jest konstruktor, niezależnie od tego, czy jest to struktura z wszystkimi składowymi publicznymi, czy też klasa o prywatnych danych składowych, inicjalizacja musi odbywać się za pośrednictwem konstruktora — nawet gdy jest wykorzystywana inicjalizacja agregatowa.

Poniżej zamieszczono drugi przykład, prezentujący konstruktory posiadające wiele argumentów:

```
//: C06:Multiarg.cpp
// Konstruktory z wieloma argumentami.
// używane do inicjalizacji aggregatowej
#include <iostream>
using namespace std;

class L {
    int i, j;
public:
    Z(int ii, int jj):
        void print();
};

Z::Z(int ii, int jj) {
    i = ii;
    j = jj;
}

void Z::print() {
    cout << "i = " << i << ", j = " << j << endl;
}
```

```
int main() {  
    Z zz[] = { Z(1.2), Z(3.4), Z(5.6), Z(7,8) };  
    for(int i = 0; i < sizeof zz / sizeof *zz; i++)  
        zz[i].print();  
} //~-
```

Wygląda to w taki sposób, jakby konstruktor był wywoływany jawnie dla każdego obiektu znajdującego się w tablicy.

Konstruktory domyślne

Konstruktorem domyślnym (ang. *default constructor*) jest konstruktor, który może być wywoływany bez argumentów. Służy on do tworzenia „zwykłych obiektów”, ale pełni również funkcję w przypadku, gdy kompilator ma za zadanie utworzenie jakiegoś obiektu, ale nie zostaną podane mu żadne dodatkowe informacje. Na przykład jeżeli struktury Y, zdefiniowanej w poprzednim przykładzie, używa się w następującej definicji:

```
Y y2[2] = { Y(1) };
```

to kompilator zgłosi, że nie może odnaleźć domyślnego konstruktora tej struktury. Drugi obiekt, znajdujący się w tablicy, ma zostać utworzony bez argumentów, więc kompilator poszukuje jego domyślnego konstruktora. W istocie, jeżeli zdefiniuje się po prostu tablicę obiektów struktury Y:

```
Y y3[7];
```

to kompilator zgłosi błąd, ponieważ do zainicjowania **każdego** obiektu, znajdującego się w tablicy, potrzebny jest konstruktor domyślny.

Ten sam problem występuje w przypadku tworzenia pojedynczych obiektów, jak np.:

```
Y y4;
```

Pamiętaj, że jeżeli istnieje konstruktor, kompilator gwarantuje, że zostanie on wykonyany *zawsze*, niezależnie od sytuacji.

Domyślny konstruktor odgrywa ważną rolę — *jeżeli* (i tylko jeżeli) struktura (lub klasa) nie posiada w ogóle konstruktorów, kompilator automatycznie tworzy konstruktor domyślny. Dzięki temu działa zamieszczony poniżej przykład:

```
//: C06:AutoDefaultConstructor.cpp  
// Automatycznie tworzony konstruktor domyślny  
  
class V {  
    int i; // składowa prywatna  
}: // Brak konstruktora  
  
int main() {  
    V v; v2[10];  
} //:-~
```

Gdyby jednak zostały zdefiniowane jakiekolwiek konstruktory i nie byłoby wśród nich konstruktora domyślnego, próba utworzenia widocznych powyżej obiektów klasy V spowodowałaby wystąpienie błędów komplikacji.

Można by odnieść wrażenie, że konstruktor utworzony przez kompilator powinien dokonać jakiejś inteligentnej inicjalizacji, polegającej na wyzerowaniu całego obszaru pamięci przydzielonej obiektowi. Tak jednak nie jest — stanowiłoby to dodatkowy narzut, znajdujący się poza kontrolą programisty. Aby pamięć obiektu została wyzerowana, należy uczynić to samodzielnie — w jawnym sposób tworząc konstruktor domyślny.

Mimo że kompilator sam tworzy konstruktor domyślny, to jego działanie rzadko odpowiada rzeczywistym potrzebom. Należy traktować tę właściwość języka jako „siatkę zabezpieczającą”, wykorzystującą jednak z umiarem. Na ogół trzeba jawnie zdefiniować konstruktory, nie pozwalając na to, by zrobił to za ciebie kompilator.

Podsumowanie

Pozornie skomplikowane mechanizmy, dostępne w języku C++, powinny dostarczyć ci wyraźnych wskazówek, dotyczących szczególnej roli inicjalizacji i sprzątania. Jedną z pierwszych obserwacji dotyczących wydajności programowania w języku C, które dokonał Stroustrup podczas projektowania języka C++, była ta, że wiele problemów programistycznych jest spowodowanych nieprawidłową inicjalizacją zmiennych. Błędy tego typu są trudne do wykrycia, a podobne problemy związane są również z nieprawidłowym „sprzątaniem”. Ponieważ konstruktory i destruktory gwarantują dokonanie właściwej inicjalizacji i sprzątania (kompilator nie dopuszcza do utworzenia i zniszczenia obiektu bez wywołania odpowiedniego konstruktora i destruktora), używa się dzięki nim pełną kontrolę nad programem oraz bezpieczeństwo.

Inicjalizacją agregatową została zaprojektowana w podobnym duchu — zapobiega ona typowym pomyłkom, związanym z inicjalizacją agregatów wbudowanych typów, czyniąc kod programu bardziej zwięzły.

Bezpieczeństwo związane z pisaniem kodu jest w języku C++ niezwykle istotną kwestią. Jego ważnym elementem jest inicjalizacją i sprzątanie, ale w dalszej części książki zapoznasz się również z innymi problemami dotyczącymi bezpieczeństwa

Ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Utwórz prostą klasę o nazwie **Simple**, posiadającą konstruktor drukujący cokolwiek — po to, by było wiadomo, że został wywołany. Utwórz obiekt tej klasy w funkcji **main()**.
2. Do klasy, utworzonej w poprzednim ćwiczeniu, dodaj destruktora, drukującą cokolwiek — aby było wiadomo, że został on wywołany.
3. Zmodyfikuj poprzednie ćwiczenie w taki sposób, aby klasa zawierała składową całkowitą. Zmodyfikuj konstruktor tak, aby pobierał on argument całkowity i zapisywał go w składowej klasy. Zarówno konstruktor, jak i destruktory powinny drukować wartość tej składowej w swoich komunikatach, umożliwiając obserwację tworzonych i niszczonych obiektów.
4. Wykaż, że **destruktory** jest wywoływanym nawet w przypadku, gdy do opuszczenia pętli jest używana instrukcja **goto**.
5. Utwórz dwie pętle **for** drukujące wartości od zera do dziesięciu. Zdefiniuj licznik pierwszej pętli przed instrukcją **for**, a drugiej — w wyrażeniu sterującym pętli. W ramach drugiej części ćwiczenia zmień identyfikator licznika drugiej pętli w taki sposób, aby miał on taką samą nazwę, jak licznik pierwszej pętli. Prześledź, co zrobi kompilator.
6. Zmodyfikuj pliki **Handle.h**, **Handle.cpp**, i **UseHandle.cpp**, znajdujące się na końcu 5. rozdziału w taki sposób, by wykorzystywały konstruktory i destruktory.
7. Użyj inicjalizacji agregatowej do utworzenia tablicy elementów typu **double**, w której definicji określisz wielkość tablicy, ale nie dostarczysz dostatecznej liczby inicjatorów. Wydrukuj zawartość tablicy, stosując operator **sizeof** do uzyskania informacji ojej wielkości. Następnie utwórz tablicę wartości typu **double**, używając inicjalizacji agregatowej oraz automatycznego zliczania. Wydrukuj zawartość tablicy.
8. Użyj inicjalizacji agregatowej do utworzenia tablicy obiektów typu **string** (łańcuchów). Utwórz klasę **Stack** (stos), przechowującą łańcuchy, a następnie przejdź przez wszystkie elementy tablicy łańcuchów, umieszczając je kolejno na stosie. Wreszcie pobierz wszystkie łańcuchy ze stosu za pomocą funkcji **pop()**, drukując każdy z nich.
9. Zademonstruj automatyczne zliczanie i inicjalizację agregatową tablicy obiektów klasy, utworzonej w ćwiczeniu 3. Dodaj do tej klasy funkcję **składową**, drukującą komunikat. Oblicz wielkość tablicy, a następnie przejdź przez wszystkie znajdujące się w niej obiekty, wywołując dla każdego z nich dodaną funkcję składową.
10. Utwórz klasę pozabawioną wszelkich konstruktorów i wykaż, że wykorzystując domyślny konstruktor, możesz utworzyć jej obiekty. Następnie utwórz dla tej klasy konstruktor niebędący domyślnym konstruktorem (**posiadający jakiś argument**) i spróbuj skompilować program ponownie. Wyjaśnij, co się stało.

Rozdział 7.

Przeciążanie nazw funkcji i argumenty domyślne

Jedną z istotnych cech każdego języka programowania jest wygodne używanie nazw.

Tworząc obiekt (zmienną), nadaje się nazwę określonemu obszarowi pamięci. Funkcja jest nazwą pewnego działania. Określając samodzielnie nazwy, opisujące system, tworzymy program, który jest łatwiejszy do zrozumienia i do modyfikacji. Przypomina to pisanie prozą — głównym celem jest komunikacja z czytelnikami.

Problemy pojawiają się, gdy usiłujemy oddać subtelności pojęć języka naturalnego w języku programowania. To samo słowo, w zależności od kontekstu, oznacza często zupełnie co innego. Jedno słowo mające wiele znaczeń określamy mianem *przeciążonego* (ang. *overloaded*). To bardzo wygodne, zwłaszcza gdy dotyczy prostych różnic. Mówimy: „umyj samochód, umyj ręce”. Byłoby bezsensowne, gdybyśmy byli zmuszeni do mówienia: „samochód_umyj samochód, ręce_umyj ręce” — po prostu dlatego, że słuchacz nie musi dokonywać żadnego rozróżnienia pomiędzy wykonywanymi czynnościami. Języki naturalne zawierają w sobie pewną nadmiarowość, więc nawet w przypadku opuszczenia kilku słów, możemy nadal określić znaczenie wypowiedzi. Nie potrzebujemy unikatowych identyfikatorów — potrafimy wywnioskować znaczenie z kontekstu.

Jednak większość języków programowania wymaga nadania unikatowego identyfikatora każdej funkcji. Jeżeli zamierzasz wydrukować wartości trzech różnych typów danych: **int**, **char** i **float**, to na ogół musisz utworzyć trzy różne nazwy funkcji, np. **print_int()**, **print_char()** oraz **print_float()**. Wymaga to sporego nakładu dodatkowej pracy — dla ciebie, w czasie pisania programu, a także dla osoby, która będzie próbowała go zrozumieć.

W języku C++ jest jeszcze jeden czynnik, wymuszający przeciążanie nazw funkcji — konstruktor. Z uwagi na to, że nazwa konstruktora jest zdeterminowana nazwą klasy, wydawałoby się, że klasa może zawierać tylko jeden konstruktor. Co jednak zrobić w przypadku, gdy chcemy utworzyć obiekt w więcej niż jeden sposób? Na przykład założmy, że tworzymy klasę, która może się albo zainicjować sama, w standardowy

sposób, albo czytając informacje z pliku. Potrzebne są dwa konstruktory — jeden, niepobierający argumentów (konstruktor domyślny), oraz drugi, który pobiera argument typu string, będący nazwą pliku inicjującego obiekt. Oba są konstruktorami, muszą więc posiadać tę samą nazwę — nazwę klasy. Tak więc przeciążanie nazw funkcji jest niezbędne, by ta funkcja o tej samej nazwie — w tym przypadku konstruktor — mogła być używana z różnymi typami argumentów.

Mimo że przeciążanie nazw funkcji jest w przypadku konstruktorów konieczne, to stanowi ono ułatwienie o charakterze ogólnym, które może być używane z każdą funkcją, a nie tylko z funkcjami składowymi klas. Ponadto przeciążanie nazw funkcji oznacza, że jeżeli dwie różne biblioteki posiadają funkcje o tych samych nazwach, to nie będą one ze sobą kolidowały, pod warunkiem, że różnią się listami argumentów. W niniejszym rozdziale przyjrzymy się bliżej wszystkim tym czynnikom.

Tematem rozdziału jest wygodne używanie nazw funkcji. Przeciążanie ich pozwala na stosowanie tej samej nazwy dla różnych funkcji, jest jednak jeszcze jeden sposób, ułatwiający wywoływanie funkcji. Jak postąpić w przypadku, gdybyśmy chcieli wywołać tę samą funkcję na różne sposoby? Jeżeli funkcja ma długą listę argumentów, a większość z nich jest taka sama dla wszystkich jej wywołań, to wpisywanie wywołań funkcji staje się uciążliwe (a program — trudny w czytaniu). Powszechnie stosowaną właściwością języka C++ są *domyślne argumenty*. Domyślny argument jest argumentem wstawianym przez kompilator w przypadku, gdy nie został on określony w wywołaniu funkcji. A zatem wywołania: `f("cześć")`, `f("hej", 1)` i `f("witaj", 2, 'c')` mogą być wywołaniami tej samej funkcji. Mogą one stanowić również wywołania trzech różnych funkcji o przeciążonych nazwach, ale jeśli listy argumentów są podobne, oczekujemy zazwyczaj również podobnego zachowania, wymagającego pojedynczej funkcji.

Przeciążanie nazw funkcji oraz domyślne argumenty nie są naprawdę niczym bardzo skomplikowanym. Po zapoznaniu się z tym rozdziałem będzie już wiadomo, kiedy ich używać i jak działa wewnętrzny mechanizm, odpowiedzialny za ich obsługę w czasie kompilacji i łączenia.

Dalsze uzupełnienia nazw

W rozdziale 4. zostało wprowadzone pojęcie *uzupełnienia nazw* (ang. *name decoration*). W poniższym przykładzie:

```
void f();  
class X { void f(); };
```

funkcja `f()`, znajdująca się wewnątrz zasięgu klasy X, nie koliduje z globalną wersją funkcji `f()`. Kompilator uwzględnia ten zasięg, tworząc różne wewnętrzne nazwy dla globalnej wersji funkcji `f()` oraz dla funkcji `X::f()`. W rozdziale 4. zasugerowano, że nazwy te są nazwą klasy, uzupełnioną nazwą funkcji, a zatem wewnętrzne nazwy, używane przez kompilator, mogą mieć postać: `_f` i `_X_f`. Okazuje się jednak, że uzupełnienie nazwy funkcji zawiera coś więcej niż tylko nazwę klasy.

Oto przyczyna. Założymy, że zamierzamy przeciążyć dwie nazwy funkcji:

```
void print(char);
void print(float);
```

Nie ma znaczenia, czy obie funkcje znajdują się wewnątrz klasy, czy też w zasięgu globalnym. Kompilator nie może utworzyć unikatowych wewnętrznych identyfikatorów, używając jedynie zasięgu nazw funkcji, bo w obu przypadkach byłaby to nazwa `_print`. Ideą używania przeciążonych nazw funkcji jest stosowanie takich samych ich nazw, ale różnych list argumentów. A zatem, aby przeciążanie działało poprawnie, kompilator musi uzupełniać nazwy funkcji nazwami typów argumentów. W przypadku powyższych funkcji, zdefiniowanych w zasięgu globalnym, wewnętrzne nazwy, utworzone przez kompilator, mogą mieć postać: `_print_char` i `_print_float`. Warto w tym miejscu nadmienić, że nie ma standardowego sposobu, w jaki muszą być uzupełniane nazwy, więc dla różnych kompilatorów można uzyskać różne rezultaty (aby zobaczyć, jak to działa, należy nakazać kompilatorowi generowanie kodu w asemblerze). Oczywiście, powoduje to problemy w przypadku zamiaru nabycia skompilowanych bibliotek, przeznaczonych dla konkretnego kompilatora i programu łączącego. Jednak nawet gdyby uzupełnienia nazw miały postać standardową, i tak pozostałaby bariera, spowodowana sposobem, w jaki różne kompilatory generują kod.

To już naprawdę wszystko, jeżeli chodzi o przeciążanie nazw funkcji — można używać tej samej nazwy dla różnych funkcji, pod warunkiem, że różnią się one listami argumentów. Kompilator uzupełnia nazwy, wykorzystując zasięg i listy argumentów — tworzy w ten sposób wewnętrzne nazwy, wykorzystywane przez siebie oraz przez program łączący.

Przeciążanie na podstawie zwracanych wartości

Dość powszechną reakcją jest zdziwienie: „Dlaczego tylko zasięgi i listy argumentów? Dlaczego nie zwracane wartości?”. Dodatkowe uzupełnianie wewnętrznych nazw funkcji zwracanymi wartościami wydaje się rozsądne na pierwszy rzut oka. Można by wówczas przeciążać nazwy funkcji również na podstawie zwracanych wartości:

```
void f();
int f();
```

Działa to wspaniale, jeśli kompilator jest w stanie jednoznacznie określić znaczenie na podstawie kontekstu, jak np. w instrukcji `int x = f();`. Jednakże w języku C można było zawsze wywołać funkcję, ignorując zwracaną przez nią wartość (tj. wywołać funkcję dlatego *skutków ubocznych*). W jaki sposób kompilator ma odróżnić, o wywołanie której funkcji w takim przypadku chodzi? Być może jeszcze gorsze jest to, że osoba czytająca kod też będzie miała trudności z ustaleniem, która funkcja jest wywoływana. Przeciążanie wyłącznie na podstawie zwracanej wartości jest zbyt trudno uchwytne i dlatego niejest dozwolone w języku C++.

Łączenie bezpieczne dla typów

Z wszystkich, opisanych powyżej, uzupełnień nazw wynika dodatkowa korzyść. Przyczyną szczególnie skomplikowanych problemów w języku C było niepoprawne zadeklarowanie funkcji przez klienta-programistę lub, co gorsza, wywołanie funkcji w ogóle bez jej uprzedniej deklaracji, co powodowało, że kompilator wnioskował o deklaracji funkcji na podstawie sposobu, w jaki była ona wywołana. Czasami taka deklaracja była poprawna, ale w przeciwnym przypadku mogła stanowić trudny do wykrycia błąd.

Ponieważ w języku C++ wszystkie funkcje *muszą* zostać przed użyciem zadeklarowane, możliwość zaistnienia takiego błędu jest bardzo ograniczona. Kompilator języka C++ odmawia automatycznego deklarowania funkcji za programistę, jest więc prawdopodobne, że do programu zostanie dołączony odpowiedni plik nagłówkowy. Jeżeli programista z jakiegoś powodu zdoła jednak błędnie zadeklarować funkcję, działając samodzielnie lub dołączając niewłaściwy plik nagłówkowy (być może po prostu **nieaktualny**), uzupełnienia nazw zapewnia „siatkę zabezpieczającą”, nazywaną często *łączeniem bezpiecznym dla typów* (ang. *type-safelinkage*).

Weź pod uwagę następujący scenariusz. W pierwszym pliku znajduje się definicja funkcji:

```
//: C07:Def.Cpp {0}
// Definicja funkcji
void f(int) {}
///-
```

W drugim natomiast funkcja jest niepoprawnie zadeklarowana, a następnie wywołana:

```
//: C07:Use.Cpp
//{L} Def
// Błędna deklaracja funkcji
void f(char);

int main() {
    f(1); // Powoduje błąd programu łączącego
} //
```

Mimo że funkcja ma w rzeczywistości postać `f(int)`, to kompilator o tym nie wie, ponieważ poinformowano go — za pomocą jawniej deklaracji — że jest to funkcja `f(char)`. Dlatego też komplikacja przebiegła poprawnie. W języku C łączenie również zakończyłoby się pomyślnie, ale *nie* w języku C++. Ponieważ kompilator uzupełnia nazwy, definicja uzyska postać w rodzaju `f_int`, podczas gdy użycie funkcji ma postać `f_char`. Kiedy program łączący próbuje ustalić odwołanie do `f_char`, może odnaleźć jedynie `f_int` i zgłasza komunikat o błędzie. Jest to właściwie łączenie bezpieczne dla typów. Chociaż problem ten nie występuje często, to kiedy się pojawi, może być niewiarygodnie trudny do wykrycia — szczególnie w dużych projektach. Jest to jeden z przypadków, w którym można w łatwy sposób odnaleźć poważny błąd w programie napisanym w języku C, kompilując go za pomocą kompilatora C++.

Przykładowe przeciążenie

Możemy obecnie zmodyfikować wcześniejsze przykłady, tak by zawierały one przeciążenia nazw funkcji. Jak już napisano powyżej, miejscem, w którym od razu przydają się przeciążenia, są konstruktory. Można się o tym przekonać w zamieszczonej poniżej wersji klasy **Stash**:

```
//: C07:Stash3.h
// Przeciążanie nazw funkcji
#ifndef STASH3_H
#define STASH3_H

class Stash {
    int size;      // Wielkość każdego elementu
    int quantity; // Liczba elementów pamięci
    int next;     // Następny pusty element
    // Dynamicznie przydzielana tablica bajtów:
    unsigned char* storage;
    void inflate(int increase);
public:
    Stash(int size); // Zerowa liczba elementów
    Stash(int size, int initQuantity);
    ~Stash();
    int add(void* element);
    void* fetch(int index);
    int count();
};

#endif // STASH3_H //:-
```

Pierwszy konstruktor **Stash()** jest taki sam, jak poprzednio, lecz drugi posiada argument **Quantity**, określający początkową liczbę elementów, dla których przydzielana jest pamięć. Jak można zobaczyć w definicji, wartość zmiennej **quantity** jest ustalana na zero, podobnie jak wskaźnika **storage**. W drugim konstruktorze wywołanie **inflate(initQuantity)** zwiększa wartość **quantity** do wielkości przydzielonego obszaru pamięci:

```
//: C07:Stash3.cpp {0}
// Przeciążanie nazw funkcji
#include "Stash3.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
    size = sz;
    quantity = 0;
    next = 0;
    storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
    size = sz;
    quantity = 0;
```

```

next = 0;
storage = 0;
inflate(initQuantity);
}

Stash::~Stash() {
    if(storage != 0) {
        cout << "zwalnianie pamięci" << endl;
        delete []storage;
    }
}

int Stash::add(void* element) {
    if(next >= quantity) // Czy wystarczy pamięci?
        inflate(increment);
    // Kopiowanie elementu do pamięci,
    // począwszy od następnego wolnego miejsca:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return(next - 1); // Numer indeksu
}

void* Stash::fetch(int index) {
    require(0 <= index, "Stash::fetch indeks ma wartosc ujemna");
    if(index > next)
        return 0; // Oznaczenie końca
    // Tworzenie wskaźnika do zadanego elementu:
    return &(storage[index * size]);
}

int Stash::count() {
    return next; // Liczba elementów w Stash
}

void Stash::inflate(int increase) {
    assert(increase > 0);
    if(increase == 0) return;
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Kopiowanie starego obszaru do nowego
    delete []storage; // Zwolnienie starego obszaru pamięci
    storage = b; // Wskaźnik do nowego obszaru
    quantity = newQuantity; // Aktualizacja liczby elementów
} //:-

```

W przypadku użycia pierwszego konstruktora wskaźnikowi **storage** nie zostaje przydzielona żadna pamięć. Przydzielenie pamięci odbywa się wówczas, gdy za pomocą funkcji **add()** po raz pierwszy dodawany jest obiekt, a także ilekroć wewnętrz funkcji **add()** zostanie przekroczena aktualna wielkość bloku pamięci.

Oba konstruktory zostały użyte w programie testowym:

```
//: C07:Stash3Test.cpp
//{L} Stash3
// Przeciążanie funkcji
#include "Stash3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
    Stash intStash(sizeof(int));
    for(int i = 0; i < 100; i++)
        intStash.add(&i);
    for(int j = 0; j < intStash.count(); j++)
        cout << "intStash.fetch(" << j << ") - "
            << *(int*)intStash.fetch(j)
            << endl;
    const int bufsize = 80;
    Stash stringStash(sizeof(char) * bufsize, 100);
    ifstream in("Stash3Test.cpp");
    assure(in, "Stash3Test.cpp");
    string line;
    while(getline(in, line))
        stringStash.add((char*)line.c_str());
    int k = 0;
    char* cp;
    while((cp = (char*)stringStash.fetch(k++))!=0)
        cout << "stringStash.fetch(" << k << ") = "
            << cp << endl;
} //:~
```

Wywołanie konstruktora dla obiektu **stringStash** wykorzystuje drugi argument — przypuszczalnie na temat specyfiki rozwiązywanego problemu wiadomo *coś* istotnego, co pozwala na określenie początkowej wielkości klasy **Stash**.

Unie

Jak już wspomniano, w języku C++ jedną różnicą pomiędzy strukturami i klasami jest to, że składowe struktury są domyślnie publiczne, a składowe struktury — domyślnie prywatne. Jak się również można spodziewać, struktury mogą posiadać konstruktory i destruktory. Okazuje się jednak, że także unie mogą posiadać konstruktory, destruktory, funkcje składowe, a nawet kontrolę dostępu. W poniższym przykładzie można ponownie prześledzić użycie przeciążania oraz wynikające z niego korzyści:

```
//: C07:UnionClass.cpp
// Unie z konstruktorami i funkcjami składowymi
#include<iostream>
using namespace std;
```

```

union U {
private: // Kontrola dostępu!
    int i;
    float f;
public:
    U(int a);
    U(float b);
    ~U();
    int read_int();
    float read_float();
};

U::U(int a) { i = a; }

U::U(float b) ( f = b; )

U::~U() { cout << "U::~U()\n"; }

int U::read_int() { return i; }

float U::read_float() { return f; }

int main() {
    U X(12), Y(1.9F);
    cout << X.read_int() << endl;
    cout << Y.read_float() << endl;
} //-

```

Na podstawie powyższego programu możesz wyciągnąć wniosek, że jedyna różnica pomiędzy uniami i klasami sprowadza się do sposobu przechowywania danych (tj. składowe typów **int** i **float** są umieszczone w pokrywających się obszarach pamięci). Unie nie mogą być jednak użyte w charakterze klas podstawowych podczas dziedziczenia, co — z punktu widzenia programowania obiektowego — jest istotnym ograniczeniem (dziedziczenie zostanie przedstawione w rozdziale 14.).

Mimo że funkcje składowe czynią dostęp do unii bardziej „cywilizowanym”, to po inicjalizacji unii nadal nie ma sposobu na to, by powstrzymać klienta-programistę przed wyborem niewłaściwego elementu. W powyższym przykładzie można użyć funkcji **X.read_float()**, choć jest to niepoprawne. Jednakże, „bezpieczna” unia może zostać zamknięta w klasie. Zwróć uwagę na to, w jaki sposób użycie **enum** zwiększa przejrzystość kodu i jak przydatne jest przeciążanie konstruktorów:

```

//: C07:SuperVar.cpp
// Superzmienna
#include <iostream>
using namespace std;

class SuperVar {
    enum {
        character,
        integer,
        floating_point
    } vartype; // Definicja zmiennej
    union { // Anonimowa unia
        char c;

```

```

    int i;
    float f;
};

public:
    SuperVar(char ch);
    SuperVar(int ii);
    SuperVar(float ff);
    void print();
};

SuperVar::SuperVar(char ch) {
    vartype = character;
    c = ch;
}

SuperVar::SuperVar(int ii) {
    vartype = integer;
    i = ii;
}

SuperVar::SuperVar(float ff) {
    vartype = floating_point;
    f = ff;
}

void SuperVar::print() {
    switch (vartype) {
        case character:
            cout << "znak: " << c << endl;
            break;
        case integer:
            cout << "liczba całkowita: " << i << endl;
            break;
        case floating_point:
            cout << "liczba zmiennopozycyjna: " << f << endl;
            break;
    }
}

int main() {
    SuperVar A('c'), B(12), C(1.44F);
    A.print();
    B.print();
    C.print();
} //:-

```

W powyższym programie deklaracja **enum** nie zawiera nazwy typu (**jest wyliczeniem nienazwanym**). Jest to dopuszczalne, jeżeli zamierza się, jak w tym przypadku, od razu zdefiniować egzemplarz zmiennej wyliczeniowej. Nie ma potrzeby odwoływania się do nazwy typu wyliczenia w przyszłości, więc podanie jej jest opcjonalne.

W przypadku unii nie podano ani nazwy typu, ani nazwy zmiennej. Konstrukcja taka jest nazywana **unią anonimową** (ang. *anonymous union*). Rezerwuje ona pamięć dla unii, nie wymaga jednak podawania nazwy zmiennej i kropki przy dostępie do jej elementów. Poniżej zamieszczono przykład wykorzystania unii anonimowej:

```
//: C07:AnonymousUnion.cpp
int main() {
    union {
        int i;
        float f;
    };
    // Dostęp do składowych bez użycia kwalifikatorów:
    i = 12;
    f = 1.22;
} //:~
```

Zwróć uwagę, że do składowych unii anonimowej można się odwoływać, tak jakby były zwykłymi zmiennymi. Jedyna różnica polega na tym, że obie te zmienne zajmują ten sam obszar pamięci. Jeżeli anonimowa unia znajduje się w zasięgu pliku (na zewnątrz wszystkich funkcji i klas), to musi być zadeklarowana jako statyczna, by była łączona wewnętrznie.

Mimo że klasa **SuperVar** jest obecnie bezpieczna, jej użyteczność jest nieco dyskusyjna, z uwagi na to, że z jednej strony używa ona unii, w celu zaoszczędzenia pamięci, natomiast z drugiej — dodanie zmiennej **vartype** zajmuje pamięć w wielkości porównywalnej z wielkością danych zawartymi w unii, co w efekcie niweluje oszczędności. Istnieje jednak wiele sposobów, dzięki którym mogłoby to działać zgodnie z oczekiwaniemi. Gdyby zmienna **vartype** określała typ więcej niż jednego egzemplarza unii — które byłyby w takim przypadku tego samego typu — to dla całej grupy unii wystarczyłaby tylko jedna taka zmienna, a zatem nie zajmowałaby ona tyle miejsca. Bardziej praktycznym sposobem byłoby umieszczenie kodu wykorzystującego zmienną **vartype** w obrębie dyrektywy **#ifdef**, co zapewniłoby, że wszystko działa prawidłowo w trakcie pisania programu i jego testowania, a następnie pozwoliłoby na łatwe pozbycie się zajmowanego miejsca i narzutu czasowego z gotowego kodu.

Argumenty domyślne

Przyjrzymy się dwóm konstruktorom klasy **Stash()**, zdefiniowanym w pliku **Stash3.cpp**. Nie wyglądają one na zupełnie od siebie różne. W istocie, pierwszy z konstruktorów wydaje się szczególnym przypadkiem drugiego, posiadającym zerową wartość argumentu **initQuantity**. Tworzenie i pielegnacja dwóch odmiennych wersji podobnych do siebie funkcji stanowi do pewnego stopnia marnotrawstwo wysiłku programisty.

Lekarstwem na to są *domyślne argumenty*, dostępne w języku C++. Domyślnym argumentem jest wartość, podana w deklaracji, który kompilator wstawi automatycznie, jeśli w czasie wywołania funkcji nie zostanie dostarczony argument. W przypadku klasy **Stash** możemy zastąpić dwie funkcje:

```
Stash(int size); // Zeroła liczba elementów
Stash(int size, int initQuantity):
```

pojedynczą funkcją:

```
Stash(int size, int initQuantity = 0);
```

Definicja konstruktora **Stash(int)** jest usuwana — potrzebna jest wyłącznie jedna definicja funkcji **Stash(int, int)**.

Obecnie dwie zamieszczone poniżej definicje obiektów:

```
Stash A(100). B(100, 0);
```

będą miały taki sam skutek. W obu przypadkach wywołany zostanie identyczny konstruktor. Jednakże co do obiektu A, to drugi argument zostanie automatycznie podstawiony przez kompilator, gdy zauważy on, że pierwszy argument jest typu int, i nie podano drugiego argumentu. Jeżeli kompilator został wcześniej poinformowany o istnieniu argumentu domyślnego, to wie, że będzie mógł wywołać funkcję, podstawiając go jako jej drugi argument, czyli zrobi dokładnie to, czego żądał od niego programista, nadając argumentowi charakter domyślny.

Argumenty domyślne są udogodnieniem, podobnie jak przeciążanie nazw funkcji. Oba mechanizmy umożliwiają wykorzystanie pojedynczej nazwy funkcji w różnych sytuacjach. Różnica polega na tym, że w przypadku argumentów domyślnych kompilator podstawi argumenty w sytuacji, gdy nie chce ich podać programista. Przedstawiony poprzednio przykład stanowi ilustrację trafnego wykorzystania argumentów domyślnych zamiast przeciążonych nazw funkcji — w przeciwnym razie skończyłoby się bowiem na dwóch lub większej liczbie funkcji, wyglądających i działających podobnie. W przypadku gdy funkcje mają zupełnie różne **działanie**, używanie domyślnych argumentów na ogół nie ma sensu (można wówczas zapytać, czy dwie funkcje o zupełnie różnym działaniu powinny nazywać się tak samo).

Istnieją dwie reguły, o których trzeba pamiętać, używając domyślnych argumentów. Po pierwsze, domyślne mogą być tylko końcowe argumenty. Oznacza to, że po domyślnym argumencie nie może następować zwyczajny argument. Po drugie, gdy zacznie się używać domyślnych argumentów w wywołaniu funkcji, wszystkie następne argumenty na liście argumentów tej funkcji muszą być argumentami domyślnymi (wynika to z pierwszej reguły).

Domyślne argumenty są umieszczone wyłącznie w deklaracji funkcji (na ogół znajdują się one w pliku nagłówkowym). Kompilator musi widzieć wartość **domyślną**, zanim będzie mógł jej użyć. Niektórzy dla celów dokumentacyjnych umieszczają wartości domyślnych argumentów w definicji funkcji, w charakterze komentarzy:

```
void fn(int x /* - 0 */) { // ...
```

Argumenty-wypełniacze

W definicji funkcji argumenty mogą być zadeklarowane bez identyfikatorów. Gdy są one używane wraz z domyślnymi argumentami, wyglądają naprawdę zabawnie. W rezultacie można uzyskać:

```
void f(int x. int - 0. float = 1.1);
```

W języku C++ identyfikatory nie są również konieczne w definicjach funkcji:

```
void f(int x. int. float fit) { /* ... */ }
```

W ciele funkcji można się odwołać do argumentów x oraz flt, ale nie do środkowego argumentu, ponieważ nie ma on nazwy. Wywołania funkcji muszą jednak zawierać wartość przeznaczoną dla „wypełniacza”: f(l) lub **f(1,2,3,0)**. Taka konstrukcja pozwala na umieszczenie argumentu w charakterze wypełniacza, bez jego wykorzystywania. Pomysł polega na tym, by można było w przyszłości zmienić definicję funkcji w taki sposób, by używała ona argumentu-wypełniacza, nie uciekając się do zmiany kodu zawierającego wywołania funkcji. Oczywiście można osiągnąć to samo, wykorzystując nazwany argument, ale w przypadku zdefiniowania argumentu, który nie jest używany w ciele funkcji, większość kompilatorów zgłosi ostrzeżenie, zakładając, że został popełniony błąd logiczny. Celowe pominięcie nazwy argumentu zapobiegnie pojawiению się ostrzeżenia.

Co ważniejsze, jeżeli rozpoczniesz używanie argumentu funkcji, a później postanowisz, że nie jest on potrzebny, możesz usunąć go, nie generując ostrzeżeń i nie zakłócając w żaden sposób kodu klienta, wywołującego poprzednią wersję funkcji.

Przeciążanie kontra argumenty domyślne

Zarówno przeciążanie nazw funkcji, jak i argumenty domyślne stanowią udogodnienie w wywoływaniu nazw funkcji. Jednakże czasami trudno określić, której techniki użyć. Jako przykład rozważmy przedstawiony poniżej program pomocniczy, przeznaczony do automatycznego zarządzania blokami pamięci:

```
//: C07:Mem.h
#ifndef MEM_H
#define MEM_H
typedef unsigned char byte;

class Mem {
    byte* mem;
    int size;
    void ensureMinSize(int minSize);
public:
    Mem();
    Mem(int sz);
    ~Mem();
    int mszie();
    byte* pointer();
    byte* pointer(int minSize);
};

#endif // MEM_H //:-
```

Obiekt Mem przechowuje blok bajtów, sprawdzając, czy dostępna jest dostateczna ilość pamięci. Domyślny konstruktor nie przydziela jej, natomiast drugi konstruktor zapewnia, że w obiekcie Mem dostępna jest pamięć o wielkości sz bajtów. Destruktor zwalnia pamięć, funkcja **mszie()** informuje, ile bajtów znajduje się obecnie w obiekcie Mem, a funkcja **pointer()** zwraca wskaźnik do początku obszaru pamięci (klasa Mem to narzędzie dość niskiego poziomu). Istnieje przeciążona wersja funkcji **pointer()**, dzięki której klient-programista może określić, że potrzebny jest mu wskaźnik do bloku pamięci o wielkości przynajmniej minSize, a funkcja składowa go dostarcza.

Zarówno konstruktor, jak i funkcja składowa **pointer()** używają do zwiększenia wielkości bloku pamięci funkcji składowej **private ensureMinSize()** (zwróć uwagę na to, że w przypadku, gdy wielkość pamięci uległa zmianie, przechowywanie wartości zwracanej przez funkcję **pointer()** nie jest bezpieczne).

Poniżej przedstawiono implementację klasy:

```
//: C07:Mem.cpp {0}
#include "Mem.h"
#include <cstring>
using namespace std;

Mem::Mem() { mem = 0; size = 0; }

Mem::Mem(int sz) {
    mem = 0;
    size = 0;
    ensureMinSize(sz);
}

Mem::~Mem() { delete []mem; }

int Mem::msize() { return size; }

void Mem::ensureMinSize(int minSize) {
    if(size < minSize) {
        byte* newmem = new byte[minSize];
        memset(newmem + size, 0, minSize - size);
        memcpy(newmem, mem, size);
        delete []mem;
        mem = newmem;
        size = minSize;
    }
}

byte* Mem::pointer() { return mem; }

byte* Mem::pointer(int minSize) {
    ensureMinSize(minSize);
    return mem;
} III-
```

Nie ulega wątpliwości, że **ensureMinSize()** jest jedyną funkcją odpowiedzialną za przydzielanie pamięci; jest ona wywoływana przez drugi konstruktor oraz przez drugą, przeciążoną postać funkcji **pointer()**. Jeżeli aktualna wielkość bloku pamięci jest dostatecznie duża, to wewnątrz funkcji **ensureMinSize()** nic się nie dzieje. Jeżeli w celu zwiększenia bloku musi zostać przydzielony nowy obszar pamięci (co ma również miejsce po wywołaniu domyślnego konstruktora, gdy blok ma wielkość zerową), to „dodatkowa” porcja pamięci jest wypełniana zerami za pomocą funkcji **memset()**, należącej do standardowej biblioteki języka C (funkcja ta została przedstawiona w rozdziale 5.). Następnie wywoływana jest funkcja **memcpy()**, również należąca do standardowej biblioteki języka C, która w tym przypadku kopiuje istniejące już bajty — z obszaru **mem** do **newmem** (zazwyczaj w efektywny sposób). Wreszcie poprzedni obszar pamięci jest zwalniany, a jej nowy obszar oraz rozmiar są kopiowane do odpowiednich składowych.

Klasa **Mem** została zaprojektowana w taki sposób, by służyła jako narzędzie wewnętrz innych klas, upraszczając ich zarządzanie pamięcią (można jej użyć również po to, by ukryć bardziej wyrafinowany system zarządzania **pamięcia**, udostępniany, na przykład, przez system operacyjny). Takie właśnie zastosowanie klasy **Mem** zostało przetestowane poniżej. Wchodzi ona w skład prostej klasy obsługującej łańcuchy:

```
//: C07:MemTest.cpp
// Test klasy Mem
//{L} Mem
#include "Mem.h"
#include <cstring>
#include <iostream>
using namespace std;

class MyString {
    Mem* buf;
public:
    MyString();
    MyString(char* str);
    ~MyString();
    void concat(char* str);
    void print(ostream& os);
};

MyString::MyString() { buf = 0; }

MyString::MyString(char* str) {
    buf = new Mem(strlen(str) + 1);
    strcpy((char*)buf->pointer(), str);
}

void MyString::concat(char* str) {
    if(!buf) buf = new Mem;
    strcat((char*)buf->pointer(),
           buf->msize() + strlen(str) + 1, str);
}

void MyString::print(ostream& os) {
    if(!buf) return;
    os << buf->pointer() << endl;
}

MyString::~MyString() { delete buf; }

int main() {
    MyString s("Moj testowy 1ancuch");
    s.print(cout);
    s.concat(" cos dodatkowego");
    s.print(cout);
    MyString s2;
    s2.concat("Uzycie domyslnego konstruktora");
    s2.print(cout);
} //:-
```

Wykorzystując powyższą klasę, można jedynie utworzyć obiekt typu **MyString**, dołączyć tekst oraz drukować do strumienia **ostream**. Klasa zawiera wyłącznie wskaźnik do obiektu **Mem**, należy jednak zwrócić uwagę na różnicę pomiędzy domyślnym konstruktorem, ustawiającym ten wskaźnik na zero, i drugim konstruktorem, tworzącym obiekt typu **Mem** i kopującym do niego dane. Przykładem korzyści wynikającej z domyślnego konstruktora jest to, że można utworzyć dużą tablicę pustych obiektów klasy **MyString** niewielkim kosztem, ponieważ wielkość każdego obiektu jest tylko wielkość pojedynczego wskaźnika, a jedynym narzutem, wnoszonym przez domyślny konstruktor – przypisanie wskaźnikowi wartości zerowej. Koszt klasy **MyString**aczyna wzrastać dopiero wówczas, gdy dołączane są dane – w tym momencie jest tworzony obiekt **Mem**, o ile nie stało się to wcześniej. Jeżeli jednak zostanie wykorzystany domyślny konstruktor i nigdy nie będzie wykonywana operacja dołączania danych, to wywołanie destruktora będzie nadal bezpieczne. Wywołanie **delete** dla wartości zerowej jest bowiem zdefiniowane w taki sposób, że nie usiłuje zwalniać pamięci ani nie wywołuje innych niepożądanych skutków.

Gdy przyjrzesz się tym dwóm konstruktorom, to mogą się one na pierwszy rzut oka wydawać głównymi kandydatami do użycia argumentów domyślnych. Jeżeli jednak usuniesz domyślny konstruktor i uzupełnisz pozostały konstruktor o argument domyślny:

```
MyString(char* str = "");
```

wszystko będzie działać poprawnie. Utracisz wszakże poprzednią efektywność konstruktora, ponieważ obiekt **Mem** będzie zawsze tworzony. Aby przywrócić jego efektywność, należy zmodyfikować konstruktor:

```
MyString::MyString(char* str) {
    if(!*str) { // Wskazuje pusty łańcuch
        buf = 0;
        return;
    }
    buf = new Mem(strlen(str) + 1);
    strcpy((char*)buf->pointer(), str);
}
```

W rezultacie oznacza to, że wartość domyślna stanie się znacznikiem, wywołującym wykonanie innego kodu, jeśli zostanie użyta wartość inna niż domyślna. Mimo że w przypadku tak niewielkich konstruktorów jak przedstawiony wygląda to dość nie-winnie, na ogół stosowanie takich praktyk może powodować problemy. Jeżeli musisz sprawdzać, czy argument nie zawiera wartości domyślnej, zamiast traktować gojako zwykłą wartość, to w efekcie zostaną utworzone dwie funkcje, zawarte w ciele pojedynczej funkcji — jedna, obsługująca normalne przypadki, i druga, przeznaczona dla przypadku domyślnego. Równie dobrze można podzielić ją na dwie różne funkcje, zezwalając, by tego wyboru dokonał kompilator. W rezultacie uzyskuje się niewielki (ale zazwyczaj niewidoczny) wzrost wydajności, ponieważ nie jest w tym przypadku przekazywany dodatkowy argument i nie jest również wykonywany dodatkowy kod, związany ze sprawdzeniem warunku. Co ważniejsze, kod wykonywany przez dwie oddzielne funkcje jest rzeczywiście umieszczony w dwóch oddzielnych funkcjach, a nie połączony w jedną za pomocą domyślnego argumentu. Ułatwia to jego pielęgnację — zwłaszcza gdy funkcje te są duże.

Z drugiej strony warto rozważyć klasę **Mem**. Analizując definicje dwóch konstruktów i dwóch funkcji **pointer()**, można zauważyc, że wprowadzenie domyślnych argumentów nie spowoduje w żadnym z tych przypadków konieczności zmiany definicji funkcji składowej. Tak więc definicja klasy może być łatwo przekształcona do postaci:

```
//: C07:Mem2.h
#ifndef MEM2_H
#define MEM2_H
typedef unsigned char byte;

class Mem {
    byte* mem;
    int size;
    void ensureMinSize(int minSize);
public:
    Mem(int sz = 0);
    ~Mem();
    int msize();
    byte* pointer(int minSize = 0);
};

#endif // MEM2_H III-
```

Zwróć uwagę na to, że wywołanie **ensureMinSize(0)** zawsze będzie dość efektywne.

Mimo że w obu tych przypadkach autor książki kierował się pewnego rodzaju procesem decyzyjnym, uwzględniając kwestie **efektywności**, to należy uważać, by nie wpaść w pułapkę rozważania wszystkiego wyłącznie w tych kategoriach (efektywność jest sama w sobie fascynująca). Najistotniejszym zagadnieniem, związanym z projektowaniem klasy, jest jej interfejs (czyli publiczne składowe, widoczne dla **klienta-programisty**). Jeżeli dzięki niemu klasa będzie łatwa w użyciu i w ponownym wykorzystaniu, osiągniesz sukces — w razie potrzeby zawsze możesz poprawić jej efektywność. Natomiast klasa zaprojektowana źle, ponieważ programista nadmiernie skupił się na kwestii jej efektywności, jest niekiedy prawdziwą zmorą. Należy troszczyć się głównie o to, by interfejs klasy wydawał się logiczny tym, którzy go używają, oraz tym, którzy będą czytać powstały kod. Zwróć uwagę na to, że nie zmienił się sposób użycia klasy **MyString** w pliku **MemTest.cpp** — bez względu na to, czy używała ona domyślnego konstruktora czy też nie, ajej efektywność była mała czy duża.

Podsumowanie

Nie należy używać domyślnego argumentu w charakterze znacznika, od którego zależy warunkowe wykonanie kodu. Natomiast jeżeli możesz, rozbij funkcję na dwie lub większą liczbę przeciążonych funkcji. Domyślny argument powinien być wartością, która byłaby normalnie wpisana na swojej pozycji. Jest to wartość, która występuje częściej niż inne, dzięki czemu **klient-programista** może ją pominąć albo użyć jej tylko w przypadku, gdy chce ją zmienić na inną.

Domyślne argumenty mają na celu ułatwienie wywoływania **funkcji**, zwłaszcza gdy funkcje te pobierają wiele argumentów, posiadających typowe wartości. Nie tylko

prościej jest napisać ich wywołania, ale łatwiej również przeczytać zawierający je kod — przede wszystkim wówczas, gdy twórcą klasy uporządkował argumenty w taki sposób, by najrzadziej modyfikowane znajdowały się na końcu listy.

Szczególnie istotnym sposobem wykorzystania domyślnych argumentów jest przy-padek, gdy po pewnym czasie używania funkcji o określonej **liczbie** argumentów okazuje się, że potrzebne są jej dodatkowe argumenty. Nadając wszystkim nowym argumentom wartości domyślne, można zapewnić, że kod klienta, wykorzystujący poprzedni interfejs, pozostanie nienaruszony.

Ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Utwórz klasę **Text**, która zawiera obiekt klasy **string**, przechowujący tekst znajdujący się w pliku. Zdefiniuj dwaj konstruktory — konstruktor domyślny oraz konstruktor przyjmujący łańcuch, będący nazwąpliku, który ma zostać otwarty. W przypadku użycia drugiego konstruktora otwórz plik i wczytaj jego zawartość do składowej, będącej obiektem klasy **string**. Dodaj funkcję składową **contentns()**, zwracającą przechowywany łańcuch, dzięki czemu będzie go można (na przykład) wydrukować. Wykorzystując klasę **Text**, otwórz w funkcji **main()** plik i wydrukuj jego zawartość.
2. Utwórz klasę **Message** (wiadomość), która zawiera konstruktor pobierający pojedynczy łańcuch, stanowiący wiadomość o określonej domyślnej wartości. Utwórz prywatną składową, będącąłańcuchem typu **string**, a w konstruktorze przypisz tej składowej wartość jego argumentu. Utwórz dwie przeciążone funkcje składowe o nazwach **print() — jedną**, która nie przyjmuje argumentów, drukując po prostu przechowywaną wiadomość, i **drugą**, przyjmującą jako argument łańcuch drukowany wraz z przechowywaną wiadomością. Czy takie podejście, zamiast używanych w konstruktorze wartości domyślnych, ma sens?
3. Dowiedz się, w jaki sposób wygenerować na wyjściu kompilatora kod w asemblerze i przeprowadź eksperymenty, mające na celu określenie sposobu uzupełniania nazw przez kompilator.
4. Utwórz klasę, zawierającą cztery funkcje składowe, pobierające odpowiednio: 0, 1, 2 i 3 argumenty całkowite. Utwórz funkcję **main()**, w której tworzony jest obiekt twojej klasy, a następnie jest wywoływana kolejno każda z funkcji składowych. Teraz zmodyfikuj swoją klasę w taki sposób, by zawierała tylko jedną funkcję składową o wszystkich argumentach domyślnych. Czy zmieni to kod, znajdujący się w funkcji **main()**?
5. Utwórz funkcję, posiadającą dwa argumenty, a następnie wywołaj ją z wnętrza funkcji **main()**. Potem zmień jeden z jej argumentów w „argument-wypełniacz”

(nieposiadający identyfikatora) i sprawdź, czy zmienił się sposób wywołania tej funkcji w funkcji **main()**.

6. Zmodyfikuj pliki **Stash3.h** i **Stash3.cpp** w taki sposób, by w konstruktor klasy używały domyślnych argumentów. Przetestuj konstruktor, tworząc dwie różne wersje obiektu **Stash**.
7. Utwórz nową wersję klasy **Stack** (opisanej w rozdziale 6.), **zawierającą**, jak poprzednio, domyślny konstruktor, a także drugi konstruktor, przyjmujący jako argumenty tablicę wskaźników do obiektów oraz wielkość tej tablicy. Konstruktor powinien przejść przez tę tablicę, umieszczając na stosie każdy z zawartych w niej wskaźników. Przetestuj klasę, używając tablicę łańcuchów.
8. Zmodyfikuj klasę **SuperVar** w taki sposób, aby kod, wykorzystujący zmienną **vartype**, znajdował się w obrębie dyrektywy **#ifdef**, jako opisano w części poświęconej deklaracji **enum**. Przekształć zmienną **vartype** w zwykłe, publicznie dostępne wyliczenie (bez egzemplarza zmiennej) i zmodyfikuj funkcję **print()** w taki sposób, by wymagała argumentu typu **vartype**, określającego, jakie działania ma wykonać.
9. Dokonaj implementacji klasy, której definicja znajduje się w pliku **Mem2.h**, upewniając się, że działa nadal z programem **MemTest.cpp**.
10. Wykorzystaj klasę **Mem** do zaimplementowania klasy **Stash**. Zwróć uwagę na to, że ponieważ implementacja ma charakter prywatny, dzięki czemu jest ukryta przed klientem-programistą, program testowy klasy nie wymaga żadnych modyfikacji.
11. Do klasy **Mem** dodaj funkcję **moved()**, która pobiera rezultat wywołania funkcji **pointer()** i zwraca wartość **bool**, **informującą**, czy wskaźnik został przesunięty (z powodu powtórnego przydziału pamięci). Napisz funkcję **main()**, testującą nową funkcję składową. Czy używanie funkcji w rodzaju **moved()** ma sens, czy też lepiej wywoływać funkcję **pointer()** ilekroć potrzebny jest dostęp do pamięci przechowywanej w obiekcie klasy **Mem**?

Rozdział 8.

Stałe

Pojęcie **stałej** (wyrażone za pomocą słowa kluczowego **const**) zostało utworzone po to, by umożliwić programistom wytyczenie granicy pomiędzy tym, co się zmienia, i tym, co niezmienne. Zapewnia ono bezpieczeństwo i kontrolę w projektach programistycznych języka C++.

Pojęcie to, od momentu powstania, miało wiele różnych zastosowań. W tym czasie wchodziło ono stopniowo także do języka C, ale jego znaczenie uległo zmianie. Wszystko to może wydawać się na pierwszy rzut oka nieco skomplikowane, ale na podstawie lektury rozdziału dowiesz się, kiedy, dlaczego i w jaki sposób używa się słowa kluczowego **const**. Pod koniec rozdziału zamieszczono dyskusję na temat słowa kluczowego **volatile**, blisko spokrewnionego ze słowem **const** (oba dotyczą zmian), posiadającym identyczną składnię.

Wydaje się, że najważniejszym powodem wprowadzenia słowa kluczowego **const** było zaprzestanie używania dyrektywy preprocesora **#define** do zastępowania wartości. Od tej pory słowo to jest wykorzystywane w stosunku do wskaźników, argumentów funkcji, zwracanych wartości, obiektów klas i funkcji składowych. Wszystkie one mają nieco różne, ale pojawiowo zbliżone znaczenia i zostaną omówione w kolejnych podrozdziałach, zawartych w rozdziale.

Podstawianie wartości

W celu tworzenia makroinstrukcji i podstawiania wartości podczas programowania w języku C często używa się preprocesora. Ponieważ preprocesor dokonuje tylko prostego zastępowania tekstu, nie posiadając ani wiedzy na temat typów, ani możliwości ich kontroli, podstawianie wartości za pomocą preprocesora powoduje trudno uchwytnie problemy, których można uniknąć, używając w języku C++ wartości stałych.

Typowy sposób używania preprocesora do podstawiania wartości jest następujący w języku C:

```
#define BUFSIZE 100
```

BUFSIZE jest nazwą istniejącą wyłącznie w trakcie pracy preprocesora, a zatem nie zajmuje ona pamięci i może zostać umieszczona w pliku nagłówkowym, udostępniając tę samą wartość wszystkim wykorzystującym ją jednostkom translacji. Dla pielęgnacji kodu bardzo ważne jest, by używać podstawiania wartościami zamiast tak zwanych „magicznych liczb”. Jeżeli w swoim programie używasz magicznych liczb, to nie tylko jego czytelnik nie wie, skąd się one wzięły i co oznaczają, ale również w razie konieczności zmiany którejś z nich jesteś zdany na żmudną edycję, nie mając żadnej gwarancji, że nie została opuszczona żadna wartość (albo przypadkowo nie zmieniłeś niewłaściwej).

W większości przypadków **BUFSIZE** będzie zachowywała się jak zwykła zmienna, ale nie w każdym. W dodatku nie posiada ona żadnej informacji, dotyczącej typu. Może to doprowadzić do powstania ukrytych, bardzo trudnych do wykrycia błędów. Do eliminacji tego typu problemów język C++ wykorzystuje słowo kluczowe **const**, przesuwające proces zastępowania wartości do kompilatora. Dzięki temu można napisać:

```
const int bufsize = 100;
```

Stałej **bufsize** można używać w każdym miejscu, w którym kompilator musi znać wartość w czasie komplikacji. Kompilator może stosować wartości **bufsize** do dokonania *składania stałych* (ang. *constant folding*) co oznacza, że uproszczy on złożone wyrażenia, przeprowadzając niezbędne obliczenia podczas komplikacji. Jest to szczególnie ważne w przypadku definicji tablic:

```
char buf[bufsize];
```

Słowa kluczowego **const** można użyć w stosunku do wszystkich wbudowanych typów (**char**, **int**, **float** i **double**) oraz ich wariantów (podobnie jak i w stosunku do obiektów klas, co zostanie zaprezentowane w dalszej części rozdziału). Z uwagi na trudno uchwytnie błędy, które może spowodować **użycie** preprocesora, zamiast podstawić za pomocą dyrektywy **#define** należy zawsze posługiwać się słowem kluczowym **const**.

State w plikach nagłówkowych

Aby zamiast dyrektywy **#define** używać modyfikatora **const**, musi istnieć możliwość umieszczenia definicji stałej w pliku nagłówkowym, tak jak w przypadku **#define**. Dzięki temu można umieścić definicję stałej w jednym miejscu, a następnie dostarczyć ją poszczególnym jednostkom translacji, dołączając do nich odpowiedni plik nagłówkowy. Stałe są w języku C++ domyślnie *łączone wewnętrznie* — to znaczy są widoczne wyłącznie w pliku, w którym zostały zdefiniowane, a podczas łączenia nie są dostępne dla innych jednostek translacji. W czasie definiowania stałej trzeba zawsze nadać jej wartość, z wyjątkiem przypadku, gdy dokonuje się jawniej deklaracji stałej za pomocą słowa kluczowego **extern**:

```
extern const int bufsize;
```

Zwykle kompilator C++ unika rezerwacji pamięci, zawierającej wartość stałej, **przechowując jej definicję w tablicy symboli**. Jednak w przypadku użycia w deklaracji **const** słowa kluczowego **extern**, wymusza się przydzielenie stałej pamięci (dzieje się

tak również w niektórych innych przypadkach, takich jak pobieranie adresu stałej). Pamięć musi zostać przydzielona, ponieważ słowo kluczowe **extern** nakazuje: „użyj łączenia **zewnętrznego**”. Oznacza to, że wiele jednostek translacji musi być w stanie odwołać się do tej wartości, co wymaga przydzielenia jej pamięci.

Zazwyczaj — gdy definicja nie zawiera słowa **extern** — pamięć nie jest przydzielana. Kiedy używane jest słowo kluczowe **const**, wartość jest po prostu wstawiana do programu podczas komplikacji.

Postulat, by nigdy nie przydzielać stałym pamięci, zawodzi również w przypadku złożonych struktur. Gdy kompilator musi przydzieleć stałej pamięć, nie dokonuje składania stałych (ponieważ nie potrafi określić wartości tej stałej — gdyby ją znał, nie musiałby przydzieleć jej pamięci).

Z uwagi na to, że kompilator nie zawsze zdoła uniknąć przydzielenia stałej pamięci, jej definicja *musi* być domyślnie łączona wewnętrznie, to znaczy wyłącznie w *obrębie* danej jednostki translacji. W przeciwnym razie spowodowałoby to zgłoszenie przez program łączący błędu w przypadku złożonych stałych, ponieważ w wielu plikach **cpp** zostałaaby im przydzielona pamięć. Program łączący „**zaprotestowałby**” z powodu tej samej definicji zawartej w wielu plikach wynikowych. Ponieważ stałe podlegają domyślnie łączeniu wewnętrzemu, program łączący nie próbuje kojarzyć ich definicji pomiędzy różnymi jednostkami translacji, co pozwala na uniknięcie kolizji. W przypadku typów wbudowanych, występujących w większości wyrażeń zawierających stałe, kompilator zawsze może przeprowadzić składanie stałych.

Bezpieczeństwo stałych

Zastosowanie stałych nie ogranicza się do zastępowania dyrektyw **#define** w wyrażeniach stałych. Gdy zmienna jest inicjalizowana wartością, wyznaczaną w trakcie pracy programu, i wiadomo, że wartość tej zmiennej nie zmieni się w czasie jej życia, to dobrym programistycznym zwyczajem jest uczynienie jej stałą. Dzięki temu kompilator zgłosi komunikat o błędzie w razie przypadkowej próby zmiany jej wartości. Poniżej zamieszczono odpowiedni przykład:

```
//: C08:Safecons.cpp
// Wykorzystywanie stałych dla bezpieczeństwa
#include <iostream>
using namespace std;

const int i = 100; // Typowa stała
const int j = i + 10; // Wartość wyrażenia stałego
long address = (long)&j; // Wymuszenie przydziału pamięci
char buf[j + 10]; // Nadal jest to wyrażenie stałe

int main() {
    cout << "wpisz znak i nacisnij Enter:";
    const char c = cin.get(); // Tej wartości nie będzie można zmienić
    const char c2 = c + 'a';
    cout << c2;
    // ...
} //
```

A zatem i jest **stałą**, określona w czasie kompilacji, lecz wartość stałej j jest wyliczana na podstawie wartości i. Ponieważ jednak i jest **stałą**, wartość stałej j jest obliczana również na podstawie wyrażenia stałego, w związku z tym jest ona również stałą o wartości wyznaczonej podczas kompilacji. Już w następnym wierszu wymagany jest adres stałej j, co wymusza na kompilatorze przydzielenie jej pamięci. Nie przeszkadza to jednak w użyciu stałej j do określenia wielkości tablicy **buf** — kompilator wie bowiem, że j jest stałą, której wartość jest poprawna, nawet jeśli w jakimś miejscu programu została przydzielona pamięć przechowująca tę wartość.

W funkcji **main()**, pod nazwą c, zdefiniowano inny rodzaj stałej, ponieważ jej wartość nie może być znana w trakcie kompilacji. Oznacza to, że przydzielana jest jej pamięć, a kompilator nie próbuje zapisywać niczego w swojej tablicy symboli (zachowuje się tak samo, jak w języku C). Inicjalizacja stałej musi nadal występować w miejscu jej definicji, ale od momentu **inicjalizacji** wartość tej stałej nie może już być zmieniana. Na podstawie wartości stałej c wyznaczana jest wartość stałej **c2**, a w przypadku stałych obowiązują również reguły dotyczące zasięgów, co stanowi jeszcze jedną przewagę stałych nad **dyrektywą #define**.

Z praktyki wynika, że jeżeli wartość nie powinna być zmieniana, to należy uczynić ją stałą. Stanowi to nie tylko ochronę przed jej przypadkowymi zmianami, ale pozwala również kompilatorowi na wygenerowanie bardziej efektywnego kodu — dzięki uniknięciu przydzielania pamięci i jej późniejszego odczytywania.

Agregaty

Możliwe jest użycie modyfikatora **const** w odniesieniu do agregatów, ale jest właściwie pewne, że kompilator nie zadziała na tyle przemyślnie, by umieścić agregat w tablicy symboli, w związku z czym przydzieli mu pamięć. W takich przypadkach pojęcie stałej oznacza: „obszar pamięci, który nie może być zmieniany”. Jednak jego wartości nie wolno używać w czasie kompilacji, ponieważ nie można wymagać od kompilatora, aby znał zawartość pamięci w czasie tego procesu. W zamieszczonym poniżej programie zaznaczono instrukcje, które nie są dozwolone:

```
//: C08:Constag.cpp
// Stałe a agregaty
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Niedozwolone
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Niedozwolone
int main() {} //:~
```

Aby ulokować zdefiniowaną tablicę, kompilator musi być w stanie wygenerować kod, przesuwający wskaźnik stosu. W obu niedozwolonych definicjach tablic (przedstawionych powyżej) kompilator zgłasza sprzeciw, ponieważ nie potrafi znaleźć w nich stałego wyrażenia.

Różnice w stosunku do języka C

Stałe zostały wprowadzone do wczesnych wersji języka C++, gdy specyfikacja standaryzacyjna języka C wciąż jeszcze nie była ukończona. Mimo że komitet standaryzacyjny języka C zdecydował później o wprowadzeniu do tegoż języka stałych, to nadał im znaczenie „zwykłych zmennych, których wartości nie mogą być zmieniane”. W języku C stałe zawsze zajmują pamięć, a ich nazwy są globalne. Kompilator języka C nie może traktować stałej jako stałej dostępnej w trakcie komplikacji. Po zapisaniu w języku C:

```
const int bufsize = 100;  
char buf[bufsize];
```

otrzymamy komunikat o błędzie, mimo że taki zapis wydaje się racjonalny. Ponieważ stała **bufsize** znajduje się gdzieś w pamięci, kompilator C nie może znać jej wartości w czasie komplikacji. Opcjonalnie, można napisać w języku C:

```
const int bufsize;
```

ale nie jest to dozwolone w języku C++. Kompilator języka C przyjmuje taki zapis jako deklarację obszaru pamięci, przydzielonego w jakimś innym miejscu programu. To dopuszczalne, ponieważ w języku C stałe są domyślnie łączone zewnętrznie. W języku C++ domyślnie w stosunku do stałych jest stosowane wewnętrzne łączenie; chcąc zatem uzyskać taki sam efekt w C++, należy jawnie zmienić sposób wiązania na zewnętrzny, używając do tego celu słowa kluczowego **extern**:

```
extern const int bufsize; // Tylko deklaracja
```

Zapis taki jest poprawny również w języku C.

W języku C++ stałe nie zawsze zajmują pamięć, w przeciwieństwie do języka C. O tym, czy pamięć jest w języku C++ rezerwowana dla stałej, czy też nie, decyduje sposób używania tej stałej. Na ogół jeżeli jest ona używana w celu zastąpienia nazwy wartością (w taki sposób, jak byłaby używana dyrektywa **#define**), to nie ma konieczności przydzielania stałej pamięci. Jeżeli pamięć nie jest przydzielana (zależy to od złożoności typu danych i przemyślności kompilatora), wartości mogą zostać umieszczone — dla uzyskania większej efektywności — bezpośrednio w kodzie, po dokonaniu kontroli typów, a nie przed nią, jak w przypadku użycia dyrektywy **#define**. Jeżeli jednak pobierany jest adres stałej (nawet niejawnie, podczas wywołania funkcji pobierającej argument przekazywany przez referencję) albo stała ta zostanie zdefiniowana z użyciem słowa kluczowego **extern**, wówczas przydzielana jest pamięć.

W języku C++ stałe znajdujące się poza ciałami wszystkich funkcji posiadają zasięg ograniczony do pliku (nie są one poza tym plikiem widoczne). Oznacza to, że podlegają one domyślnie łączeniu wewnętrzemu. Jest to zupełnie inaczej niż w przypadku wszystkich pozostałych identyfikatorów w języku C++ (i stałych w C), które są domyślnie łączone zewnętrznie. Tak więc, jeżeli zadeklaruje się w dwóch różnych plikach stałe o tych samych nazwach i nie zostaną wykorzystane ich adresy ani nie zadeklaruje się ich przy użyciu słowa kluczowego **extern**, to idealny kompilator języka C++ nie przydzieli tym stałym pamięci, umieszczając przypisane im wartości bezpośrednio w kodzie programu. Ponieważ stałe mają domyślnie zasięg pliku, można umieścić je w pliku nagłówkowym programu w języku C++, nie powodując konfliktów w czasie łączenia.

Z uwagi na to, że stałe są w języku C++ domyślnie **łączone** wewnętrznie, nie można zadeklarować stałej w jednym pliku, a następnie odwoływać się do niej jako do obiektu zewnętrznego w drugim pliku. Dla zapewnienia stałej łączenia zewnętrznego, umożliwiającego odwołanie się do niej w innym pliku, trzeba jawnie ją zdefiniować przy użyciu słowa kluczowego **extern**, jak w poniższym przykładzie:

```
extern const int x = 1;
```

Zwróci uwagę, że przypisanie stałej wartości i zadeklarowanie jej jako zewnętrznej wymusza utworzenie dla niej pamięci (chociaż kompilator nadal ma w takim przypadku możliwość dokonania składania **stałych**). **Inicjalizacja** określa, że jest to definicja, a nie deklaracja. Deklaracja:

```
extern const int x;
```

oznacza bowiem w języku C++, że definicja stałej znajduje się w jakimś innym miejscu (w języku C niekoniecznie musi być to prawda). Wiadomo zatem, dlaczego język C++ wymaga, by definicja stałej posiadała inicjator — pozwala on na odróżnienie deklaracji od definicji (w języku C jest to zawsze definicja, więc inicjator nie jest potrzebny). W przypadku deklaracji **extern const** kompilator nie może dokonać składania stałych, ponieważ nie zna on jej wartości.

Ujęcie stałych, prezentowane w języku C, nie jest zbyt użyteczne. Jeżeli więc zamierzasz używać wewnętrz wyrażenia stałego (takiego, które musi być obliczone podczas komplikacji) nazwy, posiadającej przypisaną wartość, to język C niemal *zmusza cię* do używania dyrektywy **#define** preprocessora.

Wskaźniki

Wskaźniki również mogą być stałymi. Mając do czynienia ze stałymi, będącymi wskaźnikami, kompilator nadal usiłuje uniknąć przydzielenia im pamięci i dokonuje składania stałych, ale właściwości te wydają się mieć w tym przypadku mniejsze znaczenie. Ważniejsze jest, że kompilator poinformuje cię o próbie zmiany stałej, będącej wskaźnikiem, co znacznie zwiększa poziom bezpieczeństwa programu.

Podczas stosowania modyfikatora **const** ze wskaźnikami istnieją dwie możliwości — może on być zastosowany do tego, co wskazuje wskaźnik, albo do adresu przechowywanego w samym wskaźniku. W obu tych przypadkach składnia wydaje się na pierwszy rzut oka dość niejasna, ale w praktyce okaże się ona wygodna.

Wskaźniki do stałych

Podobnie jak w każdej skomplikowanej definicji, również w przypadku definicji wskaźników sztuka polega na tym, by odczytywać ją zaczynając od identyfikatora, a następnie przesuwać się stopniowo na zewnątrz. Modyfikator **const** jest związany z tym, co znajduje się w definicji „najbliżej niego”. Dlatego **też**, aby zapobiec wszelkim zmianom wskazywanego elementu, należy zapisać **następującą definicję**:

```
const int* u;
```

Rozpoczynając od identyfikatora, odczytujemy: „u jest wskaźnikiem, wskazującym na element typu **const int** (stałą całkowitą)”. W tym przypadku nie jest konieczna żadna inicjalizacja, ponieważ według definicji wskaźnik u może wskazywać dowolny element (wskaźnik nie jest stały), ale nie może on być zmieniany.

W tym miejscu zaczynają się trudności. Może się wydawać, że, aby uczynić wskaźnik samym w sobie niezmiennym, to znaczy, uniemożliwić zmianę adresu, pamiętanego wewnątrz u, należy po prostu przenieść słowo **const** na drugą stronę słowa kluczowego **int**, jak poniżej:

```
int const* v;
```

Odczytywanie takiego zapisu jako: „v jest stałym wskaźnikiem do liczby całkowitej” nie jest zupełnie pozbawione logiki. Jednakże *w rzeczywistości* oznacza on: „v jest zwykłym wskaźnikiem do liczby całkowitej, która jest akurat stałą”. Wynika z tego, że modyfikator **const** został ponownie związany ze słowem kluczowym **int** i w rezultacie uzyskaliśmy identyczną definicję jak poprzednio. Fakt, że obie te definicje są takie same, jest z pewnością irytujący — aby zapobiec podobnym odczuciom osoby czytającej program, lepiej jest poprzestać na pierwszej wersji definicji.

State wskaźniki

Aby wskaźnik sam w sobie uczynić stałym, należy umieścić modyfikator **const** po prawej stronie znaku „*”, jak w poniższym przykładzie:

```
int d = 1;
int* const w = &d;
```

Obecnie odczytuje się ten zapis następująco: „**w** jest wskaźnikiem będącym stałą, który wskazuje liczbę całkowitą”. Ponieważ sam wskaźnik jest obecnie stałą, to kompilator wymaga, by została mu nadana wartość inicjującą, która pozostanie niezmieniona przez cały czas jego życia. Można jednak zmodyfikować wskazywaną przez niego wartość, zapisując:

```
*w = 2;
```

Można również utworzyć stały wskaźnik, wskazujący obiekt będący stałą, używając jednej ze znajdujących się poniżej, dopuszczalnych postaci:

```
int d = 1;
const int* const x = &d; // (1)
int const* const x2 = &d; // (2)
```

Obecnie żaden z tych wskaźników ani jakikolwiek ze wskazywanych przez nie obiektów nie może ulec zmianie.

Według niektórych opinii druga z wymienionych powyżej postaci definicji jest bardziej spójna, ponieważ słowo **const** jest umieszczone zawsze po prawej stronie tego, co modyfikuje. Musisz sam zdecydować o tym, co jest bardziej przejrzyste z punktu widzenia używanego przez ciebie stylu programowania.

A oto powyższe wiersze, umieszczone w możliwym do skompilowania pliku:

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;
int d = 1;
int* const w = &d;
const int* const x = &d; // (1)
int const* const x2 = &d; // (2)
int main() {} //:-
```

Formatowanie

W niniejszej książce został położony szczególny nacisk na to, by w każdym wierszu kodu znajdowała się tylkожedna definicja wskaźnika, a wskaźniki były inicjalizowane w miejscu definicji, ilekroć to możliwe. Dzięki temu możliwe jest zastosowanie stylu formatowania, polegającego na „doklejeniu” znaku „*” do typu danych:

`int*u=&i;`

jak gdyby `int*` było samo w sobie odrębnym typem danych. W rezultacie kod wydaje się łatwiejszy do zrozumienia, ale nie jest to, niestety, sposób zgodny z rzeczywistością. W istocie symbol „*” jest związany z identyfikatorem, a nie z typem. Może być umieszczony w dowolnym miejscu, pomiędzy typem i identyfikatorem. Można więc zapisać definicję:

`int *u = &i, v = 0;`

tworzącą, jak poprzednio, wskaźnik `int*` u oraz zmienną `int v`, niebędącą wskaźnikiem. Ponieważ jest to mylące dla osoby czytającej kod, lepiej używać konwencji stosowanej w książce.

Przypisanie a kontrola typów

Język C++ jest bardzo skrupulatny pod względem kontroli typów; obejmuje to również operacje przypisania wskaźników. Można przypisać wskaźnikowi stałej adres obiektu niebędącego **stałą**, ponieważ przypisanie takie wiąże się po prostu z obietnicą niezmieniania czegoś, co i tak może być modyfikowane. Jednakże nie wolno przypisać adresu obiektu będącego stałą do wskaźnika do obiektu niebędącego stałą, ponieważ oznaczałoby to możliwość modyfikacji obiektu za pośrednictwem wskaźnika. Oczywiście, zawsze można wymusić takie przypisania, używając rzutowań, ale jest to przykład złego stylu programowania. Narusza się bowiem w ten sposób niezmienność obiektów będących stałymi, a także bezpieczeństwo zapewnione przez słowo kluczowe **const**. Na przykład:

```
//: C08:PointerAssignment.cpp
int d = 1;
const int e = 2;
int* u = &d; // W porządku - d nie jest stałą
//! int* v = &e; // Niedozwolone - e jest stałą
int* w = (int*)&e; // Dozwolone, ale jest to zły styl
int main() {} //:-
```

Mimo że język C++ pomaga w zapobieganiu błędom, to nie uchroni cię przed konsekwencjami w przypadku, gdy chcesz naruszyć mechanizmy bezpieczeństwa.

Literały napisowe

Miejscem, w którym niezmienna nie jest bezwzględnie egzekwowana, są literały napisowe (literały będące tablicami znakowymi). Można więc zapisać:

```
char* cp = "czesc";
```

i kompilator przyjmie to bez protestu. Z technicznego punktu widzenia jest to błąd, ponieważ literały napisowe (w tym przypadku „część”) są tworzone przez kompilator jako stałe tablice znakowe, a wartością zwracaną przez tablicę znaków ujętych w cudzysłów jest adres jej początku w pamięci. Modyfikacja jakiegokolwiek znaku, znajdującego się w takiej tablicy, jest błędem wykonania programu, chociaż nie wszystkie kompilatory egzekwują go w poprawny sposób.

Tak więc literały napisowe są w rzeczywistości stałymi tablicami znakowymi. Oczywiście, kompilator pozwala traktować je tak, jakby nie były stałymi, z uwagi na to, że wykorzystuje to znaczna ilość istniejącego kodu napisanego w języku C. Jeżeli jednak spróbujesz zmienić wartości zawarte w literale napisowym, to wynik takiej operacji jest niezdefiniowany, chociaż będzie ona prawdopodobnie działać w wielu komputerach.

Jeżeli jednak zamierzasz modyfikować łańcuch, to umieść go w tablicy:

```
char cp[] = "czesc";
```

Ponieważ jednak kompilatory często nie wymuszają takiego rozróżnienia, nie będą nalegać na używanie tej drugiej postaci definicji, w związku z czym cała kwestia staje się dość trudno uchwytna.

Argumenty funkcji i zwracane wartości

Użycie modyfikatora **const** do specyfikacji argumentów funkcji i zwracanych przez nie wartości jest jeszcze jedną sytuacją, w której pojęcie stałych może wydawać się niezrozumiałe. W przypadku przekazywania obiektów *przez wartość* określenie ich jako stałych nie ma dla klienta funkcji żadnego znaczenia (oznacza, że przekazywanego argumentu nie wolno zmieniać wewnątrz funkcji). Jeżeli funkcja zwraca przez wartość, jako **stałą**, obiekt typu zdefiniowanego przez użytkownika, oznacza to, że zwracana wartość nie może być modyfikowana. Jeżeli przekazywany lub zwracany jest *adres*, to modyfikator **const** stanowi przyrzeczenie, że wartość znajdująca się pod tym adresem nie będzie zmieniana.

Przekazywanie stałej przez wartość

Przekazując funkcji argumenty przez wartość, można określić, że są one stałymi, jak w poniższym przypadku:

```
void f1(const int i) {
    i++; // Niedozwolone - błąd podczas komilacji
}
```

Cóż to jednak oznacza? Złożono obietnicę, że oryginalna wartość zmiennej nie zostanie zmodyfikowana przez funkcję **f1()**. Ponieważ jednak argument jest przekazywany przez wartość, natychmiast tworzona jest kopią jego oryginalnej wartości, więc obietnica złożona klientowi jest i tak domyślnie dotrzymywana.

Natomiast modyfikator **const** wewnątrz funkcji oznacza, że jej argument nie może być zmieniany. Jest to więc w rzeczywistości narzędzie służące twórcy funkcji, a nie komuś, który ją wywołuje.

Aby nie spowodować dezorientacji osoby wywołującej funkcję, można uczynić argument stałą *wewnątrz* funkcji, a nie na liście jej argumentów. Można uczynić to za pomocą wskaźnika, ale bardziej elegancki zapis uzyskuje się dzięki *referencji* (temat referencji zostanie gruntownie opisany rozdziale 11.). Krótko mówiąc, referencja przypomina stały wskaźnik, na którym automatycznie dokonuje się wyłuskania, dzięki czemu w efekcie staje się on synonimem obiektu. Aby utworzyć referencję, należy użyć w definicji symbolu &. Tak więc niewywołującą dezorientacji definicja funkcji jest następująca:

```
void f2(int& ic) {
    const int& i = ic;
    i++; // Niedozwolone - błąd podczas komilacji
}
```

W czasie komilacji zostanie ponownie zgłoszony **błąd**, ale tym razem będzie on wynikał z niezmienności lokalnego obiektu, niebędącego elementem sygnatury funkcji — ma on znaczenie jedynie dla jej implementacji i dlatego właśnie został ukryty przediej klientem.

Zwracanie stałej przez wartość

Podobnie dzieje się w przypadku wartości zwracanej przez funkcję. Jeżeli napiszemy, że wartość zwracana przez funkcję jest stała:

```
const int g();
```

stanowi to obietnicę, że oryginalna zmienna (znajdująca się wewnątrz funkcji) nie zostanie zmieniona. Ponieważ jednak jest ona ponownie zwracana przez wartość, wykonywana jest jej kopia, dzięki czemu początkowa wartość nie może być nigdy zmieniona za pośrednictwem zwracanej wartości.

Na pierwszy rzut oka może się wydawać, że określenie zwracanej przez funkcję wartości jako stałej jest pozbawione sensu. Pozornego braku efektu zwracania stałej przez wartość stałej ilustruje poniższy przykład:

```
//: C08:Constval.cpp
// Zwracanie stałej przez wartość nie ma
// znaczenia w przypadku typów wbudowanych
```

```

int f3() { return 1; }
const int f4() { return 1; }

int main() {
    const int j = f3(); // Działa znakomicie
    int k = f4(); // Ale to również działa znakomicie!
} //:-~

```

W przypadku typów wbudowanych nie ma żadnego znaczenia, czy zwracana wartość jest stałą. Dlatego też **należy** unikać wprawiania w zakłopotanie klienta programistę, pomijając modyfikator **const** w przypadku, gdy funkcja zwraca wartość wbudowanego typu.

Zwracanie przez wartość stałych jest ważne w przypadku typów **zdefiniowanych przez użytkownika**. Jeżeli funkcja zwraca przez wartość obiekt jako stałą, to zwróci na wartość nie może być *l-wartością* (to znaczy nie można do niej dokonać przypisania ani zmodyfikować jej w jakiś inny sposób). Na przykład:

```

//: C08:ConstReturnValues.cpp
// Stała zwracana przez wartość
// Wynik nie może być użyty jako l-wartość

class X {
    int i;
public:
    X(int ii = 0);
    void modify();
};

X::X(int ii) { i = ii; }

void X::modify() { i++; }

X f5() {
    return X();
}

const X f6() {
    return X();
}

// Przekazanie przez referencje obiektu
// nie będącego stałą:
void f7(X& x) {
    x.modify();
}

int main() {
    f5() = X(1); // W porządku - zwracana wartość nie jest stałą
    f5().modify(); // W porządku
    // Instrukcje wywołujące błędy podczas kompilacji:
    //! f7(f5());
    //! f6() = X(1);
    //! f6().modify();
    //! f7(f6());
} //:-~

```

Funkcja **f5()** zwraca obiekt X niebędący stałą, natomiast funkcja **f6()** — obiekt X będący stałą. W charakterze *l-wartości* może być użyta jedynie taka zwracana wartość, która nie jest stałą. Tak więc istotne jest stosowanie modyfikatora **const** w przypadku, gdy obiekt jest zwracany przez wartość i chcemy zapobiec używaniu go jako *l-wartości*.

Modyfikator **const** nie ma znaczenia w przypadku typów wbudowanych dlatego, że kompilator nie dopuszcza do używania ich jako *l-wartości* (ponieważ zawsze są one wartościami, a nie zmiennymi). Kwestia ta ma znaczenie dopiero w przypadku, gdy zwracane są przez wartość obiekty typów zdefiniowanych przez użytkownika.

Funkcja **f7()** pobiera argument niebędący stałą poprzez *referencję* (dodatkowy sposób obsługi adresów w języku C++, opisany w rozdziale 11.). Jest to niemal równoważne pobraniu wskaźnika do obiektu niebędącego stałą — różnica polega jedynie na składni. Powodem, dla którego instrukcja ta nie kompliluje się w języku C++, jest tworzenie obiektu tymczasowego.

Obiekty tymczasowe

Czasami, w czasie obliczania wyrażenia, kompilator jest zmuszony do utworzenia *obiektów tymczasowych*. Są one podobne do innych — zajmują pamięć, a także muszą być utworzone oraz zniszczone. Różnica polega na tym, że nie są one widoczne — kompilator decyduje o tym, kiedy są one potrzebne, i zajmuje się szczegółami dotyczącymi ich istnienia. Jest jednak pewna istotna kwestia, związana z obiektami tymczasowymi — są one tworzone automatycznie jako stałe. Z uwagi na to, że zazwyczaj nie istnieje możliwość wykonywania operacji na obiektach tymczasowych, nakazanie zrobienia czegoś, co zmieniłoby obiekt tymczasowy, jest z pewnością pomocą, ponieważ nie ma sposobu na to, by później wykorzystać tę informację. Dzięki temu, że wszystkie obiekty tymczasowe stają się automatycznie stałymi, kompilator ma możliwość sygnalizowania takich pomyłek.

W powyższym przykładzie funkcja **f5()** zwraca obiekt typu X niebędący stałą. Jednak **w wyrażeniu:**

```
f7(f5());
```

kompilator musi utworzyć tymczasowy obiekt, przechowujący wartość, zwracaną przez funkcję **f5()**, by można ją było przekazać funkcji **f7()**. Problemu nie istniałby, gdyby funkcja **f7()** pobierała argument przez wartość — w takim przypadku obiekt tymczasowy byłby kopowany w funkcji **f7()** i jego dalsze losy nie miałyby znaczenia. Jednakże funkcja **f7()** pobiera swój argument *przez referencję*, co w tym przypadku oznacza, że pobiera ona adres tymczasowego obiektu typu X. Ponieważ argument pobierany przez funkcję **f7()** nie jest referencją do stałej, funkcja ta ma możliwość zmiany obiektu tymczasowego. Jednak kompilator wie, że obiekt tymczasowy przestanie istnieć zaraz po zakończeniu obliczania wyrażenia, a zatem wszelkie dokonane w nim zmiany zostaną utracone. Dzięki temu, że wszystkie obiekty tymczasowe są domyślnie obiektami stałymi, sytuacje tego typu powodują błędy na etapie komplikacji, co umożliwia uniknięcie pomyłek trudnych do wykrycia.

Warto jednak zwrócić jeszcze uwagę na wyrażenia, które są dozwolone:

```
f5() - X(1);  
f5().modify();
```

Mimo że są one poprawne z punktu widzenia kompilatora, to w rzeczywistości budzą one wątpliwości. Funkcja **f5()** zwraca obiekt typu X, natomiast kompilator, aby zrealizować powyższe wyrażenia, musi utworzyć obiekt tymczasowy, przechowujący zwracaną wartość. Tak więc w obu wyrażeniach modyfikowane są obiekty tymczasowe, które są usuwane natychmiast po zakończeniu związań z tymi wyrażeniami operacji. W rezultacie dokonane zmiany są tracone, a więc powyższy kod jest prawdopodobnie błędny — ale kompilator nie zgłosi w tym przypadku żadnych uwag. Wyrażenia takie, jak widoczne powyżej, są dostatecznie proste, aby można było samodzielnie odkryć związań z nimi problemy. Jednak w bardziej skomplikowanych przypadkach można przeoczyć błędy.

Sposób, w jaki chroniona jest niezmiennałość obiektów klas, został opisany w dalszej części rozdziału.

Przekazywanie i zwracanie adresów

W przypadku przekazywania lub zwracania adresów (zarówno za pomocą wskaźników, jak i referencji) możliwe jest wykorzystanie ich przez klienta-programistę do zmodyfikowania wskazywanej przez nie wartości. Aby temu zapobiec, należy uczyć się wskaźnik lub referencję stałymi, co może uchronić cię od wielu nieszczęść. Właściwie ilekroć przekazujesz funkcji adres, powinieneś określić, że zawiera on stałą — o ile jest to możliwe. W przeciwnym przypadku wykluczysz możliwość używania takiej funkcji w stosunku do stałych.

Decyzja, czy funkcja powinna zwracać wskaźnik, czy też referencję do stałej, zależy od tego, na jakie działania w stosunku do niej zamierzasz pozwolić klientowi-programistie. Poniższy przykład prezentuje użycie wskaźników do stałych — zarówno argumentów funkcji, jak i zwracanych przez nie wartości:

```
//: C08:ConstPointer.cpp  
// Wskaźniki do stałych jako argumenty funkcji  
// i zwracane przez nie wartości  
  
void t(int*) {}  
  
void u(const int* cip) {  
    /* *cip = 2; // Niedozwolone - modyfikacja wartości  
    int i = *cip; // W porządku - kopianie wartości  
    */ int* ip2 = cip; // Niedozwolone - nie jest to stałą  
}  
  
const char* v() {  
    // Zwraca adres statycznej tablicy znaków:  
    return "wynik funkcji v()";  
}  
  
const int* const wC() {  
    static int i;  
    return &i;  
}
```

```

int main() {
    int x = 0;
    int* ip = &x;
    const int* cip = &x;
    t(ip); // W porządku
    //! t(cip); // Źle
    u(ip); // W porządku
    u(cip); // Również w porządku
    //! char* cp = v(); // Źle
    const char* ccp = v(); // W porządku
    //! int* ip2=w(); // Źle
    const int* const ccip = w(); // W porządku
    const int* cip2 = w(); // W porządku
    //! *w() = 1; // Źle
} //:-

```

Funkcja **t()** pobiera jako argument zwykły wskaźnik (niewskazujący stałej), natomiast funkcja **u()** pobiera wskaźnik do stałej. Jak można dostrzec wewnątrz funkcji **u()**, próba modyfikacji wskazywanej przez argument wartości nie jest dozwolona, ale można, oczywiście, przepisać wskazywaną przez niego wartość do zwykłej zmiennej. Kompilator zapobiega również utworzeniu wskaźnika niebędącego wskaźnikiem do stałej, który wykorzystuje adres przechowywany wewnątrz wskaźnika do stałej.

Funkcje **v()** oraz **w()** umożliwiają sprawdzenie znaczenia zwracanych wartości. Funkcja **v()** zwraca wartość typu **const char***, utworzoną na podstawie literała napisowego. Instrukcja ta zwraca w rzeczywistości adres literała napisowego po tym, jak zostanie on utworzony przez kompilator i zapisany w obszarze danych statycznych. Jak już wspomniano, tablica ta jest z technicznego punktu widzenia stała, co zostało prawidłowo wyrażone za pomocą wartości zwracanej przez funkcję **v()**.

Wartość zwracana przez funkcję **w()** określa, że zarówno wskaźnik, jak i to, co on wskazuje, muszą być stałymi. Podobnie jak w przypadku funkcji **v()**, wartość zwracana przez funkcję **w()** jest poprawna po powrocie z tej funkcji tylko dlatego, że jest ona statyczna. Nie chcemy nigdy zwracać wskaźników do lokalnych zmiennych umieszczonych na stosie, ponieważ po powrocie z funkcji i oczyszczeniu stosu nie są już one dostępne (innym powszechnie używanym wskaźnikiem, który mógłby zostać użyty, jest adres obszaru przydzielonego na stercie, dostępny po powrocie z funkcji).

Poszczególne funkcje są testowane, w obrębie funkcji **main()**, z różnymi argumentami. Funkcja **t()** przyjmuje jako argumenty wyłącznie wskaźniki niewskazujące stałych, natomiast w przypadku przekazania funkcji wskaźnika do stałej nie można zagwarantować, że funkcja ta pozostawi nienaruszoną wartość wskazywaną przez argument, dlatego też kompilator zgłasza komunikat o błędzie. Ponieważ funkcja **u()** pobiera wskaźnik do stałej, akceptuje oba rodzaje argumentów. Tak więc funkcja pobierająca wskaźnik do stałej jest funkcją bardziej ogólną niż taka, która pobiera jako argument zwykły wskaźnik.

Zgodnie z oczekiwaniemi, wartość zwracana przez funkcję **v()** może być przypisana tylko wskaźnikowi do stałej. Można się również spodziewać, że kompilator odrzuci przypisanie wartości zwracanej przez funkcję **w()** wskaźnikowi niewskazującemu stałej, natomiast zaakceptuje przypisanie jej zmiennej typu **const int* const**, a także, co może stanowić pewną niespodziankę, zaakceptuje przypisanie jej do zmiennej typu **const int***, która nie

odpowiada dokładnie zwracanemu typowi. Z uwagi na to, że kopowaną jest wartość (która jest adres przechowywany we wskaźniku), zapewnienie, że wskazywana przez nią wartość pozostanie nienaruszona, jest automatycznie podtrzymywane. Tak więc drugi modyfikator **const** w specyfikacji **const int* const** ma znaczenie tylko w przypadku użycia go jako *l-wartości*, co zostało obiektywnie przez kompilator.

Standardowy sposób przekazywania argumentów

W języku C typowe jest przekazywanie argumentów przez wartość, a w przypadku zamiaru przekazania adresu jedynym dostępnym rozwiązaniem jest użycie wskaźników¹. Jednakże żadne z tych podejść nie jest preferowane w języku C++. Najlepszym zamiast nich sposobem przekazywania argumentów jest przekazywanie ich przez referencję oraz przez referencję do stałej. Z punktu widzenia klienta-programistów używana składnia jest identyczna ze składnią przekazywania argumentów przez wartość, unika się więc zamieszania wywołanego wskaźnikiem. Z punktu widzenia twórcy funkcji przekazywanie adresu jest zawsze bardziej efektywne niż całego obiektu klasy, natomiast przekazywanie referencji do stałej oznacza, że funkcja nie zmienia wskazywanego przez nią obiektu, więc w efekcie, z punktu widzenia klienta-programistów taki sposób przekazywania argumentu będzie dokładnie tym samym, co przekazywanie go przez wartość (zapewni tylko większą efektywność).

Z uwagi na składnię referencji (z punktu widzenia wywołującego ma tę samą postać jak przekazywanie przez wartość) możliwe jest przekazanie tymczasowego obiektu funkcji, która pobiera referencję do stałej, podczas gdy nie ma nigdy możliwości przekazania obiektu tymczasowego funkcji pobierającej wskaźnik — w przypadku wskaźnika musi być bowiem jawnie podany adres. Tak więc przekazywanie przez referencję tworzy zupełnie nową sytuację, która nie występowała nigdy w języku C++ — obiekt tymczasowy, będący zawsze stałą, może przekazać funkcji swój adres. Dlatego właśnie, aby obiekty tymczasowe mogły być przekazywane funkcji przez referencję, jej argument musi być referencją do stałej. Demonstruje to poniższy przykład:

```
//: C08:ConstTemporary.cpp
// Obiekty tymczasowe są stałymi

class X {};

X f() { return X(); } // Wynik zwracany przez wartość

void g1(X&) {} // Przekazywanie przez referencję nie do stałej
void g2(const X&) {} // Przekazywanie przez referencję do stałej

int main() {
    // Błąd: stary obiekt tymczasowy tworzony przez f():
    //! g1(f());
    // W porządku: g2 pobiera referencję do stałej:
    g2(f());
} //:~
```

Niektórzy posuwając się do twierdzenia, że wszysko jest w języku C przekazywane przez wartość, ponieważ w przypadku przekazywania wskaźnika, tworzona jest jego kopia (a zatem wskaźnik jest przekazywany przez wartość). Jakkolwiek wydawałoby się to precyzyjne, myślę, że w rzeczywistości wprowadza zamieszanie w tej całej kwestii.

Funkcja **f()** zwraca obiekt klasy X *przez wartość*. Oznacza to, że jeżeli bezpośrednio przekaże się wartość zwracaną przez funkcję **f()** innej funkcji, jak w wywołaniu funkcji **g1()** i **g2()**, zostanie utworzony obiekt tymczasowy, i obiekt ten jest stałą. Wywołanie funkcji **g1()** jest błędem, ponieważ funkcja ta nie pobiera referencji do stałej, natomiast wywołanie funkcji **g2()** jest poprawne.

Istoty

W podroziale opisano, w jaki sposób używać modyfikatora `const` w stosunku do klas. Można utworzyć w obrębie klasy lokalną stałą, wykorzystywaną w stałych wyrażeniach, które będą wyliczane podczas komplikacji. Jednakże znaczenie modyfikatora `const` wewnątrz klas jest nieco odmienne, dlatego też, aby tworzyć stałe dane składowe klasy, trzeba najpierw zrozumieć wszystkie dostępne możliwości.

Można również uczynić cały obiekt stałą (jak się przekonaliśmy, kompilator zawsze tworzy stałe obiekty tymczasowe). Jednak kwestia zapewnienia niezmienności obiektów jest bardziej złożonym zagadnieniem. Kompilator może zapewnić niezmienność wbudowanego typu, ale nie jest w stanie śledzić zawiłości klas. W celu zagwarantowania niezmienności obiektu klasy wprowadzono stałe funkcje składowe — jedynie takie funkcje mogłyby być wywoływane w stosunku do obiektu będącego stałą.

Stałe w klasach

Jednym z miejsc, w których z pewnością zechcesz wykorzystać stałe umieszczone w wyrażeniach stałych, są klasy. Typowym tego przykładem jest tablica tworzona w obrębie klasy; do określenia jej wielkości zamiast dyrektywy `#define` wolno użyć stałej, wykorzystując ją również w obliczeniach odwołujących się do tej tablicy. Wielkość tablicy jest czymś, co można by ukryć wewnątrz klasy. Dzięki temu używając do jej określenia takiego identyfikatora, jak np. **size**, można by zastosować go również w innych klasach, bez obawy wywołania konfliktów nazw. Preprocesor traktuje wszystkie symbole zdefiniowane za pomocą dyrektywy `#define` jako globalne od miejsca wystąpienia ich definicji, więc używając ich nie można uzyskać pożądanego rezultatu.

Załóżmy, że umieszczenie stałej wewnątrz klasy jest decyzyją logiczną. Niestety, nie zapewnia ona uzyskania oczekiwanej efektu. W obrębie klas stałe w pewnym stopniu powracają do znaczenia, jakie miały one w języku C. Wewnątrz każdego obiektu jest im przydzielana pamięć i reprezentująone wartości, które sajednokrotnie inicjalizowane i nie mogą być później zmieniane. Użycie modyfikatora `const` wewnątrz klasy oznacza: „jest to wartość stała przez cały czas życia obiektu”. Jednak wartości tej stałej w różnych obiektach mogą być zróżnicowane.

Tak więc tworząc wewnątrz klasy zwykłą (nie statyczną) stałą, nie można jej nadać wartości początkowej. Oczywiście, **inicjalizacja taka** musi nastąpić w konstruktorze, ale służy do tego specjalna część konstruktora. Ponieważ stałą należy zainicjować w miejscu, w którym jest tworzona, musi ona być już zainicjowana wewnątrz ciała konstruktora. W przeciwnym przypadku można by wstrzymać się z inicjalizacją do jakiegoś

miejsca w dalszej części ciała konstruktora, co oznaczałoby, że stała była przez pewien czas niezainicjowana. W takim przypadku nic również nie mogłoby powstrzymać programisty przed zmianą wartości stałej w różnych miejscach konstruktora.

Lista inicjatorów konstruktora

To szczególnie miejsce, w którym odbywa się inicjalizacja, nazywane jest *listą inicjatorów konstruktora* i powstało pierwotnie z myślą o dziedziczeniu (opisany w rozdziale 14.). Lista inicjatorów konstruktora — która, jak wynika z nazwy, występuje wyłącznie w definicji konstruktora — jest listą „wywołań konstruktora”, umieszczoną po liście jego argumentów i dwukropku, a przed nawiasem klamrowym rozpoczynającym ciało konstruktora. Przypomina to o tym, że inicjalizacje znajdujące się na liście odbywają się przed wykonaniem jakiegokolwiek kodu, wchodzącego w skład konstruktora. Jest to miejsce, w którym lokuje się wszystkie inicjalizacje stałych. Poniżej przedstawiono przykład prawidłowej inicjalizacji stałych, znajdujących się wewnątrz klasy:

```
//: C08:ConstInitialization.cpp
// Inicjalizacja stałych w klasach
#include <iostream>
using namespace std;

class Fred {
    const int size;
public:
    Fred(int sz);
    void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
    Fred a(1), b(2), c(3);
    a.print(), b.print(), c.print();
} III-
```

Przedstawiona powyżej postać listy inicjatorów konstruktora wyda ci się początkowo dziwna, ponieważ nie znasz jeszcze typu wbudowanego, traktowanego jakby posiadał konstruktor.

„Konstruktory” typów wbudowanych

W miarę rozwoju języka i wysiłków czynionych w celu upodobnienia typów zdefiniowanych przez użytkownika do typów wbudowanych stało się oczywiste, że w niektórych sytuacjach typy wbudowane powinny działać podobnie do typów zdefiniowanych przez użytkownika. Na liście inicjatorów konstruktora typ wbudowany może być traktowany w taki sposób, jakby miał konstruktor:

```
//: C08:BuiltInTypeConstructors.cpp
#include <iostream>
using namespace std;
```

```

class B {
    int i;
public:
    B(int ii);
    void print();
};

B::B(int ii) : i(ii) {}
void B::print() { cout << i << endl; }

int main(){
    B a(1), b(2);
    float pi(3.14159);
    a.print(); b.print();
    cout << pi << endl;
} //:-

```

Ma to szczególnie znaczenie podczas inicjalizacji danych składowych, będących stałymi, ponieważ muszą one zostać zainicjalowane jeszcze przed wejściem do ciała funkcji.

Rozszerzenie tego „konstruktora”, przeznaczonego dla typów wbudowanych (w tym przypadku oznacza on przypisanie), na przypadki ogólne wydaje się logiczne, dzięki czemu definicja **float pi(3.14159)**, widoczna w powyższym kodzie, działa prawidłowo.

Często przydatne jest zamknięcie wbudowanego typu w klasie, co gwarantuje mu inicjalizację za pomocą konstruktora. Poniższy przykład przedstawia utworzoną w taki sposób klasę **Integer**:

```

//: C08:EncapsulatingTypes.cpp
#include <iostream>
using namespace std;

class Integer {
    int i;
public:
    Integer(int ii = 0);
    void print();
};

Integer::Integer(int ii) : i(ii) {}
void Integer::print() { cout << i << ' ' }

int main() {
    Integer i[100];
    for(int j = 0; j < 100; j++)
        i[j].print();
} //:-

```

Wszystkie elementy tablicy obiektów klasy **Integer**, zdefiniowanej w funkcji **main()**, są automatycznie **inicjalizowane** wartością zerową. Inicjalizacja taką niekoniecznie musi być bardziej kosztowna niż przeprowadzona w pętli **for** lub za pomocą funkcji **memset()**. Wiele kompilatorów bez trudu optymalizuje taki kod, dzięki czemu cały proces odbywa się bardzo szybko.

State o wartościach określonych podczas komplikacji, zawarte w klasach

Zaprezentowany powyżej sposób użycia stałych jest interesujący i prawdopodobnie przydatny w wielu przypadkach. Jak jednak rozwiązać pierwotny problem, zawarty w pytaniu: „w jaki sposób utworzyć wewnątrz klas stałe o wartościach określonych podczas komplikacji?”. Odpowiedź wymaga użycia dodatkowego słowa kluczowego **static**, które zostanie wyczerpująco opisane w rozdziale 10. W tym przypadku słowo kluczowe **static** oznacza: „istnieje tylko jeden egzemplarz, niezależnie od tego, ile utworzono obiektów tej klasy”. O to nam właśnie chodziło — o składową klasy, będącą **stałą**, która ma taką samą wartość dla wszystkich obiektów tej klasy. Tak więc statyczne stałe wbudowanych typów danych mogą być traktowane jako stałe o wartościach określonych podczas komplikacji.

Istnieje pewna niezwykła cecha stałych statycznych, używanych wewnątrz klas — wartość inicjująca musi być im dostarczona w miejscu, w którym zostały one zdefiniowane. Dzieje się tak tylko w przypadku stałych statycznych — jeśliby taki sposób inicjalizacji został użyty w stosunku do innych danych składowych, nie będzie on poprawny, ponieważ wszystkie pozostałe dane składowe klasy muszą być zainicjowane w konstruktorze lub w innych funkcjach składowych.

W poniższym przykładzie przedstawiono sposób utworzenia i wykorzystania statycznej stałej **size** wewnątrz klasy reprezentującej stos wskaźników do łańcuchów :

```
//: C08:StringStack.cpp
// Wykorzystanie stałej statycznej do utworzenia
// wewnątrz klasy stałej o wartości określonej
// podczas komplikacji
#include <string>
#include <iostream>
using namespace std;

class StringStack {
    static const int size = 100;
    const string* stack[size];
    int index;
public:
    StringStack();
    void push(const string* s);
    const string* pop();
};

StringStack::StringStack() : index(0) {
    memset(stack, 0, size * sizeof(string*));
}

void StringStack::push(const string* s) {
    if(index < size)
        stack[index++] = s;
}
```

² Kiedy pisany był niniejszy tekst, nie wszystkie kompilatory umożliwiali wykorzystywanie tej cechy.

```

const string* StringStack::pop() {
    if(index > 0) {
        const string* rv = stack[--index];
        stack[index] = 0;
        return rv;
    }
    return 0;
}

string iceCream[] = {
    "bakallowo-smietankowe",
    "karmelowe",
    "migdalowe",
    "czarna porzeczka",
    "sorbet malinowy",
    "cytrynowe",
    "czekoladowe",
    "karmelowe w gorzkiej czekoladzie"
};

const int iCsz =
    sizeof iceCream / sizeof *iceCream;

int main() {
    StringStack ss;
    for(int i = 0; i < iCsz; i++)
        ss.push(&iceCream[i]);
    const string* cp;
    while((cp = ss.pop()) != 0)
        cout << *cp << endl;
} //:-

```

Ponieważ stała **size** została użyta do określenia rozmiaru tablicy **stack**, jest ona rzeczywiście stałą o wartości określonej w czasie komplikacji, ale została ukryta wewnątrz klasy.

Zwróć uwagę na to, że funkcja **push()** pobiera argument typu **const string***, funkcja **pop()** zwraca wartość typu **const string***, a klasa **StringStack** przechowuje wartości typu **const string***. Gdyby tak nie było, nie można by używać obiektu klasy **StringStack** do przechowywania wskaźników, zawartych w tablicy **iceCream**. Równocześnie uniemożliwia to zrobienie czegokolwiek, co zmieniłoby zawartość obiektów przechowywanych w **StringStack**. Oczywiście, nie wszystkie klasy-kontenery są projektowane z uwzględnieniem takiego ograniczenia.

„Wyliczeniowy wytrych” w starym kodzie

W starszych wersjach języka C++ stałe statyczne nie mogły znajdować się w klasach. Oznacza to, że stałych nie dało się również wykorzystać w wyrażeniach stałych, zawartych w klasach. Istniała jednak potrzeba takiego stosowania stałych, więc typowym rozwiązaniem (nazywanym zazwyczaj „wyliczeniowym wytrychem”) było użycie nienazwanego wyliczenia, nieposiadającego egzemplarza zmiennej. Wyliczenie jest lokalne w stosunku do zawierającej je klasy; jego wszystkie wartości muszą być określone w czasie komplikacji, a ponadto można je wykorzystać w wyrażeniach stałych. Tak więc często występuje następujące zastosowanie wyliczeń:

```
//: C08:EnumHack.cpp
#include <iostream>
using namespace std;

class Bunch {
    enum { size = 1000 };
    int i[size];
};

int main() {
    cout << "sizeof(Bunch) = " << sizeof(Bunch)
        << ", sizeof(i[1000]) = "
        << sizeof(int[1000]) << endl;
} ///:~
```

Prezentowany tutaj sposób użycia wyliczenia na pewno nie zajmuje pamięci w obiekcie, a wszystkie wyliczenia są wyliczane w trakcie komplikacji. Można również w jawnym sposób nadać wartości poszczególnym elementom wyliczenia:

```
enum { jeden = 1, dwa = 2, trzy };
```

W przypadku liczb całkowitych kompilator kontynuuje liczenie, zaczynając od ostatniej nadanej wartości, więc element wyliczenia trzy otrzyma wartość 3.

W zamieszczonym wcześniej pliku przykładowym StringStack.cpp wiersz:

```
static const int size = 100;
```

mógliby mieć postać:

```
enum { size = 100 };
```

Mimo że w starszych programach często można spotkać technikę wykorzystującą wyliczenia, stałe statyczne zostały dodane do języka właśnie po to, aby rozwiązać ten problem. Nie jednak żadnego istotnego powodu, dla którego *należało*by używać stałych statycznych zamiast „wytrychu wyliczeniowego”. W przykładach zawartych w książce jest on używany tylko dlatego, że kiedy była ona pisana, mechanizm ten obsługiwał większą liczbę kompilatorów.

Stałe obiekty i funkcje składowe

Funkcje składowe klas mogą być również określone jako stałe. Cóż to oznacza? Aby to zrozumieć, trzeba najpierw poznać pojęcie stałych obiektów.

Stały obiekt typu zdefiniowanego przez użytkownika definiuje się tak samo, jak obiekt typu wbudowanego. Na przykład:

```
const int i = 1;
const blob b(2);
```

Powyższa instrukcja definiuje b jako stały obiekt typu **blob**. Jego konstruktor jest wywoływany z argumentem równym dwa. Aby kompilator mógł zagwarantować niezmienność obiektu, musi on zapewnić, by w czasie życia tego obiektu nie zostały

zmienione żadne jego dane składowe. Nie sprawia to trudności w stosunku do publicznych danych klasy, ale skąd wiadomo, które funkcje składowe są „bezpieczne” dla stałych obiektów, a które zmieniają ich dane?

Deklarując funkcję składową z modyfikatorem **const**, informuje się kompilator, że funkcja ta może zostać wywołana w stosunku do obiektu będącego stałą. Funkcja składowa niezadeklarowana jawnie z tym modyfikatorem jest traktowana jako taka, która modyfikuje dane obiektu. Kompilator nie pozwoli na jej wywołanie w stosunku do stałego obiektu.

Na tym się jednak nie kończy. *Ogłoszenie*, że funkcja składowa jest „stała”, nie gwarantuje jeszcze, że będzie ona działała w taki właśnie sposób. Kompilator wymusza więc powtórzenie modyfikatora **const** w definicji tej funkcji (słowo **const** staje się częścią sygnatury funkcji, dzięki czemu zarówno kompilator, jak i program łączący sprawdzają jej „stałość”). Następnie kompilator wymusza nienaruszalność danych w obrębie definicji funkcji, zgłaszając błąd w przypadku próby zmiany jakiekolwiek składowej lub wywołania funkcji nieoznaczonej modyfikatorem **const**. Tak więc istnieje gwarancja, że każda funkcja składowa zadeklarowana jako „stała” będzie zachowywała się w taki sposób w swojej definicji.

Aby zrozumieć składnię deklaracji stałych funkcji składowych, należy zwrócić uwagę na to, że poprzedzenie nazwy funkcji modyfikatorem **const** oznacza, że funkcja ta zwraca wartość będącą stałą, nie przynosi więc spodziewanego rezultatu. Zamiast tego należy umieścić słowo kluczowe **const** po liście argumentów. Na przykład:

```
//: C08:ConstMember.cpp
class X {
    int i;
public:
    X(int ii);
    int f() const;
};

X::X(int ii) : i(ii) {}

int X::f() const { return i; }

int main() {
    X x1(10);
    const X x2(20);
    x1.f();
    x2.f();
} //:-
```

Zwróć uwagę na to, że słowo kluczowe **const** musi być powtórzone w definicji funkcji, bowiem w przeciwnym razie kompilator uznajązą definicję innej funkcji. Ponieważ **f()** jest stałą funkcją składową, więc gdyby próbowała ona w jakikolwiek sposób zmienić wartość składowej i lub wywołać inną funkcję składową, która nie została zadeklarowana jako stała, kompilator zgłosiłby błąd.

A zatem stałe funkcje składowe mogą być wywołane zarówno w stosunku stałych obiektów, jak i obiektów niebędących stałymi. Można je zatem uznać za bardziej ogólną postać funkcji składowych (to niefortunne, że funkcje składowe klas nie są

domyślnie funkcjami stałymi). Każda funkcja, która nie modyfikuje danych, powinna zostać zadeklarowana jako funkcja stała, dzięki czemu wcześniej można użyć w stonku do stałych obiektów klasy.

Poniżej zamieszczono przykład przedstawiający porównanie stałych i „niestandardowych” funkcji składowych:

```
//: C08:Quoter.cpp
// Losowy wybór cytatu
#include <iostream>
#include <cstdlib> // Generator liczb losowych
#include <ctime> // Oo inicjalizacji generatora liczb losowych
using namespace std;

class Quoter {
    int lastquote;
public:
    Quoter();
    int lastQuote() const;
    const char* quote();
};

Quoter::Quoter(){
    lastquote = -1;
    srand(time(0)); // Inicjalizacja generatora liczb losowych
}

int Quoter::lastQuote() const {
    return lastquote;
}

const char* Quoter::quote() {
    static const char* quotes[] = {
        "Czy mamy się juz smiac?",
        "Lekarze zawsze wiedza najlepiej",
        "Sowy nie sa tym. czym sie wydaja",
        "Strach ma wielkie oczy",
        "Nie ma naukowego dowodu ",
        "na poparcie tezy ",
        "ze zycie jest na serio",
        "To. co nas bawi, czyni nas mądrzejszymi",
    };
    const int qsize = sizeof quotes/sizeof *quotes;
    int qnum = rand() % qsize;
    while(lastquote >= 0 && qnum == lastquote)
        qnum = rand() % qsize;
    return quotes[lastquote - qnum];
}

int main() {
    Quoter q;
    const Quoter cq;
    cq.lastQuote(); // W porządku
    //! cq.quote(); // Źle - nie jest to funkcja stala
    for(int i = 0; i < 20; i++)
        cout << q.quote() << endl;
} //!~
```

Ani konstruktor, ani destruktor nie mogą być stałymi funkcjami, ponieważ zawsze dokonują modyfikacji obiektu w czasie inicjalizacji oraz sprzątania. Funkcja składowa **quote()** również nie jest stała, ponieważ modyfikuje ona składową **lastquote** (co widać w instrukcji **return**). Jednakże funkcja **lastQuote()** nie dokonuje żadnych zmian, więc może ona być funkcją stałą i można ją bezpiecznie wywołać dla stałego obiektu **eq**.

mutable: niezmienność fizyczna kontra niezmienność logiczna

Co zrobić w przypadku, gdy zamierzasz utworzyć stałą funkcję składową, ale nadal chcesz zmieniać w obiekcie pewne dane? Czasami określa się to jako różnicę pomiędzy *niezmiennością fizyczną* (zwana również *niezmiennością bitową*) i *niezmiennością logiczną*. Pierwszy z wymienionych terminów oznacza, że każdy bit obiektu jest niezmienny, dzięki czemu obraz bitów obiektu nigdy się nie zmienia. Niezmienność logiczna natomiast, że chociaż cały obiekt jest pojęciowo stały, to jednak mogą zmieniać się niektóre jego składowe. Jednakże jeżeli określi się, że obiekt jest stały, to kompilator będzie strzegł go zazdrośnie, zapewniając jego niezmienność fizyczną. Istnieją dwa sposoby uzyskania niezmienności logicznej, umożliwiające zmianę danych składowych wewnętrz stałych funkcji składowych.

Pierwsze podejście ma charakter historyczny i nazywane jest *odrzuceniem niezmienności*. Uzyskuje się je w dość dziwny sposób. Pobiera się wskaźnik **this** (słowo kluczowe, zwracające adres bieżącego obiektu) i rzutuje się go na wskaźnik do obiektu bieżącego typu. Wydawałoby się, że **this** już jest takim wskaźnikiem. Jednak wewnątrz stałej funkcji składowej jest on w rzeczywistości wskaźnikiem do stałej, więc rzutując go na zwykły wskaźnik, usuwa się niezmienność wskazywanego obiektu. Poniżej zamieszczono odpowiedni przykład:

```
//: C08:Castaway.cpp
// "Odrzucenie" niezmienności

class Y {
    int i;
public:
    Y();
    void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
// i++; // Błąd - stała funkcja składowa
((Y*)this)->i++; // W porządku - odrzucenie niezmienności
// Lepsze rozwiązanie - użycie składni jawnego rzutowania C++:
(const_cast<Y*>(this))->i++;
}

int main() {
    const Y yy;
    yy.f(); // W rzeczywistości zmienia obiekt!
} //:-
```

Podejście takie jest skuteczne i występuje w starszych programach, ale nie jest to zalecana technika. Problem polega na tym, że brak niezmienności obiektu został ukryty w definicji funkcji **składowej**, i nie ma jakichkolwiek przesłanek, zawartych w interfejsie klasy, wskazujących na to, że dane obiektu są w rzeczywistości modyfikowane, chyba że ma się dostęp do kodu źródłowego (a także jeśli ze względu na podejrzenie, że niezmienność obiektu została naruszona, poszuka się w tym kodzie rzutowań). Aby podać to wszystko do ogólnej wiadomości, należy w deklaracji klasy użyć słowa klu-czowego **mutable** („podlegający zmianom”), oznaczającego, że określona składowa może zostać zmodyfikowana wewnątrz stałego obiektu:

```
//: C08:Mutable.cpp
// Słowo kluczowe "mutable"

class Z {
    int i;
    mutable int j;
public:
    Z();
    void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
//! i++; // Błąd - stała funkcja składowa
    j++; // W porządku - składowa może być zmieniana
}

int main() {
    const Z zz;
    zz.f(); // Funkcja modyfikuje obiekt!
} ///:~
```

Dzięki temu użytkownik klasy wie na podstawie jej deklaracji, które składowe mogą być modyfikowane wewnątrz stałych funkcji składowych.

Możliwość umieszczenia obiektu w pamięci ROM

Jeżeli obiekt został zdefiniowany jako stała, jest kandydatem do umieszczenia w pa-mięci przeznaczonej tylko do odczytu (ROM), co jest często istotną **okoliczością** w przypadku programowania systemów wbudowanych. Jednak nie wystarczy w tym przypadku zwykłe uczynienie obiektu stałą — wymagania umożliwiające umieszcze-nie obiektu w pamięci ROM są znacznie ostrzejsze. Oczywiście, obiekt musi być nie-zmienny fizycznie, a nie logicznie. Łatwo to zauważyc, gdy niezmienność logiczna jest implementowana wyłącznie za pomocą słowa klu-czowego **mutable**. Jednakże przypuszczalnie kompilator nie jest w stanie wykryć tego, że niezmienność jest od-rzucana wewnątrz stałej funkcji składowej. Dodatkowo:

1. Klasa lub struktura nie **może posiadać** zdefiniowanych przez użytkownika konstruktów ani destruktów.
2. Nie mogą istnieć żadne klasy podstawowe (opisane w rozdziale 14.) ani obiekty składowe, posiadające zdefiniowane przez użytkownika konstruktory lub destruktory.

Wynik operacji zapisywania, dotyczącej jakiejkolwiek części obiektu stałego, zapisanego w pamięci ROM, nie jest zdefiniowany. Mimo że odpowiednio przygotowane obiekty mogłyby umieszczone w pamięci ROM, to żaden obiekt nigdy tego *nie wymaga*.

/volatile

Składnia modyfikatora **volatile** (ulotny) jest identyczna jak składnia słowa kluczowego **const**, lecz **volatile** oznacza: „te dane mogą być modyfikowane bez wiedzy kompilatora”. Czasami dane są modyfikowane przez środowisko (prawdopodobnie za pomocą wielozadaniowości, wielowątkowości lub przerwań) i w takim przypadku słowo kluczowe **volatile** informuje kompilator, by nie przyjmował żadnych założeń dotyczących danych — szczególnie związanych z optymalizacją.

Jeżeli kompilator wczytał już dane do rejestru i nie zmieniał od tej pory jego zawartości, to zwykle nie musiałby odczytywać tych danych ponownie. Jednak jeżeli dane te zostały oznaczone modyfikatorem **volatile**, to kompilator nie może już robić takich założeń, ponieważ dane mogły być zmienione przez inny proces. W związku z tym kompilator musi odczytać te dane ponownie, nie dokonując optymalizacji kodu, która polegałaby na usunięciu z tego, co zwykle byłoby nadmiarową operacją odczytu.

Obiekty „ulotne” są tworzone za pomocą tej samej składni, której używa się w przypadku obiektów stałych. Można nawet generować obiekty typu **const volatile**, które nie mogłyby być modyfikowane przez klienta-programistę, ale zamiast tego byłyby zmieniane przez jakiś czynnik zewnętrzny. Oto przykład, mogący stanowić klasę związaną z jakimś urządzeniem komunikacyjnym:

```
//: C08:Volatile.cpp
// Słowo kluczowe volatile

class Comm {
    const volatile unsigned char byte;
    volatile unsigned char flag;
    enum { bufsize = 100 };
    unsigned char buf[bufsize];
    int index;
public:
    Comm();
    void isr() volatile;
    char read(int index) const;
}:

Comm::Comm() : index(0), byte(0), flag(0) {}

// To tylko demonstracja - funkcja w rzeczywistości
// nie będzie działać jako procedura obsługi przerwań:
void Comm::isr() volatile {
    flag = 0;
    buf[index++] = byte;
    // Przewinięcie do początku bufora:
    if(index >= bufsize) index = 0;
}
```

```
char Comm::read(int index) const {
    if(index < 0 || index >= bufsize)
        return 0;
    return buf[index];
}

int main() {
    volatile Comm Port;
    Port.isr(); // W porządku
    //! Port.read(0); // Błąd, funkcja read() nie jest typu volatile
} //!:-
```

Podobnie jak w przypadku modyfikatora **const**, słowa kluczowe **volatile** można używać w stosunku do danych składowych, funkcji składowych oraz samych obiektów. W stosunku do obiektów, zdefiniowanych z modyfikatorem **volatile**, można wywoływać jedynie funkcje typu **volatile**.

Powodem, dla którego funkcja **isr()** nie mogłaby być w rzeczywistości wykorzystana w charakterze procedury obsługi przerwań, jest to, że w funkcjach składowym klas musi być przekazany, w ukryty sposób, wskaźnik bieżącego obiektu (**this**), natomiast procedury obsługi przerwań na ogół nie wymagają żadnych argumentów. Aby rozwiązać ten problem, można uczynić funkcję **isr()** statyczną funkcją składową — funkcje takie zostały opisane w rozdziale 10.

Składnia modyfikatora **volatile** jest taka sama, jak składnia modyfikatora **const**, dla tego też oba modyfikatory są często omawiane razem. Są one określane łącznie mianem *modyfikatora c-v*.

Podsumowanie

Słowo kluczowe **const** umożliwia definiowanie obiektów, argumentów funkcji, zwartych wartości oraz funkcji składowych jako stałych, pozwalając zarazem na rezygnację z oferowanego przez preprocesor zastępowania nazw, bez utraty żadnej z korzyści wynikających ze stosowania preprocesora. Wszystko to umożliwia istotną, dodatkową formę kontroli typów, a także bezpieczeństwo związane z programowaniem. Wykorzystanie tak zwanej *poprawności stosowania stałych*³ (używanie stałych wszędzie, gdzie to możliwe) może ocalić niejeden projekt.

Mimo że można zignorować słowo kluczowe **const** i używać nadal dawnych praktyk, stosowanych w języku C, to istnieje ono po to, by ci pomóc. Poczynając od rozdziału 11. są intensywnie wykorzystywane referencje; dopiero wówczas przekonasz się w pełni o tym, jak ważne jest używanie modyfikatora **const** w stosunku do argumentów funkcji.

W oryginalne *const correctness* — określenie będące parafrą „poprawności politycznej” (*political correctness*) — przyp. tłum.

ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Utwórz trzy **stałe** całkowite, następnie dodaj je do siebie w celu utworzenia wartości, określającej wielkość tablicy (w definicji tej tablicy). Spróbuj skompilować ten sam kod źródłowy w języku C i zobacz, co się stanie (na ogół można zmusić kompilator C++ do tego, by działał jako kompilator języka C, używając do tego parametru podanego w wierszu polecen).
2. Udowodnij, że kompilatory języków C i C++ rzeczywiście różnie traktują stałe. Utwórz stałą globalną i wykorzystaj ją w globalnym wyrażeniu stałym. Następnie skompiluj ten program, używając zarówno kompilatora C, jak i C++.
3. Utwórz przykładowe definicje stałych dla wszystkich typów wbudowanych oraz ich wariantów. Użyj ich w połączeniu z innymi stałymi, tworząc nowe definicje stałych. Upewnij się, że kompilują się one poprawnie.
4. W pliku nagłówkowym utwórz definicję stałej. Następnie dołącz ten plik nagłówkowy do dwóch plików **.cpp**, skompiluj oba pliki i dokonaj ich łączenia. Nie powinno się otrzymać żadnych komunikatów o błędach. Teraz spróbuj przeprowadzić ten sam eksperyment z językiem C.
5. Utwórz **stałą**, której wartość jest określana w trakcie pracy programu, przez odczytanie — podczas uruchamiania programu — aktualnego czasu. (Należy użyć standardowego pliku nagłówkowego **<ctime>**). W dalszej części programu spróbuj ponownie wczytać do swojej stałej aktualny czas i zobacz, co się stanie
6. Utwórz stałą tablicę znaków, a następnie spróbuj zmienić jeden z nich.
7. Utwórz deklarację stałej, zadeklarowanej w jednym pliku jako **extern const**, a następnie umieść w tym pliku funkcję **main()**, drukującą wartość tej stałej. Umieść definicję tej stałej w drugim pliku, a następnie skompiluj i połącz ze sobą oba pliki.
8. Utwórz dwa **wskaźniki** do typu **const long**, używając obu postaci deklaracji. Przypisz jednemu z nich adres tablicy wartości **long**. Wykaż, że możesz inkrementować i dekrementować wskaźnik, ale nie możesz zmienić wskazywanej przez niego wartości.
9. Utwórz stały wskaźnik do typu **double** i przypisz mu adres tablicy wartości typu **double**. Wykaż, że możesz zmieniać wartość, wskazywaną przez wskaźnik, ale nie możesz go inkrementować ani dekrementować.
10. Utwórz stały wskaźnik do stałego obiektu. Wykaż, że możesz jedynie odczytać wskazywaną przez niego wartość, ale nie możesz zmieniać ani jego, ani wartości przez niego wskazywanej.
11. Usuń komentarz w wierszu wywołującym błąd w programie **PointerAssignment.cpp** i zobacz, jaki błąd zgłosi Twój kompilator.

12. Utwórz literał napisowy oraz wskaźnik, który wskazuje na początek tego literała. Następnie użyj wskaźnika do modyfikacji obiektów, znajdujących się w tablicy. Czy twój kompilator zasygnalizuje tojako błąd? Jeżeli nie, to co otym myślisz?
13. Utwórz funkcję, pobierającą przez wartość argument będący stałą, a następnie spróbuj, w ciele funkcji, zmienić wartość tego argumentu.
14. Utwórz funkcję, pobierającą przez wartość argument typu **float**. Wewnątrz funkcji powiąż z tym argumentem — przez referencję — stałą typu **const float&** i odtąd używaj wyłącznie tej referencji, by zagwarantować, że argument funkcji nie zostanie zmieniony.
15. Zmodyfikuj program **ConstReturnValue.cpp**, usuwając indywidualnie komentarze w wierszach wywołujących błędy, by przekonać się, jakie komunikaty o błędach zgłasza twój kompilator.
16. Zmodyfikuj program **ConstPointer.cpp**, usuwając indywidualnie komentarze w wierszach wywołujących błędy, by zobaczyć, jakie komunikaty o błędach zgłasza twój kompilator.
17. Utwórz nową wersję pliku **ConstPointer.cpp**, noszącą nazwę **ConstReference.cpp**, która demonstruje użycie referencji zamiast wskaźników (być może będziesz musiał w tym celu zatrzymać się na rozdziale 11.).
18. Zmodyfikuj program **ConstTemporary.cpp**, usuwając komentarz w wierszu wywołującym błąd, by zobaczyć, jakie komunikaty o błędach zgłasza twój kompilator.
19. Utwórz klasę zawierającą zarówno stałe, jak i „niestale” elementy typu **float**. Zainicjalizuj je, używając listy inicjatorów konstruktora.
20. Utwórz klasę o nazwie **MyString**, zawierającąłańcuch (string), konstruktor inicjalizujący ten łańcuch oraz funkcję **print()**. Zmodyfikuj plik **StringStack.cpp** w taki sposób, aby zawarty w nim kontener przechowywał obiekty klasy **MyString**, a funkcję **main()** — w taki sposób, byje wydrukowała.
21. Utwórz klasę **zawierającą składową stałą**, która zostanie zainicjowana na liście inicjatorów konstruktora, a także nienazwane wyliczenie, którego użyjesz do określenia rozmiaru tablicy,
22. W pliku **ConstMember.cpp**, z definicji funkcji składowej usuń modyfikator **const**, ale zostaw go w deklaracji, aby zobaczyć, jaki otrzymasz rodzaj komunikatu o błędzie.
23. Utwórz klasę, zawierającą zarówno stałe funkcje składowe, jak i funkcje składowe niebędące stałymi. Utwórz stałe obiekty tej klasy oraz obiekty niebędące stałymi, a następnie spróbuj wywołać różne rodzaje funkcji składowych dla różnych typów obiektów.
24. Utwórz klasę, zawierającą zarówno stałe funkcje składowe, jak i funkcje składowe niebędące stałymi. Spróbuj wywołać funkcję składową niebędącą stałą ze stałej funkcji składowej i zobaczyć, jaki otrzymasz rodzaj komunikatu o błędzie.

25. W pliku **Mutable.cpp** usuń komentarz w wierszu wywołującym błąd, aby zobaczyć, jaki rodzaj komunikatu o błędzie zgłosi twój kompilator.
26. Zmodyfikuj plik **Quoter.cpp**, czyniąc funkcję **quote()** stałą funkcją składową i dodając do składowej **lastquote** słowo kluczowe **mutable**.
27. Utwórz klasę o danych składowych, posiadających modyfikator **volatile**. Utwórz funkcje składowe — zarówno posiadające modyfikator **volatile**, jak i go nieposiadające — które **modyfikują dane składowe typu volatile** i zobaczą, w jaki sposób zareaguje na to kompilator. Utwórz obiekty swojej klasy, zarówno z modyfikatorem **volatile**, jak i bez tego modyfikatora. Spróbuj w stosunku do nich wywoływać zarówno funkcje składowe zadeklarowane z modyfikatorem **volatile,jak** i bez niego, by zobaczyć, czy się to powiedzie, a także jakie komunikaty o błędach zgłosi kompilator.
28. Utwórz klasę o nazwie **bird**, zawierającą funkcję składową **fly()**, oraz klasę **rock**, nieposiadającą takiej funkcji. Utwórz obiekt klasy **rock**, pobierz jego adres i przypisz go wskaźnikowi typu **void***. Następnie przypisz wartość tego wskaźnika wskaźnikowi typu **bird*** (należy użyć do tego celu rzutowania) i za pomocą tego wskaźnika wywołaj funkcję **fly()**. Czy teraz już rozumiesz, dlaczego dostępna w języku C możliwość swobodnego przypisywania wartości wskaźnika **void*** (bez rzutowania) stanowi w tym języku „lukę”, która nie powinna być rozszerzana na język C++?

Rozdział 9.

Funkcje inline

Do ważniejszych cech języka C++, wywodzących się z języka C, należy efektywność. Gdyby była ona znacznie niższa niż w przypadku języka C, liczni programiści nie potrafiliby znaleźć przekonujących argumentów, przemawiających na korzyść C++.

Jednym ze sposobów osiągnięcia efektywności w języku C jest wykorzystanie makroinstrukcji. Pozwalają one na utworzenie czegoś, co przypomina wywołanie funkcji, ale pozbawione jest związanego z tym zazwyczaj narzutu. Makroinstrukcje są implementowane za pomocą preprocesora, a nie właściwości kompilatora. Preprocesor zastępuje bezpośrednio wszystkie wywołania makroinstrukcji ich kodem, co pozwala na uniknięcie kosztów związanych z umieszczeniem argumentów na stosie, wykonaniem instrukcji CALL asemblera, pobraniem zwracanych argumentów i wykonaniem instrukcji RETURN asemblera. Cała praca zostaje wykonana przez preprocesor, dzięki czemu uzyskuje się wygodę i czytelność, odpowiadającą wywołaniu funkcji, nie ponosząc żadnych kosztów.

Z użyciem makroinstrukcji preprocesora w języku C++ wiążą się dwa problemy. Pierwszy z nich występuje również w języku C — makroinstrukcja wygląda co prawda identycznie jak wywołanie funkcji, ale nie zawsze działa tak samo. Może to prowadzić do powstania trudnych do wykrycia błędów. Drugi problem jest związany z językiem C++ — preprocesor nie ma prawa dostępu do danych składowych klas. Oznacza to, że makroinstrukcje preprocesora nie mogą zostać użyte w charakterze funkcji składowych.

W celu utrzymania efektywności makroinstrukcji procesora i równocześnie zapewnienia bezpieczeństwa i zasięgu w obrębie klasy, właściwych prawdziwym funkcjom, udostępniono w języku C++ *funkcje inline*. W niniejszym rozdziale prześledzimy problemy związane z wykorzystaniem makroinstrukcji preprocesora w języku C++, a także opiszemy, w jaki sposób zostały one rozwiązane za pomocą funkcji inline. Omówione zostaną również wskazówki oraz szczegóły związane z działaniem funkcji inline.

Ułapki preprocessora

Istotą problemów związanych z wykorzystaniem makroinstrukcji preprocessora jest błędne mniemanie, że preprocesor zachowuje się identycznie jak kompilator. Oczywiście, intencją twórców było to, aby makroinstrukcje *wyglądały* i działały tak, jak wywołania funkcji, więc dość łatwo można nabrać takiego przekonania. Problemy wynikają wówczas, gdy ujawniają się niewielkie różnice między nimi.

Tytułem prostego przykładu przyjrzyjmy się definicji:

```
#define F (x) (x + 1)
```

Jeżeli wykonamy wywołanie makroinstrukcji F:

```
F(1)
```

to preprocesor rozwinięje nieco niespodziewanie do postaci:

```
(x) (x + 1)(1)
```

Przyczyną pojawienia się problemu jest występujący w definicji makroinstrukcji odstęp pomiędzy literą F i nawiasem otwierającym. Po usunięciu z definicji odstępu można wywołać makroinstrukcję, nawet wpisując odstęp przed nawiasem:

```
F (1)
```

i nadal będzie ona rozwijana poprawnie do postaci:

```
(1 + 1)
```

Przedstawiony powyżej przykład jest dość banalny i zawarty w nim problem natychmiast by się ujawnił. Prawdziwe problemy występują wówczas, gdy w charakterze parametrów wywołania makroinstrukcji używane są wyrażenia.

Występują tu dwa problemy. Pierwszy z nich polega na tym, że wyrażenie może być rozwinięte wewnętrz makroinstrukcji, co powoduje, że kolejność wykonywania operatorów będzie odbiegała od spodziewanej. Rozważmy na przykład definicję:

```
#define FLOOR(x.b) x>=b?0:l
```

Jeżeli teraz jako argumenty użyte zostaną wyrażenia:

```
if(FLOOR(a&0x0f,0x07)) // ...
```

to makroinstrukcja zostanie rozwinięta do postaci:

```
if(a&0x0f>=0x07?0:l)
```

Priorytet operatora & jest niższy niż >=, więc sposób wyliczenia makroinstrukcji nie będzie zgodny z oczekiwaniami. Po ustaleniu przyczyny problemu można go rozwiązać, umieszczając w nawiasach wszystkie elementy, znajdujące się w definicji makroinstrukcji (warto stosować ten sposób w przypadku tworzenia makroinstrukcji preprocessora). Należy więc zapisać:

```
#define FLOOR(x.b) ((x)>-(b)?0:l)
```

Jednak odkrycie problemu jest niekiedy trudne i może on pozostać nieauważony, jeśli przyjmie się poprawne działanie makroinstrukcji za pewnik. W pozbawionej nawiasów, poprzedniej wersji makroinstrukcji, większość wyrażeń będzie obliczana prawidłowo, ponieważ priorytet operatora \geq jest niższy niż większości takich operatorów, jak $+$, $/$, $-$, a nawet bitowych operatorów przesunięcia. Tak więc łatwo jest nabrac przekonania, że makroinstrukcja działa prawidłowo w przypadku wszystkich wyrażeń, w tym w razie użycia bitowych operatorów logicznych.

Poprzedni problem można rozwiązać, zachowując ostrożność podczas programowania — należy umieścić w nawiasach wszystko to, co znajduje się w makroinstrukcji. Jednak drugi problem jest bardziej subtelny. W odróżnieniu od zwykłej funkcji, podczas każdego użycia argumentu w makroinstrukcji obliczana jest jego wartość. Dopóki makroinstrukcja jest wywoływana ze zwykłymi zmiennymi, nie jest to groźne, lecz gdy argument posiada skutki uboczne, to rezultat okaże się zaskakujący i zupełnie nie przypomina zachowania funkcji.

Na przykład poniższa makroinstrukcja sprawdza, czy jej argument jest zawarty w pewnym zakresie wartości:

```
#define BANO(x) (((x)>5 && (x)<10) ? (x) : 0)
```

Dopóki używane są „zwykłe” argumenty, makroinstrukcja działa jak prawdziwa funkcja. Kiedy jednak stracisz czujność i uwierzysz, że to jest prawdziwa funkcja, zacząć się kłopoty. Na przykład:

```
//: C09:MacroSideEffects.cpp
#include "../require.h"
#include <fstream>
using namespace std;

#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)

int main() {
    ofstream out("macro.out");
    assure(out, "macro.out");
    for(int i = 4; i < 11; i++) {
        int a = i;
        out << "a = " << a << endl << '\t';
        out << "BAND(++a)=" << BAND(++a) << endl;
        out << "\t a = " << a << endl;
    }
} //:~
```

Zwróć uwagę na użycie wielkich liter w nazwie makroinstrukcji. To dobry zwyczaj, ponieważ dzięki temu czytelnik widzi, że ma do czynienia z makroinstrukcją, a nie z funkcją, więc w przypadku problemów będzie mu łatwiej przypomnieć sobie o tym.

A oto wyniki działania programu, znacznie odbiegające od tego, czego można by się spodziewać po prawdziwej funkcji:

```
a = 4
BAND(++a)=0
a - 5
a - 5
BAND(++a)=8
```

```

    a = 8
a - 6
BAND(++a)=9
    a = 9
a - 7
BAND(++a)=10
    a - 10
a = 8
BAND(++a)=0
    a = 10
a - 9
BAND(++a)=0
    a = 11
a - 10
BAND(++a)=0
    a = 12

```

Kiedy wartość zmiennej a wynosi cztery, wykonywana jest tylko pierwsza część wyrażenia warunkowego. A zatem wyrażenie jest obliczane tylko jeden raz, a w wyniku skutków ubocznych wywołania makroinstrukcji zmienna a uzyskuje wartość pięć. Zachowania takiego można by się spodziewać po wywołaniu w tej samej sytuacji normalnej funkcji. Jednakże gdy wartość zmiennej znajduje się w przedziale środkowym, sprawdzane są oba warunki, w wyniku czego dwukrotnie wykonywana jest jej inkrementacja. W czasie wyznaczania wyniku argument obliczany jest ponownie, czego skutkiem jest trzecia inkrementacja. Gdy wartość argumentu przekracza górny zakres, nadal sprawdzane są oba warunki, więc wynikiem tego jest dwukrotne wykonanie inkrementacji. Skutki uboczne mogą więc być różne, w zależności od wartości argumentu.

Z pewnością nie jest to sposób zachowania oczekiwany od makroinstrukcji przypominającej wywołanie funkcji. W takim przypadku oczywistym rozwiązaniem jest utworzenie prawdziwej funkcji, która będzie wiązała się z dodatkowym narzutem i która może zmniejszyć efektywność całego programu, 'W przypadku gdy będzie wielokrotnie wywoływana. Niestety, problem ten nie zawsze jest równie oczywisty; możesz nieświadomie wejść w posiadanie biblioteki zawierającej wymieszane ze sobą funkcje i makroinstrukcje, więc problem, podobny do opisanego, może kryć w sobie trudne do wykrycia błędy. Na przykład makroinstrukcja **putc()**, zawarta w **cstdio**, może dwukrotnie wyznaczyć wartość swojego drugiego argumentu. Zostało to wyszczególnione w standardzie języka C. Również nieuwaga implementacja **toupper()** jako makroinstrukcji może wyliczyć argument więcej niż jednokrotnie, co da nieprzewidziane wyniki wywołania tej makroinstrukcji w postaci **toupper(*p++)**¹.

Makroinstrukcje a dostęp

Oczywiście, uważne programowanie i używanie makroinstrukcji preprocesora jest konieczne w języku C i z pewnością moglibyśmy poradzić sobie stosując te same metody w języku C++, gdyby nie pewien problem — makrodefinicja nie zawiera pojęcia zasięgu, wymaganego w przypadku funkcji składowych. Preprocesor dokonuje po prostu podstawień tekstu, nie można więc na przykład napisać:

Andrew Koenig opisuje to bardziej szczegółowo w swojej książce *C Traps & Pitfalls*(Addison-Wesley, 1989).

```
class X {  
    int i;  
public:  
#define VAL(X)::i) // Błąd
```

ani nawet wykonać podobnego zapisu. Ponadto nie można by wskazać, do którego obiektu następuje odwołanie. Nie ma bowiem sposobu na to, by w makroinstrukcji określić zasięg klasy. Ponieważ programiści nie dysponują żadnym rozwiązańem alternatywnym w stosunku do makroinstrukcji preprocesora, w imię efektywności będą zmuszeni do określenia niektórych danych składowych jako publiczne, eksponując w ten sposób wewnętrzną implementację i uniemożliwiając jej zmiany, a także rezygnując z ochrony, zapewianej przez specyfikator **private**.

Funkcje inline

Zgodnie z przyjętym w języku C++ rozwiązańem problemu dostępu makroinstrukcji do prywatnych składowych klasy usunięte zostały *wszystkie* problemy związane z makroinstrukcjami preprocesora. Zostało to osiągnięte przez umieszczenie pojęcia makroinstrukcji pod kontrolą kompilatora, gdzie w istocie jest jego miejsce. Język C++ implementuje makroinstrukcję jako *funkcję inline*, będącą pod każdym względem prawdziwą funkcją. Działa ona dokładnie tak, jak można tego oczekiwać po zwykłych funkcjach. Jedyna różnica polega na tym, że funkcje inline są rozwijane w miejscach ich wywołania, podobnie jak makroinstrukcje preprocesora; eliminowany jest więc nárazut związany z wywołaniem funkcji. Tak więc nie powinno się (niemal) nigdy używać makroinstrukcji, a zamiast nich należy stosować wyłącznie funkcje inline.

Każda funkcja, definiowana w ciele klasy, jest automatycznie funkcją inline, lecz można uczynić funkcjami inline również funkcje niebędące składowymi klas, poprzezając je słowem kluczowym **inline**. Aby wywołało to jednak jakkolwiek skutek, trzeba dołączyć do deklaracji ciało (treść) funkcji, bo w przeciwnym razie zostanie to przez kompilator potraktowane tak, jak zwykła deklaracja funkcji. Tak więc zapis:

```
inline int plusOne(int x);
```

nie wywołuje żadnego skutku, poza tym, że deklaruje funkcję (która później może, ale nie musi, posiadać definicji inline). Właściwym rozwiązańiem byłoby określenie ciała funkcji:

```
inline int plusOne(int x) { return ++x; }
```

Zwrót uwagę na to, że kompilator sprawdzi (**jak** zawsze to robi) poprawne wykorzystanie listy argumentów i zwracanej wartości (dokonując wszystkich niezbędnych konwersji), czego preprocesor w ogóle nie jest w stanie dokonać. Również próba zapisania powyższej funkcji w postaci makroinstrukcji preprocesora spowodowałaby niepożądany skutek uboczny.

Definicje inline prawie zawsze muszą być umieszczane w plikach nagłówkowych. Gdy kompilator napotka taką definicję, umieszcza w tablicy symboli typ funkcji (**jej** sygnaturę, połączoną ze zwracaną wartością) oraz ciało funkcji. Kiedy funkcja taka

jest używana, kompilator upewnia się, że jej wywołanie jest poprawne, a zwracana wartość wykorzystywana prawidłowo. Następnie podstawią w miejsce wywołania treść funkcji, eliminując w taki sposób narzut. Kod funkcji inline zajmuje pamięć, ale w przypadku gdy funkcja jest niewielka, to w rzeczywistości może zająć mniej miejsca niż kod wykonujący zwykłe wywołanie funkcji (umieszczający argumenty na stosie i wykonujący instrukcję CALL).

Funkcja inline zawarta w pliku nagłówkowym posiada szczególny status. Plik nagłówkowy zawierający funkcję *oraz jej definicję* musi być bowiem dołączony w każdym pliku, w którym ta funkcja jest **wykorzystywana**. Nie wywołuje to jednak błędu wynikającego z wielokrotnej definicji funkcji (chociaż jej definicja musi być identyczna we wszystkich miejscach, w których dołączona jest funkcja inline).

Funkcje inline wewnętrz klas

Aby zdefiniować funkcję inline, trzeba jedynie umieścić na końcu jej **definicji** słowo kluczowe **inline**. Nie jest to jednak konieczne w obrębie definicji klasy. **Każda** funkcja definiowana wewnętrz definicji klasy staje się automatycznie funkcją inline. Na przykład:

```
//: C09:inline.cpp
// Funkcje inline wewnętrz klas
#include <iostream>
#include <string>
using namespace std;

class Point {
    int i, j, k;
public:
    Point(): i(0), j(0), k(0) {}
    Point(int ii, int jj, int kk)
        : i(ii), j(jj), k(kk) {}
    void print(const string& msg = "") const {
        if(msg.size() != 0) cout << msg << endl;
        cout << "i - " << i << ", "
            << "j - " << j << ", "
            << "k - " << k << endl;
    }
};

int main() {
    Point p, q(1,2,3);
    p.print("wartosc p");
    q.print("wartosc q");
} //:-
```

W tym przypadku zarówno konstruktor, jak i funkcja **print()** są domyślnie funkcjami inline. Zwróć uwagę na to, że w funkcji **main()** nie ma śladu używania funkcji inline, i tak właśnie powinno być. Logiczne działanie funkcji musi być takie samo, niezależnie od tego, czy jest ona funkcją inline, czy też nie (w przeciwnym razie oznaczałoby to, że **uszkodzony jest kompilator**). Jedyna widoczna różnica polega na wydajności.

Oczywiście, kuszące jest używanie funkcji inline w każdym miejscu deklaracji **klasy**, ponieważ pozwalają one na uniknięcie dodatkowego kroku, polegającego na utworzeniu zewnętrznej definicji funkcji **składowej**. Trzeba jednak pamiętać, że ideą funkcji inline jest zapewnienie kompilatorowi większych możliwości dokonania optymalizacji. Jednak definiowanie dużych funkcji jako funkcji inline spowoduje, że kod będzie powtarzany w każdym miejscu, w którym ta funkcja jest wywoływana. Wywoła to rozdziele kodu, niewspółmierne do korzyści wynikających ze zwiększenia jego szybkości (**jedynym** niezawodnym sposobem przekonania się o tym jest przeprowadzenie eksperymentów, pozwalających na sprawdzenie wyników uzyskiwanych dzięki wykorzystaniu funkcji inline za pomocą używanego przez siebie kompilatora).

Funkcje udostępniające

Jednym z najważniejszych zastosowań funkcji inline w klasach są **funkcje udostępniające**. Te niewielkie funkcje pozwalają na odczyt lub zmianę stanu części obiektu — to znaczy wewnętrznej zmiennej (lub zmiennych) składowej. Powód, dla którego funkcje inline są takie ważne dla funkcji udostępniających, można poznać analizując poniższy przykład:

```
//: C09:Access.Cpp
// Funkcje inline jako funkcje udostępniające

class Access {
    int i;
public:
    int read() const { return i; }
    void set(int ii) { i = ii; }
};

int main() {
    Access A;
    A.set(100);
    int x = A.read();
} //:-~
```

W powyższym przykładzie użytkownik klasy nie ma nigdy bezpośredniego kontaktu ze zmiennymi stanu, zawartymi w klasie, i mogą one pozostać prywatne, znajdująając się pod kontrolą projektanta klasy. Każdy dostęp do prywatnych danych składowych może być nadzorowany za pomocą interfejsu funkcji składowych. Ponadto dostęp ten jest niezwykle efektywny. Przyjrzyjmy się, na przykład, funkcji **read()**. Gdyby nie została użyta funkcja inline, kod wygenerowany dla funkcji **read()** obejmowałby umieszczenie wskaźnika **this** na stosie i wywołanie instrukcji CALL asemblera. W przypadku większości maszyn kod taki zajmowałby więcej miejsca niż kod generowany dla funkcji inline, a czas jego wykonania z pewnością byłby większy.

Gdyby nie było funkcji inline, to zwracający uwagę na efektywność projektant klasy byłby zmuszony do uczynienia zmiennej i składową publiczną. Dzięki temu wyeliminowałby narzut, pozwalając użytkownikowi na bezpośredni dostęp do tej zmiennej. Z punktu widzenia projektu wywołaloby to katastrofalne skutki, ponieważ składowa i staje się w ten sposób elementem publicznego interfejsu, co oznacza, że projektant

klasy nie będzie mógł jej już nigdy zmienić. Zostanie więc niejako „skazany” na **zmienną całkowitą** o nazwie `i`. Stanowi to problem, bowiem później może się okazać, że znacznie lepiej reprezentować informacje dotyczące stanu jako liczbę typu **float**, a nie **int**, lecz z uwagi na to, że deklaracja zmiennej `i` jako całkowitej stanowi element publicznego interfejsu klasy, nie sposobjuje zmienić. Może się również zdarzyć, że odczytując lub zapisując wartość zmiennej `i` trzeba będzie przeprowadzić jakieś dodatkowe obliczenia, co nie będzie możliwe w przypadku, gdy zmienna ta będzie zmienną publiczną. Z drugiej strony — jeżeli do odczytu i zapisu informacji o stanie obiektu zawsze używane są funkcje składowe, to można dowolnie zmieniać jego wewnętrzną reprezentację.

Ponadto wykorzystanie funkcji składowych do nadzoru nad zmiennymi składowymi pozwala na dodanie do funkcji składowych kodu wykrywającego zmiany danych, który może okazać się bardzo pomocny w czasie uruchamiania programu. Natomiast gdy składowa jest publiczna, każdy może ją zmienić w dowolnej chwili, nie informując o tym nikogo.

Obserwatory i modyfikatory

Niektórzy dokonują dalszego podziału funkcji udostępniających na **obserwatory** (odczytujące informacje o stanie obiektu, zwane również **akcesorami**) i **modyfikatory** (zmieniające stan obiektu). Ponadto w celu umożliwienia użycia tej samej nazwy funkcji zarówno w stosunku do obserwatora, jak i modyfikatora można zastosować przeciążenie nazwy — sposób wywołania funkcji zadecyduje o tym, czy informacja o stanie jest **odczytywana**, czy też modyfikowana. Ilustruje to poniższy przykład:

```
//: C09:Rectangle.cpp
// Obserwatory i modyfikatory

class Rectangle {
    int wide, high;
public:
    Rectangle(int w = 0, int h = 0)
        : wide(w), high(h) {}
    int width() const { return wide; } // Odczyt
    void width(int w) { wide = w; } // Zapis
    int height() const { return high; } // Odczyt
    void height(int h) { high = h; } // Zapis
};

int main() {
    Rectangle r(19, 47);
    // Zmiana szerokości (width) i wysokości (height):
    r.height(2 * r.width());
    r.width(2 * r.height());
} //:~
```

Konstruktor wykorzystuje listę inicjalizatorów konstruktora (przedstawioną krótko w rozdziale 8. i opisaną wyczerpująco w rozdziale 14.) do inicjalizacji wartości składowych **wide** i **high** (używając **pseudokonstruktorów** w stosunku do typów wbudowanych).

Funkcje składowe nie mogą używać tych samych nazw, co dane składowe, może więc zainstnieć konieczność ich odróżnienia. Należy poprzedzić ich nazwy znakami podkreślenia. Jednak identyfikatory poprzedzone znakami podkreślenia są zastrzeżone, więc nie należy ich używać.

Zamiast tego można użyć przedrostków „get” (pobierz) i „set” (ustaw) do oznaczenia obserwatorów i modyfikatorów:

```
//: C09:Rectangle2.cpp
// Obserwatory i modyfikatory z przedrostkami "get" i "set"

class Rectangle {
    int width, height;
public:
    Rectangle(int w = 0, int h = 0)
        : width(w), height(h) {}
    int getWidth() const { return width; }
    void setWidth(int w) { width = w; }
    int getHeight() const { return height; }
    void setHeight(int h) { height = h; }
};

int main() {
    Rectangle r(19, 47);
    // Change width & height:
    r.setHeight(2 * r.getWidth());
    r.setWidth(2 * r.getHeight());
} ///:-
```

Oczywiście, obserwatory i modyfikatory nie muszą być zwykłymi „pasami transmisyjnymi” zmiennych wewnętrznych. Czasami mogą one wykonywać bardziej skomplikowane obliczenia. W poniższym przykładzie wykorzystano do utworzenia prostej klasy **Time** funkcje związane z czasem, zawarte w standardowej bibliotece języka C:

```
//: C09:Cptime.h
// Prosta klasa, reprezentująca czas
#ifndef CPPTIME_H
#define CPPTIME_H
#include <ctime>
#include <cstring>

class Time {
    std::time_t t;
    std::tm local;
    char asciiRep[26];
    unsigned char lflag, aflag;
    void updateLocal() {
        if(!lflag) {
            local = *std::localtime(&t);
            lflag++;
        }
    }
    void updateAscii() {
        if(!aflag) {
            updateLocal();
            std::strcpy(asciiRep, std::asctime(&local));
            aflag++;
        }
    }
};
```

```
    }
public:
    Time() { mark(); }
    void mark() {
        lflag = aflag = 0;
        std::time(&t);
    }
    const char* ascii() {
        updateAscii();
        return asciiRep;
    }
    // Różnica w sekundach:
    int delta(Time* dt) const {
        return int(std::difftime(t, dt->t));
    }
    int daylightSavings() {
        updateLocal();
        return local.tm_isdst;
    }
    int dayOfYear() { // Liczba dni od 1 stycznia
        updateLocal();
        return local.tm_yday;
    }
    int dayOfWeek() { // Liczba dni od niedzieli
        updateLocal();
        return local.tm_wday;
    }
    int since1900() { // Liczba lat od 1900
        updateLocal();
        return local.tm_year;
    }
    int month() { // Liczba miesięcy od stycznia
        updateLocal();
        return local.tm_mon;
    }
    int dayOfMonth() {
        updateLocal();
        return local.tm_mday;
    }
    int hour() { // Liczba godzin od północy (zegar 24-godzinny)
        updateLocal();
        return local.tm_hour;
    }
    int minute() {
        updateLocal();
        return local.tm_min;
    }
    int second() {
        updateLocal();
        return local.tm_sec;
    }
};

#endif // CPPTIME_H ///:~
```

Funkcje zawarte w standardowej bibliotece języka C zawierają wiele reprezentacji czasu, z których każda jest elementem klasy **Time**. Nie ma jednak potrzeby aktualizowania

ich wszystkich. Dlatego też zmienna t typu **t_time** jest wykorzystywana jako reprezentacja podstawowa, natomiast zmienne **local** typu **tm** oraz znakowa reprezentacja ASCII **asciiRep** posiadają znaczniki informujące o tym, czy były one aktualizowane zgodnie z bieżącą wartością zmiennej t. Dwie funkcje prywatne **updateLocal()** oraz **updateAscii()** sprawdzają wartości tych znaczników i warunkowo dokonują aktualizacji.

Konstruktor wywołuje funkcję **mark()** (może ją również wywołać użytkownik, wymuszając, by obiekt reprezentował bieżący czas), która zeruje oba znaczniki w celu oznaczenia, że wartości zmiennych **local** oraz **asciiRep** są nieaktualne. Funkcja **ascii()** wywołuje funkcję **updateAscii()**, która kopiuje do lokalnego bufora wynik zwracany przez funkcję **asctime()**, pochodząą ze standardowej biblioteki języka C. Funkcja **asctime()** wykorzystuje bowiem statyczny obszar danych, zapisywany w przypadku, gdy jest ona wywoływana w jakimś innym miejscu. Wartość zwracana przez funkcję **ascii()** jest adresem tego lokalnego bufora.

Wszystkie funkcje, poczynając od **daylightSavings()**, wykorzystują funkcję **updateLocal()**, co powoduje, że funkcje powstałe w wyniku ich rozwinięcia mogą być stosunkowo duże. Wydaje się to niepotrzebne, zwłaszcza gdy weźmie się pod uwagę, że prawdopodobnie nie będą one wywoływanie zbyt często. Jednak nie oznacza to, że wszystkie te funkcje inline powinno się zamienić w zwykłe funkcje. Jeśli dokonasz takiego przekształcenia, to pozostaw jako inline przynajmniej funkcję **updateLocal()**. Dzięki temu jej kod zostanie powtórzony w zwykłych funkcjach składowych, eliminując dodatkową nazwę związanej z jej wywołaniem.

Poniżej znajduje się krótki program testowy:

```
//: C09:Cptime.cpp
// Test prostej klasy, reprezentującej czas
#include "Cptime.h"
#include <iostream>
using namespace std;

int main() {
    Time start;
    for(int i = 1; i < 1000; i++) {
        cout << i << ' ';
        if(i%10 == 0) cout << endl;
    }
    Time end;
    cout << endl;
    cout << "początek = " << start.ascii();
    cout << "koniec = " << end.ascii();
    cout << "roznica - " << end.delta(&start);
} //:-
```

Tworzony jest obiekt klasy **Time**, następnie wykonywane są jakieś operacje, trwające przez pewien czas, po czym jest generowany drugi obiekt klasy **Time**, oznaczający czas zakończenia operacji. Oba obiekty zostały użyte do wyświetlenia czasów rozpoczęcia i zakończenia operacji oraz czasu, który pomiędzy nimi upłynął.

Klasy Stash i Stack z funkcjami inline

Dzięki funkcji inline możemy obecnie w taki sposób przekształcić klasy **Stash** i **Stack**, aby stały się one bardziej efektywne:

```
//: C09:Stash4.h
// Funkcje inline
#ifndef STASH4_H
#define STASH4_H
#include "../require.h"

class Stash {
    int size;           // Wielkość każdego elementu
    int quantity;      // Liczba elementów pamięci
    int next;          // Następny pusty element
    // Dynamicznie przydzielana tablica bajtów:
    unsigned char* storage;
    void inflate(int increase);

public:
    Stash(int sz) : size(sz), quantity(0),
                    next(0), storage(0) {}
    Stash(int sz, int initQuantity) : size(sz),
                                      quantity(0), next(0), storage(0) {
        inflate(initQuantity);
    }
    Stash::~Stash() {
        if(storage != 0)
            delete []storage;
    }
    int add(void* element);
    void* fetch(int index) const {
        require(0 < index, "Stash::fetch indeks ma wartość ujemną");
        if(index >= next)
            return 0; // Oznaczenie końca
        // Tworzenie wskaźnika do żądanego elementu:
        return &(storage[index * size]);
    }
    int count() const { return next; }
};

#endif // STASH4_H ///:-
```

Oczywiście, niewielkie funkcje będą działać dobrzejako funkcje inline. Zwróć jednak uwagę na to, że dwie największe funkcje pozostały zwykłymi funkcjami, ponieważ przekształcenie ich w funkcje inline nie miałoby prawdopodobnie żadnego wpływu na zwiększenie wydajności:

```
//: C09:Stash4.cpp {0}
#include "Stash4.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

int Stash::add(void* element) {
    if(next >= quantity) // Czy wystarczy pamięci?
```

```

    inflate(increment);
    // Kopiowanie elementu do pamięci.
    // począwszy od następnego wolnego miejsca:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return(next - 1); // Numer indeksu
}

void Stash::inflate(int increase) {
    assert(increase >= 0);
    if(increase == 0) return;
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Kopiowanie starego obszaru do nowego
    delete [] (storage); // Zwolnienie starego obszaru pamięci
    storage = b; // Wskaźnik do nowego obszaru
    quantity = newQuantity; // Aktualizacja liczby elementów
} //:-

```

Program testujący ponownie sprawdza, czy wszystko działa poprawnie:

```

//: C09:Stash4Test.cpp
//{L} Stash4
#include "Stash4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
    Stash intStash(sizeof(int));
    for(int i = 0; i < 100; i++)
        intStash.add(&i);
    for(int j = 0; j < intStash.count(); j++)
        cout << "intStash.fetch(" << j << ") = "
            << *(int*)intStash.fetch(j)
            << endl;
    const int bufsize = 80;
    Stash stringStash(sizeof(char) * bufsize, 100);
    ifstream in("Stash4Test.cpp");
    assure(in, "Stash4Test.cpp");
    string line;
    while(getline(in, line))
        stringStash.add((char*)line.c_str());
    int k = 0;
    char* cp;
    while((cp = (char*)stringStash.fetch(k++))!=0)
        cout << "stringStash.fetch(" << k << ") = "
            << cp << endl;
} //:-

```

Jest to ten sam program testowy, którego użyto ostatnim razem, więc wyniki jego działania powinny być zasadniczo identyczne.

Klasa **Stack** wykorzystuje funkcje inline w jedyne lepszy sposób:

```
//: C09:Stack4.h
// Z funkcjami inline
#ifndef STACK4_H
#define STACK4_H
#include "../require.h"

class Stack {
    struct Link {
        void* data;
        Link* next;
        Link(void* dat, Link* nxt):
            data(dat), next(nxt) {}
    }* head;
public:
    Stack() : head(0) {}
    ~Stack() {
        require(head == 0, "Stos nie jest pusty");
    }
    void push(void* dat) {
        head = new Link(dat, head);
    }
    void* peek() const {
        return head ? head->data : 0;
    }
    void* pop() {
        if(head == 0) return 0;
        void* result = head->data;
        Link* oldHead = head;
        head = head->next;
        delete oldHead;
        return result;
    }
};

#endif // STACK4_H III-
```

Warto zwrócić uwagę na to, że został usunięty destruktor struktury **Link**, który istniał w poprzedniej wersji klasy **Stack**, ale był funkcją pustą. Wyrażenie **delete oldHead**, zawarte w funkcji **pop()**, zwalnia pamięć zajmowaną przez obiekt typu **Link** (nie niszczy ono natomiast wskazywanego przez niego obiektu **data**).

Większość funkcji została określona jako funkcje inline w dość elegancki i oczywisty sposób, szczególnie w przypadku struktury **Link**. Nawet co do funkcji **pop()** wydaje się to usprawiedliwione, mimo że za każdym razem, gdy w funkcji występują wyrażenia warunkowe lub zmienne lokalne, nie wiadomo, czy zastosowanie funkcji inline będzie tak korzystne. W tym przypadku funkcja jest na tyle mała, że prawdopodobnie nie stanowi to żadnej przeszkody.

Gdy wszystkie funkcje są funkcjami inline, wykorzystywanie biblioteki staje się dość proste, ponieważ nie wymaga ono łączenia. Ilustruje to przykładowy program testowy (zwróć uwagę na to, że nie istnieje plik **Stack4.cpp**):

```
//: C09:Stack4Test.cpp
//{T} Stack4Test.cpp
#include "Stack4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // Argumentem jest nazwa pliku
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack textlines;
    string line;
    // Odczytanie pliku i zapamiętanie wierszy na stosie:
    while(getline(in, line))
        textlines.push(new string(line));
    // Pobranie wierszy ze stosu i wydrukowanie ich:
    string* s;
    while((s = (string*)textlines.pop()) != 0) {
        cout << *s << endl;
        delete s;
    }
}
```

Czasami tworzy się klasy, w których wszystkie funkcje są funkcjami inline, dzięki czemu klasy zawarte są w całości w plikach nagłówkowych (podczas lektury książki przekonasz się, że także jej autor niekiedy przekracza tę granicę). Podczas tworzenia programu praktyka taka nie powoduje na ogół większych szkód, chociaż czasami prowadzi do wydłużenia czasu komplikacji. Kiedy program już nieco „okrzepnie”, warto powrócić i zamienić funkcje inline w zwykłe funkcje — wszędzie tam, gdzie jest to uzasadnione.

Funkcje inline a kompilator

Aby rozumieć, w jakich przypadkach korzystniejsze jest wykorzystywanie funkcji inline, warto wiedzieć, jak działa kompilator, napotykając taką funkcję. Podobnie jak w przypadku każdej innej funkcji, kompilator zapamiętuje w tablicy symboli *typ* funkcji (to znaczy jej prototyp, uwzględniający jej nazwę i typy argumentów, połączone z wartością zwracaną przez funkcję). Ponadto gdy kompilator sprawdzi, że typ funkcji *oraz* jej ciało nie zawierają błędów składniowych, kod zawarty w tym ciele również jest umieszczany w tablicy symboli. To, czy jest on zapisywany jako kod źródłowy, skompilowane instrukcje asemblera, czy jeszcze w jakiejś innej postaci, zależy od kompilatora.

W przypadku wywołania funkcji inline kompilator upewnia się najpierw, że można ją poprawnie wykonać. Oznacza to, że typy wszystkich argumentów muszą albo dokładnie odpowiadać typom znajdującym się na liście argumentów funkcji, albo kompilator musi być w stanie przeprowadzić ich konwersję do odpowiednich typów, a także, że wartość zwracana przez funkcję musi mieć właściwy typ (albo musi istnieć

możliwość jej konwersji do właściwego typu) w wyrażeniu, w którym się znajduje. Oczywiście, kompilator robi dokładnie to samo w przypadku każdej funkcji i zasadniczo różni się to od tego, co wykonuje preprocesor, który nie jest w stanie kontrolować typów ani dokonywać konwersji.

Jeżeli wszystkie informacje dotyczące typu funkcji odpowiadają kontekstowi, w którym została wywołana, wywołanie funkcji jest bezpośrednio zastępowane jej kodem, dzięki czemu wyeliminowany zostanie narzut, związany z wywołaniem funkcji i możliwe będzie przeprowadzenie przez kompilator dalszej optymalizacji. W przypadku gdy funkcja inline jest **funkcją składową**, w odpowiednim miejscu umieszczany jest również adres obiektu (this), czego, oczywiście, również nie potrafi zrobić preprocesor.

Ograniczenia

Istnieją dwie sytuacje, w których kompilator nie może dokonać rozwinięcia funkcji. W przypadkach tych powraca on do normalnej postaci funkcji, pobierając definicję funkcji inline i rezerwując dla niej pamięć, tak jak w przypadku funkcji **niedbących** funkcjami inline. Gdy musi zrobić to w wielu jednostkach translacji (co zwykle spowodowałoby zgłoszenie błędu wielokrotnej definicji), informuje program łączący o tym, że powinien zignorować wielokrotne definicje.

Kompilator nie potrafi rozwinać funkcji w przypadku, gdy jest ona zbyt złożona. Jest to uzależnione od konkretnego kompilatora, ale w sytuacjach, w której poddaje się większość kompilatorów, rozwinięcie funkcji prawdopodobnie i tak nie spowodowałoby żadnego zwiększenia efektywności. Na ogół za zbyt skomplikowane, by je rozwijać, uważa się sąsiedzkie rodzaje pętli i, jeśli się nad tym zastanowić, wykonywanie pętli zajmuje prawdopodobnie znacznie więcej czasu niż narzut związany z wywołaniem funkcji. Jeżeli funkcja jest zbiorem prostych instrukcji, to prawdopodobnie kompilator nie napotka żadnych trudności z jej rozwinięciem. Jednakże jeśli tych instrukcji będzie bardzo wiele, narzut związany z wywołaniem funkcji będzie znacznie mniejszy niż koszt wykonania jej ciała. Trzeba również pamiętać, że ilekroć wywoływana jest duża funkcja inline, cała jej treść jest wstawiana w miejsce każdego wywołania funkcji. Z tego powodu łatwo uzyskać nadmiernie rozdzielony kod, nie uzyskując w zamian żadnej zauważalnej poprawy jego wydajności (niektóre z przykładów, zawartych w tej książce, mogą zawierać funkcje inline o wielkości przekraczającej rozsądne rozmiary, przygotowane w tej postaci w celu ograniczenia długości wydruków).

Kompilator nie może również przeprowadzić rozwijania funkcji w przypadku gdy bezpośrednio lub pośrednio pobierany jest jej adres. Jeśli kompilator musi określić adres funkcji, przydziela pamięć, przeznaczoną na jej kod, i wykorzystuje uzyskany w ten sposób adres. Jednak w miejscach, w których adres funkcji nie jest używany, kompilator będzie prawdopodobnie nadal wstawiał jej rozwinięty kod.

Warto podkreślić, że słowo kluczowe **inline** stanowi tylko wskazówkę dla kompilatora — nie jest on w ogóle zmuszony do rozwijania czegokolwiek. Dobry kompilator będzie rozwijał niewielkie, proste funkcje, w inteligentny sposób ignorując te funkcje inline, które są **zbyt** skomplikowane. Zapewni ci to pożądany rezultat — semantykę rzeczywistego wywołania funkcji, połączoną z efektywnością makroinstrukcji.

Odwołania do przodu

Jeśli wyobrazić sobie, co robi kompilator, realizując funkcje inline, można nabrac przekonania, że istnieje tu więcej ograniczeń niż w rzeczywistości. W szczególności może się wydawać, że kompilator nie jest w stanie obsłużyć funkcji inline, odwołującej się „do przodu” — do funkcji, która nie została jeszcze zadeklarowana w obrębie klasy (niezależnie od tego, czy jest ona funkcją inline, czy też nie):

```
//: C09:EvaluationOrder.cpp
// Kolejność wyliczania funkcji inline

class Forward {
    int i;
public:
    Forward() : i(0) {}
    // Wywołanie niewydeklarowanej funkcji:
    int f() const { return g() + 1; }
    int g() const { return i; }
};

int main() {
    Forward fwd;
    fwd.f();
} //:-
```

W funkcji **f()** występuje wywołanie funkcji **g()**, mimo że funkcja **g()** nie została jeszcze zadeklarowana. Funkcjonuje to prawidłowo, ponieważ definicja języka określa, że żadne funkcje inline nie są wyliczane, dopóki nie napotka się nawiasu klamrowego, zamykającego deklarację klasy.

Oczywiście, gdyby funkcja **g()** wywoływała z kolei funkcję **f()**, skończyłoby się to na szeregu wywołań rekurencyjnych, które byłyby dla kompilatora zbyt trudne do rozwiniecia w postaci funkcji inline (należałyby również wstawić sprawdzanie jakich warunków w funkcji **f()** lub **g()**, by umożliwić „wycofanie się” z rekurencji, który w przeciwnym wypadku nie miałaby końca).

Działania ukryte w konstruktorach i destruktورach

Konstruktory i destruktory są dwoma miejscami, co do których można nabrac błędnego mniemania, że wykorzystanie funkcji inline jest bardziej efektywne niż w rzeczywistości. Konstruktory i destruktory mogą wykonywać ukryte działania. Wynikają one z tego, że klasa może zawierać obiekty podrzędne, wymagające wywołania swoich konstruktorów i destruktatorów. Obiekty podrzędne te mogą być obiektami składowymi albo mogą one istnieć dzięki dziedziczeniu (opisanemu w rozdziale 14.). Przykładem niech będzie klasa zawierająca obiekty składowe:

```
//: C09:Hidden.cpp
// Ukryte działania w funkcjach inline
#include <iostream>
using namespace std;
```

```

class Member {
    int i, j, k;
public:
    Member(int x = 0) : i(x), j(x), k(x) {}
    ~Member() { cout << "~Member" << endl; }
};

class WithMembers {
    Member q, r, s; // Posiadają konstruktory
    int i;
public:
    WithMembers(int ii) : i(ii) {} // Banalne?
    ~WithMembers() {
        cout << "~WithMembers" << endl;
    }
};

int main() {
    WithMembers w(1);
} III-

```

Konstruktor klasy **Member** jest wystarczająco prosty, by być funkcją inline, ponieważ nie zachodzi w nim nic szczególnego — ani dziedziczenie, ani obiekty składowe nie powodują żadnych dodatkowych działań. Jednak w klasie **WithMembers** dzieje się więcej niż tylko to, co widać. Konstruktory i destruktory obiektów składowych q, r i s są wywoływane automatycznie; są one również funkcjami inline, więc różnią się znacznie od normalnych funkcji składowych. Nie oznacza to wcale, że konstruktorów i destruktorek nie można nigdy definiować jako funkcji inline — są przypadki, w których jest to uzasadnione. Gdy tworzy się wstępny projekt programu, szybko pisząc kod, używanie funkcji inline jest często wygodniejsze. Warto się im jednak bliżej przyjrzeć, kiedy dąży się do efektywności.

Walka z bałaganem

W książce takiej, jak niniejsza, prostota i zwięzłość, wynikające z umieszczenia funkcji inline w klasach, są bardzo użyteczne, gdyż dzięki nim więcej tekstu programu mieści się na stronie lub ekranie (w przypadku seminariów). Jednak, jak wykazał Dan Saks², w prawdziwych projektach wywołuje to w efekcie niepotrzebny bałagan w interfejsie klasy, czyniąc ją zarazem trudniejszą w użyciu. Odwołuje się on do funkcji składowych definiowanych wewnątrz klas, używając łacińskiego terminu *in situ* (w miejscu). Saks twierdzi, że dla zachowania przejrzystości interfejsu wszystkie definicje powinny być umieszczone na zewnątrz klasy. Optymalizacja jest natomiast, jak utrzymuje, odrębną kwestią. Jeżeli jest potrzebna, należy użyć słowa kluczowego **inline**. W przypadku zastosowania takiego ujęcia przedstawiony wcześniej program **Rectangle.cpp** miałaby następującą postać:

```

//: C09:Noinsitu.cpp
// Usuwanie funkcji deklarowanych in situ

```

² Współautor — wraz z Tomem Plumem — książki *C++ Programming Guidelines* (Plum Hall, 1991).

```
class Rectangle {
    int width, height;
public:
    Rectangle(int w = 0, int h = 0);
    int getWidth() const;
    void setWidth(int w);
    int getHeight() const;
    void setHeight(int h);
};

inline Rectangle::Rectangle(int w, int h)
: width(w), height(h) {}

inline int Rectangle::getWidth() const {
    return width;
}

inline void Rectangle::setWidth(int w) {
    width = w;
}

inline int Rectangle::getHeight() const {
    return height;
}

inline void Rectangle::setHeight(int h) {
    height = h;
}

int main() {
    Rectangle r(19, 47);
    // Transpose width & height:
    int iHeight = r.getHeight();
    r.setHeight(r.getWidth());
    r.setWidth(iHeight);
} //
```

Obecnie, chcąc porównać działanie funkcji inline z działaniem zwyczajnych funkcji wystarczy jedynie usunąć słowo kluczowe **inline** (funkcje inline powinny być jednak umieszczone w plikach nagłówkowych, podczas gdy zwyczajne funkcje muszą znajdować się w swoich jednostkach translacji). Jeżeli chcesz ulokować te funkcje w dokumentacji, wystarczy prosta operacja typu „wytnij i wklej”. Funkcje *in situ wymagają* więcej pracy i łatwiej jest w ich przypadku o błędę. Innym argumentem przemawiającym za takim podejściem jest to, że dzięki niemu można zawsze zachować jednolity styl formatowania definicji funkcji, co nie zawsze się udaje w przypadku funkcji *in situ*.

Dodatkowe cechy preprocesora

Jak już wspomniano, *niemal zawsze* jest preferowane używanie funkcji **inline** zamiast makroinstrukcji preprocesora. Wyjątkami są sytuacje, w których zamierzasz wykorzystać trzy specjalne funkcje preprocesora języka C (będącego równocześnie **preprocessorem językaC++**) — łąńcuchowanie, łączenie łańcuchów i sklejanie symboli. Łąńcuchowanie, wprowadzone we wcześniejszej części książki, jest wykonywane za pomocą

dyrektywy `#`, pozwalającej na zmianę identyfikatora w tablicę znakową. Łączenie łańcuchów zachodzi wtedy, gdy dwie sąsiednie tablice znakowe nie są oddzielone od siebie znakami interpunkcyjnymi, dzięki czemu następuje ich połączenie. Te dwie właściwości są szczególnie przydatne w czasie tworzenia kodu uruchomieniowego. Na przykład makroinstrukcja:

```
#define DEBUG(x) cout << fx " = " << x << endl
```

drukuję wartość dowolnej zmiennej. Można również utworzyć makroinstrukcję „śleczną” aktualnie wykonywane instrukcje:

```
#define TRACE(s) cerr << #s << endl: s
```

Dyrektywa `#s` przekształca instrukcję w łańcuch, by go wydrukować, natomiast drugi s powtarza instrukcję, dzięki czemu jest ona wykonywana. Oczywiście, może to powodować problemy, szczególnie w przypadku jednowierszowych pętli `for`:

```
for(int i = 0; i < 100; i++)
    TRACE(f(i));
```

Z uwagi na to, że makroinstrukcja `TRACE()` zawiera w rzeczywistości dwie instrukcje, jednowierszowa pętla `for` wykonuje tylko pierwszą z nich. Rozwiązaniem jest zamiana występującego w makroinstrukcji średnika na przecinek.

Sklejanie symboli

Sklejanie symboli, zaimplementowane w postaci dyrektywy `##`, jest bardzo przydatne podczas tworzenia kodu. Pozwala ono na sklejenie ze sobą dwóch symboli, tworzące w wyniku automatycznie nowy identyfikator. Naprzkład:

```
#define FIELO(a) char* a##_string; int a##_size
class Record {
    FIELO(one);
    FIELD(two);
    FIELD(three);
    // ...
};
```

Każde wywołanie makroinstrukcji `FIELD()` utworzy pole składające się z dwóch identyfikatorów — przeznaczonego do przechowywania tablicy znakowej i służącego do przechowywania długości tej tablicy. Jest to nie tylko łatwiejsze w czytaniu, ale pozwala również na wyeliminowanie błędów związanych z kodowaniem, a także ułatwia pielęgnację programu.

Udoskonalona kontrola błędów

Do tej pory były używane funkcje zawarte w pliku nagłówkowym `require.h`, ale nie zostały one jeszcze zdefiniowane (chociaż tam, gdzie to było celowe, do wykrywania błędów programistycznych była również stosowana makroinstrukcja `assert()`). Nadszedł czas na zdefiniowanie tego pliku nagłówkowego. Funkcje inline są wygodne, ponieważ

pozwalały na umieszczenie wszystkiego w pliku nagłówkowym, co upraszcza wykorzystywanie pakietu. Dołącza się jedynie plik nagłówkowy i nie trzeba się troszczyć o dołączenie pliku zawierającego implementację.

Należy również pamiętać, że wyjątki (opisane szczegółowo w drugim tomie książki) udostępniają znacznie bardziej efektywny sposób obsługi wielu rodzajów błędów — szczególnie tych, z którymi dobrze byłoby sobie poradzić zamiast po prostu przerwać program. **Zdarzenia** obsługiwane przez funkcje, zawarte w pliku **require.h**, uniemożliwiają jednak kontynuację programu — jak np. niepodanie przez użytkownika, w wierszu poleceń, odpowiedniej liczby argumentów lub brak możliwości otwarcia pliku. Tak więc jest do przyjęcia, że powodują one wywołanie funkcji **exit()**, pochodzącej ze standardowej biblioteki języka C.

Przedstawiony poniżej plik nagłówkowy został umieszczony w głównym katalogu, do którego zostały rozpakowane pliki zawierające programy źródłowe książki. Dzięki temu jest on łatwo dostępny z poziomu wszystkich rozdziałów:

```
//: require.h
// Kontrola wystąpienia błędów w programach
// Lokalne "using namespace std" dla starych kompilatorów
#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <string>

inline void require(bool requirement,
    const std::string& msg = "Warunek nie został spełniony"){
    using namespace std;
    if (!requirement) {
        fputs(msg.c_str(), stderr);
        fputs("\n", stderr);
        exit(1);
    }
}

inline void requireArgs(int argc, int args,
    const std::string& msg =
        "Należy podać %d argumentów") {
    using namespace std;
    if (argc != args + 1) {
        fprintf(stderr, msg.c_str(), args);
        fputs("\n", stderr);
        exit(1);
    }
}

inline void requireMinArgs(int argc, int minArgs,
    const std::string& msg =
        "Należy podać przynajmniej %d argumentów") {
    using namespace std;
    if(argc < minArgs + 1) {
        fprintf(stderr, msg.c_str(), minArgs);
        fputs("\n", stderr);
        exit(1);
    }
}
```

```
}

inline void assure(std::ifstream& in,
    const std::string& filename = "") {
    using namespace std;
    if(!in) {
        fprintf(stderr, "Nie možna otworzyć pliku %s\n",
            filename.c_str());
        exit(1);
    }
}

inline void assure(std::ofstream& out,
    const std::string& filename = "") {
    using namespace std;
    if(!out) {
        fprintf(stderr, "Nie možna otworzyć pliku %s\n",
            filename.c_str());
        exit(1);
    }
}

#endif // REQUIRE_H //:-~
```

Domyślne wartości zapewniają rozsądne komunikaty, które mogą być w razie potrzeby zmienione.

Zamiast argumentów typu **char*** używane są argumenty typu **const string&**. Pozwala to na stosowanie w charakterze argumentów tych funkcji zarówno tablic znakowych (**char***), jak i łańcuchów (**string**), co powoduje ich większą użyteczność (możesz wykorzystać ten wzór w pisany przez siebie kodzie).

W definicjach funkcji **requireArgs()** i **requireMinArgs()** liczba argumentów, których podanie jest wymagane w wierszu poleceń, powiększana jest o jeden, ponieważ zmienna **argc** zawsze uwzględnia nazwę uruchamianego programu jako argument zerowy. Z tego powodu przyjmuje ona wartość o jeden większą niż liczba argumentów faktycznie podanych w wierszu poleceń.

Zwróci uwagę na lokalne użycie deklaracji „**using namespace std**”, znajdujące się wewnętrz každej funkcji. Wynika ono z faktu, że w momencie pisania książki niektóre kompilatory nie dołączały prawidłowo funkcji biblioteki języka C w przestrzeni nazw **std**, więc jej jawną specyfikację wywołałaby błąd w trakcie komplikacji. Lokalne deklaracje umożliwiają funkcjom, zawartym w pliku **require.h**, poprawne działanie, zarówno z prawidłowymi, jak i nieprawidłowymi bibliotekami. Nie zachodzi konieczność otwierania przestrzeni nazw **std** każdemu, kto dołączył ten plik nagłówkowy.

Poniżej zamieszczono prosty program, testujący funkcje zawarte w pliku **require.h**:

```
//: C09:ErrTest.cpp
//{T} ErrTest.cpp
// Test funkcji zawartych w pliku require.h
#include "../require.h"
#include <iostream>
using namespace std;
```

```
int main(int argc, char* argv[]) {
    int i = 1;
    require(i, "wartosc musi być niezerowa");
    requireArgs(argc, 1);
    requireMinArgs(argc, 1);
    ifstream in(argv[i]);
    assure(in, argv[1]); // Wykorzystanie nazwy pliku
    ifstream nofile("nofile.xxx");
    // Fails:
    //! assure(nofile); // Argument domyślny
    ofstream out("tmp.txt");
    assure(out);
} ///:-
```

Być może pokusisz się o to, by pojść w sprawie otwierania plików o krok dalej, dodając do pliku nagłówkowego **require.h** makroinstrukcję:

```
#define IFOPEN(VAR, NAME) \
    ifstream VAR(NAME); \
    assure(VAR, NAME);
```

która powinna być następnie używana w następujący sposób:

```
IFOPEN(in, argv[1])
```

Na pierwszy rzut oka może to wyglądać interesująco, ponieważ oznacza oszczędność w pisaniu. Nie jest to jednak zalecane, choć nie stanowi zagrożenia. Zwróć ponownie uwagę na to, że przedstawiona makroinstrukcja wygląda co prawda jak funkcja, ale działa inaczej — w rzeczywistości tworzy ona obiekt (in) o zasięgu wykraczającym poza makroinstrukcję. Dla początkujących programistów i tych, którzy zajmują się pielęgnacją kodu, to kolejny twardy orzech do zgryzienia. Język C++ jest dostatecznie skomplikowany, bez wprowadzania dodatkowych niejasności, więc, o ile możesz, unikaj używania makroinstrukcji preprocessora.

Podsumowanie

Bardzo istotna jest możliwość ukrycia wewnętrznej implementacji klasy, gdyż w przyszłości możesz zechcieć ją zmienić. Zmiany, których dokonasz, będą wynikały z potrzeby zwiększenia efektywności, lepszego zrozumienia problemu lub dlatego, że dostępne będą jakieś nowe klasy, których zechcesz użyć w swojej implementacji. Wszystko, co zagraża prywatności wewnętrznej implementacji klasy, zmniejsza zatem elastyczność języka. Dlatego też funkcje inline odgrywają bardzo ważną rolę, ponieważ praktycznie eliminują one potrzebę używania makroinstrukcji preprocessora i towarzyszących im problemów. W przypadku użycia funkcji inline funkcje składowe mogą być również efektywne, co makroinstrukcje preprocessora.

Oczywiście, funkcje inline, występujące w definicjach klas, mogą być również nadużywane. Programista jest wręcz do tego zachęcany, ponieważ łatwiej napisać kod w taki sposób. Nie stanowi to jednak poważnego problemu, ponieważ później, poszukując możliwości zmniejszenia kodu, można zawsze zamienić funkcje inline w zwykłe funkcje, bez wpływu na ich działanie. Dewizą projektanta powinno być powiedzenie: „najpierw spraw, aby działało, a później optymalizuj”.

ćwiczenia

Rozwiązań wybranych ćwiczeń znajdują się w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Napisz program, który używa przedstawionej na początku rozdziału makroinstrukcji **F()** i pokazuje, że nie jest ona rozwijana poprawnie, jak to opisano w tekście. Popraw makroinstrukcję i udowodnij, że teraz już działa poprawnie.
2. Napisz program, wykorzystujący makroinstrukcję **FLOOR()**, przedstawioną na początku rozdziału. Wskaż warunki, w których nie działa ona poprawnie.
3. Zmodyfikuj plik **MacroSideEffects.cpp** w taki sposób, aby makroinstrukcja **BAND()** działała prawidłowo.
4. Utwórz dwie identyczne funkcje — **f1()** i **f2()**. Określ funkcję **f1()** jako funkcję inline, pozostawiając **f2()** zwykłą funkcją. Wykorzystaj do oznaczenia punktów początkowych i końcowych standardową funkcję biblioteczną **clock()**, dostępną w pliku nagłówkowym **<ctime>**, a następnie porównaj czasy wykonania obu funkcji, aby przekonać się, która z nich jest szybsza. Może trzeba będzie dokonać wielokrotnego wywołania funkcji w pętli, w której mierzony jest czas, aby uzyskać możliwe do wykorzystania rezultaty.
5. Przeprowadź eksperymenty z wielkością i złożonością kodu, zawartego w funkcjach, występujących w poprzednim ćwiczeniu. Sprawdź, czy da się określić graniczny punkt, w którym wykonanie funkcji inline i zwyczajnej funkcji zajmuje tyle samo czasu. Jeżeli masz taką możliwość, to wypróbuj różne kompilatory i zanotuj uzyskane wyniki.
6. Udowodnij, że funkcje inline podlegają domyślnie wewnętrznemu łączeniu.
7. Utwórz klasę **zawierającą tablicę znaków**. Dodaj konstruktor typu **inline**, wykorzystujący funkcję **memset()**, pochodzącej ze standardowej biblioteki języka C, do inicjalizacji tablicy **wartością argumentu** konstruktora (domyślnie powinna wynosić ona " "), oraz funkcję inline o nazwie **print()**, drukującą wszystkie znaki znajdujące się w tablicy.
8. W przykładowym programie **NestFriend.cpp**, opisanym w rozdziale 5., wymień wszystkie funkcje na funkcje inline — w taki sposób, aby nie były funkcjami *in situ*. Zamień również funkcje **initialize()** na konstruktory.
9. Zmodyfikuj program **StringStack.cpp**, opisany w rozdziale 8., w taki sposób, aby wykorzystywał funkcje inline.
10. Utwórz wyliczenie o nazwie **Hue**, zawierające elementy **red**, **blue** i **yellow**. Następnie utwórz klasę o nazwie **Color**, zawierającą składową typu **Hue**, oraz konstruktor, przypisujący tej składowej wartość, zgodnie z podanym argumentem. Dodaj funkcje udostępniające, umożliwiające „pobieranie” i „ustawianie” wartości składowej typu **Hue**. Uczyń wszystkie te funkcje funkcjami inline.

11. Zmodyfikuj poprzednie ćwiczenie w taki sposób, by zostało wykorzystane podejście związane z „obserwatorem” i „modyfikatorem”.
±2. Zmodyfikuj program **CppTime.cpp** w taki sposób, by mierzył czas od rozpoczęcia działania programu do chwili, gdy użytkownik naciśnie klawisz „Enter” albo „Return”.
13. Utwórz klasę, posiadającą dwie funkcje składowe, będące funkcjami inline. Pierwsza z funkcji, zdefiniowana wewnątrz klasy, powinna wywoływać drugą funkcję, bez potrzeby tworzenia jej wcześniejszej deklaracji. Napisz funkcję **main() tworzącą obiekt** tej klasy i wywołaj pierwszą funkcję.
14. Utwórz klasę A, posiadającą domyślny konstruktor typu inline, który ogłasza swoje wywołanie. Następnie utwórz klasę B, umieść w niej, w charakterze składowej, obiekt klasy A i utwórz konstruktor klasy B, będący funkcją inline. Utwórz tablicę obiektów klasy B i zobacz, co się stanie.
15. Utwórz dużą liczbę obiektów z poprzedniego ćwiczenia, używając klasy **Time** do określenia różnicy w czasie pomiędzy wykorzystaniem zwyczajnych konstruktorów i konstruktorów inline (jeżeli dysponujesz programem profilującym, spróbuj go również użyć).
16. Napisz program, który przyjmuje łańcuch jako argument wiersza poleceń. Utwórz pętlę **for**, usuwającą tego łańcucha w każdym przebiegu pojedynczym znaku, i wykorzystaj opisaną w rozdziale makroinstrukcję **DEBUG()** do drukowania za każdym razem aktualnej postaci łańcucha.
17. Popraw makroinstrukcję **TRACE()** w taki sposób, jak opisano w rozdziale, i udowodnij, że działa ona poprawnie.
18. Zmodyfikuj makroinstrukcję **FIELD()** w taki sposób, aby zawierała dodatkowo numer indeksowy. Utwórz klasę, której składowe zostaną utworzone za pomocą wywołań makroinstrukcji **FIELD()**. Dodaj funkcję **składową**, pozwalającą na dostęp do pola, utworzonego za pomocą makroinstrukcji na podstawie jego numeru indeksowego. Napisz funkcję **main()**, umożliwiającą przetestowanie tej klasy.
19. Zmodyfikuj makroinstrukcję **FIELD()** w taki sposób, aby dla każdego utworzonego przez siebie pola automatycznie tworzyła funkcje udostępniające (dane powinny jednak nadal pozostać danymi prywatnymi). Utwórz klasę, której składowe utworzono za pomocą wywołań makroinstrukcji **FIELD()**. Napisz funkcję **main()**, umożliwiającą przetestowanie klasy.
20. Napisz program przyjmujący w wierszu poleceń dwa argumenty — liczbę całkowitą i nazwę pliku. Użyj funkcji znajdujących się w pliku nagłówkowym **require.h**, by upewnić się, że: podano odpowiednią liczbę argumentów, wartość liczby całkowitej zawiera się w przedziale od 5 do 10, a plik może być pomyślnie otwarty.
21. Napisz program wykorzystujący makroinstrukcję **ffOPEN()** do otwarcia pliku w charakterze strumienia wejściowego. Zwróć uwagę na tworzenie obiektu **ifstream** oraz jego zasięg.

- 22. (Trudne)** Dowiedz się, w jaki sposób wygenerować na wyjściu kompilatora kod w asemblerze. Utwórz plik zawierający bardzo małą funkcję oraz funkcję **main()**, w której ta funkcja jest wywoływana. Wygeneruj kod w języku asemblera, gdy funkcja ta jest rozwijana jako funkcja inline i gdy nie jest ona rozwijana, a następnie pokaż, że w przypadku wersji inline nie występuje narzut związany z wywołaniem funkcji.

Rozdział 10.

Zarządzanie nazwami

Tworzenie nazw jest jedną z podstawowych czynności wykonywanych podczas programowania — w miarę rozrastania się projektu liczba używanych nazw może wydać się przytłaczająca.

C++ pozwala na zachowanie doskonałej kontroli nad tworzeniem nazw, ich widocznością, przydzielaną nazwom pamięcią oraz łączeniem.

Słowo kluczowe **static** zostało w języku C przeciążone, kiedy nikt jeszcze nie wieǳiał, co oznacza termin „przeciążenie”, a język C++ nadał jeszcze słowu **static** dodatkowe znaczenia. Podstawowym pojęciem, wspólnym dla wszystkich zastosowań słowa **static**, wydaje się: „coś, co pozostaje na swoim miejscu” (**jak** statyczne ładunki elektryczne), niezależnie od tego, czy oznacza to fizyczne miejsce w pamięci, czy też widoczność w obrębie pliku.

W tym rozdziale dowiesz się, w jaki sposób słowo kluczowe **static** steruje pamięcią i **widocznością**, a także poznasz osiągalny w języku C++, lepszy sposób kontroli dostępu do nazw, w którym jest wykorzystana *przestrzeń nazw* (ang. *namespace*). Dowiesz się również, jak używać funkcji, które zostały napisane i skompilowane w języku C.

Statyczne elementy języka C

Zarówno w języku C, jak i w C++ słowo kluczowe **static** ma dwa podstawowe znaczenia, które, niestety, często ze sobą kolidują:

1. Umieszczony pod raz ustalonym adresem — oznacza to, że obiekt został utworzony w specjalnym *statycznym obszarze danych* zamiast być tworzony na stosie podczas każdego wywołania funkcji. Odpowiada to **pojęciu pamięci statycznej**.
2. Lokalny w stosunku do określonej jednostki translacji (a także w stosunku do zasięgu klasy w języku C++, jak zobaczymy później). W tym przypadku słowo kluczowe **static** decyduje o *widoczności* nazwy, w taki sposób, że nie jest ona widoczna **poza jednostką translacji** lub klasą. Opisuje to również pojęcie *łączenia*, określające, które nazwy są widziane przez program łączący.

W tej części przyjrzymy się powyższym znaczeniom słowa kluczowego **static**, w taki postaci, w jakiej zostały one odziedziczone pojęzyku C.

mienne statyczne znajdujące się wewnątrz funkcji

Kiedy wewnątrz funkcji tworzona jest zmienna lokalna, kompilator przydziela jej pamięć, za każdym razem, gdy wywoływana jest funkcja — przesuwając wskaźnik stosu o odpowiednią wielkość w dół. Jeżeli istnieje inicjator tej zmiennej, to inicjalizacja następuje, ilekroć jest przekraczany punkt sekwencyjny.

Jednak czasami istnieje potrzeba zachowania wartości pomiędzy wywołaniami funkcji. Można to osiągnąć, definiując zmienną globalną, ale w takim przypadku zmienna nie będzie znajdowała się pod wyłączną kontrolą funkcji. Języki C i C++ pozwalają na utworzenie w obrębie funkcji obiektu statycznego — pamięć nie jest przydzielana temu obiekowi na stosie, tylko w obszarze danych statycznych programu. Obiekt taki jest inicjalizowany tylko jednokrotnie — podczas pierwszego wywołania funkcji; zachowuje on ponadto swoją wartość pomiędzy wywołaniami funkcji. Na przykład poniższa funkcja przy każdym wywołaniu zwraca następny znak znajdujący się w tablicy:

```
//: C10:StaticVariablesInfunctions.cpp
#include "../require.h"
#include <iostream>
using namespace std;

char oneChar(const char* charArray = 0) {
    static const char* s;
    if(charArray) {
        s = charArray;
        return *s;
    }
    else
        require(s, "niezainicjowana zmienna s");
    if(*s == '\0')
        return 0;
    return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
    // oneChar(); // require() kończy się niepowodzeniem
    oneChar(a); // Inicjalizacja s wartością a
    char c;
    while((c = oneChar()) != 0)
        cout << c << endl;
} //:-
```

Statyczna zmienna, zdefiniowana jako **static char* s**, przechowuje swoją wartość pomiędzy wywołaniami funkcji **oneChar()**, ponieważ przydzielona jej pamięć **nie** stanowi części ramki stosu funkcji, ale znajduje się w obszarze danych statycznych programu. Kiedy funkcja **oneChar()** zostanie wywołana z argumentem typu **char**, to wartość tego argumentu przypisywana jest zmiennej **s** i zwracany jest pierwszy

znak, znajdujący się w tablicy. Każde kolejne wywołanie funkcji **oneChar()** bez argumentu powoduje przypisanie argumentowi **charArray** domyślnej, zerowej wartości, co informuje funkcję, że nadal pobierane są znaki, wskazywane przez uprzednio zainicjowaną wartość zmiennej **s**. Funkcja ta będzie nadal zwracać znaki, aż do momentu, gdy zostanie napotkana wartość zerowa, kończąca tablicę znakową. W tym miejscu funkcja przestanie inkrementować wskaźnik, dzięki czemu nie wykroczy on poza koniec tablicy.

Co się jednak stanie w przypadku, gdy funkcja **oneChar()** zostanie wywołana bez argumentów, a wartość zmiennej **s** nie zostanie uprzednio zainicjowana? W definicji zmiennej **s** można by dostarczyć wartość inicjującą:

```
static char* s = 0;
```

Lecz jeżeli w przypadku statycznej zmiennej wbudowanego typu nie zostanie określona inicjująca ją wartość, to kompilator gwarantuje, że zmienna ta zostanie zainicjowana wartością zerową (przekształconą w odpowiedni typ) w czasie rozpoczęcia pracy programu. Tak więc gdy funkcja **oneChar()** jest wywoływana po raz pierwszy, zmienna **s** ma wartość zerową. W tym przypadku zostanie to wykryte przez instrukcję warunkową **if(!s)**¹.

Przedstawiona powyżej inicjalizacja zmiennej **s** jest bardzo prosta. Jednakże inicjalizacja obiektów statycznych (podobnie jak wszystkich innych obiektów) może być dowolnym wyrażeniem, zawierającym stałe oraz uprzednio zadeklarowane zmienne i funkcje.

Należy być świadomym tego, że przedstawiona powyżej funkcja jest bardzo podatna na problemy związane z **wielowatkowością** — ilekroć projektowane są funkcje, zawierające zmienne statyczne, należy pamiętać o kwestiach dotyczących wielowatkowości.

Statyczne obiekty klas znajdujące się wewnątrz funkcji

Zasady są identyczne jak w przypadku statycznych obiektów, włącznie z tym, że w przypadku obiektów wymagany jest jakiś rodzaj inicjalizacji. Jednak przypisanie wartości zerowej ma sens jedynie w przypadku typów wbudowanych — typy zdefiniowane przez użytkownika muszą być inicjalizowane za pomocą wywołania konstruktora. Jeżeli zatem podczas definicji statycznego obiektu nie zostaną określone argumenty konstruktora, to klasa będzie musiała posiadać konstruktor domyślny, jak w poniższym przykładzie:

```
//: C10:StaticObjectsInFunctions.cpp
#include <iostream>
using namespace std;

class X {
    int i;
public:
    X(int ii = 0) : i(ii) {} // Domyślny
    ~X() { cout << "X::~X()" << endl; }
};
```

¹ Wywoływaną w funkcji **require()** — przyp. tłum.

```

void f() {
    static X x1(47);
    static X x2; // Wymagany jest konstruktor domyślny
}

int main() {
    f();
} //-

```

Statyczny obiekt typu X, znajdujący się wewnętrz funkcji **f()**, może być zainicjowany zarówno za pomocą listy argumentów konstruktora, jak i konstruktora domyślnego. Inicjalizacja ta zachodzi wówczas, gdy sterowanie po raz pierwszy przechodzi przez definicję obiektu — i tylko po raz pierwszy.

Destruktory obiektów statycznych

Destruktory obiektów statycznych (czyli wszystkich obiektów umieszczonych w statycznej pamięci, a nie tylko lokalnych obiektów statycznych, jak w powyższym przykładzie) są wywoływane wtedy, gdy kończy pracę funkcja **main()** lub wywołana jest jawnie funkcja **exit()**, zawarta w standardowej bibliotece C++. W przypadku większości implementacji funkcja **main()**, kończąc pracę, wywołuje po prostu funkcję **exit()**. Oznacza to, że wywołanie funkcji **exit()** wewnętrz destruktora jest potencjalnie niebezpieczne, może bowiem prowadzić do nieskończonej rekurencji. Destruktory obiektów statycznych nie są wywoływane, jeżeli opuści się program, używając funkcji **abort()**, należącej do standardowej biblioteki języka C.

Można określić działania, które następują po opuszczeniu funkcji **main()** (albo wywołania funkcji **exit()**), używając do tego celu funkcji **atexit()**, zawartej w standardowej bibliotece języka C. W takim przypadku funkcja zarejestrowana przez funkcję **atexit()** może zostać wywołana przed destruktorem **wszelkich** obiektów, utworzonych przed opuszczeniem funkcji **main()** (lub wywołaniem funkcji **exit()**).

Podobnie jak w przypadku zwykłego niszczenia obiektów, niszczenie obiektów statycznych następuje w kolejności odwrotnej do kolejności ich inicjalizacji. Jednak niszczone są tylko te obiekty, które zostały utworzone. Na szczęście, narzędzia programistyczne języka C++ umożliwiają zapamiętanie kolejności tworzenia obiektów. Obiekty globalne są zawsze tworzone przed wejściem do funkcji **main()**, a usuwane po jej zakończeniu. Jednakże w przypadku gdy funkcja zawierająca lokalne obiekty statyczne nie jest nigdy wywoływana, nie jest też nigdy wykonywany jej konstruktor, a więc także i jej destruktor. Ilustruje to poniższy przykład:

```

//: C10:StaticDestructors.cpp
// Destruktory obiektów statycznych
#include <iostream>
using namespace std;
ofstream out("statdest.out"); // Plik śledzenia

class Obj {
    char c; // Identyfikator
public:
    Obj(char cc) : c(cc) {
        out << "Obj::Obj() dla " << c << endl;
    }
}

```

```

~Obj() {
    out << "Obj::~Obj() dla " << c << endl;
}
};

Obj a('a'); // Obiekt globalny (pamięć statyczna)
// Konstruktor i destruktor są zawsze wywoływane

void f() {
    static Obj b('b');
}

void g() {
    static Obj c('c');
}

int main() {
    out << "wewnętrz funkcji main()" << endl;
    f(); // Wywołanie statycznego konstruktora obiektu b
    // Funkcja g() nie została wywołana
    out << "opuszczanie funkcji main()" << endl;
} //:~

```

W klasie **Obj** składowa znakowa `c` działa jako identyfikator, dzięki czemu zarówno konstruktor, jak i destruktor mogą drukować informacje o obiekcie, na którym działają. Obiekt `a` jest obiektem globalnym, więc jego konstruktor jest wywoływany zawsze przed wejściem do funkcji **main()**. Jednakże konstruktorzy obiektów statycznych `b` i `c`, zdefiniowanych odpowiednio wewnątrz funkcji **f()** i **g()**, są uruchamiane tylko wówczas, gdy wywoływane są te funkcje.

Aby zademonstrować, które konstruktorы i destruktorы są uruchamiane, wywoływana jest tylko funkcja **f()**. Wyniki działania programu są następujące:

```

Obj::Obj() dla a
wewnętrz funkcji main()
Obj::Obj() dla b
opuszczanie funkcji main()
Obj::~Obj() dla b
Obj::~Obj() dla a

```

Konstruktor obiektu `a` jest wywoływany przed wejściem do funkcji **main()**, a konstruktor obiektu `b` — tylko dlatego, że wywoływana jest funkcja **f()**. Kiedy kończy się funkcja **main()**, wywoływane są destruktorы tych obiektów, które zostały utworzone — w kolejności odwrotnej do kolejności ich tworzenia. Oznacza to, że gdyby wywołana została funkcja **g()**, to kolejność wykonania destruktorów obiektów `b` i `c` zależałaby od tego, czy jako pierwsza wywołana została funkcja **f()**, czy też **g()**.

Zwróć uwagę na to, że obiekt `out`, reprezentujący plik śledzenia typu **ofstream**, jest również obiektem statycznym — ponieważ został on zdefiniowany poza wszystkimi funkcjami, więc znajduje się w obszarze danych statycznych. Ważne jest, że jego definicja (w odróżnieniu od deklaracji **extern**) znajduje się na początku pliku, przed jakimkolwiek możliwym użyciem obiektu `out`. W przeciwnym razie obiekt byłby bowiem używany, zanim zostałby prawidłowo zainicjowany.

W języku C++ konstruktor globalnego obiektu statycznego jest uruchamiany jeszcze przed wywołaniem funkcji **main()**, co udostępnia prosty i przenośny sposób wykonywania kodu przed wejściem do funkcji **main()**, a także wykonanie kodu zawartego w destruktorze — po opuszczeniu funkcji **main()**. Uzyskanie tego w języku C stanowiło zawsze eksperiment, wymagający przeszukiwanie asemblerowego kodu, inicjującego pracę programu dostarczanego przez producenta kompilatora.

Śterowanie łączeniem

W normalnym przypadku każda nazwa znajdująca się w *zasięgu pliku* (to znaczy niezagnieżdżona w żadnej klasie ani funkcji), jest widoczna we wszystkich jednostkach translacji programu. Jest to często określane mianem *łączenia zewnętrznego*, ponieważ w trakcie łączenia nazwa jest widoczna dla programu łączącego w każdym miejscu, znajdującym się na zewnątrz jednostki translacji, w której się ona znajduje. Zmienne globalne oraz zwykłe funkcje podlegają łączeniu zewnętrznemu.

Zdarzają się jednak przypadki, w których chcielibyśmy ograniczyć widoczność nazwy. Moglibyśmy potrzebować zmiennej, widocznej w zasięgu pliku, dzięki czemu istniałaby możliwość wykorzystania jej przez wszystkie funkcje, zawarte w tym pliku, ale jednocześnie nie chcieć, by funkcje, znajdujące się w innych plikach, widziały ją lub miały do niej dostęp. Nie chcielibyśmy również niechcący wywołać konfliktu nazw w stosunku do identyfikatorów, znajdujących się w innych plikach.

Obiekt lub nazwa funkcji, znajdujący się w zasięgu pliku, który został jawnie zadeklarowany jako statyczny, jest lokalny w stosunku do swojej jednostki translacji (zgodnie z terminologią stosowaną w książce — w stosunku do pliku .cpp, w którym wstępuje jego deklaracja). Jego nazwa podlega *łączению wewnętrznemu*. Oznacza to, że ta sama nazwa może być używana w innych jednostkach translacji, nie wywołując konfliktu nazw.

Jedną z zalet łączenia wewnętrznego jest ta, że nazwa może zostać umieszczona w pliku nagłówkowym i nie ogrywa żadnej roli ewentualny konflikt nazw. Nazwy, które są typowo umieszczane w plikach nagłówkowych, takie jak definicje stałych lub funkcje inline, są domyślnie łączone wewnętrznie (**jednak** stałe podlegają domyślnie wewnętrznemu łączeniu tylko w języku C++ — w języku C są one domyślnie łączone zewnętrznie). Zwróć uwagę na to, że łączenie odnosi się jedynie do elementów posiadających w chwili łączenia (ładowania) swój adres. Deklaracje klas oraz zmienne lokalne nie podlegają zatem łączeniu.

Zamieszczanie

Poniżej zamieszczono przykład, w którym oba znaczenia słowa „statyczny” mogą się ze sobą pokrywać. Wszystkie obiekty globalne należą w sposób niejawny do klasy pamięci statycznej, więc jeżeli napiszemy (w zasięgu pliku):

```
int a = 0;
```

to zmiennej a zostanie przydzielona pamięć w obszarze danych statycznych, a inicjalizacja tej zmiennej nastąpi przed wejściem do funkcji **main()**. W dodatku zmienna ta będzie widoczna globalnie, we wszystkich jednostkach translacji. W terminologii

dotyczącej widoczności przeciwieństwem słowa kluczowego **static** (widoczny tylko w bieżącej jednostce translacji) jest słowo kluczowe **extern**, określające w jawnym sposób, że widoczność nazwy obejmuje wszystkie jednostki translacji. Tak więc powyższa **definicja jest równoważna zapisowi:**

```
extern int a = 0;
```

Jeżeli jednak zamiast tego napiszemy:

```
static int a = 0;
```

to jedynym rezultatem będzie zmiana widoczności zmiennej, dzięki czemu zmienna **a** podlega od tej pory łączeniu wewnętrzemu. Klasa pamięci pozostaje jednak niezmieniona — obiekt znajduje się w obszarze danych statycznych, niezależnie od tego, czy jego widoczność ma charakter „statyczny” (**static**) czy też „zewnętrzny” (**extern**).

Kiedy przechodzimy do zmiennych lokalnych, słowo kluczowe **static** przestaje zmieniać widoczność zmiennych, wywierając natomiast wpływ na ich klasę pamięci.

Jeżeli zmienną, która wygląda na zmienną **lokalną**, zadeklaruje się przy użyciu słowa kluczowego **extern**, oznacza to, że jest ona przechowywana w jakimś innym miejscu pamięci (jest więc ona w rzeczywistości globalna w stosunku do funkcji). Na przykład:

```
//: C10:LocalExtern.cpp
//{ LocalExtern2
#include <iostream>

int main() {
    extern int i;
    std::cout << i;
} //:~

//: C10:LocalExtern2.cpp {0}
int i = 5;
//:~
```

W przypadku nazw funkcji (dla funkcji niebędących składowymi klas) słowa kluczowe **static** oraz **extern** zmieniają tylko ich widoczność. Dlatego też zapis:

```
extern void f();
```

jest równoważny zwykłej deklaracji:

```
void f();
```

Natomiast następujący zapis:

```
static void f();
```

oznacza, że funkcja **f()** jest widoczna tylko w obrębie bieżącej jednostki translacji, — czasami określa się, że jest ona *statyczna w obrębie pliku*.

Inne specyfikatory klas pamięci

Najczęściej używane są specyfikatory **static** i **extern**; są jeszcze dwa inne, rzadziej stosowane. Specyfikator **auto** nie jest **niemal** nigdy wykorzystywany, ponieważ informuje on kompilator, że zmienna jest zmienną lokalną. Słowo kluczowe **auto** jest skrótem słowa „automatyczny” i odnosi się do sposobu, w jaki kompilator przydziela zmiennym pamięć. Ponieważ kompilator potrafi zawsze określić to na podstawie kontekstu, w którym zdefiniowane są zmienne, więc słowo kluczowe **autojest** słowem nadmiarowym.

Zmienna zdefiniowana za pomocą słowa kluczowego **register** jest zmienną lokalną (**auto**), zawierającą wskazówkę dla kompilatora, że będzie ona intensywnie używana, w związku z czym kompilator powinien — o ile to możliwe — przechowywać ją w rejestrze. Specyfikator **register** wspomaga więc proces optymalizacji. Różne kompilatory w rozmaity sposób reagują na tę wskazówkę — zazwyczaj posiadają również opcję umożliwiającą zignorowanie. W przypadku pobrania adresu zmiennej specyfikator **register** zostanie niemal na pewno zignorowany. Należy unikać stosowania słowa kluczowego **register**, ponieważ kompilator potrafi na ogół przeprowadzić optymalizację lepiej niż programista.

Przestrzenie nazw

Mimo że nazwy mogą być zagnieżdżane w klasach, to nazwy globalnych funkcji oraz klas znajdują się nadal w jednej, globalnej przestrzeni nazw. Słowo kluczowe **static** zapewnia nad tym pewną kontrolę, umożliwiając określenie, że zmienne i funkcje podlegają łączeniu wewnętrznemu (czyli są one statyczne w obrębie pliku). Jednak w przypadku dużego projektu brak kontroli nad globalną przestrzenią nazw może powodować problemy. Aby poradzić sobie z nimi w stosunku do klas, producenci tworzą często długie, skomplikowane nazwy. Zapobiegają one co prawda wystąpieniu konfliktu, ale ich wpisywanie jest niewygodne (w celu ich uproszczenia często używana jest deklaracja **typedef**). Nie jest to eleganckie rozwiązanie wspierane przez język.

Wykorzystując dostępne w języku C++ *przestrzenie nazw* (ang. *namespaces*), można podzielić globalną przestrzeń nazw na łatwiejsze w zarządzaniu części. Podobnie jak słowa kluczowe **class**, **struct**, **enum** i **union**, słowo **namespace** umieszcza nazwy swoich składowych w odrębnej przestrzeni. Podczas gdy inne wymienione słowa kluczowe mają jeszcze inne zastosowania, to w przypadku słowa kluczowego **namespace** sprowadza się ono jedynie do utworzenia nowej przestrzeni nazw.

Tworzenie przestrzeni nazw

Tworzenie przestrzeni nazw jest dość podobne do tworzenia klas:

```
//: C10:MyLib.cpp
namespace MyLib {
    // Deklaracje
}
int main() {} ///:-
```

Powoduje ono utworzenie nowej przestrzeni **nazw**, obejmującej zawarte w niej deklaracje. W porównaniu ze sposobem użycia słów kluczowych **class**, **struct**, **union** i **enum** występują jednak dość istotne różnice:

- ◆ Definicja przestrzeni nazw może występować tylko w zasięgu globalnym lub być zagnieźdzona w innej przestrzeni nazw.
- ◆ Po nawiasie klamrowym, zamkającym **definicję** przestrzeni nazw, nie jest wymagany średnik.
- ◆ Definicja przestrzeni nazw może być „kontynuowana” w wielu plikach **nagłówkowych** — za pomocą składni, która w przypadku klas wyglądałaby na **powtózoną definicję**:

```
//: C10:Header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
    extern int x;
    void f();
    // ...
}

#endif // HEADER1_H //:-

//: C10:Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Dodanie do MyLib kolejnych nazw
namespace MyLib { // To nie jest powtórna definicja!
    extern int y;
    void g();
    // ...
}

#endif // HEADER2_H //:-

//: C10:Continuation.cpp
#include "Header2.h"
int main() {} //:-
```

- ◆ Można utworzyć *synonim* nazwy przestrzeni nazw, dzięki czemu unika się wpisywania niewygodnych nazw, utworzonych przez producentów bibliotek:

```
//: C10:BobsSuperDuperLibrary.cpp
namespace BobsSuperDuperLibrary {
    class Widget { /* ... */ };
    class Poppit { /* ... */ };
    // ...
}
// Zbyt dużo wpisywania! Utworzmy synonim:
namespace Bob = BobsSuperDuperLibrary;
int main() {} //:-
```

- ◆ W odróżnieniu od klasy, nie można utworzyć egzemplarza przestrzeni nazw.

bezimiennie przestrzenie nazw

Każda jednostka translacji zawiera bezimienną przestrzeń **nazw**, którą można powiększać, używając specyfikatora **namespace** bez identyfikatora:

```
//: C10:UnnamedNamespaces.cpp
namespace {
    class Arm { /* ... */ };
    class Leg { /* ... */ };
    class Head { /* ... */ };
    class Robot {
        Arm arm[4];
        Leg leg[16];
        Head head[3];
        // ...
    } xanthan;
    int i, j, k;
}
int main() {} //:~
```

Nazwy zawarte w tej przestrzeni są automatycznie dostępne w bieżącej jednostce translacji, bez potrzeby jej nazywania. Bezimienna przestrzeń nazw jest unikatowa w każdej jednostce translacji. Jeżeli w bezimiennej przestrzeni nazw zostaną umieszczone nazwy lokalne, to nie ma już potrzeby używania w stosunku do nich słowa kluczowego **static** w celu określenia, że mają podlegać łączeniu wewnętrzemu.

Aby ograniczyć zasięg nazwy do pliku w języku C++, zaleca się używanie bezimiennych przestrzeni nazw zamiast słowa kluczowego **static**.

przyjaciele

Do przestrzeni nazw można *wstrzyknąć* deklarację **friend**, umieszczając ją wewnątrz klasy, zawartej w tej przestrzeni:

```
//: C10:FriendInjection.cpp
namespace Me {
    class Us {
        //...
        friend void you();
    };
}
int main() {} //:~
```

Obecnie funkcja **you()** należy do przestrzeni nazw **Me**.

Jeżeli „przyjaciel” zostanie umieszczony w obrębie klasy, zawartej w globalnej przestrzeni nazw, będzie on „wstrzyknięty” globalnie.

Używanie przestrzeni nazw

Do nazwy zawartej w przestrzeni nazw można odwołać się na trzy sposoby: określając ją za pomocą operatora zasięgu, za pomocą dyrektywy **using**, umożliwiającej dostęp do **wszystkich** nazw zawartych w przestrzeni nazw, oraz używając deklaracji **using**, udostępniającej jednorazowo pojedynczą nazwę.

Zasięg

Każda nazwa znajdująca się w przestrzeni nazw może być wskazana **jawnie**, za pomocą operatora zasięgu — w taki sam sposób, w jaki odwołujemy się do nazw w obiekcie klas:

```
//: C10:ScopeResolution.cpp
namespace X {
    class Y {
        static int i;
    public:
        void f();
    };
    class Z;
    void func();
}
int X::Y::i = 9;
class X::Z {
    int u, v, w;
public:
    Z(int i);
    int g();
};
X::Z::Z(int i) { u - v = w - i; }
int X::Z::g() { return u = v = w = 0; }
void X::func() {
    X::Z a();
    a.g();
}
int main(){} //:~
```

Zwróci uwagę na to, że definicja X::Y::i mogłyby się równie dobrze odnosić do składowej klasy Y, zagnieżdżonej w klasie X, a nie do przestrzeni nazw X.

Na podstawie przedstawionych wyżej rozważań można stwierdzić, że przestrzenie nazw w dużym stopniu **przypominają klasy**.

Dyrektywa using

Ponieważ wpisywanie pełnych kwalifikatorów nazw, zawartych w przestrzeni nazw, szybko może stać się męczące, słowo kluczowe **using** umożliwia zainportowanie od razu całej przestrzeni nazw. Użycie tego słowa wraz ze słowem kluczowym **namespace** nosi nazwę *dyrektywy using* (ang. *using directive*). Dyrektywa **using** powoduje, że nazwy wyglądają tak, jakby należały do najbliższego zasięgu, obejmującego tę deklarację, co pozwala na wygodne **używanie** nazw **niekwalifikowanych**. Rozważmy prostą przestrzeń nazw:

```
//: C10:NamespaceInt.h
#ifndef NAMESPACEINT_H
#define NAMESPACEINT_H
namespace Int {
    enum sign { positive, negative };
    class Integer {
        int i;
        sign s;
```

```

public:
    Integer(int ii = 0)
        : i(ii),
          s(i > 0 ? positive : negative)
    {}
    sign getSign() const { return s; }
    void setSign(sign sgn) { s = sgn; }
    // ...
};

}

#endif // NAMESPACEINT_H //:~
```

Jednym z zastosowań dyrektywy **using** jest przeniesienie wszystkich nazw zawartych w przestrzeni **Int** do innej przestrzeni nazw i jednocześnie pozostawienie je zagnieżdżonymi w tej przestrzeni nazw:

```

//: C10:NamespaceMath.h
#ifndef NAMESPACEMATH_H
#define NAMESPACEMATH_H
#include "NamespaceInt.h"
namespace Math {
    using namespace Int;
    Integer a, b;
    Integerdivide(Integer, Integer);
    // ...
}
#endif // NAMESPACEMATH_H //:~
```

Można również zadeklarować wszystkie nazwy przestrzeni nazw **Int** wewnątrz funkcji, pozostawiając je jednak zagnieżdżonymi wewnątrz tej funkcji:

```

//: C10:Arithmetic.cpp
#include "NamespaceInt.h"
void arithmetic() {
    using namespace Int;
    Integer x;
    x.setSign(positive);
}
int main(){} //:~
```

Gdyby nie została użyta dyrektywa **using**, wszystkie nazwy, znajdujące się w przestrzeni nazw, musiałyby posiadać pełne kwalifikatory.

Jeden z aspektów dyrektywy **using** może wydawać się, na pierwszy rzut oka, nieco nielogiczny. Widoczność **nazw**, wprowadzona za pomocą dyrektywy **using**, jest ograniczona do zasięgu, w którym dyrektywa ta została użyta. Można jednak zasłonić nazwy pochodzące z dyrektywy **using** w taki sposób, jakby były one nazwami globalnymi w stosunku do tego zasięgu!

```

//: C10:NamespaceOverriding1.cpp
#include "NamespaceMath.h"
int main() {
    using namespace Math;
    Integer a; // Zasłania Math::a;
    a.setSign(negative);
    // Teraz użycie operatora zasięgu do
```

```
// wybrania Math::a jest niezbędne:
Math::a.setSign(positive);
} /II-
```

Założmy, że istnieje druga przestrzeń nazw, zawierająca niektóre nazwy znajdujące się w przestrzeni nazw **Math**:

```
//: C10:NamespaceOverriding2.h
#ifndef NAMESPACEOVERRIDING2_H
#define NAMESPACEOVERRIDING2_H
#include "NamespaceInt.h"
namespace Calculation {
    using namespace Int;
    Integerdivide(Integer, Integer);
    // ...
}
#endif // NAMESPACEOVERRIDING2_H ///:-
```

Ponieważ ta przestrzeń nazw została również wprowadzona za pomocą dyrektywy **using**, istnieje możliwość wystąpienia kolizji. Dwuznaczność występuje jednak w miejscu *użycia* nazwy, a nie w miejscu występowania dyrektywy **using**:

```
//: C10:OverridingAmbiguity.cpp
#include "NamespaceMath.h"
#include "NamespaceOverriding2.h"
void s() {
    using namespace Math;
    using namespace Calculation;
    // Wszystko jest w porządku, aż do instrukcji:
    //! divide(1, 2); // Dwuznaczność
}
int main() {} ///:-
```

A zatem możliwe jest użycie dyrektyw **using** w taki sposób, by wprowadzić pewną liczbę przestrzeni nazw, zawierających kolidujące ze sobą nazwy, nawet nie wywołując dwuznaczności.

Deklaracja **using**

Można wprowadzać do bieżącego zasięgu po jednej nazwie, wykorzystując **deklarację using**. W odróżnieniu od dyrektywy **using**, traktującej nazwy w taki sposób, jakby były one zadeklarowane globalnie w stosunku do zasięgu, deklaracja **using** jest deklaracją w obrębie bieżącego zasięgu. Oznacza to, że może ona zasłonić nazwy pochodzące z dyrektywy **using**:

```
//: C10:UsingDeclaration.h
#ifndef USINGDECLARATION_H
#define USINGDECLARATION_H
namespace U {
    inline void f() {}
    inline void g() {}
}
namespace V {
    inline void f() {}
    inline void g() {}
}
```

```
#endif // USINGDECLARATION_H III-
```

II: C10:UsingDeclaration1.cpp

```
#include "UsingDeclaration.h"
void h() {
    using namespace U; // Dyrektywa using
    using V::f; // Deklaracja using
    f(); // Wywołanie V::f();
    U::f(); // Nazwa funkcji musi być w pełni określona
}
int main() {} //:-~
```

Deklaracja **using** podaje w pełni określona nazwę identyfikatora, nie zawiera jednak żadnych informacji o typie. Oznacza to, że w przypadku gdy przestrzeń nazw zawiera zbiór przeciążonych funkcji o tych samych nazwach, deklaracja **using** obejmuje wszystkie funkcje należące do przeciążonego zbioru.

Można umieścić deklarację **using** w każdym miejscu, w którym może wystąpić zwykła deklaracja. Deklaracja **using** zachowuje się w taki sam sposób, jak każda inna deklaracja, z jednym wyjątkiem — ponieważ nie jest w niej podawana lista argumentów, za pomocą deklaracji **using** możliwe jest przeciążenie funkcji o identycznych typach argumentów (przeciwnie niż w przypadku zwykłego przeciążenia). Jednak ta dwuznaczność nie ujawnia się aż do miejsca użycia takiej funkcji (a nie miejsca jej deklaracji).

Deklaracja **using** może wystąpić również w przestrzeni nazw, gdzie jej znaczenie jest takie samo, jak w każdym innym miejscu — definiuje ona w tej przestrzeni nazwę:

```
//: C10:UsingDeclaration2.cpp
#include "UsingDeclaration.h"
namespace Q {
    using U::f;
    using V::g;
    // ...
}
void m() {
    using namespace Q;
    f(); // Wywołanie U::f();
    g(); // Wywołanie V::g();
}
int main() {} //:-~
```

Deklaracja **using** stanowi synonim — pozwala ona na deklarację tej samej funkcji w różnych przestrzeniach nazw. Jeżeli doprowadzi to do wielokrotnej deklaracji **tej** samej funkcji podczas importowania różnych przestrzeni nazw, nie wystąpi problem — nie pojawią się żadne dwuznaczności ani zwielokrotnienia.

Wykorzystywanie przestrzeni nazw

Niektóre z przedstawionych powyżej reguł mogą wydać się początkowo nieco zniechęcające — zwłaszcza gdy odniosłeś wrażenie, że będziesz używać ich **wszystkich** jednocześnie. Jednakna ogólna można sobie poradzić, wykorzystując przestrzenie **nazw** w bardzo prosty sposób — pod warunkiem, że wiesz, jak one działają. Należy **przez**

wszystkim pamiętać, że wprowadzając globalną dyrektywę **using** (umieszczając, poza jakimkolwiek zasięgiem, „**using namespace**”) udostępnia się temu plikowi przestrzeń nazw. Działa to zwykle bez zarzutu w przypadku plików zawierających implementację (plików „**.cpp**”), ponieważ dyrektywa **using** obowiązuje tylko do zakończenia ich komplikacji. Oznacza to, że nie ma ona wpływu na inne pliki, dzięki czemu można dostosować zarządzanie przestrzeniami nazw dla każdego pliku z osobna. Jeśli na przykład odkryjesz kolizję nazw, spowodowaną użyciem zbyt wielu dyrektyw **using** w jakimś konkretnym pliku implementacyjnym, to łatwo można wprowadzić w tym pliku zmiany — w taki sposób, aby wykorzystywał on **pełne**, jawnie nazwy lub deklaracje **using** — bez potrzeby modyfikacji innych plików implementacyjnych.

Inną kwestią są pliki nagłówkowe. Właściwie nigdy nie umieścisz globalnych dyrektyw **using** w plikach nagłówkowych, ponieważ oznaczałoby to, że we wszystkich plikach, do których zostałyby dołączone te pliki, udostępnione byłyby również określone za pomocą tych dyrektyw przestrzenie nazw (a pliki nagłówkowe mogą być dołączane również do innych takich plików).

Tak więc w plikach nagłówkowych należy używać albo jawnych, pełnych kwalifikatorów dyrektyw **using**, umieszczonych wewnątrz zasięgów, albo też deklaracji **using**. Jest to praktyka, która występuje w tej książce. Postępując zgodnie z nią, nie „zasmiecis” globalnej przestrzeni nazw, co spowodowałoby powrót do świata C++ sprzed powstania przestrzeni nazw.

Statyczne składowe w C++

Zdarzają się sytuacje, w których potrzebny jest pojedynczy obszar pamięci, wykorzystywany przez wszystkie obiekty klasy. W języku C można by użyć w takim przypadku zmiennej globalnej, ale nie jest to zbyt bezpieczne. Dane globalne mogą być używane przez każdego, a ich nazwy niekiedy kolidują z innymi nazwami, występującymi w dużych projektach. W idealnej sytuacji dane mogły być przechowywane w taki sposób, jakby były globalne, pozostając równocześnie ukryte wewnątrz klasy i przejrzyście z tą klasą powiązane.

Uzyskuje się to za pomocą statycznych danych składowych, zawartych w **klassach**. Niezależnie od tego, ile obiektów danej klasy zostanie utworzonych, każda jej składowa statyczna zajmuje pojedynczy obszar pamięci. Wszystkie obiekty dzielą między siebie ten sam obszar pamięci, przechowujący statyczne dane składowe; umożliwiają więc one „komunikację” pomiędzy obiektami. Dane statyczne należą jednak do klasy — ich nazwy znajdują się w zasięgu klasy i mogą być publiczne, prywatne lub chronione.

Definiowanie pamięci ja statycznych danych składowych

Ponieważ dane statyczne zajmują pojedynczy obszar pamięci, niezależnie od tego, ile utworzono obiektów klasy, pamięć ta musi zostać zdefiniowana w jednym miejscu.

Kompilator nie przydzieli ci tej pamięci, a program łączący zgłosi błąd, w przypadku gdy statyczne dane składowe zostaną zadeklarowane, ale nie będą zdefiniowane.

Definicja musi wystąpić na zewnątrz klasy (wstawianie jej do klasy nie jest możliwe) i dopuszczalne jest tylko jedno jej wystąpienie. Tak więc typowe jest umieszczenie definicji w pliku, zawierającym implementację klasy. Składnia definicji sprawia niektórym kłopoty, ale jest w rzeczywistości dość logiczna. Jako przykład utworzymy **statyczną składową**, wewnątrz klasy — tak jak to przedstawiono poniżej:

```
class A {  
    static int i;  
public:  
    //...  
};
```

Następnie trzeba zdefiniować pamięć dla statycznej składowej — w pliku zawierającym definicję klasy:

```
int A::i = 1;
```

Gdybyśmy definiowali zwykłą zmenną globalną, to wyglądałoby to następująco:

```
int i = 1;
```

Ale w naszym przypadku do określenia zmiennej **A::i** został użyty operator zasięgu oraz nazwa klasy.

Niektórzy mają kłopot ze zrozumieniem, że choć zmienna **A::i** jest prywatna, w zapisie tym znajduje się coś, co wygląda na wykonywanie na niej operacji w otwarty sposób. Czy nie oznacza to złamania mechanizmów zabezpieczających? Jest to zupełnie bezpieczne działanie — z dwóch powodów. Po pierwsze: jedynym miejscem, w którym wolno dokonać takiej inicjalizacji, jest definicja. Istotnie, gdyby składową statyczną był obiekt, posiadający konstruktor, to zamiast używania operatora = należało by wywołać ten konstruktor. Po drugie: po dokonaniu definicji składowej statycznej użytkownik nie może dokonać jej powtórnej definicji — spowodowałoby to zgłoszenie błędu przez program łączący. Ponadto twórca klasy jest zmuszony do utworzenia tej definicji, gdyż w przeciwnym wypadku kodu nie dałoby się połączyć podczas testowania. Wszystko to stanowi gwarancję, że definicja występuje tylko jeden raz i że znajduje się całkowicie w rękach twórcy klasy.

Całe wyrażenie inicjalizujące składową statyczną jest zawarte w zasięgu klasy. Świadczy o tym poniższy przykład:

```
//: C10:Statinit.cpp  
// Zasięg inicjatora składowej statycznej  
#include <iostream>  
using namespace std;  
  
int x = 100;  
  
class WithStatic {  
    static int x;  
    static int y;  
public:
```

```
void print() const {
    cout << "WithStatic::x = " << x << endl;
    cout << "WithStatic::y = " << y << endl;
}
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x NIE ::x

int main() {
    WithStatic ws;
    ws.print();
} //:~
```

W tym przypadku kwalifikator **WithStatic::** rozciąga zasięg klasy **WithStatic** na całą definicję.

Inicjalizacja tablicy statycznej

W rozdziale 8. zostały wprowadzone stałe statyczne (**static const**), pozwalające na zdefiniowanie wewnętrz ciała klasy stałych wartości. Możliwe jest również utworzenie tablic obiektów statycznych — zarówno stałych, jak i niebędących stałymi. Składnia jest w tym przypadku dość spójna:

```
//: C10:StaticArray.cpp
// Inicjalizacja statycznych tablic w klasach
class Values {
    // Stałe statyczne są inicjalizowane na miejscu:
    static const int scSize = 100;
    static const long scLong = 100;
    // W przypadku tablic statycznych działa automatyczne
    // zliczanie. Tablice i składowe statyczne niecałkowite
    // oraz niebędące stałymi muszą być
    // inicjalizowane zewnętrznie.
    static const int scInts[];
    static const long scLongs[];
    static const float scTable[];
    static const char scLetters[];
    static int size;
    static const float scFloat;
    static float table[];
    static char letters[];
};

int Values::size = 100;
const float Values::scFloat = 1.1;

const int Values::scInts[] = {
    99. 47. 33. 11. 7
};

const long Values::scLongs[] = {
    99. 47, 33. 11. 7
};
```

```
const float Values::scTable[] = {
    1.1, 2.2, 3.3, 4.4
};

const char Values::scLetters[] = {
    'a', 'b', 'c', 'd', 'e',
    'f', 'g', 'h', 'i', 'j'
};

floatValues::table[4] = {
    1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
    'a', 'b', 'c', 'd', 'e',
    'f', 'g', 'h', 'i', 'j'
};

int main() { Values v; } //:-
```

W przypadku statycznych stałych, całkowitych typów, definicji można dokonać wewnątrz klas. Jednakże w każdym innym przypadku (wliczając w to tablice typów całkowitych, nawet gdy są one statycznymi stałymi) należy dostarczyć pojedynczą, zewnętrzną definicję składowej. Definicje takie podlegają łączeniu wewnętrznemu, dlatego też mogą one zostać umieszczone w plikach nagłówkowych. Składnia inicjalizacji statycznych tablic jest taka sama, jak w przypadku każdego innego agregatu, w tym automatycznego zliczania.

Można również utworzyć statyczne stałe obiekty klas, a także tablice takich obiektów. Nie można **ich** jednak inicjalizować „wewnątrz **klasy**”, co jest dopuszczalne tylko w przypadku statycznych stałych, wbudowanych typów całkowitych:

```
//: C10:StaticObjectArrays.cpp
// Statyczne tablice obiektów klas
class X {
    int i;
public:
    X(int ii) : i(ii) {}
};

class Stat {
    // To nie działa, ale być może
    // chciałbyś tak właśnie zrobić:
    // static const X x(100);
    // Zarówno stałe statyczne obiekty klas, jak
    // takie, które nie są stałymi, muszą być
    // zainicjalizowane zewnętrznie:
    static X x2;
    static X xTable2[];
    static const X x3;
    static const X xTable3[];
};

X Stat::x2(100);

X Stat::xTable2[] = {
```

```
X(1).X(2).X(3).X(4)  
};  
  
const X Stat::x3(100);  
  
const X Stat::xTable3[] = {  
    X(1), X(2). X(3). X(4)  
};  
  
int main() { Stat v; } //:-~
```

Inicjalizacje zarówno stałych tablic obiektów klas, jak i tablic niebędących stałymi muszą być przeprowadzone w ten sam sposób, zgodnie z typową składnią **static**.

Klasy zagnieżdżone i klasy lokalne

Można w łatwy sposób umieścić statyczne dane składowe w klasach, które są zagnieżdżone w innych klasach. Sposób ich definiowania stanowi dość intuicyjne i oczywiste rozszerzenie — określa się po prostu dodatkowy poziom zasięgu. Jednak statyczne dane składowe nie mogą występować w klasach lokalnych (takich, które zostały zdefiniowane wewnątrz funkcji). Ilustruje to poniższy przykład:

```
//: C10:Local.cpp  
// Składowe statyczne i klasy lokalne  
#include <iostream>  
using namespace std;  
  
// Zagnieżdżona klasa MOŻE posiadać statyczne  
// dane składowe:  
class Outer {  
    class Inner {  
        static int i; // W porządku  
    };  
};  
  
int Outer::Inner::i = 47;  
  
// Klasa lokalna nie może posiadać statycznych  
// danych składowych:  
void f() {  
    class Local {  
    public:  
        //! static int i; // Błąd  
        // (Jak zostałoby zdefiniowane i?)  
    } x;  
}  
  
int main() { Outer x; f(); } //:-~
```

Nie ulega wątpliwości, na czym polega problem ze statyczną składową klasy lokalnej: **w jaki sposób odwołać się do składowej, zawartej w zasięgu pliku, by ją zdefiniować?** Klasy lokalne są w praktyce używane bardzo rzadko.

Statyczne funkcje składowe

Można również utworzyć statyczne funkcje składowe, które — podobnie jak statyczne dane składowe — działają w stosunku do klasy jako całości, a nie dla poszczególnych jej obiektów. Zamiast definiować funkcję **globalną**, znajdująca się w globalnej lub lokalnej przestrzeni nazw (i „zaśmiecającą” tę przestrzeń), można umieścić taką funkcję wewnątrz klasy. Tworząc **statyczną składową**, określa się ją powiązanie z określona klasą.

Statyczna funkcja składowa może być wywołana w zwykły sposób — za pomocą kropki lub strzałki w celu powiązania jej z jakimś obiektem. Częściej używa się jednak wywołania samej funkcji składowej, bez określenia żadnego elementu. Wykorzystuje się w tym celu operator zasięgu, jak w poniższym przykładzie:

```
//: C10:SimpleStaticMemberFunction.cpp
class X {
public:
    static void f(){}
};

int main() {
    X::f();
} //:-
```

Widząc wewnątrz klasy statyczną funkcję **składową**, należy pamiętać o tym, że projektant dążył do tego, aby była ona, w ogólnym sensie, połączona pojęciowo z tą klasą.

Statyczne funkcje składowe nie mają dostępu do zwykłych danych składowych klasy, a wyłącznie do jej danych statycznych. Mogą one również wywoływać tylko statyczne funkcje składowe. Zwykle adres bieżącego obiektu (**this**) jest niejawnie przekazywany podczas wywoływania każdej funkcji składowej klasy. Jednakże adres ten nie jest przekazywany statycznym funkcjom składowym i dlatego nie mają one dostępu do zwykłych składowych klasy. W ten sposób uzyskuje się niewielkie zwiększenie szybkości — do poziomu funkcji globalnej — ponieważ funkcja, będąca składową statyczną, nie jest obarczona dodatkowym narzutem związany z przekazywaniem parametru **this**. Równocześnie osiąga się korzyść, polegającą na zamknięciu funkcji w klasie.

W przypadku danych składowych słowo kluczowe **static** oznacza, że dla wszystkich obiektów klasy istnieje tylko jeden obszar pamięci, zawierający statyczne dane składowe. Zachodzi tu analogia do użycia słowa kluczowego **static** w celu zdefiniowania obiektu znajdującego się *wewnątrz* funkcji. Oznacza on, że dla wszystkich wywołań tej funkcji używana jest tylko jedna kopia zmiennej lokalnej.

Poniższy przykład prezentuje użycie zarówno statycznych danych składowych, jak i statycznych funkcji składowych:

```
//: C10:StaticMemberFunctions.cpp
class X {
    int i;
    static int j;
public:
    X(int ii = 0) : i(ii) {
        // Funkcje składowe, nie będące funkcjami
```

```

    // statycznymi, mają dostęp do statycznych
    // funkcji składowych oraz statycznych danych:
    j - i:
}
int val() const { return i; }
static int incr() {
    //! i++; // Błąd - statyczna funkcja składowa
    // nie ma dostępu do danych składowych, które
    // nie są statyczne
    return ++j;
}
static int f() {
    //! val(): // Błąd - statyczna funkcja składowa
    // nie ma dostępu do funkcji składowej, która
    // nie jest funkcją statyczną
    return incr(); // W porządku - wywołanie funkcji statycznej
};

int X::j = 0;

int main() {
    X x;
    X* xp = &x;
    x.f();
    xp->f();
    X::f(); // Działa tylko ze składowymi statycznymi
} //-

```

Ponieważ statyczne funkcje składowe nie posiadają wskaźnika this, nie mają one dostępu ani do danych składowych niebędących danymi statycznymi, ani do funkcji składowych niebędących funkcjami statycznymi.

Jak widać w funkcji **main()**, składowa statyczna może być wybrana zarówno przy użyciu zapisu zawierającego kropkę lub strzałkę, związującą tę funkcję z obiektem, jak i bez obiektu (ponieważ składowa statyczna jest związana z klasą, a nie z konkretnym obiektem) — za pomocą nazwy klasy i operatora zasięgu.

Jeszcze jedno interesujące spostrzeżenie. Z uwagi na sposób inicjalizacji, który następuje w przypadku składowych statycznych, można umieścić *wewnątrz* klasy statyczną składową o tym samym typie co klasa. Poniżej zamieszczono przykład, w którym został utworzony tylko jeden obiekt typu Egg dzięki uczynieniu konstruktora prywatnym. Obiekt ten jest dostępny, ale nie można utworzyć żadnego nowego obiektu klasy Egg:

```

//: C10Singleton.cpp
// Statyczna składowa tego samego typu gwarantuje
// że istnieje tylko jeden obiekt tego typu.
// Jest to również nazywane wzorcem "singla".
#include <iostream>
using namespace std;

class Egg {
    static Egg e;
    int i;
    Egg(int ii) : i(ii) {}
}

```

```
Egg(const Egg&); // Blokada konstruktora kopującego
public:
    static Egg* instance() { return &e; }
    int val() const { return i; }
};

Egg Egg::e(47);

int main() {
//! Egg x(1); // Błąd - nie można utworzyć obiektu klasy Egg
// Można odwoływać się tylko do jej pojedynczego egzemplarza:
cout << Egg::instance()->val() << endl;
} //:~
```

Inicjalizacja składowej `e` odbywa się po zakończeniu deklaracji, dzięki czemu kompilator posiada wszelkie informacje, niezbędne do przydzielenia obiektowi pamięci i wywołania jego konstruktora.

Aby skutecznie zapobiec utworzeniu jakiegokolwiek innego obiektu, zostało tu dodane coś jeszcze — drugi prywatny konstruktor, nazywany *konstruktorem kopującym* (ang. *copy-constructor*). Zostanie on wprowadzony dopiero w następnym rozdziale. Jednak, tytułem zapowiedzi, warto nadmienić, że w razie usunięcia konstruktora kopującego, zdefiniowanego w powyższym przykładzie, można by utworzyć obiekty klasy `Egg`, używając następujących sposobów:

```
Egg e = *Egg::instance();
Egg e2(*Egg::instance());
```

W obu sposobach został wykorzystany konstruktor kopujący. Aby zatem zablokować taką możliwość, konstruktor kopujący zadeklarowano jako prywatny (nie jest potrzebna żadna jego definicja, ponieważ nigdy nie zostanie on wywołany). Dyskusja dotycząca konstruktora kopującego zajmuje znaczną część następnego rozdziału, więc po przeczytaniu go wszystko powinno stać się zupełnie jasne.

Zależności przy inicjalizacji obiektów statycznych

Gwarantuje się, że w obrębie konkretnej jednostki translacji kolejność inicjalizacji obiektów statycznych jest zgodna z kolejnością występowania definicji tych obiektów w jednostce translacji. Jest również pewne, że kolejność niszczenia obiektów jest odwrotna do kolejności ich inicjalizacji.

Nie istnieją jednak żadne gwarancje dotyczące kolejności inicjalizacji obiektów statycznych, znajdujących się w *różnych* jednostkach translacji, a język nie dostarcza żadnych środków, które umożliwiłyby określenie tej kolejności. Może to stwarzać poważne problemy. Poniżej przedstawiono przykład sytuacji, prowadzącej do natychmiastowej katastrofy (zatrzymanie prostych systemów operacyjnych oraz przerwanie działania procesu w przypadku bardziej wyrafinowanych). Jeżeli jeden z plików będzie miał **następującą zawartość**:

```
// Pierwszy plik
#include <fstream>
std::ofstream out("out.txt");
```

a w drugim pliku obiekt out wykorzystywany będzie w jednym z inicjatorów:

```
// Drugi plik
#include <fstream>
extern std::ofstream out;
class Oof {
public:
    Oof() { std::out << "ojej"; }
} oof;
```

to program taki może działać albo nie. Jeżeli środowisko programistyczne zbuduje program w taki sposób, że najpierw zostanie zainicjowany pierwszy plik, to nie wystąpią problemy. Jeżeli jednak kolejność inicjalizacji będzie odwrotna, to konstruktor klasy Oof, zakładający istnienie obiektu out, który nie został jeszcze utworzony, spowoduje chaos.

Problem ten występuje tylko w przypadku inicjalizacji obiektów statycznych, które są wzajemnie od siebie *uzależnione*. Obiekty statyczne umieszczone w jednostce translacji są **inicjalizowane** przed pierwszym wywołaniem funkcji znajdującej się w tej jednostce — ale może to nastąpić nawet po zakończeniu funkcji **main()**. Nie można mieć pewności co do kolejności inicjalizacji obiektów statycznych, jeżeli znajdują się one w różnych plikach.

Bardziej wyrafinowany przykład takiej sytuacji można znaleźć w książce „*The Annotated C++ Reference Manual*”². W jednym z plików, w zasięgu globalnym, znajdują się następujące wiersze:

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

a w drugim, również w zasięgu globalnym — takie:

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

W przypadku wszystkich obiektów statycznych mechanizm odpowiedzialny za łączenie i ładowanie programu zapewnia, że zanim zostanie wykonana dynamiczna inicjalizacja, określona przez programistę, przeprowadzana jest inicjalizacja statyczna, polegająca na wyzerowania danych. W przedstawionym poprzednio przykładzie wyzerowanie pamięci zajmowanej przez obiekt out typu **fstream** nie ma żadnego szczególnego znaczenia, więc w rzeczywistości obiekt ten pozostaje niezdefiniowany do momentu wywołania konstruktora. Jednakże w przypadku typów wbudowanych inicjalizacja polegająca na wyzerowaniu danych ma sens. Jeżeli pliki są inicjalizowane w takiej kolejności, **w jakiej przedstawiono je powyżej**, to zmienna y jest najpierw statycznie inicjalizowana wartością zerową. W związku z tym zmienna x przyjmuje wartość jeden, a następnie zmienna y jest dynamicznie inicjalizowana wartością dwa. Jeżeli jednak pliki są inicjalizowane w odwrotnej kolejności, to najpierw zmienna x jest statycznie inicjalizowana wartością zerową, następnie zmienna y jest dynamicznie inicjalizowana wartością jeden, a na koniec x uzyskuje wartość równą dwa.

Programista musi zdawać sobie sprawę z takich sytuacji, ponieważ możliwe jest napisanie programu zawierającego zależności związane z inicjalizacją zmiennych statycznych, który będzie działał na jednej platformie, a przeniesiony do innego środowiska komplikacji — niespodziewanie przestanie działać.

ak można temu zaradzić?

Istnieją trzy sposoby, które umożliwiają ci poradzenie sobie z tym problemem:

1. Nie rób tego. Unikanie zależności, związanych ze statyczną inicjalizacją obiektów, stanowi najlepsze rozwiązanie.
2. Jeżeli to konieczne, umieść krytyczne definicje obiektów statycznych w pojedynczym pliku, co pozwoli na przenośny sposób kontroli nad ich **inicjalizacją**, polegający na uporządkowaniu ich we właściwej kolejności.
3. Jeżeli jesteś przeświadczony, że rozproszenie obiektów statycznych pomiędzy jednostki translacji jest nieuniknione — jak w przypadku biblioteki, gdzie nie ma możliwości nadzorowania używającego jej programisty — to masz do dyspozycji dwie techniki programistyczne rozwiązuające ten problem.

Pierwsza technika

Technika ta została opracowana przez Jerry'ego Schwartza podczas tworzenia przez niego biblioteki `iostream` (ponieważ obiekty `cin`, `cout` oraz `cerr` są zdefiniowane jako statyczne i znajdują się w odrębnych plikach). Jest to w rzeczywistości technika gorsza niż druga z zaprezentowanych, ale istnieje ona już dugo, dzięki czemu można natknąć wykorzystującą kod. Warto zatem zrozumieć, w jaki sposób działa.

Technika ta wymaga utworzenia dodatkowej klasy w pliku nagłówkowym biblioteki. Klasa ta jest odpowiedzialna za dynamiczną inicjalizację obiektów statycznych wchodzących w skład biblioteki. Poniżej znajduje się prosty przykład:

```
//: C10:Initializer.h
// Technika inicjalizacji obiektów statycznych
#ifndef INITIALIZER_H
#define INITIALIZER_H
#include <iostream>
extern int x; // Deklaracje, a nie definicje
extern int y;

class Initializer {
    static int initCount;
public:
    Initializer() {
        std::cout << "Initializer()" << std::endl;
        // Inicjalizacją tylko za pierwszym razem
        if(initCount++ == 0) {
            std::cout << "inicjalizacją"
                << std::endl;
            x = 100;
            y = 200;
        }
    }
};
```

```

        }
    }
-Initializer() {
    std::cout << "~Initializer()" << std::endl;
    // Sprzątanie tylko za ostatnim razem
    if(--initCount == 0) {
        std::cout << "sprzątanie"
        << std::endl;
        // Wszelkie niezbędne czynności związane ze sprzątaniem
    }
}
};

// Poniższa definicja tworzy obiekt,
// w każdym pliku, do którego jest dołączony
// plik Initializer.h, ale obiekt ten jest widoczny
// tylko w obrębie bieżącego pliku:
static Initializer init;
#endif // INITIALIZER_H //:~
```

Deklaracje zmiennych x oraz y **zapowiadają jedynie**, że obiekty te **istnieją**, ale nie powodują przydzielenia im pamięci. Jednak definicja obiektu **init** typu **Initializer** przydziela temu obiekowi pamięć — w każdym pliku, do którego został dołączony plik nagłówkowy. Jednakże z uwagi na to, że nazwa ta jest statyczna (słowo to decyduje tym razem o widoczności, a nie o sposobie przydzielenia pamięci — jest ona przydzielana domyślnie w obrębie pliku), jest ona widoczna tylko w obrębie bieżącej jednostki translacji, dzięki czemu program łączący nie zgłasza błędu wielokrotnej definicji.

Poniżej przedstawiono zawartość pliku zawierającego definicje zmiennych x, y oraz **initCount**:

```
//: C10:InitializerDefs.cpp {0}
// Definicje dla pliku Initializer.h
#include "Initializer.h"
// Statyczna inicjalizacja spowoduje
// wyzerowanie wartości poniższych zmiennych:
int x;
int y;
int Initializer::initCount;
//:~
```

(oczywiście, dołączenie pliku nagłówkowego spowoduje również utworzenie w tym pliku, statycznego względem niego, egzemplarza obiektu **init**). Założymy, że dwa pozostałe pliki zostały utworzone przez użytkownika biblioteki:

```
//: C10:Initializer.cpp {0}
// Inicjalizacja obiektów statycznych
#include "Initializer.h"
//:~

i
//: C10:Initializer2.cpp
//{L} InitializerDefs Initializer
// Inicjalizacja obiektów statycznych
#include "Initializer.h"
using namespace std;
```

```
int main() {
    cout << "wewnątrz funkcji main()" << endl;
    cout << "opuszczanie funkcji main()" << endl;
} //:-
```

Nie ma znaczenia, która jednostka translacji zostanie zainicjowana jako pierwsza. Gdy po raz pierwszy zostanie zainicjowana jednostka translacji zawierająca plik **Initializer.h**, wartość składowej **initCount** będzie wynosiła zero, więc zostanie wykonana inicjalizacja (zależy to w istotny sposób od faktu, że obszar danych statycznych jest zerowany przed wykonaniem jakiekolwiek dynamicznej inicjalizacji). We wszystkich pozostałych jednostkach translacji wartość składowej **initCount** będzie niezerowa, w związku z czym inicjalizacja zostanie pominięta. Sprzątanie odbywa się w odwrotnej kolejności, a destruktor **~Initializer()** gwarantuje, że zostanie ono wykonane tylko jeden raz.

W powyższym przykładzie wykorzystano globalne statyczne obiekty wbudowanych typów. Technika ta działa również w stosunku do klas, lecz w takim przypadku ich obiekty muszą zostać zainicjowane dynamicznie przez klasę **Initializer**. Jednym ze sposobów, umożliwiających wykonanie tego zadania, jest utworzenie klas pozbawionych konstruktorów i destruktorów, posiadających zamiast nich funkcje inicjalizujące i sprzątające o innych nazwach. Częściej spotykanym rozwiązaniem jest jednak zdefiniowanie wskaźników do obiektów i utworzenie tych obiektów wewnątrz konstruktora **Initializer()** za pomocą operatora **new**.

Druga technika

Kiedy pierwsza z wymienionych technik znajdowała się już od dłuższego czasu w użyciu, ktoś (nie wiem jednak, kto to uczynił) przedstawił technikę opisaną w tym podrozdziale, znacznie prostszą bardziej przejrzystą od poprzedniej. Fakt, że jej odkrycie zajęło tyle czasu, stanowi rodzaj hołdu oddanego złożoności języka C++.

W technice tej wykorzystano fakt, że obiekty statyczne, znajdujące się wewnątrz funkcji, są inicjalizowane pierwszy (i jedyny) raz podczas wywołania tej funkcji. Należy pamiętać, że problem, który staramy się rozwiązać, nie polega na tym, *kiedy* inicjalizowane są obiekty statyczne (można tym sterować oddziennie), tylko na zapewnieniu właściwej kolejności inicjalizacji.

Druga technika jest bardzo przemyślana i elegancka. W przypadku każdej zależności dotyczącej inicjalizacji, obiekt statyczny jest umieszczany wewnątrz funkcji, zwracającej referencję do tego obiektu. Dzięki temu jedynym sposobem uzyskania dostępu do statycznego obiektu jest wywołanie funkcji, a w przypadku gdy obiekt ten musi odwołać się do innego obiektu, od którego zależy, musi wywołać funkcję *tego* obiektu. Kiedy funkcja jest wywoływana po raz pierwszy, następuje inicjalizacja obiektu. Kolejność inicjalizacji obiektów statycznych jest na pewno właściwa, ponieważ zależy ona od projektu kodu, a nie od kolejności, wyznaczonej w dowolny sposób przez program łączący.

Jako przykład posługują dwie klasy, wzajemnie od siebie zależne. Pierwsza z nich zawiera składową typu **bool**, inicjalizowaną wyłącznie przez konstruktor, dzięki czemu, w przypadku statycznego obiektu tej klasy, można określić, czy konstruktor był już

wywoływany (obszar danych statycznych jest inicjalizowany wartością zerową w trakcie uruchamiania programu; jeśli zatem konstruktor nie był jeszcze wywoływany, składowa typu bool zawiera wartość false):

```
//: C10:Dependency1.h
#ifndef DEPENDENCY1_H
#define DEPENDENCY1_H
#include <iostream>

class Dependency1 {
    bool init;
public:
    Dependency1() : init(true) {
        std::cout << "konstrukcja Dependency1"
        << std::endl;
    }
    void print() const {
        std::cout << "Dependency1 init: "
        << init << std::endl;
    }
};

#endif // DEPENDENCY1_H //:-
```

Konstruktor informuje, że został wywołany, a ponadto stanu obiektu można wydrukować za pomocą funkcji `print()`, co umożliwia sprawdzenie, czy obiekt został zainicjowany.

Druga z klas jest inicjalizowana za pomocą obiektu pierwszej klasy, co powoduje powstanie zależności:

```
//: C10:Dependency2.h
#ifndef DEPENDENCY2_H
#define DEPENDENCY2_H
#include "Dependency1.h"

class Dependency2 {
    Dependency1 dl;
public:
    Dependency2(const Dependency1& dep1) : dl(dep1){
        std::cout << "konstrukcja Dependency2 ";
        print();
    }
    void print() const { dl.print(); }
};

#endif // DEPENDENCY2_H //:-~
```

Konstruktor tej klasy informuje, że został wywołany, oraz drukuje stan obiektu `dl`. Dzięki temu można sprawdzić, czy obiekt ten został już zainicjowany, zanim został wywołany konstruktor.

Aby zademonstrować, na czym mogą polegać problemy, statyczne definicje obiektów zostały najpierw umieszczone w poniższym pliku w nieprawidłowej kolejności, co odpowiada sytuacji, w której program łączący zainicjowałby obiekt **Dependency2** przed inicjalizacją obiektu **Dependency1**. Następnie kolejność ta jest odwrotna, dzięki czemu można zobaczyć, jak to działa w przypadku prawidłowej kolejności inicjalizacji. Na koniec została zaprezentowana druga technika.

Dla zwiększenia czytelności wyświetlanych wyników została utworzona funkcja **separator()**. Sztuczka polega na tym, że nie można globalnie wywołać funkcji, chyba że jest ona używana do inicjalizacji zmiennej. Funkcja **separator()** zwraca więc fikcyjną wartość, używaną do inicjalizacji pary zmiennych globalnych.

```
//: C10:Technique2.cpp
#include "Dependency2.h"
using namespace std;

// Funkcja zwraca wartość, dzięki czemu
// może zostać wywołana jako globalny inicjator:
int separator() {
    cout << "....." << endl;
    return 1;
}

// Symulacja problemu zależności:
extern Dependency1 dep1;
Dependency2 dep2(dep1);
Dependency1 dep1;
int x1 = separator();

// Ale w tej kolejności wszystko jest w porządku:
Dependency1 dep1b;
Dependency2 dep2b(dep1b);
int x2 = separator();

// Umieszczenie statycznych obiektów w funkcjach
// zapewnia prawidłowe działanie
Dependency1& d1() {
    static Dependency1 dep1;
    return dep1;
}

Dependency2& d2() {
    static Dependency2 dep2(d1());
    return dep2;
}

int main() {
    Dependency2& dep2 = d2();
} //:-
```

Funkcje **d1()** i **d2()** zawierają statyczne egzemplarze obiektów klas **Dependency1** i **Dependency2**. Obecnie jedynym sposobem odwołania się do tych obiektów jest wywołanie funkcji, które podczas pierwszego wywołania **inicjalizują** zawarte w sobie obiekty statyczne. Dzięki temu gwarantowana jest poprawność inicjalizacji, o czym można się przekonać po uruchomieniu programu i obejrzeniu generowanych przez niego wyników.

A **oto jak** powinno się przygotować kod w celu zastosowania tej techniki. Zwykle obiekty statyczne zostałyby zdefiniowane w oddzielnego plikach (ponieważ jesteśmy z jakichś powodów do tego zmuszeni — należy pamiętać, że to właściwe definiowanie statycznych obiektów w różnych plikach powoduje problemy), więc zamiast nich zdefiniujemy w oddzielnego plikach funkcje zawierające te obiekty. Muszą one być jednak zadeklarowane w plikach nagłówkowych:

```
//: C10:Dependency1StatFun.h
#ifndef DEPENDENCY1STATFUN_H
#define DEPENDENCY1STATFUN_H
#include "Dependency1.h"
extern Dependency1& d1();
#endif // DEPENDENCY1STATFUN_H ///:-
```

W rzeczywistości **specyfikator extern** jest niepotrzebny w przypadku deklaracji funkcji. A oto drugi plik nagłówkowy:

```
//: C10:Dependency2StatFun.h
#ifndef DEPENDENCY2STATFUN_H
#define DEPENDENCY2STATFUN_H
#include "Dependency2.h"
extern Dependency2& d2();
#endif // DEPENDENCY2STATFUN_H ///:-
```

Następnie w plikach implementacyjnych, w których poprzednio zostałyby umieszczone definicje obiektów statycznych, umieszczamy definicje **funkcji**, zawierających te obiekty:

```
//: C10:Dependency1StatFun.cpp {0}
#include "Dependency1StatFun.h"
Dependency1& d1() {
    static Dependency1 dep1;
    return dep1;
} ///:-
```

Przypuszczalnie w tym pliku mógłby znajdować się jeszcze jakiś inny kod. Tak natomiast przedstawia się drugi z plików:

```
//: C10:Dependency2StatFun.cpp {0}
#include "Dependency1StatFun.h"
#include "Dependency2StatFun.h"
Dependency2& d2() {
    static Dependency2 dep2(d1());
    return dep2;
} ///:-
```

Tak więc mamy obecnie dwa pliki, które mogłyby zostać połączone w dowolnej kolejności. W przypadku gdyby zawierały one zwykłe obiekty statyczne, spowodowałoby to dowolną kolejność inicjalizacji znajdujących się w nich obiektów. Ponieważ jednak znajdują się w nich funkcje zawierające te obiekty, nie mają już niebezpieczeństwa związanego z ich niewłaściwą inicjalizacją:

```
//: C10:Technique2b.cpp
//{L} Dependency1StatFun Dependency2StatFun
#include "Dependency2StatFun.h"
int main() { d2(); } ///:-
```

Po uruchomieniu tego programu widać, że inicjalizacją statycznego obiektu klasy **Dependency1** odbywa się zawsze przed inicjalizacją statycznego obiektu klasy **Dependency2**. Jest to znacznie prostsze podejście niż prezentowane w przypadku pierwszej techniki.

Być może zechcesz umieścić funkcje **d1()** i **d2()** jako funkcje inline wewnątrz swoich plików nagłówkowych. Nie należy jednak tego robić. Funkcja inline może być powielona w każdym pliku, w którym występuje, a powielenie to *obejmuje również* definicję znajdującego się w niej statycznego obiektu. Ponieważ funkcje inline *podlegają automatycznie* wewnętrznemu łączeniu, spowodowałoby to utworzenie wielu obiektów statycznych, znajdujących się w różnych jednostkach translacji, co z pewnością byłoby przyczyną kłopotów. Ponieważ trzeba zapewnić, by istniała tylko definicja każdej funkcji zawierającej obiekt statyczny, oznacza to, że nie można używać w tym celu funkcji inline.

Specyfikacja zmiany sposobu łączenia

Co się stanie, gdy w programie pisany w języku C++ zechcemy użyć biblioteki języka C? Jeżeli zadeklarujemy funkcję języka C:

```
float f(1nt a. char b);
```

to kompilator uzupełni jej nazwę, tworząc coś w rodzaju **_f_int_char** i umożliwiając przeciążanie nazwy tej funkcji (oraz łączenie bezpieczne dla typów). Jednak kompilator języka C, kompilujący tę funkcję, z całą pewnością *nie* dokonał uzupełnienia jej nazwy, więc jej wewnętrzna nazwa ma postać **_f**. Tak więc program łączący nie będzie w stanie określić odwołań do funkcji **f()**, występujących w programie napisanym w języku C++.

Mechanizmem umożliwiającym rozwiązanie tego problemu jest w języku C++ *specyfikacja zmiany sposobu łączenia* (ang. *alternate linkage specification*), która została wprowadzona do języka za pomocą przeciążenia słowa kluczowego **extern**. Po słowie kluczowym **extern** umieszczono łańcuch, który określa sposób łączenia, dotyczący znajdującej się po nim deklaracji:

```
extern "C" float f(int a. char b);
```

Powyższa deklaracja informuje kompilator, by w stosunku do funkcji **f()** używał łączenia właściwego dla języka C, dzięki czemu nie będzie on *stosował* uzupełniania jej nazwy. Jedynymi określonymi w standardzie specyfikacjami łączenia są "**C**" oraz "**C++**", ale producenci kompilatorów umożliwiają obsługę innych języków również w taki sam sposób.

Jeżeli zmiana sposobu łączenia dotyczy grupy deklaracji, można umieścić je w nawiasie klamrowym, jak pokazano poniżej:

```
extern "C" {  
    float f(int a. char b);  
    double d(int a. char b);  
}
```

W przypadku całych plików nagłówkowych można również stosować zapis:

```
extern "C" {  
    #include "Myheader.h"  
}
```

Większość producentów kompilatorów języka C++ używa specyfikacji zmiany sposobu *łączenia* w swoich plikach nagłówkowych, które działają zarówno w języku C, jak i w C++.

Podsumowanie

Słowo kluczowe **static** może być nieco mylące, ponieważ w pewnych sytuacjach decyduje ono o lokalizacji przydzielonej pamięci, a w innych — określa widoczność i sposób łączenia nazw.

Od momentu wprowadzenia przestrzeni nazw języka C++ dostępne jest lepsze i znacznie bardziej elastyczne narzędzie, umożliwiające nadzór nad tym, w jaki sposób nazwy są rozpowszechniane w dużych projektach.

Użycie słowa kluczowego **static** wewnątrz klasy jest kolejnym sposobem umożliwiającym zarządzanie nazwami w ramach programu. Nazwy takie nie kolidują z nazwami globalnymi; równocześnie są one widoczne i dostępne w obrębie programu, co zapewnia większą kontrolę nad pielegnacją kodu.

Ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Utwórz funkcję zawierającą statyczną zmienną będącą wskaźnikiem i pobierającą argument o zerowej domyślnej wartości. Jeżeli w wywołaniu funkcji zostanie podana wartość argumentu, to będzie on wskazywał początek tablicy liczb całkowitych. Jeżeli funkcja zostanie wywołana bez argumentu (użyty zostanie argument domyślny), to powinna ona zwrócić za każdym razem następną wartość, znajdującą się w tablicy, aż do natkania wartości -1 (oznaczającej koniec tablicy). Sprawdź działanie tej funkcji w funkcji **main()**.
2. Utwórz funkcję zwracającą przy każdym jej wywołaniu następną wartość ciągu Fibonacciego. Dodaj funkcji argument typu **bool**, o domyślnej wartości **false**, który w przypadku podania wartości **true** spowoduje „wyzerowanie” funkcji do początku ciągu Fibonacciego. Sprawdź działanie tej funkcji w funkcji **main()**.
3. Utwórz klasę zawierającą tablicę liczb całkowitych. Ustal wielkość tej tablicy, używając do tego celu statycznej stałej całkowitej, znajdującej się wewnątrz klasy. Dodaj stałą całkowitą i zainicjalizuj ją, wykorzystując listy inicjatorów konstruktora. Określ **konstruktor jako funkcję inline**. Dodaj statyczną zmienną całkowitą i zainicjuj ją pomocą określonej wartości. Utwórz statyczną funkcję składową, drukującą statyczne dane składowe. Utwórz składową funkcję inline, która wywołuje funkcję **print()**, drukującą wszystkie wartości zawarte w tablicy oraz wywołującą utworzoną poprzednio statyczną funkcję składową. Sprawdź działanie klasy w funkcji **main()**.
4. Utwórz klasę o nazwie **Monitor**, zapamiętującą, ile razy wywołana była jej funkcja składowa **incident()**. Dodaj funkcję składową **print()**, drukującą

liczbę wywołań funkcji **incident()**. Następnie utwórz funkcję globalną (niebędącą funkcją składową), zawierającą statyczny obiekt klasy **Monitor**. Podczas każdego wywołania funkcja ta powinna wywoływać funkcję **incident()**, a następnie funkcję **print()**, wyświetlającą liczbę wywołań. Sprawdź działanie tej funkcji w funkcji **main()**.

5. Za pomocą funkcji składowej **decrement()** zmodyfikuj opisaną w poprzednim ćwiczeniu klasę **Monitor** w taki sposób, aby mogła dekrementować wartość licznika. Utwórz klasę **Monitor2** tak, aby jej konstruktor przyjmował jako argument wskaźnik do obiektu klasy **Monitor1**, a następnie zapamiętywał ten wskaźnik i wywoływał funkcje **incident()** oraz **print()**. W destruktorze klasy **Monitor2** wywołaj natomiast funkcje **decrement()** i **print()**. Teraz utwórz, wewnątrz funkcji, statyczny obiekt klasy **Monitor2**. Wykonaj eksperymenty w funkcji **main()** — wywołaj tę funkcję, po czym jej nie wywołuj, obserwując, jaki ma to wpływ na destruktor klasy **Monitor2**.
6. Utwórz globalny obiekt klasy **Monitor2** i zobacz, co się stanie.
7. Utwórz klasę, która posiada destruktor drukujący komunikat, a następnie wywołujący funkcję **exit()**. Utwórz globalny obiekt tej klasy i zobacz, co się stanie.
8. Przeprowadź eksperymenty z kolejnością wywołań konstruktora i destruktora w programie **StaticDestructors.cpp** — wywołując funkcje **f()** i **g()**, zawarte w funkcji **main()**, w odwrotnej kolejności. Czy używany przez ciebie kompilator obsługuje to poprawnie?
9. W programie **StaticDestructors.cpp** przetestuj domyślną obsługę błędów swojej implementacji, zamieniając oryginalną definicję obiektu **out** w deklarację **extern** i umieszczając rzeczywistą definicję po definicji obiektu **a** (którego konstruktor **Obj()** przesyła informacje do obiektu **obj**). Upewnij się, że w czasie uruchamiania tego programu komputer nie robi niczego ważnego albo że jest on odporny na tego rodzaju błędy.
10. Udowodnij, że zmienne statyczne względem pliku, znajdujące się w plikach nagłówkowych, nie kolidują z sobą, gdy zostaną dołączone do więcej niż jednego pliku **cpp**.
11. Utwórz prostą klasę, zawierającą składową **całkowitą**, konstruktor, który inicjalizuje tę składowaną podstawie swojego argumentu, funkcję składową, przypisującą jej wartość swojego argumentu, oraz funkcję **print()**, drukującą wartość tej składowej. Umieść swoją klasę w pliku nagłówkowym i dołącz go do dwóch plików **cpp**. W jednym z tych plików utwórz egzemplarz klasy, a w drugim — zadeklaruj jego identyfikator za pomocą specyfikatora **extern** i przetestuj jego działanie w funkcji **main()**.
12. Uczyń egzemplarz obiektu, utworzonego w poprzednim ćwiczeniu, obiektem statycznym i sprawdź, czy uniemożliwia to znalezienie tego obiektu przez program łączący.
13. Zadeklaruj funkcję w pliku nagłówkowym. Zdefiniuj tę funkcję w jednym pliku **cpp** i wywołaj ją wewnątrz funkcji **main()**, w drugim pliku **cpp**. Skompiluj program i upewnij się, że działa. Następnie zmień definicję funkcji na statyczną i przekonaj się, że program łączący nie może jej odnaleźć.

14. Zmodyfikuj program **Volatile.cpp**, opisany w rozdziale 8., w taki sposób, aby mógł rzeczywiście działać jako procedura obsługi przerwań. Wskazówka: procedura obsługi przerwań nie pobiera żadnych argumentów.
15. Napisz i skompiluj prosty program, wykorzystujący słowa kluczowe **auto** i **register**.
16. Utwórz plik nagłówkowy, zawierający przestrzeń nazw. Wewnątrz tej przestrzeni umieść kilka deklaracji funkcji. Następnie utwórz drugi plik nagłówkowy, dołączający poprzedni plik, który kontynuuje tę przestrzeń nazw i dodaje do niej jeszcze kilka deklaracji funkcji. Potem utwórz plik **cpp**, w którym dołączony jest drugi plik nagłówkowy. Utwórz (krótszy) synonim swojej przestrzeni nazw. W definicji jednej z funkcji wywołaj jedną ze swoich funkcji, używając specyfikacji zasięgu. W definicji innej funkcji wpisz dyrektywę **using**, wprowadzając swoją przestrzeń nazw do zasięgu funkcji, a następnie pokaż, że do wywołania funkcji znajdujących się w tej przestrzeni nie jest potrzebna specyfikacja zasięgu.
17. Utwórz plik nagłówkowy zawierający **bezimienną przestrzeń nazw**. Dołącz ten plik nagłówkowy w dwóch oddzielnych plikach **cpp** i pokaż, że ta przestrzeń nazw jest unikatowa w każdej jednostce translacji.
18. Wykorzystując plik nagłówkowy z poprzedniego ćwiczenia, pokaż, że nazwy znajdujące się w bezimiennej przestrzeni nazw są automatycznie dostępne w obrębie jednostki translacji, bez konieczności używania kwalifikatorów.
19. Zmodyfikuj program **FriendInjection.cpp**, dodając definicję zaprzyjaźnionej funkcji i wywołując ją wewnątrz funkcji **main()**.
20. Używając pliku **Arithmetic.cpp** zademonstruj, że dyrektywa **using** nie wykracza poza funkcję, w której została użyta.
21. Rozwiąż problem występujący w pliku **OverridingAmbiguity.cpp**, używając najpierw specyfikacji zasięgu, a następnie za pomocą deklaracji **using**, wymuszającej na kompilatorze wybór jednej z funkcji posiadających identyczne nazwy.
22. W dwóch plikach nagłówkowych utwórz dwie przestrzenie nazw, z których każda będzie zawierała klasę (z wszystkimi definicjami **inline**) o takiej samej nazwie, jak znajdująca się w drugiej przestrzeni nazw. Utwórz plik **cpp**, w którym dołączone są oba pliki nagłówkowe. Utwórz funkcję i użyj wewnątrz niej dyrektyw **using**, wprowadzając obie przestrzenie nazw. Spróbuj utworzyć obiekt jednej z klas i zobacz, co się stanie. Uczyń dyrektywy **using** globalnymi (umieszczając je na zewnątrz funkcji) i zobacz, czy coś to zmieni. Rozwiąż problem, używając specyfikacji zasięgu, a następnie utwórz obiekty obu klas.
23. Rozwiąż problem przedstawiony w poprzednim zadaniu, używając deklaracji **using**, wymuszającej na kompilatorze wybór jednej z klas posiadających identyczne nazwy.
24. Pobierz deklaracje przestrzeni nazw z plików **BobsSuperDuperLibrary.cpp** oraz **UnnamedNamespaces.cpp** i umieśc je w oddzielnych plikach

nagłówkowych, nadając w trakcie tego procesu nazwę bezimiennej przestrzeni nazw. W trzecim pliku **nagłówkowym** utwórz za pomocą deklaracji **using** nową przestrzeń nazw, łączącą w sobie elementy zawarte w dwóch pozostałych przestrzeniach nazw. Wprowadź swoją owną przestrzeń nazw do funkcji **main()**, używając do tego dyrektywy **using**, i odwołaj się do wszystkich elementów zawartych w tej przestrzeni nazw.

25. Utwórz plik nagłówkowy, w którym dołączane są pliki nagłówkowe `<string>` i `<iostream>`, ale nie są stosowane żadne deklaracje ani definicje **using**. Użyj „strażników dołączania” — w taki sposób, jak w plikach nagłówkowych zawartych w książce. Utwórz klasę, posiadającą tylko funkcje inline, zawierającą składową typu **string**, konstruktor, inicjalizujący ją swoim argumentem, i funkcję **print()**, wyświetlającą zawartość. Utwórz plik `cpp` i sprawdź działanie tej klasy w funkcji **main()**.
26. Utwórz klasę **zawierającą składowe statyczne typów double i long**. Utwórz statyczną funkcję **składową**, drukującą ich wartości.
27. Utwórz klasę **zawierającą składową całkowitą**, konstruktor, inicjalizującą na podstawie swojego argumentu, oraz funkcję **print()**, wyświetlającą jej wartość. Następnie utwórz drugą klasę, zawierającą statyczny obiekt pierwszej klasy. Dodaj statyczną funkcję **składową**, wywołującą funkcję **print()** statycznego obiektu. Sprawdź działanie swojej klasy w funkcji **main()**.
28. Utwórz klasę **zawierającą stałą statyczną tablicę liczb całkowitych** oraz statyczną tablicę liczb całkowitych, niebędącą stałą. Utwórz statyczną funkcję **składową**, drukującą zawartość tych tablic. Sprawdź działanie swojej klasy w funkcji **main()**.
29. Utwórz klasę **zawierającą obiekt typu string**, konstruktor, inicjalizujący go wartością swojego argumentu, oraz funkcję **print()**, wyświetlającą jego wartość. Utwórz drugą klasę, zawierającą statyczne tablice obiektów pierwszej klasy — zarówno będące, jak i niebędące stałymi, a także statyczne funkcje składowe, drukujące zawartość tych tablic. Sprawdź działanie drugiej z klas w funkcji **main()**.
30. Utwórz strukturę, **zawierającą składową całkowitą**, oraz domyślny konstruktor, inicjalizujący tę **składową wartością zerową**. Uczyń tę strukturę lokalną w stosunku do funkcji. Wewnątrz tej funkcji utwórz tablicę obiektów swojej struktury i pokaż, że każda składowa została automatycznie zainicjowana wartością zerową.
31. Utwórz klasę **reprezentującą połączenie z drukarką**, pozwalającą posiadanie tylko jednej drukarki.
32. W pliku nagłówkowym utwórz klasę **Mirror**, zawierającą dwie dane składowe — wskaźnik do obiektu klasy **Mirror** oraz składową typu **bool**. Utwórz dwa **konstruktory**. Konstruktor domyślny powinien inicjalizować składową typu **bool** wartością **true**, a wskaźnik do obiektu **klasy Mirror** — wartością zerową. Drugi konstruktor powinien pobierać jako argument wskaźnik do obiektu klasy **Mirror**, a następnie przypisywać go wskaźnikowi zawartemu w klasie, natomiast składową typu **bool** — inicjalizować wartością **false**. Dodaj funkcję

składową **test()** — jeżeli wartość wskaźnika zawartego w obiekcie jest niezerowa, to funkcja ta powinna zwrócić wartość zwracaną przez funkcję **test()**, wywołaną za pośrednictwem tego wskaźnika. Jeżeli natomiast wartość wskaźnika jest zerowa, to funkcja powinna zwrócić wartość składowej typu **bool**. Następnie utwórz pięć plików **cpp** w taki sposób, by do każdego z nich został dołączony plik nagłówkowy klasy **Mirror**. W pierwszym pliku **cpp**, za pomocą domyślnego konstruktora, utwórz globalny obiekt klasy **Mirror**. W drugim pliku, za pomocą specyfikatora **extern**, zadeklaruj obiekt znajdujący się w pierwszym pliku, a także, przy użyciu konstruktora pobierającego wskaźnik do pierwszego obiektu, zdefiniuj globalny obiekt klasy **Mirror**. Kontynuuj taki sposób tworzenia kolejnych obiektów aż do ostatniego pliku, zawierającego również **globalną definicję** obiektu. W znajdującej się w tym pliku funkcji **main()** wywołaj funkcję **test()** i sprawdź wyświetlony przez tę funkcję wynik. Jeżeli wynikiem tym jest **true**, dowiedz się, w jaki sposób można zmienić kolejność łączenia plików przez program łączący, i zmieniaj ją dopóty, dopóki wynikiem zwracanym przez funkcję będzie **false**.

33. Rozwiąż problem, który wystąpił w poprzednim ćwiczeniu, wykorzystując pierwszą technikę przedstawioną w książce.
34. Rozwiąż problem, który wystąpił w ćwiczeniu 32., wykorzystując drugą technikę przedstawioną w książce.
35. Nie dołączając pliku nagłówkowego, zadeklaruj funkcję **puts()**, znajdująjącą się w standardowej bibliotece języka C, iwywołaj tę funkcję w funkcji **main()**.

Rozdział 11.

Referencje i konstruktor kopiący

Referencje przypominają stałe wskaźniki, automatycznie wyłuskiwane przez kompilator.

Mimo że referencje istnieją również w języku Pascal, ich wersja używana w języku C++ została zaczerpnięta z Algolu. W języku C++ są one niezbędne do obsługi składni przeciążania operatorów (opisanej w rozdziale 12.); stanowią one jednak również ogólne ułatwienie, pozwalające na nadzór nad sposobem, w jaki parametry są przekazywane do i z funkcji.

W tym rozdziale dokonamy najpierw pobieżnego przeglądu różnic pomiędzy wskaźnikami w językach C i C++, a następnie wprowadzimy pojęcie referencji. Znaczna część rozdziału poświęcona jednak będzie dość trudnej, przynajmniej dla początkujących programistów języka C++, kwestii — konstruktorowi kopującemu, czyli specjalnemu konstruktorowi (wymagającemu referencji), tworzącemu nowy obiekt na podstawie istniejącego już obiektu tego samego typu. Konstruktor kopiący jest używany przez kompilator do przekazywania obiektów *przez wartość* — do i z funkcji.

Na koniec zostanie wyjaśniona dość tajemnicza cecha języka C++ — *wskaźnik do składowej*.

Wskaźniki w C++

Najistotniejsza różnica pomiędzy wskaźnikami w językach C i C++ polega na tym, że C++ jest językiem o silniejszej kontroli typów. Szczególnie wyraźnie ujawnia się to w przypadku wskaźnika typu **void***. Język C nie pozwala na swobodne przypisywanie wskaźników jednych typów wskaźnikom innych typów, ale umożliwia dokonanie tego za pomocą wskaźnika typu **void***. Na przykład w następujący sposób:

```
ptak* p;
skala* s;
void* v;
v = s;
p = v;
```

Ta cecha języka C, umożliwiająca niejawne traktowanie dowolnego typu w taki sposób, jakby był innym typem, stanowi spory wyłom w systemie typów. Język C++ nie pozwala na takie operacje; kompilator zgłasza w takich przypadkach komunikat o błędzie i jeżeli naprawdę istnieje potrzeba traktowania jakiegoś typu tak, jakby był innym typem, to trzeba uczynić to jawnie — zarówno dla kompilatora, jak i dla osoby czytającej kod — używając rzutowania (w rozdziale 3. wprowadzono udoskonaloną „jawną” składnię rzutowania, dostępną w języku C++).

Referencje w C++

Referencje (&) przypominają stałe wskaźniki, które są automatycznie wyłuskiwane. Zazwyczaj są używane w listach argumentów funkcji i jako wartości zwracane przez funkcje. Można jednak również utworzyć samodzielne referencje. Ilustruje to poniższy przykład:

```
//: C11:FreeStandingReferences.cpp
#include <iostream>
using namespace std;

// Zwykłe, samodzielne referencje:
int y;
int& r = y;
// Podczas tworzenia, referencja musi
// zostać zainicjowana istniejącym obiektem.
// Można jednak również napisać:
const int& q = 12; // (1)
// Referencje są związane z obszarem pamięci
// jakiegoś innego obiektu:
int x = 0; // (2)
int& a<<x; // (3)
int main() {
    cout << "x = " << x << ", a = " << a << endl;
    a++;
    cout << "x = " << x << ", a = " << a << endl;
} //:-
```

W wierszu (1) kompilator przydziela obszar pamięci, inicjalizuje go wartością 12, a następnie związuje z tym obszarem referencję. Sedno problemu tkwi w tym, że referencja musi być związana z obszarem pamięci, należącym do jakiegoś *innego* obiektu. Podczas odwoływania się do referencji odnosimy się właśnie do tego obszaru. Tak więc po wpisaniu takich wierszy, jak (2) i (3), a następnie **inkrementacji** referencji a w rzeczywistości jest inkrementowana wartość zmiennej x, jak pokazano w funkcji **main()**. Najprostszym sposobem ujęcia referencji jest traktowanie jej jako szczególnego wskaźnika. Jedną z zalet tego „wskaźnika” jest to, że nigdy nie trzeba zastanawiać się nad tym, czy został on zainicjowany (zapewnia to kompilator) ani w **jaki** sposób go wyłuskać (to również wykonuje kompilator).

Z używaniem referencji związane są pewne reguły:

1. Referencja musi zostać zainicjowana podczas tworzenia (wskaźniki mogą zostać zainicjowane w dowolnej chwili).
2. Po zainicjalizowaniu referencji za pomocą jakiegoś obiektu nie można jej zmienić w taki sposób, by wskazywała inny obiekt (wskaźnikom zawsze można przypisać adres innego obiektu).
3. Nie istnieją puste referencje. **Zawsze** można założyć, że referencja jest powiązana z prawidłowym obszarem pamięci.

Wykorzystanie referencji w funkcjach

Miejscem, w którym najczęściej można spotkać referencje, są argumenty funkcji oraz wartości przez nie zwracane. W przypadku gdy referencja jest wykorzystywana w charakterze argumentu funkcji, wszelkie jej modyfikacje, zachodzące *wewnątrz* funkcji, mają wpływ na wartość argumentu znajdującego się *na zewnątrz* funkcji. Oczywiście, można osiągnąć to samo, przekazując wskaźnik, ale referencje mają znacznie bardziej przejrzystą składnię (możesz również traktować referencje wyłącznie jako ułatwienie składniowe).

W przypadku zwracania przez funkcję referencji należy zachować taką samą ostrożność, jak w razie zwracania przez funkcję wskaźnika. To, z czym związana jest referencja, nie powinno przestać istnieć po powrocie z funkcji, ponieważ w przeciwnym wypadku zakończyłoby się to odwołaniem do nieokreślonego obszaru pamięci.

Oto przykład:

```
//: C11:Reference.cpp
// Proste referencje języka C++

int* f(int* x) {
    (*x)++;
    return x; // Bezpieczne, zmienna x znajduje się poza zasięgiem
}

int& g(int& x) {
    x++; // Taki sam rezultat, jak w funkcji f()
    return x; // Bezpieczne, zmienna x znajduje się poza zasięgiem
}

int& h() {
    int q;
    //! return q; // Błąd
    static int x;
    return x; // Bezpieczne, zmienna x znajduje się poza zasięgiem
}

int main() {
    int a = 0;
    f(&a); // Brzydko (choć jawnie)
    g(a); // Przejrzyście (lecz w ukryty sposób)
} //:-
```

Wywołanie funkcji **f()** nie jest tak wygodne i przejrzyste, jak w przypadku użycia referencji, ale nie ulega wątpliwości, że jest przekazywany adres. Podczas wywołania funkcji **g()** również przekazywany jest adres (przez referencję), ale nie jest to widoczne.

Referencje do stałych

Argument funkcji, będący **referencją**, widoczny w pliku **Reference.cpp**, jest poprawny tylko w przypadku, gdy jest on obiektem **niebędącym stałą**. Jeżeli zaś jest on **stałą**, to funkcja **g()** nie przyjmie go w charakterze argumentu, co jest w istocie słuszne, ponieważ funkcja ta *modyfikuje* swój argument. Jeżeli wiadomo, że funkcja respektuje fakt, że jej argument jest **stałą**, to uczynienie argumentu referencją do stałej pozwoli na użycie tej funkcji we wszystkich sytuacjach. Oznacza to, że w przypadku wbudowanych typów funkcja nie zmodyfikuje swojego argumentu, a co do typów zdefiniowanych przez użytkownika — będzie ona wypoływać wyłącznie stałe funkcje składowe, **niemodyfikujące** publicznych danych składowych klasy.

Użycie referencji do stałych w charakterze argumentów funkcji jest szczególnie ważne z uwagi na to, że funkcja może pobrać obiekt tymczasowy. Móglby on zostać utworzony jako wartość, zwracana przez inną funkcję lub jawnie — przez użytkownika funkcji. Obiekty tymczasowe są zawsze stałymi, więc jeżeli jako argument nie zostanie określona referencja do stałej, to argument taki nie zostanie zaakceptowany przez kompilator. Poniżej przedstawiono ilustrujący to zjawisko bardzo prosty przykład:

```
//: C11:ConstReferenceArguments.cpp
// Przekazywanie referencji jako stałych

void f(int&)
void g(const int&)

int main()
{
    //! f(); // Błąd
    g();
}
```

Wywołanie **f(l)** powoduje zgłoszenie błędu w czasie komplikacji, ponieważ kompilator musi najpierw utworzyć referencję. Dokonując tego, przydzieliając pamięć dla liczby całkowitej; inicjalizuje ją wartością jeden i zwraca adres, z którym związana będzie referencja. Przydzielony obszar pamięci *musi* być **stałą**, ponieważ zmiana jego wartości nie miałaby sensu — w żaden sposób nie można by się bowiem do niego ponownie odwołać. W przypadku wszystkich obiektów tymczasowych należy przyjąć to samo założenie — są one niedostępne. Informowanie przez kompilator o modyfikacji takich danych jest korzystne, ponieważ rezultat takiej modyfikacji zostałby utracony.

Referencje do wskaźników

W przypadku zamiaru modyfikacji *wartości* wskaźnika (a nie wartości, którą on wskazuje), podanego jako parametr, należałooby użyć w języku C następującej deklaracji funkcji:

```
void f(int**);
```

a funkcji należałooby przekazać parametr będący adresem wskaźnika:

```
int i = 47;
int* ip = &i;
f(ip);
```

W przypadku referencji, dostępnych w języku C++, składnia jest prostsza. Argument funkcji staje się referencją do wskaźnika, w związku z czym nie ma już potrzeby pobieraniajego adresu. Przybiera to następującą postać:

```
//: C11:ReferenceToPointer.cpp
#include <iostream>
using namespace std;

void increment(int*& i) { i++; }

int main() {
    int* i = 0;
    cout << "i = " << i << endl;
    increment(i);
    cout << "i = " << i << endl;
}///:-
```

Po uruchomieniu powyższego programu można się przekonać, że inkrementowany jest wskaźnik, a nie wartość, którą wskazuje.

Wskazówki dotyczące przekazywania argumentów

Normalną praktyką, związaną z przekazywaniem argumentów do funkcji, powinno być przekazywanie ich do stałych w postaci referencji. Mimo że na pierwszy rzut oka może to wyglądać na podyktowane jedynie względami efektywności (a kwestii poprawy efektywności podczas projektowania i budowy programu nie należy zazwyczaj brać pod uwagę), to w tym przypadku wchodzi w rachubę coś więcej — jak przekonamy się w dalszej części rozdziału, do przekazania obiektu przez wartość niezbędny jest konstruktor kopiący, a nie zawsze jest on dostępny.

Uzyskana poprawa efektywności może być znaczna. Wynika to z prostego powodu — przekazanie argumentu przez wartość wymaga wywołania konstruktora i destruktora, ale w przypadku gdy nie zamierza się modyfikować argumentu, to do przekazania go za pomocą referencji do stałej **potrzebne jest tylko umieszczenie na stosiejego adresu**.

Właściwie przekazywanie adresu *nie jest* zalecaną metodą przekazywania argumentu jedynie wówczas, gdy zamierza się dokonać takiego zniszczenia obiektu, że przekazanie go przez wartość stanowi jedyne bezpieczne rozwiązanie (lepsze niż modyfikacja obiektu zewnętrznego, czego na ogół nie spodziewa się wywołujący funkcję). Kwestia ta jest tematem następnego podrozdziału.

Konstruktor kopiący

Zapoznawszy się z podstawowymi wiadomościami na temat referencji w języku C++, możesz zmierzyć się z jednym z trudniejszych pojęć tego języka — konstruktorem kopującym, określonym często jako X(X&) („X od referencji do X”). Konstruktor

ten ma kluczowe znaczenie dla sterowania przekazywaniem i zwracaniem przez wartość typów zdefiniowanych przez użytkownika podczas wywoływania funkcji. Jak się wkrótce przekonamy, jest on tak ważny, że kompilator zawsze automatycznie tworzy konstruktor kopiący, jeżeli nie został on zdefiniowany przez programistę.

Przekazywanie i zwracanie przez wartość

Aby zrozumieć potrzebę istnienia konstruktora kopiącego, rozważmy sposób, w jaki język C obsługuje przekazywanie i zwracanie zmiennych przez wartość w trakcie wywołania funkcji. Jeżeli zostanie zadeklarowana, a następnie wywołana funkcja:

```
int f(int x, char c);
int g = f(a, b);
```

to skąd kompilator może wiedzieć, w jaki sposób przekazać i zwrócić wartości tych zmiennych? Po prostu wie! Zakres typów, z którymi musi sobie poradzić, jest dość niewielki — char, int, **float**, double, a także ich odmiany — informacje te są wbudowane w kompilator.

Jeżeli dowieś się, w jaki sposób wygenerować program w asemblerze za pomocą używanego przez ciebie kompilatora, i odnajdziesz instrukcje, wygenerowane dla wywołania funkcji **f()**, to będą one równoważne poniższym instrukcjom:

```
push b
push a
call f()
add sp,4
mov g, register a
```

Kod ten został znacznie uproszczony, by mógł stanowić ogólny przykład — symbole zmiennych b i a mogą mieć różną postać, w zależności od tego, czy zmienne te będą globalne (w tym przypadku noszą nazwy _b i _a), czy też lokalne (kompilator będzie używał indeksów względem wskaźnika stosu). Podobnie rzecz się ma z identyfikatorem zmiennej g. Postać wywołania funkcji **f()** będzie zależała od używanego przez kompilator sposobu uzupełniania nazw, a postać tekstu „register a” („rejestra”) — od tego, w jaki sposób nazywane są rejestrze procesora w języku asemblera. Logika, zawarta w kodzie, pozostanie jednak taka sama.

W językach C oraz C++ argumenty są najpierw umieszczane na stosie — od prawej strony do lewej — a następnie wywoływana jest funkcja. Kod wywołujący funkcję jest odpowiedzialny za usunięcie argumentów ze stosu (co realizuje instrukcja add sp,4). Należy jednak zwrócić uwagę na to, że kompilator, przekazując argumenty przez wartość, umieszcza po prostu na stosie ich kopie. Wie bowiem, jaki mają one rozmiar, a także to, że umieszczenie na stosie powoduje utworzenie ich dokładnych kopii.

Wartość zwracana przez funkcję **f()** jest umieszczana w rejestrze. I w tym przypadku kompilator wie wszystko, co niezbędne na temat typu zwracanej wartości — ponieważ typ ten jest wbudowany w język, kompilator może zwrócić tę wartość, umieszczając ją w rejestrze. W przypadku prostych typów danych, dostępnych w języku C, działanie **polegające na kopiowaniu poszczególnych bitów, tworzących wartość, jest równoznaczne z kopiowaniem obiektu.**

Przekazywanie i zwracanie dużych obiektów

Zastanówmy się teraz nad typami zdefiniowanymi przez użytkownika. Jeżeli utworzymy klasę, a następnie będziemy chcieli przekazać obiekt tej klasy przez wartość, to skąd kompilator będzie wiedział, co ma w takim przypadku zrobić. Nie jest to typ wbudowany w kompilator, lecz zdefiniowany przez użytkownika.

Aby to sprawdzić, zaczniemy od prostej struktury, która jest w oczywisty sposób zbyt duża, by można ją było zwrócić, używając do tego celu rejestrów:

```
//: C11:PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Jakaś operacja na argumencie
    return b;
}

int main() {
    B2 = bigfun(B);
} //:-
```

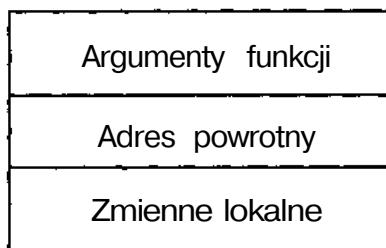
Rozszyfrowanie wygenerowanego przez kompilator programu w języku asemblera jest w tym przypadku nieco trudniejsze, ponieważ większość kompilatorów wykorzystuje funkcje pomocnicze, nie umieszczając całego kodu w miejscu wywołania funkcji. Wywołanie funkcji **bigfun()** w funkcji **main()** rozpoczyna się tak, jak można się tego spodziewać — na stosie jest umieszczana cała zawartość obiektu B (niektóre kompilatory zapisują w rejestrach adres obiektu typu **Big** oraz jego wielkość, a następnie wywołują funkcję pomocniczą, umieszczającą ten obiekt na stosie).

W rozpatrywanym poprzednio fragmencie kodu przed wywołaniem funkcji niezbędne było jedynie umieszczenie jej argumentów na stosie. Jednak w przypadku programu **PassingBigStructures.cpp** zaobserwować można jeszcze dodatkową operację — przed wywołaniem funkcji na stosie jest umieszczany adres struktury **B2**, choć jest oczywiste, że nie jest ona argumentem. Aby zrozumieć, dlaczego tak się dzieje, należy zdać sobie sprawę z ograniczeń nałożonych na kompilator podczas tworzenia przez niego kodu, wywołującego funkcję.

Ramka stosu podczas wywołania funkcji

Gdy kompilator generuje kod realizujący wywołanie funkcji, najpierw umieszcza na stosie wszystkie jej argumenty, a następnie ją wywołuje. Kod, generowany wewnątrz funkcji, powoduje przesunięcie wskaźnika stosu jeszcze bardziej w dół — w celu zapewnienia miejsca, w którym będą przechowywane jej zmienne lokalne (pojęcie „w dół” ma w tym wypadku charakter względny — komputer może zwiększać lub zmniejszać wskaźnik stosu podczas umieszczania na nim wartości). Jednak w trakcie wykonywania instrukcji CALL asemblera procesor umieszcza na stosie adres kodu

programu, *spod którego* nastąpiło wywołanie funkcji, dzięki czemu instrukcja **RETURN** asemblera może wykorzystać ten adres w celu powrotu do miejsca wywołania. Adres ten stanowi, oczywiście, świętość, gdyż bez niego program całkiem by się po-gubił. Poniżej przedstawiono postać ramki stosu po wykonaniu instrukcji **CALL** i przydzieleniu pamięci zmiennym lokalnym funkcji:



Kod, wygenerowany dla pozostały części funkcji, zakłada, że pamięć będzie zorganizowana dokładnie w taki sposób, dzięki czemu może on ostrożnie pobierać argumenty funkcji oraz zmienne lokalne, nie naruszając adresu powrotnego. Blok pamięci zawierający wszystko to, co jest używane przez funkcję w czasie jej wywołania, nazwijmy *ramką funkcji*.

Być może wydaje ci się rozsądne, by funkcja zwracała wartości na stosie. Kompilator mógłby umieścić je na stosie, a funkcja wróciłaby wartość przesunięcia; określa ono, w którym miejscu stosu rozpoczyna się zwracana wartość.

Wielobieżność

Przyczyna problemu tkwi w tym, że języki C oraz C++ akceptują występowanie przerwań — czyli są one *wielobieżne* (ang. *re-entrant*). Dopuszczają one również rekurencyjne wywołania funkcji. Oznacza to, że w dowolnym momencie wykonywania programu może nastąpić przerwanie, niewstrzymujące działania programu. Oczywiście, osoba tworząca procedurę obsługi przerwań (ang. *interrupt service routine — ISR*) jest odpowiedzialna za zachowanie — i późniejsze odtworzenie — wszystkich używanych przez nią rejestrów. Jednakże jeśli taka procedura wymaga wykorzystania pamięci znajdującej się poniżej wskaźnika stosu, musi mieć możliwość dokonania tego w bezpieczny sposób (można traktować procedurę obsługi przerwań jako zwykłą funkcję, niepobierającą argumentów i niezwracającą wartości — zachowującą, a później odtwarzającą stan procesora; wywołanie procedury obsługi przerwań następuje na skutek wystąpienia jakiegoś zdarzenia sprzętowego, a nie jawnego wywołania jej w programie).

Wyobraźmy sobie, co by się stało, gdyby zwykła funkcja próbowała pozostawić na stosie zwracaną przez siebie wartość. Nie można zmieniać żadnej części stosu znajdującej się powyżej adresu powrotnego, więc funkcja musiałaby umieścić tę wartość poniżej tego adresu. Jednak, w chwili wykonywania instrukcji **RETURN** asemblera, wskaźnik stosu musi wskazywać adres powrotny (lub znajdującą się bezpośrednio pod nim pozycję — w zależności od używanego komputera). A zatem tuż przed wykonaniem tej instrukcji funkcja musiałaby przesunąć wskaźnik stosu do góry, usuwając w ten sposób wszystkie swoje zmienne lokalne. Gdyby podjąć próbę pozostawienia

zwracanej przez funkcję wartość poniżej wskaźnika stosu, pojawiłoby się w tym momencie niebezpieczeństwo związane z nadaniem przerwania. Procedura obsługi przerwań mogłaby przesunąć wskaźnik stosu w dół, zapamiętując na stosie adres powrotu oraz używane przez siebie zmienne lokalne, co spowodowałoby zniszczenie wartości zwróconej przez naszą funkcję.

Aby rozwiązać ten problem, kod wywołujący funkcję *móglby* odpowiadać za przydzielenie na stosie — przed wywołaniem funkcji — dodatkowego obszaru pamięci, przechowującego zwracane przez nią wartości. Jednakże język C nie został w taki sposób zaprojektowany, a język C++ musi być z nim zgodny. Jak się wkrótce przekonamy, kompilator języka C++ wykorzystuje bardziej efektywny sposób przekazywania wartości zwracanej przez funkcję.

Kolejny pomysł mógłby polegać na zwracaniu wartości za pośrednictwem jakiegoś obszaru danych globalnych, ale to również nie działałoby poprawnie. Wielobieżność oznacza, że każda funkcja może być procedurą obsługi przerwania dla dowolnej innej funkcji. Procedurą taką *może więc być również funkcja, która jest aktualnie wykonywana*. Tak więc, po umieszczeniu wartości zwracanej przez funkcję w obszarze danych globalnych, mógłby nastąpić powrót do tej samej funkcji, która następnie zamazałaby zwróconą wartość¹. Ten sam tok rozumowania odnosi się do rekurencji.

Jedynym miejscem, w którym można bezpiecznie przekazać zwracane wartości, są rejestrysty. Powracamy więc do problemu, sprowadzającego się do pytania, co uczynić w przypadku, gdy rejestrysty nie są dostatecznie duże, by pomieścić zwracaną wartość. Rozwiązaniem jest umieszczenie na stosie adresu miejsca, w którym należy zapisać wartość zwracaną przez funkcję — jako jednego z jej argumentów. Jednocześnie należy pozwolić funkcji skopiowanie bezpośrednio w tym miejscu zwracanej przez nią wartości. Rozwiązuje to nie tylko wszystkie problemy, ale jest również bardziej efektywne. Z tego także powodu w programie **PassingBigStructures.cpp** kompilator — przed wywołaniem funkcji **bigfun()** w funkcji **main()** — umieścił na stosie adres struktury **B2**. Gdyby przyjrzeć się asemblerowemu kodowi funkcji **bigfun()**, to można by zauważyc, że oczekuje ona tego ukrytego argumentu. Ponadto *wewnątrz* tej funkcji jest dokonywane kopiowanie zwracanej wartości pod wskazywany przez ten argument adres.

Kopiowanie bitów kontra inicjalizacja

Posiadamy już zatem działający mechanizm, umożliwiający przekazywanie i zwracanie dużych, prostych struktur. Zwróć jednak uwagę na to, że dysponujemy jedynie sposobem kopowania bitów z jednego miejsca na drugie, co z pewnością dobrze sprawdza się w przypadku prostego sposobu postrzegania zmiennych przez język C. Jednak w języku C++ zmienne mogą być czymś dużo bardziej wyszukanym niż tylko zlepakami bitów — posiadają bowiem znaczenie. Znaczenie to może nie tolerować kopowania swoich bitów.

¹ Niejest to dobry przykład, ponieważ — jak autor napisał wcześniej — procedura obsługi przerwań nie zwraca żadnej wartości. Niebezpieczeństwo polega natomiast na tym, że procedura obsługi przerwań mógłby *wywołać* funkcję, która została przez tę procedurę przerwana — *przyp. tłum.*

Rozważmy prosty przykład — klasę zawierającą informację o tym, ile jej obiektów istnieje równocześnie. W rozdziale 10. dowiedzieliśmy się, że można to osiągnąć, włączając do klasy statyczną składową:

```
//: C11:HowMany.cpp
// Klasa zliczająca swoje obiekty
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany.out");

class HowMany {
    static int objectCount;
public:
    HowMany() { objectCount++; }
    static void print(const string& msg = "") {
        if(msg.size() != 0) out << msg << ": ";
        out << "objectCount = "
           << objectCount << endl;
    }
    ~HowMany() {
        objectCount--;
        print("~HowMany()");
    }
};

int HowMany::objectCount = 0;

// Przekazywanie i zwracanie PRZEZ WARTOŚĆ;
HowMany f(HowMany x) {
    x.print("argument x wewnętrz f()");
    return x;
}

int main() {
    HowMany h;
    HowMany::print("po utworzeniu h");
    HowMany h2 = f(h);
    HowMany::print("po wywołaniu funkcji f()");
} //:-
```

Klasa **HowMany** zawiera statyczną składową całkowitą **objectCount** (odgrywającą rolę „licznika obiektów”) oraz statyczną funkcję składową **print()**, drukującą wartość **objectCount** wraz z dodatkowym, opcjonalnym komunikatem, podanym jako argument. Konstruktor zwiększa wartość licznika, ilekroć jest tworzony obiekt, a destruktork zmniejsza tę wartość.

Wyniki działania programu nie są jednak zgodne z oczekiwaniami:

```
po utworzeniu h: objectCount = 1
argument x wewnętrz f(): objectCount = 1
~HowMany(): objectCount = 0
~HowMany(): objectCount = -1
po wywołaniu funkcji f(): objectCount = -1
~HowMany(): objectCount = -2
~HowMany(): objectCount = -3
```

Po utworzeniu obiektu `h` wartość licznika wynosi jeden, co jest poprawne. Jednak po wywołaniu funkcji `f()` można by się spodziewać zwiększenia licznika do dwóch, ponieważ w zasięgu znajduje się również drugi obiekt tej klasy — `h2`. Jednak wartość tego licznika wynosi zero, co oznacza, że doszło do katastrofy. Ponadto dwa następne destruktory spowodowały w końcu osiągnięcie przez licznik wartości ujemnej. Taka sytuacja nie powinna mieć nigdy miejsca.

Zastanówmy się, co się dzieje wewnątrz funkcji `f()` bezpośrednio po przekazaniu jej argumentu przez wartość. Początkowy obiekt `h` istnieje na zewnątrz ramki funkcji, a w obrębie jej ramki znajduje się dodatkowy obiekt będący jego kopią, przekazaną przez wartość. Argument ten został jednak przekazany za pomocą prymitywnej metody języka C, sprowadzającej się do kopiowania bit po bicie, podczas gdy klasa **HowMany** języka C++ wymaga, dla zachowania swojej integralności, rzeczywistej inicjalizacji — i dlatego właśnie kopiowanie bitów nie przynosi pożądanego rezultatu.

Pod koniec wywołania funkcji `f()`, gdy obiekt lokalny wychodzi poza zasięg, wywoływany jest destruktor, dekrementujący wartość składowej `objectCount`, co sprawia, że po opuszczeniu funkcji licznik ten ma wartość zerową. Tworzenie obiektu `h2` odbywa się także za pomocą kopiowania bitów, więc również w tym przypadku konstruktor nie jest wywoływany. Kiedy zatem obiekty `h` i `h2` wykroczą poza swój zasięg, ich destruktory spowodują nadanie zmiennej składowej `objectCount` wartości ujemnej.

Konstrukcja za pomocą konstruktora kopiącego

Przyczyną problemu jest przyjęte przez kompilator założenie dotyczące sposobu, w jaki na podstawie istniejącego obiektu tworzony jest nowy obiekt. Podczas przekazywania obiektu przez wartość jest tworzony nowy obiekt. To przekazywany obiekt wewnątrz ramki funkcji, utworzony na podstawie istniejącego obiektu, znajdującego się poza ramką funkcji. Zdarza się to również często w przypadku, gdy obiekt jest zwracany przez funkcję. W wyrażeniu:

```
HowMany h2 = f(h);
```

nieskonstruowany jeszcze obiekt `h2` jest tworzony na podstawie wartości, zwracanej przez funkcję `f()`, a więc ponownie — nowy obiekt jest tworzony na podstawie obiektu już istniejącego.

Kompilator zakłada, że tworzenie takiego obiektu powinno odbywać się metodą kopiowania bitów. Sprawdza się to w wielu przypadkach, ale nie działa w stosunku do klasy **HowMany**, ponieważ znaczenie inicjalizacji wykracza poza zakres zwykłego kopiowania. Innym typowym przykładem jest sytuacja, w której klasa zawiera wskaźniki. Trzeba odpowiedzieć sobie na pytania: co wskazują i czy należy je kopiować, czy też powiązać z jakimś innym obszarem pamięci?

Na szczęście, można ingerować w ten proces i powstrzymać kompilator przed kopiowaniem bitów. Należy w tym celu zdefiniować swoją funkcję, używaną przez kompilator w sytuacjach, gdy konieczne jest utworzenie nowego obiektu na podstawie obiektu już istniejącego. Wydaje się logiczne, że skoro tworzony jest nowy obiekt, to

funkcja ta jest konstruktorem i jego jedynym argumentem jest obiekt, na którego podstawie odbywa się tworzenie nowego obiektu klasy. Argument ten nie może być jednak przekazany przez wartość. Próbujemy bowiem zdefiniować funkcję, która będzie obsługiwała przekazywanie argumentów przez wartość, natomiast przekazywanie wskaźnika nie ma sensu, ponieważ, mimo wszystko, tworzymy nowy obiekt na podstawie istniejącego. W tym przypadku służą pomocą referencje, więc argumentem konstruktora będzie referencja do obiektu źródłowego. Funkcja ta jest nazywana *konstruktorem kopiującym* (ang. *copy-constructor*), często określonym jako X(X&), co stanowi jego postać dla klasy o nazwie X.

Jeżeli zostanie utworzony konstruktor kopiujący, to kompilator nie będzie kopował bitów podczas tworzenia nowych obiektów na podstawie już istniejących. Zawsze wywoła utworzony konstruktor kopiujący. Tak więc jeżeli nie zostanie utworzony konstruktor kopiujący, to kompilator postąpi rozsądnie, ale zawsze istnieje możliwość przejęcia pełnej kontroli nad tym procesem.

A zatem jest już możliwe usunięcie problemu, który wystąpił w programie **HowMany.cpp**:

```
//: C11:HowMany2.cpp
// Konstruktor kopiujący
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
    string name; // Identyfikator obiektu
    static int objectCount;
public:
    HowMany2(const string& id = "") : name(id) {
        ++objectCount;
        print("HowMany2()");
    }
    ~HowMany2() {
        --objectCount;
        print("~HowMany2()");
    }
    // Konstruktor kopiujący:
    HowMany2(const HowMany2& h) : name(h.name) {
        name += " kopia";
        ++objectCount;
        print("HowMany2(const HowMany2&) ");
    }
    void print(const string& msg = "") const {
        if(msg.size() != 0)
            out << msg << endl;
        out << '\t' << name << ":" " "
            << "objectCount - "
            << objectCount << endl;
    }
};

int HowMany2::objectCount = 0;
```

```
// Przekazywanie 1 zwracanie PRZEZ WARTOŚĆ:  
HowMany2 f(HowMany2 x) {  
    x.print("argument x wewnątrz f()");  
    out << "Powrot z funkcji f()" << endl;  
    return x;  
}  
  
int main() {  
    HowMany2 h("h");  
    out << "wywołanie funkcji f()" << endl;  
    HowMany2 h2 = f(h);  
    h2.print("h2 po wywołaniu funkcji f()");  
    out << "Wywołanie funkcji f(). brak zwracanej wartości" << endl;  
    f(h);  
    out << "Po wywołaniu funkcji f()" << endl;  
} //--
```

Wprowadzono tu szereg nowych elementów, dzięki którym można się lepiej zorientować w sytuacji. Po pierwsze, łańcuch **name** pełni funkcję identyfikatora obiektu podczas drukowania informacji na jego temat. Wywołując konstruktora, można podać łańcuch identyfikacyjny (zazwyczaj nazwę obiektu), który zostanie skopiowany do składowej **name** przy użyciu konstruktora typu **string**. Argument domyślny "" tworzy pusty łańcuch. Konstruktora, podobnie jak poprzednio, zwiększa wartość składowej **objectCount**, a destruktora — zmniejszają.

Nastecną nowością jest konstruktor kopiący, **HowMany2(const HowMany2&)**. Pozwala on na utworzenie nowego obiektu wyłącznie na podstawie obiektu już istniejącego, więc do składowej **name** kopiowana jest nazwa istniejącego obiektu wraz z przyrostkiem „kopia”. Dzięki temu można się zorientować, na jakiej podstawie obiekt ten został utworzony. Jeśli przyjrzej się uważniej, można zauważyć, że wywołanie **name(h.name)**, występujące na liście inicjalizatorów konstruktora, jest w rzeczywistości wywołaniem konstruktora kopiącego klasy **string**.

Wewnątrz konstruktora kopiącego zwiększany jest licznik obiektów — jak w przypadku zwykłego konstruktora. Oznacza to, że obecnie licznik obiektów będzie działał poprawnie w razie przekazywania i zwracania obiektów przez wartość.

Funkcja **print()** została zmodyfikowana w taki sposób, by drukować komunikat, identyfikator obiektu oraz licznik obiektów. Ponieważ funkcja musi obecnie mieć dostęp do składowej **name** danego obiektu, więc nie może być już statyczną funkcją składową.

Wewnątrz funkcji **main()** zostało dodane drugie wywołanie funkcji **f()**. Wywołanie to prezentuje typowe dla języka C ignorowanie wartości zwracanej przez funkcję. Wiedząc już, w jaki sposób zwracana jest wartość (kod znajdujący się wewnątrz funkcji **obsługuje** proces zwracania wartości, umieszczając wynik w miejscu **przeznaczenia**, którego adres jest przekazywany jako ukryty argument), można się dziwić, co dzieje się w przypadku, gdy jest ona ignorowana. Pewne światło na to zagadnienie rzucają wyniki działania programu.

Zanim zostaną zaprezentowane wyniki, poniżej został zamieszczony krótki program, wykorzystujący strumienie wejścia-wyjścia do ponumerowania wierszy do wolnego pliku:

```
//: C11:Linenum.cpp
//{T} Linenum.cpp
// Numerowanie wierszy
#include "../require.h"
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char* argv[]) {
    requireArgs(argc, 1, "Uzycie: linenum plik\n"
        "Numerowanie wierszy pliku");
    ifstream in(argv[1]);
    assure(in, argv[1]);
    string line;
    vector<string> lines;
    while(getline(in, line)) // Wczytanie całego pliku
        lines.push_back(line);
    if(lines.size() == 0) return 0;
    int num = 0;
    // Liczba wierszy w pliku określa szerokość:
    const int width = int(log10(lines.size())) + 1;
    for(int i = 0; i < lines.size(); i++) {
        cout.setf(ios::right, ios::adjustfield);
        cout.width(width);
        cout << ++num << " " << lines[i] << endl;
    }
} //:~
```

Caly plik jest wczytywany do obiektu klasy `vector<string>` za pomocą tego samego kodu, który był już prezentowany powyżej. Chcielibyśmy, aby podczas drukowania numerów wszystkie wiersze były względem siebie wyrównane. Wymaga to uwzględnienia liczby wierszy w pliku, dzięki czemu szerokość pola przeznaczonego na numer będzie w każdym wierszu taka sama. Liczbę wierszy można łatwo określić, używając funkcji `vector::size()`. Potrzebujemy jednak przede wszystkim informacji, czy liczba wierszy jest większa bądź równa 10, 100, 1000 itd. Jeżeli wyznaczy się logarytm dziesiętny z liczby wierszy, zaokrągli w dół do liczby całkowitej, a następnie doda do niego jeden, uzyskanym wynikiem będzie maksymalna szerokość, zajmowana przez licznik wierszy.

Łatwo dostrzec kilka dziwnych funkcji, wywoływanych wewnątrz pętli `for` — `setf()` oraz `width()`. Są to funkcje klasy `ostream`, pozwalające na kontrolę, w tym przypadku, odpowiednio — wyrównania oraz szerokości wyprowadzanego tekstu. Muszą one jednak być wywoływanie podczas wyprowadzania każdego wiersza i dlatego właśnie umieszczone je w pętli `for`. Drugi tom książki zawiera osobny rozdział poświęcony strumieniom wejścia-wyjścia. Można w nim znaleźć więcej informacji na temat tych funkcji oraz innych sposobów sterowania działaniem strumieni wejścia-wyjścia.

Rezultat użycia programu **Linenumber.cpp** w stosunku do pliku **HowMany2.out** jest następujący:

```
1) HowMany2()
2)   h: objectCount = 1
3) Wywołanie funkcji f()
4) HowMany2(const HowMany2&)
5)   h kopia: objectCount = 2
6) argument x wewnętrz f()
7)   h kopia: objectCount = 2
8) Powrot z funkcji f()
9) HowMany2(const HowMany2&)
10)  h kopia kopia: objectCount = 3
11) ~HowMany2()
12)  h kopia: objectCount = 2
13) h2 po wywołaniu funkcji f()
14)  h kopia kopia: objectCount = 2
15) Wywołanie funkcji f(), brak zwracanej wartości
16) HowMany2(const HowMany2&)
17)  h kopia: objectCount = 3
18) argument x wewnętrz f()
19)  h kopia: objectCount = 3
20) Powrot z funkcji f()
21) HowMany2(const HowMany2&)
22)  h kopia kopia: objectCount = 4
23) ~HowMany2()
24)  h kopia: objectCount = 3
25) ~HowMany2()
26)  h kopia kopia: objectCount = 2
27) Po wywołaniu funkcji f()
28) ~HowMany2()
29)  h kopia kopia: objectCount = 1
30) ~HowMany2()
31)  h: objectCount = 0
```

Jak można się było spodziewać, najpierw wywoływany jest normalny konstruktor dla obiektu h, zwiększający licznik obiektów do wartości jeden. Następnie jednak, podczas wchodzenia do funkcji **f()**, kompilator – w celu przekazania argumentu przez wartość – wywołuje niejawnie konstruktor kopiąjący. W obrębie ramki funkcji **f()** jest tworzony nowy obiekt, będący kopią obiektu h (stąd jego nazwa: „h kopia”), w związku z czym konstruktor kopiąjący zwiększa licznik obiektów do dwóch.

Wiersz ósmy sygnalizuje początek powrotu z funkcji **f()**. Zanim jednak będzie mogła być zniszczona zmienna lokalna „h kopia” (wykroczy poza zasięg pod koniec funkcji), musi ona zostać skopiowana do obiektu zawierającego wartość zwracaną przez funkcję, którym jest obiekt h2. Nieistniejący jeszcze obiekt (h2) jest tworzony na podstawie obiektu już istniejącego (zmiennej lokalnej wewnętrz funkcji **f()**), w związku z czym w dziesiątym wierszu ponownie jest wywoływany konstruktor kopiąjący. Identyfikator obiektu h2 przyjmuje nazwę „h kopia kopia”, ponieważ jest on kopią, utworzoną na podstawie kopii będącej obiektem lokalnym funkcji **f()**. Po utworzeniu zwracanego obiektu, ale jeszcze przed zakończeniem funkcji, licznik obiektów przyjmuje chwilowo wartość równą trzy, ale w następnej kolejności niszczony jest obiekt lokalny „h kopia”. Po zakończeniu wywołania funkcji **f()** w wierszu 13. istnieją już tylko dwa obiekty, h oraz h2. Można się również przekonać, że identyfikatorem obiektu h2 jest rzeczywiście „h kopia kopia”.

Obiekty tymczasowe

W wierszu 15. rozpoczyna się wywołanie funkcji **f(h)**, ignorującce tym razem zwracaną przez funkcję wartość. Jak można zobaczyć w wierszu 16., podobnie jak poprzednio, do przekazania parametru jest wywoływany konstruktor kopiujący. I również jak w poprzednim przypadku, w 21. wierszu, w celu utworzenia wartości zwracanej przez funkcję, wywoływany jest konstruktor kopiujący. Jednak konstruktor kopiujący wymaga adresu, wskazującego obiekt docelowy (wskaźnik **this**). Skąd więc bierze się ten adres?

Okazuje się, że kompilator może utworzyć obiekt tymczasowy — ilekroć potrzebuje go do poprawnego wyznaczenia wartości wyrażenia. W tym przypadku tworzy obiekt, który nie jest nawet widoczny, lecz służy tylko do przechowania ignorowanej wartości, zwracanej przez funkcję **f()**. Czas życia takiego tymczasowego obiektu jest możliwie jak najkrótszy, dzięki czemu program nie jest zaśmiecony obiektem tymczasowym, oczekującym na zniszczenie i zajmującym cenne zasoby. W pewnych przypadkach obiekt tymczasowy może zostać natychmiast przekazany innej funkcji, ale ponieważ po wywołaniu funkcji nie jest on już potrzebny, więc zaraz po tym, jak funkcja kończy swoje działanie, za pomocą wywołania destruktora obiektu lokalnego (wiersze 23. i 24.) niszczony jest również obiekt tymczasowy (wiersze 25. i 26.).

W wierszach od 28. do 31. niszczony jest obiekt **h2**, a następnie obiekt **h**, a wartość licznika obiektów powraca, prawidłowo, do zera.

Domyślny konstruktor kopiujący

Ponieważ konstruktor kopiujący realizuje przekazywanie i zwracanie przez wartość, istotne jest, aby kompilator utworzył taki konstruktor dla prostych struktur — czyli właściwie zrobił to samo, co w języku C. Jednakże do tej pory zapoznaliśmy się jedynie z domyślnym, prostym działaniem takiego konstruktora, polegającym na kopowaniu poszczególnych bitów obiektu.

W przypadku bardziej złożonych typów kompilator języka C++ będzie nadal tworzył automatycznie konstruktor kopiujący, jeżeli nie zostanie on zdefiniowany przez programistę. Wówczas kopianie bitów nie ma jednak sensu, ponieważ w niedostateczny sposób odzwierciedla ono właściwe znaczenie obiektów. Poniższy przykład ilustruje bardziej przemyślne działanie kompilatora. Założymy, że tworzymy klasę złożoną z obiektów pewnych istniejących klas. Metoda ta, nazwana dość trafnie *kompozycją*, stanowi jeden ze sposobów tworzenia nowych klas za pomocą klas już istniejących. Wyobraźmy sobie zachowanie naiwnego użytkownika, który próbuje szybko rozwiązać problem, tworząc w taki właśnie sposób klasę. Nic nie wie na temat konstruktorów kopiujących, więc nie tworzy takiego konstruktora. Poniższy przykład demonstruje, jakie działania podejmuje kompilator, tworząc domyślny konstruktor kopiujący dla utworzonej w taki sposób klasy:

```
// : C:\DefaultCopyConstructor.cpp
// Automatyczne tworzenie konstruktora kopiującego
#include <iostream>
#include <string>
using namespace std;
```

```
class WithCC { // Posiada konstruktor kopiujący
public:
    // Wymagany jawnny domyślny konstruktor:
    WithCC() {}
    WithCC(const WithCC&) {
        cout << "WithCC(WithCC&)" << endl;
    }
};

class WoCC { // Nie posiada konstruktora kopiującego
    string id;
public:
    WoCC(const string& ident = "") : idCident() {}
    void print(const string& msg = "") const {
        if(msg.size() != 0) cout << msg << ": ";
        cout << id << endl;
    }
};

class Composite {
    WithCC withcc; // Osadzony obiekt
    WoCC wocc;
public:
    Composite() : wocc("Composite()") {}
    void print(const string& msg = "") const {
        wocc.print(msg);
    }
};

int main() {
    Composite c;
    c.print("Zawartosc obiektu c");
    cout << "Wywołanie zlozonego konstruktora kopiuującego"
        << endl;
    Composite c2 = c; // Wywołanie konstruktora kopiuującego
    c2.print("Zawartosc obiektu c2");
} //:~
```

Klasa **WithCC** zawiera konstruktor kopiujący, który po prostu informuje o tym, że został wywoływany, co ujawnia interesującą kwestię. Wewnątrz klasy **Composite** obiekt **WithCC** jest tworzony za pomocą domyślnego konstruktora. Gdyby klasa **WithCC** nie posiadała w ogóle konstruktora, to kompilator automatycznie utworzyłby konstruktor domyślny, który — w tym przypadku — nic by nie zrobił. Jednakże dodanie konstruktora kopiuującego informuje kompilator, że zamierzamy samodzielnie tworzyć konstruktory. Nie generuje on już więc domyślnego konstruktora, i — gdyby ten konstruktor nie został utworzony, jak to zrobiliśmy w klasie **WithCC** — zgłosiłby błąd.

Klasa **WoCC** nie posiada konstruktora kopiuującego, ale jej konstruktor zapamiętuje, w wewnętrznej zmiennej typu **string**, komunikat, który można później wydrukować, wywołując funkcję **print()**. Konstruktor ten jest jawnie wywoływany na liście inicjalizatorów konstruktora klasy **Composite** (lista inicjalizatorów konstruktora została krótko omówiona w rozdziale 8. i zostanie wyczerpująco opisana w 12. rozdziale książki). Powód, dla którego tak zrobiono, stanie się później oczywisty.

Klasa **Composite** zawiera obiekty składowe, zarówno klasy **WithCC**, jak i **WoCC** (zwróć uwagę na to, że osadzony w klasie obiekt **woccc** musi być inicjalizowany w obrębie listy inicjatorów konstruktora), nie zawiera natomiast jawnie zdefiniowanego konstruktora kopiącego. Jednakże w funkcji **main()** obiekt tej klasy jest tworzony za pomocą konstruktora kopiącego, w następującej definicji:

```
Composite c2 = c;
```

Konstruktor kopiący klasy **Composite** jest automatycznie generowany przez kompilator, a informacje wyświetlane przez program ujawniają, w jaki sposób został on utworzony:

```
Zawartosc obiektu c: Composite()
Wywołanie zlozonego konstruktora kopiącego
WithCC(WithCC&)
Zawartosc obiektu c2: Composite()
```

Tworząc konstruktor kopiący klasy wykorzystującą kompozycję (a także dziedziczenie, o czym przekonamy się w rozdziale 14.), kompilator wywołuje rekurencyjnie konstruktory kopiące wszystkich obiektów składowych oraz klas podstawowych. Oznacza to, że jeśli obiekty składowe będą zawierały również inne obiekty, to ich konstruktory kopiące zostaną wywołane. Dlatego też, w tym przypadku, kompilator wywołuje konstruktor kopiący klasy **WithCC** (informacje wyświetlane przez program dowodzą, że konstruktor ten został wywołany). Ponieważ klasa **WoCC** nie posiada konstruktora kopiącego, kompilator tworzy dla niej konstruktor, wykorzystujący po prostu kopiowanie bitów, i wywołuje go z wnętrza konstruktora kopiącego klasy **Composite**. Dowodzi tego wywołanie funkcji **Composite::print()** w funkcji **main()**, ponieważ zawartość składowej **c2.woccc** jest taka sama, jak składowej **c.woccc**. Proces realizowany przez kompilator w trakcie tworzenia konstruktora kopiącego jest nazywany *inicjalizacją pośrednictwem elementów składowych* (ang. *memberwise initialization*).

Zawsze lepiej jest przygotować własny konstruktor kopiący niż pozwolić na to, by utworzył go kompilator. Gwarantuje to nadzór nad jego działaniem.

Możliwości zastąpienia konstruktora kopiącego

Masz zapewne ważne wątpliwości — jak można było w ogóle utworzyć działającą klasę, nie wiedząc nic na temat konstruktora kopiącego? Pamiętaj jednak, że konstruktor kopiący jest potrzebny wyłącznie w przypadku, gdy zamierzasz przekazywać obiekt klasy przez *wartość*. Jeżeli taka sytuacja nigdy nie zachodzi, to konstruktor kopiący jest niepotrzebny.

zapobieganie przekazywaniu przez wartość

Możesz jednak zadać pytanie: „Jeżeli nie utworzę konstruktora kopiącego, to kompilator zrobi to za mnie. Skąd mam więc wiedzieć, że obiekt nie zostanie nigdy przekazany przez wartość?”.

Istnieje prosty sposób, uniemożliwiający przekazywanie obiektu przez wartość — zadeklarowanie prywatnego konstruktora kopiącego. Nie trzeba nawet tworzyć jego definicji, chyba że jakaś funkcja składowa lub funkcja zaprzyjaźniona będzie musiała przekazać obiekt przez wartość. Jeżeli użytkownik spróbuje przekazać lub zwrócić obiekt tej klasy przez wartość, kompilator zgłosi komunikat o błędzie, ponieważ konstruktor kopiący jest funkcją składową prywatną. Kompilator nie może już również utworzyć domyślnego konstruktora kopiącego, **ponieważ** wyraźnie określiliśmy, że wzięliśmy to zadanie na siebie.

Poniżej zamieszczono przykład ilustrujący tę metodę:

```
//: C11:NoCopyConstruction.cpp
// Zapobieganie użytku konstruktora kopiącego

class NoCC {
    int i;
    NoCC(const NoCC&); // Brak definicji
public:
    NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);

int main() {
    NoCC n;
    //! f(n); // Błąd: wywołanie konstruktora kopiącego
    //! NoCC n2 = n; // Błąd: wywołanie konstruktora kopiącego
    //! NoCC n3(n); // Błąd: wywołanie konstruktora kopiącego
} //:-
```

Zwróć uwagę na bardziej ogólną postać konstruktora:

```
NoCC(const NoCC&);
wykorzystującą referencję do stałej.
```

Funkcje modyfikujące obiekty zewnętrzne

Zapis z zastosowaniem referencji przedstawia się lepiej niż ten wykorzystujący wskaźniki, ale przesłania nieco swoje prawdziwe znaczenie. Na przykład w bibliotece strumieni wejścia-wyjścia jedna z przeciążonych wersji funkcji `get()` pobiera argument typu `char&`, a głównym jej celem jest modyfikacja argumentu, polegająca na zapisaniu w nim wyniku działania funkcji `get()`. Jednak podczas czytania kodu, wykorzystującego tę funkcję, fakt modyfikacji przez nią obiektu zewnętrznego nie jest wcale od razu oczywisty:

```
char c;
cin.get(c);
```

Wywołanie funkcji wygląda raczej na przekazanie argumentu przez wartość, co sugeruje, że obiekt zewnętrzny *nie jest* modyfikowany.

Z tego względu — z punktu widzenia pielegnacji kodu — bezpieczniej jest użyć wskaźników, jeśli przekazuje się adres obiektu w celu jego modyfikacji. Jeżeli zawsze

przekazujesz adresy w postaci referencji do stałych, z wyjątkiem przypadków, w których zamierzasz zmieniać wartości zewnętrznych obiektów i przekazujesz je wówczas w postaci wskaźników (niebędących wskaźnikami do stałych), to tworzony w taki sposób kod jest znacznie bardziej przejrzysty.

Wskaźniki do składowych

Wskaźnik jest zmienną, przechowującą adres jakiegoś obszaru pamięci. W czasie pracy programu można zmieniać jego wartość, tak by wskazywał on różne obiekty. Obiektami wskazywanymi przez wskaźniki mogą być zarówno dane, jak i funkcje. Występujące w języku C++ *wskaźniki do składowych* realizują tę samą ideę, z wyjątkiem tego, że wskazują one elementy znajdujące się wewnątrz klas. Dylemat w tym przypadku jest następujący: wskaźniki wymagają adresów, a wewnątrz klas nie istnieje pojęcie „adresu” — wybór składowej odbywa się na podstawie określenia jej przesunięcia w obrębie klasy. Nie można określić rzeczywistego adresu, dopóki nie złoży się tego przesunięcia z adresem konkretnego obiektu. Składnia wskaźników do składowych wymaga więc, aby obiekt był określony w chwili, w której dokonuje się wyłuskania wskaźnika do składowej.

Aby zrozumieć tę składnię, rozważmy prostą strukturę **Simple**, zmienną **sp**, będącą wskaźnikiem do niej, oraz obiekt tej struktury **so**. Odwołując się do składowych struktury, można używać przedstawionej poniżej składni:

```
//: C11:SimpleStructure.cpp
struct Simple { int a; };
int main() {
    Simple so. *sp = &so;
    sp->a;
    so.a;
} ///:-
```

Założymy teraz, że posiadamy zmienną **ip**, będącą zwykłym wskaźnikiem liczby całkowitej. Aby odwołać się do tego, co wskazuje zmienna **ip**, należy dokonać wyłuskania wskaźnika, używając do tego celu operatora **“*”**:

```
*ip = 4;
```

Zastanówmy się wreszcie, co dzieje się w przypadku, gdy dysponujemy wskaźnikiem wskazującym coś, co znajduje się wewnątrz obiektu klasy, nawet jeżeli w rzeczywistości reprezentuje on jedynie przesunięcie względem początku tego obiektu. Aby uzyskać dostęp do tego, co wskazuje, trzeba go wyłuskać za pomocą operatora *****. Jednakże wskaźnik określa tylko przesunięcie w obrębie obiektu, więc trzeba również odwołać się do jakiegoś konkretnego obiektu. Dlatego też operator ***** został połączony z jego wyłuskaniem. Tak więc powstała w ten sposób składnią jest **->*** (w przypadku wskaźników do obiektów) oraz **.*** (w przypadku obiektów lub referencji), jak widać w poniższych przykładach:

```
wskaznikObiektu->*wskaznikSkadowej = 47;
obiekt.*wskaznikSkadowej = 47;
```

Jak jednak zdefiniować **wskaznikSkładowej**? Podobnie jak w przypadku każdego wskaźnika, trzeba określić wskazywany przez niego typ i użyć w definicji znaku *. Jedyna różnica polega na tym, że trzeba określić klasę, z której obiektami jest używany ten wskaźnik. Oczywiście, uzyskuje się to za pomocą nazwy tej klasy oraz operatora zasięgu. Tak więc:

```
int KlasaObiektu::*wskaznikSkładowej;
```

definiuje zmienną o nazwie **wskaznikSkładowej**, będącą wskaźnikiem do dowolnej składowej typu całkowitego, znajdującej się wewnątrz obiektu klasy **KlasaObiektu**. Można również zainicjować wskaźnik do składowej w czasie jego definicji (lub później, w dowolnej chwili):

```
int KlasaObiektu::*wskaznikSkładowej = &KlasaObiektu::a;
```

W rzeczywistości nie istnieje „adres” składowej **KlasaObiektu::a**, ponieważ odwołujemy się do klasy, a nie do obiektu tej klasy. Tak więc wyrażenie **&KlasaObiektu::a** może zostać użyte wyłącznie w kontekście wskaźnika do składowej.

Poniżej zamieszczono program przedstawiający sposoby tworzenia i używania wskaźników do składowych:

```
//: C11:PointerToMemberData.cpp
#include <iostream>
using namespace std;

class Data {
public:
    int a, b, c;
    void print() const {
        cout << "a = " << a << ", b = " << b
           << ", c = " << c << endl;
    }
};

int main() {
    Data d, *dp = &d;
    int Data::*pmInt = &Data::a;
    dp->*pmInt = 47;
    pmInt = &Data::b;
    d.*pmInt = 48;
    pmInt = &Data::c;
    dp->*pmInt = 49;
    dp->print();
} III-
```

Oczywiście, wskaźniki te są zbyt niewygodne, by ich gdziekolwiek używać, z wyjątkiem szczególnych przypadków (właśnie z myślą o nich zostały one utworzone).

Wskaźniki do składowych są również dość ograniczone — można im przypisać tylko określone miejsce w obrębie klasy. Nie można ich, na przykład, inkrementować ani porównywać, jak w przypadku zwykłych wskaźników.

Funkcje

W podobny sposób można uzyskać wskaźniki do składowych będących funkcjami. Wskaźniki do funkcji (wprowadzone pod koniec rozdziału 3.) są definiowane w następujący sposób:

```
int (*fp)(float);
```

Ujęcie wskaźnika (*fp) w nawias jest konieczne, by wymusić na kompilatorze właściwą interpretację definicji. Gdyby nie miał on nawiasu, wyglądałby na funkcję zwracającą wartość typu int*.

Nawiasy odgrywają również ważną rolę podczas definiowania i stosowania wskaźników do funkcji składowych. Wskaźnik do funkcji zawartej w klasie jest definiowany poprzez wstawienie nazwy klasy oraz operatora zakresu do zwykłej definicji wskaźnika funkcji:

```
//: C11:PmemFunDefinition.cpp
class Simple2 {
public:
    int f(float) const { return 1; }
};

int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;

int main()
{
    fp = &Simple2::f;
} //:-~
```

Na przykładzie definicji wskaźnika **fp2** przekonujemy się, że wskaźnik do funkcji składowej może być również zainicjowany podczas tworzenia lub w dowolnym innym momencie. W odróżnieniu od funkcji niebędących funkcjami składowymi, użycie operatora & podczas pobierania adresu funkcji składowej *nie jest* opcjonalne. Można jednak podać identyfikator funkcji składowej bez listy argumentów, ponieważ w przypadku przeciążenia nazwy funkcji rozstrzygnięcia wolno dokonać na podstawie typu wskaźnika do składowej.

Przykład

Korzyścią wynikającą ze stosowania wskaźników jest możliwość zmiany tego, co wskazują, podczas pracy programu. W istotny sposób zapewnia to programowaniu większą elastyczność, gdyż dzięki wskaźnikom można w czasie działania programu określać lub zmieniać jego *zachowanie*. Podobnie wygląda sprawa ze wskaźnikami do składowych — pozwalają one na wybór składowej podczas pracy programu. W typowym przypadku w utworzonych przez ciebie klasach publicznie dostępne będą jedynie funkcje składowe (dane składowe są zazwyczaj traktowane jako część wewnętrznej implementacji), zatem poniższy program wybiera w trakcie działania wywoływaną funkcję składową:

```
//: C11:PointerToMemberFunction.cpp
#include <iostream>
using namespace std;
```

```

class Widget {
public:
    void f(int) const { cout << "Widget::f()\n"; }
    void g(int) const { cout << "Widget::g()\n"; }
    void h(int) const { cout << "Widget::h()\n"; }
    void i(int) const { cout << "Widget::i()\n"; }
};

int main() {
    Widget w;
    Widget* wp = &w;
    void (Widget::*pmem)(int) const = &Widget::h;
    (w.*pmem)(1);
    (wp->*pmem)(2);
} //:-/

```

Oczywiście, nie można oczekwać, że tak skomplikowane wyrażenia utworzy jakiś przypadkowy użytkownik. W przypadku gdy użytkownik musi bezpośrednio operować wskaźnikami do składowych, powinien wykorzystać do tego deklarację **typedef**. Aby pozbyć się takich niedogodności, można wykorzystać wskaźniki do składowych w charakterze mechanizmu należącego do wewnętrznej implementacji klasy. Poniżej przedstawiono poprzedni przykład z zastosowaniem wskaźników do składowych *wewnątrz klasy*. Użytkownik przekazuje jedynie numer, wybierając za jego pomocą wywoływaną funkcję²:

```

//: C11:PointerToMemberFunction2.cpp
#include <iostream>
using namespace std;

class Widget {
    void f(int) const { cout << "Widget::f()\n"; }
    void g(int) const { cout << "Widget::g()\n"; }
    void h(int) const { cout << "Widget::h()\n"; }
    void i(int) const { cout << "Widget::i()\n"; }
    enum { cnt = 4 };
    void (Widget::*fptr[cnt])(int) const;
public:
    Widget() {
        fptr[0] = &Widget::f; // Wymagana pełna specyfikacja
        fptr[1] = &Widget::g;
        fptr[2] = &Widget::h;
        fptr[3] = &Widget::i;
    }
    void select(int i, int j) {
        if(i < 0 || i > cnt) return;
        (this->*fptr[i])(j);
    }
    int count() { return cnt; }
};

int main() {
    Widget w;
    for(int i = 0; i < w.count(); i++)
        w.select(i, 47);
} //:-/

```

² Za ten przykład dziękuję Owenowi Mortensenowi.

Zarówno w interfejsie klasy, jak i w funkcji **main()** cała implementacja klasy, włącznie z jej funkcjami, została ukryta. Kod musi nawet prosić o podanie liczby dostępnych funkcji, wywołując w tym celu funkcję **count()**. Dzięki temu osoba implementująca klasę może zmienić liczbę funkcji zawartych w wewnętrznej implementacji klasy, nie naruszając kodu wykorzystującego tę klasę.

Inicjalizacja wskaźników do składowych, zawarta w konstruktorze, wydaje się określona w zbyt szczegółowy sposób. Czy nie można by napisać:

```
fptr[i] = &g;
```

g jest bowiem nazwą funkcji składowej i, w związku z tym, jest automatycznie wiadoczną w zasięgu klasy? Problem polega na tym, że nie odpowiadałoby to składni, stosowanej w przypadku wskaźników do składowych. Jest ona wymagana po to, by każdy — a w szczególności kompilator — potrafił określić, co się w tym miejscu dzieje. Podobnie gdy wyłuskiwane są wskaźniki do składowych:

```
(this->*fptr[i])(j);
```

postać instrukcji wydaje się znowu zbyt rozbudowana — wygląda tak, jakby wskaźnik **this** nie był w niej potrzebny. Również w tym przypadku składnia wymaga, by wskaźnik do składowej był, w momencie jego wyłuskania, zawsze przypisany do konkretnego obiektu.

Podsumowanie

Wskaźniki działają w języku C++ niemal tak samo, jak w języku C, co jest dobrą wiadomością. W przeciwnym razie wiele programów, napisanych w języku C, nie kompilowałoby się poprawnie w języku C++. Jedyne błędy, które wystąpią podczas komplikacji, będą spowodowane niebezpiecznymi operacjami przypisania. Jeżeli operacje te były naprawdę zamierzane, można uniknąć błędów zgłaszanych przez kompilator, używając prostego (i jawnego!) rzutowania.

Język C++ zawiera dodatkowo *referencje*, wywodzące się z takich języków programowania, jak Algol i Pascal. W istocie są one stałymi wskaźnikami, automatycznie wyłuskiwanymi przez kompilator. Referencje przechowują adresy, ale traktuje się je jak obiekty. Referencje mają zasadnicze znaczenie dla uproszczenia składni za pomocą przeciążania operatorów (co stanowi temat następnego rozdziału). Upraszczają one również składnię związaną z przekazywaniem i zwracaniem obiektów przez zwykłe funkcje.

Konstruktor kopiący pobiera jako argument referencję do istniejącego obiektu takiego samego typu i tworzy na jego podstawie nowy obiekt. Kompilator automatycznie wywołuje konstruktor kopiący w przypadku przekazywania lub zwracania obiektów przez wartość. Kompilator automatycznie tworzy konstruktory kopiące, **lecz jeśli** konstruktor taki jest potrzebny w projektowanej klasie, należy zawsze zdefiniować go samodzielnie, by mieć pewność, że działa on prawidłowo. Jeżeli natomiast nie chcesz, by obiekty były przekazywane lub zwracane przez wartość, to powinieneś utworzyć prywatny konstruktor kopiący.

Wskaźniki do składowych mają takie samo **działanie**, jak zwykłe wskaźniki. Pozwalają na wskazanie określonego obszaru pamięci (danych lub funkcji) w czasie pracy programu. Wskaźniki do składowych działają bowiem w stosunku do składowych klas, a nie do globalnych danych lub funkcji. Wprowadzają one do programowania większą elastyczność, pozwalając na zmianę zachowania programu podczas jego pracy.

Ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Przekształć fragment kodu „ptak i skała”, znajdujący się na początku rozdziału, w program w języku C (używając struktur jako typów danych) i pokaż, że program ten się kompliluje. Następnie spróbuj go skompilować, używając kompilatora języka C++, i zobacz, co się stanie.
2. Wykorzystaj fragmenty programu znajdującego się na początku podrozdziału „Referencje w C++” i umieść je w funkcji **main()**. Dodaj do nich instrukcje, drukujące informacje, dzięki którym można będzie przekonać się, że referencje przypominają automatycznie wyłuskiwane wskaźniki.
3. Napisz program, w którym spróbujesz: 1. Utworzyć referencję niezainicjowaną podczas tworzenia. 2. Zmienić utworzoną już referencję w taki sposób, aby odwoływała się do innego obiektu. 3. Utworzyć **pustą referencję**.
4. Napisz funkcję pobierającą argument będący wskaźnikiem, która modyfikuje wskazywaną przez niego wartość, a następnie zwracającą pomocniczą referencję.
5. Utwórz **klasę** posiadającą kilka funkcji składowych, a następnie tak zmodyfikuj funkcję, utworzoną w poprzednim ćwiczeniu, by jej argument wskazywał obiekt tej klasy. Uczynь ten wskaźnik wskaźnikiem do stałej, zdefiniuj kilka funkcji **klasy jako funkcje stałe** i udowodnij, że wewnątrz funkcji wywoływane mogą być tylko stałe funkcje składowe. Przekształć wskaźnik będący argumentem funkcji w referencję.
6. Wykorzystaj fragmenty kodu znajdujące się na początku podrozdziału „Referencje do wskaźników” i przekształć je w program.
7. Utwórz funkcję, pobierającą jako argument referencję do wskaźnika, wskazującego wskaźnik i modyfikującą ten argument. Wywołaj tę funkcję w funkcji **main()**.
8. Utwórz funkcję, pobierającą argument typu **char&** i modyfikującą jego wartość. W funkcji **main()** wydrukuj wartość zmiennej znakowej (typu **char**), wywołaj dla tej zmiennej swoją funkcję, a następnie ponownie wydrukuj wartość tej zmiennej, by się przekonać, że rzeczywiście została ona zmodyfikowana. Jak wpływa to na czytelność twojego programu?

9. Utwórz klasę zawierającą stałe funkcje składowe i funkcje składowe niebędące stałymi. Napisz trzy funkcje, pobierając jako argumenty obiekt tej klasy — tak by pierwsza pobierała go przez wartość, druga przez referencję, a trzecia przez referencję do stałej. Spróbuj wywołać oba rodzaje funkcji składowych klasy wewnątrz tych funkcji i wyjaśnij uzyskane rezultaty.
10. (Nieco trudne) Napisz prostą funkcję, pobierającą jako argument liczbę całkowitą, inkrementującą jej wartość i zwracającą ją. Wywołaj tę funkcję w funkcji **main()**. Następnie dowiedz się, w jaki sposób wygenerować na wyjściu kompilatora kod w asemblerze i prześledź instrukcje tego kodu, aby zrozumieć, w jaki sposób przekazywane i zwracane są argumenty i jak przechowywane są na stosie zmienne lokalne.
11. Napisz funkcję przyjmującą argumenty typu **char**, **int**, **float** i **double**. Wygeneruj za pomocą kompilatora kod w asemblerze i odszukaj instrukcje umieszczające te argumenty na stosie przed wywołaniem funkcji.
12. Napisz funkcję zwracającą wartość typu **double**. Wygeneruj jej kod w języku asemblera i sprawdź, w jaki sposób zwracana jest wartość funkcji.
13. Wygeneruj kod asemblerowy programu **PassingBigStructures.cpp**. Prześledź i wyjaśnij sposób, w jaki używany przez ciebie kompilator generuje kod, umożliwiający przekazywanie i zwracanie dużych struktur.
14. Napisz prostą, rekurencyjną funkcję, dekrementującą swój argument i zwracającą wartość zero, gdy argument osiąga wartość zerową, a w przeciwnym przypadku wywołującą samą siebie. Wygeneruj kod tej funkcji w języku asemblera i wyjaśnij, w jaki sposób kod, wygenerowany przez kompilator, obsługuje rekurencję.
15. Napisz kod, który udowodni, że kompilator automatycznie generuje konstruktor kopiący, jeżeli nie zostanie on utworzony przez programistę. Udowodnij, że wygenerowany przez kompilator konstruktor przeprowadza kopowanie bitów w przypadku prostych typów, a dla typów zdefiniowanych przez użytkownika — wywołuje ich konstruktory kopiące.
16. Utwórz klasę **zawierającą konstruktorkopiujący**, informujący o swoim wywołaniu, przez wprowadzenie komunikatu do strumienia **cout**. Następnie napisz funkcję, pobierającą jako argument obiekt tej klasy przez wartość, i drugą funkcję — tworzącą lokalny obiekt tej klasy i zwracającą go przez wartość. Wywołaj obie te funkcje, by przekonać się, że konstruktor kopiący jest rzeczywiście niejawnie wywoływany podczas przekazywania i zwracania obiektów przez wartość.
17. Utwórz klasę **zawierającą składową typu double***. Konstruktor powinien inicjalizować tę składową, używając wywołania **new double** i przypisując przydzielonemu obszarowi pamięci wartość swojego argumentu. Destruktor powinien: drukować wartość, **wskazywaną przez składową klasy**, przypisywać wskazywanemu przez nią obszarowi wartość -1, użyć w stosunku do tego obszaru operatora **delete** i przypisać wskaźnikowi wartość zerową. Następnie utwórz funkcję, pobierającą obiekt utworzonej klasy przez wartość, i wywołaj ją w funkcji **main()**. Wyjaśnij, co się stało. Usuń problem, tworząc konstruktor kopiący.

18. Utwórz klasę zawierającą konstruktor, który wygląda tak, jak konstruktor kopiący, ale zawiera dodatkowy argument, posiadający domyślną wartość.
Wykaż, że jest on nadal używany w charakterze konstruktora kopiącego.
19. Utwórz klasę posiadającą konstruktor kopiący, informujący o swoim wywołaniu. Utwórz drugą klasę, zawierającą składową będącą obiektem pierwszej klasy, ale nie twórz dla niej konstruktora kopiącego. Pokaż, że konstruktor kopiący, wygenerowany dla drugiej klasy, automatycznie wywołuje konstruktor kopiący pierwszej klasy.
20. Utwórz bardzo prostą klasę oraz funkcję, zwracającą obiekt tej klasy przez wartość. Napisz drugą funkcję, pobierającą referencję do obiektu utworzonej klasy. Wywołaj pierwszą z tych funkcji jako argument drugiej funkcji i pokaż, że druga funkcja musi pobierać argument będący referencją do stałej.
21. Utwórz prostą klasę nieposiadającą konstruktora kopiącego i prostą funkcję, pobierającą jako argument obiekt tej klasy przez wartość. Zmień swoją klasę, dodając prywatną deklarację (tylko deklarację) konstruktora kopiącego. Wyjaśnij to, co dzieje się podczas komplikacji tej funkcji.
22. W tym ćwiczeniu tworzony jest rozwiązywanie konkurencyjne w stosunku do użycia konstruktora kopiącego. Utwórz klasę X i zadeklaruj jej prywatny konstruktor kopiący (ale nie definiuj go). Utwórz publiczną funkcję **clone()** będącą stałą funkcją składową, która zwraca kopię obiektu, utworzoną za pomocą operatora new. Następnie napisz funkcję, pobierającą argument typu **const X&** i tworzącą za pomocą funkcji **clone()**, jego lokalną kopię, która może być modyfikowana. Ważną takiego rozwiązania jest konieczność jawnego zniszczenia utworzonej kopii obiektu (za pomocą operatora delete), gdy nie jest on już potrzebny.
23. Wyjaśnij, na czym polegają problemy występujące w plikach **Mem.cpp** oraz **MemTest.cpp**, zamieszczonych w rozdziale 7. Popraw pliki w taki sposób, by rozwiązać te problemy.
24. Utwórz klasę zawierającą składową typu **double** i funkcję **print()** drukującą wartość tej składowej. W funkcji **main()** utwórz wskaźniki — zarówno do danej, jak i funkcji składowej klasy. Utwórz obiekt tej klasy oraz wskaźnik do tego obiektu, a następnie odwołuj się do obu elementów składowych klasy za pomocą wskaźników do składowych, używając zarówno obiektu, jak i wskaźnika do obiektu.
25. Utwórz klasę zawierającą tablicę liczb całkowitych. Czy można indeksować tę tablicę, wykorzystując wskaźnik do składowej klasy?
26. Zmodyfikuj program **PmemFunDefinition.cpp**, dodając przeciążoną funkcję składową **f()** (możesz dowolnie określić listę argumentów tej funkcji, umożliwiającej przeciążenie). Następnie utwórz drugi wskaźnik do składowej, przypisując mu przeciążoną wersję funkcji **f()**, i wywołaj funkcję za pomocą tego wskaźnika. W jaki sposób rozróżniane są w tym przypadku przeciążone funkcje?

27. Rozpocznij od programu **FunctionTable.cpp**, zamieszczonego w rozdziale 3.
Utwórz klasę zawierającą wektor (utworzony za pomocą kontenera **vector**) wskaźników do funkcji, wykorzystując funkcje składowe **add()** i **remove()** do dodawania i usuwania poszczególnych wskaźników do funkcji. Dodaj funkcję **run()**, przechodzącą przez wszystkie elementy zawarte w wektorze i wywołującą wskazywane przez nie funkcje.
28. Zmodyfikuj poprzednie ćwiczenie w taki sposób, aby program działał ze wskaźnikami do funkcji składowych.

Rozdział 12.

Przeciążanie operatorów

Przeciążanie operatorów jest nazywane „cukierkiem składniowym”, co oznacza, że jest ono innym sposobem wywołania funkcji.

Różnica polega na tym, że argumenty tej funkcji nie są umieszczone w nawiasie, natomiast otaczają znaki albo też sąsiadują z nimi. Tymczasem zawsze uważaliśmy za niepodlegające modyfikacjom operatory.

Istnieją dwie różnice pomiędzy użyciem operatorów i zwykłym wywołaniem funkcji. Po pierwsze, różni je składnia — operator jest często „wywoływany” poprzez umieszczenie go pomiędzy argumentami, a czasami również za lub przed nimi. Druga różnica polega na tym, że to kompilator decyduje o tym, którą „funkcję” ma wywołać. Na przykład jeżeli operator + używany jest z argumentami zmiennopozycyjnymi, to kompilator „wywołuje” funkcję, realizującą operację dodawania zmiennopozycyjnego („wywołanie” to polega na ogół na wstawieniu odpowiedniego kodu lub instrukcji procesora zmiennopozycyjnego). Jeżeli operator + zostanie użyty wraz z liczbą **zmiennopozycyjną** oraz liczbą **całkowitą**, to kompilator „wywoła” specjalną funkcję, przekształcającą liczbę typu int w liczbę typu float, a następnie „wywoła” kod wykonujący dodawanie zmiennopozycyjne.

Jednak w języku C++ możliwe jest zdefiniowanie nowych operatorów, działających w stosunku do klas. Definicja ta przypomina zwykłą definicję funkcji, z wyjątkiem tego, że nazwa funkcji składa się ze słowa kluczowego operator, po którym następuje operator. To jedyna różnica; poza tym staje się ona taką funkcją, jak każda inna wywoływana przez kompilator, gdy widzi on odpowiadający wzorzec.

Ostrzeżenie i wyjaśnienie

Łatwo jest wpaść w nadmierny entuzjazm związany z przeciążaniem operatorów. Na pierwszy rzut oka wydaje się to świetną zabawą. Trzeba jednak pamiętać, że to *tylko „cukierek składniowy”* — inny sposób wywołania funkcji. Z tego punktu widzenia nie ma żadnego powodu, by przeciążać operator, z wyjątkiem przypadku, gdy kod

zawierający odwołania do klasy będzie dzięki temu łatwiejszy do napisania, a szczególnie — łatwiejszy do *przeczytania* (pamiętaj, że kod jest znacznie częściej czytany niż pisany). Jeżeli ten przypadek nie zachodzi, nie warto zaprzątać sobie tym uwagi.

Innym często spotykanym podejściem do przeciążania operatorów jest panika — nagle okazało się, że operatory języka C utraciły swe dobrze poznane znaczenie. „Wszystko się zmieniło i cały mój kod, napisany w języku C, będzie działał zupełnie inaczej P! Nie jest to prawda. Żaden z operatorów, wykorzystywanych w wyrażenях zawierających jedynie wbudowane typu danych, nie mógł ulec zmianie. Nie sposób nigdy przeciążyć w taki sposób operatorów, by wyrażenie:

`1 << 4;`

zachowywało się odmiennie lub wyrażenie:

`1.414 << 2;`

miało jakikolwiek sens. Jedynie wyrażenia obejmujące typy zdefiniowane przez użytkownika mogą zawierać przeciążone operatory.

Składnia

Definicja przeciążonego operatora przypomina definicję funkcji, lecz nazwa tej funkcji ma postać **operator@**, gdzie znak @ reprezentuje przeciążany operator. Liczba argumentów, znajdujących się na liście argumentów przeciążanego operatora, zależy od dwóch czynników:

- ±. Czy jest to operator **jednoargumentowy** (**jeden** argument), czy też **dwuargumentowy** (dwa argumenty).
- 2. Czy operator jest **zdefiniowany jako funkcja globalna** (**jeden** argument
 - w przypadku operatora **jednoargumentowego**, dwa argumenty — w przypadku operatora **dwuargumentowego**), czy też jako funkcja składowa (brak argumentu)
 - w przypadku operatora jednoargumentowego; jeden argument — w przypadku operatora dwuargumentowego, obiekt staje się wówczas argumentem lewostronnym).

Poniżej przedstawiono niewielką klasę prezentującą składnię związaną z przeciążaniem operatorów:

```
//: C12:OperatorOverloadingSyntax.cpp
#include <iostream>
using namespace std;

class Integer {
    int i;
public:
    Integer(int ii) : i(ii) {}
    const Integer
    operator+(const Integer& rv) const {
        cout << "operator+" << endl;
        return Integer(i + rv.i);
    }
}
```

```
Integer&
operator+=(const Integer& rv) {
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
}
};

int main() {
    cout << "typy wbudowane:" << endl;
    int i = 1, j = 2, k = 3;
    k += i + j;
    cout << "typy zdefiniowane przez użytkownika:" << endl;
    Integer ii(1), jj(2), kk(3);
    kk += ii + jj;
} //:~
```

Oba przeciążone operatory zdefiniowano jako funkcje składowe inline, informujące o tym, że zostały wywołane. Pojedynczy argument jest tym, co pojawia się po prawej stronie operatora dwuargumentowego. **Operatory jednoargumentowe**, definiowane jako funkcje składowe, nie mają w ogóle argumentów. Funkcja składowa jest wywoływana dla obiektu znajdującego się po lewej stronie operatora.

W przypadku operatorów niebędących operatorami warunkowymi (operatorы warunkowe zwracają zazwyczaj wartości typu **bool**) niemal zawsze zwracany jest obiekt lub referencja do obiektu typu, na którym dokonuje się operacji, o ile typy argumentów są takie same (jeżeli nie są one identyczne, to interpretacja, co powinien zwracać operator, należy do programisty). Dzięki temu mogą być tworzone złożone wyrażenia;

```
kk += ii + jj;
```

W powyższym przykładzie **operator+** zwraca nowy (tymczasowy) obiekt typu **Integer**, który jest używany jako argument **rv** funkcji **operator+=**. Obiekt tymczasowy jest niszczony, kiedy nie jest już potrzebny.

Operatorы, które można przeciążać

Mimo że można przeciążać niemal wszystkie operatory, dostępne w języku C, to proces ten podlega dość ścisłym ograniczeniom. W szczególności nie można łączyć ze sobą operatorów, które nie posiadają znaczenia w języku C (np. tworzyć operatora ****** w celu oznaczenia potęgowania), nie wolno zmieniać kolejności obliczania operatorów, a także liczby wymaganych przez nie argumentów. Jest to logiczne — wszelkie takie działania prowadziłyby do utworzenia operatorów, które pogarszałyby czytelność kodu, zamiast ją poprawiać.

Dwa następne podrozdziały prezentują przykłady wszystkich „normalnych” operatorów, **przeciążonych** w taki sposób, w jaki najprawdopodobniej będą używane.

Operatory jednoargumentowe

Zamieszczony poniżej przykładowy program przedstawia składnię, umożliwiającą przeciążenie wszystkich operatorów jednoargumentowych, zarówno w postaci funkcji globalnych (funkcji zaprzyjaźnionych, **niebędących** funkcjami składowymi), jak i funkcji składowych. Stanowią one rozwinięcie, przedstawionej uprzednio, klasy **Integer** oraz wprowadzają nową klasę **Byte**. Znaczenie poszczególnych operatorów będzie zależało od sposobu ich użycia; zanim jednak zrobi się coś zaskakującego, należy zawsze mieć na uwadze klienta-programistę.

Poniżej znajduje się katalog zawierający wszystkie **funkcje jednoargumentowe**:

```
//: C12:OverloadingUnaryOperators.cpp
#include <iostream>
using namespace std;

// Funkcje niebędące funkcjami składowymi:
class Integer {
    long i;
    Integer* This() { return this; }
public:
    Integer(long ii = 0) : i(ii) {}
    // Brak skutków ubocznych - argumenty są
    // referencjami do stałych:
    friend const Integer&
        operator+(const Integer& a);
    friend const Integer
        operator-(const Integer& a);
    friend const Integer
        operator~(const Integer& a);
    friend Integer*
        operator&(Integer&a);
    friend int
        operator!(const Integer& a);
    // Występują skutki uboczne - argumenty nie są
    // referencjami do stałych:
    // Operator przedrostkowy:
    friend const Integer&
        operator++(Integer& a);
    // Operator przyrostkowy:
    friend const Integer
        operator++(Integer&a, int);
    // Operator przedrostkowy:
    friend const Integer&
        operator--(Integer& a);
    // Operator przyrostkowy:
    friend const Integer
        operator--(Integer&a, int);
};

// Operatory globalne:
const Integer& operator+(const Integer& a) {
    cout << "+Integer\n";
    return a; // Jednoargumentowy operator + nic nie robi
}
```

```
const Integer operator-(const Integer& a) {
    cout << "-Integer\n";
    return Integer(-a.i);
}
const Integer operator~(const Integer& a) {
    cout << "~Integer\n";
    return Integer(~a.i);
}
Integer* operator&(Integer& a) {
    cout << "&Integer\n";
    return a.This(); // &a jest rekurencyjne!
}
int operator!(const Integer& a) {
    cout << "!Integer\n";
    return !a.i;
}
// Operator przedrostkowy - zwraca wartość po inkrementacji:
const Integer& operator++(Integer& a) {
    cout << "++Integer\n";
    a.i++;
    return a;
}
// Operator przyrostkowy - zwraca wartość przed inkrementacją:
const Integer operator++(Integer& a, int) {
    cout << "Integer++\n";
    Integer before(a.i);
    a.i++;
    return before;
}
// Operator przedrostkowy - zwraca wartość po dekrementacji:
const Integer& operator--(Integer& a) {
    cout << "--Integer\n";
    a.i--;
    return a;
}
// Operator przyrostkowy - zwraca wartość przed dekrementacją:
const Integer operator--(Integer& a, int) {
    cout << "Integer--\n";
    Integer before(a.i);
    a.i--;
    return before;
}

// Prezentacja działania przekształcanych operatorów:
void f(Integer a) {
    +a;
    -a;
    ~a;
    Integer* ip = &a;
    !a;
    ++a;
    a++;
    --a;
    a--;
}
```

```
// Funkcje składowe (niejawny argument "this"):  
class Byte {  
    unsigned char b;  
public:  
    Byte(unsigned char bb = 0) : b(bb) {}  
    // Brak skutków ubocznych - stałe funkcje składowe:  
    const Byte& operator+() const {  
        cout << "+Byte\n";  
        return *this;  
    }  
    const Byte operator-() const {  
        cout << "-Byte\n";  
        return Byte(-b);  
    }  
    const Byte operator~() const {  
        cout << "~Byte\n";  
        return Byte(~b);  
    }  
    Byte operator!() const {  
        cout << "!Byte\n";  
        return Byte(!b);  
    }  
    Byte* operator&() {  
        cout << "&Byte\n";  
        return this;  
    }  
    // Występują skutki uboczne - funkcje składowe  
    // nie będące stałymi:  
    const Byte& operator++() { // Przedrostek  
        cout << "++Byte\n";  
        b++;  
        return *this;  
    }  
    const Byte operator++(int) { // Przyrostek  
        cout << "Byte++\n";  
        Byte before(b);  
        b++;  
        return before;  
    }  
    const Byte& operator--() { // Przedrostek  
        cout << "--Byte\n";  
        --b;  
        return *this;  
    }  
    const Byte operator--(int) { // Przyrostek  
        cout << "Byte--\n";  
        Byte before(b);  
        --b;  
        return before;  
    }  
};  
  
void g(Byte b) {  
    +b;  
    -b;  
    -b;  
    Byte* bp = &b;  
    !b;
```

```
++b;  
b++;  
--b;  
b--;  
}  
  
int main() {  
    Integer a;  
    f(a);  
    Byte b;  
    g(b);  
} //:-
```

Funkcje zostały pogrupowane według sposobu przekazywania argumentów. Wskazówki dotyczące przekazywania i zwracania argumentów zostaną przedstawione później. Przykłady przedstawione powyżej (oraz zawarte w następnym podrozdziale) ilustrują typowe sposoby **używania** przeciążonych operatorów, więc można potraktować je jako wzorce podczas przeciążania własnych operatorów.

Inkrementacja i dekrementacja

Przeciążone operatory **++** oraz **-** stanowią pewien problem, ponieważ zamierzamy wywołać różne funkcje, w zależności od tego, czy występują one przed (przedrostek), czy też za (przyrostek) obiektem, na którym działają. Rozwiążanie jest proste, choć niektórym wydaje się ono na początku nieco zagmatwane. Gdy kompilator widzi, na przykład, wyrażenie **++a** (**preinkrementacja**), generuje **wywołanie** funkcji **operator++(a)**, natomiast widząc wyrażenie **a++** — funkcji **operator++(a, int)**. Oznacza to, że kompilator rozróżnia te dwie postacie operatora, wywołując różne przeciążone funkcje. W przypadku funkcji składowych klasy, widocznych w pliku **OverloadingUnaryOperators.cpp**, kompilator widząc wyrażenie **++b** generuje wywołanie funkcji **B::operator++()**, natomiast widząc wyrażenie **b++** wywołuje funkcję **B::operator(int)**.

Użytkownik widzi jedynie, że dla przedrostkowych i przyrostkowych wersji operatorów wywoływanie są różne funkcje. Jednak w rzeczywistości wywołania obu tych funkcji posiadają odmienne sygnatury, więc są połączone z dwoma zupełnie różnymi ciałami funkcji. Kompilator przekazuje sztuczną, stałą wartość argumentu typu **int** (który nie posiada identyfikatora, ponieważ jego wartość nie jest nigdy wykorzystywana), by dla przyrostkowej wersji funkcji operatora utworzyć inną sygnature.

Operatorы dwuargumentowe

Poniższy program stanowi powtórzenie zawartego w pliku **OverloadingUnaryOperators.cpp** przykładu przeciążania operatorów **jednoargumentowych**, tym razem dla operatorów **dwuargumentowych**. Dzięki temu jest ilustracją wykorzystania wszystkich operatorów, które mogą być przeciążane.

Ponownie zostały one przedstawione zarówno w postaci funkcji globalnych, jak i funkcji składowych.

```
//: C12:Integer.h
// Przeciążone operatory niebędące
// funkcjami składowymi
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>

// Funkcje niebędące funkcjami składowymi:
class Integer {
    long i;
public:
    Integer(long ii = 0) : i(ii) {}
    // Operatory tworzące nową zmodyfikowaną wartość:
    friend const Integer
        operator+(const Integer& left,
                   const Integer& right);
    friend const Integer
        operator-(const Integer& left,
                   const Integer& right);
    friend const Integer
        operator*(const Integer& left,
                   const Integer& right);
    friend const Integer
        operator/(const Integer& left,
                   const Integer& right);
    friend const Integer
        operator%(const Integer& left,
                   const Integer& right);
    friend const Integer
        operator^(const Integer& left,
                   const Integer& right);
    friend const Integer
        operator&(const Integer& left,
                   const Integer& right);
    friend const Integer
        operator|(const Integer& left,
                   const Integer& right);
    friend const Integer
        operator<<(const Integer& left,
                   const Integer& right);
    friend const Integer
        operator>>(const Integer& left,
                   const Integer& right);
    // Przypisania modyfikujące i zwracające l-wartość:
    friend Integer&
        operator+=(Integer& left,
                   const Integer& right);
    friend Integer&
        operator-=(Integer& left,
                   const Integer& right);
    friend Integer&
        operator*=(Integer& left,
                   const Integer& right);
    friend Integer&
        operator/=(Integer& left,
                   const Integer& right);
    friend Integer&
        operator%=(Integer& left,
                   const Integer& right);
```

```
friend Integer&
operator^=(Integer& left,
            const Integer& right);
friend Integer&
operator&=(Integer& left,
            const Integer& right):
friend Integer&
operator|= (Integer& left,
            const Integer& right);
friend Integer&
operator>>=(Integer& left,
            const Integer& right):
friend Integer&
operator<=(Integer& left,
            const Integer& right);
// Operatory warunkowe, zwracające wartości true lub false:
friend int
operator==(const Integer& left,
            const Integer& right);
friend int
operator!=(const Integer& left,
            const Integer& right);
friend int
operator<(const Integer& left,
            const Integer& right);
friend int
operator>(const Integer& left,
            const Integer& right);
friend int
operator<=(const Integer& left,
            const Integer& right);
friend int
operator>=(const Integer& left,
            const Integer& right);
friend int
operator&&=(const Integer& left,
            const Integer& right);
friend int
operator||=(const Integer& left,
            const Integer& right);
// Zapis wartości do strumienia wyjściowego:
void print(std::ostream& os) const { os << i; }
}i
#endif // INTEGER_H //://-
//: C12:Integer.cpp {0}
// Implementacja przeciążonych operatorów
#include "Integer.h"
#include "../require.h"

const Integer
operator+(const Integer& left,
            const Integer& right) {
    return Integer(left.i + right.i);
}
const Integer
operator-(const Integer& left,
            const Integer& right) {
```

```
    return Integer(left.i - right.i);
}
const Integer
operator*(const Integer& left,
           const Integer& right) {
    return Integer(left.i * right.i);
}
const Integer
operator/(const Integer& left,
           const Integer& right) {
    require(right.i != 0, "dzielenie przez zero");
    return Integer(left.i / right.i);
}
const Integer
operator%(const Integer& left,
           const Integer& right) {
    require(right.i != 0, "modulo zero");
    return Integer(left.i % right.i);
}
const Integer
operator^(const Integer& left,
           const Integer& right) {
    return Integer(left.i ^ right.i);
}
const Integer
operator&(const Integer& left,
           const Integer& right) {
    return Integer(left.i & right.i);
}
const Integer
operator|(const Integer& left,
           const Integer& right) {
    return Integer(left.i | right.i);
}
const Integer
operator<<(const Integer& left,
            const Integer& right) {
    return Integer(left.i << right.i);
}
const Integer
operator>>(const Integer& left,
            const Integer& right) {
    return Integer(left.i >> right.i);
}
// Przypisania modyfikujące i zwracające l-wartość:
Integer& operator+=(Integer& left,
                       const Integer& right) {
    if(&left == &right) /* przypisanie do samego siebie */
        left.i += right.i;
    return left;
}
Integer& operator-=(Integer& left,
                       const Integer& right) {
    if(&left == &right) /* przypisanie do samego siebie */
        left.i -= right.i;
    return left;
}
```

```
Integer& operator*=(Integer& left,
                     const Integer& right) {
    if(&left == &right) /* przypisanie do samego siebie */
        left.i *= right.i;
    return left;
}
Integer& operator/=(Integer& left,
                     const Integer& right) {
    require(right.i != 0. "dzielenie przez zero");
    if(&left == &right) /* przypisanie do samego siebie */
        left.i /= right.i;
    return left;
}
Integer& operator%=(Integer& left,
                     const Integer& right) {
    require(right.i != 0. "modulo zero");
    if(&left == &right) /* przypisanie do samego siebie */
        left.i %= right.i;
    return left;
}
Integer& operator^=(Integer& left,
                     const Integer& right) {
    if(&left == &right) /* przypisanie do samego siebie */
        left.i ^= right.i;
    return left;
}
Integer& operator&=(Integer& left,
                     const Integer& right) {
    if(&left == &right) /* przypisanie do samego siebie */
        left.i &= right.i;
    return left;
}
Integer& operator|=(Integer& left,
                     const Integer& right) {
    if(&left == &right) /* przypisanie do samego siebie */
        left.i |= right.i;
    return left;
}
Integer& operator>=(Integer& left,
                     const Integer& right) {
    if(&left == &right) /* przypisanie do samego siebie */
        left.i >>= right.i;
    return left;
}
Integer& operator<=(Integer& left,
                     const Integer& right) {
    if(&left == &right) /* przypisanie do samego siebie */
        left.i <=> right.i;
    return left;
}
// Operatory warunkowe, zwracające wartości true lub false:
int operator==(const Integer& left,
                  const Integer& right) {
    return left.i == right.i;
}
int operator!=(const Integer& left,
                const Integer& right) {
```

```

        return left.i != right.i;
    }
int operator<(const Integer& left,
                 const Integer& right) {
    return left.i < right.i;
}
int operator>(const Integer& left,
                 const Integer& right) {
    return left.i > right.i;
}
int operator<=(const Integer& left,
                 const Integer& right) {
    return left.i <= right.i;
}
int operator>=(const Integer& left,
                 const Integer& right) {
    return left.i >= right.i;
}
int operator&&(const Integer& left,
                 const Integer& right) {
    return left.i && right.i;
}
int operator||(const Integer& left,
                 const Integer& right) {
    return left.i || right.i;
} //:-  

//: C12:IntegerTest.cpp
//{L} Integer
#include "Integer.h"
#include <iostream>
using namespace std;
ofstream out("IntegerTest.out");

void h(Integer& c1, Integer& c2) {
    // Złożone wyrażenie:
    c1 += c1 * c2 + c2 % c1;
#define TRY(OP) \
    out << "c1 = "; c1.print(out); \
    out << ", c2 = "; c2.print(out); \
    out << "; c1 " #OP " c2 daje "; \
    (c1 OP c2).print(out); \
    out << endl;
    TRY(+) TRY(-) TRY(*) TRY(/)
    TRY(%) TRY(^) TRY(&) TRY(|)
    TRY(<<) TRY(>>) TRY(+=) TRY(-=)
    TRY(*=) TRY(/=) TRY(%=) TRY(^=)
    TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
    // Operatory warunkowe:
#define TRYC(OP) \
    out << "c1 = "; c1.print(out); \
    out << ", c2 = "; c2.print(out); \
    out << "; c1 " #OP " c2 daje "; \
    out << (c1 OP c2); \
    out << endl;
    TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
    TRYC(>=) TRYC(&&) TRYC(||)
}

```

```
int main() {
    out << "funkcje zaprzyczajone:" << endl;
    Integer c1(47).c2(9);
    h(c1, c2);
} //:-
li: C12[Byte.h]
// Przeciążone operatory będące funkcjami składowymi
#ifndef BYTE_H
#define BYTE_H
#include "../require.h"
#include <iostream>
// Funkcje składowe (niejawny argument "this"):
class Byte {
    unsigned char b;
public:
    Byte(unsigned char bb = 0) : b(bb) {}
    // Brak skutków ubocznych - stałe funkcje składowe:
    const Byte
        operator+(const Byte& right) const {
            return Byte(b + right.b);
        }
    const Byte
        operator-(const Byte& right) const {
            return Byte(b - right.b);
        }
    const Byte
        operator*(const Byte& right) const {
            return Byte(b * right.b);
        }
    const Byte
        operator/(const Byte& right) const {
            require(right.b != 0, "dzielenie przez zero");
            return Byte(b / right.b);
        }
    const Byte
        operator%(const Byte& right) const {
            require(right.b != 0, "modulo zero");
            return Byte(b % right.b);
        }
    const Byte
        operator^(const Byte& right) const {
            return Byte(b ^ right.b);
        }
    const Byte
        operator&(const Byte& right) const {
            return Byte(b & right.b);
        }
    const Byte
        operator|(const Byte& right) const {
            return Byte(b | right.b);
        }
    const Byte
        operator<<(const Byte& right) const {
            return Byte(b << right.b);
        }
    const Byte
        operator>>(const Byte& right) const {
```

```
        return Byte(b >> right.b);
    }
    // Przypisania modyfikujące i zwracające 1-wartość:
    // operator= może być tylko funkcją składową:
Byte& operator=(const Byte& right) {
    // Obsługa przypisania do siebie samego:
    if(this == &right) return *this;
    b = right.b;
    return *this;
}
Byte& operator+=(const Byte& right) {
    if(this == &right) /* przypisanie do siebie samego */
    b += right.b;
    return *this;
}
Byte& operator-=(const Byte& right) {
    if(this == &right) /* przypisanie do siebie samego */
    b -= right.b;
    return *this;
}
Byte& operator*=(const Byte& right) {
    if(this == &right) /* przypisanie do siebie samego */
    b *= right.b;
    return *this;
}
Byte& operator/=(const Byte& right) {
    require(right.b != 0, "dzielenie przez zero");
    if(this == &right) /* przypisanie do siebie samego */
    b /= right.b;
    return *this;
}
Byte& operator%=(const Byte& right) {
    require(right.b != 0, "modulo zero");
    if(this == &right) /* przypisanie do siebie samego */
    b %= right.b;
    return *this;
}
Byte& operator^=(const Byte& right) {
    if(this == &right) /* przypisanie do siebie samego */
    b ^= right.b;
    return *this;
}
Byte& operator&=(const Byte& right) {
    if(this == &right) /* przypisanie do siebie samego */
    b &= right.b;
    return *this;
}
Byte& operator|=(const Byte& right) {
    if(this == &right) /* przypisanie do siebie samego */
    b |= right.b;
    return *this;
}
Byte& operator>>=(const Byte& right) {
    if(this == &right) /* przypisanie do siebie samego */
    b >>= right.b;
    return *this;
}
```

```
Byte& operator<=(const Byte& right) {
    if(th1s == &right) /* przypisanie do siebie samego */
        b <<- right.b;
    return *this;
}
// Operatory warunkowe, zwracające wartości true lub false:
int operator==(const Byte& right) const {
    return b == right.b;
}
int operator!=(const Byte& right) const {
    return b != right.b;
}
int operator<(const Byte& right) const {
    return b < right.b;
}
int operator>(const Byte& right) const {
    return b > right.b;
}
int operator<=(const Byte& right) const {
    return b <= right.b;
}
int operator>=(const Byte& right) const {
    return b >= right.b;
}
int operator&&(const Byte& right) const {
    return b && right.b;
}
int operator||(const Byte& right) const {
    return b || right.b;
}
// Zapis wartości do strumienia wyjściowego:
void print(std::ostream& os) const {
    os << "0x" << std::hex << int(b) << std::dec;
}
};

#endif // BYTE_H //:-

//: C12:ByteTest.cpp
#include "Byte.h"
#include <iostream>
using namespace std;
ofstream out("ByteTest.out");

void k(Byte& b1, Byte& b2) {
    b1 = b1 * b2 + b2 % b1;

#define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ", b1 " #OP " b2 daje "; \
    (b1 OP b2).print(out); \
    out << endl;

    b1 = 9; b2 = 47;
    TRY2(+)
    TRY2(-)
    TRY2(*)
    TRY2(/)
    TRY2(%)
    TRY2(^)
    TRY2(&)
    TRY2(||)
    TRY2(<<)
    TRY2(>>)
    TRY2(+=)
    TRY2(-=)
    TRY2(*=)
    TRY2(/=)
    TRY2(%=)
    TRY2(^=)
```

```

TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
TRY2(=) // Operator przypisania

// Operatory warunkowe:
#define TRYC2(OP) \
    out << "bl - "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << "; bl " #OP " b2 daje "; \
    out << (b1 OP b2); \
    out << endl;

bl - 9; b2 - 47;
TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
TRYC2(>=) TRYC2(&&) TRYC2(||)

// Przypisanie łańcuchowe:
Byte b3 = 92;
b1 = b2 - b3;
}

int main() {
    out << "funkcje składowe:" << endl;
    Byte b1(47). b2(9);
    k(b1, b2);
} //:-/

```

Funkcja **operator=** może zatem występować wyłącznie w postaci funkcji składowej. Zostanie to wyjaśnione poniżej.

Zwróć uwagę na to, że wszystkie operatory przypisania zawierają kod, który sprawdza, czy nie następuje przypisanie do samego siebie — jest to wskazówka o charakterze ogólnym. W pewnych sytuacjach sprawdzanie takie nie jest konieczne — na przykład w przypadku funkcji **operator+=** często możemy *celowo* napisać po prostu A += A, dodając zmienną A do samej siebie. Najważniejszym miejscem, w którym należy sprawdzić, czy nie następuje przypisanie do samego siebie, jest **operator=**. Przypisanie takie mogłoby bowiem powodować dla obiektu katastrofalne skutki (w niektórych przypadkach to dopuszczalne, ale zawsze należy o tym pamiętać, tworząc **operator=**).

Wszystkie operatory, przedstawione w poprzednich dwóch przykładach, zostały przeciążone do obsługi pojedynczego typu danych. Możliwe jest również takie przeciążenie operatorów, by obsługiwały różne typy, co pozwala, na przykład, na dodawanie jabłek do pomarańcz. Zanim jednak rozpoczniesz przeciążanie operatorów, obejmujące wszystkie możliwe przypadki, powinieneś przeglądając podrozdział dotyczący automatycznej konwersji typów, znajdujący się w dalszej części rozdziału. Często właściwie umiejscowiona konwersja typów pozwala na uniknięcie przeciążania mnóstwa operatorów.

Argumenty i zwracane wartości

Rozmaite sposoby przekazywania i zwracania wartości, widoczne podczas przeglądania plików **OverloadingUnaryOperators.cpp**, **Integer.h** oraz **Byte.h**, mogą wydawać się na pierwszy rzut oka niezrozumiałe. Mimo że można dowolnie przekazywać i zwracać

wartości, to sposoby, prezentowane w tych przykładach, nie zostały wybrane przypadkowo. Odpowiadają one pewnemu logicznemu wzorowi, który zapewne zechcesz wykorzystać w większości przypadków:

1. Podobnie jak w przypadku każdego innego argumentu funkcji, jeżeli będziesz chciał ty i kogo odczytać jego wartość, nie zmieniającej, to domyślnie powinien on zostać przekazany jako referencja do stałej. Zwyczajne operacje arytmetyczne (jak +, - itd.) oraz logiczne nie zmieniają wartości swoich argumentów, więc zazwyczaj będziesz przekazywał je w postaci referencji do stałych. Gdy funkcja jest składową klasy, odpowiada to użyciu jej funkcją stałą. Jedynie w przypadku operatorów połączonych z przypisaniem (jak np. +=) oraz funkcji **operator=**, zmieniających argument znajdujący się po lewej stronie, argument po lewej stronie *nie jest* stałą. Jednak ponieważ jest on modyfikowany, nadal przekazywany jest w postaci adresu.
2. Typ wartości zwracanej przez operator powinien zależeć od spodziewanego znaczenia operatora (podobnie jak w przypadku typów argumentów, również typ zwracanej wartości może być dowolny). Jeżeli na skutek działania operatora powstaje nowa wartość, to aby ją zwrócić, trzeba utworzyć nowy obiekt. Na przykład funkcja **Integer::operator+** musi zwracać obiekt klasy **Integer**, będący sumą swoich argumentów. Obiekt ten jest zwracany przez wartość jako stała, więc rezultat operacji nie może być modyfikowany jako l-wartość.
3. Wszystkie operatory przypisania modyfikują l-wartość. Aby rezultat operacji mógł być użyty w wyrażeniach łańcuchowych, takich jak $a=b=c$, można się spodziewać, że zwrócona zostanie referencja do tej samej l-wartości, właśnie zmodyfikowana. Ale czy referencja ta powinna być referencją do stałej, czy też nie? Mimo że wyrażenie $a=b=c$ jest czytane od lewej do prawej, to kompilator analizuje od prawej do lewej. W celu zapewnienia poprawnego działania przypisania łańcuchowego nie istnieje więc konieczność zwracania wartości, która nie byłaby stałą. Niektórzy użytkownicy oczekują jednak czasami możliwości wykonania operacji na obiekcie, do którego właśnie nastąpiło przypisanie, jak na przykład w przypadku instrukcji (**a=b.func()**), wywołującej funkcję dla obiektu a, po przypisaniu mu wartości b. Tak więc wartości zwracane przez wszystkie operatory przypisania powinny być referencjami do l-wartości, nie będącymi referencjami do stałych.
4. Wszyscy spodziewają się, że wynikiem działania operatorów logicznych będą przynajmniej liczby całkowite, a w idealnym przypadku — wartości typu **bool** (biblioteki, które zostały opracowane, zanim większość kompilatorów języka C++ zaczęły obsługiwać wbudowany typ **bool**, używały wartości całkowitych lub równoważnych im wartości, utworzonych za pomocą **typedef**).

Operatory inkrementacji i dekrementacji stanowią problem z uwagi na to, że występują one zarówno w wersji przedrostkowej, jak i przyrostkowej. Obie wersje zmieniają wartość obiektu, nie można więc traktować go jako stałej. Wersja przedrostkowa zwraca wartość obiektu po jego modyfikacji; można się więc spodziewać, że zwraca ona z powrotem zmieniony obiekt. Dlatego też, w przypadku przedrostkowej wersji operatora, można zwrócić wartość ***this** jako referencję. Natomiast jeśli chodzi o przyrostkową wersję operatora, spodziewane jest zwrócenie wartości obiektu, zanim jeszcze zostanie ona zmodyfikowana. Konieczne jest więc utworzenie oddzielnego obiektu,

reprezentującego tę wartość, i zwrócenie go. Tak więc w przypadku operatora przyrostkowego, dla zachowania spodziewanego znaczenia operacji, konieczne jest zwracanie obiektu przez wartość (warto też wiedzieć, że czasami występują operatory inkrementacji i dekrementacji, zwracające wartości całkowite lub logiczne — informują one, na przykład, że obiekt poruszający się wzduż listy znajduje się na jej końcu). Pytanie brzmi więc — czy powinny one być zwracane jako stałe, czy też nie? Jeżeli pozwolimy na modyfikację obiektu i ktoś użyje instrukcji w postaci **(*a*++).func()**, to funkcja **func()** będzie działała na obiekcie *a*, ale w razie zastosowania instrukcji **(*a*++).func()** funkcja **func()** działała będzie na tymczasowym obiekcie, zwróconym przez funkcję operatora przyrostkowego **operator++**. Obiekty tymczasowe są automatycznie stałymi, co powinno być zasygnalizowane przez kompilator. Jednakże w celu zapewnienia spójności bardziej logiczne wydaje się uczynienie ich w obu przypadkach stałymi, jak w zamieszczonych powyżej programach. Można również zdefiniować wersję przedrostkową operatora w taki sposób, że nie będzie on zwracał stałej; zwróci natomiast stałą w jego wersji przyrostkowej. Z uwagi na dużą liczbę znaczeń, które mogą zostać przypisane operatorom inkrementacji i dekrementacji, każdy taki przypadek należy potraktować indywidualnie.

Zwracanie przez wartość stałych

Zwracanie stałej przez wartość może się początkowo wydawać trochę dziwne, wymaga więc nieco obszerniejszego wyjaśnienia. Weźmy pod uwagę dwuargumentowy **operator+**. Jeżeli zostanie on zastosowany w takich wyrażeniach, jak **f(a+b)**, to wynik działania **a+b** stanie się obiektem tymczasowym, użytym do wywołania funkcji **f()**. Ponieważ to obiekt tymczasowy, więc automatycznie jest on obiektem stałym. A zatem nie ma żadnego znaczenia, czy zwracana przez operator wartość zostanie jawnie określona jako stała, czy też nie.

Możliwe jest jednak również wysłanie komunikatu do wartości zwracanej przez wyrażenie **a+b**, a nie tylko przekazanie jej funkcji. Można na przykład napisać **(*a*+*b*).g()**, gdzie **g()** oznacza pewną funkcję składową klasy **Integer**. Dzięki uczyñieniu wartości zwracanej przez operator wartością stałą określa się, że dla tej wartości mogą zostać wywołane jedynie stałe funkcje składowe klasy. Jest to poprawne z punktu widzenia stałych, ponieważ zapobiega umieszczeniu w obiekcie potencjalnie wartościowej informacji, która najprawdopodobniej zostałaby utracona.

Optymalizacja zwracania wartości

Warto zwrócić uwagę na sposób, w jaki tworzone są obiekty zwracane przez wartość. Na przykład w przypadku funkcji **operator+**:

```
return Integer(left.i + right.i);
```

Na pierwszy rzut oka może to przypominać „wywołanie funkcji konstruktora”, ale w rzeczywistości nim nie jest. Jest to składnia obiektu tymczasowego — powyższa instrukcja głosi: „utwórz tymczasowy obiekt klasy Integer i zwróć go”. Z uwagi na to można odnieść wrażenie, że rezultat jest taki sam, jak w przypadku utworzenia lokalnego obiektu, o określonej nazwie, i zwrócenia go. **Jest jednak zupełnie inaczej**. Gdyby, zamiast powyższej instrukcji, dokonać zapisu:

```
Integer tmp(left.i + right.i);
return tmp;
```

to zostałyby przeprowadzone trzy operacje. Po pierwsze, zostałyby utworzony obiekt **tmp**, włącznie z wywołaniem jego konstruktora. Po drugie, konstruktor kopiący skopiowałby wartość obiektu **tmp** do miejsca znajdującego się na zewnątrz funkcji, przeznaczonego na zwracaną przez nią wartość. Po trzecie, na końcu zasięgu zostały wywołany destruktor obiektu **tmp**.

W odróżnieniu od opisanego powyżej schematu „zwracanie wartości tymczasowej” przebiega w zupełnie inny sposób. Gdy kompilator widzi taką konstrukcję, to wie, że obiekt jest tworzony wyłącznie w celu jego zwrotnienia. Kompilator wykorzystuje to, tworząc obiekt *bezpośrednio* w miejscu przeznaczonym na wartość zwracaną przez funkcję. Wymaga to tylko jednego, normalnego wywołania konstruktora (konstruktor kopiący nie jest w ogóle potrzebny). Poza tym nie zachodzi konieczność wywoływania destruktorów, ponieważ w rzeczywistości nigdy nie jest w takim przypadku tworzony obiekt lokalny. Tak więc zastosowanie takiej metody zwracania wartości nie wiąże się z żadnymi kosztami (konieczne jest tylko zrozumienie tego mechanizmu przez programistę), a jest znacznie bardziej efektywne. Działanie takie nosi nazwę *optymalizacji zwracania wartości*.

Nietypowe operatory

W przypadku kilku dodatkowych operatorów składnia związana z przeciążaniem jest nieco odmienna.

Funkcja operatora indeksowego, **operator[]**, musi być funkcją składową i wymaga jednego argumentu. Ponieważ **operator[]** sugeruje, że obiekt, dla którego został wywołany, zachowuje się jak tablica, to operator ten zwraca zazwyczaj referencję, dzięki czemu można w wygodny sposób użyć go po lewej stronie znaku przypisania. Przeciążanie tego operatora jest dość powszechne — odpowiednie przykłady znajdują się w dalszej części książki.

Operatory **new** i **delete** zarządzają dynamicznym przydziałem pamięci i mogą być przeciążane na szereg różnych sposobów. Temat ten jest omawiany w rozdziale 13.

Operator przecinkowy

Operator przecinkowy jest wywoływanym wtedy, gdy znajduje się obok obiektu o typie, dla którego operator ten został zdefiniowany. Jednak „**operator**,” *nie jest* wywoływany w przypadku listy argumentów funkcji, a tylko w stosunku do obiektów, które są zwykle widoczne i oddzielone od siebie przecinkiem. Nie wydaje się, by operator ten mógł mieć jakieś szersze zastosowanie — jest on dostępny dla zachowania spójności języka. Poniższy przykład prezentuje, w jaki sposób wywoywana jest funkcja operatora, zarówno wtedy, gdy przecinek znajduje się *przed* obiektem, jak i *za* nim:

```
//: C12:OverloadingOperatorComma.cpp
#include <iostream>
using namespace std;
```

```
class After {  
public:  
    const After& operator.(const After&) const {  
        cout << "After::operator,()" << endl;  
        return *th1s;  
    }  
};  
  
class Before {};  
  
Before& operator,(int, BeforeS b) {  
    cout << "Before::operator,()" << endl;  
    return b;  
}  
  
int main() {  
    After a, b;  
    a, b; // Wywoływany operator przecinkowy  
  
    Before c;  
    1, c; // Wywoływany operator przecinkowy  
} //:-
```

Funkcja globalna pozwala na umieszczenie przecinka przed obiektem, którego dotyczy operator. Zaprezentowany tu sposób zastosowania operatora jest dość niejasny i kontrowersyjny. Mimo że w charakterze składników bardziej złożonych wyrażeń używa się oddzielonych przecinkami list, to w większości przypadków operator ten ma zbyt mgliste znaczenie, aby go stosować.

Operator ->

Zwykle **operator->** jest używany wówczas, gdy chcemy, by obiekt wyglądał tak, jakby był wskaźnikiem. Z uwagi na to, że obiekty takie odznaczają się większą „inteligencją” niż zwykłe wskaźniki, często nazywa się je *inteligentnymi wskaźnikami*. Są one szczególnie przydatne, gdy zamierzamy „opakować” wskaźnik klasą, by uczynić go bezpieczniejszym. Powszechnie stosuje się je w charakterze *iteratatora*, czyli obiektu poruszającego się w obrębie *kolekcji* lub *kontenera* innych obiektów, udostępniającego za każdym razem pojedynczy obiekt, ale *niezapewniającego* bezpośredniego dostępu do implementacji kontenera (kontenery i iteratory można często spotkać w bibliotekach klas, takich jak standardowa biblioteka języka C++, opisana w drugim tomie książki).

Operator wyłuskania wskaźnika musi być funkcją składową klasy. Obowiązuje w tym przypadku dodatkowe, nietypowe ograniczenie — musi on zwracać obiekt (albo referencję do obiektu), również posiadający operator wyłuskania wskaźnika, albo zwracać wskaźnik, który może zostać użyty do wybrania tego, co wskazuje strzałka operatora. Poniżej znajduje się prosty przykład użycia operatora:

```
//: C12:SmartPointer.cpp  
#include <iostream>  
#include <vector>  
#include "../require.h"  
using namespace std;
```

```
class Obj {
    static int i, j;
public:
    void f() const { cout << i++ << endl; }
    void g() const { cout << j++ << endl; }
};

// Definicje składowych statycznych:
int Obj::i = 47;
int Obj::j = 11;

// Kontener:
class ObjContainer {
    vector<Obj*> a;
public:
    void add(Obj* obj) { a.push_back(obj); }
    friend class SmartPointer;
};

class SmartPointer {
    ObjContainer& oc;
    int index;
public:
    SmartPointer(ObjContainer& objc) : oc(objc) {
        index = 0;
    }
    // Zwracana wartość sygnalizuje koniec listy:
    bool operator++() { // Przedrostek
        if(index >= oc.a.size()) return false;
        if(oc.a[++index] == 0) return false;
        return true;
    }
    bool operator++(int) { // Przyrostek
        return operator++();
    }
    Obj* operator->() const {
        require(oc.a[index] != 0, "SmartPointer::operator->() "
               "zwrocil wartosc zerowa");
        return oc.a[index];
    }
};

int main() {
    const int sz = 10;
    Obj o[sz];
    ObjContainer oc;
    for(int i = 0; i < sz; i++)
        oc.add(&o[i]); // Wypełnianie kontenera
    SmartPointer sp(oc); // Utworzenie iteratatora
    do {
        sp->f(); // Wywołanie operatora wyluskania wskaźnika
        sp->g();
    } while(sp++);
} //:~
```

Klasa **Obj** definiuje obiekt, na którym dokonywane są w tym programie operacje. Funkcje **f()** oraz **g()** wyprowadzają interesujące wartości, wykorzystując statyczne dane składowe. Wskaźniki tych obiektów są zapamiętywane w kontenerze typu **ObjContainer** za pomocą funkcji **add()**. Kontener **ObjContainer** wygląda jak tablica wskaźników, ale, jak widać, nie ma sposobu na usunięcie z niej wskaźników. Klasa **SmartPointer** została jednak zadeklarowana jako klasa zaprzajaźniona, dzięki czemu ma ona dostęp do wnętrza kontenera. Klasa **SmartPointer** przypomina inteligentny wskaźnik — można przesuwać go do przodu, używając operatora **++** (nic nie stoi na przeszędzie, aby zdefiniować również **operator--**), nie wychodzi on poza zakres wskazywanego kontenera i zwraca (za pomocą operacji wyłuskania wskaźnika) wskazywaną wartość. Zwróć uwagę na to, że **SmartPointer** jest specjalnie przystosowany do obsługi kontenera, dla którego został utworzony — w odróżnieniu od zwykłych wskaźników, nie istnieją inteligentne wskaźniki „ogólnego przeznaczenia”. Więcej informacji dotyczących inteligentnych wskaźników można znaleźć w ostatnim rozdziale książki oraz w jej drugim tomie (który można pobrać z witryny www.BruceEckel.com).

Po wypełnieniu kontenera obiektami klasy **Obj** w funkcji **main()** jest tworzony inteligentny wskaźnik **sp** klasy **SmartPointer**. Wywołania inteligentnego wskaźnika występują w wyrażeniach:

```
sp->f(); // Wywołania inteligentnego wskaźnika  
sp->g();
```

Mimo że obiekt **sp** nie posiada w rzeczywistości funkcji składowych **f()** oraz **g()**, operator wyłuskania wskaźnika automatycznie wywołuje te funkcje dla wskaźnika do obiektu klasy **Obj**, zwracanego przez funkcję **SmartPointer::operator->**. Kompilator dokonuje wszelkich niezbędnych sprawdzeń, by upewnić się, że wywołania funkcji działają poprawnie.

Chociaż wewnętrzny mechanizm działania operatora wyłuskania wskaźnika jest bardziej złożony niż innych operatorów, to jego cel jest identyczny — udostępnienie użytkownikom klasy wygodniejszej składni.

Zagnieżdżony iterator

Częściej spotyka się klasę „inteligentnego wskaźnika” lub „iteratora” w postaci zagnieżdżonej wewnętrz obsługiwanej przez nią klasy. Można napisać ponownie po przedni program, zagnieżdżając klasę **SmartPointer** wewnątrz klasy **ObjContainer**, jak w poniższym przykładzie:

```
//: C12:NestedSmartPointer.cpp  
#include <iostream>  
#include <vector>  
#include "../require.h"  
using namespace std;  
  
class Obj {  
    static int i, j;  
public:  
    void f() { cout << i++ << endl; }  
    void g() { cout << j++ << endl; }  
};
```

```
// Definicje składowych statycznych:  
int Obj::i = 47;  
int Obj::j = 11;  
  
// Kontener:  
class ObjContainer {  
    vector<Obj*> a;  
public:  
    void add(Obj* obj) { a.push_back(obj); }  
    class SmartPointer;  
    friend SmartPointer;  
    class SmartPointer {  
        ObjContainer& oc;  
        unsigned int index;  
    public:  
        SmartPointer(ObjContainer& objc) : oc(objc) {  
            index = 0;  
        }  
        // Zwracana wartość sygnalizuje koniec listy:  
        bool operator++() { // Przedrostek  
            if(index >= oc.a.size()) return false;  
            if(oc.a[+index] == 0) return false;  
            return true;  
        }  
        bool operator++(int) { // Przyrostek  
            return operator++(); // Użycie wersji przedrostkowej  
        }  
        Obj* operator->() const {  
            require(oc.a[index] != 0, "SmartPointer::operator->() "  
                "zwrocil wartosc zerowa");  
            return oc.a[index];  
        }  
    };  
    // Funkcja tworząca inteligentny wskaźnik  
    // wskazujący początek obiektu klasy ObjContainer:  
    SmartPointer begin() {  
        return SmartPointer(*this);  
    }  
};  
  
int main() {  
    const int sz = 10;  
    Obj o[sz];  
    ObjContainer oc;  
    for(int i = 0; i < sz; i++)  
        oc.add(&o[i]); // Wypełnianie kontenera  
    ObjContainer::SmartPointer sp = oc.begin();  
    do {  
        sp->f(); // Wywołanie operatora wyłuskania wskaźnika  
        sp->g();  
    } while(++sp);  
} //:~
```

Oprócz tego, że klasa iteratora jest obecnie zagnieżdżona w stosunku do poprzedniej wersji programu, występują tutaj tylko dwie różnice. Pierwszą z nich jest deklaracja klasy iteratora, dzięki której może ona zostać wskazana jako zaprzyjaźniona:

```
class SmartPointer;  
friend SmartPointer;
```

Zanim klasa zostanie określona jako zaprzyjaźniona, kompilator musi wiedzieć o tym, że w ogóle istnieje.

Drugą różnicę stanowi funkcja **begin()**, będąca składową klasy **ObjContainer**. Zwraca ona obiekt klasy **SmartPointer**, wskazujący początek zawartej w kontenerze sekwencji. Mimo że jest to naprawdę tylko kwestią wygody, korzyść wynikająca ze zdefiniowania takiej funkcji polega na tym, że naśladuje ona w pewnym stopniu konwencję stosowaną w standardowej bibliotece C++.

Operator ->*

Operator **->*** jest operatorem dwuargumentowym, działającym podobnie, jak inne operatory dwuargumentowe. Został on udostępniony z myślą o sytuacjach, w których wymagane jest naśladowanie zachowania wbudowanej składni *wskaźnika do składowej*, opisanego w poprzednim rozdziale.

Podobnie jak **operator->**, operator wyłuskania wskaźnika do składowej jest na ogół używany w stosunku do obiektów będących „inteligentnymi wskaźnikami”. Przedstawiony niżej przykład będzie jednak prostszy, aby łatwiej go zrozumieć. Trik związany z definicją funkcji **operator->*** polega na tym, że musi ona zwracać obiekt, dla którego można wywołać funkcję **operator()** z argumentami przeznaczonymi dla wywoływanej funkcji składowej.

Operator wywołania funkcji operator() musi być funkcją składową — i jest on jedyne w swoim rodzaju, ponieważ dopuszcza dowolną liczbę argumentów. Z tego powodu obiekt wygląda tak, jakby był w rzeczywistości funkcją. Mimo że można zdefiniować szereg przeciążonych funkcji **operator()**, posiadających różne argumenty, są one często używane do typów realizujących tylko jedną operację albo przynajmniej posiadających operację o charakterze dominującym. W drugim tomie książki można poznać sposób, w jaki standardowa biblioteka języka C++ wykorzystuje operator wywołania funkcji do tworzenia „obiektów funkcyjnych”.

Aby utworzyć **operator->***, trzeba najpierw utworzyć klasę zawierającą **operator()** o typie obiektu zwracanego przez **operator->***. Klasa ta musi w jakiś sposób zdobyć niezbędne informacje, dzięki którym w czasie wywołania funkcji **operator()** (co odbywa się automatycznie) wskaźnik do składowej zostanie wyłuskany dla tego obiektu. W zamieszczonym poniżej przykładzie konstruktor **FunctionObject** pobiera i zapamiętuje zarówno wskaźnik obiektu, jak i wskaźnik do funkcji składowej, które są wykorzystywane przez **operator()** do wywołania właściwego wskaźnika do funkcji składowej:

```
//: C12:PointerToMemberOperator.cpp  
#include <iostream>  
using namespace std;  
  
class Dog {  
public:  
    int run(int i) const {
```

```
cout << "biega\n";
    return i;
}
int eat(int i) const {
    cout << "je\n";
    return i;
}
int sleep(int i) const {
    cout << "spi\n";
    return i;
}
typedef int (Dog::*PMF)(int) const;
// operator->* musi zwrócić obiekt
// posiadający operator():
class FunctionObject {
    Dog* ptr;
    PMF pmem;
public:
    // Zapamiętanie wskaźnika obiektu i wskaźnika składowej
    FunctionObject(Dog* wp, PMF pmf)
        : ptr(wp), pmem(pmf) {
            cout << "Konstruktor FunctionObject\n";
    }
    // Wywołanie, wykorzystujące wskaźnik obiektu
    // i wskaźnik składowej
    int operator()(int i) const {
        cout << "FunctionObject::operator()\n";
        return (ptr->*pmem)(i); // Wywołanie
    }
};
FunctionObject operator->*(PMF pmf) {
    cout << "operator->*" << endl;
    return FunctionObject(this, pmf);
};

int main() {
    Dog w;
    Dog::PMF pmf = &Dog::run;
    cout << (w->*pmf)(1) << endl;
    pmf = &Dog::sleep;
    cout << (w->*pmf)(2) << endl;
    pmf = &Dog::eat;
    cout << (w->*pmf)(3) << endl;
} //:~
```

Klasa **Dog** zawiera trzy funkcje składowe, z których każda pobiera jako argument i zwraca liczbę całkowitą. **PMF** jest typem zdefiniowanym za pomocą deklaracji **typedef** w celu uproszczenia definiowania wskaźników do funkcji składowych klasy **Dog**.

Obiekt klasy **FunctionObject** jest tworzony i zwracany przez **operator->***. Zwróć uwagę na to, że **operator->*** wie, dla jakiego obiektu został wywołany wskaźnik do funkcji składowej (**this**), a także jaki to jest wskaźnik. Przekazuje więc te wartości konstruktorowi klasy **FunctionObject**, który je zapamiętuje. Gdy wywoływany jest **operator->***, kompilator reaguje nagłym zwrotem, wywołując **operator()** dla wartości

zwracanej przez **operator->*** i przekazując mu wszystkie argumenty, które zostały podane operatorowi **->***. Funkcja **FunctionObject::operator()** pobiera te argumenty, a następnie wyłuskuje „prawdziwy” wskaźnik do składowej, wykorzystując zapamiętany wskaźnik obiektu oraz wskaźnik do składowej.

Zwróć uwagę na to, że podobnie jak w przypadku operatora **->** wchodzimy niejako „w środek” wywołania funkcji **operator->***. Pozwala to na dokonanie jakichś dodatkowych operacji, gdyby zaszła taka potrzeba.

Zaimplementowany w programie mechanizm operatora **->*** działa tylko w przypadku funkcji składowych, które pobierają argument będący liczbą całkowitą i zwracającą liczbę całkowitą. Stanowi to pewne ograniczenie, ale próba utworzenia przeciążonych mechanizmów, obejmujących każdą dopuszczalną możliwość, wydaje się niemal niewykonalna. Na szczęście, dostępne w języku C++ szablony (opisane w ostatnim rozdziale książki oraz w jej drugim tomie) zostały zaprojektowane właśnie z myślą o takich problemach.

Operatory, których nie można przeciążać

Niektóre operatory, znajdujące się w **zbiorze** wszystkich dostępnych operatorów, nie mogą być przeciążane. Ograniczenia te wynikają na ogół ze względów bezpieczeństwa. Gdyby przeciążanie operatorów było dozwolone, mogłoby to spowodować naruszenie na szwank, a nawet naruszenie mechanizmów bezpieczeństwa, wprowadziło by utrudnienia albo byłoby sprzeczne z przyjętą praktyką.

- 4 Operator wyboru składowej **(.)**. Obecnie kropka posiada znaczenie zdefiniowane dla wszystkich składowych klasy. Gdyby pozwolić najętym przeciążanie, to nie można by było odwoływać się do składowych klasy w zwykły sposób — należałoby używać do tego celu wskaźnika oraz operatora **->**.
- 4 Operator wyłuskania wskaźnika do składowej **(*)** — z tych samych powodów, co w przypadku operatora wyboru składowej.
- ◆ Nie istnieje operator potęgowania. Najbardziej popularną propozycją był dla niego pochodzący z Fortranu **operator****, ale powodował on problemy związane z analizą składniową. Ponieważ w języku C nie ma operatora potęgowania, więc również jego obecność w języku C++ nie wydaje się konieczna — tym bardziej, że zawsze można zastąpić go wywołaniem funkcji. Operator potęgowania umożliwiałby wygodny zapis, ale nie dodawałby do języka takich nowych cech, dla których warto by wprowadzać dodatkowe utrudnienia do procesu komplikacji.
- ◆ Nie ma operatorów definiowanych przez użytkownika. Oznacza to, że nie można samodzielnie tworzyć nowych operatorów — takich, których nie mają jeszcze w języku. Wynika to częściowo z problemów dotyczących sposobu, w jaki miałyby zostać zdefiniowane **priorytety**, a częściowo z braku uzasadnienia dla nieuchronnie z tym związanych kłopotów.
- ◆ Nie można zmieniać reguł dotyczących priorytetów operatorów; sąjuż i tak wystarczająco trudne do zapamiętania.

Operatory niebędące składowymi

W niektórych z poprzednich przykładów operatory mogły być zarówno funkcjami składowymi klas, jak i zwykłymi funkcjami; wydawało się, że nie sprawia to większej różnicy. Można zatem zadać pytanie: „Który ze sposobów wybrać?”. Na ogół, jeżeli nie ma to znaczenia, powinny być one funkcjami składowymi — dla podkreślenia związku pomiędzy operatorem i jego klasą. Sprawdza się to bardzo dobrze, dopóki argument znajdujący się po lewej stronie operatora jest obiektem danej klasy.

Czasami zachodzi jednak potrzeba, by argument znajdujący się po lewej stronie operatora był obiektem jakiejś innej klasy. Typowym tego przykładem jest przeciążanie operatorów << i >> dla operacji związanych ze strumieniami wejścia-wyjścia. Ponieważ strumienie wejścia-wyjścia są jedną z podstawowych bibliotek języka C++, to prawdopodobnie zamierzasz przeciążyć te operatory w większości z tworzonych przez siebie klas, a więc proces ten wart jest zapamiętania:

```
//: C12:IostreamOperatorOverloading.cpp
// Przykłady przeciążonych operatorów, niebędących
// funkcjami składowymi
#include "../require.h"
#include <iostream>
#include <sstream> // "Strumienie łańcuchowe"
#include <cstring>
using namespace std;

class IntArray {
    enum { sz = 5 };
    int i[sz];
public:
    IntArray() { memset(i, 0, sz * sizeof(*i)); }
    int& operator[](int x) {
        require(x >= 0 && x < sz,
            "IntArray::operator[] poza zakresem");
        return i[x];
    }
    friend ostream&
    operator<<(ostream& os, const IntArray& ia) {
        friend istream&
        operator>>(istream& is, IntArray& ia);
    };
    ostream&
    operator<<(ostream& os, const IntArray& ia) {
        for(int j = 0; j < ia.sz; j++) {
            os << ia.i[j];
            if(j != ia.sz - 1)
                os << " ";
        }
        os << endl;
        return os;
    }
    istream& operator>>(istream& is, IntArray& ia){
        for(int j = 0; j < ia.sz; j++)
            is << ia.i[j];
        return is;
    }
}
```

```

    ls >> ia.i[j];
    return is;
}

int main() {
    stringstream input("47 34 56 92 103");
    IntArray I;
    input >> I;
    I[4] = -1; // Wykorzystanie przeciążonego operatora []
    cout << I;
} III-

```

Klasa ta zawiera również przeciążony **operator[]**, zwracający referencję do odpowiedniej wartości zawartej w tablicy. Ponieważ zwracana jest referencja, wyrażenie:

I[4] = -1;

nie tylko sprawia wrażenie bardziej „cywilizowanego” niż w przypadku użycia wskaźników, ale wywołuje również pożądany skutek.

Ważne jest, by przeciążone operatory przesunięć pobierały i zwracały wartości *przez referencje*, dzięki czemu operacje wywierają wpływ na obiekty zewnętrzne. W definicjach funkcji wyrażenia takie, jak:

os << ia.i[j];

powodują wywołanie istniejących funkcji przeciążonych operatorów (tych, które zdefiniowano w pliku nagłówkowym **<iostream>**). W powyższym przypadku zostanie wywołana funkcja **ostream& operator<<(ostream&, int)**, ponieważ **ia.i[j]** zwraca wartość będącą liczbą całkowitą.

Za każdym razem, gdy dokonywane są operacje na klasach **istream lub ostream**, ich wartości są zwracane. Dzięki temu można je wykorzystać w bardziej złożonych operacjach.

W funkcji **main()** został wykorzystany nowy typ strumienia wejścia-wyjścia — klasa **stringstream** (zadeklarowana w pliku nagłówkowym **<sstream>**). Pobiera ona łańcuch (który, jak widać powyżej, może zostać utworzony na podstawie tablicy znakowej) i przekształca go w strumień wejścia-wyjścia. W powyższym przykładzie oznacza to, że operatory przesunięć mogą być testowane bez otwierania plików ani wpisywania danych w wierszu poleceń.

Przedstawiona w powyższym przykładzie postać operatorów **<<** oraz **>>** jest standartowa. Jeżeli zamierzasz utworzyć takie operatory dla swojej klasy, skopuj sygnatury powyższych funkcji i zwracane przez nie typy wartości, a następnie przyjmij za wzór ciała tych funkcji.

Podstawowe wskazówki

Murray¹ zaproponował, podane poniżej, wskazówki, umożliwiające podjęcie decyzji o wyborze pomiędzy funkcjami składowymi klasy i funkcjami niebędącymi funkcjami składowymi:

¹ Rob Murray, C++ Strategies & Tactics, Addison-Wesley, 1993 r., s. 47.

Operator	Zalecany sposób użycia
Wszystkie operatory jednoargumentowe = () [] -> -*	funkcje składowe <i>muszą</i> być funkcjami składowymi
+= -= = *= ^= &= = %= >>= <<=	funkcje składowe
Wszystkie pozostałe operatory dwuargumentowe	funkcje niebędące funkcjami składowymi

Przeciążanie operacji przypisania

Dla początkujących programistów języka C++ stałym źródłem utratyń jest instrukcja przypisania. Nic dziwnego, ponieważ znak = jest fundamentalną operacją w programowaniu, sięgającą kopiowania rejestrów na poziomie maszynowym. Ponadto gdy użyty zostanie znak =, to czasami wywoływany jest również konstruktor kopiący (opisany w rozdziale 11.):

```
MojTyp b;  
MojTyp a = b;  
a = b;
```

W drugim wierszu *definiowany* jest obiekt a. Tworzony jest obiekt, którego wcześniej nie było. Ponieważ **wiemy już, jak skrupulatny** jest kompilator języka C++ w sprawie inicjalizacji obiektów, nie ulega wątpliwości, że *zawsze* w miejscu, w którym tworzyony jest obiekt, musi być wywoływany konstruktor. Ale który? Obiekt a jest tworzyony na podstawie istniejącego już obiektu klasy **MojTyp** (obiektu b, znajdującego się po prawej stronie znaku równości), więc nie ma w tym przypadku wyboru — konstruktorem tym musi być konstruktor kopiący. Mimo użycia znaku równości wywoływany jest zatem konstruktor kopiący.

Sprawa wygląda inaczej w przypadku trzeciego wiersza. Po lewej stronie znaku równości znajduje się wcześniej zainicjowany obiekt. Oczywiście, w stosunku do już utworzonego obiektu nie jest wywoływany konstruktor kopiący. W tym przypadku w stosunku do obiektu a jest wywoływana funkcja **MojTyp::operator=**, pobierająca w charakterze argumentu to, co znajduje się po prawej stronie znaku równości (można zdefiniować kilka funkcji **operator=**, przyjmujących różne typy argumentów znajdujących się po prawej stronie przypisania).

Opisane powyżej zachowanie nie ogranicza się do konstruktora kopiącego. Za każdym razem, gdy obiekt jest inicjalizowany za pomocą znaku =, zamiast konstruktora, wywołanego normalnie w postaci funkcji, kompilator poszukuje konstruktora, który **przyjmuje jako argument** to, co znajduje się po prawej stronie znaku równości:

```
//: C12:CopyingVsInitialization.cpp
class F1 {
public:
    F1() {}
};

class F2 {
public:
```

```

Fee(int) {}
Fee(const Fi&) {}
};

int main() {
    Fee fee = 1; // Fee(int)
    Fi fi;
    Fee fum = fi; // Fee(Fi)
} //:-)

```

Należy pamiętać o tym rozróżnieniu, dotyczącym znaku `=`: jeżeli obiekt nie został jeszcze **utworzony**, to konieczna jest jego inicjalizacja; w przeciwnym przypadku wykorzystywany jest **operator=**, stanowiący przypisanie.

Jeszcze lepiej jest unikać pisania kodu, wykorzystującego znak `=` do **inicjalizacji** i zawsze używać zamiast niego jawnego wywołania konstruktora. W ten sposób dwa **konstruktory**, wywoływanie w powyższym programie za pomocą znaku równości, uzykają postać:

```

Fee fee(l);
Fee fum(fi);

```

Dzięki temu uniknie się wprowadzania w błąd osób czytających kod programu.

Zachowanie się operatora =

W plikach **Integer.h** oraz **Byte.h** funkcja **operator=** może być wyłącznie funkcją składową. Jest ona bezpośrednio związana z obiektem znajdującym się po lewej stronie znaku „`=`”. Gdyby istniała możliwość zdefiniowania globalnej funkcji **operator=**, można by zdefiniować ponownie wbudowane w język znaczenie tego operatora:

```
int operator=(int, MojTyp); // Globalna definicja - nie jest dozwolona!
```

Kompilator omija tę kwestię, wymuszając definicję funkcji **operator=** jako funkcji składowej.

Podczas tworzenia operatora `=` konieczne jest skopiowanie wszystkich niezbędnych informacji z obiektu znajdującego się po prawej stronie do bieżącego obiektu (czyli tego, dla którego została wywołana funkcja **operator=()**) w celu wykonania tego, co uważamy w naszej klasie za „przypisanie”. W przypadku prostych obiektów to oczywiste:

```

//: C12:SimpleAssignment.cpp
// Prosty operator=()
#include <iostream>
using namespace std;

class Value {
    int a, b;
    float c;
public:
    Value(int aa = 0, int bb = 0, float cc = 0.0)
        : a(aa), b(bb), c(cc) {}
    Value& operator=(const Value& rv) {

```

```

    a = rv.a;
    b = rv.b;
    c = rv.c;
    return *this;
}
friend ostream&
operator<<(ostream& os, const Value& rv) {
    return os << "a - " << rv.a << ", b - "
        << rv.b << ", c = " << rv.c;
}
};

int main() {
    Value a, b(1, 2, 3.3);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    a = b;
    cout << "a po przypisaniu: " << a << endl;
} //:-

```

W powyższym **przykładzie** obiekt znajdujący się po lewej stronie znaku = kopiuje wszystkie elementy obiektu znajdującego się po prawej stronie znaku, a następnie zwraca referencję do samego siebie, co pozwala na tworzenie bardziej złożonych wyrażeń.

Przykład ten zawiera często spotykany błąd. Gdy dokonuje się operacji przypisania w stosunku do dwóch obiektów tego samego typu, należy zawsze najpierw sprawdzić, czy nie przypisuje się obiektu do samego siebie. W pewnych przypadkach, takich jak przedstawiony powyżej, wykonanie takiej operacji jest nieszkodliwe, ale w razie dokonania zmian w implementacji klasy może stać się istotne. Jeżeli nie będzie się rutynowo sprawdzać, czy nie następuje przypisanie obiektu do samego siebie, to można w końcu o tym zapomnieć, wprowadzając do programu trudne do wykrycia błędy.

Wskaźniki zawarte w klasach

Co się dzieje, gdy obiekty nie są tak proste? Na przykład gdy zawierają one wskaźniki do innych obiektów? Zwykłe kopiowanie doprowadzi do tego, że dwa obiekty będą wskazywały ten sam obszar pamięci. W takich sytuacjach trzeba radzić sobie samemu.

Stosowane są dwa podejścia do tego problemu. Prostsza technika polega na kopiowaniu wszystkiego, co jest wskazywane przez wskaźniki — podczas przypisania lub wywołania konstruktora kopującego. To dość proste:

```

//: C12:CopyingWithPointers.cpp
// Rozwiążanie problemu utożsamiania wskaźników
// za pomocą kopowania wskazywanych przez nie
// obszarów podczas przypisania i wywołania
// konstruktora kopującego
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
    string nm;

```

```
public:  
    Dog(const string& name) : nm(name) {  
        cout << "Tworzenie psa: " << *this << endl;  
    }  
    // Utworzony przez kompilator konstruktor kopiujący  
    // i operator= są prawidłowe  
    // Utworzenie obiektu klasy Dog na podstawie wskaźnika:  
    Dog(const Dog* dp, const string& msg)  
        : nm(dp->nm + msg) {  
        cout << "Pies " << *this << " skopiowany z "  
            << *dp << endl;  
    }  
    ~Dog() {  
        cout << "Usunięty pies: " << *this << endl;  
    }  
    void rename(const string& newName) {  
        nm = newName;  
        cout << "Pies zmienił imię na: " << *this << endl;  
    }  
    friend ostream&  
operator<<(ostream& os, const Dog& d) {  
    return os << "[" << d.nm << "]";  
}  
};  
  
class DogHouse {  
    Dog* p;  
    string houseName;  
public:  
    DogHouse(Dog* dog, const string& house)  
        : p(dog), houseName(house) {}  
    DogHouse(const DogHouse& dh)  
        : p(new Dog(dh.p, " skopiowany konstruktorem")),  
        houseName(dh.houseName)  
            + " skopiowany konstruktorem" {}  
    DogHouse& operator=(const DogHouse& dh) {  
        // Sprawdzanie przypisania do samego siebie:  
        if(&dh != this) {  
            p = new Dog(dh.p, " przypisany");  
            houseName = dh.houseName + " przypisany";  
        }  
        return *this;  
    }  
    void renameHouse(const string& newName) {  
        houseName = newName;  
    }  
    Dog* getDog() const { return p; }  
    ~DogHouse() { delete p; }  
    friend ostream&  
operator<<(ostream& os, const DogHouse& dh) {  
    return os << "[" << dh.houseName  
        << "] zawiera " << *dh.p;  
}  
};  
  
int main() {  
    DogHouse fafika(new Dog("Fafik"), "DomekFafika");
```

```

cout << fafika << endl;
DogHouse fafika2 = fafika; // Użycie konstruktora kopiącego
cout << fafika2 << endl;
fafika2.getDog()->rename("Burek");
fafika2.renameHouse("DomekBurka");
cout << fafika2 << endl;
fafika - fafika2; // Przypisanie
cout << fafika << endl ;
fafika.getDog()->rename("Reks");
fafika2.renameHouse("DomekReksa");
} //:-

```

Klasa Dog (pies) jest prostą **klasą, zawierającą jedyniełańcuch**, przechowujący imię psa. Jednakże na ogół wiadomo, gdy coś dzieje się z obiektem tej klasy, ponieważ zarówno konstruktor, jak i destruktor wyświetlają informację o tym, że zostały wywołane. Zwróć uwagę na to, że drugi konstruktor jest nieco podobny do konstruktora kopiącego, z wyjątkiem tego, że pobiera wskaźnik do obiektu klasy Dog, a nie referencję. Ponadto posiada drugi argument, będący komunikatem dołączanym do imienia psa, którego wskaźnik stanowi pierwszy argument. Pomaga to w prześledzeniu działania programu.

A zatem ilekroć funkcja składowa drukuje informację, nie wykonuje tego bezpośrednio, tylko przesyła wartość ***this** do strumienia **cout**. To z kolei powoduje wywołanie funkcji **operator<<**, przekazującej tekst do strumienia **ostream**. Warto robić to w taki sposób, ponieważ jeśli będzie potrzebna modyfikacja formatu wyświetlania informacji o obiekcie klasy Dog (w przykładzie zostały dodane znaki „[” i „]”), to trzeba zmienić go tylko w jednym miejscu.

Klasa **DogHouse** (psia buda²) zawiera składową typu **Dog*** oraz prezentuje cztery funkcje, które zawsze trzeba zdefiniować, gdy klasa zawiera wskaźniki: wszystkie niezbędne zwyczajne **konstruktory**, konstruktor kopiący, **operator=** (można go albo zdefiniować, albo zabronić jego użycia) oraz destruktor. Operator = rutynowo sprawdza, czy następuje przypisanie obiektu do samego siebie, mimo że w tym przypadku nie jest to absolutnie konieczne. Dzięki temu praktycznie wyklucza się możliwość zapomnienia o sprawdzeniu przypisania do samego siebie po wprowadzeniu zmian w klasie, które spowodowałyby taką konieczność.

Czyszczanie odwołań

W powyższym przykładzie zarówno konstruktor kopiący, jak i **operator=** tworzyły nową kopię tego, co wskazywał wskaźnik, a destruktor ją usuwał. Jeżeli jednak obiekty wymagają dużej ilości pamięci lub ich inicjalizacja wiąże się ze sporym narzutem, to być może warto uniknąć kopiowania. Typowe podejście do tego problemu nazywane jest **zliczaniem odwołań** (ang. *reference counting*). Dany obiekt obdarza się pewną „**inteligencją**”, dzięki której wie on, ile obiektów go wskazuje. W takim przypadku wywołanie konstruktora kopowania oraz operacja przypisania oznaczają dołączenie jeszcze jednego wskaźnika do istniejącego obiektu i **inkrementację** licznika odwołań. Destrukcja oznacza natomiast **dekrementację** licznika odwołań i zniszczenie obiektu w przypadku, gdy licznik odwołań osiągnie wartość zerową.

²wana w wyświetlanych w programie tekstach, wyłącznie ze względów gramatycznych, „**imiem**” — przyp. tłum.

Co się jednak dzieje, gdy chcemy zapisać coś w obiekcie (na przykład w obiekcie klasy Dog z poprzedniego przykładu)? Obiekt klasy Dog może być używany przez większą liczbę obiektów, więc jego modyfikacja mogłaby spowodować zarówno zmianę własnego obiektu, jak i obiektu należącego do kogoś innego, co nie byłoby pożądane. Do rozwiązania tego problemu „utożsamiania” wykorzystywana jest dodatkowa technika, nazywana *kopiowaniem przy zapisie* (ang. *copy-on-write*). Przed zapisaniem informacji w obszarze pamięci należy się upewnić, że nikt inny go nie używa. Jeżeli licznik odwołań ma wartość większą niż jeden, to przed zapisaniem informacji w tym obszarze należy wykonać jego „osobistą” kopię, dzięki czemu dane należące do innych obiektów pozostaną nienaruszone. Poniżej znajduje się prosty przykład zliczania odwołań i kopiowania przy zapisie:

```
//: C12:ReferenceCounting.cpp
// Zliczanie odwołań, kopiowanie przy zapisie
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
    string nm;
    int refcount;
    Dog(const string& name)
        : nm(name), refcount(1) {
        cout << "Tworzenie psa: " << *this << endl;
    }
    // Zablokowanie operacji przypisania:
    Dog& operator<<(const Dog& rv);
public:
    // Obiekty klasy Dog mogą zostać utworzone tylko na stercie:
    static Dog* make(const string& name) {
        return new Dog(name);
    }
    Dog(const Dog& d)
        : nm(d.nm + " kopia"), refcount(1) {
        cout << "Konstruktor kopiący psa: "
            << *this << endl;
    }
    ~Dog() {
        cout << "Usuwanie psa: " << *this << endl;
    }
    void attach() {
        ++refcount;
        cout << "Pies dołączony: " << *this << endl;
    }
    void detach() {
        require(refcount != 0);
        cout << "Odłączanie psa: " << *this << endl;
        // Niszczanie obiektu w przypadku, gdy nikt go nie używa:
        if(--refcount == 0) delete this;
    }
    // Warunkowe kopiowanie obiektu klasy Dog.
    // Należy je wywołać przed modyfikacją obiektu
    // klasy Dog'i przypisać uzyskany wskaźnik do
    // własnego wskaźnika obiektu klasy Dog.
```

```
Dog* unalias() {
    cout << "Usuwanie połączenia psa: " << *this << endl;
    // Jeżeli nie ma odwołań, to kopiowanie nie jest realizowane:
    if (refcount == 1) return this;
    --refcount;
    // Użycie konstruktora kopującego w celu utworzenia kopii:
    return new Dog(*this);
}
void rename(const string& newName) {
    nm = newName;
    cout << "Zmiana imienia psa na: " << *this << endl;
}
friend ostream&
operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "]". rc =
        << d.refcount;
}
};

class DogHouse {
    Dog* p;
    string houseName;
public:
    DogHouse(Dog* dog, const string& house)
        : p(dog), houseName(house) {
        cout << "Utworzono domek: " << *this << endl;
    }
    DogHouse(const DogHouse& dh)
        : p(dh.p),
        houseName("skopiowany konstruktorem " +
            dh.houseName) {
        p->attach();
        cout << "Konstruktor kopowania DogHouse: "
            << *this << endl;
    }
    DogHouse& operator=(const DogHouse& dh) {
        // Sprawdzanie przypisania do samego siebie:
        if (&dh == this) {
            houseName = dh.houseName + " przypisany";
            // Usuniecie poprzedniej wartości:
            p->detach();
            p = dh.p; // Podobnie jak w konstruktorze kopującym
            p->attach();
        }
        cout << "operator= klasy DogHouse: "
            << *this << endl;
        return *this;
    }
    // Zmniejszenie licznika odwołań i warunkowe zniszczenie
    ~DogHouse() {
        cout << "Destruktor klasy DogHouse: "
            << *this << endl;
        p->detach();
    }
    void renameHouse(const string& newName) {
        houseName = newName;
    }
    void unalias() { p->unalias(); }
```

```

// Kopiowanie przy zapisie. Za każdym razem, gdy
// modyfikowany jest wskazywany obiekt, trzeba go
// najpierw "odłączyć":
void renameDog(const string& newName) {
    unalias();
    p->rename(newName);
}
// ...także w przypadku udostępniania obiektu:
Dog* getOog() {
    unalias();
    return p;
}
friend ostream&
operator<<(ostream& os, const DogHouse& dh) {
    return os << "[" << dh.houseName
        << "] zawiera " << *dh.p;
}
};

int main() {
    DogHouse
        fafika(Dog::make("Fafik"), "DomekFafika"),
        burka(Dog::make("Burek"), "DomekBurka");
    cout << "Przed uruchomieniem konstruktora kopiącego" << endl;
    DogHouse reksa(fafika);
    cout << "Po utworzeniu reksa za pomocą konstruktora kopiącego" << endl;
    cout << "fafika:" << fafika << endl;
    cout << "burka:" << burka << endl;
    cout << "reksa:" << reksa << endl;
    cout << "Przed przypisaniem burka » fafika" << endl;
    burka = fafika;
    cout << "Po przypisaniu burka - fafika" << endl;
    cout << "burka:" << burka << endl;
    cout << "Rozpoczęcie kopowania do samego siebie" << endl;
    reksa = reksa;
    cout << "Po zakończeniu kopowania do samego siebie" << endl;
    cout << "reksa:" << reksa << endl;
    // Zaznacz poniższe wiersze jako komentarze:
    cout << "Przed wywołaniem funkcji rename(\"Reks\")" << endl;
    reksa.getDog()->rename("Reks");
    cout << "Po wywołaniu funkcji rename(\"Reks\")" << endl;
} //:-

```

Klasa **DogHouse** (psia buda) zawiera wskaźnik do obiektu klasy **Dog** (pies). Klasa **Dog** posiada licznik odwołań i funkcje, umożliwiające zarządzanie nim oraz odczytujące jego wartość. Zawiera również konstruktor kopujący, pozwalający na utworzenie nowego obiektu klasy **Dog** na podstawie obiektu, który już istnieje.

Funkcja **attach()** inkrementuje wartość licznika odwołań do obiektu klasy **Dog**, informując, że używa go jeszcze jeden obiekt. Funkcja **detach()** dekrementuje licznik odwołań. Gdy licznik ten osiągnie wartość zerową, oznacza to, że nikt nie używa już obiektu, więc funkcja składowa niszczy swój obiekt za pomocą instrukcji **delete this**.

Zanim zostaną dokonane jakiekolwiek modyfikacje obiektu (takie jak zmiana przechowywanego w nim imienia **psa**), trzeba się upewnić, że nie zmienia się jednocześnie obiektu, używanego **przez jakiś** inny obiekt. W tym celu należy wywołać funkcję

DogHouse::unalias(), która wywołuje z kolejnej funkcję **Dog::unalias()**. Druga z wymienionych funkcji zwraca istniejący wskaźnik obiektu Dog w przypadku, gdy licznik odwołań ma wartość jeden (co oznacza, że tego obiektu nie wskazuje żaden inny obiekt), ale gdy licznik ma wartość większą niż jeden, tworzy kopię tego obiektu.

Konstruktor kopiący klasy **DogHouse**, zamiast tworzyć własny obszar pamięci, przypisuje swojemu wskaźnikowi obiektu **Dog** wskaźnik obiektu **Dog**, który pochodzi z obiektu będącego argumentem konstruktora. Następnie, z uwagi na to, że jest już następnym obiektem, wykorzystującym ten sam blok pamięci, inkrementuje licznik odwołań, wywołując funkcję **Dog::attach()**.

Funkcja **operator=** ma do czynienia z obiektem znajdującym się po lewej stronie znaku =, który już został wcześniej utworzony. Musi więc najpierw „wyczyścić go”, wywołując w stosunku do tego obiektu funkcję **detach()**, która spowoduje usunięcie poprzedniego obiektu klasy **Dog**, jeżeli żaden inny obiekt już go nie używa. Następnie **operator=** wykonuje te same operacje, co konstruktor kopiący. Zwróć uwagę na to, że na samym początku dokonuje on sprawdzenia, czy obiekt nie jest kopiowany do samego siebie.

Destruktor wywołuje funkcję **detach()**, usuwającą warunkowo obiekt klasy **Dog**.

Aby zaimplementować kopiowanie przy zapisie, trzeba nadzorować wszystkie operacje zapisujące informacje w przydzielonym bloku pamięci. Na przykład funkcja składowa **renameDog()** pozwala na zmianę informacji zawartych w obszarze pamięci obiektu. Jednak najpierw wykorzystuje ona funkcję **unalias()**, zapobiegając w ten sposób modyfikacjom obiektu klasy **Dog**, wskazywanego przez więcej niż jeden obiekt klasy **DogHouse**. Również w przypadku konieczności zwrócenia przez klasę **DogHouse** wskaźnika do obiektu klasy Dog, w stosunku do tego wskaźnika wywoływana jest najpierw funkcja **unalias()**.

W funkcji **main()** testowane są rozmaite funkcje, które, by realizować zliczanie odwołań, muszą działać prawidłowo: konstruktor, konstruktor kopiący, **operator=** oraz destruktor. Dzięki wywołaniu funkcji **renameDog()** testowane jest również kopiowanie przy zapisie.

Poniżej zamieszczono wyniki działania programu (po ich niewielkim przeformatowaniu):

```
Tworzenie psa: [Fafik], rc = 1
Utworzono domek: [DomekFafika]
    zawiera [Fafik]. rc = 1
Tworzenie psa: [Burek], rc = 1
Utworzono domek: [DomekBurka]
    zawiera [Burek]. rc = 1
Przed uruchomieniem konstruktora kopiącego
Pies dodaczony: [Fafik], rc = 2
Konstruktor kopiowania DogHouse:
    [skopiowany konstruktorem DomekFafika]
        zawiera [Fafik], rc = 2
Po utworzeniu rekса za pomocą konstruktora kopiącego
fafika:[DomekFafika] zawiera [Fafik]. rc = 2
burka:[DomekBurka] zawiera [Burek], rc = 1
rekса:[skopiowany konstruktorem DomekFafika]
    zawiera [Fafik], rc = 2
```

Przed przypisaniem burka - fafika
 Odłączanie psa: [Burek], rc - 1
 Usuwanie psa: [Burek]. rc = 0
 Pies dolaczony: [Fafik]. rc - 3
 operator<< klasy DogHouse: [DomekFafika przypisany]
 zawiera [Fafik]. rc - 3
 Po przypisaniu burka - fafika
burka:[DomekFafika przypisany] zawiera [Fafik]. rc » 3
 Rozpoczęcie kopiowania do samego siebie
operator= klasy DogHouse: [skopiowany konstruktorem DomekFafika]
 zawiera [Fafik]. rc - 3
 Po zakończeniu kopiowania do samego siebie
reksa:[skopiowany konstruktorem DomekFafika]
 zawiera [Fafik], rc - 3
 Przed wywołaniem funkcji **rename("Reks")**
 Usuwanie połączenia psa: [Fafik], rc - 3
 Konstruktor kopiący psa: [Fafik kopia], rc - 1
 Zmiana imienia psa na: [Reks]. rc = 1
 Po wywołaniu funkcji **rename("Reks")**
 Destruktor klasy DogHouse:
 [skopiowany konstruktorem DomekFafika]
 zawiera [Reks], rc - 1
Odłączanie psa: [Reks], rc - 1
 Usuwanie psa: [Reks], rc - 0
 Destruktor klasy DogHouse:
 [DomekFafika przypisany] zawiera [Fafik], rc = 2
Odłączanie psa: [Fafik], rc = 2
 Destruktor klasy DogHouse:
 [DomekFafika] zawiera [Fafik], rc = 1
Odłączanie psa: [Fafik], rc - 1
 Usuwanie psa: [Fafik]. rc - 0

Studiując uważnie wyniki działania programu, śledząc kod źródłowy i eksperymentując z programem, pogłębisz swoją wiedzę na temat **stosowanych** w programie technik.

Automatyczne tworzenie operatora =

Z uwagi na to, że większość programistów spodziewa się, że operacja przypisywania jednego obiektu drugiemu obiektemu *tego samego typu* jest możliwa do wykonania, kompilator automatycznie tworzy operator **typ::operator=(typ)**, jeżeli nie zostanie on zdefiniowany. Działanie tego operatora naśladuje działanie automatycznie utworzonego konstruktora kopiącego — **jeżeli** klasa zawiera obiekty (albo jest klasą pochodną innej klasy), to dla każdego z tych obiektów jest wywoływany rekurencyjnie **operator=**. Operacja ta nosi nazwę *przypisania za pośrednictwem elementów składowych* (ang. **memberwise assignment**). Ilustruje to poniższy przykład:

```
//: C12:AutomaticOperatorEquals.cpp
#include <iostream>
using namespace std;

class Cargo {
public:
    Cargo& operator=(const Cargo&) {
        cout << "wewnątrz Cargo::operator=()" << endl;
        return *this;
    }
}
```

```
};

class Truck {
    Cargo b;
};

int main() {
    Truck a, b;
    a - b; // Drukuj: "wewnętrz Cargo::operator=()"
} //:-~
```

Automatycznie wygenerowany **operator=** klasy Truck wywołuje funkcję **Cargo::operator=**.

Zwykle nie należy pozwalać na to kompilatorowi. W przypadku niebanalnych klas (zwłaszcza gdy zawierają one **wskaźniki!**) należy zawsze utworzyć **operator=** w jawnym sposobie. Jeżeli naprawdę nie chcemy, aby ktokolwiek dokonywał przypisywania do obiektów klas, należy zadeklarować **operator=** jako funkcję prywatną (nie ma potrzeby jej definiowania, pod warunkiem, że nie będzie ona używana w obrębie klasy).

Automatyczna konwersja typów

Gdy w językach C i C++ kompilator widzi wyrażenie lub wywołanie funkcji wykorzystujące typ niebędący dokładnie typem, który jest w danej chwili potrzebny, to często może przeprowadzić automatyczną konwersję typów — z aktualnego do aktualnie wymaganego. W języku C++ ten sam efekt można uzyskać w przypadku typów definiowanych przez użytkownika, definiując funkcje odpowiedzialne za automatyczną konwersję typów. Funkcje te istnieją w dwóch postaciach — jako szczególny rodzaj konstruktora oraz przeciążony operator.

Konwersja za pomocą konstruktora

Jeżeli zdefiniujemy konstruktor, pobierający jako jedyny argument obiekt (lub referencję) innego typu, to taki konstruktor umożliwia kompilatorowi dokonanie automatycznej konwersji typów. Świadczy o tym poniższy program:

```
//: C12:AutomaticTypeConversion.cpp
// Konstruktor umożliwiający konwersje typów
class One {
public:
    One() {}
};

class Two {
public:
    Two(const One&) {}
};

void f(Two) {}
```

```
int main() {
    One one;
    f(one); // Wymagany obiekt klasy Two, a użyty klasa One
} //
```

Gdy kompilator widzi funkcję **f()**, wywoływaną z obiektem klasy **One**, zagląda do deklaracji funkcji **f()** i wykrywa, że wymaga ona argumentu będącego obiektem klasy **Two**. Następnie poszukuje sposobu przekształcenia obiektu klasy **One** w obiekt klasy **Two** i zauważa konstruktor **Two::Two(One)**, który zostaje niejawnie wywoływany. Utworzony w ten sposób obiekt klasy **Two** jest przekazywany funkcji **f()**.

W powyższym przypadku automatyczna konwersja typów pozwala na uniknięciu kłopotu definiowania dwóch przeciążonych wersji funkcji **f()**. Odbywa się to jednak kosztem niejawnego wywołania funkcji, które może mieć znaczenie w przypadku, gdy istotna jest efektywność wywołania funkcji **f()**.

Zapobieganie konwersji za pomocą konstruktora

Zdarzają się przypadki, w których automatyczna konwersja typów, realizowana za pomocą konstruktora, może stwarzać problemy. Aby ją wykluczyć, należy zmodyfikować konstruktor, poprzedzając go słowem kluczowym **explicit** (jawny), które działa tylko w przypadku konstruktorów. Zamieszczony poniżej przykład ilustruje wykorzystanie słowa kluczowego **explicit** do zmodyfikowania konstruktora klasy **Two**, pochodzącej z poprzedniego programu:

```
//: C12:ExplicitKeyword.cpp
// Wykorzystanie słowa kluczowego "explicit"
class One {
public:
    One() {}
};

class Two {
public:
    explicit Two(const One&) {}
};

void f(Two) {}

int main() {
    One one;
    //! f(one); // Automatyczna konwersja nie jest dozwolona
    f(Two(one)); // W porządku - konwersja dokonana przez użytkownika
} //
```

Określając konstruktor klasy **Two** jako „jawny” (**explicit**), żąda się od kompilatora, by nie przeprowadzał za pomocą tego konstruktora żadnej automatycznej konwersji (inne konstruktory tej klasy, niezdefiniowane jako „jawne”, mogą nadal dokonywać automatycznej konwersji typów). Jeżeli użytkownik chce wykonać konwersję, określoną przez ten konstruktor, musi zapisać to w programie. W powyższym kodzie podczas wywołania **f(Two(one))**, na podstawie obiektu **one**, tworzony jest tymczasowy obiekt klasy **Two** — to samo zrobił kompilator w poprzedniej wersji programu.

Operator konwersji

Drugim sposobem, umożliwiającym automatyczną konwersję typów, jest przeciążenie operatora. Można utworzyć funkcję **składową**, pobierającą aktualny typ i przekształcającą go na typ docelowy, wykorzystując słowo kluczowe **operator**, poprzedzające nazwę typu, do którego ma zostać dokonana konwersja. Jest to wyjątkowa postać przeciążenia operatora, ponieważ sprawia ona wrażenie, jakby nie określono typu zwracanego przez operator — jest nim bowiem *nazwa* przeciążanego operatora. Pоказuje to następujący przykład:

```
//: C12:OperatorOverloadingConversion.cpp
class Three {
    int i;
public:
    Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
    int x;
public:
    Four(int xx) : x(xx) {}
    operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
    Four four(1);
    g(four);
    g(1); // Wywołanie Three(1,0)
} //:~
```

W przypadku konwersji dokonywanej za pomocą konstruktora była za nią odpowiedzialna klasa docelowa. Jednak co do konwersji dokonywanej przy użyciu operatora, to jest ona realizowana przez klasę źródłową. Korzyścią wynikającą z zastosowania techniki, wykorzystującej **konstruktory**, jest możliwość dodawania do istniejącego systemu nowych sposobów konwersji podczas tworzenia nowych klas. Jednakże utworzenie jednoargumentowego konstruktora *zawsze* definiuje automatyczną konwersję typów (nawet gdy posiada on więcej argumentów — pod warunkiem, że pozostałe argumenty będą posiadały wartości domyślne), która nie zawsze może być pożądana (i w tym przypadku można ją wykluczyć, używając słowa kluczowego **explicit**). Nie ma także możliwości zdefiniowania za pomocą konstruktorów konwersji z typów zdefiniowanych przez użytkownika na typy wbudowane — to możliwe wyłącznie za pomocą przeciążania operatorów.

Jednym z najważniejszych powodów stosowania globalnych przeciążonych operatorów zamiast operatorów, będących funkcjami składowymi klas, jest fakt, że w przypadku operatorów globalnych automatyczna konwersja typów może zostać zastosowana w odniesieniu do każdego argumentu. Jeśli zaś chodzi o funkcję **składową**, to argument znajdujący się po lewej stronie operatora musi już być odpowiedniego typu.

Aby konwersji podlegały oba argumenty, należy zastosować globalną wersję operatora **operator=**, pozwalającą na uniknięciu żmudnego kodowania. Ilustruje to przedstawiony poniżej niewielki przykład:

```
//: C12:ReflexivityInOverloading.cpp
class Number {
    int i;
public:
    Number(int ii = 0) : i(ii) {}
    const Number
    operator+(const Number& n) const {
        return Number(i + n.i);
    }
    friend const Number
    operator-(const Number&, const Number&);
};

const Number
operator-(const Number& n1,
            const Number& n2) {
    return Number(n1.i - n2.i);
}

int main() {
    Number a(47), b(11);
    a + b; // W porządku
    a + 1; // Drugi argument przekształcony do typu Number
// 1 + a; // Źle! Pierwszy argument nie jest typu Number
    a - b; // W porządku
    a - 1; // Drugi argument przekształcony do typu Number
    1 - a; // Pierwszy argument przekształcony do typu Number
} //:~
```

Klasa **Number** posiada zarówno **operator+**, będący składową klasy, jak i zaprzyjaźniony **operator-**. Ponieważ istnieje konstruktor, pobierający pojedynczy argument typu **int**, liczby całkowite mogą być automatycznie przekształcane na obiekty klasy **Number**, ale musi się to odbywać w odpowiednich warunkach. Jak widać w funkcji **main()**, dodanie do siebie dwóch obiektów klasy **Number** działa poprawnie, ponieważ odpowiada dokładnie przeciążonemu operatorowi. Również w **przypadku**, gdy kompilator spotyka wyrażenie, w którym po obiekcie klasy **Number** występuje znak **+**, a następnie liczba całkowita, może dopasować to wyrażenie do funkcji składowej **Number::operator+**, przekształcającej za pomocą konstruktora argument całkowity na typ **Number**. Gdy jednak napotyka liczbę **całkowitą**, znak **+**, a następnie obiekt klasy **Number**, to nie wie, co ma robić, ponieważ dysponuje jedynie funkcją **Number::operator+, wymagającą**, by argument znajdujący się po lewej stronie był obiektem klasy **Number**. Dlatego też kompilator zgłasza wówczas błąd.

Zupełnie inaczej wygląda sprawa z zaprzyjaźnionym operatorem odejmowania. Kompilator musi co prawda uzupełnić oba argumenty, ale może to zrobić — nie jest bowiem ograniczony wymaganiem, by argument znajdujący się po lewej stronie był argumentem typu **Number**. Tak więc widząc:

1 - a

może przekształcić pierwszy argument na typ **Number**, używając w tym celu konstruktora.

Czasami występuje potrzeba ograniczenia użycia operatorów przez uczynienie ich składowymi. Na przykład w przypadku mnożenia macierzy przez wektor musi on znajdować się po prawej stronie operatora. Jeżeli chcesz jednak, by operatory mogły dokonać konwersji dowolnego argumentu, zdefiniuj gojako funkcję zaprzyjaźnioną klasę.

Na szczęście, w przypadku wyrażenia **1 - 1**, kompilator nie dokonuje przekształcenia obu argumentów na obiekty klasy **Number**, aby następnie wywołać **operator-**. Oznaczałoby to, że istniejący kod języka C zacząłby nagle działać inaczej. Kompilator rozpoczyna więc od najprostszej **możliwości**, której jest w tym przypadku wbudowany operator, obsługujący wyrażenie **1 - 1**.

Przykład konwersji typów

Przykładem, w którym automatyczna konwersja typów jest wyjątkowo przydatna, są wszelkie klasy zawierające łańcuchy znakowe (w tym przypadku implementujemy po prostu klasę, używając standardowej klasy C++ — **string** — ponieważ można to łatwo wykonać). Aby używać wszystkich istniejących funkcji operujących na łańcuchach, dostępnych w standardowej bibliotece języka C, bez korzystania z automatycznej konwersji typów, należało utworzyć funkcję składową dla każdej z nich, jak przedstawiono poniżej:

```
//: C12:Strings1.cpp
// Brak automatycznej konwersji typów
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
    string s;
public:
    Stringc(const string& str = "") : s(str) {}
    int strcmp(const Stringc& S) const {
        return ::strcmp(s.c_str(), S.s.c_str());
    }
    // ... itd., dla każdej funkcji zawartej w string.h
};

int main() {
    Stringc s1("witam"), s2("wszystkich");
    s1strcmp(s2);
} //:-
```

W powyższym przykładzie utworzona została jedynie funkcja **strcmp()**. Jednak dla każdej funkcji zawartej w pliku nagłówkowym **<cstring>**, która mogłaby być potrzebna, należało utworzyć odpowiadającą jej funkcję. Na szczęście, można udostępnić automatyczną konwersję typów, umożliwiając tym samym dostęp do wszystkich funkcji zawartych w pliku nagłówkowym **<cstring>**:

```
//: C12:Strings2.cpp
// Z automatyczną konwersją typów
```

```
#include "../require.h"
#include <cstring>
#include<cstdlib>
#include <string>
using namespace std;

class Stringc {
    string s;
public:
    Stringc(const string& str = "") : s(str) {}
    operator const char*() const {
        return s.c_str();
    }
};

int main() {
    Stringc s1("hello"), s2("there");
    strcmp(s1, s2); // Standardowa funkcja języka C
    strspn(s1, s2); // Dowolna funkcja operująca nałańcuchach!
} //:-
```

Obecnie każda funkcja pobierająca argument typu **char*** może pobierać również argument typu **Stringc**, ponieważ kompilator wie, w jaki sposób przekształcić obiekt klasy **Stringc** w obiekt typu **char***.

Pułapki automatycznej konwersji typów

Ponieważ to kompilator decyduje o sposobie niejawnego przekształcania typów, niepoprawne zaprojektowanie konwersji może być przyczyną kłopotów. Prostym i oczywistym przykładem jest **klasa X**, posiadająca możliwość przekształcenia obiektu w obiekt klasy Y za pomocą operatora **Y()**. Jeżeli klasa Y posiada konstruktor, pobierający pojedynczy argument typu X, to reprezentuje on taką samą konwersję typów. W razie wystąpienia takiej konwersji kompilator może dokonać konwersji z typu X na Y na dwa sposoby, zgłasza więc błąd dwuznaczności:

```
//: C12>TypeConversionAmbiguity.cpp
class Orange; // Deklaracja klasy

class Apple {
public:
    operator Orange() const; // Konwersja z Apple do Orange
};

class Orange {
public:
    Orange(Apple); // Konwersja z Apple do Orange
};

void f(Orange) {}

int main() {
    Apple a;
    //! f(a); // Błąd - dwuznaczna konwersja
} //:-
```

Oczywistym sposobem rozwiązywania problemu jest unikanie takich sytuacji. Należy po prostu udostępnić tylko jedną możliwość automatycznej konwersji z jednego typu na drugi.

Bardziej złożony problem, któremu należy się przyjrzeć uważniej, występuje wtedy, gdy udostępnia się automatyczną konwersję na więcej niż tylko jeden typ. Jest to czasami nazywane **przeciążeniem wyjścia** (ang. *fan-out*):

```
//: C12:TypeConversionFanout.cpp
class Orange {};
class Pear {};

class Apple {
public:
    operator Orange() const;
    operator Pear() const;
};

// Overloaded eat():
void eat(Orange);
void eat(Pear);

int main() {
    Apple c;
    //! eat(c);
    // Błąd - Apple -> Orange lub Apple -> Pear ???
} ///:-
```

Klasa **Apple** zapewnia automatyczną konwersję — zarówno na typ **Orange**, jak i na **Pear**. Pułapka polega na tym, że nie stanowi to problemu, dopóki ktoś nie utworzy dwóch niewinnie wyglądających, przeciążonych wersji funkcji **eat()** (w przypadku tylkojednej wersji funkcji **eat()** kod zawarty w funkcji **main()** działa bez problemu).

Podobnie jak w poprzednim przypadku, rozwiązywanie tego problemu jest umożliwienie tylkojednego sposobu automatycznej konwersji typu — i powinno to stanowić hasło przewodnie, związane z automatyczną konwersją typów. Można również zdefiniować konwersje na inne typy, ale nie powinny one odbywać się *automatycznie*. Wolno użyć w tym celu jawnych wywołań funkcji o nazwach w rodzaju: **utworzA()** i **utworzB()**.

Ukryte działania

Automatyczna konwersja typów powoduje niekiedy znacznie większą liczbę niejawnych działań niż przewidywane. Przedstawiona poniżej modyfikację programu **CopyingVsInitialization.cpp** można potraktować jako łamigłówkę:

```
//: C12:CopyingVsInitialization2.cpp
class Fi {};

class Fee {
public:
    Fee(int) {}
    Fee(const Fi&) {}
};
```

```
class Fo {  
    int i;  
public:  
    Fo(int x = 0) : i(x) {}  
    operator Fee() const { return Fee(i); }  
};  
  
int main(){  
    Fo fo;  
    Fee fee = fo;  
} //:-~
```

Nie ma konstruktora umożliwiającego utworzenie obiektu **fee** klasy **Fee** na podstawie obiektu klasy **Fo**. Jednakże klasa **Fo** umożliwia automatyczną konwersję obiektu swojego typu na obiekt klasy **Fee**. Nie zdefiniowano co prawda konstruktora kopiącego, zapewniającego możliwość utworzenia obiektu klasy **Fee** na podstawie istniejącego już obiektu tej klasy, ale jest to jedna z funkcji tworzonych automatycznie przez kompilator (domyślny konstruktor, konstruktor **kopiujący**, **operator=** oraz destruktor są automatycznie generowane przez kompilator). Więc dla niewinnie wyglądającej instrukcji:

```
Fee fee = fo;
```

jest wywoływany operator automatycznej konwersji typów oraz tworzony konstruktor kopiący.

Używaj automatycznej konwersji typów z rozwagą. Podobnie jak w przypadku wszystkich przeciążeń operatorów, okazuje się ona **wspaniałym** narzędziem w sytuacjach, w których umożliwia uniknięcie żmudnego kodowania, ale zazwyczaj nie warto jej stosować bez uzasadnionego powodu.

Podsumowanie

Mechanizm przeciążania operatorów wprowadzono w istocie wyłącznie po to, by ułatwić życie programistom. Nie ma w nim nic szczególnie tajemniczego — przeciążone operatory są jedynie funkcjami o dziwnych nazwach, wywoływanymi przez kompilator po napotkaniu odpowiedniego wzorca. Jeżeli jednak przeciążone operatory nie przynoszą tobie (projektantowi klasy) ani użytkownikowi klasy istotnych korzyści, to nie zaprzątaj sobie uwagi ich definiowaniem.

Ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym: *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Utwórz prostą klasę zawierającą przeciżony **operator++**. Spróbuj wywołać ten **operator** — zarówno w przedrostkowej, jak i w przyrostkowej postaci. Zobacz, jakie ostrzeżenia zostaną zgłoszone przez kompilator.
2. Utwórz prostą klasę zawierającą liczbę całkowitą i przeciążony **operator+**, będący funkcją składową. Utwórz również funkcję składową **print()**, **pobierającą argument typu ostream&** i wyprowadzającą do tego strumienia informacje. Przetestuj działanie swojej klasy, by upewnić się, że wszystko funkcjonuje prawidłowo.
3. Do klasy, utworzonej w poprzednim ćwiczeniu, dodaj dwuargumentowy **operator-**, będącej funkcją składową. Pokaż, że możesz używać obiektów tej klasy w złożonych wyrażeniach w rodzaju $a + b - c$.
4. Do klasy, utworzonej w 2. ćwiczeniu, dodaj operatory $++$ oraz $--$, zarówno w wersjach przedrostkowych, jak i przyrostkowych, w taki sposób, aby zwracały one odpowiednio — obiekt o powiększonej lub pomniejszonej wartości składowej. Upewnij się, że przedrostkowe wersje operatorów zwracają poprawne wartości.
5. Zmodyfikuj operatory inkrementacji i dekrementacji, utworzone w poprzednim ćwiczeniu, tak by ich wersje przedrostkowe nie zwracały obiektów będących stałymi, a wersje przyrostkowe — zwracały obiekty będące stałymi. Pokaż, że działają one prawidłowo i wyjaśnij, dlaczego operatory te powinno się definiować w taki właśnie sposób.
6. Zmień funkcję **print()**, utworzoną się w 2. ćwiczeniu — w taki sposób, aby była ona przeciążonym operatorem $<<$, podobnie jak w programie **IostreamOperatorOverloading.cpp**.
7. Zmodyfikuj ćwiczenie 3. w taki sposób, aby **operator+ i operator-** nie były funkcjami składowymi klasy. Pokaż, że nadal działają prawidłowo.
8. Do klasy, utworzonej w 2. ćwiczeniu, dodaj jednoargumentowy **operator-** i pokaż, że działa on poprawnie.
9. Utwórz klasę **zawierającą pojedynczą prywatną składową typu char**. Dokonaj przeciążenia operatorów $<<$ oraz $>>$ (w sposób zaprezentowany w programie **IostreamOperatorOverloading.cpp**) i przetestuj je. Możesz sprawdzić ich działanie ze strumieniami **fstreams, stringstream, cin** oraz **cout**.
10. Sprawdź, jaką sztuczną stałą wartości przekazuje używany przez ciebie kompilator w przypadku przyrostkowych wersji przeciążonych operatorów $++$ oraz $--$.
11. Utwórz klasę **Number**, przechowującą wartość typu **double**, i dodaj do niej przeciążone operatory $+$, $-$, $*$, $/$ oraz przypisania. Wybierz wartości zwracane przez te funkcje w taki sposób, by wyrażenia mogły być łączone w łańcuchy, a także by były one efektywne. Napisz **operator double ()**, umożliwiający automatyczną konwersję typu.
12. Zmodyfikuj poprzednie ćwiczenie w taki sposób, aby zastosować **optymalizację zwracania wartości**, o ile jeszcze tego nie zrobiłeś.

13. Utwórz klasę zawierającą wskaźnik i pokaż, że gdy pozwoli się kompilatorowi na wygenerowanie operatora `=`, to w rezultacie jego użycia powstaną wskaźniki wskazujące ten sam obszar pamięci. Następnie rozwiąż ten problem, definiując własny **operator=**, i pokaż, że jego działanie nie wywołuje kłopotów, polegających na utożsamianiu wskaźników. Upewnij się, że sprawdzasz, czy nie następuje przypisanie do samego siebie, i obsługujesz ten przypadek właściwie.
14. Utwórz klasę o nazwie **Bird** (ptak), zawierającą składową typu **string** oraz statyczną zmienną typu **int**. W domyślnym konstruktorze użyj zmiennej całkowitej do automatycznego wygenerowania identyfikatora, który zostanie umieszczony w zmiennej typu **string** wraz z nazwą klasy (**Bird #1**, **Bird #2** itd.). Dodaj **operator<<**, zdefiniowany dla strumieni wyjściowych **ostream**, umożliwiający drukowanie obiektów klasy **Bird**. Utwórz operator przypisania `=` oraz konstruktor kopiący. Sprawdź w funkcji **main()**, czy wszystko działa prawidłowo.
15. Utwórz klasę o nazwie **BirdHouse** (domek dla ptaków), zawierającą obiekt, wskaźnik oraz referencję do klasy **Bird**, utworzonej w poprzednim ćwiczeniu. Konstruktor klasy powinien pobierać jako argumenty trzy obiekty klasy **Bird**. W klasie **BirdHouse** utwórz **operator<<**, zdefiniowany dla strumieni wyjściowych **ostream**. Utwórz również operator przypisania `=` oraz konstruktor kopiący. Upewnij się w funkcji **main()**, że wszystko działa prawidłowo. Upewnij się, że możesz łączyć łańcuchowo przypisania do obiektów klasy **BirdHouse**, a także budować wyrażenia zawierające wiele operatorów.
16. Do klas **Bird** oraz **BirdHouse**, opisanych w poprzednim ćwiczeniu, dodaj składową typu **int**. Utwórz operatory `+`, `-`, `*`, `/`, będące funkcjami składowymi, wykonującymi operacje na odpowiednich danych składowych każdej z tych klas. Upewnij się, że działają one poprawnie.
17. Powtórz poprzednie ćwiczenie, używając operatorów niebędących funkcjami składowymi.
- 18.** Do programów **SmartPointer.cpp** oraz **NestedSmartPointer.cpp** dodaj **operator--**.
- 19.** Zmodyfikuj program **CopyingVsInitialization.cpp** w taki sposób, by wszystkie występujące w nim konstruktory drukowały komunikaty, informujące o tym, co się aktualnie dzieje. Następnie sprawdź, czy obie formy wywołanie konstruktora kopiącego (w postaci znaku przypisania i z użyciem nawiasu) są sobie równoważne.
- 20.** Spróbuj utworzyć dla jakiejś klasy **operator=** niebędący funkcją składową i zobacz, jakie błędy zgłosi w takim przypadku kompilator.
- 21.** Utwórz klasę, zawierającą konstruktor kopiący, posiadający drugi argument, którym będzie łańcuch o domyślnej wartości „Wywołanie KK”. Utwórz funkcję pobierającą obiekt tej klasy przez wartość i pokaż, że konstruktor kopiący jest wywoływany poprawnie.

22. W programie **CopyingWithPointers.cpp** usuń **operator=** klasy **DogHouse** i pokaż, że wygenerowany przez kompilator **operator=** poprawnie kopiuje łańcuchy, ale w przypadku wskaźnika do obiektu klasy Dog tworzyjedynie jego kolejną kopię.
23. Do programu **ReferenceCounting.cpp**, zarówno do klasy Dog jak i **DogHouse**, dodaj statyczną składową całkowitą oraz zwykłą składową całkowitą. We wszystkich konstruktorach obu klas inkrementuj statyczną całkowitą i jednocześnie przypisz jej wartość zwykłej składowej całkowitej obiektu — zapamiętując w ten sposób liczbę utworzonych obiektów. Dokonaj niezbędnych modyfikacji, dzięki którym wszystkie instrukcje drukujące informacje będą podawały identyfikatory całkowite, przypisane obiektem.
24. Utwórz klasę **zawierającą składową typu string**. Zainicjalizuj ją w konstruktorze, ale nie twórz konstruktora kopiącego ani operatora **=**. Utwórz drugą klasę, **zawierającą składową, będącą obiektem pierwszej klasy**, i również nie twórz dla niej konstruktora kopiącego ani operatora **=**. Pokaż, że jej konstruktor kopiąjący oraz **operator=** zostały poprawnie wygenerowane przez kompilator.
25. Połącz ze sobą klasy zawarte w plikach **OverloadingUnaryOperators.cpp** oraz **Integer.cpp**.
26. Zmodyfikuj program **PointerToMemberOperator.cpp**, dodając do klasy Dog dwie nowe funkcje składowe, niepobierające argumentów i zwracające wartość **void**. Utwórz przeciążony **operator->***, współpracujący z tymi funkcjami, i przetestuj jego działanie.
27. Do programu **NestedSmartPointer.cpp** dodaj **operator->***.
28. Utwórz dwie klasy — **Apple** i **Orange**. W klasie **Apple** utwórz konstruktor, pobierający jako argument obiekt klasy **Orange**. Utwórz funkcję pobierającą obiekt klasy **Apple** i wywołaj ją z obiektem klasy **Orange**, pokazując, że wywołanie takie jest poprawne. Następnie zmodyfikuj konstruktor klasy **Apple**, używając słowa kluczowego **explicit**, by zademonstrować, że zapobiega to automatycznej konwersji typów. Zmodyfikuj wywołanie funkcji w taki sposób, aby konwersja była dokonywana w jawnym sposób, a program ponownie działał poprawnie.
29. Do programu **ReflexivityInOverloading.cpp** dodaj globalny **operator*** i pokaż, że działa on w sposób symetryczny.
30. Utwórz dwie klasy, a następnie zdefiniuj **operator+** oraz funkcje odpowiedzialne za konwersje w taki sposób, aby dodawanie było symetryczne w stosunku do obu tych klas.
31. Popraw program **TypeConversionFanout.cpp**, tworzącą funkcję, wywoływaną w celu konwersji typów i zastępującą jedną z operatorów automatycznej konwersji typów.
32. Napisz prosty program, używający operatorów **+**, **-**, ***** oraz **/** w stosunku do liczb typu **double**. Dowiedz się, w jaki sposób wygenerować na wyjściu kompilatora C++ program w asemblerze, i przyjrzyj się wygenerowanemu kodowi, by zrozumieć, jak działa, a następnie to wyjaśnij.

Rozdział 13.

Dynamiczne tworzenie obiektów

Czasami znana jest dokładna liczba, typy i czas życia obiektów. Ale nie zawsze.

Iloma samolotami będzie zarządzał system kontroli lotów? Ilu figur geometrycznych wymaga system **CAD**? Z ilu węzłów będzie składała się sieć?

Do rozwiązywania ogólnych problemów programistycznych konieczna jest możliwość tworzenia i niszczenia obiektów w czasie pracy programu. Oczywiście, język C zawsze udostępniał funkcje **dynamicznej alokacji pamięci**, **malloc()** i **free()** (a także inne warianty funkcji **malloc()**), przydzielające w trakcie pracy programu pamięć ze **sterty** (zwanej czasami **również pamięcią wolną**).

Jednak w języku C++ to po prostu nie zadziała. Konstruktor nie pozwoli na przekazanie sobie adresu pamięci, którą ma zainicjalizować — i są ku temu powody. Gdyby było to możliwe, należałoby:

1. Zapomnieć o tym. W ten sposób gwarantowana inicjalizacja obiektów w języku C++ nie byłaby już taka.
2. Przypadkowo zrobić coś z obiektem, zanim zostałaby zainicjalizowany, spodziewając się, że będzie to **działało** poprawnie.
3. Dostarczyć obiekt niewłaściwej wielkości.

Oczywiście, nawet gdyby wszystko zostało wykonane prawidłowo, to **każdy**, kto modyfikowałby później program, mógłby popełnić te błędy. Niewłaściwa inicjalizacja jest odpowiedzialna za znaczną część problemów programistycznych, więc gwarancja wywołania konstruktorów w stosunku do obiektów utworzonych na stercie jest szczególnie istotna.

W jaki jednak sposób język C++ zapewnia poprawną inicjalizację i sprzątanie, pozwalając równocześnie użytkownikowi na dynamiczne tworzenie obiektów na stercie?

Wszystkie te trzy regiony pamięci są często umieszczane w pojedynczym, ciągłym obszarze fizycznej pamięci — znajdują się w nim kolejno: obszar danych statycznych, stos oraz sterta (w porządku określonym przez twórcę kompilatora). Jednak nie obowiązują w tej kwestii żadne reguły. Stos może znajdować się w jakimś szczególnym miejscu, a sterta może być zaimplementowana w postaci żądań przydzielenia bloków pamięci, kierowanych do systemu operacyjnego. Programista nie zajmuje się zazwyczaj tymi zagadnieniami, więc powinien myśleć jedynie o tym, że gdy potrzebna mu jest pamięć, to znajduje się ona właśnie w tym miejscu.

Obsługa sterty w języku C

Język C zawiera w swojej standardowej bibliotece następujące funkcje, umożliwiające dynamiczne przydzielanie pamięci w trakcie pracy programu: **malloc()** oraz jej warianty, **calloc()** i **realloc()**, dostarczające pamięć ze sterty, oraz **free()**, zwracającą z powrotem przydzieloną pamięć. Funkcje te są praktyczne w użyciu, lecz prymitywne — wymagają one zarówno zrozumienia, jak i uwagi ze strony programisty. Aby utworzyć na stercie egzemplarz klasy, wykorzystując w tym celu dostępne w języku C funkcje obsługujące pamięć **dynamiczną**, należałooby wykonać przykładowo następujące działania:

```
//: C13:MallocClass.cpp
// Funkcja malloc() z obiektami klas
// Co trzeba by zrobić, gdyby nie było operatora "new"
#include "../require.h"
#include <cstdlib> // malloc() i free()
#include <cstring> // memset()
#include <iostream>
using namespace std;

class Obj {
    int i, j, k;
    enum { sz = 100 };
    char buf[sz];
public:
    void initialize() { // Nie można użyć konstruktora
        cout << "inicjalizacja Obj" << endl;
        i = j = k = 0;
        memset(buf, 0, sz);
    }
    void destroy() const { // Nie można użyć destruktora
        cout << "niszczenie Obj" << endl;
    }
};

int main() {
    Obj* obj = (Obj*)malloc(sizeof(Obj));
    require(obj != 0);
    obj->initialize();
    // ... nieco później:
    obj->destroy();
    free(obj);
} ///:-
```

Sposób wykorzystania funkcji **malloc()** do przydzielenia obiektowi pamięci ilustruje poniższy wiersz:

```
Obj* obj = (Obj*)malloc(sizeof(Obj));
```

W przedstawionej instrukcji użytkownik musi określić wielkość obiektu (to pierwsze miejsce, w którym można popełnić błąd). Funkcja **malloc()** zwraca wartość typu **void***, ponieważ tworzy ona po prostu obszar pamięci, a nie obiekt. Język C++ nie pozwala na przypisanie wartości typu **void*** jakiemukolwiek innemu wskaźnikowi, trzeba więc użyć rzutowania.

Z uwagi na to, że funkcja **malloc()** może nie przydzielić wymaganej pamięci (i zwrócić w takim przypadku wartość zerową), trzeba sprawdzić wartość zwróconego przez nią wskaźnika, by upewnić się, że jej wywołanie zakończyło się powodzeniem.

Jednak znacznie poważniejszy problem stanowi wiersz:

```
Obj->initialize();
```

Jeżeli użytkownik postępował do tej pory właściwie, musi pamiętać o inicjalizacji obiektu, zanim przystąpi do korzystania z niego. Należy zwrócić uwagę na to, że nie można w tym przypadku użyć konstruktora, ponieważ nie ma sposobu wywołania go w jawnym sposób¹ — konstruktor jest bowiem wywoływany przez kompilator w trakcie tworzenia obiektu. Problem polega na tym, że użytkownik może zapomnieć o dokonaniu inicjalizacji przed rozpoczęciem używania obiektu, umieszczając w ten sposób w programie źródło potencjalnych poważnych błędów.

Okazuje się również, że dla wielu programistów funkcje dynamicznego przydziału pamięci, dostępne w języku C, wydają się niezrozumiałe i skomplikowane — nie należy do rzadkości widok programisty języka C, wykorzystującego mechanizmy pamięci wirtualnej do przydzielenia ogromnych tablic zmiennych w obszarze danych statycznych — tylko po to, by nie zaprzątać sobie głównego dynamicznego przydziału pamięci. Z uwagi na to, że język C++ próbuje uczynić wykorzystywanie bibliotek bezpiecznym i łatwym dla zwykłego programisty, nie sposób zaakceptować podejścia do pamięci dynamicznej przyjętego w języku C.

Operator new

Przyjętym w języku C++ rozwiązaniem jest połączenie w jeden operator o nazwie **new** wszystkich działań koniecznych do utworzenia obiektu. Podczas generowania obiektu za pomocą operatora **new** (poprzez wyrażenie *new*) przydziela się na stercie ilość pamięci, niezbędną do zapamiętania obiektu, i wywołuje się w stosunku do niej konstruktor. Tak więc użycie instrukcji:

```
MojTyp *fp = new MojTyp(1.2);
```

podczas pracy programu jest równoważne wywołaniu funkcji **malloc(sizeof(MojTyp))** (często jest to rzeczywiście wywołanie funkcji **malloc()**) oraz konstruktora klasy

¹ Istnieje specjalna konstrukcja, nazywana *umieszczaniem new*, pozwalająca na wywołanie konstruktora dla przydzielonego uprzednio obszaru pamięci. Została ona opisana w dalszej części rozdziału.

MojTyp, z adresem przydzielonej pamięci w charakterze wskaźnika **this** oraz listą argumentów (1,2). W chwili przypisania adresu wskaźnikowi **fp** wskazywany obszar pamięci jest już istniejącym, zainicjowanym obiektem (wcześniej nie było nawet do niego dostępu). Ponadto jest on od razu typu **MojTyp**, dzięki czemu nie ma potrzeby rzutowania wskaźnika.

Domyślnie operator **new** upewnia się, że przydzielenie pamięci zakończyło się powodzeniem, zanim jeszcze przekaże adres konstruktorowi. Nie ma więc potrzeby jawnego sprawdzania, czy jego wywołanie zakończyło się pomyślnie. W dalszej części rozdziału dowiesz się, co dzieje się w przypadku, gdy nie będzie możliwości przydzielania obiektowi pamięci.

W wyrażeniu **new** można wykorzystać dowolny dostępny konstruktor klasy. Jeżeli konstruktor nie posiada argumentów, to zapisuje się wyrażenie **new**, nie podając listy argumentów konstruktora:

```
MojTyp *fp = new MojTyp;
```

Zwróci uwagę na to, jak prosty stał się proces tworzenia obiektów na stercie — składa się on obecnie tylko z jednego wyrażenia, zawierającego wszelkie informacje o wielkości, konwersje i testy związane z bezpieczeństwem. Obiekt na stercie może być teraz utworzony równie łatwo, jak na stosie.

Operator **delete**

Wyrażeniem komplementarnym w stosunku do wyrażenia **new** jest *wyrażenie delete*, które najpierw wywołuje destruktory, a następnie zwalnia przydzieloną uprzednio pamięć (często wykonuje to, wywołując funkcję **free()**). Podobnie jak wyrażenie **new** zwraca wskaźnik do obiektu, wyrażenie **delete** wymaga podaniajego adresu:

```
delete fp;
```

Powyzsze wyrażenie powoduje destrukcję, a następnie zwolnienie pamięci zajmowanej przez utworzony wcześniej dynamicznie obiekt klasy **MojTyp**.

Operator **delete** może być wywoływany wyłącznie w stosunku do obiektu utworzonego wcześniej za pomocą operatora **new**. Jeżeli obszar pamięci obiektu został przydzielony za pomocą funkcji **malloc()** (albo **calloc()** czy **realloc()**), a następnie użyje się w stosunku do niego operatora **delete**, to działanie tego operatora nie będzie zdefiniowane. Z uwagi na to, że większość domyślnych implementacji operatorów **new** i **delete** wykorzystuje funkcje **malloc()** oraz **free()**, prawdopodobnie skończy się to zwolnieniem pamięci, bez wywołania destruktora.

Jeżeli wskaźnik usuwany za pomocą operatora **delete** jest zerowy, to nic się nie stanie. Z tego powodu niekiedy zaleca się przypisanie wskaźnikowi wartości zerowej bezpośrednio po jego usunięciu, co zapobiega ponownemu usunięciu wskazywanego obiektu. Usuwanie obiektu częściej niż jednokrotnie jest złym pomysłem i z pewnością spowoduje problemy.

P prosty przykład

Poniższy przykład pokazuje, że w trakcie dynamicznego tworzenia obiektu jest dokonywana inicjalizacja:

```
//: C13:Tree.h
#ifndef TREE_H
#define TREE_H
#include <iostream>

class Tree {
    int height;
public:
    Tree(int treeHeight) : height(treeHeight) {}
    -Tree() { std::cout << "*"; }
    friend std::ostream&
    operator<<(std::ostream& os, const Tree* t) {
        return os << "Wysokosc drzewa wynosi: "
               << t->height << std::endl;
    }
};

#endif // TREE_H //:~
//: C13>NewAndDelete.cpp
// Prosta demonstracja operatorów new i delete
#include "Tree.h"
using namespace std;

int main() {
    Tree* t = new Tree(40);
    cout << t;
    delete t;
} //:~
```

Drukując wartość obiektu klasy **Tree**, można udowodnić, że wywoływany jest jej konstruktor. Wyprowadzenie wartości jest w tym przypadku realizowane za pomocą przeciążenia operatora `<<` w taki sposób, by można go użyć w stosunku do strumienia **ostream** i wskaźnika do obiektu klasy **Tree**. Jednak mimo że funkcja ta jest zadeklarowana jako zaprzyjaźniona, jest ona zdefiniowana jako funkcja inline! Zrobiono to wyłącznie dla wygody — zdefiniowanie zaprzyjaźnionej funkcji jako inline nie zmieniaje statusu, określonego słowem kluczowym **friend**, ani faktu, że jest ona funkcją globalną, a nie funkcją składową klasy. Warto również zauważyć, że wartość zwracana przez tę funkcję jest wynikiem całego wyrażenia i jest ona typu **ostream&** (co jest konieczne dla zapewnienia zgodności z typem wartości zwracanej przez funkcję).

Narzut menedżera pamięci

Podczas automatycznego tworzenia obiektów na stosie informacja o ich wielkości i czasie życia jest umieszczana bezpośrednio w generowanym kodzie, ponieważ kompilatorowi znane są dokładnie: typ, wielkość i zasięg obiektu. Tworzenie obiektów na stercie wiąże się z dodatkowym narzutem, zarówno pod względem czasowym, jak i pamięciowym. Poniżej przedstawiono typowy scenariusz takiej operacji (funkcję **malloc()** można zastąpić przez **calloc()** lub **realloc()**).

Wywoływana jest funkcja **malloc()**, zgłaszająca żądanie przydziału pamięci z dostępnej puli (kod ten może w rzeczywistości stanowić część funkcji **malloc()**).

W celu znalezienia dostatecznie dużego bloku pamięci, by zrealizować zgłoszone żądanie, przeszukiwana jest dostępna pula pamięci. Realizuje się to zapomocą przeglądającegoś rodzący mapy lub katalogu, informujących o tym, które bloki pamięci są aktualnie używane, a które dostępne. Proces ten jest stosunkowo szybki, ale może wymagać szeregu prób, w związku z czym może nie być deterministyczny. Nie można więc liczyć na to, że wywołanie funkcji **malloc()** zajmie za każdym razem tyle samo czasu.

Zanim zostanie zwrócony wskaźnik przydzielonego bloku pamięci, jego wielkość i położenie muszą zostać zapamiętane, dzięki czemu następne wywoływanie funkcji **malloc()** nie wykorzystają go ponownie, a po wywołaniu funkcji **free()** system będzie wiedział, ile pamięci należy zwolnić.

Wszystko to można zaimplementować na różne sposoby. Na przykład nic nie stoi na przeszkodzie, aby podstawowe operacje systemu przydziału pamięci były realizowane przez procesor. Jeżeli cię to interesuje, to możesz napisać programy testowe, które pomogą w określeniu, w jaki sposób w używanym przez ciebie systemie została zaimplementowana funkcja **malloc()**. Możesz również przeczytać kod źródłowy biblioteki (zawsze dostępne sąźródła bibliotek GNU C).

Zmiany w prezentowanych wcześniej przykładach

Wykorzystując operatory **new** i **delete** oraz wszystkie omawiane powyżej cechy języka, można zmodyfikować przykładową klasę **Stash**, zaprezentowaną w pierwszej części książki. Przyjrzenie się powstałemu na nowo kodowi będzie stanowiło pożyteczny przegląd opisywanych zagadnień.

W przedstawionych powyżej rozważaniach ani klasa **Stash**, ani **Stack** nie „posiadały” wskazywanych przez siebie obiektów — gdy obiekty klas **Stash** i **Stack** wykraczały poza zasięg, klasy te nie usuwały wskazywanych przez siebie obiektów. Powodem, dla którego nie było to możliwe, był fakt, że w celu zachowania ogólnego charakteru klas **Stash** i **Stack** przechowywały one wskaźniki typu **void***. Jeżeli usunie się taki wskaźnik, można spodziewać się jedynie zwolnienia zajmowanej pamięci — z uwagi na to, że nie zawiera on żadnej informacji o typie, kompilator nie wie, jaki powinien wywołać destruktor.

usuwanie wskaźnika **void*** jest prawdopodobnie błędem

Warto w tym miejscu podkreślić, że wywołanie operatora **delete** w stosunku do wskaźnika typu **void*** stanowi niemal na pewno błąd, chyba że obiekt, wskazywany przez ten wskaźnik, jest bardzo prosty — w szczególności nie powinien on posiadać destruktora. Poniższy przykład pokazuje, co się w takim przypadku dzieje:

```
//: C13:BadVoidPointerDeletion.cpp
// Usuwanie wskaźników typu void* może powodować
// wyciekanie pamięci
#include <iostream>
using namespace std;

class Object {
    void* data; // Jakiś obszar pamięci
    const int size;
    const char id;
public:
    Object(int sz, char c) : size(sz), id(c) {
        data = new char[size];
        cout << "Konstrukcja obiektu " << id
            << ", size - " << size << endl;
    }
    ~Object() {
        cout << "Destrukcja obiektu " << id << endl;
        delete []data; // W porządku, tylko zwalniana jest
        // pamięć - nie ma potrzeby wywoływanego destruktora
    }
};

int main() {
    Object* a = new Object(40, 'a');
    delete a;
    void* b = new Object(40, 'b');
    delete b;
} //:-~
```

Klasa **Object** zawiera wskaźnik typu **void***, inicjalizowany za pomocą „zwykłych” danych (nie wskazuje obiektów posiadających destruktory). W destruktore klasie **Object** w stosunku do tego wskaźnika jest wywoływany operator **delete** — nie pochęca to za sobą żadnych niepożądanych skutków, ponieważ chcemy w ten sposób jedynie zwolnić przydzielony uprzednio obszar pamięci.

Jak jednak widać w funkcji **main()**, jest bardzo ważne, aby operator **delete** znał typ obiektu, w stosunku do którego został użyty. Wyniki działania programu są następujące:

```
Konstrukcja obiektu a, size = 40
Destrukcja obiektu a
Konstrukcja obiektu b. size = 40
```

Ponieważ wyrażenie **delete a** „wie”, że zmienna **a** jest wskaźnikiem do obiektu klasy **Object**, wywoływany jest destruktur i tym samym następuje zwalnianie pamięci wskazywanej przez składową **data**. Jednakże w przypadku gdy operuje się obiektem, wykorzystując wskaźnik typu **void***, jak w wyrażeniu **delete b**, zwalniana jest tylko pamięć, przydzielona obiekowi wskazywanemu przez ten wskaźnik. Nie jest natomiast wywoływany destruktur obiektu, a w związku z czym nie jest również zwalniana pamięć, wskazywana przez składową **data**. Podczas komplikacji tego programu nie pojawią się prawdopodobnie żadne ostrzeżenia — kompilator założy, że programista wie, co robi. Efektem będzie natomiast bardzo powolne wyciekanie pamięci.

Jeżeli w twoim programie wycieka pamięć, przyjrzyj się wszystkim zawartym w nim instrukcjom **delete** i sprawdź, jakiego typu są usuwane przez nie wskaźniki. Jeżeli któryś ze wskaźników ma typ **void***, oznacza to, że prawdopodobnie została odkryta jedna z przyczyn wyciekania pamięci (**język C++** stwarza wiele okazji umożliwiających wyciekanie pamięci).

Odpowiedzialność za sprzątanie wskaźników

W celu zapewnienia elastyczności kontenerom **Stash** oraz **Stack** (aby mogły one przechowywać dowolne typy obiektów) będą one zawierać wskaźniki typu **void***. Oznacza to, że przed użyciem wskaźnika, zwróconego przez obiekt klasy **Stash** czy **Stack**, trzeba dokonać jego rzutowania na odpowiedni typ — tak jak w powyższym przykładzie. Rzutowanie takie należy również wykonać przed usunięciem obiektu, gdyż w przeciwnym razie spowoduje to wyciekanie pamięci.

Drugą kwestią związaną z wyciekaniem pamięci jest konieczność zapewnienia, by operator **delete** został rzeczywiście wywołany w stosunku do wszystkich obiektów, przechowywanych w kontenerze. Kontener nie może być „właścicielem” wskaźnika, ponieważ przechowuje go jako wskaźnik typu **void*** i w związku z tym nie jest w stanie poprawnie przeprowadzić sprzątania. Za sprzątanie obiektu musi być odpowiedzialny użytkownik. Stanowi to poważny problem w przypadku, gdy do jednego kontenera doda się wskaźniki, wskazujące zarówno obiekty utworzone na stosie, jak i na stercie. W stosunku do wskaźników, którym nie została przydzielona pamięć na stercie, nie można bowiem w bezpieczny sposób użyć wyrażenia **delete** (a skąd, po pobraniu wskaźnika z kontenera, można się dowiedzieć, gdzie był on przydzielony?). Tak więc trzeba mieć pewność, że obiekty przechowywane w nowej wersji kontenerów **Stash** i **Stack** zostały utworzone wyłącznie na stercie — albo zachowując ostrożność podczas programowania, albo tworząc klasy, których obiekty mogą być utworzone tylko na stercie.

Należy również upewnić się, że klient-programista weźmie na siebie odpowiedzialność za posprzątanie wszystkich wskaźników zawartych w kontenerze. W zamieszczonych poprzednio przykładach można było zaobserwować, w jaki sposób destruktor klasy **Stack** sprawdza, czy wszystkie obiekty klasy **Link** zostały pobrane ze „stosu”. W przypadku klasy **Stash** trzeba jednak zastosować odmienne rozwiązanie.

Klasa **Stash** przechowująca wskaźniki

Nowa wersja klasy **Stash**, nosząca nazwę **PStash**, przechowuje wskaźniki do obiektów utworzonych na stercie, podczas gdy wersja tej klasy, przedstawiona w poprzednich rozdziałach, kopiowała obiekty przez wartość — bezpośrednio do kontenera. Gdy używa się operatorów **new** i **delete**, przechowywanie wskaźników do obiektów utworzonych na stercie jest łatwe i bezpieczne.

Poniżej zamieszczono plik nagłówkowy klasy **Stash** w wersji przechowującej wskaźniki:

```
//: C13:PStash.h
// Przechowuje wskaźniki zamiast obiektów
#ifndef PSTASH_H
#define PSTASH_H

class PStash {
    int quantity; // Liczba elementów pamięci
    int next; // Następny pusty element
    // Pamięć wskaźników:
    void** storage;
    void inflate(int increase);
public:
    PStash() : quantity(0), storage(0), next(0) {}
    ~PStash();
    int add(void* element);
    void* operator[](int index) const; // Pobranie elementu
    // Usunięcie odwołania do elementu:
    void* remove(int index);
    // Liczba zapamiętanych elementów:
    int count() const { return next; }
};

#endif // PSTASH_H ///:~
```

Podstawowe elementy danych nie uległy zasadniczym zmianom, ale składowa **storage** jest obecnie tablicą wskaźników typu **void***. Przydzielenie pamięci tej tablicy odbywa się za pomocą operatorka new, a nie funkcji **malloc()**. W wyrażeniu:

```
void** st - new void*[quantity + increase];
```

typem przydzielanego obiektu jest **void***, więc wyrażenie to przydziela po prostu tablicę wskaźników typu **void***.

Destruktor usuwa pamięć przechowującą wskaźniki typu **void***, natomiast nie próbuje usuwać tego, co one wskazują (jak już wcześniej wspomniano, zwolniłoby to jedynie pamięć, nie wywołując destruktorek —wskaźniki typu **void*** nie zawierają bowiem informacji o typie).

Kolejną zmianą jest zastąpienie funkcji **fetch()** operatorem **[]**, który — z punktu widzenia składni — wydaje się bardziej logiczny. Ponieważ jednak operator ten zwraca wartości typu **void***, więc użytkownik musi ponownie pamiętać o typach obiektów przechowywanych w kontenerze i dokonywać rzutowania pobieranych wskaźników (problem ten zostanie rozwiązany w następnych rozdziałach).

Poniżej znajdują się definicje funkcji składowych klasy:

```
//: C13:PStash.cpp {0}
// Definicje klasy Stash, przechowującej wskaźniki
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <cstring> // funkcje 'mem'
using namespace std;

int PStash::add(void* element) {
    const int inflateSize = 10;
    if(next >= quantity)
        inflate(inflateSize);
```

```

storage[next++] - element;
return(next - 1); // Numer indeksu
}

// Obiekty nie są własnością kontenera:
PStash::~PStash() {
    for(int i = 0; i < next; i++)
        require(storage[i] == 0,
               "PStash nie został uprzatniety");
    delete []storage;
}

// Zamiast funkcji fetch użyto przeciążonego operatora
void* PStash::operator[](int index) const {
    require(index >= 0,
           "PStash::operator[] indeks ma wartość ujemną");
    if(index >= next)
        return 0; // Oznaczenie końca
    // Tworzenie wskaźnika do zadanego elementu:
    return storage[index];
}

void* PStash::remove(int index) {
    void* v = operator[](index);
    // "Usuwa" wskaźnik:
    if(v != 0) storage[index] = 0;
    return v;
}

void PStash::inflate(int increase) {
    const int psz = sizeof(void*);
    void** st = new void*[quantity + increase];
    memset(st, 0, (quantity + increase) * psz);
    memcpy(st, storage, quantity * psz);
    quantity += increase;
    delete []storage; // Stary obszar pamięci
    storage = st; // Wskaźnik do nowego obszaru pamięci
} //:-
}

```

Funkcja **add()** działa właściwie tak samo, jak poprzednio, jednak obecnie, zamiast kopiowania całych obiektów, w kontenerze przechowywane są jedynie wskaźniki.

Funkcja **inflate()**, która w poprzedniej wersji operowała tylko na zwykłych ciągach bajtów, została zmodyfikowana w taki sposób, by mogła obsługiwać tablice wskaźników typu `void*`. Przydzielony obszar pamięci, zamiast stosowanego poprzednio kopowania kolejnych elementów tablicy, jest najpierw zerowany za pomocą standardowej funkcji języka C — **memset()** (nie jest to konieczne, ponieważ klasa **PStash** prawdopodobnie obsługuje pamięć prawidłowo, ale odrobina ostrożności nigdy nie zaszkodzi). Następnie funkcja **memcpy()** przenosi istniejące dane z dawnego obszaru pamięci do nowego. Takie funkcje, jak **memset()** oraz **memcpy()**, są nierzadko optymalizowane latami, dzięki czemu mogą działać znacznie szybciej niż pętle, wykorzystywane w poprzedniej wersji programu. Z drugiej strony w przypadku takich funkcji, jak **inflate()**, które prawdopodobnie nie będą używane zbyt często, można nie zauważać żadnej różnicy w wydajności. Jednakże już tylko to, że wywołania funkcji są bardziej zwięzłe niż pętle, może pomóc w ustrzeżeniu się błędów związanych z kodowaniem.

W celu obarczenia klienta-programisty odpowiedzialnością za sprzątanie obiektów udostępniono dwie metody, umożliwiające dostęp do wskaźników przechowywanych w obiekcie klasy **PStash** — **operator[]**, zwracający wskaźnik, ale pozostawiający go w kontenerze, oraz funkcję składową **remove()**, również zwracającą wskaźnik, ale zarazem usuwającą go z kontenera i przypisującą do niego wartość zerową. Gdy wywoływany jest destruktor klasy **PStash**, sprawdza on, czy wskaźniki wszystkich obiektów zostały usunięte. Jeżeli tak nie jest, wyświetla komunikat, co zapobiega wyciekaniu pamięci (bardziej eleganckie rozwiązanie tego problemu zostanie przedstawione w następnych rozdziałach).

Test

Poniżej znajduje się program testowy klasy **Stash**, dostosowany do współpracy z klasą **PStash**:

```
//: C13:PStashTest.cpp
//{L} PStash
// Test klasy Stash, przechowującej wskaźniki
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    PStash intStash;
    // Operator "new" działa również z wbudowanymi typami
    // Zwróć uwagę na składnię "pseudokonstruktora":
    for(int i = 0; i < 25; i++)
        intStash.add(new int(i));
    for(int j = 0; j < intStash.count(); j++)
        cout << "intStash[" << j << "] = "
            << *(int*)intStash[j] << endl;
    // Sprzątanie:
    for(int k = 0; k < intStash.count(); k++)
        delete intStash.remove(k);
    ifstream in ("PStashTest.cpp");
    assure(in, "PStashTest.cpp");
    PStash stringStash;
    string line;
    while(getline(in, line))
        stringStash.add(new string(line));
    // Drukowanie łańcuchów:
    for(int u = 0; stringStash[u]; u++)
        cout << "stringStash[" << u << "] = "
            << *(string*)stringStash[u] << endl;
    // Sprzątanie:
    for(int v = 0; v < stringStash.count(); v++)
        delete (string*)stringStash.remove(v);
} //:-~
```

Podobnie jak poprzednio, obiekty klasy **Stash** są tworzone, a następnie wypełniane informacjami — jednak tym razem informacjami są wskaźniki, powstałe w wyniku użycia wyrażenia **new**. Zwróć uwagę na wiersz znajdujący się w pierwszej części programu:

```
intStash.add(new int(i));
```

Wyrażenie **new int(i)** wykorzystuje składnię **pseudokonstruktora**, w związku z czym pamięć dla nowego obiektu typu **int** jest przydzielana na stercie, a następnie obiekt ten jest inicjalizowany wartością zmiennej **i**.

Podczas wydruku wartość zwracana przez funkcję **PStash::operator[]** musi być rzutowana na odpowiedni typ — odbywa się to również w stosunku do pozostałych obiektów klasy **PStash**, zawartych w programie. Jest to niepożądany efekt, związany z użyciem wskaźników typu **void*** wewnętrznej reprezentacji klasy. Problem ten zostanie rozwiązany w następnych rozdziałach książki.

Drugi z testów otwiera plik źródłowy i wczytuje jego kolejne wiersze do innego obiektu klasy **PStash**. Każdy wiersz tekstu jest wczytywany do zmiennej **line** typu **string** za pomocą funkcji **getline()**, a następnie, na podstawie zawartości tej zmiennej, na stercie jest tworzony łańcuch zawierający jego niezależną kopię. Gdybyśmy za każdym razem przekazywali funkcji **PStash::add()** adres zmiennej **line**, to w rezultacie otrzymalibyśmy całą grupę wskaźników, wskazujących zmenną **line**, a zmienna ta zawierałaby jedynie wiersz ostatnio odczytany z pliku.

Podczas pobierania wskaźników widoczne jest wyrażenie:

```
*(string*)stringStash[v]
```

Wskaźnik, zwracany przez **operator[]** w celu nadania mu odpowiedniego typu, musi być rzutowany na typ **string***. Następnie jest on wyłuskiwany, dzięki czemu wynikiem wyrażenia jest obiekt typu **string**, następnie przekazywany przez kompilator do strumienia **cout**.

Utworzone na stercie obiekty muszą zostać zniszczone za pomocą funkcji **remove()**, gdyż w przeciwnym razie w czasie pracy programu pojawi się komunikat informujący, że nie wszystkie obiekty przechowywane w obiekcie klasy **PStash** zostały prawidłowo usunięte. Zwróć uwagę na to, że w przypadku liczb całkowitych nie jest potrzebne rzutowanie, ponieważ nie posiadają one destruktora, a zależy nam tylko na zwolnieniu zajmowanej przez nie pamięci:

```
delete intStash.remove(k);
```

Jeżeli jednak w przypadku wskaźników do łańcuchów zapomni się o rzutowaniu, to rezultatem będzie kolejne (powolne) wyciekanie pamięci. Dlatego też rzutowanie jest niezbędne:

```
delete (string*)stringStash.remove(k);
```

Niektóre z przedstawionych problemów (ale nie wszystkie) można rozwiązać za pomocą szablonów (zob. rozdział 16.).

Operatory new i delete dla tablic

W języku C++ tablice mogą być tworzone z równą łatwością, zarówno na stosie, jak i na stercie, a dla każdego obiektu znajdującego się w tablicy wywoływany jest, oczywiście, konstruktor. Obowiązuje tu jednak jedno ograniczenie — z wyjątkiem przypadku inicjalizacji agregatowej, dokonywanej na stosie (opisanej w rozdziale 6.), klasa musi posiadać konstruktor domyślny, ponieważ dla każdego obiektu musi pozostać wywołyany konstruktor nieposiadający argumentów.

Podczas tworzenia tablic obiektów na stercie za pomocą operatora new należy jednak uczynić coś jeszcze. Przykładem takiej tablicy jest:

```
MojTyp* fp = new MojTyp[100];
```

Powyższa instrukcja przydziela na stercie pamięć, niezbędną dla 100 obiektów klasy MojTyp, i dla każdego z tych obiektów wywołuje konstruktor. Jednakże obecnie dysponujemy wskaźnikiem typu MojTyp*, takim samym, jak uzyskany w wyniku wykonania instrukcji:

```
MojTyp* fp2 = new MojTyp;
```

tworzącej pojedynczy obiekt. Ponieważ to my napisaliśmy kod, wiemy, że wskaźnik fp jest w rzeczywistości adresem początku tablicy, dlatego też możliwy jest wybór elementów tej tablicy za pomocą wyrażeń w rodzaju fp[3]. Co jednak dzieje się w chwili niszczenia tej tablicy? Wyrażenia:

```
delete fp2; // W porządku  
delete fp; // Nie przyniesie spodziewanego rezultatu
```

wyglądają tak samo i ich rezultat będzie również identyczny. Dla obiektu klasy MojTyp, wskazywanego przez podany adres, zostanie wywołyany destruktory, a następnie zwolniona pamięć. W przypadku wskaźnika fp2 działa to doskonale, ale dla wskaźnika fp oznacza, że nie zostanie wywołanych pozostałych 99 destruktory. Zostanie jednak nadal zwolniona właściwa ilość pamięci, ponieważ została ona przydzielona w postaci jednego, dużego obszaru. Wielkość tego obszaru jest przechowywana w jakimś miejscu przez procedurę zajmującą się przydziałem pamięci.

Rozwiążanie tego problemu wymaga poinformowania kompilatora, że zwalniany obszar jest w rzeczywistości adresem początku tablicy. Uzyskuje się to za pomocą następującej składni:

```
delete []fp;
```

Pusty nawias kwadratowy przekazuje kompilatorowi informację, by wygenerował kod. Pobiera on liczbę elementów zawartych w tablicy, zapisaną w jakimś miejscu podczas jej tworzenia, a następnie wywołuje destruktory dla takiej właśnie liczby obiektów. W rzeczywistości jest to udoskonalona składnia, wywodząca się z wcześniejszej postaci instrukcji, którą można jeszcze czasami spotkać w starszych programach:

```
delete [100]fp;
```

a która zmuszała programistę do wpisania liczby obiektów zawartych w tablicy, wprowadzając zarazem możliwość popełnienia pomyłki. Dodatkowy narzut, związany z obsługą tej liczby przez kompilator, jest bardzo niewielki, a ponadto uznano, że lepiej określić liczbę obiektów tylko w jednym miejscu programu zamiast we dwóch.

Upodabnianie wskaźnika do tablicy

Nawiasem mówiąc, wartość zdefiniowanego powyżej wskaźnika **fp** można zmienić w taki sposób, by wskazywał cokolwiek, co nie ma sensu w przypadku adresu początku tablicy. Bardziej logiczne jest zdefiniowanie go jako stałej, dzięki czemu każda próba jego modyfikacji zakończy się zgłoszeniem błędu. Aby uzyskać ten efekt, można użyć następującej konstrukcji:

```
int const* q = new int[10];
```

lub

```
const int* q = new int[10];
```

lecz w obu przypadkach słowo kluczowe **const** zostanie związane z typem **int**, to znaczy ze *wskazywanym* obiektem, a nie z samym wskaźnikiem. Zamiast tego należy napisać:

```
int* const q = new int[10];
```

Obecnie elementy tablicy wskazywanej przez **q** mogą być modyfikowane, ale niedozwolona jest jakakolwiek zmiana wartości wskaźnika **q** (np. **q++**) — podobnie jak w przypadku zwykłego identyfikatora tablicy.

Brak pamięci

Co się dzieje, gdy operator **new** nie potrafi znaleźć ciągłego obszaru pamięci, wystarczającą **dużego**, by pomieścić tworzony obiekt? W takim przypadku wywoływana jest specjalna funkcja, nazywana *funkcją obsługi operatora new* (ang. *new handler*). Ścisłej, sprawdzany jest wskaźnik do tej funkcji, a jeśli ma on wartość **niezerową**, wywoływana jest wskazywana przez niego funkcja.

Domyślnym zachowaniem funkcji obsługi operatora **new** jest *zgłoszenie wyjątku* — temat ten został opisany w drugim tomie książki. Gdy jednak używa się w programie przydziału pamięci na stercie, warto przynajmniej zastąpić funkcję obsługi operatora **new** własną **funkcją**, która wyświetli komunikat o braku pamięci, a następnie spowoduje zakończenie działania programu. W ten sposób podeczas uruchamiania programu będzie dostępna wskazówka, informująca o tym, co się stało. W ostatecznej wersji programu trzeba będzie użyć rzetelniejszej metody naprawczej.

Aby wymienić funkcję obsługi operatora **new**, należy dołączyć do programu plik nagłówkowy **new.h**, a następnie wywołać funkcję **set_new_handler()**, podając jej adres funkcji, która ma zostać zainstalowana:

```
//: C13>NewHandler.cpp
// Zmiana funkcji obsługi operatora new
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

int count = 0;
```

```
void out_of_memory() {
    cerr << "pamiec wyczerpała sie po " << count
        << " przydzielach!" << endl;
    exit(1);
}

int main() {
    set_new_handler(out_of_memory);
    while(1) {
        count++;
        new int[1000]; // Powoduje wyczerpanie pamięci
    }
} //:~
```

Funkcja obsługi operatora **new** nie może pobierać argumentów i musi zwracać wartość **void**. Pętla **while** będzie przydzielać na stercie obiekty całkowite (wyrzucając ich adresy) dopóty, dopóki całkowicie wyczerpie się dostępna pamięć. Podczas następnego użycia operatora **new** pamięć nie może już zostać przydzielona, w związku z czym jest wywoływana funkcja obsługi operatora **new**.

Zachowanie funkcji obsługi operatora **new** jest związane z operatorem **new**, więc jeśli przeciąży się **operator new** (co zostanie opisane w następnym podrozdziale), to domyślnie nie zostanie wywołana jego funkcja obsługi. Jeżeli chcesz, aby funkcja ta była nadal wywoływana, należy umieścić kod wywołujący tę funkcję wewnątrz przeciążonego operatora **new**.

Oczywiście, można napisać bardziej wyrafinowane funkcje obsługi operatora **new**, nawet takie, które podejmą próbę odzyskania pamięci (są one powszechnie znane pod nazwą *zbieraczy śmieci*). Nie jest to jednak zadanie dla początkującego programisty.

Przeciążanie operatorów **new** i **delete**

Podczas tworzenia wyrażenia **new** najpierw, za pomocą operatora **new**, przydzielana jest pamięć, a następnie wywoływany jest konstruktor. Natomiast w przypadku wyrażenia **delete** najpierw wywoływany jest destruktor, a później, za pomocą operatora **delete**, zwalniana jest pamięć. Wywołania konstruktora oraz destruktorów nie są nigdy dostępne (w przeciwnym przypadku można by je bowiem przypadkowo uszkodzić), można jednak zmienić funkcje odpowiedzialne za przydział pamięci — **operator new** oraz **operator delete**.

System przydziału pamięci, wykorzystywany przez operatory **new** i **delete**, jest przeznaczony do ogólnego użytku. W szczególnych sytuacjach może on jednak nie spełniać oczekiwanych. Najczęściej spotykana przyczyną zmian sposobu przydziału pamięci jest efektywność — może zdarzyć się sytuacja, w której przydzielanych i zwalnianych będzie tak wiele obiektów jakiejś klasy, że obsługa pamięci stanie się wąskim gardłem systemu. Język C++ pozwala na przeciążenie operatorów **new** i **delete** w celu implementacji własnego systemu alokacji pamięci, dzięki czemu można uporać się z takimi problemami.

Inną kwestią jest fragmentacja sterty. Podczas przydzielania pamięci obiektom różnych rozmiarów możliwe jest rozdrobnienie sterty, które w praktyce doprowadzi do wyczerpania się dostępnej pamięci. Pamięć mogłaby być jeszcze dostępna, ale z uwagi na fragmentację żaden jej obszar nie jest dostatecznie duży, by zaspokoić aktualne potrzeby. Tworząc własny system przydziału pamięci, przeznaczony dla konkretnej klasy, można zagwarantować, że nigdy się to nie zdarzy.

W systemach wbudowanych oraz w systemach czasu rzeczywistego może wystąpić konieczność bardzo długiej pracy programu, dysponującego nader ograniczonymi zasobami. System taki może ponadto wymagać, by przydział pamięci zabierał zawsze tyle samo czasu, nie dopuszczając możliwości wyczerpania się pamięci na stercie lub wystąpienia jej fragmentacji. W takim przypadku rozwiązaniem jest własny system przydziału pamięci — w przeciwnym razie programiści musieliby całkowicie unikać używania operatorów **new** i **delete**, nie mogąc wykorzystać tych cennych właściwości języka C++.

Podczas przeciążania operatorów **new** i **delete** należy pamiętać, że zmienia się tylko sposób, **w jaki przydzielana jest zwykła pamięć**. Kompilator wykorzysta przygotowaną przez nas funkcję **new**, zamiast używanej domyślnie do przydzielania pamięci, a następnie wywoła konstruktor dla przydzielonego obszaru pamięci. Tak więc mimo że kompilator, widząc operator **new**, przydziela pamięć *oraz* wywołuje konstruktor, to przeciążając operator **new** można zmienić jedynie jego część, odpowiedzialną za przydział pamięci (podobne ograniczenie dotyczy operatora **delete**).

Przeciążając **operator new**, zmienia się również jego zachowanie związane z brakiem pamięci. O podejmowanym wówczas działaniu trzeba zatem zdecydować wewnątrz tego operatora — zwrócić wartość **zerową**, napisać pętlę, wywołującą funkcję obsługi operatora **new** i ponawiającą próby przydziału pamięci, lub (w typowym przypadku) zgłosić wyjątek **bad_alloc** (omawiany w drugim tomie książki, dostępnym w witrynie <http://helion.pl/online/thinking/index.html>).

Przeciążanie operatorów **new** i **delete** nie różni się od przeciążania innych operatorów. Istnieje jednak możliwość wyboru, czy ma zostać przeciążony globalny system przydziału pamięci, czy też system używany do przydzielania pamięci konkretnej **klasie**.

Przeciążanie globalnych operatorów **new** i **delete**

Jest to drastyczne posunięcie, stosowane gdy globalne wersje operatorów **new** i **delete** działają w sposób niezadowalający w całym systemie. Przeciążenie globalnych wersji operatorów spowoduje, że domyślne operatory staną się całkowicie niedostępne — nie będzie można ich wywoływać nawet z wnętrza nowych, przygotowanych przez siebie definicji.

Przeciążony operator **new** musi pobierać argument typu **size_t** (standardowy typ rozmiarów, określony w standardzie języka C). Argument ten jest generowany i przekazywany funkcji przez kompilator — określa on wielkość obiektu, któremu należy przydzielić pamięć. Funkcja musi zwrócić albo wskaźnik do obiektu żądanej wielkości (albo większej, jeżeli są do tego jakieś powody), albo wartość **zerową**, w przypadku gdy nie można znaleźć wolnej pamięci (wówczas konstruktor *nie zostanie wywołany!*).

Jednakże gdy nie ma możliwości przydzielenia żądanej pamięci, należałoby prawdopodobnie zrobić coś bardziej konstruktywnego niż zwrócenie po prostu wartości zerowej — na przykład wywołać funkcję obsługi operatora new lub zgłosić wyjątek, sygnalizując wystąpienie problemu.

Wartością zwracaną przez **operator new** jest **void***, a *nie* wskaźnik do jakiegoś określonego typu. Dostarczana jest jedynie pamięć, a nie tworzony obiekt. Tworzenie obiektu dokonuje się dopiero podczas wywołania **konstruktora**, czyli działania gwarantowanego przez kompilator, znajdującego się poza naszą kontrolą.

Operator **delete** pobiera natomiast wskaźnik typu **void*** do pamięci, która została uprzednio przydzielona za pomocą operatora **new**. Jest on typu **void***, ponieważ **operator delete** otrzymuje ten wskaźnik dopiero po wywołaniu destruktora, który pozbawia zajmowaną przez obiekt pamięć cech właściwych obiektom. Wartością zwracaną przez **operator delete** jest **void**.

Poniżej znajduje się prosty przykład, ilustrujący sposób przeciążania globalnych operatorów **new** i **delete**:

```
//: C13:GlobalOperatorNew.cpp
// Przeciążanie globalnych operatorów new i delete
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
    printf("operator new: %d bajtów\n", sz);
    void* m = malloc(sz);
    if(!m) puts("brak pamięci");
    return m;
}

void operator delete(void* m) {
    puts("operator delete");
    free(m);
}

class S {
    int i[100];
public:
    S() { puts("S::S()"); }
    ~S() { puts("S::~S()"); }
};

int main() {
    puts("tworzenie i niszczenie zmiennej całkowitej");
    int* p = new int(47);
    delete p;
    puts("tworzenie i niszczenie obiektu s");
    S* s = new S;
    delete s;
    puts("tworzenie i niszczenie tablicy S[3]");
    S* sa = new S[3];
    delete []sa;
} //:~
```

W powyższym programie przedstawiono ogólną postać przeciążonych operatorów **new** i **delete**. Wykorzystują one do przydzielu pamięci funkcje standardowej biblioteki języka C — **malloc()** oraz **free()** (są one prawdopodobnie używane również przez standardowe operatory **new** i **delete!**). Ponadto drukują one komunikaty, informujące o tym, co aktualnie robią. Zwrót uwagę na to, że zamiast strumieni wejścia-wyjścia są tu używane funkcje **printf()** oraz **puts()**. Podczas tworzenia obiektu klasy **iostream** (na przykład globalne obiekty **cin**, **cout** oraz **cerr**) jest bowiem wywoływany operator **new**, przydzielający mu pamięć. Użycie funkcji **printf()** nie grozi blokadą systemu, gdyż funkcja ta nie wywołuje operatora **new** podczas swojej inicjalizacji.

W funkcji **main()** tworzone są obiekty wbudowanych typów; mają one udowodnić, że również w ich przypadku wywoływane są przeciążone wersje operatorów **new** oraz **delete**. Następnie tworzony jest pojedynczy obiekt klasy S, a później również tablica obiektów klasy S. W przypadku tablicy na podstawie liczby żądanego bajtów można wywnioskować, że przydzielana jest dodatkowa pamięć, służąca do zapisania informacji (w obrębie tablicy) o liczbie obiektów zawartych w tablicy. We wszystkich przypadkach używane są przeciążone wersje operatorów **new** i **delete**.

Przeciążanie operatorów new i delete w obrębie klasy

Mimo że nie ma konieczności jawnego deklarowania jako funkcji statycznych przeciążonych operatorów **new** i **delete**, znajdujących się w obrębie klasy, utworzymy je jako statyczne funkcje składowe. Podobnie jak poprzednio, składnia jest taka sama, jak w przypadku przeciążania każdego innego operatora. Gdy kompilator napotyka operator **new**, użyty do utworzenia obiektu naszej klasy, to zamiast globalnej wersji operatora wybiera funkcję składową **operator new**. Jednak globalne wersje operatorów **new** i **delete** są nadal używane w stosunku do wszystkich pozostałych typów obiektów (o ile nie mają one własnych wersji operatorów **new** i **delete**).

W poniższym przykładzie dla klasy **Framis** utworzono prosty system przydzielu pamięci. Fragment pamięci zostaje przydzielony „na bok” — w statycznym obszarze pamięci, na początku pracy programu; pamięć ta jest wykorzystywana do przydzielania pamięci obiektom klasy **Framis**. W celu określenia, które bloki zostały już przydzielone, jest stosowana prosta tablica bajtów, zawierająca pojedyncze bajcie dla każdego bloku:

```
//: C13:Framis.cpp
// Lokalne przeciążenie operatorów new i delete
#include <cstddef> // size_t
#include <fstream>
#include <iostream>
#include <new>
using namespace std;
ofstream out("Framis.out");

class Framis {
    enum { sz = 10 };
    char c[sz]; // Nie używana - zajmuje tylko pamięć
    static unsigned char pool[];
    static bool alloc_map[];

public:
    enum { psze = 100 }; // Liczba dopuszczalnych obiektów
```

```

Framis() { out << "Framis()\n"; }
~Framis() { out << "~Framis() ..."; }
void* operator new(size_t) throw(bad_alloc);
void operator delete(void*);
};

unsigned char Framis::pool[psize * sizeof(Framis)];
bool Framis::alloc_map[psize] = {false};

// Wielkość jest ignorowana - zakłada się,
// że jest to obiekt klasy Framis
void*
Framis::operator new(size_t) throw(bad_alloc) {
    for(int i = 0; i < psize; i++) {
        if(!alloc_map[i]) {
            out << "używany blok " << i << " ... ";
            alloc_map[i] = true; // Oznaczenie bloku jako używanego
            return pool + (i * sizeof(Framis));
        }
    }
    out << "brak pamięci" << endl;
    throw bad_alloc();
}

void Framis::operator delete(void* m) {
    if(!m) return; // Sprawdzenie czy wskaźnik nie jest pusty
    // Zakładamy, że obiekt został utworzony w dostępnej puli
    // Wyznaczanie numeru przydzielonego bloku:
    unsigned long block = (unsigned long)m
        - (unsigned long)pool;
    block /= sizeof(Framis);
    out << "zwalnianie bloku " << block << endl;
    // Oznaczenie bloku jako wolnego:
    alloc_map[block] = false;
}

int main() {
    Framis* f[Framis::psize];
    try {
        for(int i = 0; i < Framis::psize; i++)
            f[i] = new Framis;
        new Framis; // Brak pamięci
    } catch(bad_alloc) {
        cerr << "Brak pamięci!" << endl;
    }
    delete f[10];
    f[10] = 0;
    // Użycie zwolnionej pamięci:
    Framis* x = new Framis;
    delete x;
    for(int j = 0; j < Framis::psize; j++)
        delete f[j]; // Usuwanie f[10] - w porządku
    } //:-)
}

```

Pula pamięci, używana jako sterta dla obiektów klasy **Framis**, została utworzona jako tablica bajtów — dostatecznie duża, by pomieścić w sobie **psize** obiektów klasy **Framis**. Mapa przydziału zawiera **psize** elementów, więc dla każdego bloku przeznaczono jedną **zmienną** typu **bool**. Wszystkie wartości mapy przydziału zostały zainicjowane wartościami **false**, za pomocą sztuczki z inicjalizacją agregatową pierwszego

elementu tablicy. Polega ona na tym, że kompilator automatycznie inicjalizuje jej wszystkie pozostałe elementy wartością domyślną (**która**, w przypadku typu **bool**, jest wartość **false**).

Lokalny **operator new** ma taką samą składnię, jak operator globalny. Sprawdza on jedynie mapę przydzielu, poszukując w niej wartości **false**. Następnie zmienia wartość znalezionej elementu na **true**, zaznaczając w ten sposób, że został on już przydzielony, a na koniec zwraca adres odpowiadającego mu bloku pamięci. Jeżeli funkcja operatora **new** nie zdoła znaleźć żadnego wolnego bloku pamięci, to wprowadza do „pliku śledzenia” komunikat, a następnie zgłasza wyjątek **bad_alloc**.

To pierwszy przykład wyjątków zamieszczony w niniejszej książce. Z uwagi na to, że wyjątki zostały dokładnie omówione dopiero w drugim tomie książki, zaprezentowano w tym miejscu jedynie bardzo prosty sposób ich użycia. W funkcji operatora **new** można zauważać dwa elementy związane z obsługą wyjątków. Po pierwsze, bezpośrednio po liście argumentów funkcji znajduje się specyfikacja **throw(bad_alloc)**, informująca kompilator (oraz czytelnika), że funkcja może zgłosić wyjątek typu **bad_alloc**. Po drugie, jeśli nie można przydzielić pamięci, to funkcja zgłasza wyjątek za pomocą instrukcji **throw bad_alloc**. Gdy zostanie zgłoszony wyjątek, funkcja przerwuje swoją pracę, a sterowanie zostaje przekazane do *procedury obsługi wyjątku* (ang. *exception handler*), zrealizowanej w postaci klauzuli **catch**.

W funkcji **main()** można dostrzec inny element tego obrazu, którym jest klauzula **try-catch**. Blok **try** jest zawarty w nawiasie klamrowym i znajduje się w nim cały kod, który może zgłosić wyjątki — w tym przypadku są to wywołania operatora **new**, zawierające obiekty klasy **Framis**. Bezpośrednio po bloku **try** następuje jedna lub więcej klauzul **catch**, z których każda określa typ wykrywanego przez siebie wyjątku. W tym przypadku klauzula **catch(bad_alloc)** informuje, że wyłapane zostaną wyjątki **bad_alloc**. Klauzula ta zostanie wykonana tylko **wtedy**, gdy nastąpi zgłoszenie wyjątku typu **bad_alloc**, a później wykonanie programu będzie kontynuowane od miejsca znajdującego się za ostatnią spośród klauzul **catch**, tworzących grupę (w tym przypadku istnieje tylko jedna klauzula **catch**, ale może być ich więcej).

W powyższym przykładzie można używać strumieni wejścia-wyjścia, ponieważ globalne operatory **new** i **delete** pozostały nienaruszone.

Operator **delete** zakłada, że adres obiektu klasy **Framis** został utworzony w obrębie dostępnej puli. Założenie to jest usprawiedliwione, ponieważ lokalny operator **new** jest wywoływany za każdym razem, gdy na stercie tworzony jest pojedynczy obiekt klasy **Framis**. Jednak nie dzieje się tak w przypadku tablicy obiektów — w stosunku do tablic wywoływany jest globalny operator **new**. Tak więc użytkownik może przypadkowo wywołać operator **delete**, nie używając pustego nawiasu kwadratowego, sygnalizującego destrukcję tablicy. Mogłoby to spowodować problemy. Użytkownik może również podać wskaźnik do obiektu utworzonego na stosie. Jeżeli takie sytuacje wydają się możliwe, to należałoby dodać wiersz, sprawdzający, czy adres **zawarty** jest w obrębie puli, z której przydzielana jest pamięć, i czy wskazuje on początek bloku (być może dostrzegasz już możliwość **przeciążenia** operatorów **new** i **delete** w celu wykrycia wycieków pamięci).

Operator **delete** wyznacza numer bloku reprezentowanego przez wskaźnik, a następnie przypisuje znacznikowi mapy przydziału tego bloku wartość **false**, oznaczającą, że blok ten został zwolniony.

W funkcji **main()** zostaje dynamicznie przydzielonych tyle obiektów klasy **Framis**, by zabrakło dla nich pamięci — umożliwia to sprawdzenie zachowania operatora **new** w razie braku pamięci. Następnie jeden z obiektów jest usuwany, a inny tworzony, by pokazać, że zwolniona pamięć może być powtórnie wykorzystana.

Z uwagi na to, że przyjęty sposób przydziału pamięci został zaprojektowany specjalnie dla obiektów klasy **Framis**, jest prawdopodobnie znacznie szybszy niż system przydziału pamięci ogólnego zastosowania, używany w przypadku domyślnych operatorów **new** i **delete**. Należy jednak wiedzieć, że nie będzie on automatycznie działał w razie wykorzystania dziedziczenia (opisanego w rozdziale 14.).

Przeciążanie operatorów new i delete w stosunku do tablic

Jeżeli operatory **new** i **delete** zostaną przeciążone w obrębie klasy, to zostaną one wywołane za każdym razem, gdy będzie dynamicznie tworzony obiekt tej klasy. Jednak jeżeli utworzy się *tablicę* tych obiektów, to do przydzielania obszaru pamięci, który pomieści od razu całą tę tablicę, zostanie wywołany globalny **operator new[]**, a do zwolnienia tej pamięci — globalny **operator delete[]**. Przydział pamięci tablicom obiektów można kontrolować, przeciążając w obrębie klasy specjalne wersje operatora **new[]** oraz operatora **delete[]**. Poniższy przykład pokazuje, kiedy wywoływane są te wersje operatorów:

```
//: C13:ArrayOperatorNew.cpp
// Operator new dla tablic
#include <new> // Definicja size_t
#include <fstream>
using namespace std;
ofstream trace("ArrayOperatorNew.out");

class Widget {
    enum { sz = 10 };
    int i[sz];
public:
    Widget() { trace << "*"; }
    ~Widget() { trace << "~"; }
    void* operator new(size_t sz) {
        trace << "Widget::new: "
            << sz << " bajtów" << endl;
        return ::new char[sz];
    }
    void operator delete(void* p) {
        trace << "Widget::delete" << endl;
        ::delete []p;
    }
    void* operator new[](size_t sz) {
        trace << "Widget::new[]: "
            << sz << " bajtów" << endl;
        return ::new char[sz];
```

```

}

void operator delete[](void* p) {
    trace << "Widget::delete[]" << endl;
    ::delete []p;
}
};

int main() {
    trace << "new Widget" << endl;
    Widget* w = new Widget;
    trace << "\ndelete Widget" << endl;
    delete w;
    trace << "\nnew Widget[25]" << endl;
    Widget* wa = new Widget[25];
    trace << "\ndelete []Widget" << endl;
    delete []wa;
} //:~
}

```

W powyższym przykładzie wywoływane są globalne wersje operatorów **new** i **delete**. Rezultat jest więc taki sam, jakby w ogóle nie było przeciążonych wersji tych operatorów, z wyjątkiem tego, że zostały dodane informacje, umożliwiające śledzenie wykonywania kodu. Oczywiście, w przeciążonych wersjach operatorów **new** i **delete** można wykorzystać dowolny system przydzielania pamięci.

A zatem składnia operatorów **new** i **delete** przeznaczonych dla tablic jest taka sama, jak w przypadku ich wersji dla indywidualnych obiektów, z wyjątkiem użycia nawiasów kwadratowych. W obu przypadkach dostarczana jest informacja o wielkości pamięci, która musi zostać przydzielona. Wielkość przekazana wersji działającej w przypadku tablic będzie wielkością całej tablicy. Warto zapamiętać, że *jedynym* działaniem wymagany od przeciążonego operatora **new** jest zwrócenie przez niego wskaźnika do odpowiednio dużego bloku pamięci. Mimo że można by również w tym miejscu dokonać inicjalizacji przydzielonej pamięci, to zwykle jest to zadanie konstruktora, który zostanie automatycznie wywoływany dla tego obszaru pamięci przez kompilator.

Konstruktor i destruktor drukują po prostu znaki, dzięki którym można zobaczyć, kiedy zostały one wywołane. Oto jak wygląda zawartość pliku śledzenia dla przykładowego kompilatora:

```

new Widget
Widget: :new: 40 bajtów
*
delete Widget
~Widget::delete

new Widget[25]
Widget: :new[]: 1004 bajtów
*****
delete []Widget
-----Widget::delete[]

```

Jak się tego można było spodziewać, utworzenie indywidualnego obiektu wymagało 40 bajtów (na tym komputerze liczby całkowite zajmują cztery bajty). Wywoływany jest **operator new**, a następnie konstruktor (oznaczony znakiem *). Analogicznie, wywołanie **delete** powoduje, że najpierw wywoływany jest destruktor, a następnie **operator delete**.

Jak już wspomniano, podczas tworzenia tablicy obiektów klasy **Widget** jest wywoływana „tablicowa” wersja operatora new. Zwróć jednak uwagę na to, że żądanym rozmiar pamięci jest większy o cztery bajty od oczekiwanej. Te cztery bajty służą systemowi do zapamiętania informacji dotyczących tablicy, a szczególnie liczby znajdujących się w niej obiektów. Tak więc w zapisie:

```
delete []Widget;
```

nawias klamrowy informuje kompilator, że usuwana jest tablica obiektów. Dzięki temu kompilator generuje kod, który odczytuje liczbę elementów zawartych w tablicy, a następnie tyłokrotnie wywołuje destruktor. Można zauważyć, że chociaż tablicowe wersje operatorów **new** i **delete** są wywoływanie tylko jednokrotnie — dla całego obszaru zajmowanego przez tablicę — to domyślny konstruktor oraz destruktor są wywoływanie w stosunku do każdego obiektu znajdującego się w tablicy.

Wywołania konstruktora

Mając na uwadze, że instrukcja:

```
MojTyp* f * new MojTyp;
```

najpierw wywołuje operator **new**, przydzieliąc obszar pamięci mieszczący obiekt typu **MojTyp**, a następnie wywołuje dla tego obszaru konstruktor klasy **MojTyp**, można postawić pytanie, co się stanie, gdy nie powiedzie się przydzielenie pamięci za pomocą operatora **new?** W takim przypadku konstruktor nie jest wywoływany, więc mimo że nadal dysponujemy jedynie niepomyślnie skonstruowanym obiektem, to przynajmniej nie zostaje dla niego wywołany konstruktor, któremu musiałaby zostać przekazana zerowa wartość wskaźnika **this**. Oto przykład, który tego dowodzi:

```
//: C13:NoMemory.cpp
// Konstruktor nie jest wywoływany, gdy wywołanie
// operatora new skończy się niepowodzeniem
#include <iostream>
#include <new> // Definicja bad_alloc
using namespace std;

class NoMemory {
public:
    NoMemory() {
        cout << "NoMemory::NoMemory()" << endl;
    }
    void* operator new(size_t sz) throw(bad_alloc){
        cout << "NoMemory::operator new" << endl;
        throw bad_alloc(); // "Brak pamięci"
    }
};

int main() {
    NoMemory* nm = 0;
    try {
        nm = new NoMemory;
    } catch(bad_alloc) {
        cerr << "Wystąpił brak pamięci" << endl;
    }
    cout << "nm = " << nm << endl;
} //:~
```

Podczas pracy programu nie jest wyświetlany komunikat konstruktora, a jedynie komunikat operatora new oraz wyświetlany przez procedurę obsługi wyjątku. Ponieważ nigdy nie następuje powrót z funkcji new, nigdy nie jest też **wywoływany** konstruktor i w związku z tym ani razu nie jest też drukowany jego komunikat.

Ważne jest, aby wskaźnik nm został zainicjowany wartością **zerową**, ponieważ wyrażenie new nigdy nie zostaje wykonane do końca. Wartość wskaźnika powinna natomiast wynosić zero, by mieć pewność, że nigdy nie zostanie on **niewłaściwie** użyty. Jednakże w procedurze obsługi wyjątku należałoby w rzeczywistości zrobić coś więcej niż tylko wydrukowanie komunikatu i kontynuowanie programu w taki sposób, jakby obiekt został prawidłowo utworzony. W idealnym przypadku trzeba by uczynić coś, co umożliwiłoby programowi rozwiązanie problemu. Przynajmniej należy zakończyć jego pracę, odnotowując wcześniej wystąpienie błędu w dzienniku.

We wcześniejszych wersjach języka C++ standardową praktyką w przypadku braku pamięci było zwracanie przez operator new wartości zerowej. Zapobiegało to wywołaniu konstruktora. Jeżeli jednak podejmie się próbę zwrócenia wartości zerowej operatora new, to zgodny ze standardem języka kompilator powinien poinformować, że należy zamiast tego zgłosić wyjątek **bad_alloc**.

Operatory umieszczania new i delete

Istnieją jeszcze dwa inne, rzadziej spotykane zastosowania przeciążonego operatora new:

1. Zamierzamy umieścić obiekt w określonym miejscu pamięci. To szczególnie ważne w przypadku sprzętowych systemów wbudowanych, w których obiekt może być synonimem jakiegoś elementu sprzętowego.
2. Podczas wywołania funkcji new chcemy mieć możliwość wyboru spośród różnych sposobów przydziału pamięci.

W obu tych sytuacjach wykorzystamy ten sam mechanizm: przeciążony operator new może pobierać więcej niż jeden argument. Jak już wiadomo, pierwszym jego argumentem jest **zawsze** wielkość obiektu, niejawnie wyznaczana i przekazywana przez kompilator. Pozostałe argumenty mogą być jednak dowolne — adres, pod którym chcemy umieścić obiekt, referencja do funkcji przydzielającej mu pamięć, obiekt lub cokolwiek innego, co tylko może okazać się przydatne.

Sposób, w jaki przekazywane są operatorowi new dodatkowe argumenty, może na pierwszy rzut oka wydawać się nieco osobliwy. Należy umieścić listę argumentów (*bez* argumentu `size_t`, obsługiwianego przez kompilator) po słowie kluczowym `new`, a przed nazwą klasy, której obiekt jest tworzony. Na przykład w instrukcji:

```
X* xp = new(a) X;
```

zmienna `a` zostanie przekazana operatorowi `new` jako drugi argument. Oczywiście, będzie to działać tylko w przypadku, gdy taki operator `new` zostanie w ogóle zdefiniowany.

Poniższy przykład ilustruje, w jaki sposób można umieścić obiekt pod konkretnym adresem:

```
//: C13:PlacementOperatorNew.cpp
// Umieszczanie za pomocą operatora new
#include <cstddef> // Size_t
#include <iostream>
using namespace std;

class X {
    int i;
public:
    X(int ii = 0) : i(ii) {
        cout << "this = " << this << endl;
    }
    ~X() {
        cout << "X::~X(): " << this << endl;
    }
    void* operator new(size_t, void* loc) {
        return loc;
    }
};

int main() {
    int t[10];
    cout << "t = " << t << endl;
    X* xp = new(1) X(47); // Obiekt X jest umieszczany w tablicy t
    xp->~X(); // Jawne wywołanie destruktora
    // Używane TYLKO w przypadku umieszczania!
} //
```

Zwróć uwagę na to, że **operator new** zwraca jedynek wskaźnik, który jest mu przekazywany. Tak więc postać wywołującego kodu decyduje o tym, gdzie w pamięci zostanie umieszczony obiekt, a następnie — jako część wyrażenia new — dla tego obszaru pamięci zostanie wywołany konstruktor.

Mimo że powyższy przykład prezentuje tylko jeden dodatkowy argument, nic nie stoi na przeszkodzie, by dodać kolejne, jeżeli są one potrzebne do innych celów.

Problemem jest natomiast usunięcie obiektu. Istnieje tylko jedna wersja operatora **delete**, nie ma więc możliwości, by nakazać: „do tego obiektu użyj mojego specjalnego mechanizmu, zwalniającego pamięć”. Zamierzamy wywołać destruktora; nie chcemy jednak, by pamięć została zwolniona za pomocą mechanizmu obsługującego pamięć dynamiczną, ponieważ nie została ona przydzielona na stercie.

Rozwiązaniem tego problemu jest bardzo nietypowa składnia. Używając zapisu:

```
xp->~X(); // Jawne wywołanie destruktora
```

można jawnie wywołać destruktora. Jednak konieczne jest w tym miejscu poważne ostrzeżenie. Niektórzy traktują to jako sposób niszczenia obiektów w dowolnej chwili, zanim jeszcze skończy się ich zasięg. Należy natomiast skorygować ten zasięg lub (bardziej poprawnie) **użyć** dynamicznego tworzenia obiektów (w przypadku gdy czas życia obiektów ma być określany w trakcie wykonania programu). Wywołanie destruktora w stosunku do obiektu utworzonego na stose spowoduje poważne **problemy**, ponieważ zostanie on wywołany ponownie, na końcu zasięgu obiektu. Jeżeli natomiast destruktork zostanie wywołany w taki sposób w stosunku do obiektu **utworzonego**

na stercie, to zostanie on co prawda **wykonany**, ale nie nastąpi zwolnienie pamięci zajmowanej przez obiekt. A zatem cel nie zostanie prawdopodobnie osiągnięty. Jedynym powodem, dla którego destruktor może zostać wywołany jawnie w taki sposób, jest obsługa składni umieszczania operatora **new**.

Istnieje również operator umieszczania **delete**, wywoływany jedynie wówczas, gdy konstruktor wyrażenia umieszczania **new** zgłosi wyjątek (dzięki czemu pamięć zostanie automatycznie wyczyszczona w przypadku wystąpienia wyjątku). Operator umieszczania **delete** posiada listę argumentów, odpowiadającą operatorowi umieszczania **new**, który został wywołany przed zgłoszeniem wyjątku. Temat ten opisano w rozdziale poświęconym obsłudze wyjątków w drugim tomie książki.

Podsumowanie

Tworzenie automatycznych zmiennych na stosie jest wygodne i najbardziej efektywne. Jednakże do rozwiązywania ogólnych problemów programistycznych konieczna jest możliwość tworzenia i niszczenia obiektów w trakcie wykonywania programu — szczególnie gdy jest to wynikiem reakcji na informacje pochodzące spoza programu. Mimo że mechanizm dynamicznego przydziału pamięci języka C przydziela pamięć na stercie, to nie jest on łatwy w użyciu, a ponadto nie gwarantuje dokonania konstrukcji, koniecznej w języku C++. Dzięki przeniesieniu mechanizmu dynamicznego tworzenia obiektów do wnętrza języka tworzenie obiektów na stercie za pomocą operatorów **new** i **delete** jest równie proste, jak w przypadku tworzenia ich na stosie. Dodatkową korzyścią takiego rozwiązania jest jego duża elastyczność. Jeśli działanie operatorów **new** i **delete** nie odpowiada naszym potrzebom — zwłaszcza gdy nie jest ono dostatecznie efektywne — można je zmienić. Istnieje również możliwość modyfikacji zachowania programu w przypadku, gdy skończy się dostępna na stercie pamięć.

Ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Utwórz klasę **Counted**, zawierającą składową całkowitą **id**, oraz statyczną składową całkowitą **count**. Domyślny konstruktor tej klasy powinien rozpoczynać się od: „**Counted() : id(count++)**“. Powinien on również drukować wartość składowej **id** oraz informować, że obiekt został utworzony. Destruktor powinien informować o tym, że został wywołany i również drukować wartość składowej **id**. Przetestuj działanie klasy.
2. Tworząc obiekt klasy **Counted** (z poprzedniego ćwiczenia) za pomocą operatora **new** i usuwając go za pomocą operatora **delete**, upewnij się, że podczas użycia operatorów **new** i **delete** zawsze wywoływanie są konstruktory i destruktory. Utwórz również na stercie tablicę obiektów tej klasy, a następnie ją zniszcz.

3. Utwórz obiekt klasy **PStash** i wypełnij go obiektami klasy, opisanej w ćwiczeniu 1., utworzonymi za pomocą operatora **new**. Zobacz, co się stanie, gdy skończy się zasięg obiektu klasy **PStash** i wywołany zostanie jego destruktor.
4. Utwórz wektor **vector< Counted*>** i wypełnij go wskaźnikami do obiektów klasy **Counted**, opisanej w ćwiczeniu 1. Obiekty te utwórz za pomocą operatora **new**. Przejdź przez wszystkie elementy wektora, drukując zawarte w nim obiekty, a następnie przejdź przez nie ponownie, usuwając (za pomocą operatora **delete**) każdy z nich.
5. Powtórz poprzednie ćwiczenie, dodając do klasy **Counted** funkcję składową **f()**, drukującą komunikat. Przejdź przez wszystkie elementy wektora, wywołując dla każdego obiektu funkcję **f()**.
6. Powtórz ćwiczenie 5., wykorzystując klasę **PStash**.
7. Powtórz ćwiczenie 5., wykorzystując plik **Stack4.h**, zamieszczony w rozdziale 9.
8. Utwórz dynamicznie tablicę obiektów klasy **Counted** (z ćwiczenia 1.). W stosunku do otrzymanego wskaźnika zastosuj operator **delete** *nie używając nawiasu kwadratowego*. Wyjaśnij uzyskane rezultaty.
9. Utwórz obiekt klasy **Counted** (z ćwiczenia 1.) za pomocą operatora **new**, dokonaj rzutowania otrzymanego wskaźnika na typ **void***, a następnie go usuń. Wyjaśnij uzyskane rezultaty.
10. Za pomocą swojego komputera uruchom program **NewHandler.cpp** i zobacz, jakawyświetli on wartość licznika. Oblicz wielkość pamięci dostępnej dla programu.
11. Utwórz klasę posiadającą przeciążone operatory **new** i **delete**, zarówno w wersjach dla pojedynczych obiektów, jak i dla tablic. Pokaż, że obie wersje tych operatorów działają poprawnie.
12. Wymyśl test dla programu **Framis.cpp**, który pozwoli na oszacowanie, o ile szybciej działają własne wersje operatorów **new** i **delete** w porównaniu z operatorami globalnymi.
13. Zmodyfikuj klasę **NoMemory** w taki sposób, by zawierała tablicę liczb całkowitych **i**, zamiast zgłaszać wyjątek **bad_alloc**, przydzielając pamięć. W funkcji **main()** utwórz podobną do zawartej w programie **NewHandler.cpp** pętle **while**, powodującą przekroczenie dostępnej pamięci, i zobacz, co się stanie w przypadku, gdy funkcja operator **new** nie będzie sprawdzała, czy pamięć została pomyślnie przydzielona. Następnie dodaj do funkcji operator **new** test, zgłaszający wyjątek **bad_alloc**.
14. Utwórz klasę **zawierającą operator umieszczenia new**, posiadający drugi argument typu **string**. Klasa powinna zawierać statyczny wektor **vector<string>**, w którym przechowywany będzie drugi argument operatora **new**. Operator umieszczenia **new** powinien przydzielać pamięć w normalny sposób. W funkcji **main()** wywołaj przygotowany przez siebie operator umieszczenia **new**, z argumentem będącym łańcuchem, opisującym wywołania (możesz użyć do tego zadania makroinstrukcji preprocessora **_FILE_** oraz **_LINE_**).

15. Zmodyfikuj program **ArrayOperatorNew.cpp**, tworząc statyczny wektor **vector<Widget*>** i dopisując do niego adres każdego obiektu klasy **Widget**, przydzielonego za pomocą funkcji **operator new** i usuwając go, gdy jest on zwalniany za pomocą funkcji **operator delete** (aby dowiedzieć się, jak to zrobić, możesz zajrzeć do informacji dotyczących klasy **vector**, zawartych w dokumentacji standardowej biblioteki języka C++ lub w drugim tomie książki, dostępnym w Internecie). Utwórz drugą klasę o nazwie **MemoryChecker**, posiadającą destruktor, który drukuje liczbę wskaźników obiektów klasy **Widget**, zawartych w wektorze. Napisz program, zawierający pojedynczy, globalny egzemplarz obiektu klasy **MemoryChecker** i przydzielający dynamicznie oraz niszczący w funkcji **main()** szereg obiektów i tablic obiektów klasy **Widget**. Pokaż, że klasa **MemoryChecker** ujawnia występujące w programie wycieki pamięci.

Rozdział 14.

Dziedziczenie i kompozycja

Do najbardziej przekonywających cech języka C++ należy możliwość wielokrotnego wykorzystywania kodu. Jednak aby miała ona przełomowe znaczenie, musi pozwalać na więcej niż tylko na kopiowanie kodu i jego modyfikację.

To ujęcie, stosowane w języku C, nie sprawdza się zbyt dobrze. Podobnie jak wszystko w języku C++, rozwiązanie tego problemu jest związane z klasami. Kod jest wykorzystywany wielokrotnie dzięki tworzeniu nowych klas. Zamiast jednak generować je od początku, używa się w tym celu już istniejących, które zostały przez kogoś utworzone i uruchomione.

Sztuczka polega na tym, by wykorzystywać te klasy, nie naruszając istniejącego kodu. W tym rozdziale poznamy dwa sposoby osiągnięcia tego celu. Pierwszy jest dość oczywisty — wewnątrz klas istniejących tworzy się obiekty nowych. Nosi to nazwę *kompozycji* (ang. *composition*), ponieważ nowe klasy są zestawiane z obiektów tych klas, którejuż istnieją.

Drugie podejście jest bardziej wyrafinowane. Nowa klasę tworzy się jako *rodzaj* już istniejącej klasy. Pobiera się dosłownie postać istniejącej klasy, dodając do niej kod i nie modyfikując jednocześnie oryginalnej klasy. Ta magiczna czynność nazywana jest *dziedziczeniem* (ang. *inheritance*), a większość związanej z nim pracy wykonuje kompilator. Dziedziczenie jest jednym z kamieni węgielnych programowania obiektowego i pociąga za sobą dodatkowe konsekwencje, omówione w rozdziale 15.

Okazuje się, że zarówno składnia, jak i działanie kompozycji i dziedziczenia są w dużej mierze do siebie podobne (co jest logiczne — obie metody umożliwiają bowiem tworzenie nowych typów na podstawie typów już istniejących). W tym rozdziale przedstawimy mechanizmy, umożliwiające wielokrotne wykorzystywanie kodu.

Składnia kompozycji

W rzeczywistości używaliśmy kompozycji od samego początku do tworzenia klas. Klasy komponowaliśmy z obiektów typów wbudowanych (a czasami z łańcuchów). Okazuje się, że kompozycję można równie łatwo stosować w przypadku typów zdefiniowanych przez użytkownika.

Weźmy pod uwagę klasę, która mogłaby z jakiegoś powodu okazać się przydatna:

```
//: C14:Useful.h
// Klasa do ponownego wykorzystania
#ifndef USEFUL_H
#define USEFUL_H

class X {
    int i;
public:
    X() { i = 0; }
    void set(int ii) { i = ii; }
    int read() const { return i; }
    int permute() { return i - i * 47; }
};

#endif // USEFUL_H //:~
```

Dane składowe zawarte w tej klasie są prywatne, więc można zupełnie bezpiecznie osadzić obiekt klasy X w innej klasie, w charakterze obiektu publicznego, co czyni interfejs tej klasy dość prostym:

```
//: C14:Composition.cpp
// Powtórne wykorzystanie kodu za pomocą kompozycji
#include "Useful.h"

class Y {
    int i;
public:
    X x; // Osadzony obiekt
    Y() { i = 0; }
    void f(int ii) { i = ii; }
    int g() const { return i; }
};

int main() {
    Y y;
    y.f(47);
    y.x.set(37); // Dostęp do osadzonego obiektu
} //:~
```

Dostęp do funkcji składowych osadzonego obiektu (określonego również mianem *obiektu podrzędnego*) wymaga wyboru dodatkowej składowej.

Znacznie częściej można spotkać prywatne obiekty osadzone, dzięki czemu stają się one elementem wewnętrznej implementacji **klasy** (co oznacza, że implementacja ta może być dowolnie zmieniana). Funkcje tworzące publiczny interfejs nowo utworzonej klasy mogą zawierać odwołania do osadzonego obiektu, ale niekoniecznie naśladują jego interfejs:

```

//: C14:Composition2.cpp
// Prywatny obiekt osadzony
#include "Useful.h"

class Y {
    int i;
    X x; // Osadzony obiekt
public:
    Y() { i = 0; }
    void f(int ii) { i = ii; x.set(ii); }
    int g() const { return i * x.read(); }
    void permute() { x.permute(); }
};

int main() {
    Y y;
    y.f(47);
    y.permute();
} III-

```

W powyższym przykładzie funkcja `permute()` została przeniesiona do interfejsu nowej klasy, lecz pozostałe funkcje składowe klasy `X` są używane wyłącznie we wnętrzu funkcji składowych klasy `Y`.

Składnia dziedziczenia

Składnia związana z kompozycją jest dość oczywista, ale dziedziczenie wymaga zastosowania zupełnie nowej, odmiennej notacji.

Stosując dziedziczenie, oznajmia się: „nowa klasa jest podobna do tamtej, istniejącej już klasy”. Oznacza się to w kodzie programu, podając, jak zwykle, nazwę nowej klasy. Przed otwierającym nawiąsem klamrowym, rozpoczynającym ciało klasy, umieszcza się dwukropki oraz nazwę *klasy podstawowej* (lub *klas podstawowych*, oddzielonych przecinkami -- w przypadku wielokrotnego dziedziczenia). W rezultacie wszystkie dane oraz funkcje składowe klasy podstawowej znajdują się automatycznie w nowo utworzonej klasie. Ilustruje to poniższy przykład:

```

//: C14:Inheritance.cpp
// Proste dziedziczenie
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X {
    int i; // Inne niż i klasy X
public:
    Y() { i = 0; }
    int change() {
        i = permute(); // Wywołanie funkcji o innej nazwie
        return i;
    }
    void set(int ii) {

```

```

    i = ii;
    X::set(ii); // Wywołanie funkcji o tej samej nazwie
}

int main() {
    cout << "sizeof(X) = " << sizeof(X) << endl;
    cout << "sizeof(Y) = "
        << sizeof(Y) << endl;
    Y D;
    D.change();
    // Wykorzystanie funkcji interfejsu klasy X:
    D.read();
    D.permute();
    // Zdefiniowane ponownie funkcje zasłaniają
    // funkcje klasy podstawowej:
    D.set(12);
} //!:-)

```

A zatem klasa Y dziedziczy elementy klasy X, co oznacza, że klasa Y będzie posiadała wszystkie dane i funkcje składowe, zawarte w klasie X. W rzeczywistości klasa Y zawiera obiekt podrzędny klasy X — zupełnie taki sam, jaki powstałby w rezultacie utworzenia w klasie Y składowej, będącej obiektem klasy X, a nie w wyniku dziedziczenia. Zarówno obiekty składowe, jak i pamięć zajmowana przez klasę podstawową są określane mianem obiektów podrzędnych.

Wszystkie prywatne składowe klasy X pozostają składowymi prywatnymi w klasie Y — czyli fakt dziedziczenia przez klasę Y po klasie X nie oznacza wcale, że klasa Y może naruszyć mechanizmy ochrony klasy X. Prywatne elementy składowe klasy X znajdują się nadal na swoim miejscu, zajmując pamięć — nie można się tylko bezpośrednio do nich odwołać.

Jak widać w funkcji `main()`, dane składowe klasy Y zostały połączone z danymi klasy X, co można wywnioskować z faktu, że wartość zwracana przez wyrażenie `sizeof(Y)` jest dwukrotnie większa niż wartość zwracana przez `sizeof(X)`.

Nazwę klasy podstawowej poprzedzono słowem kluczowym **public**. Podczas dziedziczenia wszystko jest domyślnie prywatne. Gdyby nazwa klasy podstawowej nie została poprzedzona słowem kluczowym **public**, oznaczałoby to, że wszystkie składowe publiczne klasy podstawowej stałyby się w klasie pochodnej składowymi prywatnymi. Prawie nigdy nie jest to naszym *zamiarem*¹ — zazwyczaj pożądane jest pozostawienie wszystkich składowych publicznych klasy podstawowej składowymi publicznymi w klasie pochodnej. Uzyskuje się to, używając podczas dziedziczenia słowa kluczowego **public**.

W funkcji `change()` wywoływaną jest funkcja klasy podstawowej `permute()`. Klasa pochodna ma bezpośredni dostęp do wszystkich publicznych funkcji klasy podstawowej.

Funkcja `set()`, znajdująca się w klasie pochodnej, zmienia *definicję* funkcji `set()`, zawartej w klasie podstawowej. Oznacza to, że jeżeli w stosunku do obiektów klasy V

¹ Kompilator języka Java nie pozwala na obniżenie poziomu dostępu do składowej podczas dziedziczenia.

zostaną wywoływane funkcje `read()` oraz `permute()`, to wywoływanie zostaną wersje tych funkcji, zawarte w klasie podstawowej. Jeżeli jednak w stosunku do obiektu klasy `Y` używa się funkcji `set()`, to zostanie wywołana jej przedefiniowana wersja. Wynika z tego, że jeżeli nie odpowiada nam wersja jakiejś funkcji, pozyskanej za pomocą dziedziczenia, to możemy zmienić jej działanie (można również dodać do klasy zupełnie nowe funkcje, jak w przypadku funkcji `change()`).

Jednakże po zmianie definicji może zaistnieć potrzeba wywołania jazdy wersji, zawartej w klasie podstawowej. Jeżeli wewnątrz funkcji `set()` wywoła się po prostu funkcję `set()`, to wywołana zostanie rekurencyjnie lokalna wersja funkcji. W celu wywołania wersji funkcji, zawartej w klasie podstawowej, należy jawnie wskazać klasę podstawową, używając do tego celu operatora zasięgu.

Listy inicjalizatorów konstruktora

Wiadomo już, jak ważne jest w języku C++ zagwarantowanie prawidłowej inicjalizacji — w takim samym stopniu dotyczy to również kompozycji oraz dziedziczenia. Podczas tworzenia obiektu kompilator zapewnia wywołanie konstruktorów dla wszystkich jego obiektów podrzędnych. W przedstawionych do tej pory przykładach wszystkie obiekty podrzędne posiadały domyślne konstruktory, wywoływanie przez kompilator. Co się jednak dzieje w przypadku, gdy obiekty składowe nie mają domyślnych konstruktów, albo gdy chcemy zmienić domyślny argument któregoś z konstruktów? Stanowi to problem, ponieważ konstruktor nowej klasy nie ma dostępu do prywatnych danych składowych obiektu podrzędnego, nie może więc ich bezpośrednio zainicjować.

Rozwiążanie jest proste — należy wywołać konstruktor obiektu podrzędnego. Język C++ udostępnia w tym celu specjalną konstrukcję, nazywaną *listą inicjalizatorów konstruktora* (ang. *constructor initializer list*). Postać tej listy stanowi odzwierciedlenie składni dziedziczenia. W przypadku dziedziczenia nazwa klasy podstawowej umieszczana jest po dwukropku, a przed otwierającym nawiasem klamrowym, rozpoczynającym ciało klasy. Na liście inicjalizatorów konstruktora wywołania konstruktorów obiektów podrzędnych umieszczane są natomiast za listą argumentów konstruktora i dwukropkiem, a przed otwierającym nawiasem klamrowym, rozpoczynającym ciało funkcji. W przypadku klasy `MojTyp`, dziedziczącej z klasy `Bar`, może to przybrać następującą postać:

```
MojTyp::MojTyp(int i) : Bar(i) { // ... }
```

pod warunkiem, że klasa `Bar` posiada konstruktor, pobierający pojedynczy argument typu całkowitego.

Inicjalizacja obiektów składowych

Okazuje się, że takiej samej składni używa się do inicjalizacji obiektów składowych w przypadku kompozycji. Zamiast nazw klas podaje się jednak wówczas nazwy

obiektów. Jeśli na liście inicjatorów zostanie umieszczona większa liczba wywołań konstruktorów, to oddziela się je przecinkami:

```
MojTyp2::MojTyp2(int i) : Bar(i), m(i+1) { // ...}
```

Jest to początek konstruktora klasy **MojTyp2**, dziedziczącej po klasie Bar i zawierającej obiekt składowy o nazwie m. Zwrót uwagę na to, że o ile na liście inicjatorów konstruktora widoczny jest typ klasy podstawowej, o tyle w przypadku obiektu składowego widoczny jest już tylko jego identyfikator.

Typy wbudowane znajdujące się na liście inicjatorów

Lista inicjatorów konstruktora pozwala najawne wywołanie konstruktów obiektów składowych. W rzeczywistości nie ma żadnego innego sposobu wywołania tych konstruktów. Idea polega na tym, że wszystkie konstruktory muszą zostać wywołane przed wejściem do ciała konstruktora nowej klasy. Dzięki temu wszelkie odwołania do funkcji składowych obiektów podlegających zawsze będą dotyczyć zainicjowanych obiektów. Niemożliwe jest zaistnienie sytuacji, w której dotarlibyśmy do nawiasu klamrowego, otwierającego konstruktor, a nie zostałyby jeszcze wywołane *jakiekolwiek* konstruktory wszystkich obiektów **składowych** i wszystkich klas podstawowych — choćby miały to być niejawne wywołania ich domyślnych konstruktów. To kolejne potwierdzenie gwarancji, udzielanej przez język C++, że żaden obiekt (ani jego część) nie przekroczy „bramki startowej”, zanim nie zostanie wywołany jego konstruktor.

Idea polegająca na tym, że obiekty składowe są inicjalizowane w momencie osiągnięcia otwierającego nawiasu klamrowego, stanowi również przydatną wskazówkę programistyczną. Po natrafieniu na klamrę otwierającą można założyć, że wszystkie obiekty podlegające zostały już prawidłowo zainicjowane, i skupić się na specyficznych zadaniach, które ma do zrealizowania konstruktor. Jest jednak pewien kłopot: co z obiektami składowymi wbudowanymi typów, które *nie posiadają* konstruktów?

Dla zachowania spójności składni dozwolone jest traktowanie typów wbudowanych w taki sposób, jakby posiadały pojedynczy konstruktor, pobierający jeden argument — zmienną identycznego typu jak ta, która jest inicjalizowana. Dzięki temu można napisać:

```
//: C14:PseudoConstructor.cpp
class X {
    int i;
    float f;
    char c;
    char* s;
public:
    X() : i(7), f(1.4), c('x'), s("czesc") {}
};

int main() {
    X x;
    int i(100); // Zastosowany w zwykłej definicji
    int* ip = new int(47);
} //:-
```

Działanie tych „wywołań pseudokonstruktorów” polega na dokonaniu zwykłych przypisań. To wygodna technika, stanowiąca zarazem przykład dobrego stylu programowania, więc często się jej używa.

Składnię pseudokonstruktora można wykorzystać nawet podczas tworzenia zmiennej wbudowanego typu, nie należącej do klasy:

```
int i(100);
int* ip = new int(47);
```

Powoduje to, że zmienne wbudowanych typów funkcjonują w sposób nieco bardziej przypominający obiekty. Należy jednak pamiętać, że nie sąto prawdziwe konstruktory. W szczególności, jeżeli nie dokona się jawnego wywołania pseudokonstruktora, to nie zostanie dokonana żadna inicjalizacja.

Łączanie kompozycji i dziedziczenia

Oczywiście, kompozycji można używać łącznie z dziedziczeniem. Poniższy przykład prezentuje tworzenie bardziej skomplikowanych klas, podczas którego wykorzystano obie te techniki:

```
//: C14:Combined.cpp
// Dziedziczenie i kompozycja

class A {
    int i;
public:
    A(int ii) : i(ii) {}
    ~A() {}
    void f() const {}
};

class B {
    int i;
public:
    B(int ii) : i(ii) {}
    ~B() {}
    void f() const {}
};

class C : public B {
    A a;
public:
    C(int ii) : B(ii), a(ii) {}
    ~C() // Wywołuje ~A() i ~B()
    void f() const { // Zmiana definicji
        a.f();
        B::f();
    }
};

int main() {
    C c(47);
} //:-)
```

Klasa C dziedziczy po klasie B i posiada obiekt składowy („jest skomponowana z”) klasy A. Jak widać, lista inicjatorów konstruktora zawiera zarówno wywołanie konstruktora klasy podstawowej, jak i konstruktora obiektu składowego.

Funkcja `C::f()` zmienia definicję funkcji `B::f()`, którą dziedziczy, a także wywołuje wersję tej funkcji, zdefiniowaną w klasie podstawowej. Ponadto wywołuje funkcję `a.f()`. Zwróć uwagę na to, że jedyny przypadek dotyczący zmiany definicji ma miejsce podczas dziedziczenia — w przypadku obiektu składowego można manipulować jedyniego interfejsem publicznym, a nie wolno zmienić definicje. Oprócz tego wywołanie funkcji `f()` dla obiektu klasy C nie spowoduje wywołania funkcji `a.f()`, gdyby nie była zdefiniowana funkcja `C::f()`. Mogłoby natomiast spowodować wywołanie funkcji `B::f()`.

Automatyczne wywołania destruktów

Mimo że często zachodzi konieczność jawnego wywoływanego konstruktów poprzez umieszczenie ich na liście inicjatorów, nigdy nie ma potrzeby jawnego wywoływanego destruktów. Każda klasa posiada bowiem tylko jeden destruktor i nie pobiera on żadnych argumentów. Jednakże kompilator nadal zapewnia, że wywołane zostaną wszystkie destruktory znajdujące się w całej hierarchii klas — poczynając od destruktora ostatniej wyprowadzonej klasy i posuwając się wstecz, aż do klasy najbardziej podstawowej (głównej).

Warto podkreślić, że konstruktory i destruktory mają nieco niezwykłe cechy, ponieważ zawsze wywoływany jest konstruktor i destruktor każdej klasy znajdującej się w hierarchii. Tymczasem w przypadku zwykłych funkcji składowych wywoływana jest tylko jedna jej wersja, a nie wszystkie, zdefiniowane w klasach podstawowych. Ponadto jeżeli zamierza się wywołać zdefiniowaną w klasie podstawowej wersję normalnej funkcji składowej, która została zasłonięta, to trzeba zrobić to jawnym sposobem.

Kolejność wywoływania konstruktów i destruktów

Interesującą jest, w jakiej kolejności wywoływanie są konstruktory i destruktory, jeśli obiekt posiada wiele obiektów podległych. Poniższy przykład pokazuje, jak się to odbywa:

```
//: C14:Order.cpp
// Kolejność wywoływania konstruktów i destruktów
#include <fstream>
using namespace std;
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
    ID(int) { out << #ID " konstruktor\n"; } \
    -ID() { out << #ID " destruktor\n"; } \
};

CLASS(Order);
CLASS(Item);
CLASS(LineItem);
CLASS(Invoice);
```

```
CLASS(Base);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);

class Derived1 : public Base {
    Member1 m1;
    Member2 m2;
public:
    Derived1(int) : m2(1), m1(2), Base(3) {
        cout << "Derived1 konstruktor\n";
    }
    ~Derived1() {
        cout << "Derived1 destruktor\n";
    }
};

class Derived2 : public Derived1 {
    Member3 m3;
    Member4 m4;
public:
    Derived2() : m3(1), Derived1(2), m4(3) {
        cout << "Derived2 konstruktor\n";
    }
    ~Derived2() {
        cout << "Derived2 destruktor\n";
    }
};

int main() {
    Derived2 d2;
}
```

Najpierw tworzony jest obiekt klasy **ofstream**, umożliwiający zapisywanie danych wyjściowych w pliku. Następnie, aby uniknąć żmudnego wpisywania kodu i zadeemonstrować wykorzystanie techniki makroinstrukcji (zostanie ona zastąpiona w rozdziale 16. znacznie doskonalszą techniką), tworzona jest makroinstrukcja służąca do utworzenia klas, które następnie **zostały** użyte w dziedziczeniu i kompozycji. Wszystkie konstruktory i destruktory zapisują w pliku śledzenia informacje o tym, że zostały wywołane. Zwróć uwagę na to, że konstruktory nie są konstruktorami domyślnymi — każdy z nich pobiera jako argument liczbę całkowitą. Argumenty te nie posiadają identyfikatorów — jedynym powodem ich stosowania jest wymuszenie jawnego wywołania konstruktorów na liście inicjatorów (brak identyfikatorów zapobiega zgłoszaniu ostrzeżeń przez kompilator).

Wyniki pracy programu są następujące:

```
Base konstruktor
Member1 konstruktor
Member2 konstruktor
Derived1 konstruktor
Member3 konstruktor
Member4 konstruktor
Derived2 konstruktor
Derived2 destruktor
```

```

Member4 destruktor
Member3 destruktor
Derived1destruktor
Member2 destruktor
Member1 destruktor
Base1 destruktor

```

Konstrukcja rozpoczyna się zatem na najbardziej podstawowym poziomie hierarchii klas. Na każdym poziomie najpierw wywoływany jest konstruktor klasy podstawowej, a później konstruktory obiektów składowych. Destruktory są wywoływanie w odwrotnej kolejności niż konstruktory — **jest** to istotne z uwagi na potencjalne zależności (w konstruktorach i destruktorkach klas pochodnych musimy mieć prawo założyć, że obiekty podrzędne klas podstawowych są dostępne i zostały już skonstruowane — lub nie zostały jeszcze zniszczone).

Interesujące jest **również**, że w przypadku obiektów składowych kolejność wywołań konstruktorów nie jest wcale związana z kolejnością ich wywołań na liście inicjatorów konstruktora. Kolejność ta jest wyznaczona przez porządek, w jakim obiekty składowe zostały zadeklarowane w klasie. Gdyby istniała możliwość zmiany kolejności wywołań konstruktorów za pomocą listy inicjatorów konstruktora, to można by w dwóch różnych konstruktorach utworzyć dwie odmienne sekwencje wywołań konstruktorów obiektów składowych. Nieszczęsny destruktor nie wiedziałby wówczas, w jaki sposób prawidłowo wyznaczyć odwrotną kolejność wywołań destruktorków, co mogłoby doprowadzić do problemów związanych z zależnościami.

Ukrywanie nazw

Jeżeli utworzymy za pomocą dziedziczenia klasę i dostarczymy nową definicję dla jednej z zawartych w niej funkcji składowych, to zaistnieje jedna z dwu sytuacji. Pierwsza polega na tym, że definicja zawarta w klasie pochodnej będzie posiadała taką samą sygnaturę i typ zwracanej wartości, jak definicja znajdująca się w klasie podstawowej. Jest to **nazywane przedefiniowaniem**(ang. *redefining*), w przypadku zwykłych funkcji składowych, lub **zasłanianiem** (ang. *overriding*), w przypadku gdy funkcja składowa klasy podstawowej jest funkcją wirtualną (funkcje wirtualne stanowią normalny przypadek i zostaną opisane w rozdziale 15.). Co jednak dzieje się w sytuacji, gdy w klasie pochodnej zmienimy listę argumentów funkcji albo zwracaną przez nią wartość? Poniżej znajduje się stosowny przykład:

```

//: C14:NameHiding.cpp
// Ukrywanie przeciążonych nazw podczas dziedziczenia
#include <iostream>
#include<string>
using namespace std;

class Base {
public:
    int f() const {
        cout << "Base::f()\n";
        return 1;
    }
}

```

```

int f(string) const { return 1; }
void g() {}

};

class Derived1 : public Base {
public:
    void g() const {}
};

class Derived2 : public Base {
public:
    // Przeddefiniowanie:
    int f() const {
        cout << "Derived2::f()\n";
        return 2;
    }
};

class Derived3 : public Base {
public:
    // Zmiana zwracanego typu:
    void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base {
public:
    // Zmiana listy argumentów:
    int f(int) const {
        cout << "Derived4::f()\n";
        return 4;
    }
};

int main() {
    string s("witam");
    Derived1 dl;
    int x = dl.f();
    dl.f(s);
    Derived2 d2;
    x = d2.f();
    //! d2.f(); // Wersja z łańcuchem jest ukryta
    Derived3 d3;
    //! x = d3.f(); // Wersja zwracająca wartość int jest ukryta
    Derived4 d4;
    //! x = d4.f(); // Wersja f() jest ukryta
    x = d4.f(!);
}
}

```

W klasie **Base** widoczna jest przeciążona funkcja **f()**. Klasa **Derived1** nie wprowadza żadnych modyfikacji do funkcji **f()**, zmienia natomiast definicję funkcji **g()**. Jak widać w funkcji **main()**, w klasie **Derived1** dostępne są obie przeciążone funkcje **f()**. Jednakże w klasie **Derived2** zmieniono definicję tylkojednej z przeciążonych funkcji **f()**, a w rezultacie również druga wersja tej funkcji staje się niewidoczna. W klasie **Derived3** zmiana typu zwracanej wartości powoduje ukrycie obu wersji funkcji, zawartych w klasie podstawowej, natomiast klasa **Derived4** pokazuje, że zmiana listy argumentów funkcji również powoduje ukrycie obu jej wersji, znajdujących się w klasie.

podstawowej. Można ogólnie stwierdzić, że ilekroć przedefiniowuje się funkcję, której nazwą jest przeciążona w klasie podstawowej, to wszystkie pozostałe wersje tej funkcji są automatycznie ukrywane w nowej klasie. Jak przekonamy się w rozdziale 15., dodanie słowa kluczowego **virtual** pociąga za sobą dalsze konsekwencje, dotyczące przeciążania funkcji.

W przypadku zmiany interfejsu klasy podstawowej, polegającej na zmodyfikowaniu sygnatury lub wartości zwracanej przez jej funkcję składową, klasa jest wykorzystywana w nieco odmienny sposób niż ten, z myślą o którym utworzono dziedziczenie. Nie oznacza to wcale, że podąża się w złym kierunku; głównym zastosowaniem dziedziczenia jest bowiem wspieranie *polimorfizmu*, natomiast zmiana sygnatury funkcji lub typu zwracanej przez nią wartości jest w rzeczywistości modyfikacją interfejsu klasy podstawowej. Jeżeli jest to zgodne z naszymi zamierzeniami, używamy dziedziczenia przede wszystkim w celu powtórnego wykorzystania kodu, nie zaś utrzymania jednolitego interfejsu klasy podstawowej (będącego zasadniczym przejawem polimorfizmu). Zazwyczaj posługiwanie się dziedziczeniem w taki sposób oznacza użycie klasy o ogólnym przeznaczeniu i dostosowanie jej do konkretnych potrzeb — co jest zwykle, chociaż nie zawsze, uważane za domenę kompozycji. Weźmy na przykład pod uwagę klasę **Stack**, opisaną w 9. rozdziale książki. Jeden z problemów z nią związanych polega na tym, że każdym razem, gdy pobiera się wskaźnik z kontenera, trzeba dokonać jego rzutowania. Jest to nie tylko męczące, ale również niebezpieczne — wskaźnik taki można bowiem rzutować na dowolnie wybrany typ.

Podejście, które na pierwszy rzut oka może wydawać się lepsze, polega na wykorzystaniu dziedziczenia do utworzenia specjalizowanej wersji ogólnej klasy **Stack**. Poniżej zamieszczono przykład, wykorzystujący postać tej klasy, przedstawioną w rozdziale 9.:

```
//: C14:InheritStack.cpp
// Specjalizacja klasy Stack
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack : public Stack {
public:
    void push(string* str) {
        Stack::push(str);
    }
    string* peek() const {
        return (string*)Stack::peek();
    }
    string* pop() {
        return (string*)Stack::pop();
    }
    ~StringStack() {
        string* top = pop();
        while(top) {
            delete top;
            top = pop();
        }
    }
}
```

```

};

int main() {
    ifstream in("InheritStack.cpp");
    assure(in, "InheritStack.cpp");
    string line;
    StringStack textlines;
    while(getline(in, line))
        textlines.push(new string(line));
    string* s;
    while((s = textlines.pop()) != 0) { // Brak rzutowania!
        cout << *s << endl;
        delete s;
    }
} // :~
```

Ponieważ wszystkie funkcje składowe, zawarte w pliku **Stack4.h**, są funkcjami inline, nie ma potrzeby łączenia programu z żadnym dodatkowym modułem.

Klasa **StringStack** stanowi wersję klasy **Stack**, wyspecjalizowaną w taki sposób, by funkcja **push()** mogła pobierać wyłącznie wskaźniki do łańcuchów (**string**). Po przednio klasa **Stack** pobierała wskaźniki typu **void***, co powodowało, że użytkownik nie dysponował żadnym mechanizmem kontroli typów, który sprawdzałby, czy do kontenera zostały dodane odpowiednie wskaźniki. Ponadto funkcje **peek()** i **pop()**, zamiast wskaźników typu **void***, zwracają obecnie wskaźniki do łańcuchów, dzięki czemu można ich używać bez konieczności rzutowania.

Dość zdumiewające jest natomiast, że to dodatkowe zabezpieczenie, polegające na kontroli typów w funkcjach **push()**, **peek()** i **pop()**, zupełnie nic nie kosztuje! Kompilator otrzymuje dodatkową informację, wykorzystywaną podczas komplikacji, ale funkcje są funkcjami inline, i nie jest w nich przypadku generowany żaden dodatkowy kod.

Odgrywa tu rolę ukrywanie nazw, ponieważ w szczególności funkcja **push()**, zdefiniowana w klasie pochodnej, posiada inną sygnaturę niż jej definicja, znajdująca się w klasie podstawowej — funkcje te różnią się listą argumentów. Gdyby w obrębie klasy istniały dwie wersje funkcji **push()**, to mielibyśmy do czynienia z przeciążeniem, ale w tym przypadku przeciążenie *niejest* naszym celem, ponieważ nadal pozwalałoby na przekazywanie funkcji **push()** dowolnego wskaźnika jako wskaźnika typu **void***. Na szczęście, język C++ ukrywa wersję, zdefiniowaną jako **push(void*)** w klasie podstawowej, na rzecz nowej wersji funkcji, zdefiniowanej w klasie pochodnej, pozwalając w ten sposób na umieszczanie w kontenerze (będącym obiektem klasy **StringStack**) jedynie wskaźników do łańcuchów.

Ponieważ możemy już zagwarantować dokładną znajomość typu obiektów znajdujących się w kontenerze, destruktor klasy działa prawidłowo, co rozwiązuje problem prawa własności — a przynajmniej jeden z jego aspektów. Następnie umieszczając za pomocą funkcji **push()** wskaźnik do łańcucha w kontenerze, będącym obiektem klasy **StringStack**, przekazujemy jej (zgodnie ze znaczeniem klasy **StringStack**) również prawo własności tego wskaźnika. Gdy pobieramy wskaźnik za pomocą funkcji **pop()**, otrzymujemy nie tylko wskaźnik, ale również prawo do jego posiadania. Wszelkie wskaźniki, pozostawione w kontenerze **StringStack** do chwili wywołania

destruktora, są usuwane przez ten destraktor. A ponieważ są to zawsze wskaźniki do łańcuchów, instrukcja `delete` jest stosowana w odniesieniu do wskaźników łańcuchów, a nie do wskaźników typu `void*`, dzięki czemu destrukcja jest poprawnie wykonywana i wszystko działa prawidłowo.

Rozwiążanie to ma jednak pewną wadę — działa *wyłącznie* ze wskaźnikami do łańcuchów. Gdybyśmy potrzebowali klasy `Stack`, działającej z jakimś innym rodzajem obiektów, musielibyśmy napisać jej nową wersję, funkcjonującą tylko z tym właśnie rodzajem. Szybko stałoby się to uciążliwe — problem ten można ostatecznie rozwiązać za pomocą szablonów (zob. rozdział 16).

Możemy uczynić jeszcze jedno spostrzeżenie, dotyczącego powyższego przykładu. W procesie dziedziczenia uległ zmianie interfejs klasy `Stack`. Jeżeli zmienił się interfejs, oznacza to, że klasa `StringStack` nie jest w rzeczywistości rodzajem klasy `Stack`. Z tej przyczyny nigdy nie będzie można jej poprawnie użyć zamiast klasy `Stack`. Powoduje to, że zastosowanie dziedziczenia jest w tym przypadku dyskusyjne — jeżeli nie utworzyliśmy klasy `StringStack`, która jest rodzajem klasy `Stack`, to po co używaliśmy dziedziczenia? Bardziej poprawna wersja klasy `StringStack` zostanie przedstawiona w dalszej części rozdziału.

Funkcje, które nie są automatycznie dziedziczone

Nie wszystkie funkcje znajdujące się w klasie podstawowej są automatycznie dziedziczone przez klasę pochodną. Konstruktory i destruktory są odpowiedzialne za tworzenie i niszczenie obiektów, ale „wiedzą” one jedynie, jakie działania należy wykonać z obiektami należącymi do ich własnej klasy. Zachodzi zatem konieczność wywołania wszystkich konstruktorów i destruktatorów klas, znajdujących się niżej od nich w hierarchii. Tak więc konstruktory i destruktory nie podlegają dziedziczeniu i muszą być tworzone oddzielnie dla każdej klasy pochodnej.

Ponadto dziedziczeniu nie podlega `operator=`, ponieważ wykonuje on działania zbliżone do konstruktora. Innymi słowy — to, że wiemy, w jaki sposób wszystkim składowym obiektu, znajdującemu się po lewej stronie znaku `=`, przypisać wartości pochodzące z obiektu po jego prawej stronie, nie oznacza wcale, że operacja przypisania będzie miała w klasie pochodnej nadal to samo znaczenie.

Funkcje te, zamiast podlegać dziedziczeniu, są generowane przez kompilator — o ile nie przygotuje się ich własnych wersji (w przypadku konstruktorów nie można utworzyć żadnego konstruktora, jeżeli kompilator ma utworzyć domyślny konstruktor i konstruktor kopiący). Kwestia ta została krótko opisana w rozdziale 6. Wygenerowane konstruktory wykorzystują inicjalizację za pośrednictwem elementów składowych, natomiast utworzony przez kompilator `operator=` stosuje przypisanie za pośrednictwem elementów składowych. Poniższy przykład zawiera funkcje wygenerowane przez kompilator.

```
//: C14:SynthesizedFunctions.cpp
// Funkcje generowane przez kompilator
#include <iostream>
using namespace std;

class GameBoard {
public:
    GameBoard() { cout << "GameBoard()\n"; }
    GameBoard(const GameBoard&) {
        cout << "GameBoard(const GameBoard&)\n";
    }
    GameBoard& operator=(const GameBoard&) {
        cout << "GameBoard::operator=(())\n";
        return *this;
    }
    ~GameBoard() { cout << "~GameBoard()\n"; }
};

class Game {
    GameBoard gb; // Kompozycja
public:
    // Wywoływany domyślny konstruktor klasy GameBoard:
    Game() { cout << "Game()\n"; }
    // Trzeba jawnie wywołać konstruktor kopiujący
    // klasy GameBoard, gdyż w przeciwnym przypadku
    // zamiast niego zostanie wywołany
    // automatycznie domyślny konstruktor:
    Game(const Game& g) : gb(g.gb) {
        cout << "Game(const Game&)\n";
    }
    Game(int) { cout << "Game(int)\n"; }
    Game& operator=(const Game& g) {
        // Trzeba jawnie wywołać operator przypisania
        // klasy GameBoard, gdyż w przeciwnym przypadku
        // przypisywanie go nie będzie w ogóle
        // realizowane!
        gb = g.gb;
        cout << "Game::operator=(())\n";
        return *this;
    }
    class Other {}; // Zagnieżdżona klasa
    // Automatyczna konwersja typu:
    operator Other() const {
        cout << "Game::operator Other()\n";
        return Other();
    }
    ~Game() { cout << "~Game()\n"; }
};

class Chess : public Game {

void f(Game::Other) {}

class Checkers : public Game {
public:
    // Wywoływany domyślny konstruktor klasy podstawowej:
    Checkers() { cout << "Checkers()\n"; }
```

```

// Trzeba jawnie wywołać konstruktor kopiący
// klasy podstawowej, gdyż w przeciwnym przypadku
// zostanie zamiast niego wywołany konstruktor domyślny:
Checkers(const Checkers& c) : Game(c) {
    cout << "Checkers(const Checkers& c)\n";
}

Checkers& operator=(const Checkers& c) {
    // Trzeba jawnie wywołać wersje operatora=(),
    // znajdującej się w klasie podstawowej, gdyż
    // w przeciwnym przypadku w klasie podstawowej
    // nie zostanie zrealizowana operacja przypisania:
    Game::operator=(c);
    cout << "Checkers::operator=(c)\n";
    return *this;
};

};

int main() {
    Chess d1; // Domyślny konstruktor
    Chess d2(d1); // Konstruktor kopiący
    //! Chess d3(l); // Błąd - nie ma konstruktora z argumentem int
    d1 = d2; // Wygenerowany operator=
    f(d1); // Konwersja typów JEST dziedziczona
    Game::Other go;
    //! d1 = go; // operator= nie jest generowany
    // dla różnych typów
    Checkers c1, c2(c1);
    c1 = c2;
} III-

```

Konstruktory oraz operatory przypisania klas **GameBoard** oraz **Game** przedstawiają się, dzięki czemu można zobaczyć, kiedy zostały one użyte przez kompilator. Oprócz tego **operator Other()** dokonuje automatycznej konwersji typu obiektu klasy **Game** na obiekt zagnieżdżonej w niej klasy **Other**. Klasa **Chess** została utworzona z klasą **Game** za pomocą dziedziczenia i nie tworzy żadnych nowych funkcji (aby zobaczyć, w jaki sposób reaguje w takim przypadku kompilator). Funkcja **f()** pobiera obiekt klasy **Other**, co umożliwia przetestowanie funkcji automatycznej konwersji typu.

W funkcji **main()** wywoływane są: domyślny konstruktor i konstruktor kopiący klasy **Chess**. Wersje tych konstruktorów, zawarte w klasie **Game**, są wywoływanie w ramach hierarchii wywołań konstruktorów. Mimo że przypomina to dziedziczenie, nowe wersje konstruktorów są w rzeczywistości generowane przez kompilator. Jak się można było spodziewać, żadne konstruktory zawierające argumenty nie zostały wygenerowane automatycznie, ponieważ wymagałoby to od kompilatora zbyt dużej domyślności.

W klasie **Chess** został również wygenerowany, w postaci nowej funkcji, **operator=**. Odbyło się to z wykorzystaniem przypisania za pośrednictwem elementów składowych (tak więc wywoływaną jest jego wersja zawarta w klasie podstawowej), ponieważ funkcja tego operatora ta nie została jawnie utworzona w nowej klasie. Oczywiście, kompilator wygenerował również automatycznie destruktor klasy **Chess**.

Z uwagi na wszystkie **reguły**, dotyczące przepisywania funkcji obsługujących tworzenie obiektów, może na początku wydawać się nieco dziwne, że dziedziczony jest operator automatycznej konwersji typów. Nie jest to jednak pozbawione podstaw — jeżeli klasa **Game** posiada wystarczającą liczbę elementów, niezbędnych do utworzenia obiektu klasy **Other**, to elementy te są również zawarte w każdej klasie wprowadzonej z klasy **Game** i konwersja taka jest nadal poprawna (mimo że w rzeczywistości należałyby ją przedefiniować).

Funkcja **operator=** jest generowana *wyłącznie* w celu przypisywania obiektów tego samego typu. Jeżeli zamierzamy przypisywać obiektom jednego typu obiekty innego typu, to zawsze musimy utworzyć funkcję **operator=** samodzielnie.

Przyglądając się bliżej klasie **Game**, można zauważyc, że konstruktor kopiący oraz operator przypisania zawierają jawne wywołania — odpowiednio konstruktora kopującego oraz operatora przypisania obiektu składowego. Jest to zazwyczaj konieczne, ponieważ w przeciwnym razie zamiast konstruktora kopującego zostałby użyty domyślny konstruktor obiektu składowego, a w przypadku operatora przypisania nie nastąpiłoby w ogóle przypisanie wartości obiektowi składowemu!

Przyjrzymy się wreszcie klasie **Checkers**, zawierającej jawne definicje konstruktora domyślnego, konstruktora kopującego oraz operatora przypisania. W przypadku konstruktora domyślnego wywoływany jest automatycznie konstruktor domyślny klasy podstawowej, i zwykle o to nam chodzi. Ważne jest jednak, że jeżeli zdecydujemy się na napisanie własnego konstruktora kopującego oraz operatora przypisania, to kompilator założy, że wiemy, co robimy, i *nie wywoła* automatycznie ich wersji zawartych w klasie podstawowej, co uczyniłby w przypadku wygenerowanych przez siebie wersji tych funkcji. Jeżeli chcemy, aby wywołane zostały ich wersje zawarte w klasie podstawowej (a zazwyczaj tak właśnie jest), to musimy wywołać je sami. W konstruktorze kopującym klasę **Checkers** wywołanie to jest widoczne na liście inicjatorów konstruktora:

```
Checkers(const Checkers& c) : Game(c) {
```

W operatorze przypisania klasy **Checkers** wywołanie funkcji klasy podstawowej znajduje się w pierwszym wierszu ciała funkcji:

```
Game::operator=(c);
```

Wywołania te powinny stanowić dla ciebie wzorzec, wykorzystywany za każdym razem, gdy tworzysz jakąś klasę, używając do tego dziedziczenia.

Dziedziczenie a statyczne funkcje składowe

Statyczne funkcje **składowe** działają tak samo, jak funkcje składowe niebędące funkcjami statycznymi:

1. Są one dziedziczone w klasach pochodnych.
2. W przypadku **przedefiniowania** statycznej funkcji składowej wszystkie pozostałe przeciążone funkcje, zawarte w klasie podstawowej, są ukrywane.

3. W przypadku zmiany sygnatury funkcji, zawartej w klasie podstawowej, ukrywane są wszystkie funkcje o tej nazwie, znajdujące się w klasie podstawowej (*jest* to właściwie wariant poprzedniego punktu).

Statyczne funkcje składowe nie mogą być jednak funkcjami wirtualnymi (temat ten zostanie szczegółowo opisany w rozdziale 15.).

Wybór między kompozycją a dziedziczeniem

Zarówno kompozycja, jak i dziedziczenie powodują utworzenie w nowej klasie obiektów podległych. W obu przypadkach do skonstruowania obiektów podległych wykorzystywana jest lista inicjatorów konstruktora. Być może zastanawiasz się, jak jest pomiędzy nimi różnica i kiedy wybrać daną metodę.

Na ogół kompozycja jest wykorzystywana wówczas, gdy zamierzamy zawrzeć w nowej klasie właściwości klasy, która już istnieje, ale nie jej interfejs. Oznacza to, że obiekt jest osadzany w nowej klasie po to, by wykorzystać jego własności. Jednakże użytkownik nowo utworzonej klasy widzi interfejs tej klasy, a nie klasy oryginalnej. Robi się to zazwyczaj, osadzając w nowej klasie obiekty istniejących już klas jako składowe prywatne.

Czasami jednak jest uzasadnione, by użytkownik klasy miał bezpośredni dostęp do komponentów nowo utworzonej klasy, to znaczy, aby składowe te były publiczne. Obiekty składowe posiadają własną kontrolę dostępu, więc można to bezpiecznie wykonać. Ponadto gdy użytkownik widzi, że klasa została utworzona przez połączenie ze sobą grupy elementów, łatwiej mu zrozumieć jej interfejs. Dobrym tego przykładem jest klasa Car (samochód):

```
//: C14:Car.cpp
// Publiczna kompozycja

class Engine {
public:
    void start() const {}
    void rev() const {}
    void stop() const {}
};

class Wheel {
public:
    void inflate(int psi) const {}
};

class Window {
public:
    void rollup() const {}
    void rolldown() const {}
};
```

```

class Door {
public:
    Window window;
    void open() const {}
    void close() const {}
};

class Car {
public:
    Engine engine;
    Wheel wheel[4];
    Door left, right; // Dwoje drzwi
};

int main() {
    Car car;
    car.left.window.rollup();
    car.wheel[0].inflate(72);
} //:-

```

Z uwagi na to, że kompozycja klasy **Car** stanowi element analizy problemu (a nie tylko część jego wewnętrznego projektu), użycie **składowych** publicznymi pomaga klientowi-programiście zrozumieć, w jaki sposób należy używać tej klasy, a zarazem wymaga od twórcy klasy tworzenia mniej **złożonego kodu**.

Po chwili zastanowienia przyznasz zapewne, że nie miałooby sensu użycie do budowy klasy **Car** obiektu „pojazd” — samochód nie zawiera bowiem pojazdu, **on jest** pojazdem. Relacja **typu jest** jest wyrażana za pomocą dziedziczenia, natomiast relacja **posiada** — za pomocą kompozycji.

Tworzenie podtypów

Założmy, że zamierzamy utworzyć rodzaj obiektu klasy **ifstream**, który nie tylko będzie otwierał plik, ale również pamiętał jego nazwę. Można w tym celu wykorzystać kompozycję, zamkając obiekty klas **ifstream** oraz **string** wewnętrz nowej klasy:

```

//: C14:FName1.cpp
// Klasa ifstream z nazwa pliku
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName1 {
    ifstream file;
    string fileName;
    bool named;
public:
    FName1() : named(false) {}
    FName1(const string& fname)
        : fileName(fname), file(fname.c_str()) {
        assure(file, fileName);
        named = true;
}

```

```

string name() const { return fileName; }
void name(const string& newName) {
    if(named) return; // Nie zapisuj poprzedniej nazwy
    fileName = newName;
    named = true;
}
operator ifstream&() { return file; }
};

int main() {
    FName1 file("FName1.cpp");
    cout << file.name() << endl;
    // Błąd - close() nie jest składową:
    //! file.close();
} //:-

```

Występuje tu jednak problem. Za pomocą operatora automatycznej konwersji typu — z **FName1** do **ifstream&** — próbuje się umożliwić użycie obiektu klasy **FName1** w każdym miejscu, w którym mógłby być używany obiekt klasy **ifstream**. Jednak wiersz znajdujący się w funkcji **main()**:

```
file.close();
```

nie skompiluje się, ponieważ automatyczna konwersja typów zachodzi wyłącznie w przypadku wywoływania funkcji, a nie wyboru składowych. Sposób ten nie będzie więc działał.

Druga metoda jest dodanie do klasy **FName1** definicji funkcji **close()**:

```
void close() { file.close(); }
```

Rozwiązań to sprawdzi się wówczas, gdy będziemy chcieli udostępnić tylko niektóre funkcje klasy **ifstream**. W takim przypadku wykorzystamy tylko część klasy i za-stosowanie kompozycji będzie usprawiedliwione.

Co jednak zrobić, jeśli zamierzamy udostępnić całą zawartość klasy? Jest to nazywanie *tworzeniem podtypów* (ang. *subtyping*), ponieważ tworzymy w takim przypadku nowy typ na podstawie typu już istniejącego. Chcemy bowiem, aby miał on taki sam interfejs, jak typ istniejący (powiększony o jakieś dodatkowe funkcje, które zamierzymy do niego dodać), dzięki czemu będzie można go używać w każdym miejscu, w którym mógłby zostać użyty istniejący typ. Jest to właśnie przypadek, w którym niezbędne jest dziedziczenie. Analizując zamieszczony poniżej program można się przekonać, w jaki sposób tworzenie podtypów w idealny sposób rozwiązuje problem występujący w poprzednim przykładzie:

```

//: C14:FName2.cpp
// Utworzenie podtypu rozwiązuje problem
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName2 : public ifstream {
    string fileName;
    bool named;
public:

```

```

FName2() : named(false) {}
FName2(const string& fname)
    : ifstream(fname.c_str()), fileName(fname) {
    assure(*this, fileName);
    named = true;
}
string name() const { return fileName; }
void name(const string& newName) {
    if(named) return; // Nie zapisuj poprzedniej nazwy
    fileName = newName;
    named = true;
}
};

int main() {
    FName2 file("FName2.cpp");
    assure(file, "FName2.cpp");
    cout << "name: " << file.name() << endl;
    string s;
    getline(file, s); // Te funkcje również działają!
    file.seekg(-200, ios::end);
    file.close();
} //:-

```

A zatem każda funkcja składowa, dostępna w przypadku obiektu klasy `ifstream`, jest również dostępna w obiekcie klasy `FName2`. Również funkcje niebędące funkcjami składowymi działającą z obiektami klasy `FName2` — jak na przykład funkcja `getline()`, wymagająca obiektu klasy `ifstream`. Dzieje się tak, ponieważ klasa `FName2` jest rodzajem klasy `ifstream`, a nie dlatego, że zawiera jedynie obiekt tej klasy. Ta bardzo ważna kwestia będzie rozważana do końca bieżącego rozdziału oraz w następnym rozdziale książki.

Dziedziczenie prywatne

Można dziedziczyć po klasie podstawowej w sposób prywatny, pomijając słowo **public** na liście klas podstawowych, albo jawnie wpisując na niej słowo kluczowe **private** (co jest prawdopodobnie lepszą praktyką, ponieważ dzięki temu użytkownik nie ma wątpliwości, że o to właśnie chodziło). Używając dziedziczenia prywatnego dokonuje się „implementacji na motywach czegos” — tworzy się nową klasę, która zawiera wszystkie dane i funkcje klasy podstawowej, lecz pozostają one ukryte, więc stanowią jedynie element jej wewnętrznej implementacji. Użytkownik klasy nie ma dostępu do jej wewnętrznych funkcji, a obiekt takiej klasy nie może być traktowany jako egzemplarz klasy podstawowej (jak w przypadku klasy `FName2.cpp`).

Można się zastanawiać, jakie jest zastosowanie dziedziczenia prywatnego, skoro konkurencyjne rozwiążanie, polegające na użyciu kompozycji w celu utworzenia wewnątrz klasy obiektu prywatnego, wydaje się bardziej odpowiednie. Dziedziczenie prywatne dołączono do języka po to, by był on kompletny, ale skoro jedynym powodem jego istnienia jest niewprowadzanie bałaganu, to zapewne częściej posłużysz się kompozycją niż dziedziczeniem prywatnym. Jednak mogą się czasem zdarzyć sytuacje, w których chcemy utworzyć interfejs częściowo zgodny z tym, który występuje w klasie podstawowej, nie pozwalając na używanie obiektów tej klasy w taki sposób, jakby były one obiektami klasy podstawowej. Dziedziczenie prywatne zapewnia taką możliwość.

Upublicznianie składowych dziedziczonych prywatnie

Podczas dziedziczenia prywatnego wszystkie publiczne składowe klasy podstawowej stają się w klasie pochodnej **składowymi** prywatnymi. Jeżeli chcemy, by któryś z nich był widoczny publicznie, wystarczy wymienić ich nazwy (bez argumentów funkcji ani zwracanych przez nie wartości) w publicznej części klasy pochodnej:

```
//: C14:PrivateInheritance.cpp
class Pet {
public:
    char eat() const { return 'a'; }
    int speak() const { return 2; }
    float sleep() const { return 3.0; }
    float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // Dziedziczenie prywatne
public:
    Pet::eat; // Nazwa upublicznionej składowej
    Pet::sleep; // Widoczne są obie przeciążone funkcje składowe
};

int main() {
    Goldfish bob;
    bob.eat();
    bob.sleep();
    bob.sleep(1);
//! bob.speak(); // Błąd - prywatna funkcja składowa
} //:-
```

Tak więc dziedziczenie prywatne jest przydatne, gdy chcemy ukryć część funkcji zawartych w klasie podstawowej.

Warto zwrócić uwagę na to, że podanie nazwy przeciążonej funkcji powoduje upublicznienie wszystkiej wersji, zawartych w klasie podstawowej.

Należy gruntownie przemyśleć decyzję o zastosowaniu dziedziczenia prywatnego zamiast kompozycji — dziedziczenie prywatne powoduje pewne komplikacje, gdy używa się go łącznie z identyfikacją typów w trakcie pracy programu (stanowi ona temat jednego z rozdziałów drugiego tomu książki, który można pobrać z witryny <http://helion.pl/online/thinking/index.html>).

Specyfikator **protected**

Po wprowadzeniu do dziedziczenia słowo kluczowe **protected** (chronione) nareszcie uzyskało znaczenie. W idealnym świecie składowe prywatne klasy powinny być zawsze prywatne, ale w rzeczywistych projektach zdarzają się sytuacje, w których chcemy ukryć coś przed ogółem, umożliwiając jednak dostęp składowym klas pochodnych. Zastosowanie słowa kluczowego **protected** jest ukłonem w stronę pragmatyzmu

— oznacza ono: „jest to prywatne, dopóki chodzi o użytkownika klasy, ale dostępne dla każdego, kto dziedziczy z tej klasy”.

Najlepiej dane składowe powinny pozostać prywatne — zawsze należy zostawić sobie możliwość zmiany wewnętrznej implementacji. Następnie można pozwolić klasom pochodnym na kontrolowany dostęp do nich, wykorzystując do tego celu chronione funkcje składowe:

```
//: C14:Protected.cpp
// Słowo kluczowe protected
#include <iostream>
using namespace std;

class Base {
    int i;
protected:
    int read() const { return i; }
    void set(int ii) { i = ii; }
public:
    Base(int ii = 0) : i(ii) {}
    int value(int m) const { return m*i; }
};

class Derived : public Base {
    int j;
public:
    Derived(int jj = 0) : j(jj) {}
    void change(int x) { set(x); }
};

int main() {
    Derived d;
    d.change(10);
} //:~
```

Przykłady ilustrujące konieczność zastosowania słowa kluczowego **protected** można znaleźć w dalszej części książki, a także w jwej drugim tomie.

Dziedziczenie chronione

Podczas dziedziczenia klasa podstawowa jest domyślnie prywatna, co oznacza, że wszystkie jej publiczne funkcje składowe są prywatne dla użytkownika nowo utworzonej klasy. Zazwyczaj stosuje się dziedziczenie publiczne, dzięki czemu interfejs klasy pochodnej jest taki sam, jak interfejs klasy podstawowej. Jednakże w czasie dziedziczenia można również użyć słowa kluczowego **protected**.

Dziedziczenie chronione oznacza „implementację na motywach” w stosunku do innych klas, lecz to relacja typu „jest” dla klas pochodnych oraz zaprzyjaźnionych. Jest czymś, czego nie używa się zbyt często, ale co jest dostępne w języku po to, aby był on kompletny.

Przeciążanie operatorów a dziedziczenie

Operatory, z wyjątkiem operatora przypisania, są automatycznie dziedziczone w klasach pochodnych. Można to pokazać, tworząc klasę pochodną klasy zawartej w pliku nagłówkowym **C12[Byte.h]**:

```
//: C14:OperatorInheritance.cpp
// Dziedziczenie przeciążonych operatorów
#include "../C12/Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

class Byte2 : public Byte {
public:
    // Konstruktory nie są dziedziczone:
    Byte2(unsigned char bb = 0) : Byte(bb) {}
    // operator= nie podlega dziedziczeniu, ale jest
    // generowany dla przypisania za pośrednictwem
    // elementów składowych. Jednakże, generowany
    // jest tylko operator= umożliwiający przypisanie
    // w stosunku obiektów tego samego typu, wiec inne
    // operatory przypisania muszą być utworzone jawnie:
    Byte2& operator=(const Byte& right) {
        Byte::operator=(right);
        return *this;
    }
    Byte2& operator=(int i) {
        Byte::operator=(i);
        return *this;
    }
};

// Funkcje testowe podobne do zawartych w pliku C12[ByteTest.cpp]:
void k(Byte2& bl, Byte2& b2) {
    bl = bl * b2 + b2 % bl;

#define TRY2(OP) \
    out << "bl = "; bl.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ": bl " #OP " b2 daje "; \
    (bl OP b2).print(out); \
    out << endl;

    bl = 9; b2 = 47;
    TRY2(+)
    TRY2(-)
    TRY2(*)
    TRY2(/)
    TRY2(%)
    TRY2(^)
    TRY2(&)
    TRY2(, )
    TRY2(<<)
    TRY2(>>)
    TRY2(+=)
    TRY2(-=)
    TRY2(*=)
    TRY2(/=)
    TRY2(%=)
    TRY2(^=)
    TRY2(&=)
    TRY2(|=)
    TRY2(>>=)
    TRY2(<<=)
    TRY2(=) // Operator przypisania
```

```
// Operatory warunkowe:  
#define TRYC2(OP) \  
    out << "b1 = "; b1.print(out); \  
    out << ", b2 = "; b2.print(out); \  
    out << "; b1 " <<P " b2 daje "; \  
    out << (b1 OP b2); \  
    out << endl;  
  
b1 = 9; b2 = 47;  
TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)  
TRYC2(>=) TRYC2(&&) TRYC2(|| )  
  
// Przypisanie łańcuchowe:  
Byte2 b3 = 92;  
b1 = b2 = b3;  
}  
  
int main() {  
    out << "funkcje składowe:" << endl;  
    Byte2 b1(47). b2(9);  
    k(b1, b2);  
} //:-)
```

Kod testowy jest identyczny, jak w programie **C12:ByteTest.cpp**, z wyjątkiem tego, że zamiast klasy **Byte** została użyta klasa **Byte2**. W ten sposób sprawdzono, że dzięki dziedziczeniu wszystkie operatory działają w poprawnie w klasie **Byte2**.

Analizując dokładnie klasę **Byte2**, zauważysz, że konstruktor tej klasy musi być jawnie zdefiniowany i że generowany automatycznie jest jedynie **operator=**, przypisujący obiekt klasy **Byte2** obiektem klasy **Byte2**. Wszystkie pozostałe operatory przypisania, które będą ci potrzebne, musisz wygenerować samodzielnie.

Wielokrotne dziedziczenie

Skoro można dziedziczyć po jednej klasie, to możliwość równoczesnego **dziedziczenia** po większej liczbie klas wydaje się logiczna. W istocie, wielokrotne dziedziczenie jest możliwe, ale zagadnienie, czy stanowi ono racjonalny element projektowania, jest przedmiotem nieustającej debaty. Z całą pewnością nie należy go próbować, nie mając dłuższej praktyki w programowaniu i nie znając gruntownie języka. Zanim dojdiesz do wniosku, że musisz wykorzystać wielokrotne dziedziczenie, z pewnością niemał zawsze dasz sobie radę, używając wyłącznie jednokrotnego dziedziczenia.

Początkowo wielokrotne dziedziczenie wydaje się dość proste — podczas dziedziczenia do listy klas podstawowych dodaje się większą liczbę klas, oddzielając je przecinkami. Jednakże wielokrotne dziedziczenie tworzy wiele możliwości powstania niejednoznaczności; z tego powodu w drugim tomie książki poświęcono mu cały rozdział.

Programowanie przyrostowe

Jedną z zalet dziedziczenia i kompozycji jest to, że umożliwiają one *programowanie przyrostowe*, pozwalając na dodawanie nowego kodu bez wprowadzania błędów do kodu już istniejącego. Jeżeli pojawią się błędy, to będą one izolowane w obrębie nowego kodu. Dzięki dziedziczeniu (albo kompozycji), wykorzystującemu *istniejącą*, działającą klasę, dodając do niej dane i funkcje składowe (a także zmieniając definicje istniejących funkcji składowych za pomocą dziedziczenia), pozostawiamy istniejący kod — którego ktoś jeszcze może używać — w stanie nienaruszonym; nie wprowadzamy do niego błędów. Jeżeli wystąpi jakiś błąd, to wiemy, że znajduje się w nowym kodzie, który jest znacznie krótszy i łatwiejszy do przeczytania niż w przypadku modyfikacji treści już istniejącego kodu.

To dość zdumiewające, jak wyraźnie klasy są od siebie odgraniczone. Nie potrzebujemy nawet kodu źródłowego funkcji składowych, by móc powtórnie wykorzystać kod — wystarczy do tego plik nagłówkowy, opisujący klasę, oraz plik wynikowy lub biblioteka, zawierająca skompilowane funkcje składowe (*jest* to prawdę zarówno w przypadku dziedziczenia, jak i kompozycji).

Warto pamiętać, że projektowanie oprogramowania jest procesem przyrostowym, podobnie jak proces uczenia się ludzi. Możesz wykonać dowolną liczbę analiz, ale przystępując do realizacji projektu nadal nie znasz odpowiedzi na wszystkie pytania. Osiagniesz znacznie więcej — i znacznie szybciej ujrzesz rezultaty — jeżeli zaczniesz „hodować” swój projekt, jakby był żywą, rozwijającą się istotą, zamiast tworzyć go od razu w całości, jak wielki szklany drapacz chmur².

Mimo że dziedziczenie jest użyteczną techniką eksperymentowania, to w pewnym momencie, gdy projekt nieco okrzepnie, musisz ponownie przyjrzeć się swojej hierarchii klas pod kątem przekształcenia jej w logiczną strukturę³. Pamiętaj, że dziedziczenie służy również do odzwierciedlenia relacji, którą można wyrazić słowami: „ta nowa klasa jest typu tamtej, istniejącej już klasy”. Program nie powinien koncentrować się na sposobie upychania bitów, ale na tworzeniu i operowaniu obiektami różnych typów, przedstawiających model w kategoriach określonych przez przestrzeń problemu.

Rzutowanie w górę

We wcześniejszej części rozdziału przedstawiliśmy, jak wszystkie obiekty klasy dziedziczącej po klasie **ifstream** posiadają wszystkie cechy i zachowania właściwe obiektom klasy **ifstream**. W pliku **FName2.cpp** można było wywołać każdą funkcję klasy **ifstream** w stosunku do obiektów klasy **FName2**.

¹Więcej na temat tej idei można przeczytać w książce Kenta Becka: *Extreme Programming Explained* Addison-Wesley, 2000).

²atrz — Martin Fowler: *Refactoring.Improving the Design of Existing Code* (Addison-Wesley, 1999).

Najważniejszym aspektem dziedziczenia nie jest jednak to, że dostarcza ona nowej klasie funkcje składowe. Stanowi ona relację, ustanowioną pomiędzy nowo utworzoną klasą i klasą podstawową. Relację tę można streszczyć następująco: „ta nowa klasa jest typu tamtej, istniejącej już klasy”.

Slowa te nie stanowią jedynie abstrakcyjnego sposobu opisu dziedziczenia — są one wspierane bezpośrednio przez kompilator. Jako przykład rozważmy klasę podstawową nazwane **Instrument**, reprezentującą instrumenty muzyczne, oraz klasę pochodną o nazwie **Wind** (instrumenty dęte). Ponieważ dziedziczenie oznacza, że funkcje klasy podstawowej są dostępne również w klasie pochodnej, każdy komunikat, który może być wysłany do klasy podstawowej, może być również wysłany dojej klasy pochodnej. Tak więc jeżeli klasa **Instrument** posiada funkcję składową **play()**, to ma ją również klasa **Wind**. Wyrażając się ściśle, oznacza to, że obiekt klasy **Wind** jest również obiektem typu **Instrument**. Poniższy przykład pokazuje, w jaki sposób kompilator wspiera to stwierdzenie:

```
//: C14:Instrument.cpp
// Dziedziczenie i rzutowanie w góre
enum note { middleC, Csharp, Cflat }; // Itd.

class Instrument {
public:
    void play(note) const {}
};

// Obiekty klasy Wind są obiektami typu Instrument
// ponieważ mają one taki sam interfejs:
class Wind : public Instrument {}:

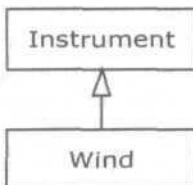
void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Rzutowanie w góre
} //
```

Interesująca jest w tym przykładzie funkcja **tune()**, pobierająca jako argument referencję do obiektu typu **Instrument**. Jednakże funkcja **tune()** została wywoływana w funkcji **main()** z argumentem będącym referencją do obiektu klasy **Wind**. Gdy przypomnimy sobie, że w języku C++ obowiązują bardzo skrupulatne reguły w sprawie kontroli typów, to może wydać się dziwne, że funkcja pobierająca argument jednego typu tak łatwo akceptuje argument innego typu. Należy jednak zdać sobie sprawę z tego, że obiekt klasy **Wind** jest również obiektem typu **Instrument**, i nie ma takiej funkcji, którą mogłaby wywołać funkcja **tune()** dla obiektu klasy **Instrument** i której nie byłoby również w klasie **Wind** (jest to gwarantowane przez dziedziczenie). Kod znajdujący się w funkcji **tune()** funkcjonuje w stosunku do klasy **Instrument** i każdej jej klasy pochodnej, a działanie, polegające na konwersji referencji lub wskaźnika do klasy **Wind** na referencję lub wskaźnik do klasy **Instrument**, nazywa się *rzutowaniem w górę* (ang. *upcasting*).

Dlaczego „rzutowanie w górę”?

Geneza tego terminu ma charakter historyczny i odwołuje się do sposobu, w jaki tradycyjnie były rysowane diagramy dziedziczenia: z klasą główną (najbardziej podstawową), znajdująca się na górze strony, rozrastające się w dół (oczywiście, diagramy te można rysować w dowolny sposób, który okaże się pomocny). A zatem w przypadku programu **Instrument.cpp** diagram dziedziczenia wygląda następująco:



Rzutowanie z klasy pochodnej do podstawowej powoduje przesunięcie się *w górę* diagramu dziedziczenia, dlatego też jest ono powszechnie nazywane rzutowaniem *w górę*. Rzutowanie *w górę* jest zawsze bezpieczne, ponieważ od typu bardziej wyspecjalizowanego przechodzimy do bardziej ogólnego — jedyna zmiana, która może zajść w interfejsie klasy, polega na tym, że może on utracić część funkcji, ale nie może w ten sposób uzyskać nowych funkcji. Dlatego też kompilator pozwala na rzutowanie *w górę*, bez konieczności używania jawnych rzutowań ani stosowania jakiejś szczególnego notacji.

Rzutowanie w górę a konstruktor kopiący

Jeżeli pozwoli się kompilatorowi na wygenerowanie konstruktora kopiącego klasy pochodnej, to automatycznie wywoła on konstruktor kopiący klasy podstawowej, a następnie konstruktory kopiące wszystkich obiektów składowych (albo przeprowadzi kopianie ich bitów — w przypadku typów wbudowanych), co zapewni jego prawidłowe działanie:

```

//: C14:CopyConstructor.cpp
// Poprawnetworzeniekonstruktorakopiującego
#include <iostream>
using namespace std;

class Parent {
    int i;
public:
    Parent(int ii) : i(ii) {
        cout << "Parent(int ii)\n";
    }
    Parent(const Parent& b) : i(b.i) {
        cout << "Parent(const Parent&)\n";
    }
    Parent() : i(0) { cout << "Parent()\n"; }
    friend ostream&
        operator<<(ostream& os, const Parent& b) {
            return os << "Parent: " << b.i << endl;
}
  
```

```

class Member {
    int i;
public:
    Member(int ii) : i(ii) {
        cout << "Member(int ii)\n";
    }
    Member(const Member& m) : i(m.i) {
        cout << "Member(const Member&)\n";
    }
    friend ostream&
    operator<<(ostream& os, const Member& m) {
        return os << "Member: " << m.i << endl;
    }
};

class Child : public Parent {
    int i;
    Member m;
public:
    Child(int ii) : Parent(ii), i(ii), m(ii) {
        cout << "Child(int ii)\n";
    }
    friend ostream&
    operator<<(ostream& os, const Child& c){
        return os << (Parent&)c << c.m
            << "Child: " << c.i << endl;
    }
};

int main() {
    Child c(2);
    cout << "wywołanie konstruktora kopującego: " << endl;
    Child c2 = c; // Wywołanie konstruktora kopującego
    cout << "wartości składowych obiektu c2:\n" << c2;
}

```

Funkcja **operator<<**, zawarta w klasie **Child**, jest interesująca ze względu na sposób, w jaki wywołuje ona funkcję **operator<<** w stosunku do zawartej w niej części klasy **Parent** — rzutując obiekt klasy **Child** na referencję **Parent&** (w przypadku rzutowania na *obiekt* klasy podstawowej, zamiast na referencję, na ogólnie uzyskalibyśmy wynik niezgodny z oczekiwany):

```
return os << (Parent&)c << c.m
```

Ponieważ dzięki temu kompilator ujmuje obiekt jako obiekt klasy **Parent**, to wywołuje zdefiniowany w tej właśnie klasie **operator<<**.

Klasa **Child** nie posiada zatem jawnie zdefiniowanego konstruktora kopującego. Kompilator generuje więc konstruktor kopujący (ponieważ jest to jedna z czterech funkcji, które generuje, obok domyślnego konstruktora — **jeżeli** nie został utworzony żaden konstruktor — **operatora** przypisania oraz destruktora), wywołując konstruktory kopujące klas **Parent** oraz **Member**. Świadczą o tym wyniki pracy programu:

```

Parent(int ii)
Member(int ii)
Child(int ii)

```

wywołanie konstruktora kopiącego:

```
Parent(const Parent&)
```

```
Member(const Member&)
```

wartości składowych obiektu c2:

```
Parent: 2
```

```
Member: 2
```

```
Child: 2
```

Jeżeli jednak spróbujemy napisać własny konstruktor kopiący klasy **Child**, ale po pełnimy niewinną pomyłkę i napiszemy go nieprawidłowo:

```
Child(const Child& c) : i(c.i), m(c.m) {}
```

to wówczas dla zawartej w klasie **Child** części, stanowiącej klasę podstawową, zostanie wywołany domyślny konstruktor. Jest to bowiem przypadek, do którego powraca kompilator, gdy nie ma możliwości wyboru żadnego innego konstruktora (trzeba pamiętać, że dla każdego obiektu, również dla obiektu podzielnego zawartego w innej klasie, zawsze musi zostać wywołany jakiś konstruktor). Wyniki działania programu byłyby w tym przypadku następujące:

```
Parent(int ii)
```

```
Member(int ii)
```

```
Child(int ii)
```

wywołanie konstruktora kopiącego:

```
Parent()
```

```
Member(const Member&)
```

wartości składowych obiektu c2:

```
Parent: 0
```

```
Member: 2
```

```
Child: 2
```

Prawdopodobnie odbiega to od naszych oczekiwaniń, ponieważ na ogół życzymy sobie, by część obiektu należąca do klasy podstawowej została skopiowana z istniejącego obiektu do nowego obiektu — w ramach działania konstruktora kopiącego.

Aby ustrzec się tego błędu, ilekroć tworzymy konstruktor kopiący, powinniśmy pamiętać o poprawnym wywołaniu konstruktora klasy podstawowej (tak jak to robi kompilator). Na pierwszy rzut oka wygląda to nieco dziwnie, ale stanowi jeszcze jeden przykład rzutowania w góre:

```
Child(const Child& c)
  : Parent(c), i(c.i), m(c.m) {
  cout << "Child(Child&)\n";
```

```
}
```

Dziwnym elementem jest tutaj wywołanie konstruktora kopiącego klasy **Parent**, zapisanym w postaci **Parent(c)**. Co jednak oznacza przekazanie obiektu klasy **Child** konstruktorowi klasy **Parent**? Ale klasa **Child** jest klasą pochodną klasy **Parent**, więc referencja do obiektu klasy **Child** jest również referencją do obiektu klasy **Parent**. Konstruktor kopiący klasy podstawowej dokonuje rzutowania w góre referencji do obiektu klasy **Child** — na referencję do obiektu klasy **Parent** — i używa go do przeprowadzenia konstrukcji przez kopiowanie. Pisząc własne konstruktory kopiące, niemal zawsze postąpisz tak samo.

Kompozycja czy dziedziczenie (po raz drugi)

Jednym z najprostszych kryteriów umożliwiających określenie, czego należy używać: kompozycji czy też dziedziczenia, jest odpowiedź na pytanie, czy w przypadku two-rzonej klasy kiedykolwiek będzie potrzebne rzutowanie w górę. W poprzedniej części rozdziału specjalizowana wersja klasy **Stack** została utworzona za pomocą dziedziczenia. Jest jednak całkiem prawdopodobne, że obiekty klasy **StringStack** będą używane wyłącznie jako kontenery przechowujące łańcuchy i nigdy nie będzie stosowane w stosunku do nich rzutowanie w górę. Bardziej odpowiednim rozwiązaniem jest więc w tym przypadku kompozycja:

```
//: C14:InheritStack2.cpp
// Kompozycja czy dziedziczenie
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack {
    Stack stack; // Osadzenie zamiast dziedziczenia
public:
    void push(string* str) {
        stack.push(str);
    }
    string* peek() const {
        return (string*)stack.peek();
    }
    string* pop() {
        return (string*)stack.pop();
    }
};

int main() {
    ifstream in("InheritStack2.cpp");
    assure(in, "InheritStack2.cpp");
    string line;
    StringStack textlines;
    while(getline(in, line))
        textlines.push(new string(line));
    string* s;
    while((s = textlines.pop()) != 0) // Brak rzutowania!
        cout << *s << endl;
} // :~
```

Program jest identyczny z programem **InheritStack.cpp**, z wyjątkiem tego, że w tym wypadku obiekt klasy **Stack** jest osadzony w klasie **StringStack**, a funkcje składowe odwołują się do tego właśnie zagnieżdżonego obiektu. Nie wiąże się to z żadnym narzutem czasowym ani pamięciowym, ponieważ obiekt podrzędny zajmuje tyle samo miejsca, a cała dodatkowa kontrola typów jest dokonywana w trakcie komplikacji.

Mimo że jest to zazwyczaj bardziej skomplikowane, można by użyć w tym przypadku również dziedziczenia prywatnego, wyrażającego „implementację na motywach”. Powinno ono również prowadzić do poprawnego rozwiązania problemu. Jednak sytuacji,

w których mogłoby mieć to znaczenie, byłaby konieczność użycia wielokrotnego dziedziczenia. Znalezienie rozwiązania, w którym zamiast dziedziczenia mogłoby zostać wykorzystana kompozycja, oznaczałoby wyeliminowanie potrzeby stosowania wielokrotnego dziedziczenia.

Rzutowanie w górę wskaźników i referencji

W programie **Instrument.cpp** rzutowanie w górę odbywa się w trakcie wywołania funkcji — na zewnątrz funkcji pobierana jest referencja do obiektu klasy **Wind**, która we wnętrzu funkcji staje się referencją do klasy **Instrument**. Rzutowanie w górę może mieć również miejsce podczas zwykłego przypisania wartości **wskaźnikowi** lub referencji:

```
Wind w;  
Instrument* ip = &w; // Rzutowanie w górę  
Instrument& ir = w; // Rzutowanie w górę
```

Podobnie jak w przypadku funkcji, żadna z tych sytuacji nie wymagającego rzutowania.

Kryzys

Oczywiście, każde rzutowanie w górę powoduje utratę informacji o typie obiektu. Jeżeli napiszemy:

```
Wind w;  
Instrument* ip = &w;
```

to kompilator będzie traktował zmienną **ip** wyłącznie jako wskaźnik do obiektu klasy **Instrument** i nic ponadto. Oznacza to, że nie może on wiedzieć, że zmienna **ip** w rzeczywistości wskazuje akurat obiekt klasy **Wind**. Tak więc gdy wywołamy funkcję składową **play()**, pisząc:

```
ip->play(middleC);
```

to kompilator wie jedynie, że wywołuje funkcję **play()** dla wskaźnika obiektu klasy **Instrument**. Wywoła zatem zawartą w klasie podstawowej wersję funkcji **Instrument::play()** zamiast funkcję **Wind::play()**. Tak więc zachowanie programu będzie niepoprawne.

Jest to poważny problem — został on rozwiązany w rozdziale 15., w którym opisano kolejny filar programowania obiektowego — **polimorfizm** (zaimplementowany w języku C++ w postaci funkcji wirtualnych).

Podsumowanie

Zarówno dziedziczenie, jak i kompozycja pozwalają na utworzenie nowego typu danych na podstawie typów już istniejących. Obie te metody osadzają w nowo utworzonym typie obiekty podległe istniejącym typów. Zazwyczaj kompozycji **używamy**

jednak w przypadku, gdy chcemy wykorzystać istniejące typy w charakterze elementu wewnętrznej implementacji nowej klasy. Dziedziczenie stosujemy zaś wtedy, gdy chcemy wymusić, by nowy typ danych był tego samego typu, co klasa podstawowa (równoważność typów gwarantuje równoważność interfejsów). Ponieważ klasa pochodząca posiada interfejs klasy podstawowej, to może ona być *rzutowana w górę* – na klasę podstawową, co ma najistotniejsze znaczenie dla **polimorfizmu**, o czym przekonamy się w rozdziale 15.

Mimo że wielokrotne wykorzystywanie kodu, uzyskiwane za pomocą kompozycji oraz dziedziczenia, znacznie pomaga w szybkim rozwoju projektu, to na ogół przed udostępnieniem hierarchii używanych klas innym programistom trzeba je jeszcze przeprojektować. Naszym głównym celem jest utworzenie hierarchii klas, w której każda klasa będzie miała określone zastosowanie i nie będzie ani zbyt duża (gdyż wówczas zawierałaby tak wiele funkcji, że jej używanie byłoby niewygodne), ani irytująco mała (bo wówczas nie można by używać jej osobno albo trzeba by dodać do niej nowe funkcje).

Ćwiczenia

Rozwiązania wybranych ćwiczeń można znaleźć w dokumencie elektronicznym *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny <http://www.BruceEckel.com>.

1. Zmodyfikuj klasę Car, zawartą w programie **Car.cpp**, w taki sposób, by dziedziczyła również z klasy o nazwie **Vehicle** (pojazd), umieszczając w klasie **Vehicle** odpowiednie funkcje składowe (tworząc w niej jakieś funkcje składowe). Dodaj do klasy **Vehicle** konstruktor niebędący konstruktorem domyślnym, który musisz wywołać w konstruktorze klasy **Car**.
2. Utwórz dwie klasy, A i B, posiadające domyślne konstruktory, sygnalizujące, że zostały wywołane. Z klasy A wyprowadź nową klasę o nazwie C i utwórz w niej obiekt **składowy** klasy B, ale nie twórz konstruktora klasy C. Utwórz obiekt klasy C i przyjrzyj się wynikom działania programu.
3. Utwórz trzypoziomową hierarchię klas, posiadających domyślne konstruktory oraz destruktory, które informują swoim uruchomieniu, wyprowadzając komunikaty do strumienia cout. Upewnij się, że w przypadku obiektu ostatniej z klas pochodnych wszystkie trzy konstruktory i destruktory są automatycznie wywoływane. Wytlumacz kolejność, w jakiej odbywają się wywoływanie.
4. Zmodyfikuj program **Combined.cpp**, by dodać jeszcze jeden poziom dziedziczenia i nowy obiekt składowy. Dodaj kod, informujący o tym, kiedy **wywonywane** są konstruktory i destruktory.
5. W programie **Combined.cpp** utwórz klasę D, dziedziczącą z klasy B oraz posiadającą obiekt składowy klasy C. Dodaj kod informujący o tym, kiedy wywoływanie są konstruktory i destruktory.

6. Zmodyfikuj program **Order.cpp**, dodając na kolejnym poziomie dziedziczenia klasę **Derived3**, posiadającą obiekty składowe klas **Member4** i **Member5**. Przyjrzyj się wynikom działania programu.
7. W programie **NameHiding.cpp** sprawdź, że w klasach **Derived2**, **Derived3** i **Derived4** nie jest dostępna żadna ze znajdujących się w klasie podstawowej wersji funkcji **f()**.
8. Zmodyfikuj program **NameHiding.cpp**, dodając do klasy **Base** trzy przeciążone funkcje **h()**, i pokaż, że przeddefiniowanie jednej z nich w klasie pochodnej powoduje ukrycie wszystkich pozostałych.
9. Z klasy **vector<void*>** wyprowadź klasą pochodną **StringVector**, a następnie przeddefiniuj funkcje składowe **push_back()** oraz **operator[]** w taki sposób, aby pobierały i zwracały wartości typu **string***. Co się stanie, jeśli spróbujesz użyć funkcji **push_back()** z argumentem typu **void***?
10. Utwórz klasę zawierającą liczbę typu **long** i użądź ją w konstruktorze składni wywołania pseudokonstruktora, by zainicjować tę liczbę.
11. Utwórz klasę o nazwie **Asteroid**. Wykorzystaj dziedzicznie, by utworzyć specjalizowaną wersję klasy **PStash**, opisanej w rozdziale 13. (**PStash.h** i **PStash.cpp**). Wersja ta pobiera i zwraca wskaźniki do obiektów klasy **Asteroid**. Zmodyfikuj również program **PStashTest.cpp**, by przetestować swoje klasy. Zmodyfikuj utworzoną klasę w taki sposób, aby zamiast dziedziczenia zawierała obiekt składowy klasy **PStash**.
12. Powtórz poprzednie ćwiczenie, używając klasę **vector** zamiast klasy **PStash**.
13. W programie **SynthesizedFunctions.cpp** zmodyfikuj klasę **Chess**, dodając do niej domyślny konstruktor, konstruktor kopiący oraz operator przypisania. Wykaż, że napisałeś prawidłowo.
14. Utwórz dwie klasy o nazwach **Traveler** i **Pager**, nieposiadające domyślnych konstruktorów, a tylko konstruktory pobierające argument typu **string**, który klasy te będą kopować do wewnętrznej zmiennej typu **string**. Dla każdej z klas utwórz prawidłowy konstruktor kopiący oraz operator przypisania. Następnie utwórz z klasy **Traveler**, poprzez dziedziczenie, klasę **BusinessTraveler** i dodaj do niej obiekt składowy typu **Pager**. Napisz poprawny konstruktor domyślny, konstruktor pobierający argument typu **string**, konstruktor kopiący i operator przypisania.
15. Utwórz klasę zawierającą dwie statyczne funkcje składowe. Utwórz ją klasę pochodną i przeddefiniuj jedną z tych funkcji składowych. Pokaż, że druga funkcja została ukryta w klasie pochodnej.
16. Poszukaj informacji o innych funkcjach składowych klasy **ifstream**. Wypróbuj je w pliku **FName2.cpp**, używając do tego celu obiektu **file**.
17. Użyj dziedziczenia prywatnego (**private**) i chronionego (**protected**) do utworzenia dwóch klas, będących klasami pochodnymi jednej klasy podstawowej. Następnie spróbuj dokonać rzutowania w górę klas pochodnych na klasę podstawową. Wyjaśnij uzyskany rezultat.

18. W programie **Protected.cpp** dodaj do klasy **Derived** funkcję składową, która wywołuje chronioną funkcję składową **read()** klasy **Base**.
19. Zmodyfikuj program **Protected.cpp** w taki sposób, aby klasa **Derived** wykorzystywała dziedziczenie chronione. Zobacz, czy dla obiektu klasy **Derived** możesz wywołać funkcję **value()**.
20. Utwórz klasę o nazwie **SpaceShip**, zawierającą funkcję **fly()**. Z klasy **SpaceShip** wyprowadź klasę **Shuttle**, dodając do niej funkcję **land()**. Utwórz obiekt klasy **Shuttle**, za pomocą wskaźnika lub referencji dokonaj jego rzutowania w górę, na klasę **SpaceShip**, a następnie spróbuj wywołać funkcję **land()**. Wyjaśnij uzyskane rezultaty.
21. Zmodyfikuj program **Instrument.cpp**, dodając do klasy **Instrument** funkcję **prepare()**. Wywołaj funkcję **prepare()** wewnątrz funkcji **tune()**.
22. Zmodyfikuj program **Instrument.cpp** w taki sposób, aby funkcja **play()** drukowała komunikat do strumienia **cout**, a klasa **Wind** zawierała przedefiniowaną funkcję **play()**, drukującą do strumienia **cout** inny komunikat. Uruchom program i wyjaśnij, dlaczego funkcjonuje on odmiennie od oczekiwanych rezultatów. Następnie umieść słowo kluczowe **virtual** (zostanie ono opisane w rozdziale 15.) przed deklaracją funkcji **play()**, znajdującej się w klasie **Instrument**, i zobacz, w jaki sposób zmieniło się działanie programu.
23. W pliku **CopyConstructor.cpp** wyprowadź z klasy **Child** nową klasę i utwórz w niej obiekt składowy m klasy **Member**. Napisz prawidłowy konstruktor, konstruktor kopiący, **operator=** oraz **operator<<** dla strumieni wejścia-wyjścia i przetestuj działanie tej klasy w funkcji **main()**.
24. Zmodyfikuj program **CopyConstructor.cpp**, dodając do klasy **Child** własny konstruktor kopiący (*nie wywoły* konstruktora kopiącego klasy podstawowej). Rozwiąż problem, dokonując na liście inicjatorów konstruktora kopiącego klasy **Child** prawidłowego, jawnego wywołania konstruktora kopiącego klasy podstawowej.
25. Zmodyfikuj program **InheritStack2.cpp**, by zamiast klasy **Stack** używał wektora **vector<string>**.
26. Utwórz klasę **Rock**, posiadającą domyślny konstruktor, konstruktor kopiący, operator przypisania oraz destruktor (z których wszystkie sygnalizują, że zostały wywołane), wyprowadzając komunikat do strumienia **cout**. W funkcji **main()** utwórz wektor **vector<Rock>** (przechowujący wartości obiektów klasy **Rock**) i dodaj do niego kilka obiektów. Uruchom program i wyjaśnij uzyskane wyniki. Sprawdź, czy są wywoływane destruktory obiektów klasy **Rock**, znajdujących się w wektorze. Następnie powtórz ćwiczenie, używając wektora **vector<Rock*>**. Czy jest możliwe utworzenie wektora **vector<Rock&>?**
27. Ćwiczenie to tworzy wzorzec projektowy o nazwie *pośrednik* (ang. *proxy*). Rozpocznij od klasy podstawowej **Subject** i utwórz w niej trzy funkcje: **f()**, **g()** i **h()**. Następnie wyprowadź z niej klasę **Proxy** oraz dwie inne, zawierające implementacje — **Implementation1** i **Implementation2**. Klasa **Proxy** powinna zawierać wskaźnik do obiektu klasy **Subject**, a wszystkie

funkcje składowe klasy **Proxy** powinny przekazywać swoje wywołania do siebie samych za pośrednictwem zawartego w klasie wskaźnika klasy **Subject**. Konstruktor klasy **Proxy** pobiera wskaźnik do obiektu klasy **Subject**, który zostaje zainstalowany w klasie **Proxy** (zazwyczaj za pomocą konstruktora). W funkcji **main()** utwórz dwa różne obiekty klasy **Proxy**, wykorzystujące dwie rozmaite implementacje. Następnie zmodyfikuj klasę **Proxy** tak, aby można było dynamicznie zmieniać implementacje.

28. Zmodyfikuj program **ArrayOperatorNew.cpp**, znajdujący się w rozdziale 13., by pokazać, że jeżeli dziedziczy się z klasy **Widget**, to przydział pamięci nadal działa prawidłowo. Wyjaśnij, dlaczego dziedziczenie *nie dzialalo*by prawidłowo w przypadku programu **Framis.cpp**, zawartego w rozdziale 13.
29. Zmodyfikuj program **Framis.cpp**, znajdujący się w rozdziale 13., dziedzicząc po klasie **Framis** i tworząc dla klasy pochodnej nowe wersje operatorów **new** i **delete**. Pokaż, że działają one prawidłowo.

Rozdział 15.

Polimorfizm

i funkcje wirtualne

Polimorfizm (zaimplementowany w języku C++ w postaci funkcji wirtualnych) stanowi trzecią fundamentalną właściwość obiektowego języka programowania — po abstrakcji danych i dziedziczeniu.

Tworzy on nowy wymiar oddzielenia interfejsu od implementacji, odgraniczając pojęcie *co* od *pojęcia jak*. Polimorfizm zapewnia lepszą organizację i czytelność kodu, a także tworzenie *możliwych do rozszerzania* programów, które mogą „rosnąć” nie tylko podczas tworzenia pierwotnego projektu, ale również wtedy, gdy trzeba wyposażyć go w nowe cechy.

Kapsułkowanie tworzy nowe typy danych, łącząc ze sobą cechy z zachowaniami. Kontrola dostępu oddziela interfejs od implementacji, czyniąc szczegóły prywatnymi. Taki sposób mechanicznej organizacji kodu jest od razu *zrozumiałym* dla każdego, kto posiada doświadczenie w programowaniu w językach proceduralnych. Jednak funkcje wirtualne dokonują podziału w kategoriach *typów*. Dzięki lekturze rozdziału 14. dowiedzieliśmy się, że dziedziczenie pozwala na traktowanie obiektu w taki sposób, jakby był obiektem swojego typu *albo* swojego typu podstawowego. Ta zdolność ma podstawowe znaczenie, ponieważ umożliwia traktowanie wielu typów danych (wy prowadzonych z tego samego typu podstawowego) w taki sposób, jakby były jednym typem. Dzięki temu pojedynczy fragment kodu może działać identycznie z różnymi typami danych. Funkcje wirtualne pozwalają typowi danych na wyrażenie swojej odmiенноści w stosunku od innego, podobnego typu, pod warunkiem, że oba typy danych są wprowadzone z tego samego typu podstawowego. Odmienna ta wyraża się poprzez różnice w zachowaniu funkcji, które można wywołać za pośrednictwem klasy podstawowej.

W niniejszym rozdziale poznamy funkcje wirtualne. Zaczniemy od podstaw, obejmujących proste przykłady programów, których pozbawiono wszystkiego, z wyjątkiem „wirtualności”.

Ewolucja programistów języka C++

Wydaje się, że programiści języka C przyswajają sobie język C++ w trzech etapach. Najpierw — jako „lepsze C”, ponieważ język C++ wymusza deklarowanie wszystkich funkcji przed ich użyciem i jest znacznie bardziej wymagający w kwestii użycia zmiennych. Często można znaleźć błędy, zawarte w programie napisanym w języku C, już podczas kompilowania go za pomocą kompilatora języka C++.

Drugim etapem jest „obiektowe” C++. Oznacza to dostrzeżenie korzyści: związanych z organizacją kodu, ze zgrupowaniem struktur danych wraz z działającymi na tych danych funkcjami, wynikających ze znaczenia konstruktorów i destruktorów, a być może również z prostego dziedziczenia.

Większość programistów mających doświadczenie w pracy z językiem C szybko dostrzega wynikające z tego pozytyki, ponieważ to właśnie próbują osiągnąć, tworząc biblioteki. W przypadku języka C++ mogalicość w tej kwestii na pomoc kompilatora.

Łatwo jest utknąć na poziomie obiektów, ponieważ szybko można go osiągnąć i zapewnia on wiele korzyści, nie wymagając zarazem większego wysiłku umysłowego. Można zasmakować w generowaniu nowych typów danych — tworzy się klasy i obiekty, wysyła do nich komunikaty, a wszystko jest łatwe i przyjemne.

Nie idź jednak na łatwiznę; jeżeli zatrzymasz się w tym miejscu, nie poznasz najwspanialszej części języka, która pozwala na przeskok do prawdziwego programowania obiektowego. Można to zrobić tylko za pomocą funkcji wirtualnych.

Funkcje wirtualne wzmacniają pojęcie typu zamiast zamykać kod w strukturach i odgradzać go; dla początkującego programisty języka C++ stanowią więc bez wątpienia najtrudniejsze do zrozumienia pojęcie. Jednakże wyznaczają one również punkt zwrotny w rozumieniu programowania obiektowego. Jeżeli nie używasz funkcji wirtualnych, oznacza to, że jeszcze go nie pojąłeś.

Z uwagi na to, że funkcje wirtualne są ściśle związane z pojęciem typów, a typy stanowią istotę programowania obiektowego, funkcje wirtualne nie posiadają żadnego odpowiednika w tradycyjnych językach proceduralnych. Będąc programistą rozumującym w kategoriach procedur, nie znajdziesz punktu odniesienia, na którego podstawie mógłbyś ująć funkcje wirtualne, jak to jest możliwe w przypadku niemal każdej innej właściwości tego języka. Cechy zawarte w języku proceduralnym mogą być zrozumiałe na poziomie algorytmicznym, lecz funkcje wirtualne można pojąć wyłącznie z punktu widzenia projektu.

Rzutowanie w górę

W rozdziale 14. przedstawiono, w jaki sposób obiekty mogły być używane, zarówno jako obiekty swoich typów, jak i jako obiekty swoich typów podstawowych. Ponadto można nimi operować za pomocą adresu typu podstawowego. Pobranie adresu

objektu (zarówno w postaci wskaźnika, jak i referencji) i traktowanie go jako adresu typu podstawowego nosi nazwę *rzutowania w górę*, z uwagi na sposób, w jaki rysowane są diagramy dziedziczenia — w postaci drzew, ze znajdująca się na górze klasą podstawową.

Zetknęliśmy się również z problemem, zawartym w poniższym kodzie:

```
//: C15:Instrument2.cpp
// Dziedziczenie i rzutowanie w górę
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Itd.

class Instrument {
public:
    void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

// Obiekty klasy Wind są obiektami typu Instrument
// ponieważ mają one taki sam interfejs:
class Wind : public Instrument {
public:
    // Redefine interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Rzutowanie w górę
} III-
```

Funkcja `tune()` pobiera (przez referencję) obiekt klasy **Instrument**, lecz bez protestu przyjmuje również obiekty wszelkich typów wyprowadzonych klasy **Instrument**. W funkcji `main()` można zauważyć, że dzieje się tak w przypadku, gdy obiekt klasy **Wind** przekazywany jest do funkcji `tune()`, bez potrzeby rzutowania. Jest to dopuszczalne — interfejs klasy **Instrument** musi być zawarty w klasie **Wind**, ponieważ klasa **Wind** dziedziczy publicznie po klasie **Instrument**. Rzutowanie w górę — z klasy **Wind** do **Instrument** — może „zawędzić” interfejs, ale nigdy nie ograniczy go w większym stopniu niż do pełnego interfejsu klasy **Instrument**.

Te same zasady dotyczą wskaźników — jedyna różnica polega na tym, że użytkownik musi jawnie pobrać adres obiektów przekazywanych funkcji.

Problem

O tym, na czym polega problem związany z programem **Instrument2.cpp**, można się przekonać uruchamiając ów program. Wynikiem jego działania jest wyprowadzenie tekstu **Instrument::play**. Nie o to nam, oczywiście, chodziło, ponieważ przypadkowo wiemy, że obiektem jest naprawdę obiekt klasy **Wind**, a nie jakiś **Instrument**. Wynikiem działania programu powinno być wywołanie funkcji **Wind::play**. Z tego właśnie powodu każdy obiekt klasy wyprowadzonej z klasy **Instrument** powinien posiadać własną wersję funkcji **play()**, używaną bez względu na sytuację.

Zachowanie programu **Instrument2.cpp** nie jest niespodzianką, jeżeli uwzględni się podejście języka C w stosunku do funkcji. Aby zrozumieć te kwestie, trzeba poznać pojęcie *wiązania*.

Wiązanie wywołania funkcji

Połączenie wywołania funkcji z jej ciałem jest nazywane *wiązaniem* (ang. *binding*). W przypadku gdy wiązanie jest dokonywane przed uruchomieniem programu (przez kompilator lub program łączący), mówimy o *wczesnym wiązaniu* (ang. *early binding*). Być może nigdy wcześniej nie zetknąłeś się z tym pojęciem, ponieważ w przypadku języków proceduralnych nigdy nie stanowiło ono możliwej do wybrania opcji — kompilatory języka C znają tylko jeden sposób wywołania funkcji i rodzajem tym jest właśnie wczesne wiązanie.

Problem, który pojawił się w powyższym programie, został spowodowany wczesnym wiązaniem, ponieważ kompilator, dysponując jedynie adresem obiektu klasy **Instrument**, nie jest w stanie określić, jaką powinien wywołać funkcję.

Rozwiązań nosi nazwę *późnego wiązania* (ang. *late binding*), co oznacza, że wiązanie dokonywane jest w trakcie wykonywania programu na podstawie informacji o typie obiektu. Późne wiązanie jest niekiedy również określane mianem *wiązania dynamicznego* (ang. *dynamische binding*) albo *wiązania podczas wykonywania programu* (ang. *runtime binding*). Jeśli język obsługuje późne wiązanie, musi istnieć jakiś mechanizm, umożliwiający określenie typu obiektu w trakcie wykonywania programu i wywołanie odpowiedniej funkcji składowej. W przypadku komplikowanego języka kompilator co prawda nadal nie wie, jaki jest rzeczywisty typ obiektu, ale wstawia kod, umożliwiający odnalezienie i wywołanie odpowiedniego ciała funkcji. Realizacja mechanizmu, odpowiedzialnego za późne wiązanie, zależy od języka, ale można sobie wyobrazić, jakiego rodzaju informacje dotyczące typu muszą zostać umieszczone w obiekcie. W dalszej części rozdziału zobaczymy, jak to funkcjonuje.

Funkcje wirtualne

Aby spowodować późne wiązanie jakiejś określonej funkcji w języku C++, wymagane jest użycie słowa kluczowego **virtual** w deklaracji tej funkcji, znajdującej się w klasie podstawowej. Późne wiązanie ma miejsce wyłącznie w przypadku funkcji

wirtualnych i tylko wówczas, gdy używany jest adres klasy podstawowej, w której zawarte są te funkcje wirtualne, mimo że mogą one być również zdefiniowane w jakiejś wcześniejszej klasie podstawowej.

Aby zadeklarować jakąś funkcję jako funkcję wirtualną, wystarczy poprzedzić deklarację tej funkcji słowem kluczowym **virtual**. Słowo to jest wymagane wyłącznie w deklaracji funkcji, a nie w jej definicji. Jeżeli funkcja została zadeklarowana jako wirtualna w klasie podstawowej, to jest ona również funkcją wirtualną we wszystkich jej klasach pochodnych. Przedefiniowanie funkcji wirtualnej w klasie pochodnej jest zazwyczaj określane mianem **zasłaniania** (ang. *overriding*).

Zwróć uwagę na to, że wystarczy zadeklarować funkcję jako wirtualną w klasie podstawowej. Wszystkie funkcje, znajdujące się w klasach pochodnych, których sygnatury będą odpowiadały deklaracji zawartej w klasie podstawowej, będą wywoływane za pomocą mechanizmu obsługującego funkcje wirtualne. W deklaracjach zawartych w klasie pochodnej *można* użyć słowa kluczowego **virtual** (nie wywołuje to żadnych niepożądanych skutków), ale jest to nadmiarowe i niekiedy mylące.

Aby uzyskać pożądane zachowanie programu **Instrument2.cpp**, wystarczy przed deklaracją funkcji **play()**, znajdującej się w klasie podstawowej, umieścić słowo kluczowe **virtual**:

```
//: C15:Instrument3.cpp
// Późne wiązanie dzięki użyciu słowa kluczowego virtual
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Itd.

class Instrument {
public:
    virtual void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

// Obiekty klasy Wind są obiektami typu Instrument
// ponieważ mają one taki sam interfejs:
class Wind : public Instrument {
public:
    // Zasłonięcie funkcji interfejsu:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Rzutowanie w gore
} //:=-
```

Program ten jest identyczny z programem **Instrument2.cpp**, z wyjątkiem dodanego słowa kluczowego **virtual**, ajednak działa od niego zupełnie inaczej. Obecnie wprowadza on tekst **Wind::play**.

Rozszerzalność

W przypadku gdy funkcja **play()** jest zadeklarowana jako wirtualna w klasie podstawowej, można utworzyć dowolną liczbę nowych typów, nie zmieniając w ogóle funkcji **tune()**. W dobrze zaprojektowanym programie obiektowym większość funkcji będzie wyglądała tak, jak funkcja **tune()** — komunikując się wyłącznie z interfejsem klasy podstawowej. Program taki jest *rozszerzalny*, ponieważ nowe funkcje są dodawane poprzez wprowadzanie nowych typów danych ze wspólnej klasy podstawowej. Nie ma potrzeby wprowadzania jakichkolwiek zmian w funkcjach, odwołujących się do interfejsu klasy **podstawowej**, by mogły one obsługiwać nowo utworzone klasy.

Poniżej zamieszczony jest przykład dotyczący instrumentów. Zawiera on większą liczbę funkcji wirtualnych i szereg nowych klas, działających prawidłowo z poprzednią, niezmienioną postacią funkcji **tune()**:

```
//: C15:Instrument4.cpp
// Możliwość rozszerzania programów w programowaniu obiektowym
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Itd.

class Instrument {
public:
    virtual void play(note) const {
        cout << "Instrument::play" << endl;
    }
    virtual char* what() const {
        return "Instrument";
    }
    // Zakładamy, że funkcja modyfikuje obiekt:
    virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
    char* what() const { return "Wind"; }
    void adjust(int) {}
};

class Percussion : public Instrument {
public:
    void play(note) const {
        cout << "Percussion::play" << endl;
    }
    char* what() const { return "Percussion"; }
    void adjust(int) {}
};
```

```
class Stringed : public Instrument {
public:
    void play(note) const {
        cout << "Stringed::play" << endl;
    }
    char* what() const { return "Stringed"; }
    void adjust(int) {}
};

class Brass : public Wind {
public:
    void play(note) const {
        cout << "Brass::play" << endl;
    }
    char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
    void play(note) const {
        cout << "Woodwind::play" << endl;
    }
    char* what() const { return "Woodwind"; }
};

// Funkcja taka sama, jak poprzednio:
void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

// Nowa funkcja:
void f(Instrument& i) { i.adjust(1); }

// Rzutowanie w góre podczas inicjalizacji tablicy:
Instrument* A[] = {
    new Wind,
    new Percussion,
    new Stringed,
    new Brass,
};

int main() {
    Wind flute;
    Percussion drum;
    Stringed violin;
    Brass flugelhorn;
    Woodwind recorder;
    tune(flute);
    tune(drum);
    tune(violin);
    tune(flugelhorn);
    tune(recorder);
    f(flugelhorn);
} //:~
```

Poniżej klasy **Wind** został utworzony dodatkowy poziom dziedziczenia, ale mechanizm funkcji wirtualnych działa prawidłowo, niezależnie od istniejącej liczby poziomów. Funkcja **adjust()** nie została zasłonięta w klasach **Brass** (blaszane instrumenty dęte) i **Woodwind** (drewniane instrumenty dęte). W takich przypadkach automatycznie używana jest definicja, znajdująca się „najbliżej” w hierarchii klas. Kompilator gwarantuje, że zawsze *istnieje jakąś* definicja funkcji wirtualnej, dzięki czemu nigdy nie może zdarzyć się wywołanie, które nie zostałoby powiązane z ciałem jakiejś funkcji (byłyby to katastrofalne).

Tablica A[] zawiera wskaźniki do klasy podstawowej **Instrument**, dzięki czemu rzutowanie w górę odbywa się podczas inicializacji tablicy. Tablica ta oraz funkcja **f()** zostaną wykorzystane w dalszej części dyskusji.

W przypadku wywołań funkcji **tune()** rzutowanie przebiega oddzielnie w stosunku do każdego rodzaju obiektu, jednak zawsze działa zgodnie z oczekiwaniami. Można to opisać jako „wysłanie komunikatu obiektowi, który zadecyduje o tym, co z nim zrobić”. Funkcja wirtualna jest lupą, której można użyć, próbując dokonać analizy projektu — gdzie powinny znajdować się klasy podstawowe i w jaki sposób zamierzamy rozszerzać program. Nawet gdy, rozpoczęwszy pracę nad programem, nie odkryjemy właściwych interfejsów klas podstawowych oraz funkcji wirtualnych, często zauważymy je później, czasami nawet znacznie później — planując jego rozszerzenie lub przeprowadzając jego pielęgnację w jakiś inny sposób. Nie wynika to z błędów popełnionych w trakcie analizy czy projektowania — oznacza jedynie, że nie znaleźliśmy lub nie mogliśmy *znać* od razu wszystkich informacji. Dzięki ścisłej modularyzacji klas w języku C++ wystąpienie takiej sytuacji nie stanowi poważnego problemu, ponieważ zmiany dokonywane w jednej części systemu nie przenoszą się do jego innych części, jak w języku C.

W jaki sposób język C++ realizuje późne wiązanie?

W jaki sposób dokonuje się późne wiązanie? Cała praca jest wykonywana w niewidoczny sposób przez kompilator, instalujący wszelkie niezbędne mechanizmy związane z późnym wiązaniem — gdy tego zażdamy, tworząc funkcje wirtualne. Z uwagi na to, że rozumienie mechanizmu działania funkcji wirtualnych w języku C++ często przydaje się podczas programowania, w podrozdziale zostanie szczegółowo opisany sposób, w jaki kompilator realizuje ten mechanizm.

Słowo kluczowe **virtual** informuje kompilator, że nie powinien dokonywać wcześniego wiązania. Powinien natomiast automatycznie zainstalować wszystkie mechanizmy, niezbędne do przeprowadzenia późnego wiązania. Oznacza to, że w przypadku gdy funkcja **play()** zostanie wywoływana dla obiektu **Brass**, *podanego w postaci adresu obiektu klasy podstawowej* **Instrument**, wywołana zostanie odpowiednia funkcja.

Aby to zrealizować, typowy kompilator¹ dla każdej klasy zawierającej funkcje wirtualne tworzy pojedynczą tablicę (nazywaną VTABLE). W tablicy VTABLE kompilator umieszcza adresy wirtualnych funkcji zawartych w klasie. W każdej klasie posiadającej funkcje wirtualne niejawnie umieszczany jest wskaźnik wirtualny (oznaczany skrótem VPTR — od ang. *vpointer, virtual pointer*), wskazujący tablicę VTABLE tej klasy. Gdy za pośrednictwem wskaźnika obiektu klasy podstawowej wywołuje się funkcję wirtualną (czyli dokonuje się wywołania polimorficznego), kompilator niejawnie wstawia kod, pobierający wskaźnik VPTR i odnajdujący adres funkcji w tablicy VTABLE. Wywołuje on w ten sposób właściwą funkcję i umożliwia realizację późnego wiązania.

Wszystkie te działania — tworzenie dla każdej klasy tablicy VTABLE i wstawianie kodu, wywołania funkcji wirtualnej — odbywają się automatycznie, w związku z czym nie trzeba sobie nimi zaprzątać uwagi. Dzięki wirtualnym funkcjom dla obiektu zostaje wywołana właściwa funkcja, nawet w przypadku gdy kompilator nie zna dokładnego typu.

W kolejnych podrozdziałach opisano ten proces bardziej szczegółowo.

Przechowywanie informacji o typie

W żadnej z klas nie jest w jawny sposób przechowywana informacja o typie. Tymczasem poprzednie przykłady i zwyczajna logika podpowiadają, że w obiektach musi być przechowywany jakiś rodzaj informacji dotyczącej ich typu — w przeciwnym razie typ obiektów nie mógłby bowiem zostać określony w trakcie działania programu. To prawda, ale informacja dotycząca typu jest ukryta. Aby się o tym przekonać, prześledźmy przykładowy program, porównujący wielkości obiektów klas, zawierających funkcje wirtualne, z obiektami tych klas, które ich nie używają:

```
//: C15:Sizes.cpp
// Wielkość obiektów klas z funkcjami
// wirtualnymi i bez nich
#include <iostream>
using namespace std;

class NoVirtual {
    int a;
public:
    void x() const {}
    int i() const { return 1; }
};

class OneVirtual {
    int a;
public:
    virtual void x() const {}
    int i() const { return 1; }
};
```

¹ Kompilatory mogą realizować działanie funkcji wirtualnych w dowolny sposób, ale przedstawiona tutaj metoda stanowi niemal uniwersalne rozwiązanie problemu.

```

class TwoVirtuals {
    int a;
public:
    virtual void x() const {}
    virtual int i() const { return 1; }
};

int main() {
    cout << "int: " << sizeof(int) << endl;
    cout << "NoVirtual: "
        << sizeof(NoVirtual) << endl;
    cout << "void*: " << sizeof(void*) << endl;
    cout << "OneVirtual: "
        << sizeof(OneVirtual) << endl;
    cout << "TwoVirtuals: "
        << sizeof(TwoVirtuals) << endl;
} //:-

```

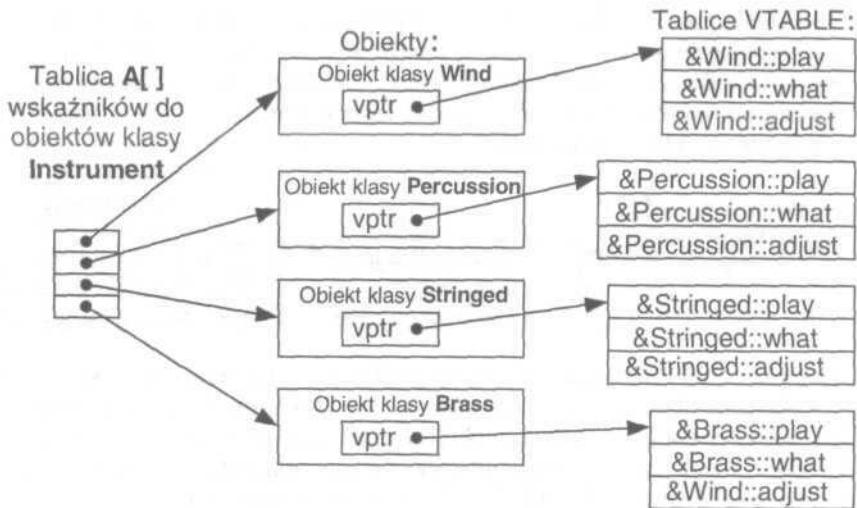
W przypadku braku funkcji wirtualnych (klasa **NoVirtual**) wielkość obiektu jest dokładnie zgodna z oczekiwaniami — jest ona wielkością **pojedynczej²** zmiennej typu **int**. Jeśli chodzi o klasę **OneVirtual**, zawierającą jedną funkcję wirtualną, wielkością obiektu jest wielkość obiektu klasy **NoVirtual**, powiększona o rozmiar wskaźnika typu **void***. Okazuje się, że w przypadku jedynej lub większej liczby funkcji wirtualnych kompilator umieszcza w strukturze klasy pojedynczy wskaźnik (**VPTR**). Nie ma żadnej różnicy wielkości pomiędzy obiekttami klas **OneVirtual** i **TwoVirtuals**. Wynika to z tego, że wskaźnik **VPTR** wskazuje tablicę zawierającą adresy funkcji. Wystarczy tylko jedna tablica, ponieważ adresy wszystkich funkcji wirtualnych klasy zawarte są w pojedynczej tablicy.

Przykład ten wymaga zastosowania przynajmniej jednej składowej. Gdyby klasy nie zawierały w ogóle danych składowych, kompilator języka C++ narzuciłby niezerową wielkość obiektów, z uwagi na to, że każdy obiekt musi posiadać unikalny adres. Latwo to zrozumieć, próbując wyobrazić sobie odwoływanie się do obiektów o zerowej wielkości, znajdujących się w tablicy. W związku z tym do obiektów, które miałyby zerową wielkość, wstawiane są „ślepe” składowe. W przypadku umieszczenia w klasie informacji o typie, będącego wynikiem użycia funkcji wirtualnych, informacja ta zastępuje „ślepą” składową. Aby się o tym przekonać, spróbuj zaznaczyć jako komentarze deklaracje **int a**, znajdujące się we wszystkich klasach powyższego programu.

Obraz funkcji wirtualnych

Do zrozumienia, co się właściwie dzieje podczas używania funkcji wirtualnych, przydatne jest graficzne przedstawienie podejmowanych niejawnie działań. Poniżej zamieszczono rysunek przedstawiający tablicę wskaźników **A[]**, zawartą w programie **Instrument4.cpp**.

² W przypadku niektórych kompilatorów mogą wystąpić niezgodności, dotyczące rozmiarów, ale zdarza się to rzadko.



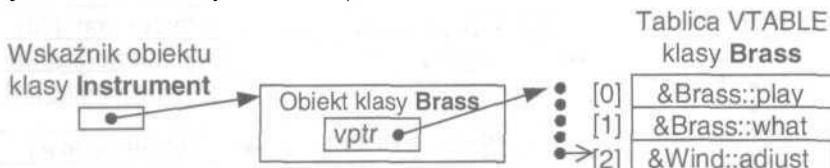
Tablica wskaźników do obiektów klasy **Instrument** nie posiada żadnej konkretnej informacji o typach — każdy z jej elementów wskazuje jedynie obiekt typu **Instrument**. Obiekty klas **Wind**, **Percussion**, **Stringed** i **Brass** należą do tej kategorii, ponieważ wszystkie te klasy są klasami pochodnymi klasy **Instrument** (a zatem mają taki sam interfejs, jak klasa **Instrument**, i mogą reagować na te same komunikaty), więc ich adresy mogą być umieszczone w tablicy. Kompilator nie wie jednak, że są one czymś więcej niż tylko obiektami klasy **Instrument**, więc gdyby decyzja należała do niego, wywołałby wersje wszystkich funkcji, zawarte w klasie podstawowej. Lecz w tym przypadku funkcje te zostały zadeklarowane ze słowem kluczowym **virtual**, więc dzieje się coś innego.

Ilekróć tworzona jest klasa zawierająca funkcje wirtualne lub wyprowadzamy klasę z innej klasy, zawierającej funkcje wirtualne, kompilator tworzy dla tej klasy unikatową tablicę VTABLE, widoczną po prawej stronie diagramu. W tablicy VTABLE umieszcza adresy wszystkich funkcji, które zostały zadeklarowane w tej klasie albo w jej klasach podstawowych. Jeżeli funkcja, która została zadeklarowana w klasie podstawowej, nie zostanie zasłonięta, to kompilator wpisuje do tablicy adres funkcji zawartej w klasie podstawowej (widać to w przypadku funkcji **adjust** tablicy VTABLE klasy **Brass**). Następnie kompilator umieszcza w klasie wskaźnik VPTR (co można było zobaczyć w programie **Sizes.cpp**). W przypadku prostego dziedziczenia, takiego jak przedstawione powyżej, każdy obiekt posiada tylko jeden wskaźnik VPTR. Wskaźnik ten musi zostać zainicjowany po to, by wskazywał adres początku właściwej tablicy VTABLE (odbywa się to w konstruktorze, co zostanie później przedstawione bardziej szczegółowo).

Kiedy wskaźnik VPTR zostanie zainicjowany za pomocą adresu odpowiedniej tablicy VTABLE, obiekt w istocie „wie”, jakiego jest typu. Ale ta „samoświadomość” jest bezużyteczna, dopóki nie zostanie on użyty w miejscu wywołania funkcji wirtualnej.

Kiedy funkcja wirtualna jest wywoływana za pośrednictwem adresu klasy podstawowej (w tej sytuacji kompilator nie posiada wszystkich informacji, niezbędnych do przeprowadzenia wcześniego wiązania), dzieje się coś szczególnego. Zamiast typowego

wywołania funkcji, które jest instrukcją **CALL** asemblera, kompilator generuje inny kod, realizujący wywołanie funkcji. Poniżej przedstawiono, w jaki sposób przebiega wywołanie funkcji **adjust()** dla obiektu klasy **Brass**, jeżeli jest dokonywane za pośrednictwem wskaźnika do obiektu klasy **Instrument** (taki sam rezultat daje referencja do obiektu klasy **Instrument**):



Kompilator zaczyna od wskaźnika do obiektu klasy **Instrument**, który wskazuje adres początku obiektu. Wszystkie obiekty klasy **Instrument**, a także klas pochodnych tej klasy, posiadają wskaźniki VPTR w tym samym miejscu (często na początku obiektu), dzięki czemu kompilator może pobrać wskaźnik VPTR obiektu. Wskazuje on początek tablicy VTABLE. Wszystkie adresy funkcji zawartych w tablicy VTABLE są ułożone w tym samym porządku, niezależnie od konkretnego typu obiektu. Adres funkcji **play()** jest pierwszy, **what()** — drugi, a funkcji **adjust()** — trzeci. Kompilator wie, że niezależnie od konkretnego typu obiektu, funkcja **adjust()** znajduje się pod adresem VPTR+2. Tak więc zamiast nakazać: „wywołaj funkcję, znajdująca się pod adresem bezwzględnym **Instrument::adjust**” (wczesne wiązanie — działanie nieprawidłowe), generuje on kod, zawierający polecenie: „wywołaj funkcję znajdująca się pod adresem **VPTR+2**”. Z uwagi na to, że pobieranie wartości wskaźnika VPTR oraz wyznaczanie aktualnego adresu funkcji odbywają się w trakcie pracy programu, wynikiem jest oczekiwane późne wiązanie. Wysyłamy do obiektu komunikat, a obiekt „domyśla się”, co powinien z nim zrobić.

Rzut oka pod maskę

Pomocne może być przeanalizowanie kodu w języku asemblera, wygenerowanego dla wywołania funkcji wirtualnej. Dzięki temu można sprawdzić, czy rzeczywiście dokonuje się w tym przypadku późne wiązanie. Poniżej przedstawiono kod wygenerowany przez kompilator dla wywołania:

```
1.adjust(1);
```

wewnątrz funkcji **f(Instrument& i)**:

```
push 1
push si
mov bx, word ptr [si]
call word ptr [bx+4]
add sp, 4
```

Argumenty wywołania funkcji języka C++, podobnie jak w języku C, są umieszczane na stosie — od prawej strony do lewej (ta kolejność jest niezbędna do obsługi zmiennej listy argumentów języka C), więc pierwszy na stosie jest umieszczany argument równy 1. W tym miejscu funkcji rejestr si (element architektury procesorów X86 firmy Intel) zawiera adres zmiennej i. Jest on również umieszczany na stosie — jako adres początku interesującego nas obiektu. Pamiętaj, że adres początku obiektu odpowiada

wartości wskaźnika **this**, który przed wywołaniem każdej funkcji składowej jest niejawnie umieszczany na stosie w charakterze argumentu. Dzięki temu funkcja ta wie, dlajakiego obiektu została wywołana. Dlatego też przed wywołaniem każdej funkcji składowej (z wyjątkiem statycznych funkcji składowych, **nieposiadających** wskaźnika **this**) na stosie jest zawsze umieszczany o jeden argument więcej niż wynosi liczba argumentów funkcji.

Następnie należy zrealizować wywołanie funkcji wirtualnej. Najpierw musi zostać pobrany wskaźnik VPTR, dzięki któremu będzie można znaleźć tablicę VTABLE. W przypadku tego kompilatora wskaźnik VPTR jeststawiany na początku obiektu, więc zawartość wskaźnika **this** odpowiada wskaźnikowi VPTR. Instrukcja znajdująca się wierszu:

```
mov bx. word ptr [si]
```

pobiera **słowo**, wskazywane przez rejestr **si** (czyli wskaźnik **this**), którym jest wskaźnik VPTR. Umieszcza ona wartość wskaźnika VPTR w rejestrze **bx**.

Wskaźnik VPTR, znajdujący się w rejestrze **bx**, wskazuje adres początku tablicy VTABLE. Jednakże wskaźnik funkcji, którą należy wywołać, znajduje się w tej tablicy nie pod adresem zerowym, tylko pod drugim (ponieważ jest to trzecia funkcja na liście). W przypadku tego modelu pamięci każdy wskaźnik funkcji zajmuje dwa bajty, więc w celu wyliczenia adresu funkcji w tablicy kompilator dodaje do VPTR wartość cztery. Warto zauważyć, że jest to stała wartość, określona podczas komplikacji, więc jedyną istotną kwestią jest to, że wskaźnik funkcji, znajdujący się pod adresem o numerze dwa, jest adresem funkcji **adjust()**. Na **szczęście**, kompilator zajmuje się wszystkimi **szczęściami** za nas, gwarantując, że wszystkie wskaźniki funkcji we wszystkich tablicach VTABLE danej hierarchii **klas** występują w tym samym porządku, niezależnie od kolejności, w jakiej mogą one być zasłaniane w klasach pochodnych.

Po wyznaczeniu adresu właściwej funkcji w tablicy VTABLE funkcja ta jest wywoływana. Za pomocą instrukcji:

```
call word ptr [bx+4]
```

równocześnie pobierany jest adres i wywoływaną funkcję, znajdująca się pod nim.

Na końcu cofany jest wskaźnik stoso w celu usunięcia wszystkich argumentów, które zostały na nim umieszczone przed wywołaniem funkcji. W kodzie asemblera, wygenerowanym dla języków C oraz C++, często można zobaczyć, że kod wywołujący funkcję usuwa jej argumenty ze stosu, ale może to przebiegać różnie, w zależności od architektury procesora i implementacji kompilatora.

Instalacja wskaźnika wirtualnego

Z uwagi na to, że wskaźnik VPTR wyznacza funkcjonowanie wirtualnej funkcji obiektu, ważne jest, aby wskaźnik ten zawsze wskazywał właściwą tablicę VTABLE. Nie chcielibyśmy nawet mieć możliwości wywołania wirtualnej funkcji, zanim wskaźnik VPTR nie zostałby prawidłowo zainicjowany. Oczywiście, miejscem, w którym inicjalizacja może być zagwarantowana, jest konstruktor, ale żaden z przykładów klas **Instrument** nie posiada konstruktörów.

Jest to przypadek, w którym niezbędnie jest utworzenie domyślnego konstruktora. W przykładach klasy **Instrument** kompilator tworzył domyślny konstruktor, który nie wykonywał żadnych działań oprócz inicjalizacji wskaźnika VPTR. Konstruktor ten jest, oczywiście, automatycznie wywoływany dla wszystkich obiektów klasy **Instrument**, zanim można cokolwiek z nimi zrobić. Dzięki temu wiadomo, że wywoływanie wirtualnych funkcji jest zawsze bezpieczne.

Konsekwencje, wynikające z automatycznej inicjalizacji wskaźnika VPTR w obrębie konstruktora, zostaną omówione w jednym z następnych podrozdziałów.

Obiekty są inne

Warto zrozumieć, że rzutowanie w góre odbywa się wyłącznie w stosunku do adresów. Jeżeli kompilator widzi obiekt, zna jego dokładny typ i dlatego (w języku C++) nie używa późnego wiązania w żadnym wywołaniu funkcji — a przynajmniej *nie musi* go stosować. Ze względu na efektywność, w razie wywołania wirtualnej funkcji w stosunku do obiektów większość kompilatorów przeprowadza wczesne łączenie, ponieważ zna ich dokładny typ. Ilustruje to poniższy przykład:

```
//: C15:Early.cpp
// Wczesne łączenie 1 funkcje wirtualne
#include <iostream>
#include <string>
using namespace std;

class Pet {
public:
    virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
    string speak() const { return "Hau!"; }
};

int main() {
    Dog ralph;
    Pet* p1 = &ralph;
    Pet& p2 = ralph;
    Pet p3;
    // Późne łączenie w obu przypadkach:
    cout << "p1->speak() = " << p1->speak() << endl;
    cout << "p2.speak() = " << p2.speak() << endl;
    // Wczesne łączenie (prawdopodobnie):
    cout << "p3.speak() = " << p3.speak() << endl;
} III-
```

Podczas wywołań **p1->speak()** oraz **p2.speak()** używany jest adres, co oznacza, że informacja nie jest pełna — zmienne **p1** i **p2** mogą reprezentować adres obiektu klasy **Pet** lub innej klasy, wyprowadzonej z klasy **Pet**, musi być więc wykorzystany mechanizm funkcji wirtualnych. W czasie wywołania **p3.speak()** nie występuje ta niejednoznaczność. Kompilator wie, że ma do czynienia z obiektem i znajego dokładny

typ, nie istnieje więc możliwość, by był to obiekt klasy pochodnej klasy Pet – to jest *dokładnie* obiekt klasy Pet. Tak więc zostanie w tym przypadku zastosowane wcześnie wiązanie. Jeżeli jednak kompilator nie chce się zbytnio trudzić, to może w tym przypadku użyć również późnego wiązania i rezultat będzie taki sam.

Dlaczego funkcje wirtualne?

Możesz w tym miejscu zapytać: „Skoro ta technika jest tak ważna i umożliwia zawsze wywoływanie tych »właściwych« funkcji, to dlaczego stanowi ona opcję? Dlaczego muszę w ogóle o niej wiedzieć?”.

To dobre pytanie, a odpowiedź na nie stanowi element podstawowej filozofii języka C++: „Ponieważ nie jest tak efektywna”. W przedstawionym wcześniej fragmencie programu w języku asemblera zamiast jednego, prostego wywołania funkcji, znajdującej się pod bezwzględnym adresem, widnieją dwie -- bardziej skomplikowane -- instrukcje języka asemblera, niezbędne do wywołania funkcji wirtualnej. Wymaga to zarówno większej ilości pamięci, jak i dłuższego czasu wykonania.

W przypadku niektórych języków obiektowych przyjęto założenie, że późne wiązanie ma dla programowania obiektowego tak fundamentalne znaczenie, że należy je realizować zawsze. W związku z tym nie powinno być ono opcjonalne, a użytkownik nie musi w ogóle o nim wiedzieć. Jest to decyzja projektowa, podjęta podczas tworzenia języka i sprawdza się ona w przypadku wielu języków programowania³. Jednakże język C++ jest spadkobiercą języka C, w którym efektywność ma podstawowe znaczenie. Język C powstał bowiem po to, by zastąpić język asemblera podczas implementacji systemu operacyjnego (w ten sposób czyniąc ten system operacyjny — Unix — znacznie bardziej przenośnym od jego poprzedników). Jednym z głównych powodów powstania języka C++ był zamiar uczynienia programowania w języku C bardziej efektywnym⁴. A pierwszym pytaniem programisty języka C po zetknięciu z językiem C++ jest: „Jaki to będzie miało wpływ na wielkość i szybkość programu?”. Gdyby odpowiedź brzmiała: „Wszystko po staremu, tylko podczas wywołania funkcji zawsze występuje niewielki narzut”, wielu pozostałoby przy języku C, nie zamieniając go na C++. Ponadto nie byłoby możliwe stosowanie funkcji inline, ponieważ funkcje wirtualne muszą mieć adres, który trzeba umieścić w tablicy VTABLE. Tak więc funkcje wirtualne stanowią opcję; domyślną opcję języka są funkcje, które nie są wirtualne, co oznacza rozwiązanie gwarantujące większą szybkość. Stroustrup stwierdził, że przywiecała mu następująca myśl: „Jeżeli czegoś nie używasz, nie płaci za to”.

Tak więc występowanie słowa kluczowego **virtual** umożliwia regulację efektywności. Jednakże podczas projektowania klas nie należy przejmować się względami efektywności. Jeżeli zamierzasz wykorzystywać polimorfizm, to używaj wszędzie

³ Na przykład Smalltalk, Java i Python wykorzystują to podejście z powodzeniem.

⁴ W Bell Labs, gdzie powstał język C++, pracuje bardzo wielu programistów języka C. Zwiększenie ich efektywności, choćby w najmniejszym stopniu, pozwala firmie na zaoszczędzenie milionów.

funkcji wirtualnych. Tylko w przypadku poszukiwania sposobów przyspieszenia kodu trzeba znaleźć funkcje, które można przekształcić w funkcje nie będące funkcjami wirtualnymi (a w takich przypadkach zazwyczaj znacznie więcej można uzyskać w innych obszarach — dobry program profilujący lepiej wskaże wąskie gardła niż programista, opierający się na przypuszczeniach).

Praktyka dowodzi, że wpływ przejęcia na język C++ na wielkość i szybkość programów sięga, w porównaniu z językiem C, około 10 procent, a częstotliwość tego znacznie mniejsza. Powodem sprawiającym, że możliwe jest uzyskanie nawet lepszej efektywności czasowej i pamięciowej, jest to, że w języku C++ można zaprojektować program w taki sposób, by był mniejszy i szybszy.

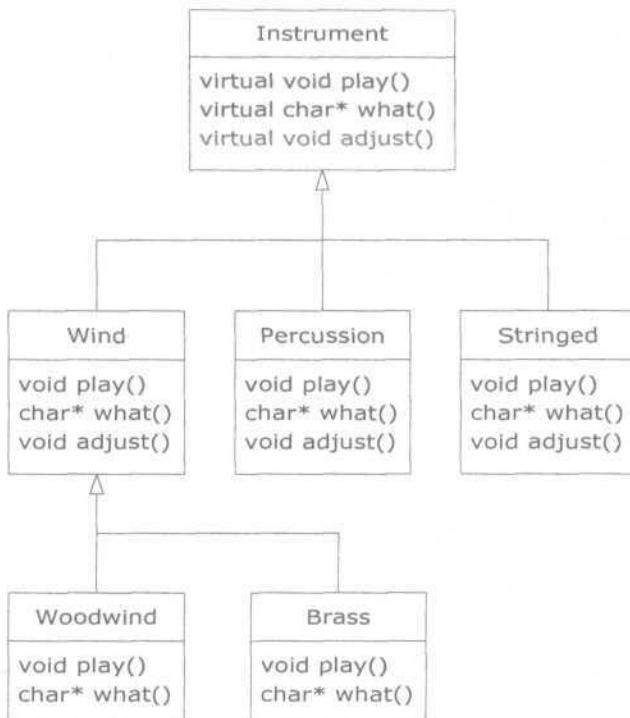
Abstrakcyjne klasy podstawowe i funkcje czysto wirtualne

W trakcie projektowania nierzadko występuje potrzeba, by klasa podstawowa stanowiła *wyłącznie* interfejs dla swoich klas pochodnych. Oznacza to, że nie chcemy, aby ktokolwiek tworzył obiekty klasy *podstawowej*, a jedynie dopuszczać wykorzystanie rzutowania w góre, umożliwiające użycie jej interfejsu. Uzyskujemy to, czyniąc klasę *abstrakcyjną*, co zachodzi w przypadku, gdy utworzy się w niej przynajmniej jedną funkcję czysto wirtualną (ang. *pure virtual function*). Funkcje czysto wirtualne można rozpoznać po tym, że używają słowa kluczowego *virtual*, a na ich końcu występuje = 0. Kompilator nie pozwoli nikomu na utworzenie obiektu takiej klasy. Klasy abstrakcyjne są narzędziem, umożliwiającym narzucenie pewnego modelu.

Gdy dziedziczoną jest klasa abstrakcyjna, to w klasie pochodnej muszą zostać zaimplementowane wszystkie jej czysto wirtualne funkcje, gdyż w przeciwnym wypadku klasa pochodna również stanie się klasą abstrakcyjną. Utworzenie funkcji czysto wirtualnej pozwala na umieszczenie jej jako funkcji składowej w interfejsie klasy, bez konieczności tworzenia (być może pozbawionego sensu) kodu, stanowiącego ciało tej funkcji. Równocześnie użycie funkcji czysto wirtualnych wymusza na klasach pochodnych dostarczenie ich definicji.

We wszystkich *przykładach* dotyczących instrumentów funkcje zawarte w klasie podstawowej **Instrument** były jedynie „wypełniaczami”. Gdyby zostały one kiedykolwiek wywołane, oznaczałoby to, że coś zadziałało nieprawidłowo. Zadaniem klasy **Instrument** jest bowiem utworzenie wspólnego interfejsu dla wszystkich wyprowadzonych z niej klas pochodnych.

Określenie wspólnego interfejsu następuje jedynie z tej przyczyny, że w ten sposób może on zostać odmiennie wyrażony dla każdej klasy pochodnej. Tworzy on podstawowy szablon, wyznaczający to, co jest wspólne dla wszystkich klas pochodnych — nic więcej. Tak więc klasa **Instrument** jest odpowiednim kandydatem do klasy abstrakcyjnej. Klasę abstrakcyjną można utworzyć, gdy zamierza się jedynie operować zbiorem klas za pośrednictwem wspólnego interfejsu, ale interfejs ten nie musi posiadać implementacji (a przynajmniej nie musi ona być pełna).



Obiekty klas, wyrażających takie pojęcia, jak klasa **Instrument**, zrealizowana w postaci klasy abstrakcyjnej, prawie nigdy nie posiadają znaczenia. Oznacza to, że klasa **Instrument** służy jedynie do określenia interfejsu, a nie konkretnej implementacji. Tworzenie obiektu, który byłby tylko instrumentem, nie ma zatem sensu i zapewne chcielibyśmy uniemożliwić użytkownikowi tworzenie takich obiektów. Można to uzyskać, definiując wszystkie wirtualne funkcje klasy **Instrument** w taki sposób, by drukowały one komunikat o błędzie — to jednak spowodowałoby opóźnienie pojawienia się informacji o błędzie aż do momentu uruchomienia programu, a ponadto wymagałoby gruntownego przetestowania przez użytkownika. Znacznie lepiej jest wykryć ten problem już na etapie komilacji.

Poniżej została przedstawiona składnia deklaracji czysto wirtualnej funkcji:

```
virtual void f() = 0;
```

W ten sposób należy zwrócić się do komplatora, by zarezerwował funkcji miejsce w tablicy VTABLE, ale nie powinien umieszczać w nim żadnego konkretnego adresu. Jeżeli choćby jedna funkcja w klasie została zadeklarowana jako funkcja czysto wirtualna, to tablica VTABLE nie jest kompletna.

Jak więc reaguje komplator w przypadku, gdy ktoś próbuje utworzyć obiekt klasy, której tablica VTABLE jest niekompletna? Nie może bezpiecznie utworzyć obiektu klasy abstrakcyjnej, zgłasza więc komunikat o błędzie. Dzięki temu komplator gwarantuje „czystość” abstrakcyjnej klasy. Tworząc klasę abstrakcyjną, mamy pewność, że klient-programista nie użyje jej w niepoprawny sposób.

Przedstawiony poniżej program **Instrument4.cpp** został zmodyfikowany w taki sposób, by wykorzystywał funkcje czysto wirtualne. Ponieważ klasa **Instrument** zawiera obecnie wyłącznie funkcje czysto wirtualne, nazywamy ją *klassą czysto abstrakcyjną* (ang. *pure abstract class*):

```
/I: C15:Instrument5.cpp
// Czysto abstrakcyjne klasy podstawowe
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Itd.

class Instrument {
public:
    // Funkcje czysto wirtualne:
    virtual void play(note) const = 0;
    virtual char* what() const = 0;
    // Zakładamy, że funkcja modyfikuje obiekt:
    virtual void adjust(int) = 0;
};

// Pozostała część programu pozostała taka sama...

class Wind : public Instrument {
public:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
    char* what() const { return "Wind"; }
    void adjust(int) {}
};

class Percussion : public Instrument {
public:
    void play(note) const {
        cout << "Percussion::play" << endl;
    }
    char* what() const { return "Percussion"; }
    void adjust(int) {}
};

class Stringed : public Instrument {
public:
    void play(note) const {
        cout << "Stringed::play" << endl;
    }
    char* what() const { return "Stringed"; }
    void adjust(int) {}
};

class Brass : public Wind {
public:
    void play(note) const {
        cout << "Brass::play" << endl;
    }
    char* what() const { return "Brass"; }
};
```

```
class Woodwind : public Wind {
public:
    void play(note) const {
        cout << "Woodwind::play" << endl;
    }
    char* what() const { return "Woodwind"; }
};

// Funkcja taka sama, jak poprzednio:
void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

// Nowa funkcja:
void f(Instrument& i) { i.adjust(l); }

int main() {
    Wind flute;
    Percussion drum;
    Stringed violin;
    Brass flugelhorn;
    Woodwind recorder;
    tune(flute);
    tune(drum);
    tune(violin);
    tune(flugelhorn);
    tune(recorder);
    f(flugelhorn);
} //:~
```

Funkcje czysto wirtualne są przydatne, ponieważ deklarują one jawnie abstrakcyjność klas, informując zarówno użytkownika, jak i kompilator o tym, w jakim celu zostały utworzone.

Warto zauważyć, że czysto wirtualne funkcje uniemożliwiają przekazywanie *przez wartość* argumentów typów abstrakcyjnych klas. Jeszcze również sposób na uniknięcie *okrajania obiektów* (krótko opisanego w dalszej części rozdziału). Dzięki uczynieniu klasy abstrakcyjnej możesz mieć pewność, że w trakcie rzutowania w góre na tę klasę jest zawsze używany wskaźnik lub referencja.

W prawdzie jedna czysto wirtualna funkcja powoduje, że tablica VTABLE jest niekompletna, lecz wcale to nie oznacza, że nie możemy zdefiniować ciał niektórych pozostałych funkcji. Często będziemy chcieli wywołać wersję funkcji, zawartą w klasie podstawowej, nawet jeśli jest ona funkcją wirtualną. Dobrym pomysłem jest zawsze umieszczanie wspólnego kodu możliwie jak najbliżej klasy *główniej* w hierarchii klas. Nie tylko zapewnia to oszczędność miejsca, ale również ułatwia propagację wprowadzanych zmian.

Czysto wirtualne definicje

Możliwe jest utworzenie w klasie podstawowej definicji czysto wirtualnej funkcji. Kompilator nadal nie pozwoli na powstanie obiektów takiej abstrakcyjnej klasy podstawowej, a czysto wirtualne funkcje muszą zostać zdefiniowane w klasach pochodnych, aby można było tworzyć ich obiekty. Jednak funkcja taka może zawierać wspólny fragment kodu, wywoływany przez niektóre lub wszystkie definicje klas pochodnych, dzięki czemu nie trzeba będzie powielać go w każdej funkcji.

Poniżej zamieszczono przykład zawierający definicje czysto wirtualnych funkcji:

```
//: C15:PureVirtualDefinitions.cpp
// Definicje czysto wirtualnych funkcji
// Klasa podstawowa
#include <iostream>
using namespace std;

class Pet {
public:
    virtual void speak() const = 0;
    virtual void eat() const = 0;
    // Czysto wirtualna funkcja nie może być funkcją inline:
    //! virtual void sleep() const = 0 {}

};

// W porządku, funkcja nie jest zdefiniowana jako funkcja inline
void Pet::eat() const {
    cout << "Pet::eat()" << endl;
}

void Pet::speak() const {
    cout << "Pet::speak()" << endl;
}

class Oog : public Pet {
public:
    // Wykorzystanie wspólnego kodu klasy Pet
    void speak() const { Pet::speak(); }
    void eat() const { Pet::eat(); }

};

int main() {
    Dog simba; // Pies Richarda
    simba.speak();
    simba.eat();
} //:-
```

Pozycje **speak** i **eat** tablicy VTABLE klasy **Pet** pozostają nadal niezapełnione, ale istnieją funkcje o takich nazwach, które można wywołać w klasach pochodnych.

Inna korzyść, wynikająca z możliwości definicji funkcji czysto wirtualnych, polega na tym, że pozwala ona na przekształcenie funkcji wirtualnej w funkcję **czysto wirtualną**, bez konieczności zmiany istniejącego kodu (**jest** to również sposób na znalezienie klas, które nie **zasłaniają tej funkcji**).

Dziedziczenie i tablica VTABLE

Można sobie wyobrazić, co nastąpi w przypadku, gdy tworzy się klasę pochodną i **zasłania** niektóre z wirtualnych funkcji. Dla nowo powstałej klasy kompilator tworzy tablicę VTABLE i umieszcza w niej adresy nowych funkcji, przepisując do niej również adresy tych wszystkich wirtualnych funkcji **klasy** podstawowej, które nie zostały zasłonięte. Tak czy inaczej, każdy obiekt, który można utworzyć (czyli obiekt klasy nie zawierającej funkcji **czysto wirtualnych**), ma do dyspozycji pełny zestaw adresów funkcji, zawarty w tablicy VTABLE. Dzięki temu nigdy nie może wystąpić sytuacja, polegająca na wywołaniu funkcji, której adresu nie ma w tablicy (co byłoby katastrofalne).

Co jednak dzieje się w przypadku, gdy używamy dziedziczenia, a następnie dodajemy *do klasy pochodnej* nowe funkcje wirtualne? Oto prosty przykład:

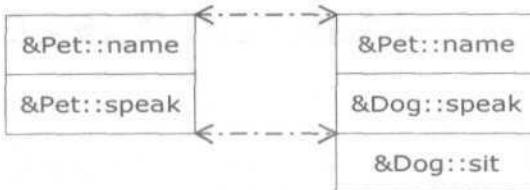
```
//: C15:AddingVirtuals.cpp
// Dodawanie funkcji wirtualnych podczas dziedziczenia
#include <iostream>
#include <string>
using namespace std;

class Pet {
    string pname;
public:
    Pet(const string& petName) : pname(petName) {}
    virtual string name() const { return pname; }
    virtual string speak() const { return ""; }
};

class Dog : public Pet {
    string name;
public:
    Dog(const string& petName) : Pet(petName) {}
    // Nowa wirtualna funkcja w klasie Dog:
    virtual string sit() const {
        return Pet::name() + " siedzi";
    }
    string speak() const { // Zastąpienie
        return Pet::name() + " mówi 'Hau!'";
    }
};

int main() {
    Pet* p[] = {new Pet("zwyczajny"), new Dog("bob")};
    cout << "p[0]->speak() = "
        << p[0]->speak() << endl;
    cout << "p[1]->speak() = "
        << p[1]->speak() << endl;
    //! cout << "p[1]->sit() = "
    //!     << p[1]->sit() << endl; // Niedozwolone
} //:~
```

Klasa Pet zawiera dwie funkcje wirtualne — **speak()** i **name()**. Klasa Dog dodaje trzecią wirtualną funkcję **sit()**, a także zasłania znaczenie funkcji **speak()**. W zrozumieniu tego, co się w tym przypadku dzieje, pomocny będzie diagram. Poniżej przedstawiono tablice VTABLE, utworzone przez kompilator dla klas Pet i Dog:



Warto podkreślić, że kompilator umieszcza adres funkcji `speak()` — zarówno w tablicy VTABLE klasy Dog, jak i w tablicy VTABLE klasy Pet — na tej samej pozycji. Podobnie by się stało w przypadku, gdyby z klasy Dog została wyprowadzona klasa Pug — jej wersja funkcji `sit()` zostały umieszczone w tablicy VTABLE tej klasy na identycznej pozycji, na której znajduje się ona w tablicy VTABLE klasy Dog. Dzieje się tak dlatego, że kompilator generuje kod, który do wyboru wirtualnej funkcji używa prostego, liczbowego przesunięcia względem początku tablicy VTABLE (co można było zobaczyć w przykładzie, prezentującym kod w języku asemblera). Niezależnie od konkretnego podtypu, do którego należy Obiekt, jego tablica VTABLE jest wypełniona zawsze w ten sam sposób, dzięki czemu wywoływanie wirtualnych funkcji odbywa się zawsze identycznie.

Jednak w tym przypadku kompilator ma do czynienia wyłącznie ze wskaźnikiem do obiektu klasy podstawowej. Klasa ta posiada tylko funkcje `speak()` i `name()`, więc są one jedynymi funkcjami, na których wywołanie pozwoli kompilatorowi. Skąd zresztą miałyby wiedzieć, że jest to obiekt klasy **Dog**, skoro posiada tylko wskaźnik do obiektu klasy podstawowej? Równie dobrze wskaźnik ten może wskazywać obiekt jakiegoś innego typu, nie posiadającego funkcji `sit()`. Obiekt ten może, ale nie musi, posiadać na tej pozycji swojej tablicy VTABLE adres jakiejś innej funkcji — ale w żadnym z tych przypadków nie chodziło nam o wywołanie wirtualnej funkcji, której adres znajduje się w tej tablicy. **Kompilator** nie pozwala więc na wywoływanie wirtualnych funkcji, które istnieją wyłącznie w klasach pochodnych.

Zdarzają się jeszcze mniej typowe przypadki, w których wiemy, że wskaźnik w rzeczywistości wskazuje obiekt jakiejś konkretnej podklasy. Jeżeli chcemy wywołać funkcję, która istnieje wyłącznie w danej klasie pochodnej, musimy zastosować rzutowanie wskaźnika. Możemy uniknąć komunikatu o błędzie, generowanego w przypadku poprzedniego programu, zapisując to w następujący sposób:

`((Dog*)p[1])->sit()`

W tym przypadku można stwierdzić, że wskaźnik `p[1]` wskazuje obiekt klasy Dog, ale na ogół nie posiadamy takiej wiedzy. Jeżeli problem jest określony w taki sposób, że musimy znać dokładnie typy wszystkich obiektów, powinniśmy go przemyśleć, ponieważ oznacza to, że prawdopodobnie nie wykorzystujemy poprawnie funkcji wirtualnych. Istnieją jednak takie sytuacje, w których projekt działa najlepiej (albo nie ma innej możliwości), gdy znane są typy wszystkich **obiektów**, przechowywanych w kontenerze ogólnego przeznaczenia. Jest to problem *identyfikacji typów podczas pracy programu* (ang. *run-time type identification* — RTTI).

Identyfikacja typów podczas pracy programu dotyczy rzutowania wskaźników klasy podstawowej *w dół* — na wskaźniki klasy pochodnej (pojęcia „w góre” i „w dół” odnoszą się do typowego diagramu klas, w którym klasa podstawowa znajduje się na

górze). Rzutowanie *w górę* odbywa się automatycznie, bez konieczności wymuszania, ponieważ jest ono **zupełnie** bezpieczne. Rzutowanie *w dół* nie jest bezpieczne, gdyż podczas komplikacji nie jest dostępna informacja, dotycząca rzeczywistych typów rzutowanych obiektów. Rzutowanie to wymaga więc precyzyjnej wiedzy, dotyczącej typu obiektu. Jeżeli dokona się rzutowania na niewłaściwy typ, spowoduje to problemy.

Identyfikacja typów podczas pracy programu została opisana w dalszej części rozdziału. W drugim tomie książki znajduje się również rozdział, poświęcony temu zagadnieniu.

Okrajanie obiektów

Gdy wykorzystywany jest **polimorfizm**, występuje wyraźna różnica pomiędzy przekazywaniem adresu obiektów a przekazywaniem ich przez wartość. We wszystkich przedstawionych do tej pory przykładach (a także w tych, które zostaną jeszcze przedstawione) przekazywane są adresy, a nie obiekty. Zrobiono tak z uwagi na to, że wszystkie adresy mają identyczną **wielkość⁵**, więc przekazywanie adresu obiektu klasy pochodnej (który jest zazwyczaj większym obiektem) odbywa się tak samo, jak przekazywanie adresu obiektu klasy podstawowej (który jest na ogół mniejszym obiektem). Jak już wspomniano, jest to właśnie celem używania **polimorfizmu** — kod, który operuje na obiektach typu podstawowego, może w przezroczysty sposób operować również na obiektach typów pochodnych.

Jeżeli dokona się rzutowania w górę na obiekt, a nie na wskaźnik czy referencję, zdarzy się coś zdumiewającego — obiekt zostanie „okrojony” w taki sposób, aby to, co z niego zostanie, stanowiło podobiekt, odpowiadający typowi, do którego dokonywane było rzutowanie. Poniższy przykład pokazuje, co się dzieje podczas **okrajania** obiektu:

```
//: C15:ObjectSlicing.cpp
#include <iostream>
#include <string>
using namespace std;

class Pet {
    string pname;
public:
    Pet(const string& name) : pname(name) {}
    virtual string name() const { return pname; }
    virtual string description() const {
        return "To jest " + pname;
    }
};

class Oog : public Pet {
    string favoriteActivity;
public:
    Oog(const string& name, const string& activity)
        : Pet(name), favoriteActivity(activity) {}

    string name() const override {
        return "Oog: " + Pet::name();
    }

    string description() const override {
        return "To jest " + Pet::description();
    }
}
```

⁵ W rzeczywistości, nie dla wszystkich komputerów, poszczególne wskaźniki mają takie same wielkości. W kontekście przedstawionej dyskusji, można je jednak traktować w taki sposób, jakby były takie same.

```

string description() const {
    return Pet::name() + " lubi " +
        favoriteActivity;
}

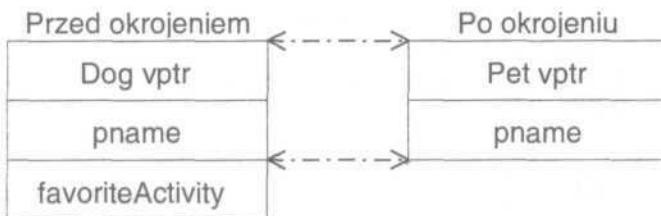
void describe(Pet p) { // Funkcja okraja obiekt
    cout << p.description() << endl;
}

int main() {
    Pet p("Alfred");
    Dog d("Puszek", "spac");
    describe(p);
    describe(d);
} //:-

```

Obiekt typu **Pet** jest przekazywany *przez wartość* funkcji **describe()**. Funkcja ta wywołuje następnie dla obiektu klasy **Pet** wirtualną funkcję **description()**. Można się było spodziewać, że pierwsze wywołanie tej funkcji, w funkcji **main()**, spowoduje wyprowadzenie tekstu: „To jest Alfred”, a drugie — „Puszek lubi spać”. W rzeczywistości, oba wywołania używają wersji funkcji **description()**, znajdującej się w klasie podstawowej.

W trakcie działania programu zachodzą dwa procesy. Po pierwsze, ponieważ funkcja **describe()** pobiera *obiekt* (a nie wskaźnik czy referencję), wszelkie wywołania tej funkcji powodują umieszczenie na stosie obiektu wielkości obiektu klasy **Pet**, a następnie, po powrocie z wywołanej funkcji, usunięcie go ze stosu. Oznacza to, że jeżeli funkcji **describe()** zostanie przekazany obiekt klasy pochodnej **Pet**, kompilator pozwoli na to, ale skopiuje jedynie tę część obiektu, którą stanowi obiekt klasy **Pet**. Powoduje to *okrejenie* części obiektu klasy pochodnej, jak przedstawia poniższy rysunek:



Zdziwienie może wywołać również wywołanie funkcji wirtualnej. Funkcja **Dog::description()** wykorzystuje zarówno część obiektu, dziedziczoną po klasie **Pet** (część ta nadal istnieje), jak i dodaną w klasie **Dog**, która już nie istnieje, gdyż została „*odkrojona*”. Co w takim razie dzieje się podczas wywołania funkcji wirtualnej?

Obyło się bez katastrofy, ponieważ obiekt został przekazany przez wartość. Kompilator znał dokładnie typ obiektu, ponieważ wymuszone zostało *przekształcenie* obiektu klasy pochodnej w obiekt klasy podstawowej. Podczas przekazywania przez wartość został użyty konstruktor kopiący klasy **Pet**. Zainicjował on wskaźnik **VPTR** adresem tablicy **VTABLE** klasy **Pet**, a następnie skopiował jedynie tę część obiektu, która została odziedziczona z klasy **Pet**. Klasa **Pet** nie posiada konstruktora kopiącego, więc został on wygenerowany przez kompilator. Niezależnie od pierwotnego typu, na skutek okrojenia obiekt naprawdę staje się obiektem typu **Pet**.

Okrajanie obiektu, które odbywa się podczas jego kopiowania do nowo utworzonego obiektu, w rzeczywistości powoduje utratę jego części, zamiast zmieniać po prostu znaczenie adresu, jak w przypadku wskaźników lub referencji. Z tego powodu nie używa się zbyt często rzutowania w górę na obiekt — w rzeczywistości, jest to jedna z tych sytuacji, na której trzeba zwracać uwagę i zapobiegać ich wystąpieniu. Warto podkreślić, że gdyby w powyższym przykładzie funkcja `description()` była w klasie podstawowej funkcją czysto wirtualną (co nie jest pozbawione logiki, gdyż tak naprawdę nie wykonuje ona w klasie podstawowej żadnych działań), to kompilator mógłby zapobiec okrajaniu obiektów, ponieważ nie pozwoliłby na „utworzenie” obiektu typu podstawowego (co ma miejsce w trakcie rzutowania w górę przez wartość). Jest to być może najważniejsza zaleta funkcji czysto wirtualnych — uniemożliwienie okrajania obiektów dzięki generowaniu błędów w trakcie komplikacji.

Przeciążanie i zasłanianie

Jak wiadomo na podstawie lektury rozdziału 14., przeddefiniowanie funkcji przeciążonej w klasie podstawowej powoduje ukrycie wszystkich pozostałych wersji tej funkcji, znajdujących się w klasie podstawowej. Gdy dotyczy to funkcji wirtualnych, sytuacja wygląda nieco inaczej. Przyjrzymy się zmodyfikowanej wersji programu `NameHiding.cpp`, pochodzącego z rozdziału 14.:

```
//: C15:NameHiding2.cpp
// Funkcje wirtualne ograniczają przeciążanie
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
    virtual int f() const {
        cout << "Base::f()\n";
        return 1;
    }
    virtual void f(string) const {}
    virtual void g() const {}
};

class Derived1 : public Base {
public:
    void g() const {}
};

class Derived2 : public Base {
public:
    // Przeciążenie funkcji wirtualnej:
    int f() const {
        cout << "Derived2::f()\n";
        return 2;
    }
};
```

```

class Derived3 : public Base {
public:
    // Nie można zmieniać typu zwracanej wartości:
    //! void f() const{ cout << "Derived3::f()\n"; }
};

class Derived4 : public Base {
public:
    // Zmiana listy argumentów:
    int f(int) const {
        cout << "Derived4::f()\n";
        return 4;
    }

    int main() {
        string s("czesc");
        Derived1 d1;
        int x = d1.f();
        d1.f(s);
        Derived2 d2;
        x = d2.f();
        //! d2.f(s); // Wersja dla łańcuchów została ukryta
        Derived4 d4;
        x = d4.f();
        //! x = d4.f(); // Wersja f() została ukryta
        //! d4.f(s); // Wersja dla łańcuchów została ukryta
        Base& br = d4; // Rzutowanie w gore
        //! br.f(); // Wersja z klasą pochodnej nie jest dostępna
        br.f(); // Dostępna wersja z klasy podstawowej
        br.f(s); // Dostępna wersja z klasy podstawowej
    } ///-
}

```

Pierwszą kwestią godną odnotowania jest to, że w klasie **Derived3** kompilator nie dopuszcza zmiany typu wartości, zwracanej przez zasłoniętą funkcję (pozwoliłby na to, gdyby funkcja **f()** nie była funkcją wirtualną). Jest to istotne ograniczenie, ponieważ kompilator musi zagwarantować możliwość polimorficznego wywołania funkcji za pośrednictwem klasy podstawowej. Gdyby zaś klasa podstawowa oczekiwała zwrócenia przez funkcję **f()** wartości typu **int**, wersja funkcji **f()**, zawarta w klasie pochodnej, musiałaby utrzymać to w mocy, gdyż w przeciwnym razie nastąpiłaby katastrofa.

Zasada przedstawiona w rozdziale 14. nadal obowiązuje — jeżeli zostanie zasłonięta jedna z przeciążonych funkcji składowych klasy podstawowej, to ich pozostałe przeciążone wersje staną się niewidoczne w klasie pochodnej. Kod testujący klasę **Derived4**, zawarty w funkcji **main()**, pokazuje, co dzieje się nawet w przypadku, gdy nowa wersja funkcji **f()** nie zasłania interfejsu istniejącej funkcji wirtualnej — obie, znajdującej się w klasie podstawowej, wersje funkcji **f()** są zakrywane przez funkcję **f(int)**. Jeżeli jednak dokona się rzutowania w góre obiektu **d4** na klasę **Base**, to dostępne są wówczas jedynie wersje funkcji zawarte w klasie podstawowej (ponieważ ich obecność jest gwarantowana przez interfejs klasy podstawowej). Z kolei jej wersje zawarte w klasie pochodnej nie będą dostępne (ponieważ nie zostały one wyspecyfikowane w interfejsie klasy podstawowej).

Zmiana typu zwracanej wartości

Przedstawiona powyżej klasa **Derived3** sugeruje, że podczas **zasłaniania** nie można zmienić typu wartości zwracanej przez funkcję. Na ogół jest to zgodne z prawdą, lecz istnieje szczególny przypadek, w którym można w pewnym stopniu zmodyfikować typ zwracanej wartości. Jeżeli funkcja zwraca wskaźnik lub referencję do obiektu klasy podstawowej, to **przedefiniowana** wersja funkcji może zwracać wskaźnik lub referencję klasy pochodnej. Ilustruje to poniższy przykład:

```
//: C15:VariantReturn.cpp
// Zwracanie wskaźnika lub referencji
// do typu pochodnego w czasie zasłaniania
#include <iostream>
#include<string>
using namespace std;

class PetFood {
public:
    virtual string foodType() const = 0;
};

class Pet {
public:
    virtual string type() const = 0;
    virtual PetFood* eats() = 0;
};

class Bird : public Pet {
public:
    string type() const { return "Ptak"; }
    class BirdFood : public PetFood {
public:
    string foodType() const {
        return "nasiona";
    }
    // Rzutowanie w górę do typu podstawowego:
    PetFood* eats() { return &bf; }
private:
    BirdFood bf;
};

class Cat : public Pet {
public:
    string type() const { return "Kot"; }
    class CatFood : public PetFood {
public:
    string foodType() const { return "ptaki"; }
};
// Zwracany jest dokładny typ:
CatFood* eats() { return &cf; }
private:
    CatFood cf;
};
```

```

int main() {
    Bird b;
    Cat c;
    Pet* p[] = { &b, &c, };
    for(int i = 0; i < sizeof p / sizeof *p; i++)
        cout << p[i]->type() << " je "
            << p[i]->eats()->foodType() << endl;
    // Funkcja może zwrócić dokładny typ:
    Cat::CatFood* cf = c.eats();
    Bird::BirdFood* bf;
    // Funkcja nie może zwrócić dokładnego typu:
    // bf = b.eats();
    // Trzeba rzutować w dół:
    bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} //:-

```

Funkcja składowa **Pet::eats()** zwraca wskaźnik do obiektu klasy **PetFood**. W klasie **Bird** ta funkcja składowa jest przeciążona dokładnie tak, jak w klasie podstawowej, włącznie z typem zwracanej wartości. Oznacza to, że funkcja **Bird::eats()** dokonuje rzutowania w górę klasy **BirdFoot** na klasę **PetFood**.

Jednakże w klasie **Cat** typem wartości zwracanej przez funkcję **eats()** jest wskaźnik do obiektu klasy **CatFood**, będącej klasą wyprowadzoną z klasy **PetFood**. Fakt, że zwracany typ jest dziedziczony po typie zwracanym przez funkcję klasy podstawowej, jest jedynym powodem, dla którego program kompliluje się poprawnie. Warunek jest zatem zawsze spełniony — funkcja **eats()** zawsze zwraca wskaźnik do obiektu klasy **PetFood**.

Z punktu widzenia polimorfizmu nie wydaje się to konieczne. Dlaczego nie rzutować po prostu w górę wszystkich zwracanych typów na typ **PetFood***, tak jak funkcja **Bird::eats()**? Jest to na ogół dobre rozwiązanie, ale różnicą jest widoczna pod koniec funkcji **main()** — funkcja **Cat::eats()** potrafi zwrócić dokładny typ, podczas gdy wartość zwracana przez funkcję **Bird::eats()** musi być rzutowana w dół, na odpowiedni typ.

Tak więc możliwość zwrócenia dokładnego typu ma trochę bardziej ogólny charakter i nie wiąże się z utratą informacji o konkretnym typie, powodowaną przez automatyczne rzutowanie w górę. Jednak zwracanie typu podstawowego zwykle rozwiązuje problemy, więc jest to nieco bardziej wyspecjalizowana właściwość.

Funkcje wirtualne a konstruktory

Kiedy tworzony jest obiekt zawierający funkcje wirtualne, jego wskaźnik VPTR musi zostać zainicjowany za pomocą adresu właściwej tablicy VTABLE. Musi to zostać wykonane, zanim zaistnieje jakakolwiek możliwość wywołania wirtualnej funkcji. Jak można się tego spodziewać, ponieważ to konstruktor wykonuje pracę polegającą na powołaniu obiektu do życia, jego zadaniem jest również ustalenie wartości wskaźnika VPTR. Kompilator potajemnie umieszcza na początku konstruktora kod, inicjalizujący wskaźnik VPTR. Poza tym, zgodnie z tym, co napisano w rozdziale 14.,

jeżeli dla jakiejś klasy nie zostaniejawnie utworzony konstruktor, to wygeneruje go kompilator. Jeżeli klasa zawiera wirtualne funkcje, to wygenerowany konstruktor będzie zawierał odpowiedni kod, inicjujący wskaźnik VPTR. Wynika z tego szereg konsekwencji.

Pierwsza dotyczy efektywności. Powodem udostępnienia funkcji **inline** było umożliwienie zmniejszenia narzutu, związanego z wywołaniem małych funkcji. Gdyby język C++ nie zawierał funkcji **inline**, to do tworzenia tych „makr” byłby wykorzystywany preprocesor. Jednakże preprocesor nie wie nic na temat dostępu do klas i w związku z tym nie można by go użyć do utworzenia makroinstrukcji, będących funkcjami składowymi. Ponadto w przypadku konstruktorów, które musiałyby posiadać ukryty kod wstawiony przez kompilator, makroinstrukcje w ogóle by nie działały.

Kiedy „poluje” się na zawarte w programie luki, związane z efektywnością, trzeba wiedzieć, że kompilator wstawia do funkcji konstruktora niewidoczny kod. Musi on nie tylko zainicjować wskaźnik VPTR, ale również sprawdzić, jaka jest wartość wskaźnika **this** (w przypadku gdyby **operator new** zwrócił wartość zerową), oraz wywołać konstruktory klas podstawowych. Wszystko to wydaje się niezgodne z wyobrażeniem, że wywołanie konstruktora polega tylko na wywołaniu niewielkiej funkcji, do której można wstawić w kod. W szczególnym przypadku wielkość konstruktora może zniweczyć oszczędności, związane z uniknięciem wywołania funkcji. Jeżeli używa się wielu wywołań konstruktorów wstawionych do kodu, może on znacznie się rozrosnąć; nie przyniesie to korzyści, związanych z poprawą szybkości.

Oczywiście, nie oznacza to, że należy od razu uczynić wszystkie niewielkie konstruktory funkcjami, które nie są wstawiane do kodu — ponieważ znacznie łatwiej jest napisać jąko funkcje **inline**. Lecz poszukując sposobów przyspieszenia programu, warto rozważyć rezygnację z wstawiania konstruktorów do kodu.

Kolejność wywoływania konstruktorów

Kolejną ciekawą kwestią, związaną z konstruktorami i funkcjami wirtualnymi, jest kolejność wywoływania konstruktorów oraz sposób, w jaki wirtualne wywołania realizowane są w obrębie konstruktorów.

Wszystkie konstruktory klas podstawowych są zawsze wywoływane wewnątrz konstruktorów klas pochodnych. Jeszcze logiczne, ponieważ konstruktory mają specjalne zadanie — nadzorują poprawne tworzenie obiektów. Konstruktor klasy pochodnej ma dostęp jedynie do własnych **składowych**, a nie do składowych zawartych w klasie podstawowej. Jedynie konstruktor klasy podstawowej może poprawnie zainicjować swoje elementy. Dlatego właśnie tak ważne jest wywołanie wszystkich konstruktorów — w przeciwnym razie cały obiekt nie byłby bowiem utworzony poprawnie. Z tego właśnie powodu kompilator wymusza wywołanie konstruktora dla każdego elementu zawartego w klasie pochodnej. Jeżeli konstruktor nie zostaniejawnie wymieniony na liście inicjatorów konstruktora, to kompilator wywoła konstruktor domyślny. Jeżeli domyślnego konstruktora nie będzie, kompilator zgłosi błąd.

Kolejność wywołania konstruktorów jest istotna. Podczas dziedziczenia wiadomo **wszystko** na temat klasy podstawowej; jest również dostęp do jej publicznych oraz chronionych składowych. Oznacza to, że będąc w klasie pochodnej mamy prawo założyć, że wszystkie składowe klasy podstawowej są prawidłowe. W chwili wywołania normalnej funkcji składowej konstrukcja jest już wykonana wcześniej, więc zostały już utworzone wszystkie składowe wszystkich części obiektu. Jednakże wewnątrz konstruktora musimy mieć możliwość założenia, że zostały już utworzone wszystkie wykorzystywane składowe. Jedynym sposobem zagwarantowania tego jest uprzednio wywołanie konstruktora klasy podstawowej. Dzięki temu, gdy znajdujemy się w konstrktorze klasy pochodnej, wszystkie składowe klasy podstawowej, do których możemy mieć dostęp, są już zainicjowane. Po to, by wewnątrz konstruktora wiedzieć, że wszystkie składowe są poprawne, należy — zawsze, gdy jest to możliwe — inicjalizować wszystkie obiekty składowe (to znaczy obiekty umieszczone wewnątrz klasy za pomocą metody kompozycji) na liście inicjatorów konstruktora. Stosując tę praktykę, można założyć, że zostały zainicjowane wszystkie składowe klasy podstawowej oraz wszystkie obiekty składowe bieżącego obiektu.

Wywoływanie funkcji wirtualnych wewnątrz konstrukturów

Hierarchia wywołań konstrukturów powoduje powstanie interesującego pytania. Co się stanie, gdy wewnątrz konstruktora zostanie wywołana funkcja wirtualna? Można sobie wyobrazić, co dzieje się w przypadku zwyczajnej funkcji składowej — wywołanie wirtualne jest rozstrzygane w trakcie pracy programu, ponieważ obiekt nie może wiedzieć, czy należy do klasy, wewnątrz której znajduje się wykonywana właśnie funkcja składowa, czy też do jakiejś jej klasy pochodnej. Z uwagi na spójność może się wydawać, że tak samo dzieje się w przypadku konstrukturów.

Jest jednak inaczej. W przypadku wywołania funkcji wirtualnej wewnątrz konstruktora wykorzystywana jest jedynie jej lokalna wersja. Oznacza to, że mechanizm wywoływania funkcji wirtualnych nie działa w przypadku konstrukturów.

Zachowanie takie jest logiczne z dwóch powodów. Pod względem pojęciowym, zadaniem konstruktora jest powołanie obiektu do istnienia (co wcale nie jest proste). Wewnątrz każdego konstruktora obiekt może być utworzony jedynie częściowo — wiadomo tylko, że zainicjowane zostały obiekty klasy podstawowej, ale nieznane są utworzone z niej klasy pochodne. Wywołanie wirtualnej funkcji sięga jednak „w głąb” lub „na zewnątrz” hierarchii klas. Powoduje wywołanie funkcji w klasie pochodnej. Gdyby można było zrobić to wewnątrz konstruktora, spowodowałoby to wywołanie funkcji, która miałaby dostęp do **niezainicjowanych** składowych, co niezawodnie doprowadziłoby do katastrofy.

Drugi powód ma charakter techniczny. Gdy wywoływany jest konstruktor, to jedna z pierwszych jego czynności jest inicjalizacja wskaźnika **VPTR**. Jednakże może on jedynie wiedzieć, że jest „bieżącego” typu — tego, dla którego napisano konstruktor. Kod zawarty w konstruktore nie ma zupełnie pojęcia o tym, czy obiekt jest klasy podstawowej, czy też jakiejś innej. Gdy kompilator generuje kod tego **konstruktora**,

tworzy kod dla konstruktora właśnie tej klasy, a nie klasy podstawowej lub klasy pochodnej (ponieważ klasa nie może wiedzieć o tym, jaka klasa po niej dziedziczy). Tak więc wskaźnik VPTR, którego używa, musi wskazywać tablicę VTABLE tej klasy. Wskaźnik VPTR pozostaje zainicjalizowany adresem tej tablicy do końca życia obiektu, *chyba że* nie jest to ostatnie wywołanie konstruktora. Jeżeli później nastąpi wywołanie konstruktora klasy, usytuowanej niżej w hierarchii dziedziczenia, konstruktor ten przypisze wskaźnikowi VPTR adres *swojej* tablicy VTABLE — i tak dalej, aż ostatni z konstruktorów skończy swoje działanie. Jest to dodatkowy powód, dla którego konstruktory są wywoływanie w kolejności od klasy znajdującej się na końcu hierarchii dziedziczenia.

Lecz podczas tej serii wywołań konstruktorów każdy z nich przypisuje wskaźnikowi VPTR adres swojej tablicy VTABLE. Jeżeli konstruktor wywoływania funkcji wykorzystuje mechanizmy funkcji wirtualnych, to wywołania te zostaną zrealizowane za pośrednictwem jego tablicy VTABLE, a nie tablicy VTABLE, znajdującej się na końcu hierarchii (*jak* po wywołaniu *wszystkich* konstruktorów). Ponadto wiele kompilatorów rozpoznaje, że wewnątrz konstruktora wywoływana jest funkcja wirtualna, dokonując wcześniego wiązania. Późne wiązanie spowodowałoby bowiem jedynie wywołanie funkcji lokalnej. W żadnym z wymienionych przypadków uzyskane rezultaty nie będą zgodne z tym, czego można by się pierwotnie spodziewać po wywołaniu wirtualnej funkcji wewnątrz konstruktora.

Destruktory i wirtualne destruktory

W stosunku do konstruktorów nie wolno używać słowa kluczowego **virtual**, ale destruktory **mogą**, a często nawet **muszą**, być wirtualne.

Konstruktor ma za zadanie zbudowanie obiektu, fragment po fragmencie, wywołując najpierw konstruktory klasy podstawowej, a następnie konstruktory klas pochodnych, w kolejności dziedziczenia (przy okazji musi również wywołać konstruktory obiektów składowych). Szczególne zadanie ma przed sobą również destruktor — musi „rozmontować” obiekt, który być może należy do hierarchii klas. Aby to wykonać, kompilator generuje kod, wywołujący wszystkie destruktory, ale w kolejności *odwrotnej niż* podczas konstrukcji. Oznacza to, że destruktor zaczyna od klasy znajdującej się na końcu hierarchii i przechodzi przez kolejne poziomy, aż do osiągnięcia **poziomu** klasy podstawowej. Działanie takie jest zarówno bezpieczne, jak i pożądane, ponieważ dzięki temu bieżący destruktor zawsze wie, że składowe klasy nadzędnej istnieją i działają. Jeżeli wewnątrz destruktora zachodzi potrzeba wywołania funkcji klasy podstawowej, również można tego bezpiecznie dokonać. Tak więc destruktor może przeprowadzić swoje porządkи, a następnie wywołać destruktor wyższego poziomu, który z kolei przeprowadzi *swoje* porządkи itd. *Każdy* destruktor zna klasę, z której została wprowadzona klasa, ale nie wie, jakie sajej klasy pochodne.

Należy pamiętać, że konstruktory i destruktory są jedynymi miejscami, w których musi wystąpić taka hierarchia *wywołań* (i dlatego jest ona automatycznie generowana przez kompilator). W przypadku wszystkich pozostałych funkcji tylko *one* zostaną wywołane

(a nie ich wersje, zawarte w klasie podstawowej), niezależnie od tego, czy są wirtualne, czy też nie. Wersja tej samej funkcji, zawarta w klasie podstawowej (niezależnie od tego, czy jest ona wirtualna, czy też nie), może być wywołana jedynie *jawnie*.

Zazwyczaj działanie destruktów jest zupełnie prawidłowe. Co jednak dzieje się, gdy zamierzamy operować obiektem za pośrednictwem wskaźnika do jego klasy podstawowej (czyli za pośrednictwem jego standardowego interfejsu)? Działanie takie jest głównym celem programowania obiektowego. Problem pojawi się, gdy zamierzamy usunąć obiekt wskazywany przez wskaźnik tego typu, utworzony uprzednio na stercie za pomocą polecenia `new`. Jeżeli wskaźnik jest typu wskaźnika do klasy podstawowej, to kompilator wie tylko, w jaki sposób — podezas realizacji instrukcji `delete` — wywołać wersję destruktora, zawartą w klasie podstawowej. Czyż nie wygląda to znajomo? To ten sam problem, do którego rozwiązania zostały utworzone funkcje wirtualne. Na szczęście, funkcje wirtualne działają w przypadku destruktów tak samo, jak w dla wszystkich innych funkcji — z wyjątkiem konstruktorów.

```
//: C15:VirtualDestructors.cpp
// Zachowanie wirtualnych i niewirtualnych destruktów
#include <iostream>
using namespace std;

class Basel {
public:
    ~Basel() { cout << "~Basel()\n"; }
};

class Derived1 : public Basel {
public:
    ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
    virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
    ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
    Basel* bp = new Derived1; // Rzutowanie w góre
    delete bp;
    Base2* b2p = new Derived2; // Rzutowanie w góre
    delete b2p;
} //:-
```

Po uruchomieniu programu zauważymy, że instrukcja `delete bp` wywoła jedynie destruktor klasy podstawowej, natomiast instrukcja `delete b2p` — destruktor klasy pochodnej, a następnie destruktor klasy podstawowej, co jest zachowaniem pożądanym. Gdyby zapomnieć o uczynieniu destruktora wirtualnym, stanowiłoby to niebezpieczny błąd. Co prawda na ogół nie wpływałoby to bezpośrednio na działanie programu, ale powodowałoby niezauważalne wycieki pamięci. Ponadto fakt, że jakas destrukcja jest jednak wykonywana, mógłby wjeszcze większym stopniu ukryć ten problem.

Mimo że destruktor, podobnie jak konstruktor, jest „wyjątkową” funkcją, to może on być funkcją wirtualną, ponieważ obiekt wie już, jakiego jest typu (co nie jest prawdą podczas konstrukcji). Gdy obiekt zostanie już skonstruowany, jego wskaźnik VPTR jest zainicjowany, więc mogąbyć wywoływanie funkcje wirtualne.

Czysto wirtualne destruktory

Mimo że czysto wirtualne destruktory są dozwolone w standardzie C++, to związane jest z nimi ograniczenie — funkcje czysto wirtualnych destruktów muszą zawierać kod. Brzmi to nielogicznie — jak funkcja może być „czysto” wirtualna, skoro wymaga kodu? Jeżeli jednak uświadomimy sobie, że konstruktory i destruktory są szczególnymi operacjami, nabiera to sensu — zwłaszcza gdy sobie przypomnimy, że wszyscy wywoływanie są wszystkie destruktory, w całej hierarchii klas. Gdyby można było pominąć definicję czysto wirtualnego destruktora, to jaka funkcja zostałaby wywołana podczas destrukcji? Istnienie kodu czysto wirtualnego destruktora jest absolutnie niezbędne kompilatorowi i programowi łączącemu.

Jeżeli destruktor jest funkcją czysto wirtualną, ale musi zawierać kod, to jaki z niego wynika pożytek? Jedyna zauważalna różnica pomiędzy czysto wirtualnymi destruktormi i zwykłymi wirtualnymi destruktormi polega na tym, że czysto wirtualne destruktory powodują, że klasa podstawowa staje się klasą abstrakcyjną. Z tego powodu nie można utworzyć jej obiektu (ale miałoby to miejsce również wówczas, gdy jakakolwiek inna funkcja składowa klasy podstawowej była funkcją czysto wirtualną).

Wszystko staje się jednak nieco niezrozumiałe, gdy utworzy się klasę pochodną klasy posiadającej czysto wirtualny destruktor. W odróżnieniu od wszystkich innych czysto wirtualnych funkcji, w klasie pochodnej *nie trzeba* tworzyć definicji czysto wirtualnych destruktów. Dowodem na to jest poprawne kompilowanie się poniższego pliku:

```
//: C15:UnAbstract.cpp
// Wydaje się, że czysto wirtualne destruktory zachowują się dziwnie
class AbstractBase {
public:
    virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// Nie jest potrzebne zasłonięcie destruktora?

int main() { Derived d; } //:-
```

Zwykłe czysto wirtualne funkcje, zawarte w klasie podstawowej, powinny spowodować, że klasa pochodna będzie klasą abstrakcyjną, o ile nie zostanie w tej klasie utworzoną ją definicja (a także definicje innych czysto wirtualnych funkcji). W tym przypadku wydaje się to jednak nie obowiązywać. Należy jednak pamiętać, że kompilator *automatycznie* tworzy definicję destruktora dla każdej klasy, w której nie został on

zdefiniowany. Konstruktor klasy podstawowej zostaje niejawnie zasłonięty, ponieważ jego definicja zostaje utworzona przez kompilator, więc klasa **Derived** nie jest w rzeczywistości klasą abstrakcyjną.

Można zatem zadać interesujące pytanie — jaki jest pożytek z czysto wirtualnego destruktora? W odróżnieniu od zwykłych funkcji czysto wirtualnych, *trzeba* utworzyć jego kod. W klasie pochodnej nie jesteśmy zmuszeni do utworzenia jego definicji, ponieważ kompilator generuje destruktora automatycznie. Jaka jest zatem różnica pomiędzy zwykłym wirtualnym destruktorem i czysto wirtualnym destruktorem?

Jedyna różnica występuje wtedy, gdy posiadamy klasę, zawierającą tylko jedną czysto wirtualną funkcję — **destraktor**. Wówczas jedynym skutkiem czystości destruktora jest uniemożliwienie tworzenia egzemplarzy klasy podstawowej. Gdyby istniały jakieśkolwiek inne czysto wirtualne funkcje, to właśnie one mogłyby uniemożliwić tworzenie egzemplarzy klasy podstawowej, ale w przypadku gdy nie ma takich funkcji, rolę taką przejmie czysto wirtualny destruktork. Tak więc, o ile dodanie wirtualnego destruktora jest niezbędne, to czy jest on wirtualny, czy też nie, nie ma już wielkiego znaczenia.

Po uruchomieniu poniższego programu można zauważyć, że kod czysto wirtualnego destruktora jest wywoływany po wersji destruktora, zawartej w klasie pochodnej (tak jak w przypadku każdego innego destruktora):

```
// : C15:PureVirtualDestructors.cpp
// Czysto wirtualny destruktork
// wymaga zdefiniowania kodu
#include <iostream>
using namespace std;

class Pet {
public:
    virtual ~Pet() = 0;
};

Pet::~Pet() {
    cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
    ~Dog() {
        cout << "~Dog()" << endl;
    }
};

int main() {
    Pet* p = new Dog; // Rzutowanie w góre
    delete p; // Wywołanie wirtualnego destruktora
} // :~
```

Wskazówka: ilekroć w klasie zawarta jest funkcja wirtualna, należy natychmiast doać do niej wirtualny destruktork (nawet jeżeli nie wykonuje on żadnych działań). Dzięki temu ustrzeżesz się niespodzianek w przyszłości.

Wirtualne wywołania w destruktورach

Podczas destrukcji dzieje się coś, czego nie od razu można by się spodziewać. Jeżeli wewnątrz zwyczajnej funkcji składowej wywoływana jest funkcja wirtualna, to wywołanie jest realizowane za pomocą mechanizmu późnego wiązania. Nie jest to jednak prawdą w przypadku destruktów, niezależnie od tego, czy są one wirtualne, czy też nie. Wewnątrz destruktora wywoływana jest wyłącznie „lokalna” wersja funkcji składowej – mechanizm funkcji wirtualnych jest ignorowany.

```
//: C15:VirtualsInDestructors.cpp
// Wirtualne wywołania w destruktach
#include <iostream>
using namespace std;

class Base {
public:
    virtual ~Base() {
        cout << "Base~()\n";
        f();
    }
    virtual void f() { cout << "Base::f()\n"; }
};

class Derived : public Base {
public:
    ~Derived() { cout << "~Derived()\n"; }
    void f() { cout << "Derived::f()\n"; }
};

int main() {
    Base* bp = new Derived; // Rzutowanie w góre
    delete bp;
} //:-
```

Podczas wywołania destruktora *nie jest* wywoływana funkcja **Derived::f()**, mimo iż funkcja **f()** jest funkcją wirtualną.

Dlaczego tak się dzieje? Założymy, że mechanizm funkcji wirtualnych *działały* wewnątrz destruktora. W takim przypadku wywołanie wirtualnej funkcji mogłoby prowadzić do funkcji znajdującej się „bardziej na zewnątrz” hierarchii klas (dalej od klasy podstawowej) niż bieżący destraktor. Jednak destruktory są wywoływanie „z zewnątrz do wewnątrz” (od destruktora klasy znajdującej się najdalej klasy podstawowej do destruktora klasy podstawowej), więc wywołana aktualnie funkcja mogłaby bazować na elementach obiektu, który *został już zniszczony!* Kompilator rozstrzyga natomiast wywołania w trakcie komplikacji, wywołując wyłącznie „lokalne” wersje funkcji. Zwróć uwagę na to, że tak samo dzieje się w przypadku konstruktów (*jak* to opisano wcześniej), lecz wówczas informacja o typie nie była dostępna. Tymczasem jeśli chodzi o destruktory, informacja (czyli wartość wskaźnika **VPTR**) jest dostępna, ale nie jest ona wiarygodna.

worzenie hierarchii bazującej na obiekcie

Kwestią, która przewija się w książce od czasu prezentacji klas kontenerowych **Stash** oraz **Stack**, jest „problem własności”. Pojęcie „właściciela” odnosi się do kogoś lub czegoś, odpowiedzialnego za wywołanie instrukcji **delete** w stosunku do obiektu utworzonego dynamicznie (za pomocą operatora **new**). Problem związany z używaniem kontenerów polega na tym, że muszą być one dostatecznie elastyczne, by mogły przechowywać różne typy obiektów. Aby to osiągnąć, kontenery przechowują wskaźniki typu **void***; nie znają więc typów przechowywanych obiektów. Usunięcie obiektu wskazywanego przez wskaźnik typu **void*** nie powoduje wywołania destruktora, a zatem kontener nie może być odpowiedzialny za sprzątanie zawartych w nim obiektów.

Jedno z rozwiązań zostało przedstawione w programie **C14:InheritStack.cpp**, w którym z klasy **Stack** wprowadzono nową klasę, pobierającą i zwracającą wyłącznie wskaźniki do łańcuchów. Ponieważ kontener wiedział, że może przechowywać wyłącznie wskaźniki do obiektów, będących łańcuchami, mógł je prawidłowo usunąć. Było to eleganckie rozwiązanie, wymagało jednak utworzenia nowej kontenerowej klasy pochodnej dla każdego typu, który zamierzalibyśmy przechowywać w kontenerze (mimo że wydaje się to teraz mało interesujące, warto wiedzieć, że kontenery zaczynającałać całkiem dobrze w rozdziale 16., gdy zostaną wprowadzone szablony).

Problem polega na tym, że pragniemy, by kontener przechowywał obiekty więcej niż jednego typu, lecz nie chcemy używać wskaźników typu **void***. Innym rozwiązaniem jest wykorzystanie **polimorfizmu**, przez wymuszenie, by wszystkie zawarte w kontenerze obiekty dziedziczyły po tej samej klasie podstawowej. Oznacza to, że kontener przechowuje obiekty klasy podstawowej, dzięki czemu można wywoływać w stosunku do nich funkcje wirtualne — w szczególności wirtualne destruktory, rozwiązujące problem własności.

W rozwiążaniu tym jest wykorzystana *hierarchia o jednym korzeniu* (ang *singly-rooted hierarchy*) lub *hierarchia bazująca na obiekcie* (ang. *object-based hierarchy*) — ponieważ główna klasa hierarchii nosi na ogół nazwę „**Object**” (obiekt). Okazuje się, że istnieje wiele dodatkowych korzyści, wynikających z używania hierarchii o jednym korzeniu — w rzeczywistości każdy język obiektowy, z wyjątkiem C++, wymusza wykorzystywanie takiej hierarchii — gdy tworzona jest klasa, to dziedziczy ona, bezpośrednio lub pośrednio, ze wspólnej klasy podstawowej, utworzonej przez twórców języka. W przypadku języka C++ sądzono, że wymuszenie użycia takiej wspólnej klasy podstawowej mogłoby wiązać się ze zbyt dużym narzutem, więc z tego zrezygnowano. Jednakże możemy zdecydować się na stosowanie wspólnej klasy podstawowej w swoich projektach — temat ten został omówiony dokładniej w drugim tomie książki.

W celu rozwiązania problemu własności możemy utworzyć skrajnie prostą klasę podstawową **Object**, zawierającą jedynie wirtualny destruktor. Kontener klasy **Stack** będzie przechowywał obiekty klas pochodnych klasy **Object**:

```
//: C15:OStack.h
// Wykorzystanie hierarchii o jednym korzeniu
#ifndef OSTACK_H
#define OSTACK_H
```

```

class Object {
public:
    virtual ~Object() = 0;
};

// Wymagana definicja:
inline Object::~Object() {}

class Stack {
    struct Link {
        Object* data;
        Link* next;
        Link(Object* dat, Link* nxt) :
            data(dat), next(nxt) {}
    }* head;
public:
    Stack() : head(0) {}
    ~Stack() {
        while(head)
            delete pop();
    }
    void push(Object* dat) {
        head = new Link(dat, head);
    }
    Object* peek() const {
        return head ? head->data : 0;
    }
    Object* pop() {
        if(head == 0) return 0;
        Object* result = head->data;
        Link* oldHead = head;
        head = head->next;
        delete oldHead;
        return result;
    }
};

#endif // OSTACK_H //:-/

```

Dla uproszczenia postanowiono umieścić wszystko w pliku nagłówkowym, w związku z czym (wymagana) definicja czysto wirtualnego destruktora została ulokowana w pliku nagłówkowym jako funkcja inline — podobnie jak funkcja `pop()` (co do której można się zastanawiać, czy nie jest zbyt duża, jak na funkcję inline).

Obiekty klasy `Link` przechowują obecnie wskaźniki do obiektów klasy `Object`, zamiast wskaźników typu `void*`, a klasa `Stack` pobiera i zwraca wyłącznie wskaźniki do obiektów typu `Object`. Obecnie klasa `Stack` jest bardziej elastyczna, ponieważ nie tylko będzie przechowywać wskaźniki do obiektów wielu różnych typów, ale również zniszczy obiekty pozostawione w kontenerze. Nowe ograniczenie (w rozdziale 16. zniesione — kiedy do rozwiązania problemu zostaną wykorzystane szablony) polega na tym, że typ każdego z obiektów umieszczanych w kontenerze musi być dziedziczone po klasie `Object`. Jest to do zaakceptowania, gdy tworzy się klasę od podstaw; co jednak zrobić w sytuacji, gdy chcemy umieścić w kontenerze obiekty istniejącej klasy, na przykład takiej jak `string`? W takim przypadku nowa klasa musi być klasą pochodną zarówno klasy `string`, jak i `Object`, co oznacza, że dziedziczy

ona po dwu klasach. Nosi to nazwę *wielokrotnego dziedziczenia* (ang. *multiple inheritance*) i stanowi temat całego rozdziału drugiego tomu książki (który można pobrać z witryny: <http://helion.pl/online/thinking/index.htm>). Po przeczytaniu tego rozdziału przekonasz się, że wielokrotne dziedziczenie może być nierzadko bardziej skomplikowane i stanowi środek, który należy stosować z umiarem. Jednak w tym przykładzie wszystko jest na tyle proste, że unikniemy pułapek wielokrotnego dziedziczenia:

```
//: C15:OStackTest.cpp
//{T} OStackTest.cpp
#include "OStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// Wykorzystanie wielokrotnego dziedziczenia.
// Chcemy, by obiekt był zarówno klasy string, jak i Object:
class MyString: public string, public Object {
public:
    ~MyString() {
        cout << "usuwanie lancucha: " << *this << endl;
    }
    MyString(string s) : string(s) {}
};

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // Argumentem jest nazwa funkcji
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack textlines;
    string line;
    // Odczytanie pliku i zapamiętanie wierszy na stosie:
    while(getline(in, line))
        textlines.push(new MyString(line));
    // Pobranie kilku wierszy ze stosu:
    MyString* s;
    for(int i = 0; i < 10; i++) {
        if((s=(MyString*)textlines.pop())==0) break;
        cout << *s << endl;
        delete s;
    }
    cout << "Niech reszte zrobi destruktor:"
        << endl;
} //:-)
```

Mimo że powyższy program jest podobny do poprzedniej wersji programu testującego klasę **Stack**, łatwo zauważyc, że ze stosu jest pobieranych jedynie 10 elementów, co oznacza, że prawdopodobnie część obiektów na nim pozostała. Ponieważ kontener **Stack** wie, że przechowuje obiekty klasy **Object**, destruktor potrafi poprawnie wszystko posprzątać. Świadczy o tym wyniki wyrowadzane przez program dzięki temu, że obiekty klasy **MyString** drukują komunikaty w chwili, gdy są one niszczone.

Utworzenie kontenera przechowującego obiekty klasy **Object** nie jest wcale nierośądnym podejściem -- pod warunkiem posiadania hierarchii o jednym korzeniu (wymuszonej albo przez język, albo przez wymaganie, by każda klasa była klasą pochodną klasy **Object**). W tym przypadku istnieje gwarancja, że wszystkie obiekty są obiektami klasy **Object**, dzięki czemu używanie kontenerów nie jest trudne. Jednak w języku C++ nie można oczekwać tego po każdej klasie, więc wybierając to rozwiązanie jesteśmy zmuszeni do przejścia przez wielokrotne dziedziczenie. Jak zobaczymy w rozdziale 16., szablonny umożliwiają rozwiązywanie tego problemu w znacznie prostszy i bardziej elegancki sposób.

Przeciążanie operatorów

Operatory, podobnie jak inne funkcje składowe, mogą być również zdefiniowane jako wirtualne. Implementacja wirtualnych operatorów jest jednak często nieprzejrzysta, ponieważ operacje mogą być wykonywane na dwóch obiektach, których typy nie są znanego. Zdarza się to zazwyczaj w przypadku obiektów matematycznych (dla których często definiowane są operatory). Przyjrzyjmy się na przykład systemowi, który operuje na macierzach, wektorach i wartościach skalarnych, będących obiektami klas wprowadzonych z klasy **Math**:

```
//: C15:OperatorPolymorphism.cpp
// Polimorfizm z przeciążonymi operatorami
#include <iostream>
using namespace std;

class Matrix;
class Scalar;
class Vector;

class Math {
public:
    virtual Math& operator*(Math& rv) = 0;
    virtual Math& multiply(Matrix*) = 0;
    virtual Math& multiply(Scalar*) = 0;
    virtual Math& multiply(Vector*) = 0;
    virtual ~Math() {}
};

class Matrix : public Math { // Macierz
public:
    Math& operator*(Math& rv) {
        return rv.multiply(this); // Orugi przydziela
    }
    Math& multiply(Matrix*) {
        cout << "Macierz * Macierz" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Skalar * Macierz" << endl;
        return *this;
    }
}
```

```

Math& multiply(Vector*) {
    cout << "Wektor * Macierz" << endl;
    return *this;
}

class Scalar : public Math { // Skalar
public:
    Math& operator*(Math& rv) {
        return rv.multiply(this); // Drugi przydziela
    }
    Math& multiply(Matrix*) {
        cout << "Macierz * Skalar" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Skalar * Skalar" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Wektor * Skalar" << endl;
        return *this;
    }
};

class Vector : public Math { // Wektor
public:
    Math& operator*(Math& rv) {
        return rv.multiply(this); // Drugi przydziela
    }
    Math& multiply(Matrix*) {
        cout << "Macierz * Wektor" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Skalar * Wektor" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Wektor * Wektor" << endl;
        return *this;
    }
};

int main() {
    Matrix m; Vector v; Scalar s;
    Math* math[] = { &m, &v, &s };
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
            Math& m1 = *math[i];
            Math& m2 = *math[j];
            m1 * m2;
        }
    }
}

```

Dla uproszczenia został przeciążony tylko **operator***. Jego zadaniem jest umożliwienie mnożenia przez siebie dwóch dowolnych obiektów klasy **Math** i zwracanie pożądanego wyniku — należy zauważyć, że mnożenie macierzy przez wektor jest zupełnie

inną operacją niż mnożenie wektora przez macierz. Problem polega na tym, że wyrażenie `m1 * m2`, znajdujące się w funkcji `main()`, zawiera dwa rzutowania w górę referencji do obiektów klasy `Math`, a zatem dwa obiekty o nieznanych typach. Funkcja wirtualna jest w stanie dokonać jedynie pojedynczego przydziału — to znaczy określenia typu nieznanego obiektu. W celu określenia obu typów w *przykładzie* została wykorzystana technika, nazywana *przydzieleniem wielokrotnym* (ang. *multiple dispatching*), dzięki której to, co wygląda na pojedyncze wywołanie wirtualnej funkcji, skutkuje jeszcze drugim wirtualnym wywołaniem. W momencie wykonania tego drugiego wywołania określone są już typy obu obiektów i w związku z tym możliwe jest podjęcie odpowiedniego działania. Nie jest to widoczne na pierwszy rzut oka, ale jeżeli przyjrzyć się przez chwilę programowi, wszystko powinno stać się zrozumiałe. Przykład ten został dokładniej omówiony w rozdziale poświęconym wzorcem projektowym, zamieszczonym w drugim tomie książki, który można pobrać z witryny <http://Melson.pl/online/thinking/index.html>.

Rzutowanie w dół

Zgodnie z oczekiwaniami, skoro istnieje rzutowanie w górę — czyli przemieszczańie się w górę hierarchii dziedziczenia — powinno też istnieć *rzutowanie w dół*, umożliwiające przejście w dół tej hierarchii. Jednak rzutowanie w górę jest proste, ponieważ w miarę poruszania się w górę hierarchii klasy zawsze łączą się w coraz bardziej ogólne klasy. Oznacza to, że podczas rzutowania w górę każda klasa pochodna ma zawsze wyraźnie określoną klasę podstawową (zazwyczaj jedną, z wyjątkiem przypadku wielokrotnego dziedziczenia), ale podczas rzutowania w dół istnieje na ogół wybór spośród wielu możliwości. Na przykład `Okrąg` jest typu `Figura` (rzutowanie w górę), ale rzutując w dół `Figura`, można otrzymać `Okrąg`, `Kwadrat`, `Trójkąt` itd⁶. Tak więc problem polega na określeniu sposobu bezpiecznego rzutowania w dół (być może jeszcze ważniejszą kwestią jest odpowiedź na pytanie, po co dokonuje się rzutowania w dół zamiast wykorzystać polimorfizm, umożliwiający automatyczne określenie typu; sposoby unikania rzutowania w dół zostały opisane w drugim tomie książki).

Język C++ umożliwia specjalne *jawne rzutowanie* (przedstawione w rozdziale 3.) o nazwie `dynamic_cast`, będące operacją *rzutowania w dół bezpiecznego dla typów* (ang. *type-safe downcast*). Gdy wykonuje się taką operację, próbując zrealizować rzutowanie w dół na określony typ, to zwrócona wartość będzie wskaźnikiem do żądanego typu tylko w przypadku, gdy rzutowanie jest poprawne i zakończy się powodzeniem. W przeciwnym razie zostanie zwrócona wartość zerowa, oznaczająca, że podany docelowy typ jest nieprawidłowy. Poniżej zamieszczono, ograniczony do minimum, przykład rzutowania w dół:

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){}};

class Cat : public Pet { public: virtual void Meow() {} };

class Dog : public Pet { public: virtual void Bark() {} };

int main() {
    Pet* p = new Cat();
    if (dynamic_cast<Cat*>(p)) {
        Cat* cat = dynamic_cast<Cat*>(p);
        cat->Meow();
    }
    delete p;
}
```

⁶ Przykład odwołuje się hierarchii typów przedstawionej na str. 33 niniejszej książki — *przyp. red.*

```

class Dog : public Pet {};
class Cat : public Pet {};

int main() {
    Pet* b = new Cat; // Rzutowanie w góre
    // Próba rzutowania na wskaźnik typu Dog*:
    Dog* d1 = dynamic_cast<Dog*>(b);
    // Próba rzutowania na wskaźnik typu Cat*:
    Cat* d2 = dynamic_cast<Cat*>(b);
    cout << "d1 = " << (long)d1 << endl;
    cout << "d2 = " << (long)d2 << endl;
} //:-

```

Rzutowania **dynamic_cast** trzeba używać w odniesieniu do prawdziwie polimorficznej hierarchii — zawierającej funkcje wirtualne — ponieważ do określenia rzeczywistego typu wykorzystuje ono informacje, zawarte w tablicy VTABLE. W powyższym przykładzie wystarcza w zupełności, że klasa podstawowa posiada wirtualny destruktor. W funkcji **main()** wskaźnik do obiektu klasy **Cat** jest rzutowany w góre na wskaźnik obiektu klasy **Pet**, a następnie próbuje się go rzutować w dół — zarówno na wskaźnik do obiektu klasy **Dog**, jak i na wskaźnik do obiektu klasy **Cat**. Wartości obu wskaźników są drukowane, dzięki czemu po uruchomieniu programu można się przekonać, że w wyniku nieprawidłowego rzutowania w dół uzyskuje się wartość zerową. Oczywiście, podczas rzutowania w dół trzeba się zawsze upewnić, że zwrócona wartość, będącą wynikiem, jest niezerowa. Nie należy się również spodziewać, że zwracany będzie zawsze taki sam wskaźnik, ponieważ w trakcie rzutowania w góre i w dół dokonywane jest czasami wyrównywanie wskaźników (szczególnie w przypadku wielokrotnego dziedziczenia).

Rzutowanie **dynamic_cast** wymaga do działania pewnego narzutu — nie jest on wielki, ale jeżeli dokonuje się licznych rzutowań (w takim przypadku należałoby się uważnie przyjrzeć projektowi programu), to kwestie związane z wydajnością mogą stać się problemem. W pewnych przypadkach podczas rzutowania w dół możemy jednak wiedzieć coś szczególnego, co sprawi, że zyskamy pewność, z jakim typem mamy do czynienia. Dzięki temu rzutowanie **dynamic_cast** stanie się niepotrzebne i będziemy mogli użyć zamiast niego rzutowania **static_cast**. Poniżej przedstawiono taką sytuację:

```

//: C15:StaticHierarchyNavigation.cpp
// Poruszanie się w hierarchii klas
// za pomocą static_cast
#include <iostream>
#include <typeinfo>
using namespace std;

class Shape { public: virtual ~Shape() {} };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
    Circle c;
    Shape* s = &c; // Rzutowanie w gore - normalne i prawidłowe
    // Bardziej jasne, ale niepotrzebne:
    s = static_cast<Shape*>(&c);
}

```

```
// (ponieważ rzutowanie w góre jest tak bezpieczną
// i typową operacją, że rzutowanie wprowadza bałagan)
Circle* cp = 0;
Square* sp = 0;
// Statyczne poruszanie się w hierarchii klas
// wymaga dodatkowej informacji o typeid:
if(typeid(s) == typeid(cp)) // RTTI C++
    cp = static_cast<Circle*>(s);
if(typeid(sp) == typeid(s))
    sp = static_cast<Square*>(s);
if(cp != 0)
    cout << "To jest okrąg!" << endl;
if(sp != 0)
    cout << "To jest kwadrat!" << endl;
// Statyczna nawigacja jest TYLKO obejściem stosowanym
// z powodu efektywności - rzutowanie dynamic_cast
// jest zawsze bezpieczniejsze. Jednak instrukcja:
// Other* op = static_cast<Other*>(s);
// powoduje zgłoszenie błędu, podczas gdy instrukcja:
Other* op2 = (Other*)s;
// tego nie robi
```

W programie wykorzystano nową cechę języka, która została w pełni opisana dopiero w drugim tomie książki, w którym poświęcono jej cały rozdział — mechanizm *identyfikacji typów podczas pracy programu* (ang. *run-time type identification* — RTTI). Mechanizm ten pozwala na uzyskanie informacji o typie, która została utracona podczas rzutowania w górę. Rzutowanie **dynamic_cast** jest w rzeczywistością jedną z postaci identyfikacji typów podczas pracy programu. W powyższym programie do określenia typu wskaźnika używane jest słowo kluczowe **typeid** (zadeklarowane w pliku nagłówkowym `<typeinfo>`). Jak widać, typ rzutowanego w górę wskaźnika obiektu typu **Shape** jest kolejno porównywany ze wskaźnikami do obiektu typu **Circle** i **Square** w celu sprawdzenia, czy któryś z nich odpowiada. Mechanizm identyfikacji typów podczas pracy programu to więcej niż tylko słowo kluczowe **typeid** — łatwo można sobie również wyobrazić, w jaki sposób można by bez trudu zaimplementować własny system informacji o typie, wykorzystujący funkcję wirtualną.

Tworzony jest obiekt klasy **Circle**, a jego adres jest rzutowany w góre na wskaźnik klasy **Shape** — druga wersja tego wyrażenia pokazuje, w jaki sposób można użyć rzutowania **static_cast**, aby rzutowanie w góre było nieco bardziej jawne. Ponieważ jednak rzutowanie w góre jest zawsze bezpieczne, i jest to często wykonywana operacja, uważam, że w przypadku rzutowania w góre jawne rzutowanie wprowadza bałągan i jest niepotrzebne.

Do określenia typu wykorzystano mechanizm identyfikacji typów podczas pracy programu, a następnie, do rzutowania w dół, zastosowano rzutowanie `static_cast`. Warto jednak podkreślić, że w tym programie odbywa się to właściwie tak samo, jakby użyte zostało rzutowanie `dynamic_cast`, natomiast klient-programista musi dokonać pewnych sprawdzeń, by dowiedzieć się, które z rzutowań zakończyło się powodzeniem. Zanim zamiast rzutowania `dynamic_cast` użyje się rzutowania `static_cast`, na ogół dobrze jest mieć do czynienia z sytuacją nieco bardziej deterministyczną niż przedstawiona powyżej (i jeszcze raz trzeba zaznaczyć, że przed wykorzystaniem rzutowania `dynamic_cast` należy ponownie przyjrzeć się swojemu projektowi).

Jeżeli hierarchia klas nie zawiera funkcji wirtualnych (co świadczy o wątpliwej jakości projektu) albo jeżeli posiadamy jakieś inne informacje, pozwalające na bezpieczne rzutowanie w dół, rzutowanie statyczne jest nieco szybsze niż rzutowanie **dynamic_cast**. Ponadto rzutowanie **static_cast** nie pozwala na przekroczenie granic hierarchii klas, jak ma to miejsce w przypadku tradycyjnego rzutowania, dzięki czemu jest od niego bezpieczniejsze. Jednakże statyczne poruszanie się po hierarchii klas jest zawsze ryzykowne i dopóki nie mamy do czynienia z jakimś szczególnym przypadkiem, powinniśmy używać rzutowania **dynamic_cast**.

Podsumowanie

Polimorfizm, zaimplementowany w języku C++ za pomocą funkcji wirtualnych, oznacza „wielopostaciowość”. W programowaniu obiektowym mamy do czynienia z tym samym obliczem (wspólnym interfejsem, zawartym w klasie podstawowej) oraz różnymi postaciami o tym obliczu — rozmaitymi wersjami funkcji wirtualnych.

Jak przekonaliśmy się w tym rozdziale, bez użycia abstrakcji danych i dziedziczenia nie jest możliwe zrozumienie ani nawet utworzenie przykładu polimorfizmu. Polimorfizm jest **własnością**, której nie sposób zaprezentować w izolacji (jak na przykład deklaracji **const** czy instrukcji **switch**); istnieje wyłącznie we współdziałaniu, jako element większego obrazu związków pomiędzy klasami. Niektórzy są zdezorientowani innymi, nieobiektowymi cechami języka C++, takimi jak przeciążanie oraz argumenty domyślne, przedstawianymi czasami jako jego cechy obiektowe. Nie dajmy się na to nabrać — jeżeli nie następuje późne wiązanie, to nie ma również mowy o polimorfizmie.

Aby w swoich programach efektywnie używać polimorfizmu — a zatem również technik obiektowych — musimy rozszerzyć swoje spojrzenie na programowanie, tak aby nie obejmowało ono jedynie składowych oraz komunikatów związanych z pojedynczą **klassą**, ale również cechy wspólne wielu klas oraz wzajemne relacje pomiędzy nimi. Mimo że wymaga to znacznego wysiłku, to rzeczą jest warta gry ze względu na istotne korzyści: szybsze tworzenie programów, lepszą organizację kodu, możliwość rozbudowy programów oraz łatwiejszą pielęgnację ich kodu.

Polimorfizm zamkna obiektowe właściwości języka. Jednak język C++ zawiera jeszcze dwa ważne elementy — szablony (które zostaną przedstawione w rozdziale 16., a bardziej szczegółowo ich opis znajduje się w drugim tomie książki) oraz obsługę wyjątków (opisaną w drugim tomie książki). Elementy te umożliwiają zwiększenie potencjału programistycznego w takim samym stopniu, jak elementy związane z programowaniem obiektowym — takie jak abstrakcyjne typy danych, dziedziczenie oraz polimorfizm.

Ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny www.BruceEckel.com.

1. Utwórz prostą hierarchię „figur” • - klasę podstawową o nazwie **Shape** (figura) oraz **klasy** pochodne o nazwach **Circle** (okrąg), **Square** (kwadrat) i **Triangle** (trójkąt). W klasie podstawowej utwórz funkcję wirtualną o nazwie **draw()** (rysuj) i zasłoń ją w klasach pochodnych. Na stercie utwórz tablicę wskaźników do obiektów klasy **Shape** (dokonując w ten sposób rzutowania w górę wskaźników) i wywołaj funkcję **draw()** za pośrednictwem wskaźników do klasy podstawowej, sprawdzając działanie funkcji wirtualnych. Jeżeli używany przez ciebie program uruchomieniowy na to pozwala, wykonaj program krokowo.
2. Zmodyfikuj poprzednie ćwiczenie w taki sposób, by funkcja **draw()** była funkcją czysto wirtualną. Spróbuj utworzyć obiekt klasy **Shape**. Wykonaj próbę wywołania tej czysto wirtualnej funkcji w obrębie konstruktora i zobacz, co się stanie. Pozostawiając funkcję **draw()** funkcją czysto wirtualną, utwórz jej definicję.
3. Rozszerzając poprzednie ćwiczenie, utwórz funkcję, *pobierającą przez wartość* argument, będący obiektem klasy **Shape**, a następnie spróbuj rzutować w górę obiekt **klasy** pochodnej, używając go jako argumentu tej funkcji. Zobacz, co się wówczas stanie. Popraw funkcję w taki sposób, by pobierała referencję do obiektu klasy **Shape**.
4. Zmodyfikuj program **C14:Combined.cpp** w taki sposób, by funkcja **f()** była w klasie podstawowej funkcją wirtualną. Zmień zawartość funkcji **main()**, by dokonać rzutowania w górę i wirtualnego wywołania funkcji.
5. Zmodyfikuj program **Instrument3.cpp**, dodając wirtualną funkcję **prepare()**. Wywołaj tę funkcję wewnątrz funkcji **tune()**.
6. Utwórz hierarchię dziedziczenia klasy **Rodent** (gryzoń): **Mouse** (mysz), **Gerbil** (myszakoczek), **Hamster** (chomik) itd. W klasie podstawowej utwórz funkcje, które są wspólne dla wszystkich gryzoni i *przedefiniuj je* w klasach pochodnych, umożliwiając różne zachowania, w zależności od konkretnego typu pochodnego klasy **Rodent**. Utwórz tablicę wskaźników do obiektów klasy **Rodent**, wypełnij ją wskaźnikami do obiektów poszczególnych klas pochodnych, a następnie wywołaj funkcje, zawarte w klasie podstawowej, obserwując, co się stanie.
7. Zmodyfikuj poprzedni przykład, używając, zamiast tablicy wskaźników, wektora **vector<Rodent*>**. Upewnij się, że pamięć jest zwalniana w odpowiedni sposób.
8. Rozpoczynając od utworzonej poprzednio hierarchii **Rodent**, wyprowadź z klasy **Hamster** klasę **BlueHamster** (błękitny chomik — *jest* takie stworzenie; miałem je kiedyś w dzieciństwie), zasłoń funkcje klasy podstawowej i pokaż, że nie trzeba zmieniać kodu wywołującego funkcje **klasy** podstawowej, by dostosować go do nowego typu.
9. Rozpoczynając od utworzonej poprzednio hierarchii **Rodent**, dodaj *niewirtualny* destruktor, utwórz obiekt klasy **Hamster**, używając do tego operatora **new**. Następnie dokonaj rzutowania uzyskanego wskaźnika na typ **Rodent***, a potem usuń wskazywany obiekt za pomocą **delete**, by pokazać, że nie spowoduje to wywołania wszystkich destruktorów znajdujących się w hierarchii. Zmień destruktor na wirtualny i pokaż, że wszystko działa teraz prawidłowo.

10. Rozpoczynając od utworzonej poprzednio hierarchii **Rodent**, zmodyfikuj klasę **Rodent**, by była czysto abstrakcyjną klasą podstawową.
11. Utwórz system kontroli ruchu powietrznego, posiadający klasę podstawową **Aircraft** (samolot) i różne typy pochodne. Utwórz klasę **Tower** (wieża), zawierającą wektor **vector<Aircraft*>**, która wysyła komunikaty do różnych samolotów, znajdujących się pod jej kontrolą.
12. Utwórz model szklarni, wyprowadzając z klasy **Plant** (roślina) różne gatunki roślin i wbudowując w swojszklarnię mechanizmy utrzymujące rośliny.
13. W programie **Early.cpp** uczyń klasę **Pet** czysto abstrakcyjną klasą podstawową.
14. W programie **AddingVirtuals.cpp** uczyń wszystkie funkcje składowe klasy **Pet** funkcjami czysto wirtualnymi, ale utwórz definicję funkcji **name()**. Popraw odpowiednio klasę **Dog**, wykorzystując definicję funkcji **name()**, zawartą w klasie **podstawowej**.
15. Napisz niewielki program, prezentujący różnicę pomiędzy wywołaniem funkcji wirtualnej wewnątrz normalnej funkcji składowej i wewnątrz konstruktora. Program powinien udowodnić, że te dwa wywołania dają różne wyniki.
16. Zmodyfikuj program **VirtualsInDestructors.cpp**, tworząc klasę pochodną klasy **Derived** i zasłaniając w niej funkcję **f()** oraz destruktor. W funkcji **main()** utwórz obiekt nowego typu, dokonaj jego rzutowania w góre, a następnie usuń go za pomocą operatora **delete**.
17. W programie **powstałym** w ramach poprzedniego ćwiczenia dodaj w każdym destruktorze wywołanie funkcji **f()**. Wyjaśnij, co się stało.
18. Utwórz klasę, zawierającą składową, oraz klasę pochodną, zawierającą dodatkową składową. Wykorzystując operator **sizeof**, napisz funkcję niebędącą funkcją składową, pobierającą przez wartość obiekt klasy podstawowej i drukującą wielkość tego obiektu. W funkcji **main()** utwórz obiekt klasy pochodnej, wydrukuj jego wielkość, a następnie wywołaj napisaną przednio funkcję. Wyjaśnij, co się stało.
19. Utwórz prosty przykład wywołania funkcji wirtualnej i wygeneruj dla niego program w języku asemblera. Odszukaj kod w języku asemblera, realizującego wirtualne wywołanie, prześledź ten kod i wyjaśnij jego działanie.
20. Utwórz klasę, zawierającą jedną funkcję **wirtualną**, i jedną funkcję, niebędącą funkcją wirtualną. Utwórz nową klasę pochodną, a następnie obiekt tej klasy i dokonaj rzutowania w górę na wskaźnik do klasy podstawowej. Użyj funkcji **clock()**, zawartej w pliku nagłówkowym **<ctime>** (należy poszukać jej w dokumentacji biblioteki języka C), by zmierzyć różnicę pomiędzy wywołaniem wirtualnym i wywołaniem **niewirtualnym**. Aby zauważyć różnicę, trzeba wykonać wiele wywołań każdej funkcji wewnątrz pętli, w której dokonywany jest pomiar czasu.
21. Zmodyfikuj program **C14:Order.cpp**, dodając w klasie podstawowej funkcję wirtualną za pomocą makroinstrukcji **CLASS** (tak aby coś drukowała) oraz czyniąc destruktor wirtualnym. Utwórz obiekty różnych klas pochodnych

- i dokonaj ich rzutowania w górę na klasę podstawową. Upewnij się, że działają wirtualne mechanizmy, a także, że dokonywaną jest poprawna konstrukcja i destrukcja.
22. Utwórz klasę, zawierającą trzy przeciążone funkcje wirtualne. Utwórz jej klasę pochodną i zasłoń w niej jedną z funkcji. Utwórz obiekt klasy pochodnej. Czy za pośrednictwem obiektu klasy pochodnej można wywołać wszystkie funkcje klasy podstawowej? Dokonaj rzutowania adresu obiektu na wskaźnik do klasy podstawowej. Czy wszystkie trzy funkcje mogą być wywołane za pośrednictwem klasy podstawowej? Usuń definicję funkcji zasłaniającej, znajdującej się w klasie pochodnej. Czy teraz można wywołać wszystkie trzy funkcje za pośrednictwem obiektu klasy pochodnej?
 23. Zmodyfikuj program **VariantReturn.cpp**, by pokazać, że działa tak samo z wykorzystaniem referencji, jak z wykorzystaniem wskaźników.
 24. W jaki sposób można określić, jak w programie **Early.cpp** kompilator realizuje wywołanie — za pomocą wczesnego czy też późnego wiązania? Ustal, w jaki sposób odbywa się to w przypadku używanego przez ciebie kompilatora.
 25. Utwórz klasę **podstawową**, zawierającą funkcję **clone()**, zwracającą wskaźnik do **kopii** bieżącego obiektu. Utwórz dwie klasy pochodne, które zawierają funkcje zasłaniające funkcję **clone()**, zwracające wskaźniki do kopii obiektów swoich typów. Utwórz w funkcji **main()** obiekty obu klas pochodnych i dokonaj ich rzutowania w górę, a następnie dla każdego z nich wywołaj funkcję **clone()** i sprawdź, czy utworzone kopie są odpowiednimi podtypami. Po eksperymentuj z funkcją **clone()** w taki sposób, aby zwracała wskaźnik do typu podstawowego, a następnie spróbuj zwracać określony dokładnie typ pochodny. Czy możesz wyobrazić sobie sytuację, w których konieczne jest zastosowanie drugiego z opisywanych sposobów?
 26. Zmodyfikuj program **OStackTest.cpp**, tworząc własną klasę. Następnie, wykorzystując wielokrotne dziedziczenie, utwórz klasę pochodną wygenerowanej uprzednio klasy i klasy **Object**. Utworzysz w ten sposób coś, co można umieścić w kontenerze **Stack**. Przetestuj działanie swojej klasy w funkcji **main()**.
 27. Do programu **OperatorPolymorphism.cpp** dodaj typ **Tensor**.
 28. (Średnio trudne) Utwórz klasę **podstawową X**, nieposiadającą danych składowych ani konstruktora, a tylko pojedynczą funkcję wirtualną. Utwórz klasę **Y**, będącą klasą pochodną klasy **X**, nieposiadającą jawnego konstruktora. Wygeneruj kod programu w języku asemblera i przeanalizuj go, aby określić, czy dla klasy **X** tworzony jest konstruktor, a jeżeli tak, to jak jest jego kod. Wyjaśnij uzyskane rezultaty. Skoro klasa **X** nie posiada domyślnego konstruktora, to dlaczego kompilator nie zgłasza błędu?
 29. (Średnio trudne) Zmodyfikuj poprzednie ćwiczenie, tworząc w obu klasach konstruktory w taki sposób, by każdy z nich wywoływał funkcję wirtualną. Wygeneruj kod programu w języku asemblera. Zobacz, gdzie przypisywana jest wartość wskaźnikowi **VPTR** wewnątrz każdego z konstraktorów.

Czy używany przez ciebie kompilator stosuje wewnątrz konstruktora mechanizm wywołań funkcji wirtualnych? Ustal, dlaczego wewnątrz konstruktora wywoływaną jest lokalna wersja funkcji.

30. (Trudne) Gdyby wywołania funkcji w stosunku do obiektu przekazywanego przez wartość *nie podlegały* wczesnemu wiążaniu, to wywołanie wirtualne mogłyby odwoływać się do nieistniejących części obiektu. Czy to możliwe? Napisz kod wymuszający wywołanie wirtualne i zobacz, czy wywoła to załamanie wykonywania programu. Aby wyjaśnić jego zachowanie, sprawdź, co się stanie w przypadku przekazania obiektu przez wartość.
31. (Trudne) Sprawdź dokładnie, ile razy więcej czasu wymaga wywołanie funkcji wirtualnej — sięgając do informacji, dotyczących języka asemblera używanego przez ciebie procesora albo innej dokumentacji technicznej, i określając liczbę cykli zegarowych wymaganych w przypadku prostego wywołania funkcji w porównaniu z liczbą cykli wymaganą dla wywołania funkcji wirtualnej.
32. Określ wielkość wskaźnika VPTR w używanej przez ciebie implementacji języka. Następnie, wykorzystując wielokrotne dziedziczenie, utwórz klasę **pochodną dwóch** klas, zawierających funkcje wirtualne. Czy powstała w ten sposób klasa posiada jeden czy dwa wskaźniki VPTR?
33. Utwórz klasę, posiadającą dane składowe oraz funkcje wirtualne. Napisz funkcję, która przegląda pamięć zajmowaną przez obiekt i drukuje różnych jego części. Aby to zrobić, należy wykonać szereg eksperymentów i metodą prób i błędów odkryć, gdzie w obiekcie przechowywany jest wskaźnik VPTR.
34. Udając, że funkcje wirtualne nie **istnieją**, zmodyfikuj program **Instrument4.cpp** w taki sposób, by zastąpić w nim wywołania funkcji wirtualnych rzutowaniem **dynamic_cast**. Wyjaśnij, dlaczego jest to zły pomysł.
35. Zmodyfikuj program **StaticHierarchyNavigation.cpp** w taki sposób, by zamiast dostępnej w C++ identyfikacji typów podczas pracy programu (RTTI) **utworzyć** swój własny mechanizm identyfikacji typów, używając do tego zawartej w klasie podstawowej funkcji **whatAmI()** („*czym jestem*”) oraz wyliczenia **enum type { Circles, Squares };**.
36. Używając jako punktu wyjścia programu **PointerToMemberOperator.cpp**, zamieszczonego w rozdziale 12., pokaż, że **polimorfizm** działa również w przypadku wskaźników do składowych, nawet gdy **operator->*** jest przeciążony.

Rozdział 16.

Wprowadzenie do szablonów

Dziedziczenie i kompozycja umożliwiają powtórne wykorzystanie kodu obiektów. Dostępne w języku C++ *szablony* pozwalają na ponowne zastosowanie *kodu źródłowego*.

Mimo że szablony dostępne w języku C++ są narzędziem programistycznym ogólnego przeznaczenia, to gdy zostały one wprowadzone do języka, wydawały się zniechęcać do używania hierarchii klas kontenerów, bazujących na obiektach (pokazanych pod koniec rozdziału 15.). Na przykład kontenery i algorytmy standardu języka C++ (opisane w dwóch rozdziałach drugiego tomu książki, który można pobrać z witryny <http://helion.pl/online/thinking/index.html>) zostały zbudowane wyłącznie z wykorzystaniem szablonów i z punktu widzenia programisty są one dość łatwe w użytku.

W rozdziale tym przedstawiono nie tylko podstawy szablonów, ale zawarto w nim również wprowadzenie do kontenerów, będących podstawowymi elementami programowania obiektowego. Zostały one zrealizowane niemal w pełni w postaci kontenerów, zawartych w standardowej bibliotece języka C++. W książce pojawiły się już przykłady kontenerów — klasy **Stash** i **Stack** — więc czytelnik nabrał już pewnej wprawy w ich używaniu. W niniejszym rozdziale do kontenerów zostanie dodane pojęcie *iteratatora*. Mimo że kontenery są idealnymi przykładami wykorzystania szablonów, w drugim tomie książki (zawierającym rozdział, opisujący bardziej zaawansowane zagadnienia dotyczące szablonów) dowiesz się, że istnieje również wiele innych sposobów ich stosowania.

Kontenery

Założmy, że zamierzamy utworzyć stos, tak jak już zostało przedstawione w książce. Stos ten będzie przechowywał liczby całkowite i zostanie zrealizowany w prosty sposób:

```

//: C16:IntStack.cpp
// Prosty stos liczb całkowitych
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
    enum { ssize = 100 };
    int stack[ssize];
    int top;
public:
    IntStack() : top(0) {}
    void push(int i) {
        require(top < ssize, "Zbyt wiele wywolan funkcji push()");
        stack[top++] = i;
    }
    int pop() {
        require(top > 0, "Zbyt wiele wywolan funkcji pop()");
        return stack[--top];
    }
};

int main() {
    IntStack is;
    // Umieszczenie na stosie kilku liczb Fibonacciego,
    // w celu uczynienia go bardziej interesującym:
    for(int i = 0; i < 20; i++)
        is.push(fibonacci(i));
    // Pobranie ze stosu i wydrukowanie liczb:
    for(int k = 0; k < 20; k++)
        cout << is.pop() << endl;
} //:-_

```

Klasa **IntStack** jest prostym przykładem rozwijanego w dół stosu. Dla uproszczenia został on utworzony jako stos stałej **wielkości**, ale nic nie stoi na przeszkodzie, by zmodyfikować go w taki sposób, aby powiększał się automatycznie, przydzielając pamięć na stercie, jak w przypadku klasy **Stack**, opisywanej w tej książce.

W funkcji **main()** na stosie jest umieszczanych kilka liczb **całkowitych**, które później są z niego z powrotem pobierane. Aby uczyć się ten przykład bardziej interesującym, umieszczane na stosie liczby tworzone są za pomocą funkcji **fibonacci()**, generującej znany ciąg liczb, opisujący rozmnazanie się królików. Poniżej zamieszczono plik nagłówkowy, deklarujący tę funkcję:

```

//: C16:fibonacci.h
// Generator liczb Fibonacciego
int fibonacci(int n); //:-_

```

A oto jego implementacja:

```

//: C16:fibonacci.cpp {0}
#include "../require.h"

```

```

int fibonacci(int n) {
    const int sz = 100;
    require(n < sz);
    static int f[sz]; // Inicjalizowana wartościami zerowymi
    f[0] = f[1] = 1;
    // Poszukiwanie niewypełnionych elementów tablicy:
    int i;
    for(i = 0; i < sz; i++) {
        if(f[i] == 0) break;
    while(i <= n) {
        f[i] = f[i-1] + f[i-2];
        i++;
    }
    return f[n];
} //:-)

```

Jest to dość efektywna implementacja, ponieważ nigdy nie generuje ona żadnej liczby więcej niż jednokrotnie. Zastosowano w niej statyczną tablicę liczb całkowitych, a ponadto wykorzystano fakt, że kompilator inicjalizuje tablice statyczne wartościami **zerowymi**¹. Pierwsza pętla **for** powoduje przesunięcie indeksu *i* do pierwszego elementu tablicy, którego wartość jest zerowa, a następnie pętla **while** dodaje do tablicy kolejne liczby ciągu Fibonacciego, aż do osiągnięcia żądanego elementu. Zwrót uwagi na to, że jeżeli pozycja *n* tablicy została już zainicjowana liczbami ciągu Fibonacciego, to pętla **while** jest w całości **pomijana**².

Potrzeba istnienia kontenerów

Oczywiście, stos liczb całkowitych nie jest narzędziem o decydującym znaczeniu. Prawdziwa potrzeba używania kontenerów pojawia się w przypadku tworzenia obiektów na stercie za pomocą operatora **new** i w razie usuwania ich instrukcją **delete**. Na ogół w trakcie pisania programu nie wiadomo, ile obiektów będzie potrzebnych. Na przykład w systemie obsługującym kontrolę ruchu lotniczego nie zamierzamy ograniczać liczby samolotów, obsługiwanych przez system. Nie chcielibyśmy, aby program przerwał pracę tylko dlatego, że przekroczyliśmy jakąś ich liczbę. W komputerowym systemie wspomagającym projektowanie mamy do czynienia z dużą liczbą symboli, ale to użytkownik określa (w trakcie pracy programu) ile ich dokładnie potrzeba. Kiedy uświadomasz sobie tę tendencję, zauważysz wiele podobnych przykładów, dotyczących sytuacji występujących w twoich programach.

Programiści języka C, którzy w swoim „systemie obsługi pamięci” wykorzystywali pamięć **wirtualną**, są często zaniepokojeni pomysłem stosowania operatorów **new** i **delete** oraz klas kontenerowych. Najwidoczniej jedynym sposobem stosowanym

¹ W tym przypadku lepiej byłoby wykorzystać **inicjalizację agregatową**, definiując tablicę *f* [] w postaci: **static int f[sz] = {1, 1};**, co zapobiegłoby wielokrotnemu przypisywaniu wartości jej dwóm pierwszym elementom, podczas każdego wywołania funkcji — **przyp. tłum.**

² Znacznie efektywniej byłoby od razu, na początku funkcji, zwracać wartość elementu **f[n]**, w przypadku, gdy nie jest on zerowy, co pozwoliłoby uniknąć niepotrzebnego poszukiwania niewypełnionych elementów tablicy w sytuacji, gdy wartość żądanego elementu ciągu została już wcześniej wyliczona — **przyp. tłum.**

w języku C było utworzenie ogromnej, globalnej tablicy, większej niż to, czego mógłby kiedykolwiek potrzebować program. Nie wymaga to co prawda dużo myślenia (ani znajomości funkcji **malloc()** i **free()**), lecz powoduje tworzenie programów, które nie są łatwo przenośne i mogą zawierać trudne do znalezienia błędy.

Ponadto jeżeli w języku C++ zostanie utworzona ogromna tablica obiektów, narzut wprowadzony przez konstruktor i destruktor może w znacznym stopniu spowolnić je-gó działanie. Podejście stosowane w języku C++ sprawdza się znacznie lepiej — gdy potrzebny jest obiekt, tworzymy go za pomocą operatora new i umieszczamy jego wskaźnik w kontenerze. Następnie pobieramy go i wykonujemy na nim jakieś operacje. Tworzymy zatem tylko te obiekty, które są niezbędne. Zazwyczaj również kiedy uruchamiany jest program, nie są dostępne wszystkie informacje, potrzebne do inicjalizacji obiektów. Operator **new** pozwala na wstrzymanie się z utworzeniem obiektu do chwili, gdy wydarzy się coś w jego otoczeniu.

Tak więc w najbardziej typowej sytuacji będziemy tworzyć kontenery, przechowujące wskaźniki do interesujących nas obiektów. Obiekty te będą tworzone za pomocą operatora new, a uzyskany w ten sposób wskaźnik zostanie umieszczony w kontenerze (potencjalnie dokonując jednocześnie rzutowania w góre), a następnie pobierany, gdy będziemy chcieli coś z tym obiektem zrobić. Technika ta pozwala na tworzenie bardziej elastycznych i ogólnych programów.

Podstawy szablonów

Teraz jednak pojawiają się problemy. Posiadamy klasę **IntStack**, przechowującą liczby całkowite. Chcemy jednak, by stos przechowywał symbole, samoloty, rośliny itd. Każdorazowa modyfikacja kodu źródłowego nie wydaje się, w przypadku języka promującego jego wielokrotne wykorzystywanie, najbardziej inteligenckim rozwiązaniem. Należy wskazać jakiś lepszy sposób.

Istnieją trzy techniki wielokrotnego wykorzystywania kodu, których można użyć w tym przypadku: stosowana w języku C (zaprezentowana tutaj dla kontrastu), używana w **Smalltalku** (która istotnie wpłynęła na język C++) oraz wykorzystywana w języku C++ — czyli szablony.

Rozwiązanie stosowane w języku C. Oczywiście, będziemy starali się unikać rozwiązania stosowanego w języku C, ponieważ jest ono chaotyczne, podatne na błędy i nieeleganckie. W podejściu tym kopujemy kod źródłowy klasy **Stack** i ręcznie wprowadzamy jego modyfikacje (i zarazem nowe błędy). Z pewnością jest to zbyt wydajna technika.

Rozwiązanie języka Smalltalk W języku **Smalltalk** (a w ślad za nim również w Javie) przyjęto proste i oczywiste rozwiązanie — jeżeli chcemy wykorzystać ponownie kod, należy użyć dziedziczenia. Aby to zrealizować, klasa każdego kontenera zawiera elementy standardowej klasy podstawowej **Object** (podobnej do przedstawionej w przykładzie, zamieszczonym pod koniec rozdziału 15.). Jednak z uwagi na to, że w języku

Smalltalk podstawowe znaczenie ma biblioteka, klasy nie są nigdy tworzone od podstaw. Muszą one zostać wyprowadzone z jakiejś istniejącej już klasy. Należy zawsze znaleźć klasę najbardziej odpowiadającą potrzebom, utworzyć jej klasę pochodną i wprowadzić w niej niezbędne zmiany. Oczywiście, jest to korzystne, ponieważ minimalizuje nakład pracy (dlatego, zanim zostanie się produktywnym programistą języka Smalltalk, trzeba spędzić wiele czasu na uczeniu się biblioteki klas).

Oznacza to jednak również, że wszystkie klasy język Smalltalk są w końcu elementami pojedynczego drzewa dziedziczenia. Tworząc nową klasę, zawsze musimy wprowadzić ją z jakiejś gałęzi tego drzewa. Większa część drzewa już istnieje (tworzą ją biblioteka klas języka Smalltalk), a jego korzeniem jest klasa o nazwie **Object** — ta sama, którą zawiera każdy kontener, dostępny w Smalltalku.

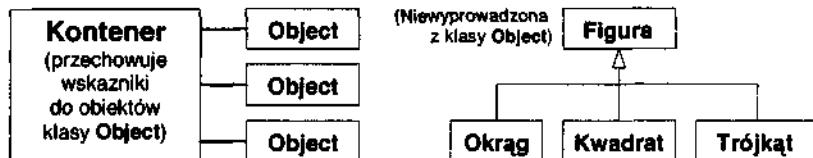
To sprytnej sztuczki, ponieważ oznacza ona, że każda klasa znajdująca się w hierarchii klas języka Smalltalk (i w Javie³) jest **klassą pochodną klasy Object**, a zatem każda klasa może być przechowywana w dowolnym kontenerze (włączając w to również sam kontener). Jest to rodzaj hierarchii o jednym korzeniu, opartej na podstawowym, ogólnym typie (noszącym często nazwę **Object**, podobnie jak w języku Java) i określanej mianem „hierarchii bazującej na obiekcie”. Być może znasz już ten termin i sądzisz, że jest to jakieś podstawowe pojęcie dotyczące programowania obiektowego, jak na przykład **polimorfizm**. W rzeczywistości odnosi się ono do hierarchii klas, w której korzeniu znajduje się klasa **Object** (albo jakaś inna, nosząca podobną nazwę). Posiada ona klasy kontenerów, przechowujące obiekty klasy **Object**.

Ponieważ biblioteka klas języka Smalltalk ma znacznie dłuższą historię i związanych jest z nią znacznie więcej doświadczeń niż w przypadku języka C++, a także z uwagi na to, że pierwotnie kompilatory języka C++ *nie zawierają* bibliotek klas kontenerowych, dobrym pomysłem wydaje się skopiowanie w języku C++ biblioteki języka Smalltalk. Zostało to wykonane w charakterze eksperymentu za pomocą wczesnej implementacji języka C++⁴, a ponieważ implementacja ta zawiera pokaźną ilość kodu, wielu programistów zaczęło jej używać. Próbujeć używać klas kontenerowych, napotkali oni jednak problem.

Problem polegał na tym, że w języku Smalltalk (a także w większości innych języków obiektowych, jakie znam) wszystkie klasy automatycznie dziedziczą po pojedynczej hierarchii, co nie jest prawdą w języku C++. Możemy posiadać elegancką hierarchię bazującą na obiekcie, wraz z jej klasami kontenerowymi, ale pewnego dnia zakupimy zestaw klas, zawierających figury lub samoloty, u innego producenta, niewykorzystującego tej hierarchii (wyłącznie dlatego, że stosowanie takiej hierarchii wiąże się z pewnym narzutem, którego wystrzegają się programiści). W jaki sposób umieścić oddzielne drzewo klas w naszej hierarchii bazującej na obiekcie? Poniżej zamieszczono rysunek, pokazujący, na czym polega problem:

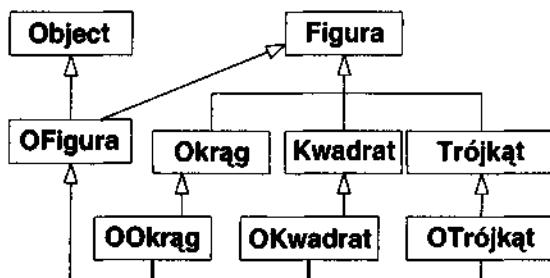
³ Z wyjątkiem prymitywnych typów danych, dostępnych w Javie. Z uwagi na efektywność, nie zostały one wyprowadzone z klasy **Object**.

⁴ Biblioteka OOPS, zaimplementowana przez Keitha Gorlena, gdy pracował w National Institutes of Health.



Z uwagi na to, że język C++ umożliwia istnienie wielu niezależnych hierarchii klas, zapożyczona z języka Smalltalk hierarchia bazująca na obiekcie, nie sprawdza się w nim zbyt dobrze.

Rozwiążanie wydaje się oczywiste. Jeżeli możemy posiadać wiele hierarchii dziedziczenia, to powinniśmy mieć możliwość dziedziczenia po więcej niż jednej klasie — problem zostanie rozwiązany przez wielokrotne dziedziczenie. Postępujemy więc następująco (podobny przykład znajduje się na końcu rozdziału 15.):



Obecnie klasa **OFigura** posiada cechy i zachowanie klasy **Figura**, ale ponieważ dziedziczy ona również po klasie **Object**, może być umieszczona w **Kontenerze**. Dodatkowe dziedziczenie klas **OOkrag**, **OKwadrat** itd. jest niezbędne, bo umożliwia ono rzutowanie tych klas w góre na klasę **OFigura**, dzięki czemu będą one poprawnie funkcjonować. A zatem sprawy zaczynają się nieco komplikować.

Producenci kompilatorów tworzyli i udostępniali swoje hierarchie klas kontenerów, bazujące na obiektach, z których większość została już zastąpiona wersjami opracowanymi z wykorzystaniem szablonów. Można twierdzić, że wielokrotne dziedziczenie jest niezbędne do rozwiązywania ogólnych problemów programistycznych, ale, jak przekonamy się w drugim tomie książki, związanej z nim złożoności lepiej unikać, wykorzystując je wyłącznie w szczególnych przypadkach.

rozwiążanie z wykorzystaniem szablonów

Mimo że hierarchia klas bazująca na obiekcie, wykorzystująca wielokrotne dziedziczenie, jest pojęciowo dość prosta, to jej stosowanie okazuje się w praktyce niewygodne. W swojej pierwszej książce⁵ Stroustrup zaprezentował to, co uważa za preferowane rozwiązanie konkurencyjne w stosunku do hierarchii bazującej na obiekcie. Klasy kontenerowe zostały utworzone jako ogromne makroinstrukcje preprocesora,

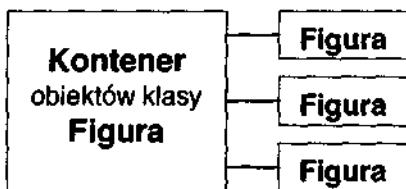
posiadające argumenty, za pomocą których można było określić pożądany typ. Gdy chcieliśmy utworzyć kontener, przechowujący określony typ, należało w tym celu wykonać kilka wywołań makroinstrukcji.

Niestety, metoda ta była pomieszana z rozwiązaniami prezentowanymi w istniejącej wówczas literaturze, dotyczącej **Smalltalka** oraz doświadczeniami programistycznymi, a poza tym był

Niestety, rozwiązanie to było mieszką metod prezentowanych w istniejącej wówczas literaturze dotyczącej **Smalltalka** i doświadczeń programistycznych, a poza tym było niezbyt poręczne. W zasadzie nikt go nie używał.

W tym czasie Stroustrup i zespół zajmujący się językiem C++ w Bell Labs zmodyfikowali oryginalne rozwiązanie z wykorzystaniem makroinstrukcji, upraszczając je i przenosząc z domeny preprocesora do kompilatora. Ten nowy mechanizm zastępowania kodu został nazwany *szablonem*⁶ (ang. *template*) — reprezentował on zupełnie inne ujęcie kwestii wielokrotnego użycia kodu. W szablonach, zamiast ponownego stosowania kodu obiektu, jak w przypadku dziedziczenia i kompozycji, wykorzystano *kod źródłowy*. Zamiast obiektów ogólnej klasy podstawowej o nazwie **Object** kontener przechowuje nieokreślony parametr. Gdy wykorzystujemy szablony, zastąpieniem tego parametru zajmuje się *kompilator* — wykonuje to podobnie do działania dawnych makroinstrukcji, ale w sposób znacznie bardziej przejrzysty i łatwiejszy w użyciu.

Obecnie, zamiast zajmować się dziedziczeniem czy kompozycją, związanymi ze stosowaniem klas kontenerowych, używamy szablonowej wersji kontenera, nadając mu postać odpowiadającą naszemu problemowi, jak na rysunku poniżej:



Kompilator wykona całą pracę za nas, czego ostatecznym efektem będzie utworzenie kontenera dokładnie spełniającego nasze wymagania. Nastapi ono bez potrzeby używania niewygodnej hierarchii dziedziczenia. Szablony dostępne w języku C++ realizują koncepcję *spараметrowanego typu*. Inna korzyść, wynikająca z działania wykonywanego przez kontenery, polega na tym, że poczynający programiści, którzy nie znają dziedziczenia (albo może ono wydawać się im niewygodne), mogą od razu przystąpić do używania klas kontenerowych (jak to uczyniliśmy z wektorami, wykorzystywany w całej książce).

⁶ Wydaje się, że inspiracją dla szablonów były **generalia języka ADA**.

Składnia szablonów

Słowo kluczowe **template** (szablon) informuje kompilator, że następująca po nim definicja klasy będzie operowała jednym lub większą liczbą niesprecyzowanych typów. Do momentu wygenerowania kodu rzeczywistej klasy na podstawie szablonu typy te muszą zostać określone, dzięki czemu kompilator będzie w stanie je zastąpić.

Poniżej znajduje się niewielki przykład, ilustrujący składnię szablonów. Zostaje utworzona tablica sprawdzająca zakres indeksów:

```
//: C16:Array.cpp
#include "../require.h"
#include <iostream>
using namespace std;

template<class T>
class Array {
    enum { size = 100 };
    T A[size];
public:
    T& operator[](int index) {
        require(index >= 0 && index < size,
               "Indeks poza zakresem");
        return A[index];
    }
};

int main() {
    Array<int> ia;
    Array<float> fa;
    for(int i = 0; i < 20; i++) {
        ia[i] = i * i;
        fa[i] = float(i) * 1.414;
    }
    for(int j = 0; j < 20; j++)
        cout << j << ":" << ia[j]
           << "," << fa[j] << endl;
} ///:~
```

A zatem przypomina to całkiem zwyczajną klasę, z wyjątkiem wiersza:

```
template<class T>
```

który oznacza, że symbol **T** stanowi parametr podstawienia oraz że reprezentuje on nazwę typu. Parametr **T** jest używany wewnątrz klasy wszędzie tam, gdzie zwykle występowałby normalny typ, przechowywany przez kontener.

W klasie **Array** elementy są wstawiane *oraz* pobierane za pomocą tej samej funkcji — przeciążonego operatora **[]**. Zwraca on referencję, więc może być używany po obu stronach znaku równości (zarówno jako **l-wartość**, jak i jako **p-wartość**). Zwróć uwagę na to, że w przypadku gdy indeks znajduje się poza zakresem, do wydrukowania komunikatu wykorzystywana jest funkcja **require()**. Ponieważ **operator[]** jest funkcją **inline**, można używać takiego sposobu w celu upewnienia się, że nie następuje naruszanie granic tablicy, a z gotowego kodu usunąć funkcję **require()**.

W funkcji **main()** łatwo utworzyć tablice przechowujące różne typy obiektów. Gdy napiszemy:

```
Array<int> ia;
Array<float> fa;
```

kompilator rozwinie szablon **Array** (jest to nazywane *tworzeniem wystąpienia lub konkretyzacją*) dwukrotnie, tworząc dwie *wygenerowane klasy*, których można traktować jako klasy **Array_int** oraz **Array_float** (różne kompilatory mogą uzupełniać nazwy na różnego sposoby). Są to klasy podobne do tych, które można by utworzyć dokonując zastąpienia własnoręcznie, z jednym wyjątkiem — kompilator tworzy je automatycznie w trakcie definicji obiektów **ia** oraz **fa**. Zwróć również uwagę na to, że powtarzaniu się nazw klas albo zapobiega kompilator, albo są one scalane przez program łączący.

Definicje funkcji niebędących funkcjami inline

Oczywiście, niekiedy chcemy, aby funkcje składowe klasy nie były definicjami funkcji inline. W takim przypadku kompilator musi „widzieć” deklarację **template** przed definicją funkcji składowej. Poniżej przedstawiono poprzedni przykład, prezentujący definicję funkcji składowej, niebędącej funkcją inline:

```
//: C16:Array2.cpp
// Definicja szablonu funkcji,
// niebędącej funkcją inline
#include "../require.h"

template<class T>
class Array {
    enum { size = 100 };
    T A[size];
public:
    T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
    require(index >= 0 && index < size,
        "Indeks poza zakresem");
    return A[index];
}

int main() {
    Array<float> fa;
    fa[0] = 1.414;
} //:~
```

Każdemu odwołaniu do nazwy klasy szablonu musi towarzyszyć lista argumentów szablonu, jak w nazwie funkcji **Array<T>::operator[]**. Można sobie wyobrazić, że wewnętrznie nazwa kasy jest uzupełniana argumentami znajdującymi się na liście argumentów szablonu, dając w wyniku unikatowy identyfikator nazwy klasy dla każdej konkretyzacji szablonu.

Iliki nagłówkowe

Nawet w przypadku tworzenia definicji funkcji niebędących funkcjami inline wszystkie deklaracje *oraz* definicje, dotyczące szablonu, będziemy zazwyczaj umieszczać w pliku nagłówkowym. Może to wyglądać na pogwałcenie zasady dotyczącej plików nagłówkowych, polegającej na tym, by „nie umieszczać w nich nic takiego, co powoduje przydzielenie pamięci” (co zapobiega błędem wielokrotnych definicji w trakcie łączenia), ale definicje zawarte w szablonach mają charakter szczególny. Poprzedzenie jakiegokolwiek elementu tekstem **template<...>** oznacza, że kompilator nie przydziela mu w tym miejscu pamięci, czekając, aż programista każe to zrobić (za pomocą konkretyzacji szablonu). Gdzieś w kompilatorze ł programie łączącym istnieje mechanizm usuwający wielokrotne definicje identycznych szablonów. Tak więc ze względu na łatwość użycia niemal zawsze będziemy umieszczać w plikach nagłówkowych (w całości) zarówno deklaracje, jak i definicje szablonów.

Zdarzają się sytuacje, w których trzeba umieścić definicję szablonu w oddzielnym pliku .cpp z uwagi na konieczność spełnienia jakichś szczególnych wymagań (na przykład wymuszając istnienie konkretyzacji szablonu w tylko jednym pliku dll systemu Windows). Większość kompilatorów jest wyposażonych w mechanizmy, które na to pozwalają — by je wykorzystać, musisz przejrzeć dokumentację używanego przez siebie kompilatora.

Niektórzy przeczuwają, że umieszczenie całego kodu źródłowego implementacji w pliku nagłówkowym umożliwia nabyciarom biblioteki kradzież i modyfikację zawartego w niej kodu. Może to stanowić problem, w zależności jednak od sposobu, w jaki postrzegasz tę kwestię — czy sprzedajesz produkt, czy też usługę. Jeżeli jest to produkt, musisz zrobić wszystko, co w twojej mocy, by go zabezpieczyć i prawdopodobnie nie będziesz chciał udostępniać nabyciarzy kodu źródłowego, a jedynie skompilowany kod. Lecz wiele osób postrzega oprogramowanie jako usługę, a nawet jako coś więcej — subskrypcję usług. Klient oczekuje twojej porady i chce, byś nadal zajmował się pielęgnacją fragmentu kodu przeznaczonego do wielokrotnego użytku. Dzięki temu sam nie będzie musiał tego robić i skupi się na wykonywaniu *swojej* pracy. Uważam, że większość klientów będzie cię traktować jako wartościowe źródło pomocy, w związku z czym nie narażą na szwank waszych stosunków. A jeżeli chodzi o tych nielicznych, którzy wolą coś ukraść niż kupić albo samodzielnie zrobić coś oryginalnego, to prawdopodobnie i tak nie będą w stanie dotrzymać ci kroku.

Klasa IntStack jako szablon

Poniżej zamieszczono kontener oraz iterator, pochodzące z pliku **IntStack.cpp**, które zostały zrealizowane jako ogólny kontener za pomocą szablonów:

```
//: C16:StackTemplate.h
// Prosty szablon stosu
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H
#include "../require.h"

template<class T>
class StackTemplate {
```

```

enum { ssize = 100 };
T stack[ssize];
int top;
public:
    StackTemplate() : top(0) {}
    void push(const T& i) {
        require(top < ssize, "Zbyt wiele wywołan funkcji push()");
        stack[top++] = i;
    }
    T pop() {
        require(top > 0, "Zbyt wiele wywołan funkcji pop()");
        return stack[--top];
    }
    int size() { return top; }
};

#endif // STACKTEMPLATE_H //:-:

```

Zwróć uwagę na to, że szablon opiera się na pewnych założeniach, dotyczących przechowywanych w nim obiektów. Na przykład klasa **StackTemplate** zakłada, we wnętrzu funkcji **push()**, że klasa T posiada jakiś rodzaj operacji przypisania. Można powiedzieć, że szablon „sugeruje interfejs” typów, który jest w stanie przechowywać.

Inaczej mówiąc, szablony tworzą rodzaj mechanizmu *słabej kontroli typów* (ang *weak typing*) w języku C++, który jest normalnie językiem o ścisłej kontroli typów. Zamiast żądać, by dopuszczalny obiekt był jakiegoś konkretnego typu, słaba kontrola typów wymagajedynie, by dla danego obiektu były *dostępne* funkcje składowe, które zamierza wywołać szablon. Tak więc kod zawierający **słabą kontrolę typów** może być zastosowany w odniesieniu do dowolnego obiektu, który akceptuje wywoływanie tych funkcji, co sprawia, że jest on znacznie bardziej **elastyczny**⁷.

Poniżej zamieszczono poprawiony kod, testujący działanie szablonu:

```

//: C16:StackTemplateTest.cpp
// Test prostego szablonu stosu
//{L} fibonacci
#include "fibonacci.h"
#include "StackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    StackTemplate<int> is;
    for(int i = 0; i < 20; i++)
        is.push(fibonacci(i));
    for(int k = 0; k < 20; k++)
        cout << is.pop() << endl;
    ifstream in("StackTemplateTest.cpp");
    assure(in, "StackTemplateTest.cpp");
}

```

⁷ Wszystkie metody w językach Smalltalk i Python są metodami o słabej kontroli typów, w związku z czym języki te nie potrzebują mechanizmu szablonów. W rezultacie w językach tych uzyskuje się szablony bez stosowania szablonów.

```
string line;
StackTemplate<string> strings;
while(getline(in, line))
    strings.push(line);
while(strings.size() > 0)
    cout << strings.pop() << endl;
} // :~
```

Jedyna różnica w stosunku do poprzedniej wersji programu polega na sposobie utworzenia obiektu `is`. Wewnątrz listy argumentów szablonu określa się typ obiektu obsługiwanej przez `stos` i przez iterator. Aby zademonstrować ogólny charakter szablonu **StackTemplate**, został również utworzony obiekt przechowujący łańcuchy. Został on przetestowany przez odczytywanie wierszy tekstu, zawartych w pliku źródłowym programu.

State w szablonach

Argumenty **szablonu** nie ograniczają się do typów klas — można używać w nich również typów wbudowanych. Wartości tych typów mogą stać się stałymi czasu komplikacji dla określonych konkretizacji szablonu. W stosunku do takich argumentów można używać nawet wartości domyślnych. Poniższy przykład pozwala na ustalenie podczas konkretyzacji wielkości klasy **Array**, lecz określa również domyślną wielkość:

```
//: C16:Array3.cpp
// Wbudowane typy jako argumenty szablonów
#include "../require.h"
#include <iostream>
using namespace std;

template<class T, int size = 100>
class Array {
    T array[size];
public:
    T& operator[](int index) {
        require(index >= 0 && index < size,
               "Indeks poza zakresem");
        return array[index];
    }
    int length() const { return size; }
};

class Number {
    float f;
public:
    Number(float ff = 0.0f) : f(ff) {}
    Number& operator=(const Number& n) {
        f = n.f;
        return *this;
    }
    operator float() const { return f; }
    friend ostream&
    operator<<(ostream& os, const Number& x) {
        return os << x.f;
    }
};
```

```

template<class T, int size = 20>
class Holder {
    Array<T, size>* np;
public:
    Holder() : np(0) {}
    T& operator[](int i) {
        require(0 <= i && i < size);
        if(!np) np = new Array<T, size>;
        return np->operator[](i);
    }
    int length() const { return size; }
    ~Holder() { delete np; }
};

int main() {
    Holder<Number> h;
    for(int i = 0; i < 20; i++)
        h[i] = i;
    for(int j = 0; j < 20; j++)
        cout << h[j] << endl;
} //:-)

```

Podobnie jak poprzednio, klasa `Array` jest kontrolowaną tablicą obiektów, uniemożliwiającą przekroczenie zakresu indeksów. Klasa `Holder` przypomina klasę `Array`, z wyjątkiem tego, że zawiera ona wskaźnik do obiektu klasy `Array`, a nie osadzony wewnątrz obiekt tej klasy. Wskaźnik ten niejest inicjalizowany wewnątrz konstruktora — inicjalizacja jest *odłożona* do momentu pierwszego dostępu do kontenera. Technika ta nosi nazwę *leniwej inicjalizacji*(ang. *lazy initialization*) — możemy ją stosować w przypadku, gdy tworzymy wiele obiektów, ale nie odwołujemy się do nich wszystkich i zależy nam na oszczędzaniu pamięci.

Można zauważyc, że w przypadku żadnego z szablonów wartość parametru `size` nie jest zapisywana wewnątrz klasy. Wykorzystuje się ją w taki sposób, jakby była skądową, używaną wewnątrz funkcji składowych.

Klasy Stack i Stash jako szablony

Powracające problemy „prawa własności”, dotyczące klas kontenerów `Stash` i `Stack`, które były rozważane w książce, wynikają z faktu, że kontenery te nie mogły dokładnie „wiedzieć”, jakiego dokładnie typu są przechowywane w nich obiekty. Najbliższym rozwiązania problemu był przykład klasy `Stack` jako „kontenera obiektów klasy `Object`”, zaprezentowany w programie `OStackTest.cpp`, zamieszczonym w rozdziale 15.

W przypadku gdy **klient-programista** nie usunie jawnie wszystkich wskaźników do obiektów, przechowywanych w kontenerze, kontener powinien poprawnie usunąć te wskaźniki. Innymi słowy, kontener jest „właścicielem” wszystkich obiektów, które nie zostały z niego usunięte, odpowiada zatem za ich usunięcie. Problem polega na tym, że usuwanie wymaga znajomości typu obiektu, natomiast *nie jest* ona potrzebna do **utworzenia** ogólnej klasy kontenerowej. Jednakże w przypadku szablonów możemy napisać

kod nieznający typów obiektów, bez trudu konkretyzując nowe wersje kontenera dla każdego typu, który będziemy chcieli w nim przechowywać. Indywidualnie skonkretyzowane kontenery *znają* typ przechowywanych przez siebie obiektów, dlatego też mogą wywołać dla nich właściwe destruktory (zakładając w typowym przypadku, polegającym na zastosowaniu polimorfizmu, że dostępne są wirtualne destruktory).

W przypadku klasy **Stack** okazuje się to dość proste, ponieważ wszystkie jej funkcje składowe mogą być bez trudu zdefiniowane jako funkcje inline:

```
//: C16:TStack.h
// Klasa Stack jako szablon
#ifndef TSTACK_H
#define TSTACK_H

template<class T>
class Stack {
    struct Link {
        T* data;
        Link* next;
        Link(T* dat, Link* nxt):
            data(dat), next(nxt) {}
    }* head;
public:
    Stack() : head(0) {}
    ~Stack(){
        while(head)
            delete pop();
    }
    void push(T* dat) {
        head = new Link(dat, head);
    }
    T* peek() const {
        return head ? head->data : 0;
    }
    T* pop(){
        if(head == 0) return 0;
        T* result = head->data;
        Link* oldHead = head;
        head = head->next;
        delete oldHead;
        return result;
    }
};

#endif // TSTACK_H ///:-
```

Jeżeli porównamy powyższy plik z przykładem zamieszczonym w pliku nagłówkowym **OStack.h**, znajdującym się na końcu rozdziału 15., zobaczymy, że klasa **Stack** właściwie wcale się nie zmieniła, z wyjątkiem tego, że klasa **Object** została zastąpiona przez **T**. Program testowy jest również prawie taki sam, oprócz tego, że wyeliminowano w nim konieczność wielokrotnego dziedziczenia po klasach **string** oraz **Object** klasy określającej typ przechowywanych obiektów (a nawet konieczność istnienia samej klasy **Object**). Ponieważ nie ma obecnie klasy **MyString**, sygnalizującej wywołanie swojego destruktora, dodano niewielką klasę, dzięki której można się przekonać o tym, że kontener **Stack** usuwa znajdujące się w nim obiekty:

```

//: C16:TStackTest.cpp
//{T} TStackTest.cpp
#include "TStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class X {
public:
    virtual ~X() { cout << "~X" << endl; }
};

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // Argumentem jest nazwa pliku
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack<string> textlines;
    string line;
    // Odczytywanie pliku i zapamiętywanie wierszy na stosie:
    while(getline(in, line))
        textlines.push(new string(line));
    // Pobranie kilku wierszy ze stosu:
    string* s;
    for(int i = 0; i < 10; i++) {
        if((s - (string*)textlines.pop())==0) break;
        cout << *s << endl;
        delete s;
    } // Pozostałe łańcuchy usunie destruktor.
    // Prezentacja poprawnej destrukcji:
    Stack<X> xx;
    for(int j = 0; j < 10; j++)
        xx.push(new X);
} //:-

```

Destruktor klasy X jest destruktorem wirtualnym — nie dlatego, że jest w tym przypadku niezbędny, ale ponieważ obiekty tej klasy mogłyby w przyszłości zostać wykorzystane do przechowywania obiektów klas dziedziczących po klasie X.

Zwróćmy uwagę na to, jak łatwo można, używając klasy **Stack**, tworzyć rozmaite wersje stosów — zarówno dla łańcuchów, jak i obiektów klasy X. Dzięki szablonom połączymy ze sobą to, co najlepsze — łatwość użycia klasy **Stack** z zapewnieniem poprawnego usuwania obiektów.

Kontener wskaźników **Stash**, wykorzystujący szablony

Przekształcenie kodu klasy **PStash** w wersję wykorzystującą szablony nie jest już tak proste, ponieważ klasa ta zawiera kilka funkcji, które nie powinny być funkcjami inline. Jednakże definicje tych funkcji w postaci szablonów nadal mogą być umieszczone w pliku nagłówkowym (komplilator oraz program łączący zapewnia rozwiązanie wszelkich problemów związanych z wielokrotnymi definicjami). Kod wygląda

bardzo podobnie do kodu zwykłej klasy **PStash**, z wyjątkiem tego, że wielkość, o której powiększany jest przydzielony obszar pamięci (używana przez funkcję **inflate()**), została włączona do szablonu jako niebędący klasą parametr o określonej wartości domyślnej. Dzięki temu jego wartość może być modyfikowana w miejscu konkretyzacji (oznacza to jednak, że wartość ta jest stała — można również twierdzić, że powinna istnieć możliwość jej zmiany w czasie życia obiektu):

```
//: C16:TPStash.h
#ifndef TPSTASH_H
#define TPSTASH_H

template<class T, int incr = 10>
class PStash {
    int quantity; // Liczba elementów pamięci
    int next; // Następny pusty element
    T** storage;
    void inflate(int increase = incr);
public:
    PStash() : quantity(0). next(0). storage(0) {}
    ~PStash();
    int add(T* element);
    T* operator[](int index) const; // Pobranie elementu
    // Usunięcie odwołania do elementu:
    T* remove(int index);
    // Liczba zapamiętanych elementów:
    int count() const { return next; }
};

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
    if(next >= quantity)
        inflate(incr);
    storage[next++] = element;
    return(next - 1); // Numer indeksu
}

// Pozostawione obiekty są własnością kontenera:
template<class T, int incr>
PStash<T, incr>::~PStash() {
    for(int i = 0; i < next; i++) {
        delete storage[i]; // Zerowe wskaźniki nie są problemem
        storage[i] = 0; // Dla pewności
    }
    delete []storage;
}

template<class T, int incr>
T* PStash<T, incr>::operator[](int index) const {
    require(index >= 0,
        "PStash::operator[] indeks ma wartosc ujemna");
    if(index > next)
        return 0; // Oznaczenie końca
    require(storage[index] != 0,
        "PStash::operator[] zwrócony pusty wskaznik");
    // Tworzenie wskaźnika do żądanego elementu:
    return storage[index];
}
```

```

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
    // operator[] dokonuje sprawdzenia poprawności indeksu:
    T* v = operator[](index);
    // "Usunięcie" wskaźnika:
    if(v != 0) storage[index] = 0;
    return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
    const int psz = sizeof(T*);
    T** st = new T*[quantity + increase];
    memset(st, 0, (quantity + increase) * psz);
    memcpy(st, storage, quantity * psz);
    quantity += increase;
    delete []storage; // Stary obszar pamięci
    storage = st; // Wskaźnik do nowego obszaru pamięci
}
#endif // TPSTASH_H //:-

```

Domyślna wartość, o jaką zwiększany jest przydzielony obszar pamięci, jest niewielka, co gwarantuje wywoływanie funkcji **inflate()**. Dzięki temu możemy się upewnić, że działa ona poprawnie.

Aby sprawdzić kontrolę prawa własności w wersji klasy **PStash**, wykorzystującej szablony, została utworzona, widoczna poniżej, klasa **AutoCounter**. Informuje ona o wywołaniu swojego konstruktora i destruktora, co pozwala na upewnienie się, że wszystkie utworzone obiekty zostały również zniszczone. Klasa **AutoCounter** umożliwia wyłącznie tworzenie obiektów swojego typu na stercie:

```

//: C16:AutoCounter.h
#ifndef AUTOCOUNTER_H
#define AUTOCOUNTER_H
#include "../require.h"
#include <iostream>
#include <set> // Kontener standardowej biblioteki C++
#include <string>

class AutoCounter {
    static int count;
    int id;
    class CleanupCheck {
        std::set<AutoCounter*> trace;
    public:
        void add(AutoCounter* ap) {
            trace.insert(ap);
        }
        void remove(AutoCounter* ap) {
            require(trace.erase(ap) == 1,
                    "Próba dwukrotnego usunięcia obiektu klasy AutoCounter");
        }
    ~CleanupCheck() {
        std::cout << "~CleanupCheck()" << std::endl;
        require(trace.size() == 0,
                "Nie wszystkie obiekty klasy AutoCounter zostały usunięte");
    }
}

```

```

};

static CleanupCheck verifier;
AutoCounter() : id(count++) {
    verifier.add(this); // Rejestracja samego siebie
    std::cout << "utworzony[" << id << "]"
        << std::endl;
}
// Zapobieganie przypisaniu i wywołaniu konstruktora kopiącego:
AutoCounter(const AutoCounter&);
void operator=(const AutoCounter&);

public:
    // Obiekty można tworzyć tylko za pomocą tej funkcji:
    static AutoCounter* create() {
        return new AutoCounter();
    }
    ~AutoCounter() {
        std::cout << "usuwanie[" << id
            << "]" << std::endl;
        verifier.remove(this);
    }
    // Wydruk zarówno obiektów, jak i wskaźników:
    friend std::ostream& operator<<(
        std::ostream& os, const AutoCounter& ac){
        return os << "AutoCounter " << ac.id;
    }
    friend std::ostream& operator<<(
        std::ostream& os, const AutoCounter* ac){
        return os << "AutoCounter " << ac->id;
    }
};

#endif // AUTOCOUNTER_H //:-

```

Klasa **AutoCounter** realizuje dwa zadania. Po pierwsze, kolejno numeruje każdy utworzony obiekt klasy **AutoCounter** — numer każdego z nich jest przechowywany w składowej **id** i jest on generowany za pomocą statycznej składowej **count**.

Drugie zadanie jest bardziej złożone. Statyczny egzemplarz zagnieżdzonej klasy **CleanupCheck** (noszący nazwę **verifier**) zapamiętuje wszystkie tworzone i niszczone obiekty klasy **AutoCounter** i zgłasza komunikat, w przypadku gdy nie wszystkie obiekty zostały przez nas usunięte (czyli nastąpił wyciek pamięci). Działanie to jest realizowane za pomocą klasy **set** (zbior), pochodzącej ze standardowej biblioteki języka C++. Stanowi ona wspaniały przykład tego, jak przydatne są dobrze zaprojektowane szablony (o wszystkich kontenerach, zawartych w standardowej bibliotece klas języka C++, można przeczytać w drugim tomie książki, dostępnym w Internecie).

Parametrem szablonu klasy **set** jest przechowywany typ — w tym przypadku został on skonkretyzowany w celu przechowywania wskaźników w **obiektach** klasy **AutoCounter**. Klasa **set** pozwala na dodanie do zbioru tylko jednego egzemplarza każdego unikatowego obiektu — w funkcji **add()** odbywa się to za pomocą wywołania funkcji **set::insert()**. Funkcja **insert()** w rzeczywistości zwraca wartość, która zawiera informację o ewentualnej próbie dodania do zbioru czegoś, cojuż wcześniej zostało do niego dołączone. Jednakże w tym przypadku, ponieważ dodajemy jedynie adresy obiektów, możemy polegać na udzielanej przez język C++ gwarancji, że wszystkie obiekty posiadają unikatowe adresy.

W funkcji **remove()** do usunięcia ze zbioru wskaźnika do obiektu klasy **AutoCounter** jest wykorzystywana funkcja **set::erase()**. Zwracana wartość informuje, ile egzemplarzy danego elementu zostało usuniętych — w naszym przypadku spodziewamy się tylko wartości wynoszącej zero lub jeden. Jeżeli jednak wartość wynosiła zero, oznacza to, że obiekt został już usunięty ze zbioru i próbujemy wyeliminować go z niego po raz drugi, co stanowi błąd w programie, który zostanie zgłoszony za pośrednictwem funkcji **require()**.

Destruktor klasy **CleanupCheck** dokonuje ostatecznego sprawdzenia, upewniając się, że wielkość zbioru jest zerowa, czyli wszystkie obiekty zostały poprawnie usunięte. Jeżeli nie jest ona zerowa, oznacza to wyciek pamięci, co jest sygnalizowana za pomocą funkcji **require()**.

Konstruktor oraz destruktor klasy **AutoCounter** — odpowiednio rejestruje i wyrejestrowuje się, używając do tego celu obiektu **verifier**. Zwróć uwagę na to, że konstruktor, konstruktor kopiący oraz operator przypisania są prywatne, dzięki czemu jedynym sposobem utworzenia obiektu jest wywołanie statycznej funkcji składowej **create()**. Stanowi to prosty przykład *fabryki* (ang. *factory*) i gwarantuje tworzenie wszystkich obiektów na stercie, dzięki czemu obiekt **verifier** nie będzie niepokojony przypisiami i konstrukcjami realizowanymi za pomocą konstruktora kopiącego.

Ponieważ wszystkie funkcje składowe są funkcjami inline, jedynym powodem utworzenia pliku, zawierającego implementację, jest umieszczenie w nim definicji statycznych danych składowych:

```
//: C16:AutoCounter.cpp {0}
// Definicje statycznych danych składowych klasy
#include "AutoCounter.h"
AutoCounter::CleanupCheckAutoCounter::verifier;
int AutoCounter::count = 0;
///:-
```

Dysponując klasą **AutoCounter**, możemy obecnie przetestować usługi, udostępniane przez klasę **PStash**. Poniższy przykład nie tylko dowodzi tego, że destruktor klasy **PStash** usuwa wszystkie obiekty, które znajdują się aktualnie w jego posiadaniu, ale pokazuje też, w jaki sposób klasa **AutoCounter** wykrywa wszystkie niusunięte obiekty.

```
//: C16:TPStashTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "TPStash.h"
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    PStash<AutoCounter> acStash;
    for(int i = 0; i < 10; i++)
        acStash.add(AutoCounter::create());
    cout << "Reczne usunięcie 5 elementów:" << endl;
    for(int j = 0; j < 5; j++)
        delete acStash.remove(j);
```

```

cout << "Usuniecie dwóch elementów bez ich niszczenia:"
    << endl;
// ... wymuszenie generacji komunikatów o błędach.
cout << acStash.remove(5) << endl;
cout << acStash.remove(6) << endl;
cout << "Reszte usuwa destruktor:"
    << endl;
// Powtórzenie testów z wcześniejszych rozdziałów:
ifstream in("TPStashTest.cpp");
assure(in, "TPStashTest.cpp");
PStash<string> stringStash;
string line;
while(getline(in, line))
    stringStash.add(new string(line));
// Drukowanie łańcuchów:
for(int u = 0; stringStash[u]; u++)
    cout << "stringStash[" << u << "] - "
        << *stringStash[u] << endl;
} //:-
```

Gdy z kontenera **PStash** usuwany jest piąty i szósty element klasy **AutoCounter**, odpowiedzialność za nie przejmuje użytkownik. Ponieważ jednak nigdy ich nie niszczy, powodują one wyciekanie pamięci, które jest wykrywane przez klasę **AutoCounter** w trakcie pracy programu.

Po uruchomieniu programu zauważysz, że wyświetlany komunikat o błędzie nie jest dość precyzyjny. Jeżeli w swoich programach używasz mechanizmu zaprezentowanego na przykładzie klasy **AutoCounter**, zapewne będziesz chciał wyświetlić bardziej szczegółowe informacje, dotyczące obiektów, które nie zostały usunięte. W drugim tomie książki zaprezentowano bardziej wyrafinowane sposoby, które to umożliwiają.

Przydzielanie i odbieranie prawa własności

Powróćmy do problemu prawa własności. Kontenery przechowujące obiekty w postaci wartości nie muszą na ogół **uwzględniać** prawa własności obiektów, ponieważ w oczywisty sposób są właścicielami przechowywanych w nich obiektów. Jeżeli jednak kontener przechowuje wskaźniki (co zdarza się w języku C++ częściej, głównie z uwagi na **polimorfizm**), to jest bardzo prawdopodobne, że wskaźniki te są jeszcze używane w jakimś innym miejscu programu. Usuwanie obiektu nie jest zatem konieczne, ponieważ do usuniętego obiektu mogłyby się odwoływać inne wskaźniki, znajdujące się w programie. Aby temu zapobiec, należy zastanowić się nad prawem własności podczas projektowania i używania kontenera.

Wiele programów jest znacznie prostszych i nie występuje w nich problem związany z prawem własności — wskaźniki do obiektów przechowuje pojedynczy kontener i są one wykorzystywane wyłącznie przez niego. W takich przypadkach prawo własności jest określone **bardzo** prosto — właścicielem obiektów jest kontener.

Najlepszym sposobem rozwiązyania problemu prawa własności jest pozostawienie wyboru **klientowi-programiście**. Często jest to realizowane za pomocą argumentu konstruktora, który domyślnie określa prawo własności (to najprostszy przypadek). Mogą również istnieć funkcje typu „pobierz” i „ustaw”, umożliwiające sprawdzenie i zmianę prawa własności kontenera. Jeżeli kontener posiada funkcję, usuwającą obiekty, stan prawa własności na ogół wpływa na ten proces. A zatem opcje sterujące niszczeniem obiektów można również znaleźć w funkcjach usuwających elementy z kontenera. Można wyobrazić sobie dodanie danych, określających prawo własności każdego obiektu znajdującego się w kontenerze, dzięki czemu byłoby wiadomo, czy musi on zostać zniszczony. Stanowi to wariant zliczania odwołań, z wyjątkiem tego, że to kontener, a nie obiekt, zna liczbę odwołań, wskazujących każdy obiekt:

```
//: C16:OwnerStack.h
// Stos z prawem własności określonym
// podczas pracy programu
#ifndef OWNERSTACK_H
#define OWNERSTACK_H

template<class T> class Stack {
    struct Link {
        T* data;
        Link* next;
        Link(T* dat, Link* nxt)
            : data(dat), next(nxt) {}
    }* head;
    bool own;
public:
    Stack(bool own = true) : head(0), own(own) {}
    ~Stack();
    void push(T* dat) {
        head = new Link(dat,head);
    }
    T* peek() const {
        return head ? head->data : 0;
    }
    T* pop();
    bool ownsC() const { return own; }
    void owns(bool newownership) {
        own = newownership;
    }
    // Automatyczna konwersja typu - zwarca
    // wartość true. Jeżeli stos nie jest pusty:
    operator bool() const { return head != 0; }
};

template<class T> T* Stack<T>::pop() {
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}
```

```
template<class T> Stack<T>::~Stack() {
    if(!own) return;
    while(head)
        delete pop();
}
#endif // OWNERSTACK_H //:-:
```

Domyślnym zachowaniem kontenera jest niszczenie obiektów, ale można je zmienić — albo modyfikując argument konstruktora, albo używając funkcji składowych **owns()**, umożliwiających przydzielanie i odbieranie prawa własności.

Jak w przypadku większości szablonów, cała implementacja jest zawarta w pliku nagłówkowym. Poniżej zamieszczono niewielki test, sprawdzający możliwości kontenera, związane z prawem własności:

```
//: C16:OwnerStackTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "OwnerStack.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    Stack<AutoCounter> ac; // Przydzielenie prawa własności
    Stack<AutoCounter> ac2(false); // Odebranie prawa własności
    AutoCounter* ap;
    for(int i = 0; i < 10; i++) {
        ap = AutoCounter::create();
        ac.push(ap);
        if(i % 2 == 0)
            ac2.push(ap);
    }
    while(ac2)
        cout << ac2.pop() << endl;
    // Nie jest konieczne niszczenie obiektów,
    // ponieważ kontener ac jest "właścicielem"
    // wszystkich obiektów
} //:-:
```

Kontener **ac2** nie jest właścicielem umieszczanych w nim obiektów, w związku z czym „nadziednym” kontenerem jest obiekt **ac**, który „ponosi odpowiedzialność”, związaną z prawem własności umieszczonych w nim obiektów. Jeżeli w trakcie istnienia kontenera zachodzi potrzeba zmiany jego prawa własności, to można wykorzystać do tego celu funkcję **owns()**.

Możliwa byłaby również indywidualna zmiana prawa własności, tak by dotyczyła ona każdego obiektu z osobna. Prawdopodobnie jednak uczyniłaby ona rozwiązanie problemu prawa własności bardziej skomplikowanym niż sam problem.

Przechowywanie obiektów jako wartości

W przypadku gdy nie dysponuje się szablonami, tworzenie kopii obiektów wewnątrz ogólnych kontenerów stanowi poważny problem. Gdy używa się szablonów, wszystko jest względnie proste — wystarczy określić, że zamiast wskaźników przechowuje się obiekty:

```
//: C16:ValueStack.h
// Stos przechowujący wartości obiektów
#ifndef VALUESTACK_H
#define VALUESTACK_H
#include "../require.h"

template<class T, int ssize = 100>
class Stack {
    // Domyślny konstruktor dokonuje inicjalizacji
    // obiektu dla każdego elementu zawartego w tablicy:
    T stack[ssize];
    int top;
public:
    Stack() : top(0) {}
    // Konstruktor kopiujący kopiuje obiekty do tablicy:
    void push(const T& x) {
        require(top < ssize, "Zbyt wiele wywołan funkcji push()");
        stack[top++] = x;
    }
    T peek() const { return stack[top]; }
    // Po pobraniu ze stosu, obiekty nadal istnieją
    // - nie są już tylko dostępne:
    T pop() {
        require(top > 0, "Zbyt wiele wywołan funkcji pop()");
        return stack[--top];
    }
};

#endif // VALUESTACK_H ///:-
```

Większość pracy wykonuje konstruktor kopiujący obiektów, przechowywanych w kontenerze — pobierając i zwracając obiekty przez wartość. Wewnątrz funkcji **push()** umieszczenie obiektu w tablicy obsługującej stos jest dokonywane za pomocą funkcji **T::operator=**. Aby sprawdzić, czy kontener działa poprawnie, utworzono **klasę SelfCounter, zapamiętującą tworzenie obiektów** oraz wywołania konstruktora kopiującego:

```
//: C16:SelfCounter.h
#ifndef SELFCOUNTER_H
#define SELFCOUNTER_H
#include "ValueStack.h"
#include <iostream>

class SelfCounter {
    static int counter;
    int id;
public:
```

```

SelfCounter() : id(counter++) {
    std::cout << "Utworzony: " << id << std::endl;
}
SelfCounter(const SelfCounter& rv) : id(rv.id){
    std::cout << "Skopiowany: " << id << std::endl;
}
SelfCounter operator=(const SelfCounter& rv) {
    std::cout << "Przypisany " << rv.id << " do "
        << id << std::endl;
    return *this;
}
~SelfCounter() {
    std::cout << "Zniszczony: " << id << std::endl;
}
friend std::ostream& operator<<(std::ostream& os, const SelfCounter& sc){
    return os << "SelfCounter: " << sc.id;
}
};

#endif // SELF COUNTER_H // :~

//: C16:SelfCounter.cpp {0}
#include "SelfCounter.h"
int SelfCounter::counter = 0; // :-

//: C16:ValueStackTest.cpp
//{L} SelfCounter
#include "ValueStack.h"
#include "SelfCounter.h"
#include <iostream>
using namespace std;

int main() {
    Stack<SelfCounter> sc;
    for(int i = 0; i < 10; i++)
        sc.push(SelfCounter());
    // Wywołanie funkcji peek() jest prawidłowe,
    // wynik jest obiektem tymczasowym:
    cout << sc.peek() << endl;
    for(int k = 0; k < 10; k++)
        cout << sc.pop() << endl;
}
///-

```

Podczas tworzenia konteneru **Stack** dla każdego obiektu zawartego w tablicy jest wywoływany domyślny konstruktor klasy, której obiekty są przechowywane w kontenerze. Po uruchomieniu programu bez widocznego powodu jest tworzonych 100 obiektów klasy **SelfCounter** — ale jest to jedynie inicjalizacja tablicy. Może się ona okazać nieco kosztowna, ale nie ma sposobu, by jej uniknąć w tak prostym przykładzie. Pojawią się jeszcze bardziej złożone problemy, gdy uczyni się kontener **Stack** bardziej ogólnym, pozwalając mu na dynamiczne powiększanie swojego rozmiaru. W przedstawionej powyżej implementacji wymagałoby to bowiem utworzenia nowej (większej) tablicy, skopiowania elementów zawartych w starej tablicy do nowej tablicy, a następnie usunięcia starej tablicy (tak właśnie działa klasa **vector**, zawarta w standardowej bibliotece języka C++).

Wprowadzenie do iteratorów

Iterator jest obiektem, który porusza się w obrębie kontenera innych obiektów. Udoszczęnia on za każdym razem pojedynczy, zawarty w kontenerze obiekt, nie zapewniając jednak bezpośredniego dostępu do implementacji tego kontenera. Iteratory stanowią standardowy sposób dostępu do elementów, niezależnie od tego, czy kontener umożliwia bezpośredni dostęp do swoich elementów. Iteratory są najczęściej stosowane wraz z klasami kontenerów — stanowią one podstawową koncepcję, zawartą w projekcie i sposobie wykorzystywania standardowych kontenerów języka C++, które zostały wyczerpująco opisane w drugim tomie książki (można go pobrać z witryny <http://helion.pl/online/thinking/index.html>). Iteratory są również rodzajem wzorców projektowych, będących tematem jednego z rozdziałów drugiego tomu książki.

Pod wieloma względami iteratory są „sprytnymi wskaźnikami” i, jak się przekonamy, naśladują one na ogół operacje realizowane przez wskaźniki. Jednak, w odróżnieniu od wskaźników, iteratory zostały zaprojektowane w bezpieczny sposób, więc znacznie mniej prawdopodobne jest w ich przypadku wykonanie operacji odpowiadającej przekroczeniu zakresu indeksów tablicy (albo, jeżeli już się to zrobi, łatwiej będzie się można o tym dowiedzieć).

Powróćmy do pierwszego przykładu w rozdziale. Oto jak wygląda po dodaniu do niej prostego iteratora:

```
//: C16:IterIntStack.cpp
// Prosty stos liczb całkowitych z iteratorami
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
    enum { ssize = 100 };
    int stack[ssize];
    int top;
public:
    IntStack() : top(0) {}
    void push(int i) {
        require(top < ssize, "Zbyt wiele wywolan funkcji push()");
        stack[top++] = i;
    }
    int pop() {
        require(top > 0, "Zbyt wiele wywolan funkcji pop()");
        return stack[--top];
    }
    friend class IntStackIter;
};

// Iterator przypomina "sprytny" wskaźnik:
class IntStackIter {
    IntStack& s;
    int index;
```

```

public:
    IntStackIter(IntStack& is) : s(is). index(0) {}
    int operator++() { // Przedrostkowy
        require(index < s.top,
            "iterator przesunięty poza zakres");
        return s.stack[++index];
    }
    int operator++(int) { // Przyrostkowy
        require(index < s.top,
            "iterator przesunięty poza zakres");
        return s.stack[index++];
    }
};

int main() {
    IntStack is;
    for(int i = 0; i < 20; i++)
        is.push(fibonacci(i));
    // Przejście przez kontener za pomocą iteratora:
    IntStackIter it(is);
    for(int j = 0; j < 20; j++)
        cout << it++ << endl;
} //:-
```

Iterator **IntStackIter** został utworzony wyłącznie w celu współpracy z kontenerem **IntStack**. Zwróć uwagę na to, że klasa **IntStackIter** jest klasą zaprzyjaźnioną (**friend**) klasy **IntStack**, co daje jej dostęp do prywatnych elementów klasy **IntStack**.

Podobnie jak w przypadku wskaźnika, zadaniem klasy **IntStackIter** jest poruszanie się w obrębie obiektu klasy **IntStack** i zwracanie wartości. W tym prostym przykładzie iterator **IntStackIter** potrafi przemieszczać się tylko do przodu (używając zarówno przedrostkowej, jak i przyrostkowej postaci operatora `++`). Nie obowiązują jednak żadne ograniczenia dotyczące tego, w jaki sposób **iterator** może być zdefiniowany, z wyjątkiem tych związanych z kontenerem, z którym współpracuje. Jest jak najbardziej możliwe (z uwzględnieniem ograniczeń, narzuconych przez kontener), że iterator porusza się w dowolnym kierunku w obrębie stwarzyszonego ze sobą kontenera, a także że umożliwia modyfikację elementów przechowywanych przez kontener.

Tradycyjnie iterator jest tworzony za pomocą konstruktora, który przyłącza go do obiektu pojedynczego kontenera; w czasie swojego życia iterator nie jest również zazwyczaj przyłączany do żadnego innego kontenera (iteratory są zazwyczaj małe i niewiele „kosztują” — zawsze można więc w łatwy sposób utworzyć nowy iterator).

Korzystając z iteratora, można przejść przez elementy umieszczone na stosie, nie pobierając ich — w taki sam sposób, w jaki wskaźnik przebiega przez kolejne elementy tablicy. Jednakże iterator zna wewnętrzną strukturę stosu i sposób, w jaki można poruszać się w obrębie jego elementów. Mimo że poruszamy się w obrębie stosu, mając złudzenie „**inkrementacji wskaźnika**”, w rzeczywistości podejmowane są bardziej skomplikowane działania. Oto istota iteratorów — upraszczają one złożony proces, polegający na przemieszczaniu się od jednego elementu kontenera do drugiego, przedstawiając go w **sposób**, który przypomina używanie wskaźnika. Celem jest, by każdy iterator zawarty w naszym programie posiadał taki sam interfejs, dzięki czemu

kod wykorzystujący iterator nie musi zwracać uwagi na to, co on wskazuje. „Wie” po prostu, że może zmieniać pozycje wszystkich **iteratorów** w taki sam **sposób**, co sprawia, że kontener, wskazywany przez iterator, staje się nieważny. Można zatem tworzyć bardziej ogólny kod. Wszystkie kontenery i algorytmy, zawarte w standardowej bibliotece języka C++, opierają się na tej zasadzie dotyczącej iteratorów.

Aby wszystko uczynić nieco bardziej ogólnym, należałoby stwierdzić: „każdy kontener posiada związaną ze sobą klasę, noszącą nazwę **iterator**”. Jednakże w typowym przypadku spowodowałoby to problemy, związane z nazwami. Rozwiążaniem jest dodanie do klasy każdego kontenera zagnieżdzonego iteratora (zwróć uwagę na to, że w tym przypadku nazwa „**iterator**” rozpoczyna się małą literą, dzięki czemu jest zgodna z konwencją stosowaną w standardowej bibliotece C++). Poniżej przedstawiono wersję programu **IterIntStack.cpp**, w której w klasie kontenera zagnieżdżono **iterator**:

```
//: C16:NestedIterator.cpp
// Zagnieżdżenie iteratora w kontenerze
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class IntStack {
    enum { ssize = 100 };
    int stack[ssize];
    int top;
public:
    IntStack() : top(0) {}
    void push(int i) {
        require(top < ssize, "Zbyt wiele wywołan funkcji push()");
        stack[top++] = i;
    }
    int pop() {
        require(top > 0, "Zbyt wiele wywołan funkcji pop()");
        return stack[--top];
    }
    class iterator;
    friend class iterator;
    class iterator {
        IntStack& s;
        int index;
public:
        iterator(IntStack& is) : s(is), index(0) {}
        // Do utworzenia iteratora-wartownika
        // będącego "znacznikiem końca":
        iterator(IntStack& is, bool)
            : s(is), index(s.top) {}
        int current() const { return s.stack[index]; }
        int operator++() { // Przedrostkowy
            require(index < s.top,
                "iterator przesunięty poza zakres");
            return s.stack[++index];
    }
}
```

```

int operator++(int) { // Przyrostkowy
    require(index < s.top,
        "iterotor przesunięty poza zakres");
    return s.stack[index++];
}
// Przeskoczenie iteratorem do przodu:
iterator& operator+=(int amount) {
    require(index + amount < s.top,
        "IntStack::iterator::operator+=() "
        "proba przekroczenia zakresu");
    index += amount;
    return *this;
}
// Aby sprawdzić, czy znajdujemy się na końcu:
bool operator==(const iterator& rv) const {
    return index == rv.index;
}
bool operator!=(const iterator& rv) const {
    return index != rv.index;
}
friend ostream&
operator<<(ostream& os, const iterator& it) {
    return os << it.current();
}
};

iterator begin() { return iterator(*this); }
// Tworzenie "znacznika końca":
iterator end() { return iterator(*this, true); }
};

int main() {
    IntStack is;
    for(int i = 0; i < 20; i++)
        is.push(fibonacci(i));
    cout << "Przejście przez cały kontener IntStack\n";
    IntStack::iterator it = is.begin();
    while(it != is.end())
        cout << *it++ << endl;
    cout << "Przejście przez część kontenera IntStack\n";
    IntStack::iterator
        start = is.begin(), end = is.begin();
    start += 5, end += 15;
    cout << "początek = " << start << endl;
    cout << "koniec = " << end << endl;
    while(start != end)
        cout << *start++ << endl;
} //:~
```

Podczas tworzenia zagnieżdzonej, zaprzyjaźnionej (**friend**) klasy należy najpierw **zadeklarować jej** nazwę, następnie **zadeklarować ją jako klasę zaprzyjaźnioną**, a dopiero potem zdefiniować samą klasę. W innym przypadku kompilator tego nie zrozumie.

Iterator wzbogacono o nowe elementy. Funkcja **składowa current()** zwraca element kontenera, który jest aktualnie wybrany przez iterator. Można „przeskoczyć” iteratorem do przodu, o dowolną liczbę elementów, wykorzystując do tego celu **operator+=**. Dostępne są również dwa przeciążone operatory: **==** i **!=**, porównujące ze sobą

dwa iteratory. Mogą one zestawiać ze sobą dwa dowolne obiekty klasy **IntStack::iterator**, ale utworzono je głównie z myślą o sprawdzeniu, czy iterator znajduje się na końcu ciągu elementów — w taki sam sposób, jak czynią to „prawdziwe” iteratory standardowej biblioteki języka C++. Pomyśl polega na tym, że dwa iteratory wyznaczają zakres, obejmujący pierwszy element wskazywany przez pierwszy iterator, ale *nieobejmując* ostatniego elementu, wskazywanego przez drugi z iteratorów. Jeżeli zatem chcemy przejść przez zakres elementów, wyznaczony przez te dwa iteratory, powinniśmy zrobić to w następujący sposób:

```
while(start != end)
    cout << start++ << endl;
```

gdzie **start** i **end** są dwoma **iteratorami**, wyznaczającymi zakres. Zwróć uwagę, że iterator **end**, nazywany często *wartownikiem* albo *znacznikiem końca* (ang. *end sentinel*), nie jest wyłuskiwany; informuje nasjedynie, że znajdujemy się już na końcu sekwencji. Tak więc reprezentuje on „element następny po ostatnim”.

Zazwyczaj zamierzamy przechodzić przez wszystkie elementy znajdujące się w kontenerze, dlatego też musi istnieć jakiś sposób umożliwiający kontenerowi utworzenie iteratorów, oznaczających początek sekwencji oraz znacznik końca. W tym przypadku, podobnie jak w standardowej bibliotece języka C++, iteratory te są zwracane przez funkcje składowe kontenera, noszące nazwy **begin()** oraz **end()**. Funkcja **begin()** wykorzystuje pierwszy konstruktor klasy **iterator**, który domyślnie wskazuje początek kontenera (*jest to pierwszy element umieszczony na stosie*). Jednakże niezbędny jest również drugi konstruktor, wykorzystywany przez funkcję **end()**, który umożliwia utworzenie iteratora, będącego znacznikiem końca. Znajdowanie się „na końcu” oznacza w tym przypadku wskazywanie szczytu **stosu**, ponieważ składowa **top** wyznacza zawsze następną dostępną (ale nieużywaną) pozycję na stosie. Konstruktor ten pobiera drugi argument typu **bool**, będący jedynie fikcyjnym argumentem, umożliwiającym rozróżnienie obu konstruktorów.

Do wypełnienia kontenera **IntStack** w funkcji **main()** użyto ponownie generatora liczb ciągu Fibonacciego, natomiast do poruszania się w obrębie całego kontenera (a także węższego zakresu znajdujących się w nim elementów) wykorzystano **iteratory**.

Następnym krokiem jest, oczywiście, uczynienie kodu bardziej ogólnym. Używamy do tego celu szablonów do **sparametryzowania** typu elementów, przechowywanych w kontenerze. Dzięki temu nie musimy przechowywać w nim liczb całkowitych, a zatem możemy umieszczać w nim elementy dowolnego typu:

```
//: C16:IterStackTemplate.h
// Prosty szablon stosu z zagnieżdzonym iteratorem
#ifndef ITERSTACKTEMPLATE_H
#define ITERSTACKTEMPLATE_H
#include "../require.h"
#include <iostream>

template<class T, int ssize - 100>
class StackTemplate {
    T stack[ssize];
    int top;
public:
```

```
StackTemplate() : top(0) {}
void push(const T& i) {
    require(top < ssize, "Zbyt wiele wywolan funkcji push()");
    stack[top++] = i;
}
T pop() {
    require(top > 0, "Zbyt wiele wywolan funkcji pop()");
    return stack[--top];
}
class iterator; // Wymagana deklaracja
friend class iterator; // Uczyn klasę przyjacielem
class iterator { // A teraz ją zdefiniuj
    StackTemplate& s;
    int index;
public:
    iterator(StackTemplate& st): s(st), index(0){}
    // Do utworzenia iteratora-wartownika
    // będącego "znacznikiem końca":
    iterator(StackTemplate& st, bool)
        : s(st). index(s.top) {}
    T operator*() const { return s.stack[index]; }
    T operator++() { // Postać przedrostkowa
        require(index < s.top,
            "iterator przesuniety poza zakres");
        return s.stack[++index];
    }
    T operator++(int) { // Postać przyrostkowa
        require(index < s.top,
            "iterator przesuniety poza zakres");
        return s.stack[index++];
    }
    // Przeskoczenie iteratorem do przodu:
    iterator& operator+=(int amount) {
        require(index + amount < s.top,
            "StackTemplate::iterator::operator+=() "
            "proba przekroczenia zakresu");
        index += amount;
        return *this;
    }
    // Aby sprawdzić, czy znajdujemy się na końcu:
    bool operator==(const iterator& rv) const {
        return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
        return index != rv.index;
    }
    friend std::ostream& operator<<(std::ostream& os, const iterator& it) {
        return os << *it;
    }
};
iterator begin() { return iterator(*this); }
// Tworzenie "znacznika końca":
iterator end() { return iterator(*this, true); }
};

#endif // ITERSTACKTEMPLATE_H ///:-
```

Przejście ze zwykłej klasy do szablonu odbywa się więc w dość przejrzysty sposób. Postępowanie polegające na utworzeniu i uruchomieniu najpierw zwykłej klasy, a następnie na przekształceniu jej w szablon jest na ogół uważane za łatwiejsze niż tworzenie szablonu od podstaw.

Zwróć uwagę na to, że zamiast napisać po prostu:

```
friend iterator; // Uczyń klasę przyjacielem
```

w programie użyto postaci:

```
friend class iterator; // Uczyń klasę przyjacielem
```

Jest to istotne, ponieważ nazwa „iterator” znajduje się już w zasięgu wyznaczonym przez dołączony plik.

Zamiast funkcji składowej **current()** klasa **iterator** posiada **operator***, umożliwiający wybór bieżącego elementu. Dzięki temu **iterator** przypomina raczej wskaźnik — to często stosowana praktyka.

Poniżej zamieszczono poprawiony program przykładowy, umożliwiający przetestowanie szablonu:

```
/// C16:IterStackTemplateTest.cpp
//{L} fibonacci
#include "fibonacci.h"
#include "IterStackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    StackTemplate<int> is;
    for(int i = 0; i < 20; i++)
        is.push(fibonacci(i));
    // Przejście za pomocą iteratora:
    cout << "Przejście przez cały kontener StackTemplate\n";
    StackTemplate<int>::iterator it = is.begin();
    while(it != is.end())
        cout << it++ << endl;
    cout << "Przejście przez część kontenera\n";
    StackTemplate<int>::iterator
        start = is.begin(), end = is.begin();
    start += 5, end += 15;
    cout << "początek = " << start << endl;
    cout << "koniec = " << end << endl;
    while(start != end)
        cout << start++ << endl;
    ifstream in("IterStackTemplateTest.cpp");
    assure(in, "IterStackTemplateTest.cpp");
    string line;
    StackTemplate<string> strings;
    while(getline(in, line))
        strings.push(line);
    StackTemplate<string>::iterator
```

```

    sb - strings.begin(), se = strings.end();
    while(sb != se)
        cout << sb++ << endl;
} //:-
```

Pierwsze użycie iteratora polega jedynie na przejściu od początku do końca kontenera (co pokazuje, że znacznik końca działa prawidłowo). W przypadku jego drugiego użycia widzimy, jak łatwo można, dzięki wykorzystaniu **iteratorów**, określić zakres elementów kontenera (kontenery i iteratory, zawarte w standardowej bibliotece języka C++, niemal wszędzie wykorzystują rozumiane w ten sposób pojęcie zakresu). Przeiążony **operator+** przenosi iteratory **start** i **end** do pozycji znajdujących się wewnątrz zakresu elementów, umieszczonych w kontenerze **is**, a następnie elementy te są drukowane. Zwróć uwagę na to, że element wskazywany przez znacznik końca *nie należy* już do zakresu, dlatego też, by poinformować nas o przekroczeniu zakresu, może on wskazywać element znajdujący się tuż za jego końcem. Nie należy jednak wyłuskiwać znacznika końca, bo skończy się to wyłuskiwaniem zerowego wskaźnika (w klasie **StackTemplate::iterator** umieściłem zapobiegające temu zabezpieczenie, ale w kontenerach i iteratorach, znajdujących się standardowej bibliotece C++ — ze względu na efektywność — nie ma takiego kodu, więc trzeba zwrócić na to uwagę).

Aby wreszcie zweryfikować, że kontener **StackTemplate** działa również z obiektami klas, jest konkretyzowany jego egzemplarz dla klasy **string**. Następnie jest on wypełniany wierszami pochodząymi z pliku źródłowego, które są na końcu drukowane.

asa Stack z iteratorami

Zaprezentowany proces możemy powtórzyć w przypadku dynamicznie powiększanej klasy **Stack**, która była przedstawiona jako przykład w całej książce. Poniżej zaprezentowano klasę **Stack**, zawierającą dołączony do niej, zagnieżdżony iterator:

```

//: C16:TStack2.h
// Szablon klasy Stack, zawierający zagnieżdżony iterator
#ifndef TSTACK2_H
#define TSTACK2_H

template<class T> class Stack {
    struct Link {
        T* data;
        Link* next;
        Link(T* dat, Link* nxt)
            : data(dat), next(nxt) {}
    }* head;
public:
    Stack() : head(0) {}
    ~Stack();
    void push(T* dat) {
        head = new Link(dat, head);
    }
    T* peek() const {
        return head ? head->data : 0;
    }
    T* pop();
    // Zagnieżdżona klasa iteratora:
    class iterator; // Wymagana deklaracja
```

```
friend class iterator; // Uczyń klasę przyjacielem
class iterator { // A teraz ją zdefiniuj
    Stack::Link* p;
public:
    iterator(const Stack<T>& tl) : p(tl.head) {}
    // Konstruktor kopiący:
    iterator(const iterator& tl) : p(tl.p) {}
    // Iterator będący znacznikiem końca:
    iterator() : p(0) {}
    // operator++ zwraca wartość logiczną sygnalizującą koniec:
    bool operator++() {
        if(p->next)
            p = p->next;
        else p = 0; // Koniec listy
        return bool(p);
    }
    bool operator++(int) { return operator++(); }
    T* current() const {
        if(!p) return 0;
        return p->data;
    }
    // Operator wyluskania wskaźnika:
    T* operator->() const {
        require(p != 0,
                "PStack::iterator::operator-> zwrocil 0");
        return current();
    }
    T* operator*() const { return current(); }
    // Konwersja do typu bool, do testów warunków:
    operator bool() const { return bool(p); }
    // Porównanie, umożliwiające wykrycie końca:
    bool operator==(const iterator&) const {
        return p == 0;
    }
    bool operator!=(const iterator&) const {
        return p != 0;
    }
};
iterator begin() const {
    return iterator(*this);
}
iterator end() const { return iterator(); }
};

template<class T> Stack<T>::~Stack() {
    while(head)
        delete pop();
}

template<class T> T* Stack<T>::pop() {
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}
#endif // TSTACK2_H ///:-
```

Jak można również zauważyc, klasa została zmodyfikowana w taki sposób, by obsługiwać prawo własności. Działa ono obecnie poprawnie, ponieważ klasa zna dokładnie typ **obiektów**, przechowywanych w kontenerze (albo przynajmniej ich typ podstawowy, w przypadku którego wszystko będzie działać pod warunkiem wykorzystania wirtualnych destruktorów). Kontener domyślnie niszczy pozostałe w nim obiekty, ale jesteśmy odpowiedzialni za zniszczenie wszystkich wskaźników, które pobraliśmy z kontenera.

Iterotor jest prosty i fizycznie zajmuje bardzo mało miejsca — ma wielkość pojedynczego wskaźnika. Gdy tworzony jest obiekt klasy **iterator**, jest inicjalizowany głową listy powiązanej i można go jedynie przesuwać do przodu, wzdłuż listy. Jeżeli zamierzamy zacząć od nowa, od początku listy, tworzymy nowy iterotor; jeżeli natomiast chcemy zapamiętać jakąś pozycję listy, na podstawie istniejącego **iteratorka**, wskazującego tę pozycję, tworzymy nowy iterotor (używając do tego celu konstruktora kopiącego iteratorka).

Do wywołania funkcji obiektu, wskazywanego przez iterotor, możemy wykorzystać funkcję **current()**, **operator*** lub operator wyłuskania wskaźnika \rightarrow (często występujący wśród iterotorów). Implementacja ostatniego z wymienionych operatorów wygląda jak funkcja **current()**, ponieważ zwraca wskaźnik do bieżącego obiektu. Jest jednak zupełnie od niej inna, gdyż operator wyłuskania wskaźnika realizuje dodatkowe poziomy wyłuskania (zob. — rozdział 12.).

Klasa **iterator** odpowiada wzorcowi, znanemu z poprzedniego przykładu. Została ona osadzona wewnątrz klasy kontenera; zawiera **konstruktory**, tworzące zarówno iterotor wskazujący element w kontenerze, jak i iterotor stanowiący „**znacznik końcowy**”. Z kolei klasa kontenera posiada funkcje **begin()** oraz **end()**, umożliwiające utworzenie tych iterotorów. (Gdy dowiesz się więcej na temat standardowej biblioteki języka C++, zobaczysz, że używane tutaj nazwy **iterator**, **begin()** oraz **end()** wywodzą się w oczywisty sposób ze standardowych klas kontenerowych. Pod koniec rozdziału przekonasz się, że dzięki nim zamieszczone tutaj klasy kontenerów mogą być używane w taki sposób, jakby były klasami standardowej biblioteki kontenerów języka C++).

Cała implementacja kontenera została zawarta w pliku nagłówkowym; nie ma więc oddzielnego pliku **cpp**. Poniżej przedstawiono niewielki test, sprawdzający działanie iteratorka:

```
//: C16:TStack2Test.cpp
#include "TStack2.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    ifstream file("TStack2Test.cpp");
    assure(file. "TStack2Test.cpp");
    Stack<string> textlines;
    // Odczytywanie pliku i zapamiętywanie wierszy na stosie:
    string line;
    while(getline(file, line))
```

```

textlines.push(new string(11ne));
int i = 0;
// Użycie iteratora do wydrukowania wierszy
// znajdujących się na liście:
Stack<string>::iterator it = textlines.begin();
Stack<string>::iterator* it2 = 0;
while(it != textlines.end()) {
    cout << it->c_str() << endl;
    it++;
    if(++i == 10) // Zapamiętanie 10. wiersza
        it2 = new Stack<string>::iterator(it);
}
cout << (*it2)->c_str() << endl;
delete it2;
} //:-

```

Klasa **Stack** jest konkretyzowana w celu przechowywania obiektów klasy **string**, a następnie wypełniana wierszami odczytanymi z pliku. Następnie tworzony jest iterator, wykorzystywany do poruszania się w obrębie utworzonego ciągu elementów. Dziesiąty wiersz pliku jest zapamiętywany dzięki wygenerowaniu za pomocą konstruktora kopiącego drugiego iteratora, utworzonego na podstawie pierwszego. Później ten wiersz jest drukowany, a utworzony dynamicznie iterator — niszczony. W tym przypadku do kontroli czasu życia obiektu wykorzystano dynamiczne tworzenie obiektów.

Klasa **PStash** z iteratorami

Posiadanie iteratorów ma sens w przypadku większości klas kontenerów. Poniżej pokazano iterator dodany do klasy **PStash**:

```

//: C16:TPStash2.h
// Szablon klasy PStash, zawierający zagnieżdzony iterator
#ifndef TPSTASH2_H
#define TPSTASH2_H
#include "../require.h"
#include <cstdlib>

template<class T, int incr = 20>
class PStash {
    int quantity;
    int next;
    T** storage;
    void inflate(int increase - incr);
public:
    PStash() : quantity(0), storage(0), next(0) {}
    ~PStash();
    int add(T* element);
    T* operator[](int index) const;
    T* remove(int index);
    int count() const { return next; }
    // Zagnieżdzona klasa iteratora:
    class iterator; // Wymagana deklaracja
    friend class iterator; // Uczyń klasę przyjacielem
    class iterator { // A teraz ją zdefiniuj

```

```
16  
PStash& ps;  
int index;  
public:  
    iterator(PStash& pStash)  
        : ps(pStash), index(0) {}  
    // Do utworzenia iteratora-wartownika  
    // będącego "znacznikiem końca":  
    iterator(PStash& pStash, bool)  
        : ps(pStash).index(ps.next) {}  
    // Konstruktor kopiący:  
    iterator(const iterator& rv)  
        : ps(rv.ps), index(rv.index) {}  
    iterator& operator=(const iterator& rv) {  
        ps = rv.ps;  
        index = rv.index;  
        return *this;  
    }  
    iterator& operator++() {  
        require(++index < ps.next,  
            "PStash::iterator::operator++ "  
            "przesunął indeks poza zakres");  
        return *this;  
    }  
    iterator& operator++(int) {  
        return operator++();  
    }  
    iterator& operator--() {  
        require(--index >= 0,  
            "PStash::iterator::operator-- "  
            "przesunął indeks poza zakres");  
        return *this;  
    }  
    iterator& operator--(int) {  
        return operator--();  
    }  
    // Przeskoczenie iteratorem do przodu lub tyłu:  
    iterator& operator+=(int amount) {  
        require(index + amount < ps.next &&  
            index + amount >= 0,  
            "PStash::iterator::operator+= "  
            "próba uzyskania indeksu poza zakresem");  
        index += amount;  
        return *this;  
    }  
    iterator& operator-=(int amount) {  
        require(index - amount < ps.next &&  
            index - amount >= 0,  
            "PStash::iterator::operator-= "  
            "próba uzyskania indeksu poza zakresem");  
        index -= amount;  
        return *this;  
    }  
    // Utworzenie nowego iteratora. przesuniętego do przodu  
    iterator operator+(int amount) const {  
        iterator ret(*this);  
        ret += amount; // operator+= sprawdza zakres  
        return ret;  
    }
```

```
T* current() const {
    return ps.storage[index];
}
T* operator*() const { return current(); }
T* operator->() const {
    require(ps.storage[index] != 0,
        "PStash::iterator::operator-> zwrocil 0");
    return current();
}
// Usunięcie bieżącego elementu:
T* remove(){
    return ps.remove(index);
}
// Testy porównania do wykrycia końca:
bool operator==(const iterator& rv) const {
    return index == rv.index;
}
bool operator!=(const iterator& rv) const {
    return index != rv.index;
}
iterator begin() { return iterator(*this); }
iterator end() { return iterator(*this, true); }
};

// Usuwanie obiektów zawartych w kontenerze:
template<class T, int incr>
PStash<T, incr>::~PStash() {
    for(int i = 0; i < next; i++) {
        delete storage[i]; // Zeroowe wskaźniki nie są problemem
        storage[i] = 0; // Dla pewności
    }
    delete []storage;
}

template<class T, int incr>
int PStash<T, incr>::add(T*element) {
    if(next >= quantity)
        inflate();
    storage[next++] = element;
    return(next - 1); // Numer indeksu
}

template<class T, int incr> inline
T* PStash<T, incr>::operator[](int index) const {
    require(index >= 0,
        "PStash::operator[] indeks ma wartosc ujemna");
    if(index >= next)
        return 0; // Oznaczenie końca
    require(storage[index] != 0,
        "PStash::operator[] zwrocony pusty wskaznik");
    return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
    // operator[] sprawdza zakres:
```

```

T* v = operator[](index);
// "Usunięcie" wskaźnika:
storage[index] = 0;
return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
    const int tsz = sizeof(T*);
    T** st = new T*[quantity + increase];
    memset(st, 0, (quantity + increase) * tsz);
    memcpy(st, storage, quantity * tsz);
    quantity += increase;
    delete []storage; // Stary obszar pamięci
    storage = st; // Wskaźnik do nowego obszaru pamięci
}
#endif // TPSTASH2_H //:-~

```

Większą część programu stanowi dość oczywiste przeniesienie poprzedniej wersji klasy **PStash** oraz zagnieźdzonego iteratora do postaci szablonu. Jednakże tym razem operatory zwracają referencje do bieżącego **iteratora**, co jest sposobem bardziej typowym i elastycznym.

Destruktor wywołuje operator **delete** dla wszystkich wskaźników przechowywanych w kontenerze, a ponieważ ich typ jest określony za pomocą szablonu, destrukcja jest dokonywana w prawidłowy sposób. Należy pamiętać o tym, że jeżeli kontener przechowuje wskaźniki do obiektów klasy podstawowej, klasa ta powinna posiadać wirtualny destruktör, zapewniający poprawne usuwanie obiektów klas pochodnych, których adresy są rzutowane w góre podczas umieszczania ich w kontenerze.

Klasa **PStash::iterator** odpowiada modelowi operatora, związanego przez cały czas życia z jednym obiektem kontenera. Ponadto konstruktor kopiący pozwala na powstanie nowego iteratora. Wskazuje on ten sam element, co istniejący iterator, na podstawie którego jest tworzony. W efekcie w kontenerze zostaje utworzona zakładka. Funkcje składowe **operator+=** oraz **operator-=** pozwalają na przeniesienie iteratora o określoną liczbę pozycji, zwracając uwagę na granice kontenera. Przeciążone operatory **inkrementacji** i **dekrementacji** przesuwają iterator o jedną pozycję. Funkcja składowa **operator+** zwraca nowy iterator, przesunięty do przodu o wielkość składnika. Podobnie jak w poprzednim przykładzie, do wykonywania operacji na elemencie wskazywanym przez iterator używane są operatory wyłuskania wskaźnika, natomiast funkcja **remove()** niszczy bieżący obiekt, wywołując funkcję **remove()** kontenera.

Kod tego samego rodzaju, co poprzednio (w rodzaju kontenerów standardowej biblioteki C++) jest używany do utworzenia znacznika końca — funkcja składowa **end()** kontenera oraz operatory **==** i **!=**, służące do porównań.

W zamieszczonym poniżej przykładzie są tworzone i testowane dwie różne wersje obiektów klasy **Stash** — jedna dla obiektów klasy o nazwie **Int**, informująca o wywołaniu swoich konstruktorów i destruktatorów, a druga — przechowująca obiekty klasy **string**, zawartej w bibliotece standardowej:

```
///: C16.TPStash2Test.cpp
#include "TPStash2.h"
#include "../require.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Int {
    int i;
public:
    Int(int ii = 0) : i(ii) {
        cout << ">" << i << ' ';
    }
    ~Int() { cout << "~" << i << ' ';}
    operator int() const { return i; }
    friend ostream&
        operator<<(ostream& os, const Int& x) {
            return os << "Int: " << x.i;
        }
    friend ostream&
        operator<<(ostream& os, const Int* x) {
            return os << "Int: " << x->i;
        }
};

int main() {
    // Wymuszenie wywołania destruktorów
    PStash<Int> ints;
    for(int i = 0; i < 30; i++)
        ints.add(new Int(i));
    cout << endl;
    PStash<Int>::iterator it = ints.begin();
    it += 5;
    PStash<Int>::iterator it2 = it + 10;
    for(; it != it2; it++)
        delete it.remove(); // Domyślne usuwanie
    cout << endl;
    for(it = ints.begin(); it != ints.end(); it++)
        if(*it) // Funkcja remove() powoduje powstawanie "dziur"
            cout << *it << endl;
    } // W tym miejscu wywoływany jest destruktor obiektu "ints"
    cout << "\n.....\n";
ifstream in("TPStash2Test.cpp");
assure(in, "TPStash2Test.cpp");
// Konkretyzacja dla klasy string:
PStash<string> strings;
string line;
while(getline(in, line))
    strings.add(new string(line));
PStash<string>::iterator sit = strings.begin();
for(; sit != strings.end(); sit++)
    cout << **sit << endl;
sit = strings.begin();
int n = 26;
sit += n;
for(; sit != strings.end(); sit++)
    cout << n++ << ":" << **sit << endl;
} //:-
```

Dla ułatwienia klasa **Int** posiada związaną ze sobą **operator<<** klasy **ostream** — zarówno dla argumentów typu **Int&**, jak i **Int***.

Pierwszy blok kodu, znajdujący się w funkcji **main()**, jest zawarty w nawiasie klamrowym, by wymusić destrukcję obiektu klasy **PStash<Int>** i zarazem automatyczne usuwanie obiektów przez destruktory tej klasy. Pewien zakres elementów jest pobierany z kontenera i usuwany ręcznie, by pokazać, że kontener **PStash** sam usunie pozostałe elementy.

Dla obu konkretyzacji klasy **PStash** są tworzone iteratory, wykorzystywane do poruszania się w obrębie kontenera. Zwróć uwagę na elegancję uzyskaną dzięki użyciu tych konstrukcji — nie jesteśmy przytłaczani szczegółami implementacyjnymi, związanymi z używaniem tablicy. Informujemy jedynie kontener i iterator *co* mają robić, a nie w jaki sposób. Dzięki temu uzyskane rozwiązanie jest łatwiejsze do zrozumienia, zbudowania i modyfikacji.

Dlaczego iteratory?

Powyżej zostały przedstawione przykłady iteratorów, ale zrozumienie, dlaczego są one takie ważne, wymaga bardziej złożonego przykładu.

W prawdziwym programie obiektowym często spotyka się, wykorzystywane razem — **polimorfizm**, dynamiczne tworzenie obiektów oraz kontenery. Kontenery oraz dynamiczne tworzenie obiektów rozwiązują problem, polegający na nieznajomości liczby i typów niezbędnych obiektów. Ponadto jeżeli kontener jest skonfigurowany w taki sposób, że przechowuje wskaźniki do obiektów klasy podstawowej, to za każdym razem, gdy w kontenerze umieszczany jest wskaźnik do obiektu klasy pochodnej, odbywa się rzutowanie w górę (z towarzyszącą mu organizacją kodu i korzyściami, wynikającymi z rozszerzalności). Zamieszczony poniżej przykład, ostatni program w pierwszym tomie książki, łączy ze sobą rozmaite aspekty wszystkiego, co poznaliśmy do tej pory. Jeżeli rozumiesz działanie tego programu, jesteś gotowy, by przystąpić do lektury drugiego tomu książki.

Założymy, że tworzymy program, pozwalający użytkownikowi na edycję i tworzenie różnego rodzaju rysunków. Każdy rysunek jest obiektem, zawierającym kolekcję obiektów klasy **Shape**:

```
//: C16:Shape.h
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>
#include <string>

class Shape {
public:
    virtual void draw() = 0;
    virtual void erase() = 0;
    virtual ~Shape() {}
};
```

```
class Circle : public Shape {
public:
    Circle() {}
    ~Circle() { std::cout << "Circle::~Circle\n"; }
    void draw() { std::cout << "Circle::draw\n"; }
    void erase() { std::cout << "Circle::erase\n"; }
};

class Square : public Shape {
public:
    Square() {}
    ~Square() { std::cout << "Square::~Square\n"; }
    void draw() { std::cout << "Square::draw\n"; }
    void erase() { std::cout << "Square::erase\n"; }
};

class Line : public Shape {
public:
    Line() {}
    ~Line() { std::cout << "Line::~Line\n"; }
    void draw() { std::cout << "Line::draw\n"; }
    void erase() { std::cout << "Line::erase\n"; }
};
#endif // SHAPE_H //:-
```

W powyższym pliku nagłówkowym wykorzystano klasyczną strukturę funkcji wirtualnych, znajdujących się w klasie podstawowej i zasłoniętych w klasie pochodnej. Zwróć uwagę na to, że klasa **Shape** zawiera wirtualny destruktor. Powinien on być dodawany automatycznie do każdej klasy, zawierającej funkcje wirtualne. Jeżeli kontener przechowuje wskaźniki lub referencje do obiektów klasy **Shape**, to w przypadku gdy dla tych obiektów wywoływane są wirtualne destruktory — wszystko zostanie usunięte poprawnie.

Każdy z rodzajów rysunków, znajdujących się w poniższym programie, wykorzystuje inny rodzaj szablonu klasy kontenera — używane są klasy **PStash** oraz **Stack**, zdefiniowane w bieżącym rozdziale, oraz klasa **vector**, pochodząca ze standardowej biblioteki języka C++. „Wykorzystywanie” kontenerów jest tutaj skrajnie proste i dziedziczenie może okazać się wcale nie najlepszym podejściem (bardziej odpowiednia wydaje się kompozycja). Jednakże w tym przypadku dziedziczenie jest prostym rozwiązaniem i nie wpływa ujemnie na istotę przykładu.

```
//: C16:Drawing.cpp
#include <vector> // Używane są również standardowe wektory!
#include "TPStash2.h"
#include "TStack2.h"
#include "Shape.h"
using namespace std;

// Klasa Drawing jest podstawowym kontenerem klasy Shape:
class Drawing : public PStash<Shape> {
public:
    ~Drawing() { cout << "~Drawing" << endl; }
};
```

```

// Klasa Plan jest innym kontenerem klasy Shape:
class Plan : public Stack<Shape> {
public:
    -Plan() { cout << "~Plan" << endl; }
};

// Klasa Schematic jest innym kontenerem klasy Shape:
class Schematic : public vector<Shape*> {
public:
    -Schematic() { cout << "~Schematic" << endl; }
};

// Szablon funkcji:
template<class Iter>
void drawAll(Iter start, Iter end) {
    while(start != end) {
        (*start)->draw();
        start++;
    }
}

int main() {
    // Każdy rodzaj kontenera
    // posiada inny interfejs:
    Drawing d;
    d.add(new Circle);
    d.add(new Square);
    d.add(new Line);
    Plan p;
    p.push(new Line);
    p.push(new Square);
    p.push(new Circle);
    Schematic s;
    s.push_back(new Square);
    s.push_back(new Circle);
    s.push_back(new Line);
    Shape* sarray[] = {
        new Circle, new Square, new Line
    };
    // Iteratory oraz szablon funkcji pozwalają
    // na traktowanie ich w sposób ogólny:
    cout << "Drawing d:" << endl;
    drawAll(d.begin(), d.end());
    cout << "Plan p:" << endl;
    drawAll(p.begin(), p.end());
    cout << "Schematic s:" << endl;
    drawAll(s.begin(), s.end());
    cout << "Array sarray:" << endl;
    // Działa to nawet w przypadku tablic wskaźników:
    drawAll(sarray,
            sarray + sizeof(sarray)/sizeof(*sarray));
    cout << "Koniec funkcji main" << endl;
} //:-
}

```

Wszystkie różniące się od siebie typy kontenerów przechowują wskaźniki do obiektów klasy **Shape**, a także, podlegające rzutowaniu w góre, wskaźniki do obiektów klas pochodnych klasy **Shape**. Jednakże z uwagi na **polimorfizm** wszystko działa poprawnie podczas wywoływania wirtualnych funkcji.

Zwróć uwagę na to, że tablica wskaźników do obiektów typu **Shape** — **sarray** — może być również uważana za kontener.

Szablony funkcji

W funkcji **drawAll()** pojawia się coś nowego. W niniejszym rozdziale używaliśmy wyłącznie szablonów *klas*, które powodowały konkretyzację nowych klas na podstawie jednego lub większej liczby parametrów określających typy. Jednak również łatwo można tworzyć **szablony funkcji**, generujące nowe funkcje na podstawie parametrów określających typy. Powód, dla którego tworzone są szablony funkcji, jest taki sam jak w przypadku szablonów klas — próbujemy utworzyć ogólny kod i robimy to, opóźniając specyfikację jednego lub większej liczby typów. Określamy **jedynie**, że typy te są parametrami, obsługującymi pewne operacje, nie precyzując dokładnie, o jakie typy chodzi.

Szablon funkcji **drawAll()** można traktować jako *algorytm* (i tak właśnie nazywana jest większość szablonów funkcji, zawartych w standardowej bibliotece języka C++). Określa on po prostu, w jaki sposób zrobić coś w stosunku do danych iteratorów, opisujących zakres elementów, pod warunkiem, że iteratory te mogą być wyłuskiwane, inkrementowane i porównywane. Są one rodzajami iteratorów, które tworzyliśmy w tym rozdziale, a także — nieprzypadkowo — rodzajem iteratorów tworzonym przez kontenery, zawarte w standardowej bibliotece języka C++, czego dowodem jest wykorzystanie w powyższym przykładzie klasy *vector*.

Chcielibyśmy również, aby szablon **drawAll()** był *algorytmem ogólnym* (ang. *generic algorithm*) w taki sposób, by kontenery mogły być dowolnych typów i abyśmy nie musieli pisać nowej wersji algorytmu dla każdego nowego typu kontenera. Jest to sytuacja, w której szablony funkcji są niezbędne, ponieważ generujące **automatycznie** określony kod dla każdego kolejnego typu kontenera. Lecz bez dodatkowego pośrednictwa, zapewnionego przez iteratory, uzyskanie takiej ogólności nie byłoby możliwe. To dlatego właśnie iteratory są takie **ważne** — pozwalają na napisanie kodu ogólnego przeznaczenia, wykorzystującego kontenery, nieznającego wewnętrznej struktury żadnego kontenera (zwróć uwagę na to, że w języku C++ iteratory oraz algorytmy ogólne wymagają do działania szablonów funkcji).

Dowód na to można dostrzec w funkcji **main()**, ponieważ szablon **drawAll()** działa bez żadnych modyfikacji z wszystkimi, różnymi od siebie rodzajami kontenerów. Co ciekawsze, szablon ten funkcjonuje również ze wskaźnikami do początku i końca tablicy *sarray*. Możliwość traktowania tablic jako kontenerów stanowi integralny element projektu standardowej biblioteki języka C++, której algorytmy bardzo przypominają szablon funkcji **drawAll()**.

Z uwagi na to, że szablony klas kontenerów są rzadko przedmiotem dziedziczenia i rzutowania w góre, widocznych w przypadku „**zwyczajnych**” klas, w klasach kontenerów niemal nigdy nie spotyka się funkcji wirtualnych. Klasę kontenerów umożliwiają ponowne wykorzystywani kodu za pomocą szablonów, a nie dziedziczenia.

'odsumowanie

Klasy kontenerów stanowią zasadniczy element programowania obiektowego. Są one innym sposobem uproszczenia oraz ukrycia szczegółów programu, a także przyspieszenia procesu jego powstawania. Ponadto zapewniają one duży stopień bezpieczeństwa i elastyczności, zastępując proste tablice oraz stosunkowo łatwe techniki strukturyzacji danych, dostępne w języku C.

Z uwagi na to, że klienci-programiści potrzebują kontenerów, powinny być one proste w użyciu. W tym właśnie względzie przychodzą z pomocą szablony. Dzięki nim składnia wielokrotnego wykorzystania kodu źródłowego (w przeciwieństwie do wielokrotnego wykorzystania kodu obiektów, dostępnego dzięki dziedziczeniu i kompozycji) staje się oczywista nawet dla poczynającego użytkownika. W istocie wielokrotne wykorzystywanie kodu, możliwe dzięki zastosowaniu szablonów, jest wyraźnie łatwiejsze niż dziedziczenie i kompozycja.

Mimo że dzięki tej książce wiesz, w jaki sposób tworzyć klasy kontenerów i iteratorów, w praktyce znacznie bardziej przydatne jest poznanie kontenerów i iteratorów zawartych w standardowej bibliotece języka C++, ponieważ można się spodziewać, że są one dostępne dla każdego kompilatora. Dzięki lekturze drugiego tomu książki (który można pobrać z witryny <http://helion.pl/online/thinking/index.html>) zauważysz, że kontenery i algorytmy zawarte w standardowej bibliotece języka C++ zaspokoją wszystkie twoje potrzeby, dzięki czemu nie będziesz musiał samodzielnie tworzyć nowych.

W rozdziale zostały zasignalizowane kwestie związane z klasami kontenerów, ale, jak się można tego spodziewać, mogą one być znacznie bardziej skomplikowane. Złożona biblioteka klas kontenerowych może rozwiązywać wszystkie rodzaje dodatkowych problemów, związanych z kontenerami, m.in. wielowątkowość, trwałość (ang. *persistence*) oraz zbieranie nieużytków (odśmiecanie).

Ćwiczenia

Rozwiązania wybranych ćwiczeń znajdują się w dokumencie elektronicznym *The Thinking in C++ Annotated Solution Guide*, który można pobrać za niewielką opłatą z witryny www.BruceEckel.com.

1. Zaimplementuj hierarchię dziedziczenia, widoczną znajdującym się w rozdziale diagramie klasy **OKształt**.
2. Zmień rozwiązanie ćwiczenia 1., zamieszczonego w rozdziale 15., by zamiast tablicy wskaźników do obiektów klasy **Shape** wykorzystać klasę **Stack** oraz iterator, zawarte w pliku nagłówkowym **TStack2.h**. Do hierarchii klas dodaj destruktory, dzięki którym będzie można zobaczyć, że obiekty klasy **Shape** są niszczone w momencie wyjścia kontenera **Stack** poza zasięg.

3. Zmodyfikuj plik nagłówkowy **TPStash.h** w taki sposób, by wielkość, o której powiększany jest przydzielony obszar pamięci, **używana** przez funkcję **inflate()**, mogła być zmieniana w trakcie życia danego obiektu kontenera.
4. Zmodyfikuj plik nagłówkowy **TPStash.h** w taki sposób, by wielkość, o której powiększany jest przydzielony obszar pamięci, używana przez funkcję **inflate()**, automatycznie zmieniała się w celu ograniczenia liczby wywołań tej funkcji. Na przykład za każdym razem, gdy wywoływana jest funkcja **inflate()**, można by podwajać wielkość, używaną przy kolejnym wywołaniu. Zademonstruj działanie zmodyfikowanej w taki sposób klasy, wyświetlając komunikat podczas każdego wywołania funkcji **inflate()** i pisząc program testowy, zawarty w funkcji **main()**.
5. Przekształć funkcję **fibonacci()** w szablon funkcji o parametrze określającym typ zwracanej przez funkcję wartości (zamiast wartości typu **int** będzie ona mogła zwracać wartości typów **long**, **float** itp.).
6. Wykorzystującą jako **wewnętrzna implementację** klasę **vector**, zawartą w standardowej bibliotece języka C++, utwórz szablon klasy **Set**, umożliwiający umieszczanie w kontenerze tylko jednego egzemplarza każdego unikatowego obiektu. Utwórz zagnieżdżoną klasę iteradora, obsługującą przedstawione w rozdziale pojęcie „znacznika końca”. W funkcji **main()** napisz kod, testujący działanie klasy **Set**, a następnie zastąp ją klasą **set**, zawartą w standardowej bibliotece języka C++, aby sprawdzić, czy działa ona poprawnie.
7. Zmodyfikuj plik nagłówkowy **AutoCounter.h** w taki sposób, aby można było użyć obiektu zawartej w nim klasy jako obiektu składowego dowolnej klasy, której tworzenie i niszczenie zamierza się śledzić. Dodaj składową typu **string**, przechowującą nazwę tej klasy. Przetestuj to narzędzie wewnątrz dowolnej własnej klasy.
8. Utwórz wersję klasy zawartej w pliku nagłówkowym **OwnerStack.h**, która wykorzystuje w charakterze wewnętrznej implementacji klasę **vector** standardowej biblioteki języka C++. Być może, aby to zrobić, konieczne będzie uzyskanie informacji o niektórych funkcjach składowych klasy **vector** (albo przynajmniej przejrzenie zawartości pliku nagłówkowego **<vector>**).
9. Zmodyfikuj klasę zawartą w pliku nagłówkowym **ValueStack.h** w taki sposób, by w przypadku umieszczania na stosie funkcja **push()** większej liczby obiektów, które powoduje przekroczenie dostępnego miejsca, automatycznie powiększała przydzieloną pamięć. Zmień program **ValueStackTest.cpp** w taki sposób, by przetestować tę nową cechę klasy.
10. Powtórz poprzednie ćwiczenie, ale jako wewnętrznej implementacji klasy **ValueStack** użyj klasy **vector**, zawartej w standardowej bibliotece języka C++. Zwróć uwagę na to, o ile prostsze jest takie rozwiązanie.
11. Zmodyfikuj program **ValueStackTest.cpp** w taki sposób, by w funkcji **main()** zamiast klasy **Stack** wykorzystywał klasę **vector**, zawartą w standardowej bibliotece języka C++. Zwróć uwagę na zachowanie się programu po jego uruchomieniu — czy podczas tworzenia obiektu klasy **vector** również tworzona jest grupa domyślnych obiektów?

12. Zmodyfikuj klasę zawartą w pliku nagłówkowym **TStack2.h** w taki sposób, by wykorzystywała w charakterze wewnętrznej implementacji klasę **vector**, zawartą w standardowej bibliotece języka C++. Upewnij się, że nie zmienił się interfejs klasy, dzięki czemu program **TStack2Test.cpp** będzie działał bez modyfikacji.
13. Powtórz poprzednie ćwiczenie, zamiast klasy **vector** używając klasy **stack**, zawartej w standardowej bibliotece języka C++ (może to wymagać uzyskania informacji dotyczących klasy **stack** lub przeszukania pliku nagłówkowego **<stack>**).
14. Zmodyfikuj klasę **zawartą** w pliku nagłówkowym **TStash2.h** w taki sposób, by wykorzystywała w charakterze wewnętrznej implementacji klasę **vector**, zawartą w standardowej bibliotece języka C++. Upewnij się, że nie zmienił się interfejs klasy, dzięki czemu program **TStash2Test.cpp** będzie działał bez modyfikacji.
15. Zmodyfikuj iterator **IntStackIter** zawarty w pliku **IterIntStack.cpp**, tak by dodać do niego konstruktor „znacznika końca” oraz operatory `==` i `!=`. W funkcji **main()** wykorzystaj ten iterator do przejścia przez wszystkie elementy zawarte w kontenerze, aż do napotkania znacznika końca.
16. Wykorzystując pliki nagłówkowe **TStack2.h**, **TPStash2.h** oraz **Shape.h**, utwórz konkretyzacje szablonów kontenerów **Stack** oraz **PStash** dla typu **Shape***, następnie wypełnij każdy z nich różnymi rzutowanymi w góre wskaźnikami do obiektów klasy **Shape**, a potem użyj iteratorów do przejścia przez każdy kontener i wywołania dla każdego obiektu funkcji **draw()**.
17. Przekształć klasę **Int**, zawartą w programie **TPStash2Test.cpp**, w szablon klasy, dzięki czemu będzie ona mogła przechowywać dowolny typ obiektów (w dowolny sposób zmień nazwę tej klasy na coś bardziej odpowiedniego).
18. Przekształć klasę **IntArray**, zawartą w programie **IostreamOperatorOverloading.cpp**, znajdującym się w rozdziale 12., w szablon klasy, parametryzując zarówno typ przechowywanych obiektów, jak i wielkość wewnętrznej tablicy.
19. Przekształć klasę **ObjContainer**, zawartą w programie **NestedSmartPointer.cpp**, znajdującym się w rozdziale 12., w szablon klasy. Przetestuj go za pomocą dwóch różnych klas.
20. Zmodyfikuj programy **C15:OStack.h** oraz **C15:OStackTest.cpp**, przekształcając klasę **Stack** w szablon, dzięki czemu będzie on automatycznie wielokrotnie dziedziczył zarówno po zawartej w nim klasie, jak i po klasie **Object**. Skonkretyzowana klasa Stack powinna pobierać i zwracać wyłącznie wskaźniki zawartego w niej typu.
21. Powtórz poprzednie ćwiczenie, wykorzystując klasę **vector** zamiast klasy **Stack**.
22. Z klasy **vector<void*>** wyprowadź klasę **StringVector**, a następnie przedefiniuj funkcje składowe **push_back()** oraz **operator[]** w taki sposób, by pobierały i zwracały wyłącznie wskaźniki typu **string*** (dokonując odpowiedniego rzutowania). Następnie przygotuj szablon, który będzie

automatycznie tworzył klasę kontenera, wykonującą to samo zadanie w stosunku do wskaźników dowolnego typu. Technika ta jest często stosowana w celu ograniczenia rozrastania się kodu, spowodowanego zbyt wieloma konkretyzacjami szablonów.

23. Dodaj do iteratora **PStash::iterator**, zawartego w pliku nagłówkowym **TPStash2.h**, **operator-**, działający w sposób podobny do funkcji **operator+**, i przetestuj jego działanie.
24. W programie **Drawing.cpp** dodaj szablon funkcji, wywołującej funkcje składowe **erase()**, i przetestuj go.
25. (Trudne) Zmodyfikuj klasę **Stack**, zawartą w pliku nagłówkowym **TStack2.h**, zapewniając pełną „ziarnistość” prawa własności — do każdego wskaźnika dodaj znacznik, określający, czy ten wskaźnik jest właściwicielem wskazywanego przez siebie obiektu, i obsługuj tę informację w funkcji **push()** oraz w destruktorze. Dodaj funkcje składowe, umożliwiające odczytanie i zmianę prawa własności dla każdego wskaźnika.
26. (Trudne) Zmodyfikuj program **PointerToMemberOperator.cpp**, znajdujący się w rozdziale 12., w taki sposób, by przekształcić klasę **FunctionObject** oraz **operator->*** w szablon, działające z dowolnym zwracanym typem (w przypadku **operatora->*** należy użyć szablonów *składowych*, opisanych w drugim tomie książki). Dodaj i przetestuj możliwość podawania zera, jednego i dwóch argumentów w funkcjach składowych klasy **Dog**.

Dodatek A

Styl kodowania

Tematem tego dodatku nie jest sposób robienia wcięć w kodzie i umieszczania nawiasów okrągłych oraz klamrowych, mimo że o tych kwestiach również się w nim wspomina. Dotyczy on ogólnych zasad stosowanych w książce, związanych z organizacją wydruków zawierających kod programów.

Wiele spośród poruszonych tu zagadnień było stopniowo wprowadzanych w poszczególnych rozdziałach książki, ale ponieważ dodatek ten znajduje się na jej końcu, można założyć, że każdy temat stanowi otwartą kwestię, a w przypadku wątpliwości można *zajrzeć* do odpowiedniego podrozdziału.

Wszystkie decyzje dotyczące stylu kodowania przedstawionego w książce zostały podjęte po dokładnym przemyśleniu, co czasami zajęło lata. Oczywiście, każdy ma jakieś powody, dla których formatuje kod w taki a nie inny sposób — zamierzam jedynie opowiedzieć o tym, w jaki sposób wypracowałem własny styl kodowania, a także o ograniczeniach oraz o czynnikach zewnętrznych, które wpłynęły na podjęte przez mnie decyzje.

Ogólne zasady

Zamieszczone w tekście książki identyfikatory (funkcje, zmienne oraz nazwy klas) zostały oznaczone czcionką **pogrubioną**. Taką samą czcionką wyróżniono większość występujących w tekście słów kluczowych.

W celu prezentacji przykładów zawartych w książce używam odrębnego stylu, rozwijanego latami. Jego szczególną inspirację stanowił styl, który **Bjarne Stroustrup** zastosował w swojej pierwszej książce „*The C++ Programming Language*”¹. Temat stylu formatowania może być przedmiotem wielogodzinnej gorącej dyskusji, więc pragnę zaznaczyć, że nie zamierzam, odwołując się do moich przykładów, narzucać

¹ Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley, 1986 (pierwsze wydanie).

komukolwiek poprawnego stylu formatowania — **mam jedynie własne** powody, które sprawiają, że zyskał on właśnie taki kształt. Ponieważ język C++ jest językiem programowania o swobodnej strukturze, możesz używać takiego stylu kodowania, jaki ci najbardziej odpowiada.

Uważam, że stosowanie w obrębie projektu jednolitego stylu formatowania jest istotne. W **Internetie** można znaleźć wiele narzędzi, umożliwiających **przeformatowanie** kodu całego projektu w celu osiągnięcia tej cennej jednolitości.

Programy zawarte w tej książce są plikami, które zostały automatycznie wyodrębnione z tekstu książki, co pozwoliło mi na ich przetestowanie w celu upewnienia się, że działają poprawnie. Tak więc wszystkie wydrukowane w książce programy powinny działać i kompilować się bez błędów, jeżeli używany kompilator jest zgodny ze standardem języka C++ (należy pamiętać, że nie wszystkie kompilatory języka C++ obsługują wszystkie właściwości języka). Zawarte w programach błędy, które *powinny* spowodować zgłoszenie komunikatów o błędach podczas komplikacji, zostały umieszczone w komentarzach, po znakach `//!`, dzięki czemu można łatwo je odnaleźć, a także przetestować za pomocą metod automatycznych. Odkryte i zgłoszone autorowi błędy pojawią się najpierw w elektronicznej wersji książki (w witrynie www.BruceEckel.com), a dopiero później w jącej poprawionych wydaniach.

Zgodnie z jednym ze standardów stosowanych w książce wszystkie zawarte w niej programy są kompilowane i łączone bez błędów (mimo że mogą czasami powodować wyświetlanie ostrzeżeń). Aby to uzyskać, niektóre z programów, zawierające jedynie przykłady kodu i niesamodzielne, kryją puste funkcje **main()** — jak widoczna poniżej:

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

Pozwala to programowi łączącemu na zakończenie pracy bez zgłaszania błędów.

W przypadku funkcji **main()** standardem jest zwracanie przez nią wartości całkowitej, lecz standard języka C++ określa, że jeżeli w funkcji **main()** nie ma instrukcji **return**, to kompilator automatycznie generuje kod, powodujący zwrócenie przez funkcję wartości zerowej. Ta właśnie możliwość (brak instrukcji **return** w funkcji **main()**) jest wykorzystywana w książce (niektóre kompilatory mogą zgłaszać z tego powodu ostrzeżenia, ale nie są one zgodne ze standardem języka C++).

azwy plików

W języku C pliki nagłówkowe (obejmujące deklaracje) tradycyjnie posiadają nazwy o rozszerzeniach **.h**, natomiast pliki zawierające implementację (powodujące przydzielanie pamięci i generację kodu) zawarte są w plikach o nazwach z rozszerzeniem **.c**. Język C++ przeszedł ewolucję. Początkowo był rozwijany w systemie Unix, w którym system operacyjny rozróżnia wielkie i małe litery w nazwach plików. Na początku rozszerzenia nazw plików były zapisanymi wielkimi literami wariantami rozszerzeń, stosowanych w języku C — odpowiednio **.H** i **.C**. To nie działało, oczywiście, w przypadku systemów operacyjnych, które nie rozróżniają wielkich i małych

liter, takich jak DOS. Producenci kompilatorów języka C++ dla systemu DOS używali takich rozszerzeń, jak hxx i cxx — odpowiednio do plików nagłówkowych oraz plików zawierających implementację albo do rozszerzeń **hpp** oraz **cpp**. Później ktoś doszedł do wniosku, że jedynym powodem stosowania różnych rozszerzeń nazw plików jest umożliwienie kompilatorowi określenia, czy kompiluje plik zawierający program w języku C, czy też C++. Ponieważ kompilator nigdy nie kompiluje bezpośrednio plików nagłówkowych, należy zmienić tylko rozszerzenie nazwy pliku, zawierającego implementację. Obecnie właściwie we wszystkich systemach przyjęto zwyczaj, polegający na używaniu rozszerzenia nazwy **cpp** do plików zawierających implementację oraz rozszerzenia **h** — do plików nagłówkowych. Zwróć uwagę na to, że w przypadku dodawania standardowych plików nagłówkowych języka C++ wykorzystywana jest możliwość stosowania plików nieposiadających rozszerzeń nazw, np. **#include <iostream>**.

Znaczniki początkowych i końcowych komentarzy

Bardzo istotną kwestią w tej książce jest to, by wszystkie widoczne w niej programy zostały zweryfikowane pod kątem poprawności (przynajmniej przez kompilator). Uzyskano to dzięki automatycznemu wyodrębnieniu zamieszczonych w niej plików. W tym celu wszystkie wydruki programów, które przewidziano do komplikacji (w odróżnieniu od nielicznych fragmentów kodu), zawierają znaczniki i komentarze, będące znacznikami początku i końca. Znaczniki te są wykorzystywane przez program narzędziowy wyodrębniający kod **ExtractCode.cpp**, zawarty w drugim tomie książki (dostępny w Internecie pod adresem <http://helion.pl/online/thinking/index.html>), wyrzaczający pliki zawierające kod na podstawie wydruków zawartych w tekście wersji książki.

Znacznik końca wydruku informuje program **ExtractCode.cpp**, że w tym miejscu kończy się program. Jednakże po znaczniku początku znajduje się informacja dotycząca podkatalogu, do którego plik należy (podkatalogi odpowiadają strukturze rozdziałów, więc plik zawarty w rozdziale 8. powinien mieć w tym miejscu znacznik C08). Po tej informacji następuje dwukropka i nazwa pliku źródłowego.

Ponieważ program **ExtractCode.cpp** dla każdego podkatalogu tworzy również pliki **makefile**, informacja o tym, jak utworzyć program oraz wiersz poleceń wykorzystywany do jego przetestowania, zawarte są również w kodzie źródłowym. Jeżeli program ma charakter samodzielny (nie musi być łączony z żadnym innym programem), to nie posiada żadnych dodatkowych informacji. Tak jest również w przypadku plików nagłówkowych. Jeżeli jednak plik nie zawiera funkcji **main()** i należy go połączyć z jakimś innym programem, to po nazwie pliku występuje znacznik {O}. Jeżeli wydruk stanowi natomiast program główny, ale musi zostać połączony z jakimś innymi komponentami, to znajduje się w nim oddzielnny wiersz, rozpoczynający się znacznikiem //{L}, poprzedzającym nazwy wszystkich plików, które muszą być z nim połączone (bez rozszerzeń nazw, ponieważ mogą być one różne dla różnych platform).

Przykłady wykorzystania tych znaczników można znaleźć w całej książce.

Jeżeli plik powinien zostać wyodrębniony, ale znaczniki początku i końca nie powinny znaleźć się w utworzonym pliku (jeżeli ten plik zawiera na przykład dane testowe), to bezpośrednio po znacznikach początku i końca następuje znak „;”.

Iawiasy okrągłe, klamrowe i wcięcia

Nietrudno zauważać, że stosowany w książce styl formatowania odbiega od wielu tradycyjnie stosowanych stylów języka C. Oczywiście każdy sądzi, że używany przez niego styl jest najbardziej praktyczny. Jednakże za stylem stosowanym w książce kryje się prosta logika. Zostanie ona przedstawiona poniżej wraz z przemyśleniami na temat tego, dlaczego rozwinięły się niektóre inne style kodowania.

Zastosowany styl formatowania jest spowodowany tylko jedną przyczyną — wymogami prezentacji zarówno w druku, jak i podczas prowadzonych na żywo seminariów. Być może twoje potrzeby są inne, ponieważ nie tworzysz zbyt wielu prezentacji. Jednakże kod jest znacznie częściej czytany niż pisany, więc powinien być łatwy w odbiorze. Moimi dwoma najważniejszymi kryteriami są „czytelność” (jak łatwiej jest dla czytelnika uchwycenie sensu pojedynczego wiersza) oraz liczba wierszy, które mieszczą się na stronie. To drugie może wydawać się zabawne, ale podczas prowadzenia prezentacji konieczność przesuwania przeszrocz do przodu i do tyłu jest bardzo irytująca — zarówno dla publiczności, jak i dla osoby prowadzącej.

Każdy chyba zgodzi się z tym, że kod zawarty w nawiasie klamrowym powinien być wcięty. Tym, co wywołuje kontrowersje, i jest zarazem sytuacja, w której poszczególne style formatowania najbardziej się między sobą różnią, jest problem umieszczenia otwierającego nawiasu klamrowego. Sądzę, że odpowiedź na to pytanie jest przyczyną powstania tylu odmian stylów kodowania (wyliczenie stosowanych stylów kodowania można znaleźć w książce Toma Pluma i Dana Saksa: „C++ Programming Guidelines”, Plum Hall, 1991). Podejmę próbę wykazania, że wiele stosowanych dzisiaj stylów kodowania wynika z ograniczeń istniejących przed utworzeniem standardu języka C (zanim powstały prototypy funkcji) i w związku z tym używanie ich nie jest uzasadnione.

Najpierw przedstawię odpowiedź na kluczowe pytanie: otwierający nawias klamrowy powinien zawsze znajdować się w tym samym wierszu, co jego „poprzednik” (pod tą nazwą rozumiem: „cokolwiek, czego dotyczy zawarte w nawiasie klamrowym ciało — klasę, funkcję, definicję obiektu, w przypadku instrukcji itp.”). Jest to pojedyncza, spójna reguła, którą stosuję w całym pisany przez siebie kodzie, która znacznie upraszcza formatowanie. Zwiększa również „czytelność” kodu. Odczytując wiersz:

```
int func(int a);
```

dzięki średnikowi znajdującemu się na końcu wiersza poznajemy, że jest to deklaracja, która nie jest już kontynuowana, ale w przypadku tego wiersza:

```
int func(int a) {
```

od razu wiemy, że jest to definicja, ponieważ wiersz kończy się nie średnikiem, tylko klamrowym nawiasem otwierającym. Dzięki zastosowaniu takiego sposobu miejsce, w którym jest umieszczony otwierający nawias klamrowy, jest identyczne w przypadku definicji wielowierszowej:

```
int func(int a) {  
    int b = a + 1;  
    return b * 2;  
}
```

i gdy definicja znajduje się tylko w jednym wierszu, co jest często stosowane w przypadku funkcji inline:

```
int func(int a) { return (a + 1) * 2; }
```

Podobnie w przypadku klas — wiersz:

```
class Thing;
```

jest deklaracją nazwy klasy, a zapis:

```
class Thing {
```

oznacza definicję klasy. We wszystkich tych przypadkach na podstawie pojedynczego wierszu kodu można określić, czy jest on **deklaracją**, czy też definicją. No i oczywiście, ulokowanie otwierającego nawiasu klamrowego w tym samym wierszu, zamiast w oddzielnym, umożliwia zmieszczenie na stronie większej liczby wierszy kodu.

Dlaczego istnieje zatem tyle innych stylów? Można w szczególności zauważyć, że większość programistów tworzy klasy, używając przedstawionego powyżej stylu (którym posłużył się również **Stroustrup** we wszystkich wydaniach swojej książki „*The C++ Programming Language*”, opublikowanych przez Addison-Wesley). Tworząc funkcje, umieszczają oni jednak otwierający nawias klamrowy w oddzielnym wierszu (co implikuje również stosowanie rozmaitych sposobów wcinania tekstu). Stroustrup również stosował taki zapis — z wyjątkiem krótkich funkcji inline. Dzięki stosowaniu opisanego przeze mnie sposobu, wszystko jest spójne — nazwa, czymkolwiek ona jest (**klassą, funkcją, wyliczeniem itp.**), poprzedza usytuowany w tym samym wierszu otwierający nawias klamrowy, który sygnalizuje, że dalej znajduje się ciało dla tej nazwy. Ponadto **klamrowy** nawias otwierający znajduje się zawsze w tym samym miejscu — zarówno w przypadku krótkich funkcji inline, jak i zwykłych definicji funkcji.

Twierdzę, że używany przez wielu styl, stosowany w przypadku definicji funkcji, ma swoje źródło w wersji języka C istniejącej przez utworzeniem w tym języku prototypów funkcji. Wówczas nie deklarowało się argumentów funkcji wewnątrz nawiasu, tylko pomiędzy nawiasem zamkającym i klamrowym nawiasem otwierającym (co świadczy o asemblerowych korzeniach języka C):

```
void bar()  
int x;  
float y;  
{  
/* ciało funkcji */  
}
```

W tym przypadku umieszczenie otwierającego nawiasu klamrowego w poprzednim wierszu byłoby dość niezręczne, więc nikt tego nie robił. Jednakże programiści musieli zdecydować się, czy nawiasy klamrowe powinny być wcięte, razem z zawartym w nich kodem, czy też powinny znajdować się na poziomie „poprzednika”. Stąd też wynika mnogość stosowanych stylów formatowania.

Są również argumenty, przemawiające za umieszczaniem otwierającego nawiasu klamrowego w następnym wierszu, za deklarację (klasy, struktury, funkcji itp.). Przytoczony poniżej argument pochodzi od czytelnika i został tu zamieszczony, by można było się zorientować, na czym polega problem:

Doświadczeni użytkownicy edytora „vi” (vim) **wiedzą**, że dzięki dwukrotnemu przyciśnięciu klawisza „J” użytkownik przechodzi do następnego wystąpienia w kolumnie zerowej znaku „{” (albo **^L**). Właściwość ta jest wyjątkowo przydatna podczas poruszania się w obrębie kodu (przeskok do następnej funkcji lub definicji klasy). [Mój komentarz: gdy rozpoczynałem pracę w systemie Unix, pojawił się **właśnie** edytor GNU Emacs, który bardzo mnie zaabsorbował. W rezultacie nigdy nie poznałem edytora „vi”, i dlatego nie rozumuję w terminach „pozycji kolumny zerowej”. Jednakże istnieje wielu użytkowników edytora „vi”, których ta kwestia dotyczy].

Umieszczenie znaku „{” w następnym wierszu eliminuje nieczytelny kod, zawarty w złożonych wyrażeniach warunkowych, zwiększać ich czytelność. Oto przykład:

```
if(cond1
  && cond2
  && cond3) {
    statement;
}
```

Powyższy fragment [twierdzi czytelnik] jest trudno czytelny. Natomiast następujący:

```
if (cond1
  && cond2
  && cond3)
{
  statement;
}
```

oddziela instrukcję „**if**” od jej ciała, dzięki czemu jest ona bardziej czytelna. [Twoje zdanie na ten temat będzie zależało od tego, do czego jesteś **przyzwyczajony**].

W końcu łatwiej jest ogarnąć wzrokiem nawiasy klamrowe, gdy są one wyrównane do tej samej kolumny. W ten sposób znacznie lepiej wyróżniają się one w tekście. [Koniec komentarza **czytelnika**].

Kwestia miejsca umieszczenia otwierającego **nawiasu** klamrowego wywołuje prawdopodobnie najwięcej sporów. Nauczyłem się czytać obie postacie zapisu i doszczętnie do końca do wniosku, że źródłem problemu są **przyzwyczajenia** użytkowników. Zwracam jednak uwagę na to, że oficjalny standard kodowania w języku Java (który znalazłem w **internetowej** witrynie firmy Sun, poświęconej językowi Java) jest właściwie taki sam, jak zaprezentowany w książce — ponieważ coraz więcej osób rozpoczęła programowanie w obu językach, spójność w używanych przez nie stylach kodowania może się **okazać** pomocna.

Dzięki stosowanej przeze mnie metodzie zostały wyeliminowane wszystkie wyjątki i przypadki szczególne, a także w logiczny sposób powstał jednolity styl stosowania wcięć. Spójność zostaje utrzymana nawet wewnątrz ciała funkcji, jak w poniższym przykładzie:

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

Styl ten jest łatwy — i do nauczenia, i do zapamiętania. Używana jest jedna, spójna reguła, dotycząca całego formatowania, a nie jedna dla klas, druga dla funkcji (z wariantami dla funkcji jednowierszowych i wielowierszowych), a jeszcze inne dla pętli **for**, instrukcji **if** itd. Spójność jest wartością, która sama w sobie warta jest rozważenia. Język C++ jest mimo wszystko **nowszym językiem** niż C i chociaż w wielu przypadkach musimy czynić ustępstwa na rzecz języka C, to nie powinniśmy czerpać z niego zbyt wielu elementów, które mogą się stać w przyszłości przyczyną problemów. Drobne problemy, pomnożone przez wiele wierszy kodu, stają się wielkimi problemami. Gruntowne omówienie tego tematu (choć w odniesieniu do języka C) znaleźć można w książce Davida Strakera: „C Style: Standards and Guidelines” (Prentice-Hall, 1992).

Innym ograniczeniem, z którym musiałem się liczyć, była szerokość wiersza. W książce może się bowiem znaleźć jedynie 50 znaków; jak jednak postąpić w przypadku, gdy się one nie mieścią? Ponownie starałem się wypracować spójną strategię dotyczącą tego, w jaki dzielone są wiersze, by były one wyraźnie widoczne. Dopóki stanowią one częśćjednej definicji, listy argumentów itp., dopóty wiersze stanowiące kontynuację powinny być wcięte o jeden poziom w stosunku do początku tej definicji, listy argumentów itp.

Nazwy identyfikatorów

Ci, którzy znajajęzyk Java, zauważą zapewne, że używam stylu tego języka w odniesieniu do wszystkich nazw identyfikatorów. Nie mogę być jednak w tej kwestii konsekwentny, ponieważ identyfikatory, zawarte w standardowych bibliotekach języków C oraz C++, nie stosują się do tej konwencji.

Styl ten jest dość prosty. Pierwsza litera identyfikatora jest zapisywana jako wielka tylko wówczas, gdy identyfikator jest nazwą klasy. Jeżeli jest nazwą funkcji lub zmiennej, to jest pisana małą literą. Pozostała część identyfikatora składa się z jednego lub większej liczby słów, połączonych razem, wyróżnionych jednak przez zapisanie wielką literą pierwszego znaku każdego słowa. Tak więc nazwa klasy może być następująca:

```
class FrenchVanilla : public IceCream {
```

z kolei identyfikator obiektu:

```
FrenchVanilla myIceCreamCone();
```

a funkcja — w taki sposób:

```
void eatIceCreamCone();
```

(zarówno funkcja składowa, jak i zwyczajna funkcja).

Jedyny wyjątek dotyczy stałych czasu komplikacji (zdefiniowanych za pomocą słowa kluczowego **const** lub dyrektywy **#define**), których identyfikatory pisane są w całości wielkimi literami.

Korzyścią wynikającą z tego stylu jest to, że wielkość liter ma swoje znaczenie — już na podstawie pierwszej wiadomo, czy mamy do czynienia z **klassą**, czy też obiektem lub funkcją składową. Jest to szczególnie przydatne przy odwołaniach do statycznych składowych klas.

olejnośćłączania lików nagłówkowych

Pliki nagłówkowe są dołączane w kolejności od „najbardziej wyspecjalizowanego do najbardziej ogólnego”. Oznacza to, że kolejno dodawane są: wszelkie pliki nagłówkowe, znajdujące się w lokalnym katalogu, pliki nagłówkowe przygotowanych przeze mnie „narzędzi”, takich jak **require.h**, pliki nagłówkowe **niezależnych** dostawców, pliki nagłówkowe standardowej biblioteki języka C++, a na końcu pliki nagłówkowe biblioteki języka C.

Uzasadnienie dołączania plików nagłówkowych w takiej właśnie kolejności pochodzi z książki Johna Lakosa „**Large-Scale C++ Software Design**” (Addison-Wesley, 1996):

Można umknąć ukrytych błędów, zapewniając, że pliki .h komponentów będą sprawdzali się samoczynnie — bez udziału jakichkolwiek dostarczonych z zewnątrz deklaracji czy definicji... Dołączenie plików .h na samym początku plików .c gwarantuje, że w plikach .h nie zabraknie żadnej istotnej informacji, niezbędnej do fizycznego interfejsu komponentów (ażejeli jej nie ma, to przekonasz się o tym, gdy tylko zaczniesz kompilować plik .c).

Jeżeli pliki nagłówkowe są dołączane w kolejności od „najbardziej wyspecjalizowanego do najbardziej ogólnego”, to jest bardziej prawdopodobne, że w przypadku gdy plik nagłówkowy będzie zawierał błędy, dowiemy się o tym szybciej, zapobiegając przykrym niespodziankom.

Strażniki dołączania w plikach nagłówkowych

Strażniki dołączania (ang. *include guards*) są używane zawsze w plikach nagłówkowych, zapobiegając wielokrotnemu dołączeniu tego samego pliku nagłówkowego w trakcie komplikacji pojedynczego pliku .cpp. Są one **zaimplementowane** za pomocą dyrektywy preprocessora **#define** i sprawdzenia, czy dany strażnik nie został już zdefiniowany. Nazwa strażnika jest tworzona na podstawie nazwy pliku nagłówkowego, zapisanej wielkimi literami, z kropką zastąpioną znakiem podkreślenia. Na przykład:

```
// IncludeGuard.h
#ifndef INCLUDEGUARD_H
#define INCLUDEGUARD_H
// Zawartość pliku nagłówkowego...
#endif // INCLUDEGUARD_H
```

Widoczny w ostatnim wierszu identyfikator został dodany w celu zwiększenia przejrzystości. Mimo że niektóre preprocessory ignorują wszelkie znaki znajdujące się po dyrektywie **#endif**, nie jest to **reguła**, więc identyfikator został poprzedzony znakami komentarza.

Wykorzystanie przestrzeni nazw

Należy skrupulatnie wystrzegać się „zanieczyszczenia” przestrzeni **nazw**, w której dołączany jest plik nagłówkowy. A zatem zmieniając przestrzeń nazw na zewnątrz funkcji lub klasy spowodujemy, że modyfikacja ta będzie dotyczyła każdego pliku, w którym dołączany jest nasz plik nagłówkowy, skutkując wszelkiego rodzaju problemami. W plikach nagłówkowych nie są więc dozwolone żadne deklaracje **using**, znajdujące się poza funkcjami lub klasami, ani żadne globalne dyrektywy **using**.

Użycie globalnych dyrektyw **using** w plikach **cpp** dotyczy wyłącznie bieżącego pliku, więc w książce są one na ogół używane — w celu zwiększenia czytelności kodu, szczególnie w przypadku małych programów.

Użycie funkcji **require()** i **assure()**

Funkcje **require()** i **assure()**, zdefiniowane w pliku **require.h**, są konsekwentnie używane w większości przykładów zawartych w książce, umożliwiając poprawne informowanie o sytuacjach problemowych. Jeżeli znane ci są pojęcia *warunków wstępnych* (ang. *preconditions*) i *warunków końcowych* (ang. *postconditions*), wprowadzone przez **Bertranda Mayera**, zauważysz, że użycie funkcji **require()** i **assure()** w mniejszym lub większym stopniu określa warunki wstępne (zazwyczaj) oraz warunki końcowe (czasami). Tak więc na początku funkcji, przed wykonaniem jakichkolwiek

instrukcji tworzących jej „rdzeń”, sprawdzane są warunki wstępne, by się przekonać, że wszystko jest w porządku i wszystkie niezbędne warunki zostały spełnione. Następnie wykonywane są instrukcje „rdzenia” funkcji i czasami weryfikowane są pewne warunki końcowe, by sprawdzić, czy nowy stan danych mieści się w przyjętym zakresie parametrów. Nietrudno zauważyć, że warunki końcowe sprawdzane są w książce rzadko, a funkcja **assure()** służy przede wszystkim do upewnienia się, że otwarcie plików zakończyło się powodzeniem.

Dodatek B

Wskazówki

dla programistów

Dodatek ten jest zbiorem sugestii dotyczących programowania w języku C++. Sformułowałem je na podstawie doświadczeń w dziedzinie pracy dydaktycznej oraz programowania, a także wskazówek grona przyjaciół, do których zaliczają się: Dan Saks (wraz z Tomem Plumem, współautor „**C++ Programming Guidelines**”, Plum Hall, 1991), Scott Meyers (autor „**Effective C++**”, wydanie drugie, Addison-Wesley, 1998) oraz Rob Murray (autor „**C++ Strategies & Tactics**”, Addison-Wesley, 1993). Znajduje się w nim również podsumowanie wielu wskazówek, przedstawionych na kartach książki „**Thinking in C++**”.

1. Niech przede wszystkim działa, a dopiero potem niech działa szybko.
Jest to **prawdą**, nawet gdy **masz** pewność, że jakiś fragment kodujesz naprawdę ważny i będzie głównym wąskim gardłem twojego systemu. Nie rób tego. Niech system będzie najpierw możliwie jak najprostszym projektem. Dopiero później, gdy nie będzie zbyt szybki, zajmij się jego profilowaniem. Niemal zawsze przekonasz się, że „twoje” wąskie gardło niejest problemem. Oszczędzaj czas na naprawdę ważne kwestie.
2. Elegancja zawsze się opłaca. To nie tylko sztuka dla sztuki. Powoduje, że uzyskujesz program, który nie tylko jest łatwiejszy do zbudowania i uruchomienia, ale również do zrozumienia i utrzymania, a to już na wymiar finansowy. Aby w to uwierzyć, potrzebne jest pewne doświadczenie. Można bowiem odnieść wrażenie, że próbując uczynić jakiś fragment programu eleganckim, nie pracuje się wydajnie. Na wydajność przyjdzie pora, gdy kod bez problemów zintegruje się z resztą systemu, a w jeszcze większym stopniu — gdy kod lub system będą modyfikowane.
3. Pamiętaj o zasadzie „*dziel i rządź*”. Jeżeli problem, z którym masz do czynienia, jest zbyt złożony, spróbuj wyobrazić sobie podstawowe operacje, które realizowałby program, przyjmując założenie **istnienia jakiegoś** czarodziejskiego „fragmentu”, obsługującego najtrudniejsze części. Ten „fragment” jest obiektem — napisz najpierw wykorzystujący go kod, a następnie przyjrzyj się samemu obiekutowi, **zamykając jego** najtrudniejsze elementy w kolejnych obiektach itd.

4. Nie przepisuj automatycznie kodu napisanego w języku C na kod w języku C++, o ile nie zamierzasz znacznie zmienić jego funkcjonalności (to znaczy – nie naprawiaj go, jeżeli działa poprawnie). **Przekompilowanie** kodu języka C w języku C++ jest pożyteczne, ponieważ może ujawnić ukryte w programie błędy. Jednakże przepisywanie w języku C++ dobrze działającego kodu napisanego w języku C może nie być najlepszym sposobem wykorzystania czasu, chyba że wersja języka C++ nadaje się do wielokrotnego wykorzystania jako **klasa**.
5. Jeżeli posiadasz duży fragment kodu w języku C, wymagający wprowadzenia zmian, najpierw wyodrębnij tejego części, które nie będą modyfikowane, być może zamkając je w „klasach interfejsu programowego” w postaci statycznych funkcji składowych. Następnie skup się na kodzie, który będzie zmieniany, dzieląc go na klasy, co ułatwi wprowadzanie modyfikacji, wymaganych przez pielęgnację kodu.
6. Oddziel twórcę klasy od jej użytkownika (**klienta-programisty**). Użytkownik klasy jest „klientem” i nie musi (albo nie chce) wiedzieć, co dzieje się wewnątrz klasy. Twórca klasy powinien być ekspertem w projektowaniu i tworzeniu klasy w taki sposób, aby mogła zostać ona wykorzystana nawet przez najbardziej niedoświadczonych programistów, działając nadal niezawodnie w aplikacjach. Korzystanie z biblioteki będzie łatwe tylko wówczas, gdy będzie ona przejrzysta.
7. Tworząc klasę, używaj w miarę możliwości zrozumiałych nazw. Twoim celem powinno być utworzenie interfejsu **klienta-programisty** w taki sposób, aby był prosty pojęciowo. Spróbuj użyć nazw na tyle zrozumiałych, by nie wymagały dodatkowych komentarzy. Aby to uzyskać, wykorzystuj przeciążanie nazw funkcji i argumenty domyślne, tworząc intuicyjny, łatwy w użyciu interfejs.
8. Kontrola dostępu pozwala ci (twórcy klasy) na wprowadzenie w przyszłości zmian, bez potrzeby modyfikacji kodu użytkownika, w którym klasa ta jest używana. Zachowaj zatem wszystko prywatne, o ile to możliwe, pozostawiając **publicznym jedynie** interfejs klasy. Niech dane będą publiczne tylko wówczas, gdy jesteś do tego zmuszony. Jeżeli użytkownik klasy nie potrzebuje dostępu do jakiejś funkcji, uczyń ją **prywatną**. Jeżeli jakaś część klasy musi być dostępna klasom **pochodnym jako** chroniona, to udostępnij w tym celu interfejs w postaci funkcji, nie za dane. Dzięki temu zmiany w implementacji tej klasy będą miały minimalny wpływ na jej klasy pochodne.
9. Nie popadaj w analityczną niemoc. Są kwestie, o których się nie dowiesz, dopóki nie rozpoczęsz kodowania i twój system nie stanie się jako tako działającym systemem. Język C++ posiada wbudowane zapory **przeciwogniowe** – pozwól im pracować dla ciebie. Pomyłki, popełnione przez ciebie w jakiejś klasie lub grupie klas, nie naruszą integralności całego systemu.
10. Wynikiem analizy i projektowania musząbyć co najmniej: klasy zawarte w systemie, ich publiczne interfejsy oraz związki z innymi klasami – szczególnie z klasami podstawowymi. Jeżeli stosowana przez ciebie metoda programowania generuje coś więcej, to odpowiedz sobie na pytanie, czy wszystkie tworzone przez nią elementy mają wartość w trakcie życia programu. Jeżeli nie, to właśnie ty poniesiesz ich koszt. Członkowie zespołów

tworzących oprogramowanie zazwyczaj nie zajmują się sprawami, które nie wpływają na ich wydajność — to prawda życiowa, której nie uwzględniono w wielu metodach projektowych.

11. Zanim utworzysz klasę, napisz kod, który ją testuje, i pozostaw go wewnątrz klasy. Automatyzuj uruchamianie testów za pomocą pliku **makefile** lub **jakiegoś** podobnego narzędzia. Wszelkie zmiany będą mogły zostać automatycznie zweryfikowane dzięki uruchomieniu kodu testującego, co pozwoli na natychmiastowe wykrycie błędów. Ponieważ wiesz, iż posiadasz siatkę zabezpieczającą w postaci szkieletu testującego, będzieś odważniejszy we wprowadzaniu gruntownych zmian, gdy odkryjesz konieczność ich dokonania. **Pamiętaj**, że największe udoskonalenia wjazykach programowania wynikają wzbudowanych w nie testów, które udostępniają kontrolę typów, obsługę wyjątków itp., ale siegają one tylko do pewnego miejsca. Musisz pójść dalej, tworząc niezawodny system za pomocą testów, weryfikujących cechy specyficzne dla określonej klasy lub całego programu.
12. Zanim utworzysz klasę, napisz kod, który będzie ją testował. Upewnij się dzięki temu, że projekt klasy jest kompletny. Jeżeli nie potrafisz napisać kodu testującego klasę, oznacza to, że nie wiesz, jak powinna ona wyglądać. Ponadto pisanie kodu testującego nierzadko prowadzi do ujawnienia dodatkowych właściwości lub ograniczeń, których wymagać będzie klasa — często nie ujawniają się one podczas analizy i projektowania.
13. Pamiętaj o fundamentalnej zasadzie inżynierii programowania¹: *Wszystkie problemy projektowe powstałe podczas projektowania oprogramowania mogą zostać uproszczone przez wprowadzenie dodatkowego, pośredniego poziomu pojęciowego*. Ta idea jest podstawą abstrakcji, podstawowej właściwości programowania obiektowego.
14. Niech klasy będą tak niepodzielne, jak to tylko możliwe — każda klasa powinna mieć tylko jedno, wyraźnie określone zadanie. Jeżeli projekt klas lub całego systemu staje się zbyt skomplikowany, rozbij złożone klasy na prostsze. Najbardziej oczywistym wskaźnikiem jest wielkość klasy — jeżeli klasa jest duża, to prawdopodobnie pełni zbyt wiele funkcji i powinna zostać podzielona.
15. Przyjrzyj się długim definicjom funkcji składowych. Długie i skomplikowane funkcje są również trudne i kosztowne w utrzymaniu i prawdopodobnie próbują zrealizować samodzielnie zbyt wiele zadań. Funkcję taką **należy** przynajmniej rozbić na wiele funkcji. Może to również stanowić sugestię utworzenia dodatkowej klasy.
16. Przyjrzyj się długim listom argumentów. Powodują one, że wywołania funkcji są **trudne** do napisania, przeczytania i pielęgnacji. Spróbuj przenieść funkcję **składową do klasy**, do której pasowałaby bardziej, i (lub) **przekazywać jej** obiekty w charakterze argumentów.

¹ Którą wyjaśnił mi Andrew Koenig.

17. Nie powtarzaj się. Jeżeli jakiś kod powieła się w wielu funkcjach klas pochodnych, umieść go w jednej funkcji klasy podstawowej i wywołuj tę funkcję w funkcjach klas pochodnych. Nie tylko zmniejszysz w ten sposób wielkość kodu, ale również ułatwisz propagację wprowadzonych zmian. Dla zachowania efektywności możesz użyć w tym celu funkcji inline. Czasami odkrycie takiego wspólnego kodu zwiększa w istotny sposób funkcjonalność interfejsu.
18. Przyjrzyj się instrukcjom **switch** i łańcuchowym instrukcjom **if-else**. Świadczą one zazwyczaj o *kodowaniu ze sprawdzaniem typów*, co oznacza, że na podstawie jakiejś informacji o typie wybierany jest kod, który będzie wykonywany (na pierwszy rzut oka dokładny typ może nie wydawać się oczywisty). Taki kod można na ogół zastąpić, wykorzystując do tego celu dziedziczenie i **polimorfizm – polimorficzne** wywołanie funkcji dokona za ciebie sprawdzenia typu, a ponadto będzie bardziej niezawodne i łatwiejsze w rozbudowie.
19. Patrząc z punktu widzenia projektu, znajdź i oddziel to, co będzie ulegało modyfikacjom, od tego, co pozostanie niezmienne. Poszukaj elementów, które będziesz chciał zmieniać, nie modyfikując samego projektu, a następnie zamknij je w **klasach**. Znacznie więcej informacji na temat tej idei można znaleźć w rozdziale poświęconym wzorom projektowym, zamieszczonym w drugim tomie książki, dostępnym w witrynie <http://helion.pl/online/thinking/index.html>.
20. Zwróć uwagę na *wariancję*. Dwa znaczeniowo różne obiekty mogą podejmować identyczne działania albo mieć takie same obowiązki, w związku z czym pokusa uczynienia jednego z nich **podklassą drugiego**, w celu wykorzystania dziedziczenia, może wydawać się naturalna. Jest to nazywane *wariancją*, ale naprawdę nie ma żadnego usprawiedliwienia dla wymuszania wzajemnych relacji nadklasy i podklasy tam, gdzie ich naprawdę nie ma. Lepszym rozwiązaniem jest utworzenie ogólnej klasy podstawowej, tworzącej interfejs dla obu tych klas, będących jej klasami pochodnymi — wymaga to nieco więcej miejsca, ale zachowuje się korzyści wynikające z dziedziczenia. Przy okazji można także dokonać jakiegoś ważnego odkrycia związanego z projektem.
21. Zwróć uwagę na *ograniczenia* związane z dziedziczeniem. Dobre projekty prowadzą dodawania w klasach pochodnych nowych możliwości. Podejrzane projekty usuwają natomiast podczas dziedziczenia istniejące własności, nie dodając żadnych nowych. Reguły są jednak po to, by je łamać, i jeśli używasz starej biblioteki klas, to bardziej efektywne może okazać się ograniczenie istniejącej klasy w jej klasie pochodnej niż zmiana struktury całej hierarchii w taki sposób, by nowa klasa została umieszczona tam, gdzie powinna — powyżej starej klasy.
22. Nie rozszerzaj podstawowej funkcjonalności **metodą tworzenia** klas pochodnych. Jeżeli jakiś element interfejsu jest dla klasy niezbędny, to powinien on znajdować się w klasie podstawowej, a nie zostać dodany podczas dziedziczenia. Jeżeli dodajesz funkcje składowe, używając do tego celu dziedziczenia, to powinieneś jeszcze raz zastanowić się nad swoim projektem.

23. Mniej znaczy więcej. *Zacznij od minimalnego interfejsu klasy, tak małego i prostego, by rozwiązywał on aktualny problem, nie próbując przewidywać wszystkich sposobów, na jakie **mogłyby** zostać wykorzystana ta klasa.* W trakcie używania klasy dowiesz się, w jaki sposób musisz rozszerzyć jej interfejs. Jednak gdy klasa jest używana, nie można ograniczyć jej interfejsu, nie naruszając kodu jej użytkowników. Jeżeli trzeba dodać większą liczbę funkcji, nie stanowi to problemu — nie wpłynie na kod użytkowników w żaden inny sposób niż konieczność powtórnej komplikacji. Ale nawet w przypadku gdy nowe funkcje zastępują funkcjonalnie stare, należy pozostawić bez zmian istniejący interfejs (funkcjonalność można połączyć na poziomie wewnętrznej implementacji). Jeżeli zamierzasz rozszerzyć interfejs istniejącej funkcji, dodając do niej większą liczbę argumentów, to pozostaw istniejące argumenty w ich aktualnym porządku, określając wartości domyślne wszystkich dodatkowych argumentów — w taki sposób nie wpłynie to na postać istniejących już wywołań tej funkcji.
24. Odczytuj nazwy swoich klas na głos, by upewnić się, że są one logiczne, określając relację pomiędzy klasą podstawową i klasą pochodną jako „**jest**”, a w przypadku obiektów składowych używając pojęcia „**posiada**”.
25. Gdy wahasz się pomiędzy dziedziczeniem i kompozycją, odpowiedz sobie na pytanie, czy będziesz potrzebował rzutowania w górę do klasy podstawowej. Jeżeli nie, to zamiast dziedziczenia wybierz kompozycję (obiekty składowe). Pozwoli to na uniknięcie konieczności wielokrotnego dziedziczenia. Gdy zastosujesz dziedziczenie, użytkownicy będą sądzili, że oczekuje się od nich rzutowania w górę.
26. Czasami występuje konieczność dziedziczenia w celu uzyskania dostępu do chronionych składowych klasy podstawowej. Może to prowadzić do powstania konieczności wielokrotnego dziedziczenia. Jeżeli nie jest ci potrzebne rzutowanie w górę, to najpierw utwórz klasę pochodną, by uzyskać dostęp do składowych chronionych. **Następnie**, zamiast używać dziedziczenia, utwórz obiekt składowy powstały wcześniej klasy we wszystkich klasach, w których jest ci to potrzebne.
27. Klasa **podstawowa jest** na ogół używana przede wszystkim dla określenia interfejsu wyprowadzonych z niej klas. Tak więc gdy tworzysz klasę **podstawową**, domyślnie utwórz jej funkcje **składowe jako** funkcje czysto wirtualne. Jej destruktor również powinien być czysto wirtualny (by wymusić na klasach pochodnych jego zasłonięcie); pamiętaj jednak, by utworzyć treść tego destruktora, ponieważ zawsze wywoływane są wszystkie destruktory znajdujące się w hierarchii.
28. Gdy umieszczasz w **klasie jakąś** funkcję **wirtualną**, uczynь wszystkie funkcje tej klasy funkcjami wirtualnymi i ulokuj w niej wirtualny destruktor. Pozwoli to na uniknięcie niespodzianek w funkcjonowaniu interfejsu. *Zacznij usuwać słowa kluczowe **virtual** dopiero wówczas, gdy będziesz szukał metod poprawy efektywności, a używany przez ciebie program profilujący wskaże ci taki kierunek.*

29. Do wyrażenia zmian dotyczących wartości używaj danych składowych, a w przypadku zmian w zachowaniu — funkcji wirtualnych. Innymi słowy, jeżeli znajdziesz klasę, wykorzystującą zmienne stanu oraz funkcje składowe, zmieniające swoje działanie na podstawie wartości tych zmiennych, to należy prawdopodobnie przeprojektować, wyrażając różnice w zachowaniu za pomocą klas pochodnych i zasłaniania funkcji wirtualnych.
30. Jeżeli musisz wykonać jakąś nieprzenośną usługę, to utwórz dla niej abstrakcję i umieść ją wewnątrz klasy. Ten dodatkowy poziom pośrednictwa zapobiegnie rozprzestrzenieniu się tego nieprzenośnego działania na cały program.
31. Unikaj wielokrotnego dziedziczenia. Umożliwia ono znalezienie wyjścia z niekorzystnych sytuacji — szczególnie naprawę interfejsu nieprawidłowo działającej klasy, nad którą nie mamy kontroli (zob. drugi tom książki). Zanim przystąpisz do projektowania wielokrotnego dziedziczenia w swoim systemie, powinieneś stać się najpierw doświadczonym programistą.
32. Nie używaj prywatnego dziedziczenia. Mimo że jest ono w języku i czasami wydaje się przydatne, to w połączeniu z identyfikacją typów podczas pracy programu wprowadza do niego znaczny bałagan. Zamiast używać prywatnego dziedziczenia utwórz wewnątrz klasy prywatną składową.
33. Jeżeli dwie klasy są w jakiś sposób funkcjonalnie powiązane (tak jak kontenery i iteratory), to spróbuj uczynić jedną z nich publiczną, a drugą zagnieżdżoną w pierwszej klasie, tak jak robi to standardowa biblioteka C++ w przypadku iteratorów i kontenerów (odpowiednie przykłady zaprezentowano w ostatniej części rozdziału 16.). Nie tylko podkreśla to związek pomiędzy tymi dwoma klasami, ale również pozwala na powtórne użycie nazwy klasy przez zagnieżdżenie jej wewnątrz innej klasy. Standardowa biblioteka C++ wykorzystuje to, definiując zagnieżdżoną klasę iteratora wewnątrz każdej klasy kontenera, co zapewnia kontenerom jednolity interfejs. Innym powodem takiego działania może być zamiar zagnieżdżenia klasy, stanowiącej element prywatnej implementacji klasy. W tym przypadku korzyścią wynikającą z zagnieżdżania jest raczej ukrycie implementacji niż wymieniony powyżej związek pomiędzy klasami, czy choć ochrony przestrzeni nazw przed jej „zaśmiecieniem”.
34. Przeciążanie operatorów to tylko „cukierek składniowy” — inny sposób wywoływania funkcji. Jeżeli przeciążenie operatora nie powoduje, że interfejs klasy jest bardziej przejrzysty i łatwiejszy w użyciu, to nie należy tego robić. Utwórz w każdej klasie tylko jeden operator automatycznej konwersji typu. Na ogół podczas przeciążania operatorów postępuj zgodnie ze wskazówkami i przykładowym formatem, przedstawionymi w rozdziale 12.
35. Nie stań się ofiarą przedwczesnej optymalizacji. W szczególności nie przejmuj się pisaniem (albo unikaniem pisania) funkcji **inline**, zamianą niektórych funkcji na niewirtualne czy też cyzelowaniem kodu, by był on efektywny, gdy jesteś dopiero na etapie tworzenia systemu. Twoim pierwszym celem powinno być dowiedzenie poprawności projektu, chyba że sam projekt wymaga już pewnej efektywności.

36. W normalnym przypadku nie pozwól na to, by kompilator tworzył za ciebie **konstruktory, destruktory i operator=**. Projektanci klas powinni zawsze dokładnie określić, co powinny one robić, i przez cały czas zachowywać nad nimi pełną kontrolę. Jeżeli nie chcesz mieć konstruktora kopiącego ani operatora =, to zadeklaruj je jako prywatne. Pamiętaj, że jeżeli utworzysz jakikolwiek konstruktor, zapobiegnie to wygenerowaniu przez kompilator domyślnego konstruktora.
37. Jeżeli Twoja klasa zawiera wskaźniki, to, aby działała ona poprawnie, musisz utworzyć dla niej konstruktor kopiący, **operator=** oraz destruktor.
38. Pamiętaj, że kiedy dla klasy pochodnej tworzysz konstruktor kopiący, musisz jawnie wywołać konstruktor kopiący klasy podstawowej (a także obiektów składowych — zob. rozdział 14.) Jeżeli tego nie zrobisz, dla klasy podstawowej (oraz obiektów składowych) zostanie wywołany domyślny konstruktor, a do tego zapewne dążysz. Podczas wywołania konstruktora kopiącego klasy podstawowej należy przekazać mu dziedziczony obiekt, z którego odbywa się kopianie:
39. Pochodna(const Pochodna& p) : Podstawa(p) { // ...}
40. Podczas tworzenia dla klasy pochodnej operatora przypisania pamiętaj o jawnym wywołaniu operatora przypisania klasy podstawowej (zob. rozdział 14.). Jeżeli tego nie zrobisz, kopianie nie zostanie zrealizowane (to samo dotyczy obiektów składowych). Aby wywołać operator przypisania klasy podstawowej, użyj nazwy klasy podstawowej i operatora zasięgu:
 41. Pochodna& operator=(const Pochodna& p) {
 42. Podstawa::operator=(p);
43. Jeżeli chcesz zminimalizować konieczność powtórnej komplikacji w trakcie tworzenia dużego systemu, użyj techniki wykorzystującej **klasy-uchwyty** (zwane również „kotem z Cheshire”), zademonstrowane w rozdziale 5., a później usuń je tylko w **przypadku**, gdyby efektywność wykonania programu stanowiła problem.
44. Unikaj preprocessora. Używaj zawsze słowa kluczowego **const** do podstawiania wartości, a funkcji **inline** zamiast makroinstrukcji.
45. Zachowaj zasięgi **możliwie jak najmniejsze**, aby jak najbardziej zminimalizować widoczność i czas twoich obiektów. Zabieg ten zmniejsza prawdopodobieństwo użycia obiektu w niewłaściwy sposób, a także ogranicza możliwości powstania ukrytych, trudnych do wykrycia błędów. Założymy na przykład, że masz kontener i fragment kodu, który porusza się po zawartym w nim elementach. Jeżeli skopiujesz ten kod z zamiarem użycia go z nowym kontenerem, możesz przypadkowo użyć wielkości starego kontenera jako górnej granicy liczby elementów zawartych w nowym kontenerze. Jeżeli jednak stary kontener będzie znajdował się poza zasięgiem, błąd zostanie wykryty na etapie komplikacji.
46. Unikaj zmiennych globalnych. Zawsze staraj się umieszczać dane w obrębie klas. Bardziej prawdopodobne jest naturalne wystąpienie funkcji globalnych niż zmiennych globalnych, chociaż możesz później odkryć, że jakaś funkcja globalna powinna być raczej statyczną funkcją składową klasy.

47. Jeżeli musisz zadeklarować jakąś klasę lub **funkcję**, zawartą w **bibliotece**, zawsze rób to dołączając jej plik nagłówkowy. Jeżeli na przykład zamierzasz napisać funkcję, która zapisuje informacje do strumienia typu **ostream**, to nigdy nie deklaruj tego strumienia, używając niepełnej **specyfikacji** typu w rodzaju:
48. class ostream;
49. Taki postępowanie powoduje, że kod staje się podatny na zmiany dotyczące reprezentacji (na przykład identyfikator **ostream** mógłby być w rzeczywistości zadeklarowany za pomocą słowa kluczowego **typedef**). Zawsze natomiast używaj pliku nagłówkowego:
50. #include <iostream>
51. Podczas tworzenia własnych klas, jeśli biblioteka jest duża, udostępnij jej użytkownikowi skróconą postać pliku nagłówkowego, zawierającego niekompletne specyfikacje typów (czyli deklaracje nazw klas), na wypadek, gdyby używał on wyłącznie wskaźników (pozwoli mu to na przyspieszenie komplikacji).
52. Wybierając typ wartości, zwracanej przez przeciążony operator, zastanów się, co się stanie, gdy wyrażenia są połączone łańcuchowo. Zwróć kopię lub referencję do **l-wartości** (używając instrukcji **return *this**), dzięki czemu operator będzie mógł zostać wykorzystany w wyrażeniach łańcuchowych ($A = B = C$). Gdy definiujesz **operator=**, pamiętaj o przypadku $x=x$.
53. Gdy piszesz funkcję, preferuj przekazywanie jej argumentów w postaci referencji do stałych. Dopóki nie musisz modyfikować przekazywanego obiektu, taki sposób przekazywania **argumentów jest najlepszy**. Cechuje go bowiem prostota przekazywania przez wartość, lecz nie wymaga kosztownych konstrukcji i destrukcji, związanych z tworzeniem **obiektu** lokalnego. Zwykle projektując i tworząc swój system, nie powinieneś przejmować się zbytnio kwestiami efektywności, ale nawyk taki gwarantuje powodzenie.
54. Zwracaj uwagę na obiekty tymczasowe. Gdy zależy ci na wydajności, śledź pilnie tworzenie obiektów tymczasowych, szczególnie w przypadku przeciążania operatorów. Jeżeli konstraktor i destruktor twojej klasy są skomplikowane, to koszt tworzenia i niszczenia obiektów tymczasowych może być wysoki. Gdy zwracasz wartość z funkcji, zawsze staraj się tworzyć obiekt „w miejscu”, bez wywoływania konstruktora w instrukcji **return**, dokonując następującego zapisu:
55. return MojTyp(i,j);
56. a nie taki:
- 57. MojTyp x(i, j);**
58. return x;
59. Pierwsza z widocznych powyżej instrukcji **return** (wykorzystująca tak zwaną *optymalizację zwracania wartości*) eliminuje wywołanie konstruktora kopiującego oraz destruktora.

60. Gdy tworzysz **konstruktory**, weź pod uwagę wyjątki. W najlepszym razie konstruktor nie powinien zrobić niczego, co powoduje zgłoszenie wyjątku. Dobrym rozwiązaniem jest również przypadek, w którym klasa jest skomponowana i dziedziczy wyłącznie po solidnych klasach, dzięki czemu dokonają one automatycznie porządkowania w razie zgłoszenia wyjątku. Jeżeli musisz używać zwykłych wskaźników, to jesteś odpowiedzialny za wykrycie swoich wyjątków i zwolnienie wszelkich wskaazywanych przez nie zasobów, zanim zgłosisz wyjątek w swoim konstruktorze. Jeśli konstruktor musi zakończyć się **niepowodzeniem**, to właściwym **działaniem jest zgłoszenie wyjątku**.
61. Wykonuj w konstruktorach wyłącznie działania absolutnie konieczne. Nie tylko powoduje to mniejszy narzut, związany z wywołaniem konstruktora (spośród których wiele może znajdować się poza twoją kontrolą), ale również jest mniej prawdopodobne, że konstruktory takie zgłoszą wyjątki lub będą przyczyną problemów.
62. Obowiązkiem **destruktora jest** zwolnienie zasobów przydzielonych w czasie życia obiektu, a nie tylko podczas konstrukcji.
63. Używaj hierarchii wyjątków, najlepiej wyprowadzonych ze standardowej hierarchii wyjątków języka C++ i zagnieżdżonych jako klasy publiczne wewnętrz tych klas, które zgłaszają wyjątki. Dzięki temu osoba wyławiająca wyjątki może wykrywać szczególne typy wyjątków, a następnie wyjątki typu podstawowego. Jeżeli dodasz nowe, pochodne wyjątki, to istniejący kod klienta nadal je wyławia, używając ich typu podstawowego.
64. Zgłaszasz wyjątki przez wartość, a wyławuj je przez referencję. Pozwól, by mechanizm obsługi wyjątków zajął się obsługą pamięci. Jeżeli będziesz zgłaszał wskaźniki do obiektów wyjątków, które zostały utworzone na stercie, to kod wychwytyujące musi wiedzieć o tym, że powinien zniszczyć wyjątek, co **nie jest** najlepszym rozwiązaniem. Jeżeli wyławujesz wyjątek przez wartość, wywołujesz dodatkowe konstrukcje i destrukcje, a co gorsza — pochodna część wykrytego przez ciebie obiektu wyjątku może zostać okrojona podczas rzutowania w góre.
65. Nie pisz własnych szablonów klas, dopóki nie musisz. Zajrzyj najpierw do standardowej biblioteki języka C++, a następnie poszukaj ich u producentów, tworzących specjalizowane narzędzia. Biegły opanuj ich używanie, co znacznie zwiększy **twoją produktywność**.
66. Podczas tworzenia szablonów przyjrzyj się kodowi, który nie zależy od typu, umieszczając go w klasie podstawowej, niebędącej szablonem; zapobiegajesz w ten sposób niepotrzebnemu rozrastaniu się kodu. Używając dziedziczenia lub kompozycji, możesz utworzyć szablony, w których większość kodu zależy od typu i jest zatem niezbędna.
67. Nie używaj funkcji zawartych w pliku nagłówkowym **<cstdio>**, takich jak **printf()**. Naucz się natomiast używać strumieni **wejścia-wyjścia** — są one bezpieczne dla typów, można je rozszerzać o obsługę nowych, są także znacznie bardziej efektywne. Twój trud będzie regularnie nagradzany. Zawsze raczej używaj bibliotek C++ niż bibliotek języka C.

68. Unikaj typów **wbudowanych języka C**. Sąone obsługiwane przez język C++ w celu zapewnienia wstępnej zgodności, ale są znacznie mniej solidne niż klasy języka C++. Używanie ich spowoduje więc wydłużeniu się czasu, poświęconego poszukiwaniu błędów.
69. Gdy używasz typów wbudowanych jako zmiennych globalnych lub automatycznych, nie definiuj ich, dopóki nie będziesz ich mógł również zainicjalizować. Definiuj każdą zmienną w oddzielnym wierszu, od razu ją inicjalizując. Gdy definiujesz wskaźniki, umieść znak „*” zaraz za nazwą typu. Możesz to robić w bezpieczny sposób pod warunkiem, że w każdym wierszu definiujesz **tylko jedną zmienną**. Taki styl wydaje się mniej mylący dla osoby czytającej kod.
70. Zapewnij, by inicjalizacja dotyczyła wszystkich aspektów twojego kodu. Dokonuj inicjalizacji wszystkich składowych na liście inicjatorów konstruktora, nawet w przypadku typów wbudowanych (używając wywołań pseudokonstruktorów). Wykorzystywanie listy inicjatorów konstruktora jest często znacznie bardziej efektywne w przypadku inicjalizacji obiektów podrzędnych — w przeciwnym razie wywoływany jest bowiem domyślny konstruktor, co prowadzi do wywołania innych funkcji składowych (przypuszczalnie operatora =) w ramach przygotowań do wymaganej przez ciebie inicjalizacji.
71. Nie używaj definicji obiektu w postaci **MojTyp a = b;**. Zapis taki jest głównym źródłem nieporozumień, ponieważ powoduje wywołanie konstruktora, a nie operatora =. Dlążej używaj zamiast tego zawsze precyzyjnej postaci definicji: **MojTyp a(b);**. Rezultat jest taki sam, ale nie wprowadzisz w błąd innych programistów.
72. Używaj jawnego rzutowania, opisanego w rozdziale 3. Rzutowanie ignoruje normalny system typów, będąc miejscem potencjalnych błędów. Ponieważ jawnie rzutowania dzielądysponowane w języku C rzutowanie na kategorie dobrze oznaczonych rzutowań, każdy, kto uruchamia i utrzymuje kod, może w łatwy sposób odnaleźć wszystkie miejsca, w których wystąpienie błędów logicznych jest najbardziej prawdopodobne.
73. Aby program był niezawodny, pewny musi być każdego element. Wykorzystuj wszystkie narzędzia udostępniane przez język C++, m.in. kontrolę dostępu, wyjątki, poprawność stosowania stałych, sprawdzanie typów, w każdej tworzonej przez siebie klasie. Dzięki temu tworząc system, w bezpieczny sposób przeniesiesz się na następny poziom abstrakcji.
74. Wykorzystuj poprawność stosowania stałych. Umożliwi to kompilatorowi wskazanie błędów, które w innym przypadku byłyby trudno uchwytne i niełatwe do odnalezienia. Praktyka ta wymaga pewnej dyscypliny i musi być stosowana konsekwentnie we wszystkich klasach, ale jest opłacalna.
75. Wykorzystuj kontrolę błędów zapewianą przez kompilator. Dokonuj wszystkich komplikacji z włączonymi wszystkimi możliwymi ostrzeżeniami i poprawiaj swój kod, by je wszystkie usunąć. Pisz kod, który wykorzystuje raczej błędy i ostrzeżenia czasu komplikacji niż czasu wykonania programu (na przykład nie używaj zmiennej liczby argumentów wywołań funkcji,

które powodują zablokowanie wszelkiej kontroli typów). Do uruchamiania programu używaj makroinstrukcji **assert()**, ale w stosunku do błędów czasu wykonania wykorzystuj wyjątki.

76. Przedkładaj błędy czasu komplikacji nad błędy czasu wykonania. Próbuj obsługiwać błędy możliwie jak najbliżej miejsc ich powstania. Staraj się raczej poradzić sobie z nimi na miejscu niż zgłaszać wyjątek. Wyłapuj wszystkie wyjątki w najbliższej procedurze obsługi, która posiada dostateczną liczbę informacji, by sobie z nimi poradzić. Zrób **wszystko**, co w twojej mocy, z wyjątkiem zgłoszonym na bieżącym poziomie — **jeżeli** nie rozwiąże to problemu, zgłoś go ponownie (więcej szczegółów na ten temat można znaleźć w drugim tomie książki).
77. Jeżeli wykorzystujesz specyfikację wyjątków (informacje na ten temat znajdują się w drugim tomie książki, który można pobrać z witryny <http://helion.pl/online/thinking/index.html>), instaluj własną funkcję **unexpected()**, wykorzystując do tego celu funkcję **set_unexpected()**. Funkcja **unexpected()** powinna rejestrować błąd, zgłaszając ponownie wyjątek. Dzięki temu w przypadku, gdy istniejące funkcje zostaną zasłonięte i zacząć zgłaszać wyjątki, będziesz dysponował „nagraniem podejrzaneego”, co pozwoli na modyfikację wywołującego kodu w celu obsługi wyjątku.
78. Utwórz zdefiniowaną samodzielnie funkcję **terminate()** (sygnalizującą błąd programisty), która rejestruje błąd, będący przyczyną wyjątku, a następnie zwalnia zasoby i powoduje zakończenie pracy programu.
79. Jeżeli destruktor wywołuje jakiekolwiek funkcje, to mogą one zgłosić wyjątki. Destruktory nie mogą zgłaszać wyjątków (może to skutkować wywołaniem funkcji **terminate()**, oznaczającej błąd programowy), w związku z czym każdy destruktor, który wywołuje funkcje, musi wyłapywać i obsługiwać własne wyjątki.
80. Nie twórz własnych „uzupełnień” prywatnych danych składowych (poprzedzających podkreślenie, notacji węgierskiej itp.), chyba że masz do czynienia z mnóstwem istniejących już wcześniej wartości globalnych — w przeciwnym razie pozwól wyznaczyć zasięgi klasom i przestrzeniom nazw.
81. Przyjrzyj się przeciążeniom. Funkcje nie powinny warunkowo wykonywać kodu na podstawie wartości swoich argumentów, niezależnie od tego, czy są one domyślne, czy też nie. W takim przypadku należy, **zamiast jednej** funkcji, utworzyć dwie lub większą liczbę przeciążonych funkcji.
82. Ukryj wskaźniki wewnętrz klas kontenerowych. Pobieraj je tylko wówczas, gdy zamierzasz wykonywać operacje bezpośrednio na nich. Wskaźniki zawsze stanowią źródło licznych błędów. Gdy używasz operatora new, spróbuj umieścić zwrócony wskaźnik w kontenerze. Preferuj rozwiązanie, w którym kontener jest „właścicielem” wskaźników, więc odpowiada również za ich sprzątanie. Jeszcze lepiej będzie, gdy „opakujesz” wskaźnik w klasę — **jeżeli** chcesz, aby nadal wyglądał jak wskaźnik, przeciąż **operator->** oraz **operator***. Jeżeli jest ci potrzebny samodzielny wskaźnik, zawsze go zainicjalizuj — najlepiej adresem obiektu, ale **wartością zerową**, gdy jest to konieczne. Przypisz mu wartość **zerową**, gdy usuwasz wskazywany przez niego obszar pamięci, by zapobiec wielokrotnemu powtarzaniu tej czynności.

83. Nie przeciążaj globalnych operatorów **new** i **delete** — zawsze rób to w stosunku do klasy. Przeciążenie globalnych wersji operatorów wpłynie na cały projekt **klienta-programisty** — coś, nad czym powinni mieć kontrolę wyłącznie twórcy projektu. Gdy przeciążasz operatory **new** i **delete** w stosunku do klas, nie zakładaj, że znasz wielkość obiektu — ktoś mógł bowiem utworzyć klasę pochodną. Wykorzystaj dostarczony argument. Jeżeli robisz coś szczególnego, zastanów się, jaki będzie to miało wpływ na klasy pochodne.
84. Zapobiegaj okrajaniu obiektów. Rzutowanie w górę obiektów przez wartość właściwie nigdy nie ma sensu. Aby zapobiec rzutowaniu w górę przez wartość, umieść w swojej klasie podstawowej funkcje czysto wirtualne.
85. Czasami wystarczy prosta agregacja. „System zapewnienia wygody pasażerowi” linii lotniczej składa się z rozłącznych elementów: fotela, klimatyzacji, sprzętu wideo itd., a w samolocie musisz utworzyć wiele takich systemów. Czy będziesz tworzył prywatne składowe i zbudujesz dla nich zupełnie nowy interfejs? Nie — w tym przypadku poszczególne elementy są również składnikami publicznego interfejsu, należy zatem uczynić je publicznymi obiektami składowymi. Obiekty te posiadają swoje implementacje, które nadal pozostają bezpieczne. Pamiętaj, że taka prosta agregacja nie jest często stosowanym rozwiązaniem, ale się zdarza.

Dodatek C

Zalecana literatura

Dodatkowe źródła informacji

Język C

„Thinking in C: Foundations for Java & C++”, Chuck Allison (MindView, Inc., 2000, kurs na CD-ROM-ie, dostępny również w witrynie www.BruceEckel.com), Jest to niezbyt wyczerpujący kurs przygotowujący do nauki języków Java oraz C++; zawiera wykład oraz slajdy dotyczące podstaw języka C. Zawiera jedynie informacje niezbędne do rozpoczęcia nauki innych języków. Dodatkowe podrozdziały, dotyczące konkretnych języków, wprowadzają cechy języków C++ i Java i są przeznaczone dla osób zamierzających programować w tych językach. Zalecane przygotowanie czytelnika: pewne doświadczenie w programowaniu w językach wysokiego poziomu, takich jak Pascal, BASIC, Fortran albo LISP (możliwe jest przebrnięcie przez ten CD-ROM bez takiego doświadczenia, ale kurs nie został pomyślany jako wprowadzenie do podstaw programowania).

Ogólnie o języku C++

„The C++ Programming Language”, wydanie trzecie, Bjarne Stroustrup, Addison-Wesley, 1997. W pewnym stopniu celem mojej książki było przybliżenie czytelnikowi możliwości posłużenia się książką Bjarna jako informatorem. Ponieważ książka ta stanowi opis języka, napisany przez autora tego języka, to na ogół sięga się po nią, by wyjaśnić jakieś wątpliwości dotyczące oczekiwów związanych z językiem C++. Jeżeli nabrałeś już pewnej wprawy w używaniu tego języka i zamierzasz zająć się nim poważnie, to będzie ci ona potrzebna.

„C++ Primer”, wydanie trzecie, Stanley Lippman i Josee Lajoie, Addison-Wesley, 1998. Nie jest to bynajmniej książka dotycząca podstaw języka — liczy wiele stron

i zawiera mnóstwo szczegółów. Zawsze po nią sięgam (oprócz książki Stroustrup'a), gdy próbuję rozstrzygnąć jakąś kwestię. „Thinking in C++” powinna dostarczyć podstaw do zrozumienia zarówno „C++ Primer”, jak i książki Stroustrup'a.

„**C & C++ Code Capsules**”, Chuck Allison, Prentice-Hall, 1998. Autor zakłada, że **znasz już języki C i C++**; przedstawia on pewne kwestie, które mogłeś zaniedbać albo których mogłeś nie zrozumieć zbyt dobrze za pierwszym razem. Dzięki książce można wypełnić luki w znajomości zarówno języka C, jak i C++.

„**The C++ Standard**”. Jest to dokument, nad którym komitet standaryzacyjny tak ciężko pracował przez te wszystkie lata. Niestety, nie jest on dostępny bezpłatnie. Można go jednak nabyć w postaci dokumentu elektronicznego w formacie PDF za jedynie 18 USD — w witrynie www.cssinfo.com.

Książki, które napisałem

Książki wymieniono w kolejności publikacji. Nie wszystkie spośród wymienionych pozycji są obecnie dostępne.

„**Computer Interfacing with Pascal & C**”, wydana nakładem własnym pod egidą wydawnictwa **Eisys**, 1988 (książka jest dostępna wyłącznie za pośrednictwem witryny www.BruceEckel.com). Wprowadzenie do elektroniki z czasów, gdy wciąż królował **CP/M**, a DOS był **parwieniuszem**. Do realizacji różnych projektów elektronicznych używałem języków wysokiego poziomu, a nieradko również portu równoległego mojego komputera. Książka stanowi adaptację moich artykułów, zamieszczanych w pierwszym i zarazem najlepszym czasopiśmie, z którym współpracowałem — *Micro Cornucopia* (parafrując Larry'ego O'Briena, wieloletniego wydawcę *Software Development Magazine*, było to najlepsze, wydawane kiedykolwiek czasopismo na temat komputerów — snuli nawet plany zbudowania robota w doniczce!). Niestety, Micro C przestało istnieć na długo, zanim pojawił się Internet. Książka ta była dla mnie wyjątkowo satysfakcyjnym doświadczeniem wydawniczym.

„**Using C++**”, Osborne/McGraw-Hill, 1989. Jedna z pierwszych **książek** na temat języka C++. Nakład wyczerpany — ukazało się drugie wydanie, pod zmienionym tytułem „**C++ Inside & Out**”.

„**C++ Inside & Out**”, Osborne/McGraw-Hill, 1993. Jak już wspomniałem, jest to w rzeczywistości drugie wydanie książki „**Using C++**”. Zawarty w tej książce opis języka C++ jest w miarę dokładny, ale pochodzi on mniej więcej z roku 1992 i celem „Thinking in C++” było zastąpienie tej książki. Więcej na jej temat można się dowieść w witrynie www.BruceEckel.com (dostępne są tam również kody źródłowe).

„**Thinking in C++**”, wydanie pierwsze, Prentice-Hall, 1995.

„**Black Belt C++, the Master's Collection**”, pod redakcją **Bruce'a Eckela**, M&T Books, 1994. Nakład wyczerpany. Zbiór rozdziałów, napisanych przez różne znakomitości związane z językiem C++, przygotowany na podstawie ich prezentacji na Software Development Conference, dokonanej w grupie tematycznej C++, której prowadniczyłem. Widok okładki tej **książki** skłonił mnie do przejęcia kontroli nad wszystkimi przyszłymi projektami okładek.

„Thinking in Java”, wydanie drugie, Prentice-Hall, 2000¹. Pierwsze wydanie tej książki otrzymało w 1999 roku nagrody: Productivity Award, przyznaną przez *Software Development Magazine*, oraz Editor's Choice Award, wręczaną przez *Java Developer's Journal*. Książkę można pobrać z witryny www.BruceEckel.com.

Głębia i mroczne zauleki

Książki te umożliwiają pogłębienie wiedzy o języku, pomagając w uniknięciu typowych pułapek, towarzyszących nieodłącznie tworzeniu programów w języku C++.

„Effective C++” wydanie drugie, Scott Meyers, Addison-Wesley 1998 oraz „More Effective C++”, Scott Meyers, Addison-Wesley, 1996. Klasyczne, obowiązkowe teksty, dotyczące rozwiązywania poważnych problemów i projektowania kodu w języku C++. W książce „Thinking in C++” próbowałem przejąć i opisać wiele pojęć zawartych w tych książkach, ale niejestem tak naiwny, by sądzić, że mi się to udało. Jeżeli spędzasz odpowiednio dużo czasu z językiem C++, to z pewnością sięgniesz po te pozycje. Dostępne są one również na płytach CD-ROM.

„Ruminations on C++”, Andrew Koenig i Barbara Moo, Addison-Wesley, 1996. Andrew współpracował ze **Stroustrupem**, zajmując się wieloma kwestiami związanymi z językiem C++, i zyskał status autorytetu w tej dziedzinie. Odkryłem również, że precyzjajego wskazówek jest inspirująca. Przez lata wiele się od niego nauczyłem — zarówno dzięki lekturze książki, jak i osobistym kontaktom.

„Large-Scale C++ Software Design”, John Lakos, Addison-Wesley, 1996. Omawia problemy i odpowiada na pytania, które napotkasz tworząc duże projekty, a często również nieco mniejsze.

„C++ Gems”, pod redakcją Stana Lippmana, publikacje SIGS, 1996. Wybór artykułów z *The C++ Report*.

„The Design & Evolution of C++”, Bjarne Stroustrup, Addison-Wesley, 1994. Informacje pochodzące od twórcy języka C++, dotyczące przyczyn podjęcia przez niego różnych decyzji projektowych. Nie jest wprawdzie niezbędna, ale warto ją przeczytać.

Analiza i projektowanie

„Extreme Programming Explained”, Kent Beck, Addison-Wesley, 2000. Uwielbiam tę książkę. Owszem, stosuję na ogół radykalne metody, ale zawsze przeczuwałem, że mógłby istnieć jakiś zupełnie inny, znacznie lepszy proces tworzenia oprogramowania i myślę, że programowanie ekstremalne jest bliższe tej idei. Jedyną książką, która wywarła na mnie podobny wpływ, była „PeopleWare” (omówiona po-

¹ Przetłumaczona książka została wydana w 2001 r. nakładem Wydawnictwo Helion pt. „Thinking in Java. Edycja polska”.

niżej). Opisywała przede wszystkim środowisko i poruszała zagadnienia kultury organizacyjnej. „**Extreme Programming Explained**” dotyczy programowania, odrzucając większość przyjętych rozwiązań, nawet najnowsze „wynalazki” w tej dziedzinie. Autor posuwa się nawet do twierdzenia, iż diagramy są w porządku, pod warunkiem, że nie spędza się nad nimi zbyt wiele czasu i zamierza się je wszystkie wyrzucić (książka *nie ma na okładce „aprobującej pieczęci UML”*). Mógłbym zdecydować się na pracę w firmie wyłącznie na podstawie tego, czy stosowane jest w niej programowanie ekstremalne. Niewielka książka, zawierająca krótkie rozdziały, łatwa w lekturze i eksytująca, gdy się o niej rozmyśla. Zaczynasz wyobrażać sobie pracę w takiej atmosferze i pojawia się wizja zupełnie innego świata.

„**UML Distilled**”, Martin Fowler, wydanie drugie, Addison-Wesley, 2000. Przy pierwszym kontakcie język UML jest zniechęcający, ponieważ jest w nim tyle diagramów i szczegółów. Według Fowlera, **większość** z tych elementów jest niepotrzebna, więc przechodzi on od razu do istoty. W przypadku większości projektów wystarczy tylko znajomość kilku narzędzi, umożliwiających tworzenie diagramów. Fowler chce uzyskać dobry projekt, nie przejmując się wszystkimi elementami, które mają to umożliwić. Cienka, przyjemna w lekturze książka — pierwsza, którą należy kupić, aby zrozumieć UML.

„**The Unified Software Development Process**”, Ivar Jacobson, Grady Booch, i James Rumbaugh, Addison-Wesley, 1999. Spodziewałem się, że książka nie będzie mi się podobać. Wyglądał jak nudne szkolne podręczniki. Byłem mile zaskoczony — tylko niektóre fragmenty zawierały wyjaśnienia, które sprawiały wrażenie, jakby nie były zrozumiałe dla autorów. Przeważająca część książki jest nie tylko przejrzysta, ale również przyjemna w lekturze. A co najważniejsze, opisany w niej proces ma wymiar praktyczny. Nie jest to programowanie ekstremalne (i nie cechuje go jego jasność w sprawie testów), ale są tu silnie zaznaczone elementy UML. Nawet gdy nie możesz zaadaptować programowania ekstremalnego, to większość dołączyła już do jego zwolenników, zjednoczonych pod hasłem „UML jest **dobry**” (niezależnie od ich *rze* *czwistego* z nim doświadczenia), więc również ty prawdopodobnie będziesz w stanie go przyjąć. Sądę, że książka ta mógłaby być przewodnikiem po języku UML, czyli **tą**, którą należy przeczytać po „**UML Distilled**” Fowlera, aby poznać więcej szczegółów.

Zanim wybierzesz którykolwiek z metod, warto poznać punkt widzenia kogoś, kto nie próbuje nam żadnej z nich sprzedać. Łatwo jest zastosować metodę, nie rozumiejąc tak naprawdę, czego się od niej oczekuje i w jaki sposób będzie pomocna. Przekonujące wydaje się to, że inni jej używają. Jednakże ludzie kierują się osobliwym przesądem — jeśli chcą wierzyć, że coś rozwiąże ich problemy, to będą tego próbować (eksperymentowanie jest godne pochwały). Jeżeli jednak nie doprowadzi to do rozwiązania problemu, zdwoją wysiłki, ogłaszaając wszystkim swoje cudowne odkrycie (zaprzeczenie — i to już nie jest dobre). Czynione tu założenie może polegać na tym, że gdy inni płyną wraz z tobą tą samą **łodzią**, nie czujesz się samotny, nawet jeżeli ta łódź zmierza donikąd (albo tonie).

Nie zamierzam sugerować, że wszystkie metodyki wiodą donikąd; należy jednak uzbroić się po zęby w „narzędzia psychiczne”, które pozwalają ci pozostać w fazie eksperymentowania („to nie działa — spróbujmy czegoś innego”), nie przechodząc w fazę zaprzeczenia („nie, to naprawdę nie jest problem — wszystko w porządku, nie musimy niczego zmieniać”). Myślę, że wymienione poniżej książki, przeczytane *zanim wybierzesz jakąś metodę, dostarczą ci odpowiednich narzędzi*.

„**Software Creativity**”, Robert Glass, Prentice-Hall, 1995. Najlepsza ze znanych książek, omawiająca perspektywy metodyk. Jest to zbiór krótkich esejów i artykułów, które Glass napisał lub otrzymał (jednym z autorów jest P.J. Plauger). Są one rezultatem wielu lat rozmyślań i studiów poświęconych temu tematowi. Autor nie zbacza z tematu ani nie zanudza czytelnika, lecz w atrakcyjnej formie przedstawiajedynie to, co niezbędne. W książce nie brak również konkretów — zawiera setki odnośników do innych artykułów i książek. Wszyscy programiści i menedżerowie powinni ją przeczytać, nim ugrzenną w bagnie metodyk.

„**Software Runaways: Monumental Software Disasters**”, Robert Glass, Prentice-Hall, 1997. Książka szczęśliwie wydobywa na światło dzienne to, o czym się zazwyczaj nie wspomina — o projektach, które upadają, i to w sposób spektakularny. Myślę, że większość z nas wciąż myśli: „mnie się to nie może zdarzyć” (albo „to się nie może powtórzyć”) i sądzę, że nie jest to korzystne. Pamiętając, że wszystko może się zawalić, masz dużo lepszy punkt wyjścia do tego, aby sprawić, by wszystko poszło dobrze.

„**Object Lessons**”, Tom Love, SIGS Books, 1993. Jeszcze jedna dobra „perspektywiczna” książka.

„**Peopleware**”, Tom DeMarco i Timothy Lister, wydanie drugie, Dorset House, 1999. Mimo że autorzy mają przygotowanie w zakresie tworzenia oprogramowania, książka dotyczy projektów i zespołów programistycznych. Skupia się jednak bardziej na ludziach i ich potrzebach niż na technologii i jej wymogach. Opowiada o kształtowaniu środowiska, w którym ludzie będą szczęśliwi i produktywni, a nie o tym, jakie powinni spełniać warunki, by być sprawnymi trybami w maszynie. Ten ostatni pogląd odnosi się, jak sądzę, do programistów uśmiechających się i przytakujących podczas wprowadzania w firmie metody XYS, a potem robiących po cichu to samo, co wcześniej.

„**Complexity**”, M. Mitchell Waldrop, Simon & Schuster, 1992. Książka jest kroniką spotkań grupy naukowców reprezentujących różne dyscypliny, które odbyły się w Santa Fe, w Nowym Meksyku. Uczestnicy dyskutowali o problemach, których nie potrafią rozwiązać poszczególne dyscypliny nauki (rynek giełdowy w ekonomii, powstawanie życia w biologii, przyczyny zachowania się ludzi w socjologii itd.). W toku spotkań wypracowali oni interdyscyplinarne ujęcie tych problemów, obejmujące wiedzę z dziedziny fizyki, ekonomii, chemii, matematyki, informatyki, socjologii i innych nauk. A co ważniejsze, powstają nowe sposoby myślenia o takich niezwykle złożonych problemach — dalekie od matematycznego determinizmu oraz iluzji, że napiszemy równanie, dzięki któremu zdołamy przewidzieć wszelkie zachowania. Są one nastawione na obserwację i szukanie wzorców, a następnie próby ich naśladowania wszelkimi dostępnymi środkami (książka opisuje, między innymi, genezę algorytmów genetycznych). Wierzę, że taki sposób myślenia jest użyteczny, w miarę jak powstają metody zarządzania coraz bardziej skomplikowanymi projektami programistycznymi.

Skorowidz

#define, 127, 160, 200, 267, 271
#endif, 200
#ifdef, 160, 200, 258
#include, 74, 98
#undef, 160

A

abort(), funkcja, 326
abstrakcja, 26
 danych, 179
abstrakcyjne
 klasy podstawowe, 518
 typy danych, 28, 108, 195
access specifiers, 30, 212
ADA, 557
adres, 112
 funkcji, 163
 obiektu, 312
 powrotny, 366
 wirtualnych funkcji, 511
aggregate, 242
aggregate initialization, 242
agregacja, 31
agregat, 89, 242, 270
Algol, 19, 359
algorytm, 45, 593
 ogólny, 593
Allison, Chuck, 621, 622
alternate linkage specification, 352
analiza, 42
 składniowa, 70
 wymagań, 45
and, 144
and_eq, 144
anoniemowe argumenty, 96
anonymous union, 257
APL, 26
argc, 155, 156

argumenty, 71
 domyślne, 249, 258
konstruktora, 325
przekazywanie przez
 referencję, 117
 przez wartość, 115
wierszpoleczeń, 155
argumenty-wypełniacze, 259
argv, 154
argv[0], 155
arytmetyka wskaźników, 157
ASCI, 81, 307
assembler, 70, 297
asm, 143
assert(), 162, 183, 239, 316
assure(), funkcja, 607
atexit(), funkcja, 326
ATM, 47
atof(), funkcja, 155
atoi(), funkcja, 155
atol(), funkcja, 155
auto, 330
automatic counting, 243
automatyczna konwersja typów, 425
automatyczne zliczanie, 243

B

bad_alloc, 453, 457
bajt, 109
BASIC, 26, 61
BCD, 109
Beck, Kent, 623
bezpieczeństwo, 59, 229
 stałe, 269
biblioteki, 30, 60, 69, 70, 76, 180
 funkcji języka C, 98
iostream, 346
klas kontenerowych, 20

biblioteki
 komercyjne, 181
 napisane w czystym C, 78
 plik nagłówkowy, 76
 program zarządzający, 99
 SGI STL, 88
 standardowe, 77
 tworzenie, 99
 wątków POSIX, 78
binary-coded decimal, 109
binding, 506
bit, 109
bitand, 144
bitor, 144
blob, 287
blok
 dostępu, 219
 pamięci, 231
błędny, 41, 179, 229, 600
 projektowe, 65
Bloch, Grady, 624
bool, 110, 349
brak pamięci, 451
break, 103

C

C, 13, 25, 59, 95, 517
 funkcje, 95
C++, 13, 58, 319, 348, 382, 438, 517
 standardowy format dołączania plików, 75
C99, 234
CAD, 437
CALL, 297, 366, 514
calloc(), funkcja, 183, 439, 441
case, 104
cassert, 163
cast, 83
catch, 457
cerr, 346, 455
cfloat, 109
cfront, 194
char, 83, 109, 249
char*, 154
chroniony dostęp, 212
ciąg
 Fibonacciego, 579
 znaków, 85
cin, 84, 346, 455
class, 27, 32, 144, 220, 330
Class-Responsibility-Collaboration, 49
clib.h, 186
client programmers, 30
climits, 109

compl, 144
Composite, 376
composition, 467
console input, 84
const, 19, 127, 267, 270, 277, 288, 451, 546
const char*, 280
const correctness, 293
const_cast, 139
constant folding, 268
Constantine Larry, 57
constraint-based programming, 26
constructor initializer list, 471
continue, 103
copy-constructor, 344, 370
copy-on-write, 420
cout, 78, 86, 91, 346, 419, 455
cpp, 172
CRC, 49
cstdio, 300
cstdlib, 84
cstring, 218, 429
cykl projektowy, 53
czyste zastępowanie, 35
czysto wirtualne destruktory, 535
czytelność kodu, 69, 602

D

dane statyczne programu, 324
data, 204
debugger, 159
default, 104
default constructor, 245
definicja, 71
deklaracja, 71, 191
 funkcji, 191
 using, 335
dekrementacja, 393
delete, 20, 40, 41, 136, 183, 184, 241, 441, 534
 przeciążanie, 452
 globalnych operatorów, 453
delete this, 422
Demarco, Tom, 625
dereference, 115
destruktor, 19, 232, 255, 348, 423, 432, 462,
 533, 564
 automatyczne wywołania, 474
 czysto wirtualny, 535
 funkcje inline, 313
 kolejność wywoływanego, 474
 obiektów statycznych, 326
 wirtualne wywołania, 537
 wirtualny, 533, 565

diagram UML, 29, 32
diagramy przypadków, 46
długość słowa, 112
dobry styl programowania, 233
dokumentacja, 198
dołączanie
 nagłówków, 74
 plików, 199
domyślne argumenty, 250, 258
domyślnie prywatne, 221
domyślny
 konstruktor, 482
 kopiujący, 374
DOS, 78, 84
double, 109, 156
do-while, 99, 101
drukowanie, 263
drzewo składniowe, 70
dynamie binding, 506
dynamic_cast, 139, 544, 545
dynamiczna
 alokacja pamięci, 437
 kontrola typów, 70
dynamiczne tworzenie obiektów, 437
dynamiczny przydział pamięci, 183
dyrektywy preprocesora, 69
 #define, 127, 160, 200
 #endif, 200
 #ifndef, 160, 200
 #include, 74
 #undef, 160
 DEBUG, 160
 NDEBUG, 160
dziedziczenie, 20, 32, 283, 467, 484, 497, 613
 chronione, 489
 diagram dziedziczenia, 494
 łączenie kompozycji, 473
 obniżenie poziomu dostępu, 470
 ograniczenia, 612
 poziomy, 510
 protected, 488
 prywatne, 487
 przeciążanie operatorów, 490
 rzutowanie w górę, 492
 składnia, 469
 statyczne funkcje składowe, 483
 tablica VTABLE, 523
 upublicznianie składowych, 488
wielokrotne, 491, 540

E

early binding, 37, 506
efektywność, 60
egzemplarz klasy, 27

Ellis, Margaret, 345
else, 100
encapsulation, 195
end sentinel, 579
endl, 81, 91
enum, 148, 256, 330
escape sequences, 81
etapy projektowania obiektów, 50
 konstrukcja systemu, 51
 rozbudowa systemu, 51
 składanie obiektów, 51
 wielokrotne wykorzystywanie obiektów, 51
 znajdowanie obiektów, 51
ewolucja, 53
exception
 handler, 41, 457
 handling, 41
exit(), funkcja, 156, 326
explicit, 426, 427
extern, 73, 123, 126, 268, 271, 329, 352
extreme programming, 55, 492

F

fabryka, 569
factory, 569
false, 87, 100, 110
fałsz, 87
fan-out, 431
float, 109, 156, 187, 249
float.h, 109
for, 90, 99, 101, 102, 121, 235
 inicjalizacja, 102
 krok, 102
 warunek, 102
Fortran, 26, 412
Fowler, Martin, 624
fragmentacja
 pamięci, 184
 sterty, 453
free(), funkcja, 15, 183, 437, 439, 554
friend, 214, 216, 218, 332, 442
fstream, 86, 345
function prototyping, 96
fundament intelektualny, 58
funkcja
 abort(), 326
 assure(), 607
 atexit(), 326
 atof(), 155
 atoi(), 155
 atol(), 155
 calloc(), 183, 439, 441
 definicja, 73
 exit(), 156, 326

- funkcja
free(), 15, 183, 437, 439, 554
getline(), 86
longjmp(), 15
main(), 63, 80, 161, 302
malloc(), 15, 183, 437, 439, 440, 441, 446, 554
memcpy(), 261, 447
memset(), 218, 284, 447
print(), 154, 302
printf(), 198, 617
push_back(), 89
push_front(), 89
puts(), 455
realloc(), 183, 439, 441
require(), 204, 325, 607
requireArgs(), 318
requireMinArgs(), 318
setjmp(), 15, 232
strcmp(), 429
system(), 84
funkcje, 71, 187, 380
adres, 163
anomimowe argumenty, 96
argument definicji, 96
argumenty, 71
 - domyślne, 249
 - wypełniacze, 259
czysto **wirtualne**, 518, 527
definicja, 71
definicje wskaźnika, 163
deklaracja, 71, 72, 74, 301
dynamicznego przydziału pamięci, 440
globalne, 388
inline, 19, 297, 301, 308, 328, 352, 531, 564
in situ
modyfikujące obiekty zewnętrzne, 377
nie dziedziczone automatycznie, 480
obsługi operatora new, 451
pobieranie argumentu przez referencję, 278
prototyp, 96
przeciążanie, 59
 - na podstawie zwracanych wartości, 251
 - nazw, 249
przekazywanie
 - adresów, 279
 - argumentów, 363
 - stałej przez wartość, 275
pusta lista argumentów, 72
ramka stosu, 365
referencja, 361
składnia deklaracji, 72
składowe, 29, 191, 256, 305, 429, 468
standardowy sposób przekazywania argumentów, 281
statyczne obiekty klas, 325
szablony, 593
tworzenie, 95
udostępniające, 303
virtual, 506
wartość zwracana, 71, 97
wirtualne, 20, 38, 476, 498, 503, 504, 517
zgodne z POSIX, 78
zmiana typu zwracanej wartości, 529
zmienna lista argumentów, 97, 198
zmienne statyczne, 324
zwracanie
 - adresów, 279
 - przez wartość, 364
 - stałej przez wartość, 276

G

- g++**, 82
garbage collector, 41
generator kodu, 70
generic algorithm, 593
getline(), funkcja, 86
getval, 134
Glass, Robert, 625
global optimizer, 70
globalna przestrzeń nazw, 202, 330
globalnyzasięg, 19
GNU C++, 82, 167
GNU Emacs, 604
goto, 105, 232
 - dalekie, 232.

H

- handle classes, 224
header file, 74
heap, 20, 40, 183
heurystyka, 42
hierarchia
 - bazująca na obiekcie, 538, 555
 - dziedziczenia, 543
 - klas, 532, 546
 - o jednym korzeniu, 538
 - typów, 33
 - wywołań konstruktorów, 532**hybrydowy język obiektowy**, 17

I

- identyfikacja typów podczas pracy programu, 524, 545
identyfikator, 71, 180, 599
ffIEE, 109, 156

- if, 98
if-else, 99, 136
ifstream, 86, 485
iloczyn logiczny, 131
implementacja, 29, 31, 197
 języka, 23
include **guards**, 607
indeksowanie zerowe, 151
inheritance, 32, 467
inicjalizacja, 229, 367, 471
 agregatowa, 166, 242
 obiektów
 składowych, 471
 stałych, 325, 344
 wskaźników do składowych, 382
 za pośrednictwem elementów składowych, 376
inkrementacja, 90, 393
inline, 297, 301, 302, 312, 314, 558, 614
input-output stream, 78
instalacja wskaźnika wirtualnego, 515
int, 97, 148, 187, 249
inteligentny wskaźnik, 406
interfejs, 27, 29, 44
 klasy, 505
 użytkownika, 45
interpreter, 68
 BASIC, 68
interrupt service routine, 366
iostream, 75, 77, 78, 86, 199, 346
iostream.h, 75
ISO, 22
ISR, 366
iteracja, 53
iterator, 406, 551, 575, 579, 582, 590
 zaginębiony, 408, 582
- J**
- Jacobsen, Ivar, 624
Java, 15, 235, 470, 517, 555, 604
 oficjalny standard kodowania, 604
jawne rzutowanie, 139, 543
jadro, 52
jednostka translacji, 187, 323, 328, 344
język
 APL, 26
 asemblera, 26, 68
 BASIC, 26
 C, 13, 26, 59
 C++, 13
 C99, 234
 Fortran, 26
 imperatywny, 26
 interpretowany, 68
- Java, 15, 64
LISP, 26
o słabej kontroli typów, 38
obiektowy, 58
proceduralny, 58
programowania wieloparadygmatowego
PROLOG, 26
Python, 66
Smalltalk, 27, 555
UML, 29, 50
wysokiego poziomu, 15
- K**
- kapsułkowanie, 195, 219, 229, 503
karty CRC, 50
klasa-obowiązek-współpraca, 49
klasy, 27, 28, 60, 67, 144, 219, 282, 504
 abstrakcyjne, 518
 adresy wirtualnych funkcji, 511
 bazowe, 32
 const, 282
 czysto abstrakcyjne, 520
 definicja, 218, 225
 deklaracja, 225
 destruktor, 232, 237
 dziedziczące, 31
 dziedziczenie, 32, 34, 467
 funkcje
 czysto wirtualne, 518
 inline, 302
 składowe, 469
 udostępniające, 303
 wirtualne, 476, 507
ifstream, 485
interfejs, 56, 508
iterator, 577
kapsułkowanie, 195
konstruktor, 230, 237
 kopiący, 376
kontenerowe, 556
kontroladostępu, 30, 219
lista inicjatorów konstruktora, 283
lokalne, 341
modularyzacja, 510
modyfikatory, 304
nadzędne, 32
nazwa, 49
obiektów, 28
obserwatory, 304
ofstream, 475
okrojenie, 526
ostream, 372

- klasy**
pochodne, 32, 507
podzielne, 32
podstawowe, 32, 36, 256, 469, 591
polimorfizm, 478
potomne, 32
przechowywanie informacji o typie, 511
przeciążanie, 527
 delete, 455
 new, 455
przedefiniowanie, 476
public, 470
rzutowanie, 494
set, 568
składniki, 28
składowe, 30, 34, 220
stałe, 282
 o wartościach określonych podczas komplikacji, 285
statyczne
 funkcje składowe, 342, 483
 obiekty, 325
string, 67, 85, 161, 429
stringstream, 414
strumieni wejścia-wyjścia, 78
tworzenie, 188
ukrywanie nazw, 476
vector, 67, 89
wrażliwa klasa podstawowa, 224
zagnieżdżone, 341
zaprzyjaźnione, 408, 576
zasłanianie, 476, 527
klasy-uchwyty, 223, 224
klient, 46
klient-programista, 30, 61, 211
kod, 16
 assemblera, 143
 błędu, 42
 źródłowy, 68
kodowanie, 13
 ze sprawdzaniem typów, 612
Koenig, Andrew, 300, 623
 kolejność
 dolaczania plików nagłówkowych, 606
 wywoływanie
 destruktorów, 474
 konstruktorów, 474, 531
kolekcja, 406
kolizja nazw, 188
komentarz, 601
 znaczniki, 601
komercyjna biblioteka, 181
Komitet Standardyzacyjny C++, 22
komplikacja, 56, 69, 219
analiza składniowa, 70
drzewo składniowe, 70
generator kodu, 70
kontrola typów, 70
optymalizator, 70
preprocesor, 69
program łączący, 70
przebiegi, 70
rozłączna, 69, 71, 167
w pamięci, 69
komplikator, 23, 37, 56, 68, 70, 81, 167, 187, 194, 382, 467, 482, 493, 514, 557
funkcje inline, 311
 GNUC++, 82
jednostkatranslacji, 187
odwołania do przodu, 313
ograniczenia funkcji inline, 312
komplikowanie kodu, 63
komponenty, 229
kompozycja, 20, 31, 374, 467, 484, 492, 497, 557, 613
 łączenie dziedziczenia, 473
 składnia, 468
kompresja, 184
komunikacja, 49
komunikaty, 27, 34
konformizm, 57
koniunkcja, 133
konkretyzacja, 559
konstruktor, 19, 230, 255, 261, 327, 440, 453
 argumenty, 325
 definicja, 283
 domyślne argumenty, 258
 domyślny, 245, 263
 funkcje
 inline, 313
 wirtualne, 530
 jawne, 426
 kolejność wywoływania, 474, 531
 kopiący, 19, 344, 359, 363, 369, 374, 495, 526
 lista inicjatorów, 283, 471
 typy wbudowane, 283
 wywoływanie funkcji wirtualnych, 532
 zapobieganie konwersji, 426
kontenery, 67, 88, 406, 551, 614
 iterator, 575
 prawa własności, 570
 wskaźników, 565
kontrola
 błędów, 316
 dostępu, 30, 195, 212, 219, 222, 503, 610
 nazw, 59
 typów, 274

dynamiczna, 70
literały napisowe, 275
przypisania wskaźników, 274
statyczna, 70, 224
konwersja typów, 429
 automatyczna, 425
operator, 427
pułapki automatycznej konwersji, 430
ukryte działania, 431
za pomocą konstruktora, 425
zapobieganie konwersji za pomocą
 konstruktora, 426
konwersje zauważające, 141
końcowe porządkи, 229
kopiowanie
 bitów, 367
 przy zapisie, 420
koszty początkowe, 64
kot z Cheshire

L

Lajoie, Josee, 621
Lakos, John, 623
late binding, 38, 506
lazy initialization, 563
leniwa inicjalizacja, 563
librarians, 70
libraries, 70
liczby
 dziesiętne kodowane dwójkowo, 109
 zmiennopozycyjne, 28, 109, 156
limits.h, 109
linker, 69, 70
Linux, 172
Lippman, Stanley, 621
LISP, 26
lista
 inicjatorów konstruktora, 283, 471
 typy wbudowane, 472
 powiązana, 180, 240
Lister, Timothy, 625
long, 111
longjmp(), funkcja, 15
Love, Tom, 625
I-wartość, 130, 133, 277, 558

L

łańcuchowanie, 159, 162
łańcuchy, 81, 85
łączenie, 76, 126, 323
 tablic znakowych, 83
 wewnętrzne, 126
 zewnętrzne, 126, 127, 328

M

main(), funkcja, 63, 80, 161, 302
maintenance, 53
make, 167
 domyślne pliki wynikowe, 170
 reguły przyrostkowe, 169
Make, 167
makefile, 160, 167
.SUFFIXES, 169
all, 170
CPP, 169
 domyślne pliki wynikowe, 170
makrodefinicje, 168
OFLAG, 172
przykładowy plik, 171
makroinstrukcje, 19, 131, 159, 162, 198, 297,
319, 615
dostęp, 300
malloc(), funkcja, 15, 183, 437, 439, 440, 441,
446, 554
manipulator strumieni, 83
mechanizm funkcji wirtualnych, 510
mem, 261
member, 222
memberwise
 assignment, 424
 initialization, 376
memcpy(), funkcja, 261, 447
memset(), funkcja, 218, 284, 447
menedżer sterty, 185
metody, 42
metodyka, 42, 44
mikroprocesor, 41
MindView, Inc., 14
miniaturowabiblioteka, 180
model
 maszyny, 26
 rozwiązywanego problemu, 26
modularyzacja klas, 510
modulo, 130
moduł startowy, 77
modyfikator c-v, 293
modyfikatory, 304
Moo, Barbara, 623
multiparadigm programming languages, 27
multiple
 dispatching, 543
 inheritance, 540
Murray, 414
mutable, 290, 291

N

nadklasa, 32
name decoration, 250
namespace, 79, 323, 333
namespaces, 330
narzędzie profilujące, 65
narzut menedżera pamięci, 442
nawiasy, 602
nazwy, 323
 identyfikatorów, 605
 ograniczanie widoczności, 328
 plików, 600
 widoczność, 323
 zasięg, 333
NDEBUG, 160, 163
negacja, 135
new, 20, 40, 136, 183, 184, 237, 241, 440, 455, 461, 531
 brak pamięci, 451
 przeciążanie, 452
 globalnych operatorów, 453
new handler, 451
newmem, 261
niejawna konwersja typów, 128
niepełna specyfikacja typu, 216, 225
nieukończona rekurencja, 326
niezmienność
 bitowa, 290
 fizyczna, 290
 logiczna, 290
niszczenie obiektów, 326
 statycznych, 326
not, 144
not_eq, 144
notacja UML, 50

O

obiektowe metody projektowania, 25
obiekty, 27, 194, 504
 adres, 312
 delete, 41, 441
 destruktor, 232
 dynamiczne tworzenie, 40, 437
 funkcje składowe, 29
 globalne, 328
 hierarchia, 538
 inicjalizacja, 230
 obiektów składowych, 471
 obiektów statycznych, 344
 interfejs, 27, 44
 kontenery, 406
 new, 40, 440

niszczenie, 40
okrajanie, 525
 podzielne, 468
polimorfizm, 36
projektowanie, 51
 przekazywanie obiektów przez wartość, 359
 składowe, 32
 Stack, 240
 statyczne, 324, 348
 stos, 40
 struktura pamięci, 219
 this, 192
 tworzenie, 40, 234, 438
 tymczasowe, 278, 374
 uzależnione, 345
 wysyłanie komunikatów, 195
 zewnętrzne, 116
Object, 555
object oriented programming, 18, 25
object-based hierarchy, 538
obraz funkcji wirtualnych, 512
obserwatory, 304
obsługa
 błędów, 61
 wyjątków, 14, 41, 457
obszar pamięci, 194
odrzucenie niezmienności, 290
odśmietanie, 594
 odwołania do przodu, 313
OFLAG, 172
ofstream, 86, 87, 327, 475
 ograniczanie powtórznych komplikacji, 224
 ograniczenia semantyczne, 57
 okrajanie obiektu, 521, 525, 527
OOP, 18, 25
OOPS, 555
operacja wyluskania, 115
operator, 387
operators, 107, 129, 438
 !, 135
 #, 162
 &, 113, 132, 136, 298
 &&, 131
 *, 114, 130, 136
 ***this**, 403
 /, 130
 :, 136
 ::, 189
 @, 388
 [], 449
 ^, 132
 |, 132
 ||, 131
 ~, 132

- +⁺, 86, 107, 130
++⁺, 108, 136, 157, 408, 576
<<[<], 79, 82, 134, 419
=⁼, 86, 107, 416
->^{->}, 406
->^{->}, 136, 147
->*^{->*}, 410
>=^{>=}, 299
>>^{>>}, 134
adresu, 136
alternatywy, 132, 138
argumenty, 402
automatyczne tworzenie operatora =, 424
automatycznej
 dekrementacji, 108
 inkrementacji, 108
 konwersji typów, 483
bitowe, 132
dekrementacji, 129, 393
delete^{20,405,441}
dodawania, 107
dosłowne, 144
dwuargumentowe, 388, 393
globalne przeciążone, 427
iloczynu logicznego, 131
indeksowe, 405
inkrementacji, 129, 393
jednoargumentowe, 135, 388, 390
koniunkcji, 132, 138
konwersji, 427
których nie można przeciążać, 412
logiczne, 131
łańcuchowania, 162
matematyczne, 130
mnożenia, 107
modulo, 130
negacji, 132, 135
negacji logicznej, 135
new^{20, 405, 440, 450}
nietypowe, 405
odejmowania, 107
potęgowania, 412
priorytety, 107
przeciążanie, 79, 387, 413, 452
przecinkowe, 137, 405
przedrostkowa wersja, 403
przesunięć, 133, 414
przypisania, 107, 129
relacji, 131
różnicy symetrycznej, 132
rzutowania, 136, 138
sizeof, 111, 143, 196
składnia przeciążania, 388
sumy logicznej, 131
trójargumentowe, 136
tworzenie, 416
umieszczenia
 delete, 461
 new, 461
wirtualne, 541
wyłączania, 136
wyłączania wskaźnika, 406
wywołania funkcji, 410
zaprzyjaźnione, 428
zasiegu, 189, 206, 333, 471
zwracający adres elementu, 113
zwracane wartości, 402
zwracanie
 stałych przez wartość, 404
 wartości przez referencje, 414
optymalizacja, 124, 330, 614
 zwracania wartości, 404, 616
optymalizator
 globalny, 70
 lokalny, 70
or, 144
or_eq, 144
ostream, 263, 372, 616
overloaded, 249
overloading, 79
overriding, 35, 476, 507

P

- pamięć**, 27, 41, 112
 dynamiczna alokacja, 437
 dynamiczny przydział, 183
fragmentacja, 184
narzut menedżera, 442
ROM, 291
stacjonarna, 323
stacjonne składowe, 337
uchwyty, 185
wirtualna, 440
parsing, 70
Pascal, 19
pass by reference, 117
pass by value, 115
peephole optimizer, 70
persistence, 594
pętle
 do-while, 102
 for, 90, 102, 235
 while, 87
pielegnacja, 53
planowanie, 55

- pliki, 71, 78
.a, 99
.cpp, 169
.h, 75
.lib, 99
.o, 76
.obj, 76
makefile, 167, 168
nagłówkowe, 74, 80, 191, 197, 199, 319, 328, 429, 539
nagłówkowe biblioteki, 76
nazwy, 600
odczytywanie, 86
standardowy format dołączania plików, 75
śledzenia, 327
wynikowe, 187
zapisywanie, 86
zasięg, 328
źródłowe, 449
podklasa, 32
podtypy, 485
polimorficzna hierarchia, 544

- wykorzystywanie, 336
- zasięg, 333
- problemu, 26
- rozwiązań, 26
- przesunięcia, 133
- przydzielanie
 - pamięci**, 72, 184, 236, 438
 - wielokrotne, 543
- przyjaciele, 214
- zagnieżdżeni, 216
- przypadki użycia, 46, 53
- przypisanie**, 92, 130, 274
 - automatyczne tworzenie operatora, 424
 - kontrola typów, 274
 - literał napisowe, 275
 - przeciążanie, 415
 - wskaźniki zawarte w klasach, 417
 - za pośrednictwem elementów składowych, 424
 - zliczanie odwołań, 419
- pseudokonstruktor**, 473
- public**, 30, 212, 470
- publiczny, 212
- punkt sekwencyjny, 231, 324
- pure abstract class, 520
- pure virtual function, 518
- push_back(), funkcja, 89
- push_front()**, funkcja, 89
- putc(), 300
- puts(), funkcja, 455
- p-wartość**, 130, 133, 136, 558
- Python**, 66, 68, 517, 561

R

- ramka
 - funkcji, 366
 - stosu, 324
 - wywołanie funkcji, 365
- realloc(), funkcja, 183, 439, 441
- redefining**, 476
- reference counting, 419
- referencja, 19, 59, 117, 119, 271, 276, 359, 360, 382, 414, 505, 529, 591, 617
 - do stałej, 281
 - do stałych, 362
 - do wskaźnika, 362
 - funkcje, 361
 - rzutowanie w górę, 498
 - typu void, 119
- register, 124, 330
- regułejednej definicji, 72, 199
- reguły przyrostkowe, 169
- reinterpret_cast, 139
- rejestry procesora, 364
- rekurencja**, 106

- require(), funkcja, 204, 325, 607
- require.h**, 194, 206, 317
- requireArgs(), funkcja, 318
- requireMinArgs()**, funkcja, 318
- return, 97, 600
- RETURN**, 297, 366
- ROM, 291
- rozkazy komputera, 68
- rozłączna komplikacja, 69, 71, 167
- rozszerszalność, 508
- rozwinięcie funkcji, 312
- RTTI**, 524, 545
- Rumbaugh, James, 624
- runtime binding**, 506
- run-time type identification, 524, 545
- rzutowanie, 39, 83, 113, 119, 136, 138
 - const_cast, 141
 - dynamic_cast, 544
 - jawne, 139, 543
 - konstruktor kopiący, 494
 - reinterpret_cast, 142
 - static_cast, 140, 544
 - w dół, 525, 543
 - bezpieczne dla typów, 543
- w góre, 39, 492, 498, 504, 516

S

- scalanie programów wynikowych, 72
- scenariusz, 46
- sekwenca znaków specjalnych, 81
- selektor, 104
- setjmp()**, funkcja, 15, 232
- SGI STL**, 88
- short, 111
- sideeffect, 129
- signed, 111
- singly-rooted hierarchy, 538
- sizeof, 111, 143, 196
- sklejanie symboli, 316
- składanie stałych, 268
- składnia, 58, 67
 - deklaracji, 164
 - funkcji, 72
 - zmiennej, 73
- destruktora, 232
- dziedziczenia, 469, 471
- kompozycji, 468
- konstruktora, 232
- referencji, 281
- szablonów, 558
- skadowe, 222
- chronione, 31
- prywatne, 31, 34

skrypt powłoki, 84
słaba kontrola typów, 561
słowa kluczowe
 and, 144
 and_eq, 144
 asm, 143
 auto, 330
 bitand, 144
 bitor, 144
 break, 103
 case, 104
 catch, 457
 char, 83, 109, 249
 class, 27, 32, 144, 220, 330
 compl, 144
 const, 19, 127, 267, 270, 277, 288, 451, 546
 continue, 103
 default, 104
 delete, 20, 40, 41, 136, 183, 241, 534
 do, 101
 double, 109
 else, 100
 enum, 148, 256, 330
 explicit, 426, 427
 extern, 73, 123, 126, 268, 271, 329, 352
 float, 109, 249
 for, 90, 101, 102, 121, 235
 friend, 214, 216, 218, 332, 442
 goto, 105, 232
 if, 98
 inline, 301, 302, 312, 314
 int, 97, 249
 long, 111
 mutable, 291
 namespace, 79, 333
 new, 20, 40, 136, 183, 241, 461
 not, 144
 not_eq, 144
 operator, 387
 or, 144
 or_eq, 144
 private, 30, 212, 487
 protected, 30, 212, 214, 488, 489
 public, 30, 212, 470
 register, 124, 330
 return, 97, 600
 short, 111
 signed, 111
 static, 19, 124, 285, 323, 329, 342, 353
 struct, 144, 145, 220, 330
 switch, 104, 121, 236, 546, 612
 template, 558
 this, 192, 382
 throw, 457
try, 457
typedef, 144
typeid, 545
union, 330
unsigned, 111, 134
using, 79, 333
virtual, 20, 38, 478, 506, 517, 533, 613
void, 97, 119, 164
volatile, 19, 129, 267, 292
while, 87, 101, 121, 122
xor, 144
xor_eq, 144
Smalltalk, 27, 517, 554, 561
source-level debuggers, 69
sparametryzowany typ, 557
specyfikacja
 przydziału pamięci, 122
 systemu, 45
 zmiany sposobu łączenia, 352
specyfikator, 111
 klas pamięci, 330
dostępu, 30, 212, 219
 private, 213
 protected, 214
 public, 212
spójność składni, 472
sprzątanie, 229
 wskaźników, 445
sstream, 414
stacja robocza, 57
Stack, 223, 240
stałe, 127, 267
 argumenty funkcji, 275
 bezpieczeństwo, 269
 const, 276, 278, 282
 funkcje składowe, 287
język C, 271
literały napisowe, 275
łączone wewnętrznie, 268
obiekty, 287
pliki nagłówkowe, 268
podstawianie wartości, 267
przekazywanie stałej przez wartość, 275
składanie, 268
static, 285
statyczne, 339
szablony, 562
w klasach, 282
wartości, 128
 określone podczas kompilacji, 285
wskaźniki, 272, 273, 359
zasięg, 271
zwracanie stałej przez wartość, 276

- standard
ISO, 22.
 plików nagłówkowych, 201
Standard Template Library, 88
 standardowa biblioteka, 77
 C++, 88
 szablonów, 88
 standardowe
 C++, 22
 funkcje biblioteczne, 78
 wejście, 84
 standardowy sposób przekazywania argumentów, 281
standardy języka, 22
startBytes, 182
Stash, 188
static, 19, 124, 285, 323, 329, 342, 353
static const, 339
static_cast, 139, 544, 545
statyczna kontrola typów, 70
statyczne
 funkcje składowe, 342, 483
składowe, 337
 definiowanie pamięci, 337
 inicjalizacja tablicy statycznej, 339
 klasy, 341
statyczny obszar danych, 323
std, 80, 85, 318
sterowanie
 łączeniem, 328
 wykonywaniem programu, 99
sterta, 20, 40, 183, 437
 fragmentacja, 453
 obsługa wjęzyku C, 439
STL, 88
storage, 182
stos, 40, 297, 324, 552, 573
 kontrola dostępu, 223
 liczb całkowitych, 553
strażnik dołączania, 201, 607
strcmp(), funkcja, 429
string, 67, 81, 85, 88, 161, 429
 scalenie tablic, 86
Stroustrup, Bjarne, 15, 345, 517, 556, 599, 621, 623
struct, 144, 145, 220, 330
struktura programu, 80
strukturny, 144, 196, 211, 255
 dane składowe, 189, 197
 deklaracja, 214
friend, 214
 funkcje składowe, 198
 kontrola dostępu, 212
 składowe prywatne, 218
 wskaźniki, 147
 zagniezdzone, 202
strumień wejścia-wyjścia, 77; 78, 82, 199, 372, 4
 manipulator, 83
 odczytywanie wejścia, 84
 styl kodowania, 599
 substitutability, 27
substitution principle, 35
 subtyping, 486
suma logiczna, 131
Sun, 604
switch, 104, 121, 236, 546, 612
system
 dwójkowy, 109
 operacyjny, 70
 uruchomieniowy, 57
system(), funkcja, 84
szablony, 20, 61, 89, 546, 551, 557
 argumenty, 562
 funkcji, 593
 klas, 593
 pliki nagłówkowe, 560
 podstawy, 554
 składnia, 558
 stałe, 562
szkolenie, 62
szybkie projektowanie, 68
- Ś**
- ścieżka wyszukiwania, 75
środowisko
 komplikacji, 69
 programistyczne, 159
- T**
- table-driven code, 166**
tablice, 89, 151
 automatyczne zliczanie, 243
 delete, 450
 dynamiczne tworzenie, 151
 elementy, 151
 indeksowanie zerowe, 151
 new, 450
 przeciążanie
 delete, 458
 operatorów new, 458
 rozmiar, 151
 upodabnianie wskaźnika do tablicy, 451
 usuwanie, 184
 wskaźników do funkcji, 166
 znakowe, 81, 83, 154, 184
template, 557, 558
testowanie, 56

this, 192, 231, **290**, 303, 342, 382, 515
this->, 192;
throw, 457;
Time, 305 .
tłumaczeniejęzyka, 68
toupper(), 300
translator, 75
true, 87, 100, 110
trwałość, 594
try, 457
try-catch, 457
tworzenie wystąpienia, 559
twórca klas, 30
type **definition**, 144
typedef, 144, 146, 180, 189, 330, 381
typeid, 545
typeinfo, 545
type-safe
 downcast, 543
 Linkage, 252
typy danych, 108
 abstrakcyjne, 28, 108
 char, 83, 109
 char*, 154
 const, 127
 double, 109
 dynamiczna kontrola, 70
 float, 109
 int, 97, 148
 klasy, 144
 kontrola typów w wyliczeniach, 149
 łączenie bezpieczne, 252
 rzutowanie, 119
specyfikatory, 111
statyczna kontrola, 70
string, 250
struktury, 144, 196
tablice, 151
tworzenie
 podtypów, 485
 typów złożonych, 144
unie, 150
void, 97
wbudowane, **28**, 109
wskaźniki, 114
wyliczeniowe, 148
zdefiniowane przez użytkownika, 195

U

uchwyty pamięci, 185
ukrywanie
 implementacji, 219, 224
nazw, 476

umieszczanie new, 440
UML, 50
unie, 150, 255
 anonimowe, 257
Unified Modeling Language, 29
unikatowy
 adres pamięci, 194
 identyfikator, 194
union, 330
Unix, 84, 172, 517
unsigned, 111, 134
upakowanie sterty, 184
upcasting, 39, 493
upublicznianie składowych, 488
uruchamianie
 kompilatora, 82
 programów, 159
use cases, 46
using, **79**, 85, 202, 333
 deklaracja, 335
 dyrektywa, 333
using directive, 333
uzależnione obiekty, 345
uzupełnienia nazw, 250

V

varargs, 198
vartype, 258
vector, 67, 88, 90, 372
vi, **604**
vim, 604
virtual, 20, **38**, **478**, **506**, **517**, **533**, **613**
virtual pointer, **511**
void, 97, 119, 164
void*, 119, **186**, **359**, **440**, **443**, **446**
volatile, 19, 129, 267, 292
vpointer, **511**
VPTR, **511**, **512**, **513**, **514**, **530**
VTABLE, **511**, **513**, **517**, **519**
dziedziczenie, 523

W

Waldrop, M. Mitchell, 625
wariancja, 612
wartość
 domyślna, 263
 zwracana, 71
wartownik, 579
warunki
 końcowe, 607
 wstępne, 607

wcięcia, 599, 602
 wczesne wiązanie, 37, 506
weak typing, 561
 wejście konsoli, 84
 wektor zmian, 54
wektory, 88, 151
while, 87, 90, 99, 101, 121, 122
 wiązanie, 506
 dynamiczne, 506
 podczas wykonywania programu, 506
 późne, 506
 wczesne, 506
 wywołania funkcji, 506
 widoczność, 19, 323
wielobieżność, 366
 wielokrotne
 deklaracje, 199
 dziedziczenie, 14, 491, 540, 544, 556, 614
 używanie kodu, 61
 wykorzystywanie kodu, 467
 wielowątkowość, 594
wild-card, 43
 Windows, 78
 wirtualne
 definicje, 522
 operatory, 541
 wywołania w destruktorach, 537
 wirtualny, 38
 destruktor, 533
wolna pamięć, 184
 wskaźnik stosu, 324, 367
 wskaźniki, 112, 114, 116, 118, 181, 184, 359
 tablice, 152
 arytmetyka, 157
 definicja, 114, 274
 dofunkcji, 165, 380
 do obiektów utworzonych na stercie, 445
do składowej, 359, 410
 do składowych, 378
 funkcji, 163
inicjalizowanie, 274
 inteligentne, 406
 rzutowanie w górę, 498
 stałe, 272
 struktury, 147
 tablice, 152
 this, 290
 upodabnianie do tablicy, 451
void, 119
void*, 241, 359
VPTR, 513, 514
 wirtualne, 511, 515
 wycieki pamięci, 184, 445
 wydajność, 64, 179

wyjątki
 bad_alloc, 453
 obsługa, 41
 procedura obsługi, 41
 zgłoszenie, 42
 wyliczenia, 149
 wyłuskanie, 115, 136
 wyrażenia, 107, 137
 delete, 441
 new, 440
 warunkowe, 310
 wysyłanie
 funkcji, 117, 324
 wirtualnej, 514
 wirtualnych wewnętrz konstruktorów, 532
 generowane przez kompilator, 194
 komunikatów, 28, 195
polimorficzne, 511
 programów, 84
 wzorce projektowe, 14, 54, 63

X

xor, 144
 xor_eq, 144
 XP, 55

Z

zagieźdżeni przyjaciele, 216
 zagieźdżone struktury, 202
 zagieźdżony iterator, 408
 zapisywane danych wyjściowych, 475
 zapobieganie przekazywaniu przez wartość, 376
zarządzanie pamięcią, 41
 zasady zastępowania, 35
 zasięg, 120, 333
 globalny, 206
 pliku, 328
 zasłanianie, 35, 476, 507, 527
 zastępowałośc, 27
 zastępowanie
 czyste, 35
 obiektów, 36
 zasady, 35
 zbieracz śmieci, 41, 452
 zbieranie nieużytków, 594
zbiornik na bity, 134
 zbiór, 568
 zewnętrzne
 deklaracje, 74
 odwołanie, 187
 zgłoszenie wyjątku, 451
 zliczanie odwołań, 419

zmiana typu zwracanej wartości, 529

zmienna lista argumentów, 97, 198

zmienne

automatyczne, 40, 124

definiowanie „w locie”, 120

deklaracja, 73

globalne, 122, 615

inicjator, 324

lokalne, 40, 115, 124, 310, 324

łańcuchowe, 186

łączenie, 126

niezainicjowane, 234

register, 330

rejestrowe, 124

składnia deklaracji, 73

stałe, 127

stanu, 166

static, 124

statyczne, 324

vartype, 258

volatile, 129

wskaźniki, 114

zasieg, 120

zewnętrzne, 123

znacznik, 263

końca, 579

znaczniki uruchomieniowe, 160

preprocesora, 160

sprawdzanie w czasie pracy programu, 161

znak

kodASCn, 81

nowego wiersza, 81

specjalny, 81

zunifikowany język modelowania, 29

zwalnianie pamięci, 462

zwracanie

adresów, 279

przez wartość, 364

stałych przez wartość, 404

ż

źródła bibliotek GNU C, 443

ż

żądanie, 28, 29

księgarnia internetowa <http://helion.pl>

Ćwiczenia praktyczne



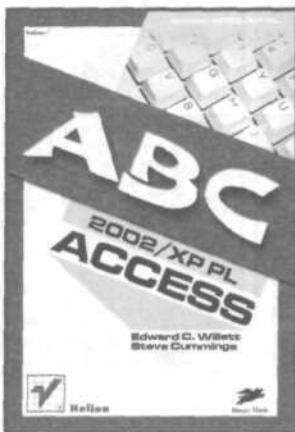
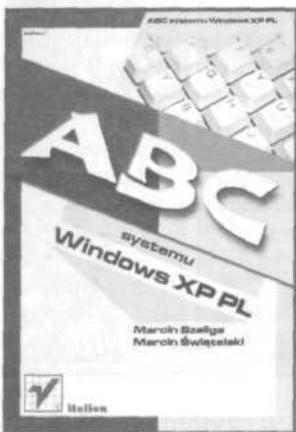
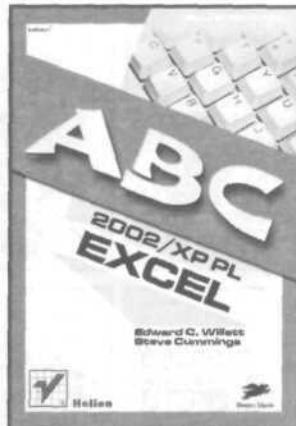
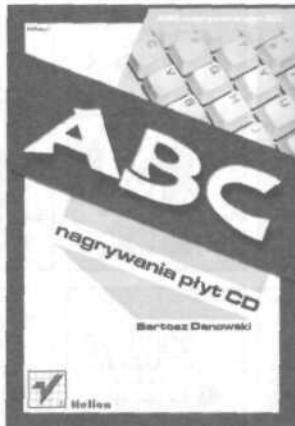
„Ćwiczenia praktyczne” to seria przeznaczona dla tych czytelników, którzy pragną od podstaw poznać konkretny temat. Książki te składają się z ćwiczeń, dzięki czemu czytelnik stopniowo zgłębia dane zagadnienie, mając jednocześnie możliwość systematycznego sprawdzania swojej wiedzy. Napisane przezroczystym i prostym językiem, bogato ilustrowane, opatrzone czytelnymi przykładami są łatwą lekturą nawet dla tych, którzy stawiają pierwsze kroki w świecie informatyki. Książki z tej serii mogą być wykorzystywane na kursach i szkoleniach.



Informatyka w najlepszym wydaniu

Wydawnictwo Helion

ABC



W przypadku książek tej serii szczególny nacisk położyliśmy na prostotę przekazu. Nasi autorzy unikają stosowania specjalistycznego języka, który zniechęca wielu początkujących użytkowników komputerów do dalszej nauki. Praca z książkami z serii „ABC” to doskonaly sposób na stworzenie solidnych podstaw do dalszej nauki. Jeśli natomiast Twoim zamiarem jest opanowanie obsługi programu komputerowego w stopniu średnio zaawansowanym, książki „ABC” to najwahlściwszy wybór.

Czytelnik znajdzie w nich opis wszystkich najważniejszych funkcji programu, ze szczególnym naciukiem na optymalizację pracy i uniwersalność przekazu.



Wydawnictwo Helion

ul. Chopina 6, 44-100 Gliwice; skr. poczt. 462
8(32) 230-98-63, (32) 231-22-19; e-mail: helion@helion.pl

Dla każdego



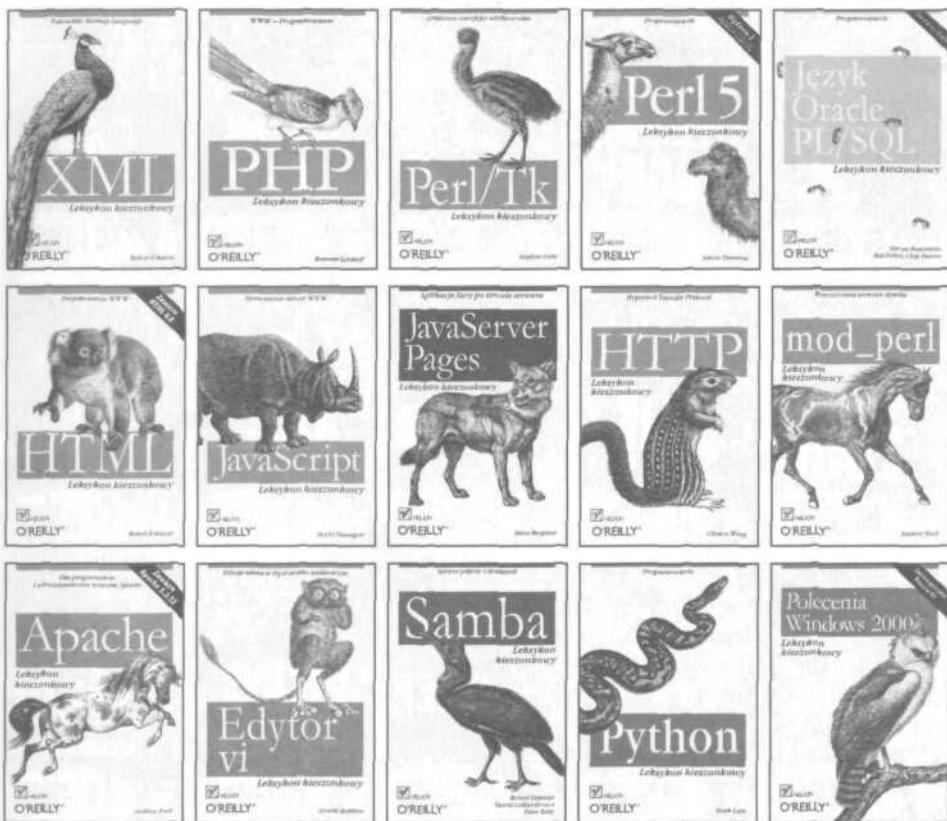
Książki z tej serii to poradniki, które przydadzą się każdemu, kto poważnie myśli o rozwoju komputerowych umiejętności. Zazwyczaj przygody z kolejnym programem komputerowym zaczynasz od wstępnego zapoznania się z jego podstawowymi możliwościami. Metoda „kliknę i sprawdzę, co się stanie” to bardzo dobry sposób nauki, jednak w ten sposób nie poznasz całego potencjału swojego oprogramowania. Książki z serii „Dla każdego” pozwolą Ci usystematyzować zdobytą wiedzę, umożliwią poznanie większości opcji i polecień, podpowiadają, jak optymalnie wykorzystać funkcje programu. Po ich lekturze nie będzie mowy o przypadkowości działań, a efekty Twojej pracy będą w pełni zgodne z zamierzonymi. Dzięki prostym przykładom zawartym w tych książkach będziesz mógł samodzielnie zapoznać się z omawianymi zagadnieniami. Książki są kierowane do początkujących i średnio zaawansowanych czytelników.



Informatyka w najlepszym wydaniu

Wydawnictwo Helion

Leksykon kieszonkowy



Zwięzłe i wyczerpujące kompendium wiedzy, zawierające niemal wszystkie najistotniejsze dla informatyka-praktyka informacje. Opracowując ich strukturę, autorzy położyli nacisk na szybkość dostępu do poszukiwanej informacji. W leksykach kieszonkowych nie znajdziesz wykładów o technologii, ale usystematyzowany opis poleceń, opcji i zmiennych oraz wielu innych elementów, których objaśnienie nie musisz już szukać w kilkusetstronicowych opracowaniach. Jeśli zdobyleś już podstawowe informacje na temat opisywanej aplikacji, administrowania systemem czy programowania w danym języku, to leksykony kieszonkowe będą doskonałym uzupełnieniem Twojej wiedzy. Dzięki swojemu małemu rozmiarowi mogą zastąpić podręczne notatki i przydać Ci się w każdej chwili pracy.



Informatyka w najlepszym wydaniu

Wydawnictwo Helion

O'Reilly



Innowacyjność i profesjonalizm są nierozerwalnie związane z dziedziną informatyki. Jeśli spełniasz **oba warunki, jesteś na najlepszej drodze do osiągnięcia sukcesu.** Dostęp do fachowej literatury znacznie ułatwi Ci poznanie najgłębszych tajemnic technik programowania czy **obsługi** najbardziej zaawansowanych programów komputerowych. Książki Wydawnictwa O'Reilly to źródło rzetelnej wiedzy, podanej w prosty i przystępny sposób. Książki z charakterystycznymi zwierzątkami umieszczonymi na białej okładce cechuje nie tylko bardzo wysoki poziom merytoryczny, ale również przejrzystość i kompleksowość.



Informatyka w najlepszym wydaniu

Wydawnictwo Helion

Księga eksperta



Już sama nazwa serii wyjaśnia treść tych książek. „Księgi eksperta” to prawdziwa skarbnica wiedzy. Znajdziesz w nich właściwie wszystko, co jest potrzebne użytkownikom poszczególnych rodzajów oprogramowania. Już na pierwszy rzut oka można przekonać się, że „Księgi eksperta” to solidne opracowania, których lektura stanowi kolejny krok w świat profesjonalnych zastosowań oprogramowania komputerowego. Konkretnie zagadnienia omówiono w nich w sposób dogłębny i nad wyraz *wyczerpujący*. Ich szczególnowość dodatkowo podkreśla odpowiedzi na „z życia wzięte” pytania. Po nabyciu książki z tej serii nie będziesz już musiał wertować kartek kilku innych książek - znajdziesz w niej wszystko, co potrzebne.



Informatyka w najlepszym wydaniu

Wydawnictwo Helion