helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

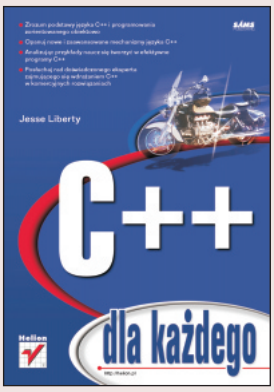
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++ dla każdego



Autor: Jesse Liberty
Tłumaczenie: Marcin Pancewicz
ISBN: 83-7197-538-4
Tytuł oryginału: Teach Yourself C++ in 21 Days. Fourth Edition
Format: B5, stron: 712

Materiał zawarty w tej książce to podstawowe zagadnienia i koncepcje związane z programowaniem w C++, tak więc nie musisz posiadać żadnego doświadczenia w programowaniu w tym języku. Liczne przykłady składni oraz szczegółowa analiza kodu stanowią doskonały przewodnik na początku podróży, której celem jest opanowanie programowania w języku C++. Bez względu na to, czy jesteś początkujący, czy też posiadasz pewne doświadczenie w programowaniu, przekonasz się, że dzięki przejrzystej organizacji tej książki nauka C++ będzie szybka i łatwa.

Książka ta stanowi wprowadzenie do programowania w języku C++. Nie koncentruje się na konkretnej implementacji, lecz raczej opisuje standard ANSI/ISO; znajdziesz w niej również omówienie biblioteki STL (Standard Template Library). Jeśli nawet nie masz doświadczenia w pisaniu programów, to dzięki tej książce:

- Będziesz tworzył szybkie i wydajne programy w C++.
- Zrozumiesz standard ANSI/ISO i skorzystasz z wprowadzonych w nim zmian.
- Opanujesz zaawansowane programowanie z użyciem funkcji, tablic, zmiennych i wskaźników.
- Poznasz C++ oraz obiektowo zorientowane projektowanie, programowanie i analizę.
- Nauczysz się wzbogacać swoje programy za pomocą dziedziczenia i polimorfizmu.
- Będziesz mógł korzystać z dowolnego kompilatora zgodnego ze standardem ANSI/ISO C++.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

O Autorze **15**

Wstęp **17**

Część I **19**

Rozdział 1. Zaczynamy **21**

Wprowadzenie 21
Krótka historia języka C++ 21
Rozwiązywanie problemów 22
Programowanie proceduralne, strukturalne i obiektowe 23
C++ i programowanie zorientowane obiektowo 25
Jak ewoluowało C++ 26
Czy należy najpierw poznać C? 26
C++ a Java i C# 27
Standard ANSI 27
Przygotowanie do programowania 28
Twoje środowisko programowania 28
Tworzenie programu 29
Tworzenie pliku obiektowego za pomocą kompilatora 29
Tworzenie pliku wykonywalnego za pomocą linkera 30
Cykl tworzenia programu 30
HELLO.cpp — twój pierwszy program w C++ 32
Zaczynamy pracę z kompilatorem 34
Budowanie projektu Hello World 34
Błędy kompilacji 36

Rozdział 2. Anatomia programu C++ **37**

Prosty program 37
Rzut oka na obiekt cout 39
Używanie przestrzeni nazw standardowych 41
Komentarze 43
Rodzaje komentarzy 44
Używanie komentarzy 44
Jeszcze jedna uwaga na temat komentarzy 45
Funkcje 45
Korzystanie z funkcji 46

Rozdział 3. Zmienne i stałe....49
Czym jest zmienna?....49
Dane są przechowywane w pamięci....49
Przydzielanie pamięci....50
Rozmiar liczb całkowitych....51
Zapis ze znakiem i bez znaku....52
Podstawowe typy zmiennych....53
Definiowanie zmiennej....54
Uwzględnianie wielkości liter....55
Słowa kluczowe....56
Tworzenie kilku zmienych jednocześnie....56
Przypisywanie zmiennym wartości....56
typedef....58
Kiedy używać typu short, a kiedy typu long?....59
Zawinięcie liczby całkowitej bez znaku....60
Zawinięcie liczby całkowitej ze znakiem....61
Znaki....62
Znaki i liczby....62
Znaki specjalne....63
Stałe....64
Literały....64
Stałe symboliczne....64
Stałe wyliczeniowe....66

Rozdział 4. Wyrażenia i instrukcje....69
Instrukcje....69
Białe spacje....70
Bloki i instrukcje złożone....70
Wyrażenia....70
Operatory....72
Operator przypisania....72
Operatory matematyczne....73
Dzielenie całkowite i reszta z dzielenia....73
Łączenie operatora przypisania z operatorem matematycznym....74
Inkrementacja i dekrementacja....75
Przedrostki i przyrostki....75
Kolejność działań....77
Zagnieżdżanie nawiasów....78
Prawda i fałsz....79
Operatory relacji....79
Instrukcja if....80
Styl wcięć....83
else....84
Zaawansowane instrukcje if....86
Użycie nawiasów klamrowych w zagnieżdżonych instrukcjach if....87
Operatory logiczne....89
Logiczne I....90
Logiczne LUB....90
Logiczne NIE....90
Skrócone obliczanie wyrażeń logicznych....91
Kolejność operatorów logicznych....91
Kilka słów na temat prawdy i fałszu....92
Operator warunkowy (trójelementowy)....93

Rozdział 5. Funkcje 95
Czym jest funkcja? 95
Zwracane wartości, parametry i argumenty 96
Deklarowanie i definiowanie funkcji 97
Deklarowanie funkcji 97
Prototypy funkcji 98
Definiowanie funkcji 99
Wykonywanie funkcji 100
Zmienne lokalne 101
Zakres 102
Zmienne globalne 102
Zmienne globalne: ostrzeżenie 104
Kilka słów na temat zmiennych lokalnych 104
Instrukcje funkcji 106
Kilka słów na temat argumentów funkcji 106
Użycie funkcji jako parametrów funkcji 106
Parametry są zmiennymi lokalnymi 107
Kilka słów na temat zwracanych wartości 109
Parametry domyślne 111
Przeciążanie funkcji 113
Zagadnienia związane z funkcjami 116
Funkcje typu inline 116
Rekurencja 118
Jak działają funkcje — rzut oka „pod maskę” 123
Poziomy abstrakcji 123
Podział pamięci 123
Stos i funkcje 126

Rozdział 6. Programowanie zorientowane obiektowo 129
Czy C++ jest zorientowane obiektowo? 129
Tworzenie nowych typów 130
Po co tworzyć nowy typ? 131
Klasy i składowe 131
Deklarowanie klasy 132
Kilka słów o konwencji nazw 132
Definiowanie obiektu 133
Klasy a obiekty 133
Dostęp do składowych klasy 134
Przypisywać należy obiektom, nie klasom 134
Czego nie zadeklarujesz, tego klasa nie będzie miała 134
Prywatne i publiczne 135
Oznaczanie danych składowych jako prywatnych 137
Prywatność a ochrona 138
Implementowanie metod klasy 140
Konstruktory i destruktory 142
Domyślne konstruktory i destruktory 143
Użycie domyślnego konstruktora 143
Funkcje składowe const 146
Interfejs a implementacja 147
Gdzie umieszczać deklaracje klasy i definicje metod 150
Implementacja inline 151
Klasy, których danymi składowymi są inne klasy 153
Struktury 157
Dlaczego dwa słowa kluczowe spełniają tę samą funkcję 157

Rozdział 7. Sterowanie przebiegiem działania programu 159
Pętle 159
Początki pętli: instrukcja goto 159
Dlaczego nie jest zalecane stosowanie instrukcji goto? 160
Pętle while 160
Bardziej skomplikowane instrukcje while 162
continue oraz break 163
Pętla while (true) 166
Pętla do...while 167
do...while 168
Pętle for 170
Zaawansowane pętle for 171
Puste pętle for 174
Pętle zagnieżdżone 175
Zakres zmiennych w pętlach for 177
Podsumowanie pętli 178
Instrukcja switch 180
Użycie instrukcji switch w menu 182
Program podsumowujący wiadomości 186

Część II 191

Rozdział 8. Wskaźniki 193
Czym jest wskaźnik? 193
Kilka słów na temat pamięci 193
Użycie operatora adresu (&) 194
Przechowywanie adresu we wskaźniku 195
Puste i błędne wskaźniki 195
Nazwy wskaźników 196
Operator wyłuskania 197
Wskaźniki, adresy i zmienne 197
Operowanie danymi poprzez wskaźniki 199
Sprawdzanie adresu 200
Do czego służą wskaźniki? 202
Stos i sterta 203
Operator new 204
delete 204
Wycieki pamięci 206
Tworzenie obiektów na stercie 207
Usuwanie obiektów 207
Dostęp do składowych klasy 208
Dane składowe na stercie 210
Wskaźnik this 212
Utracone wskaźniki 214
Wskaźniki const 217
Wskaźniki const i funkcje składowe const 218
Wskaźniki const this 219
Działania arytmetyczne na wskaźnikach — temat dla zaawansowanych 220

Rozdział 9. Referencje 223
Czym jest referencja? 223
Użycie operatora adresu z referencją 225
Nie można zmieniać przypisania referencji 226

Do czego mogą odnosić się referencje? 227
Zerowe wskaźniki i zerowe referencje 229
Przekazywanie argumentów funkcji przez referencję 229
Tworzenie funkcji swap() otrzymującej wskaźniki 231
Implementacja funkcji swap() za pomocą referencji 233
Nagłówki i prototypy funkcji 234
Zwracanie kilku wartości 235
Zwracanie wartości przez referencję 237
Przekazywanie przez referencję zwiększa efektywność działania programu 238
Przekazywanie wskaźnika const 241
Referencje jako metoda alternatywna 243
Kiedy używać wskaźników, a kiedy referencji 245
Łączenie referencji i wskaźników 246
Nie pozwól funkcji zwracać referencji do obiektu, którego nie ma w zakresie! 247
Zwracanie referencji do obiektu na stercie 249
Wskaźnik, wskaźnik, kto ma wskaźnik? 251

Rozdział 10. Funkcje zaawansowane 253
Przeciążone funkcje składowe 253
Użycie wartości domyślnych 255
Wybór pomiędzy wartościami domyślnymi a przeciążaniem funkcji 257
Konstruktor domyślny 258
Przeciążanie konstruktorów 258
Inicjalizowanie obiektów 260
Konstruktor kopiujący 261
Przeciążanie operatorów 265
Pisanie funkcji inkrementacji 266
Przeciążanie operatora przedrostkowego 267
Zwracanie typów w przeciążonych funkcjach operatorów 268
Zwracanie obiektów tymczasowych bez nadawania im nazw 270
Użycie wskaźnika this 271
Dlaczego stała referencja? 273
Przeciążanie operatora przyrostkowego 273
Różnica pomiędzy przedrostkiem a przyrostkiem 273
Operator dodawania 275
Przeciążanie operatora dodawania 277
Zagadnienia związane z przeciążaniem operatorów 278
Ograniczenia w przeciążaniu operatorów 278
Co przeciążać? 279
Operator przypisania 279
Obsługa konwersji typów danych 282
Operatory konwersji 285

Rozdział 11. Analiza i projektowanie zorientowane obiektowo 287
Budowanie modeli 287
Projektowanie oprogramowania: język modelowania 288
Projektowanie oprogramowania: proces 289
Programowanie ekstremalne 292
Pomysł 292
Analiza wymagań 293
Przypadki użycia 293
Analiza aplikacji 303
Analiza systemów 304

Tworzenie dokumentacji....304
Wizualizacje....305
Dokumentacja produktu....305
Projektowanie....306
Czym są klasy?....306
Przekształcenia....308
Model statyczny....309
Model dynamiczny....318

Rozdział 12. Dziedziczenie....321
Czym jest dziedziczenie?....321
Dziedziczenie i wyprowadzanie....322
Królestwo zwierząt....323
Składnia wyprowadzania....323
Prywatne kontra chronione....325
Konstruktory i destruktory....327
Przekazywanie argumentów do konstruktorów bazowych....329
Przesłanianie funkcji....333
Ukrywanie metod klasy bazowej....335
Wywoływanie metod klasy bazowej....337
Metody wirtualne....338
Jak działają funkcje wirtualne....342
Nie możesz przejść stąd dotąd....343
Okrajanie....344
Destruktory wirtualne....346
Wirtualne konstruktory kopiujące....346
Koszt metod wirtualnych....349

Rozdział 13. Tablice i listy połączone....351
Czym jest tablica?....351
Elementy tablicy....352
Zapisywanie za końcem tablicy....353
Błąd słupka w płocie....356
Inicjalizowanie tablic....356
Deklarowanie tablic....357
Tablice obiektów....358
Tablice wielowymiarowe....360
Inicjalizowanie tablic wielowymiarowych....360
Kilka słów na temat pamięci....362
Tablice wskaźników....363
Deklarowane tablic na stercie....364
Wskaźnik do tablicy a tablica wskaźników....365
Wskaźniki a nazwy tablic....365
Usuwanie tablic ze sterty....367
Tablice znaków....368
strcpy() oraz strncpy()....370
Klasy łańcuchów....371
Listy połączone i inne struktury....378
Analiza listy połączonej....378
Przeniesienie odpowiedzialności....379
Części składowe....379
Czego się nauczyłaś, Dorotko?....388
Klasy tablic....388

Rozdział 14. Polimorfizm 391
Problemy z pojedynczym dziedziczeniem 391
Przenoszenie w górę 394
Rzutowanie w dół 394
Połączenie dwóch list 397
Dziedziczenie wielokrotne 397
Części obiektu z dziedziczeniem wielokrotnym 400
Konstruktory w obiektach dziedziczonych wielokrotnie 401
Eliminowanie niejednoznaczności 403
Dziedziczenie ze wspólnej klasy bazowej 404
Dziedziczenie wirtualne 408
Problemy z dziedziczeniem wielokrotnym 411
Mixiny i klasy metod 412
Abstrakcyjne typy danych 413
Czyste funkcje wirtualne 416
Implementowanie czystych funkcji wirtualnych 417
Złożone hierarchie abstrakcji 420
Które typy są abstrakcyjne? 424
Program podsumowujący wiadomości 425

Część III 435

Rozdział 15. Specjalne klasy i funkcje 437
Statyczne dane składowe 437
Statyczne funkcje składowe 442
Wskaźniki do funkcji 444
Dlaczego warto używać wskaźników do funkcji? 447
Tablice wskaźników do funkcji 450
Przekazywanie wskaźników do funkcji innym funkcjom 452
Użycie instrukcji typedef ze wskaźnikami do funkcji 455
Wskaźniki do funkcji składowych 457
Tablice wskaźników do funkcji składowych 459

Rozdział 16. Dziedziczenie zaawansowane 463
Zawieranie 463
Dostęp do składowych klasy zawieranej 469
Filtrowany dostęp do składowych zawieranych 469
Koszt zawierania 470
Kopiowanie przez wartość 473
Implementowanie poprzez dziedziczenie i zawieranie oraz poprzez delegację 476
Delegacja 477
Dziedziczenie prywatne 485
Klasy zaprzyjaźnione 493
Funkcje zaprzyjaźnione 501
Funkcje zaprzyjaźnione i przeciążanie operatorów 501
Przeciążanie operatora wstawiania 505

Rozdział 17. Strumienie 511
Przegląd strumieni 511
Kapsułkowanie 512
Buforowanie 512
Strumienie i bufory 514
Standardowe obiekty wejścia-wyjścia 514

Przekierowywanie 515
Wejście z użyciem cin 516
Łańcuchy 517
Problemy z łańcuchami 517
Zwracanie referencji do obiektu istream przez operator>> 520
Inne funkcje składowe w dyspozycji cin 520
Wprowadzanie pojedynczych znaków 521
Odczytywanie łańcuchów z wejścia standardowego 523
Użycie cin.ignore() 526
peek() oraz putback() 527
Wyjście poprzez cout 528
Zrzucanie zawartości bufora 528
Powiązane funkcje 528
Manipulatory, znaczniki oraz instrukcje formatowania 530
Użycie cout.width() 530
Ustawianie znaków wypełnienia 531
Funkcja setf() 532
Strumienie kontra funkcja printf() 534
Wejście i wyjście z użyciem plików 537
ofstream 538
Stany strumieni 538
Otwieranie plików dla wejścia i wyjścia 538
Zmiana domyślnego zachowania obiektu ofstream w trakcie otwierania pliku 540
Pliki binarne a pliki tekstowe 542
Przetwarzanie linii polecenia 544

Rozdział 18. Przestrzenie nazw 549
Zaczynamy 549
Funkcje i klasy są rozpoznawane poprzez nazwy 550
Tworzenie przestrzeni nazw 553
Deklarowanie i definiowanie typów 554
Definiowanie funkcji poza przestrzenią nazw 554
Dodawanie nowych składowych 555
Zagnieżdżanie przestrzeni nazw 555
Używanie przestrzeni nazw 556
Słowo kluczowe using 558
Dyrektywa using 558
Deklaracja using 559
Alias przestrzeni nazw 561
Nienazwana przestrzeń nazw 561
Standardowa przestrzeń nazw std 562

Rozdział 19. Wzorce 565
Czym są wzorce? 565
Typy parametryzowane 566
Tworzenie egzemplarza wzorca 566
Definicja wzorca 566
Użycie nazwy 568
Implementowanie wzorca 568
Funkcje wzorcowe 571
Wzorce i przyjaciele 572
Niewzorcowe zaprzyjaźnione klasy i funkcje 572
Ogólne wzorcowe zaprzyjaźnione klasy i funkcje 575

Użycie elementów wzorca 579
Funkcje specjalizowane 582
Wzorce i składowe statyczne 588
Standardowa biblioteka wzorców 591
Kontenery 591
Kontenery sekwencyjne 592
Kontener vector 592
Kontener list 598
Kontener deque 599
Stosy 600
Kolejki 601
Kontenery asocjacyjne 601
Kontener map 601
Inne kontenery asocjacyjne 604
Klasy algorytmów 604
Bezmutacyjne operacje sekwencyjne 605
Mutacyjne algorytmy sekwencyjne 606

Rozdział 20. Wyjątki i obsługa błędów 609

Pluskwy, błędy, pomyłki i „psujący" się kod 609
Wyjątki 610
Wyjątki 611
Jak używane są wyjątki 611
Użycie bloków try oraz bloków catch 616
Wychwytywanie wyjątków 616
Wychwytywanie więcej niż jednego rodzaju wyjątków 617
Hierarchie wyjątków 620
Dane w wyjątkach oraz nazwane obiekty wyjątków 622
Wyjątki i wzorce 629
Wyjątki bez błędów 631
Kilka słów na temat „psującego" się kodu 632
Pluskwy i odpluskwianie 633
Punkty wstrzymania 633
Śledzenie wartości zmiennych 633
Sprawdzanie pamięci 634
Asembler 634

Rozdział 21. Co dalej 635

Preprocesor i kompilator 635
Przeglądanie formy pośredniej 636
Użycie dyrektywy #define 636
Użycie #define dla stałych 636
Użycie #define do definiowania symboli 636
Dyrektywa #else preprocesora 637
Dołączanie i wartowniki dołączania 638
Funkcje makro 640
Po co te wszystkie nawiasy? 641
Makra a funkcje i wzorce 642
Funkcje inline 642
Manipulacje łańcuchami 644
Zamiana w łańcuch 644
Konkatenacja 644
Makra predefiniowane 645

Makro assert() 645
Debuggowanie za pomocą makra assert() 647
Makro assert() a wyjątki 647
Efekty uboczne 648
Niezmienniki klas 648
Wypisywanie wartości tymczasowych 653
Poziomy debuggowania 654
Operacje na bitach 660
Operator AND 661
Operator OR 661
Operator XOR 661
Operator negacji 661
Ustawianie bitów 661
Zerowanie bitów 662
Zmiana stanu bitów na przeciwny 662
Pola bitowe 663
Styl 666
Wcięcia 666
Nawiasy klamrowe 666
Długość linii 666
Instrukcje switch 667
Tekst programu 667
Nazwy identyfikatorów 668
Komentarze 669
Dostęp 670
Definicje klas 670
Dołączanie plików 670
assert() 671
const 671
Następne kroki 671
Gdzie uzyskać pomoc i poradę 671
Przejść do C#? 672
Bądź w kontakcie 672
Program podsumowujący wiadomości 672

Dodatki 683

Dodatek A Dwójkowo i szesnastkowo 685
Inne podstawy 686
Wokół podstaw 686
Dwójkowo 688
Dlaczego podstawa 2? 689
Bity, bajty, nible 689
Co to jest KB? 690
Liczby dwójkowe 690
Szesnastkowo 691

Dodatek B Słowa kluczowe C++ 695

Dodatek C Kolejność operatorów 697

Skorowidz 699

Rozdział 6.

Programowanie zorientowane obiektowo

Klasy rozszerzają wbudowane w C++ możliwości, ułatwiające rozwiązywanie złożonych, „rzeczywistych” problemów.

Z tego rozdziału dowiesz się:

- czym są klasy i obiekty,
- jak definiować nową klasę oraz tworzyć obiekty tej klasy,
- czym są funkcje i dane składowe,
- czym są konstruktory i jak z nich korzystać.

Czy C++ jest zorientowane obiektowo?

Język C++ stanowi pomost pomiędzy programowaniem zorientowanym obiektowo a językiem C, najpopularniejszym językiem programowania aplikacji komercyjnych. Celem jego autorów było stworzenie obiektowo zorientowanego języka dla tej szybkiej i efektywnej platformy.

Język C jest etapem pośrednim pomiędzy wysokopoziomowymi językami aplikacji „firmowych”, takimi jak COBOL, a niskopoziomowym, wysokowydajnym, lecz trudnym do użycia asemblerem. C wymusza programowanie „strukturalne”, w którym poszczególne zagadnienia są dzielone na mniejsze jednostki powtarzalnych działań, zwanych funkcjami.

Programy, które piszemy na początku dwudziestego pierwszego wieku, są dużo bardziej złożone niż te, które były pisane pod koniec wieku dwudziestego. Programy stworzone w językach proceduralnych są trudne w zarządzaniu i konserwacji, a ich

rozbudowa jest niemożliwa. Graficzne interfejsy użytkownika, Internet, telefonia cyfrowa i bezprzewodowa oraz wiele innych technologii, znacznie zwiększyły poziom skomplikowania nowych projektów, a wymagania konsumentów dotyczące jakości interfejsu użytkownika wzrosły.

W obliczu rosnących wymagań, programiści bacznie przyjrzeli się przemysłowi informatycznemu. Wnioski, do jakich doszli, były co najmniej przygnębiające. Oprogramowanie powstawało z opóźnieniem, posiadało błędy, działało niestabilnie i było drogie. Projekty regularnie przekraczały budżet i trafiały na rynek z opóźnieniem. Koszt obsługi tych projektów był znaczny, zmarnowano ogromne ilości pieniędzy.

Jedynym wyjściem z tej sytuacji okazało się tworzenie oprogramowania zorientowanego obiektowo. Języki programowania obiektowego stworzyły silne więzy pomiędzy strukturami danych a metodami manipulowania tymi danymi. A co najważniejsze, w programowaniu zorientowanym obiektowo nie już musisz myśleć o strukturach danych i manipulujących nimi funkcjami; myślisz o obiektach. Rzeczach.

Świat jest wypełniony przedmiotami: samochodami, psami, drzewami, chmurami, kwiatami. Rzeczy. Każda *rzecz* ma charakterystykę (szybki, przyjazny, brązowy, puszysty, ładny). Większość rzeczy cechuje jakieś zachowanie (ruch, szczekanie, wzrost, deszcz, uwiąd). Nie myślimy o danych psa i o tym, jak moglibyśmy nimi manipulować — myślimy o psie jako o rzeczy: do czego jest podobny i co robi.

Tworzenie nowych typów

Poznałeś już kilka typów zmiennych, m.in. liczby całkowite i znaki. Typ zmiennej dostarcza nam kilka informacji o niej. Na przykład, jeśli zadeklarujesz zmienne `Height` (wysokość) i `Width` (szerokość) jako liczby całkowite typu `unsigned short int`, wiesz, że w każdej z nich możesz przechować wartość z przedziału od 0 do 65 535 (przy założeniu że typ `unsigned short int` zajmuje dwa bajty pamięci). Są to liczby całkowite bez znaku; próba przechowania w nich czegokolwiek innego powoduje błąd. W zmiennej typu `unsigned short` nie możesz umieścić swojego imienia, nie powinieneś nawet próbować.

Deklarując te zmienne jako `unsigned short int`, wiesz, że możesz dodać do siebie wysokość i szerokość oraz przypisać tę wartość innej zmiennej.

Typ zmiennych informuje:

- o ich rozmiarze w pamięci,
- jaki rodzaj informacji mogą zawierać,
- jakie działania można na nich wykonywać.

W tradycyjnych językach, takich jak C, typy były wbudowane w język. W C++ programista może rozszerzyć język, tworząc potrzebne mu typy, zaś każdy z tych nowych typów może być w pełni funkcjonalny i dysponować tą samą siłą, co typy wbudowane.

Po co tworzyć nowy typ?

Programy są zwykle pisane w celu rozwiązania jakiegoś realnego problemu, takiego jak prowadzenie rejestru pracowników czy symulacja działania systemu grzewczego. Choć istnieje możliwość rozwiązywania tych problemów za pomocą programów napisanych wyłącznie przy użyciu liczb całkowitych i znaków, jednak w przypadku większych, bardziej rozbudowanych problemów, dużo łatwiej jest stworzyć reprezentacje obiektów, o których się mówi. Innymi słowy, symulowanie działania systemu grzewczego będzie łatwiejsze, gdy stworzymy zmienne reprezentujące pomieszczenia, czujniki ciepła, termostaty i bojlery. Im bardziej te zmienne odpowiadają rzeczywistości, tym łatwiejsze jest napisanie programu.

Klasy i składowe

Nowy typ zmiennych tworzy się, deklarując klasę. Klasa jest właściwie grupą zmiennych — często o różnych typach — skojarzonych z zestawem odnoszących się do nich funkcji.

Jedną z możliwości myślenia o samochodzie jest potraktowanie go jako zbioru kół, drzwi, foteli, okien itd. Inna możliwość to wyobrażenie sobie, co samochód może zrobić: jeździć, przyspieszać, zwalniać, zatrzymywać się, parkować itd. Klasa umożliwia kapsułkowanie, czyli upakowanie, tych różnych części oraz różnych działań w jeden zbiór, który jest nazywana obiektem.

Upakowanie wszystkiego, co wiesz o samochodzie, w jedną klasę przynosi programiście liczne korzyści. Wszystko jest na miejscu, ułatwia to odwoływanie się, kopiowanie i manipulowanie danymi. Klienty twojej klasy — tj. te części programu, które z niej korzystają — mogą używać twojego obiektu bez zastanawiania się, co znajduje się w środku i jak on działa.

Klasa może składać się z dowolnej kombinacji zmiennych prostych oraz zmiennych innych klas. Zmienna wewnątrz klasy jest nazywana zmienną składową lub daną składową. Klasa `Car` (samochód) może posiadać składowe reprezentujące siedzenia, typ radia, opony itd.

Zmienne składowe są zmiennymi w danej klasie. Stanowią one część klasy, tak jak koła i silnik stanowią część samochodu.

Funkcje w danej klasie zwykle manipulują zmiennymi składowymi. Funkcje klasy nazywa się funkcjami składowymi lub metodami klasy. Metodami klasy `Car` mogą

być Start() (uruchom) oraz Brake() (hamuj). Klasa Cat (kot) może posiadać zmienne składowe, reprezentujące wiek i wagę; jej metodami mogą być Sleep() (śpij), Meow() (miaucz) czy ChaseMice() (łap myszy).

Funkcje składowe (metody) są funkcjami w klasie. Podobnie jak zmienne składowe, stanowią część klasy i określają, co dana klasa może zrobić.

Deklarowanie klasy

Aby zadeklarować klasę, użyj słowa kluczowego class, po którym następuje otwierający nawias klamrowy, a następnie lista danych składowych i metod tej klasy. Deklaracja kończy się zamykającym nawiasem klamrowym i średnikiem. Oto deklaracja klasy o nazwie Cat (kot):

```
class Cat
{
    unsigned int  itsAge;
    unsigned int  itsWeight;
    void Meow();
};
```

Zadeklarowanie takiej klasy nie powoduje zaalokowania pamięci dla obiektu Cat. Informuje jedynie kompilator, czym jest typ Cat, jakie dane zawiera (itsAge — jego wiek oraz itsWeight — jego waga) oraz co może robić (Meow() — miaucz). Informuje także kompilator, jak duża jest zmienna typu Cat — to jest, jak dużo miejsca w pamięci ma przygotować w przypadku tworzenia zmiennej typu Cat. W tym przykładzie, o ile typ int ma cztery bajty, zmienna typu Cat zajmuje osiem bajtów: cztery bajty dla zmiennej itsAge i cztery dla zmiennej itsWeight. Funkcja Meow() nie zajmuje miejsca, gdyż dla funkcji składowych (metod) miejsce nie jest rezerwowane.

Kilka słów o konwencji nazw

Jako programista, musisz nazwać wszystkie swoje zmienne składowe, funkcje składowe oraz klasy. Jak przeczytałeś w rozdziale 3., „Stałe i zmienne”, nazwy te powinny być zrozumiałe i znaczące. Dobrymi nazwami klas mogą być wspomniana Cat, Rectangle (prostokąt) czy Employee (pracownik). Meow(), ChaseMice() czy StopEngine() (zatrzymaj silnik) również są dobrymi nazwami funkcji, gdyż informują, co robią te funkcje. Wielu programistów nadaje nazwom zmiennych składowych przedrostek „its” (jego), tak jak w zmiennych itsAge, itsWeight czy itsSpeed (jego szybkość). Pomaga to w odróżnieniu zmiennych składowych od innych zmiennych.

Niektórzy programiści wolą przedrostek „my” (mój), tak jak w nazwach myAge, myWeight czy mySpeed. Jeszcze inni używają po prostu litery m (od słowa *member* — składowa), czasem wraz ze znakiem podkreślenia (_): mAge i m_age, mWeight i m_weight czy mSpeed i m_speed.

Język C++ uwzględnia wielkość liter, dlatego wszystkie nazwy klas powinny przestrzegać tej samej konwencji. Dzięki temu nigdy nie będziesz musiał sprawdzać pisowni nazwy klasy (czy to było `Rectangle`, `rectangle` czy `RECTANGLE`?).

Niektórzy programiści lubią poprzedzić każdą nazwę klasy określoną literą — na przykład `cCat` czy `cPerson` — podczas, gdy inni używają wyłącznie dużych lub małych liter. Ja sam korzystam z konwencji, w której wszystkie nazwy klas rozpoczynają się od dużej litery, tak jak `Cat` czy `Person` (osoba).

Wielu programistów rozpoczyna wszystkie nazwy funkcji od dużej litery, zaś wszystkie nazwy zmiennych — od małej. Słowa zwykle rozdzielane są znakiem podkreślenia — tak jak w `Chase_Mice` — lub poprzez zastosowanie dużej litery dla każdego słowa — na przykład `ChaseMice` czy `DrawCircle` (rysuj okrąg).

Ważne jest, by wybrać określony styl i trzymać się go w każdym programie. Z czasem rozwiniesz swój styl nie tylko na konwencje nazw, ale także na wcięcia, wyrównanie nawiasów klamrowych oraz styl komentarzy.



W firmach programistycznych powszechne jest określenie standardu wielu elementów stylu zapisu kodu źródłowego. Sprawia on, że wszyscy programiści mogą łatwo odczytywać wzajemnie swój kod.

Definiowanie obiektu

Definiowanie obiektu nowego typu przypomina definiowanie zmiennej całkowitej:

```
unsigned int GrossWeight; // definicja zmiennej typu unsigned int
Cat Mruczek;              // definicja zmiennej typu Cat
```

Ten kod definiuje zmienną o nazwie `GrossWeight` (łączna waga), której typem jest `unsigned int`. Oprócz tego definiuje zmienną o nazwie `Mruczek`, która jest obiektem klasy (typu) `Cat`.

Klasy a obiekty

Nigdy nie karmi się definicji kota, lecz konkretnego kota. Należy dokonać rozróżnienia pomiędzy ideą kota a konkretnym kotem, który właśnie ociera się o twoje nogi. C++ również dokonuje rozróżnienia pomiędzy klasą `Cat`, będącą ideą kota, a poszczególnymi obiektami typu `Cat`. Tak więc `Mruczek` jest obiektem typu `Cat`, tak jak `GrossWeight` jest zmienną typu `unsigned int`.

Obiekt jest indywidualnym egzemplarzem klasy.

Dostęp do składowych klasy

Gdy zdefiniujesz już faktyczny obiekt Cat — na przykład Mruczek — w celu uzyskania dostępu do jego składowych możesz użyć operatora kropki (.). Aby zmiennej składowej itsWeight obiektu Mruczek przypisać wartość 50, powinieneś napisać:

```
Mruczek.itsWeight = 50;
```

Aby wywołać funkcję Meow(), możesz napisać:

```
Mruczek.Meow();
```

Gdy używasz metody klasy, oznacza to, że wywołujesz tę metodę. W tym przykładzie wywołałeś metodę Meow() obiektu Mruczek.

Przypisywać należy obiektom, nie klasom

W C++ nie przypisuje się wartości typom; przypisuje się je zmiennym. Na przykład, nie można napisać:

```
int = 5;              // źle
```

Kompilator uzna to za błąd, gdyż nie można przypisać wartości pięć typowi całkowitemu. Zamiast tego musisz zdefiniować zmienną typu całkowitego i przypisać jej wartość 5. Na przykład:

```
int x ;               // definicja zmiennej typu int
x = 5;                // ustawienie wartości zmiennej x na 5
```

Jest to skrócony zapis stwierdzenia: „Przypisz wartość 5 zmiennej x, która jest zmienną typu int". Nie można również napisać:

```
Cat.itsAge = 5;       // źle
```

Kompilator uzna to za błąd, gdyż nie możesz przypisać wartości 5 do elementu itsAge klasy Cat. Zamiast tego musisz zdefiniować egzemplarz obiektu klasy Cat i dopiero wtedy przypisać wartość jego składowej. Na przykład:

```
Cat Mruczek;          // podobnie jak  int x;
Mruczek.itsAge = 5;   // podobnie jak  x = 5;
```

Czego nie zadeklarujesz, tego klasa nie będzie miała

Przeprowadź taki eksperyment: podejdź do trzylatka i pokaż mu kota. Następnie powiedz: To jest Mruczek. Mruczek zna sztuczkę. Mruczek, zaszczekaj! Dziecko roześmieje się i powie: „Nie, głuptasie, koty nie szczekają!".

Jeśli napisałeś:

```
Cat Mruczek;          // tworzy obiekt Cat o nazwie Mruczek
Mruczek.Bark();       // nakazuje Mruczkowi szczekać
```

Kompilator wypisze: „Nie, głuptasie, koty (cats) nie szczekają!" (Być może w twoim kompilatorze ten komunikat będzie brzmiał nieco inaczej.). Kompilator wie, że Mruczek nie może szczekać, gdyż klasa `Cat` nie posiada metody `Bark()` (szczekaj). Kompilator nie pozwoli Mruczkowi nawet zamiauczeć, jeśli nie zdefiniujesz dla niego funkcji `Meow()` (miaucz).

TAK	NIE
Do deklarowania klasy używaj słowa kluczowego `class`.	Nie myl deklaracji z definicją. Deklaracja mówi czym jest klasa, a definicja przygotowuje pamięć dla obiektu.
W celu uzyskania dostępu do zmiennych i funkcji składowych klasy używaj operatora kropki (`.`).	Nie myl klasy z obiektem.
	Nie przypisuj klasie wartości. Wartości przypisuj danym składowym obiektu.

Prywatne i publiczne

W deklaracji klasy używanych jest także kilka innych słów kluczowych. Dwa najważniejsze z nich to: `public` (publiczny) i `private` (prywatny).

Wszystkie składowe klasy — dane i metody — są domyślnie prywatne. Prywatne składowe mogą być używane tylko przez metody należące do danej klasy. Składowe publiczne są dostępne dla innych funkcji i klas. To rozróżnienie jest ważne, choć na początku może sprawiać kłopot. Aby to lepiej wyjaśnić, spójrzmy na poprzedni przykład:

```
class Cat
{
    unsigned int  itsAge;
    unsigned int  itsWeight;
    viod Meow();
};
```

W tej deklaracji, składowe `itsAge`, `itsWeight` oraz `Meow()` są prywatne, gdyż wszystkie składowe klasy są prywatne domyślnie. Oznacza to, że dopóki nie postanowisz inaczej, pozostaną one prywatne.

Jeśli jednak w funkcji `main()` napiszesz na przykład:

```
Cat Bobas;
Bobas.itsAge = 5;      // błąd! nie można używać prywatnych danych!
```

kompilator uzna to za błąd. We wcześniejszej deklaracji powiedziałeś kompilatorowi, że składowych `itsAge`, `itsWeight` oraz `Meow()` będziesz używał tylko w funkcjach składowych klasy `Cat`. W powyższym fragmencie kodu próbujesz odwołać się do zmiennej składowej obiektu `Bobas` spoza metody klasy `Cat`. To, że `Bobas` jest obiektem klasy `Cat`, nie oznacza, że możesz korzystać z tych elementów obiektu `Bobas`, które są prywatne.

Właśnie to jest źródłem niekończących się kłopotów początkujących programistów C++. Już słyszę, jak narzekasz: „Hej! Właśnie napisałem, że Bobas jest kotem, tj. obiektem klasy Cat. Dlaczego Bobas nie ma dostępu do swojego własnego wieku?”. Odpowiedź brzmi: Bobas ma dostęp, ale ty nie masz. Bobas, w swoich własnych metodach, ma dostęp do wszystkich swoich składowych, zarówno publicznych, jak i prywatnych. Nawet, jeśli to ty tworzysz obiekt klasy Cat, nie możesz przeglądać ani zmieniać tych jego składowych, które są prywatne.

Aby mieć dostęp do składowych obiektu Cat, powinieneś napisać:

```
class Cat
{
public:
    unsigned int  itsAge;
    unsigned int  itsWeight;
    void Meow();
};
```

Teraz składowe itsAge, itsWeight oraz Meow() są publiczne. Bobas.itsAge = 5; kompiluje się bez problemów.

Listing 6.1 przedstawia deklarację klasy Cat z publicznymi zmiennymi składowymi.

Listing 6.1. *Dostęp do publicznych składowych w prostej klasie*

```
 0:  // Demonstruje deklaracje klasy oraz
 1:  // definicje obiektu tej klasy.
 2:
 3:  #include <iostream>
 4:
 5:  class Cat           // deklaruje klasę Cat (kot)
 6:  {
 7:  public:             // następujące po tym składowe są publiczne
 8:      int itsAge;     // zmienna składowa
 9:      int itsWeight;  // zmienna składowa
10:  };                  // zwróć uwagę na średnik
11:
12:
13:  int main()
14:  {
15:      Cat Mruczek;
16:      Mruczek.itsAge = 5;    // przypisanie do zmiennej składowej
17:      std::cout << "Mruczek jest kotem i ma " ;
18:      std::cout << Mruczek.itsAge << " lat.\n";
19:      return 0;
20:  }
```

wynik

```
Mruczek jest kotem i ma 5 lat.
```

analiza

Linia 5. zawiera słowo kluczowe class. Informuje ono kompilator, że następuje po nim deklaracja klasy. Nazwa nowej klasy następuje bezpośrednio po słowie kluczowym class. W tym przypadku nazwą klasy jest Cat (kot).

Ciało deklaracji rozpoczyna się w linii 6. od otwierającego nawiasu klamrowego i kończy się zamykającym nawiasem klamrowym i średnikiem w linii 10. Linia 7. zawiera słowo kluczowe `public`, które wskazuje, że wszystko, co po nim nastąpi, będzie publiczne, aż do natrafienia na słowo kluczowe `private` lub koniec deklaracji klasy.

Linie 8. i 9. zawierają deklaracje składowych klasy, `itsAge` (jego wiek) oraz `itsWeight` (jego waga).

W linii 13. rozpoczyna się funkcja `main()`. W linii 15. `Mruczek` jest definiowany jako egzemplarz klasy `Cat` — tj. jako obiekt klasy `Cat`. W linii 16. wiek `Mruczka` jest ustawiany na `5`. W liniach 17. i 18. zmienna składowa `itsAge` zostaje użyta do wypisania informacji o kocie `Mruczku`.



Spróbuj wykomentować linię 7., po czym skompiluj program ponownie. W linii 16. wystąpi błąd, gdyż zmienna składowa `itsAge` nie będzie już składową publiczną. Domyślnie, wszystkie składowe klasy są prywatne.

Oznaczanie danych składowych jako prywatnych

Powinieneś przyjąć jako ogólną regułę, że dane składowe klasy należy utrzymywać jako prywatne. W związku z tym musisz stworzyć publiczne funkcje składowe, zwane funkcjami dostępowymi lub *akcesorami*. Funkcje te umożliwią odczyt zmiennych składowych i przypisywanie im wartości. Te funkcje dostępowe (akcesory) są funkcjami składowymi, używanymi przez inne części programu w celu odczytywania i ustawiania prywatnych zmiennych składowych.

Publiczny akcesor jest funkcją składową klasy, używaną albo do odczytu wartości prywatnej zmiennej składowej klasy, albo do ustawiania wartości tej zmiennej.

Dlaczego miałbyś utrudniać sobie życie dodatkowym poziomem pośredniego dostępu? Łatwiej niż posługiwać się akcesorami jest używać danych,.

Akcesory umożliwiają oddzielenie szczegółów przechowywania danych klasy od szczegółów jej używania. Dzięki temu możesz zmieniać sposób przechowywania danych klasy bez konieczności przepisywania funkcji, które z tych danych korzystają.

Jeśli funkcja, która chce poznać wiek kota, odwoła się bezpośrednio do zmiennej `itsAge` klasy `Cat`, będzie musiała zostać przepisana, jeżeli ty, jako autor klasy `Cat`, zdecydujesz się na zmianę sposobu przechowywania tej zmiennej. Jednak posiadając funkcję składową `GetAge()` (pobierz wiek), klasa `Cat` może łatwo zwrócić właściwą wartość bez względu na to, w jaki sposób przechowywany będzie wiek. Funkcja wywołująca nie musi wiedzieć, czy jest on przechowywany jako zmienna typu `unsigned int` czy `long` lub czy wiek jest obliczany w miarę potrzeb.

Ta technika ułatwia zapanowanie nad programem. Przedłuża istnienie kodu, gdyż zmiany projektowe nie powodują, że program staje się przestarzały.

Listing 6.2 przedstawia klasę `Cat` zmodyfikowaną tak, by zawierała prywatne dane składowe i publiczne akcesory. Zwróć uwagę, że ten listing przedstawia wyłącznie deklarację klasy, nie ma w nim kodu wykonywalnego.

Listing 6.2. *Klasa z akcesorami*

```
 0:  // Deklaracja klasy Cat
 1:  // Dane składowe są prywatne, publiczne akcesory pośredniczą
 2:  // w ustawianiu i odczytywaniu wartości składowych prywatnych
 3:
 4:  class Cat
 5:  {
 6:  public:
 7:      // publiczne akcesory
 8:      unsigned int GetAge();
 9:      void SetAge(unsigned int Age);
10:
11:      unsigned int GetWeight();
12:      void SetWeight(unsigned int Weight);
13:
14:      // publiczna funkcja składowa
15:      void Meow();
16:
17:      // prywatne dane składowe
18:  private:
19:      unsigned int  itsAge;
20:      unsigned int  itsWeight;
21:
22:  };
```



Ta klasa posiada pięć metod publicznych. Linie 8. i 9. zawierają akcesory dla składowej `itsAge`. Linie 11. i 12. zawierają akcesory dla składowej `itsWeight`. Te akcesory ustawiają zmienne składowe i zwracają ich wartości.

W linii 15. jest zadeklarowana publiczna funkcja składowa `Meow()`. Ta funkcja nie jest akcesorem. Nie zwraca wartości ani ich nie ustawia; wykonuje inną usługę dla klasy — wypisuje słowo `Miau`.

Zmienne składowe są zadeklarowane w liniach 19. i 20.

Aby ustawić wiek `Mruczka`, powinieneś przekazać wartość metodzie `SetAge()` (ustaw wiek), na przykład:

```
Cat Mruczek;
Mruczek.SetAge(5); // ustawia wiek Mruczka
                   // używając publicznego akcesora
```

Prywatność a ochrona

Zadeklarowanie metod lub danych jako prywatnych umożliwia kompilatorowi wyszukanie w programach pomyłek, zanim staną się one błędami. Każdy szanujący się

programista potrafi znaleźć sposób na obejście prywatności składowych. Stroustrup, autor języka C++, stwierdza, że „...mechanizmy ochrony z poziomu języka chronią przed pomyłką, a nie przed świadomym oszustwem" (WNT, 1995).

Słowo kluczowe class

Składnia słowa kluczowego `class` jest następująca:

```
class nazwa_klasy
{
// słowa kluczowe kontroli dostępu
// zadeklarowane zmienne i składowe klasy
};
```

Słowo kluczowe `class` służy do deklarowania nowych typów. Klasa stanowi zbiór danych składowych klasy, które są zmiennymi różnych typów, także innych klas. Klasa zawiera także funkcje klasy — tzw. metody — które są funkcjami używanymi do manipulowania danymi w danej klasie i wykonywania innych usług dla klasy.

Obiekty nowego typu definiuje się w taki sam sposób, w jaki definiuje się inne zmienne. Należy określić typ (klasę), a po nim nazwę zmiennej (obiektu). Do uzyskania dostępu do funkcji i danych klasy służy operator kropki (`.`).

Słowa kluczowe kontroli dostępu określają, które sekcje klasy są prywatne, a które publiczne. Domyślnie wszystkie składowe klasy są prywatne. Każde słowo kluczowe zmienia kontrolę dostępu od danego miejsca aż do końca klasy, lub kontrolę wystąpienia następnego słowa kluczowego kontroli dostępu. Deklaracja klasy kończy się zamykającym nawiasem klamrowym i średnikiem.

Przykład 1

```
class Cat
{
public:
    unsigned int Age;
    unsigned int Weight;
    void Meow();
};

Cat  Mruczek;
Mruczek.Age = 8;
Mruczek.Weight = 18;
Mruczek.Meow();
```

Przykład 2

```
class Car
{
public:           // pięć następnych składowych jest publicznych
    void Start();
    void Accelerate();
    void Brake();
    void SetYear(int year);
    int GetYear();

private:  // pozostała część jest prywatna
    int Year;
    char Model [255];
```

```
};      // koniec deklaracji klasy

Car OldFaithful;   // tworzy egzemplarz klasy
int bought;  // lokalna zmienna typu int
OldFaithful.SetYear(84);  // ustawia składową Year na 84
bought = OldFaithful.GetYear(); // ustawia zmienną bought na 84
OldFaithful.Start(); //wywołuje metodę Start
```

TAK	NIE
Deklaruj zmienne składowe jako prywatne. Używaj publicznych akcesorów, czyli publicznych funkcji dostępowych. Odwołuj się do prywatnych zmiennych składowych z funkcji składowych klasy.	Nie używaj prywatnych zmiennych składowych klasy poza tą klasą.

Implementowanie metod klasy

Akcesory stanowią publiczny interfejs do prywatnych danych klasy. Każdy akcesor musi posiadać, wraz z innymi zadeklarowanymi metodami klasy, implementację. Implementacja jest nazywana definicją funkcji.

Definicja funkcji składowej rozpoczyna się od nazwy klasy, po której występują dwa dwukropki, nazwa funkcji i jej parametry. Listing 6.3 przedstawia pełną deklarację prostej klasy `Cat`, wraz z implementacją jej akcesorów i jednej ogólnej funkcji tej klasy.

Listing 6.3. *Implementacja metod prostej klasy*

```
0:  // Demonstruje deklarowanie klasy oraz
1:  // definiowanie jej metod
2:
3:  #include <iostream>        // dla cout
4:
5:  class Cat                   // początek deklaracji klasy
6:  {
7:  public:                     // początek sekcji publicznej
8:      int GetAge();           // akcesor
9:      void SetAge (int age);  // akcesor
10:     void Meow();            // ogólna funkcja
11: private:                    // początek sekcji prywatnej
12:     int itsAge;             // zmienna składowa
13: };
14:
15: // GetAge, publiczny akcesor
16: // zwracający wartość składowej itsAge
17: int Cat::GetAge()
18: {
19:     return itsAge;
```

```
20:  }
21:
22:  // definicja SetAge, akcesora
23:  // publicznego
24:  // ustawiającego składową itsAge
25:  void Cat::SetAge(int age)
26:  {
27:      // ustawia zmienną składową itsAge
28:      // zgodnie z wartością przekazaną w parametrze age
29:      itsAge = age;
30:  }
31:
32:  // definicja metody Meow
33:  // zwraca: void
34:  // parametery: brak
35:  // działanie: wypisuje na ekranie słowo "miauczy"
36:  void Cat::Meow()
37:  {
38:      std::cout << "Miauczy.\n";
39:  }
40:
41:  // tworzy kota, ustawia jego wiek, sprawia,
42:  // że miauczy, wypisuje jego wiek i ponownie miauczy.
43:  int main()
44:  {
45:      Cat Mruczek;
46:      Mruczek.SetAge(5);
47:      Mruczek.Meow();
48:      std::cout << "Mruczek jest kotem i ma " ;
49:      std::cout << Mruczek.GetAge() << " lat.\n";
50:      Mruczek.Meow();
51:      return 0;
52:  }
```

wynik

```
Miauczy.
Mruczek jest kotem i ma 5 lat.
Miauczy.
```

analiza

Linie od 5. do 13. zawierają definicję klasy `Cat` (kot). Linia 7. zawiera słowo kluczowe `public`, które informuje kompilator, że to, co po nim następuje, jest zestawem publicznych składowych. Linia 8. zawiera deklarację publicznego akcesora `GetAge()` (pobierz wiek). `GetAge()` zapewnia dostęp do prywatnej zmiennej składowej `itsAge` (jego wiek), zadeklarowanej w linii 12. Linia 9. zawiera publiczny akcesor `SetAge()` (ustaw wiek). Funkcja `SetAge()` otrzymuje parametr typu `int`, który następnie przypisuje składowej `itsAge`.

Linia 10. zawiera deklarację metody `Meow()` (miaucz). Funkcja `Meow()` nie jest akcesorem. Jest to ogólna metoda klasy, wypisująca na ekranie słowo „Miauczy.”

Linia 11. rozpoczyna sekcję prywatną, która obejmuje jedynie zadeklarowaną w linii 12. prywatną składową `itsAge`. Deklaracja klasy kończy się zamykającym nawiasem klamrowym i średnikiem.

Linie od 17. do 20. zawierają definicję składowej funkcji `GetAge()`. Ta metoda nie ma parametrów i zwraca wartość całkowitą. Zauważ, że ta metoda klasy zawiera nazwę klasy, dwa dwukropki oraz nazwę funkcji (linia 17.). Ta składnia informuje kompilator, że definiowana funkcja `GetAge()` jest właśnie tą funkcją, która została zadeklarowana w klasie `Cat`. Poza formatem tego nagłówka, definiowanie funkcji własnej `GetAge()` niczym nie różni się od definiowania innych (zwykłych) funkcji.

Funkcja `GetAge()` posiada tylko jedną linię i zwraca po prostu wartość zmiennej składowej `itsAge`. Zauważ, że funkcja `main()` nie ma dostępu do tej zmiennej składowej, gdyż jest ona prywatna dla klasy `Cat`. Funkcja `main()` ma za to dostęp do publicznej metody `GetAge()`. Ponieważ ta metoda jest składową klasy `Cat`, ma pełny dostęp do zmiennej `itsAge`. Dzięki temu może zwrócić funkcji `main()` wartość zmiennej `itsAge`.

Linia 25. zawiera definicję funkcji składowej `SetAge()`. Ta funkcja posiada parametr w postaci wartości całkowitej i przypisuje składowej `itsAge` jego wartość (linia 29.). Ponieważ jest składową klasy `Cat`, ma bezpośredni dostęp do jej zmiennych prywatnych i publicznych.

Linia 36. rozpoczyna definicję (czyli implementację) metody `Meow()` klasy `Cat`. Jest to jednoliniowa funkcja wypisująca na ekranie słowo „Miaucz”, zakończone znakiem nowej linii. Pamiętaj, że znak `\n` powoduje przejście do nowej linii.

Linia 43. rozpoczyna ciało funkcji `main()`, czyli właściwy program. W tym przypadku funkcja `main()` nie posiada argumentów. W linii 45., funkcja `main()` deklaruje obiekt `Cat` o nazwie `Mruczek`. W linii 46. zmiennej `itsAge` tego obiektu jest przypisywana wartość 5 (poprzez użycie akcesora `SetAge()`). Zauważ, że wywołanie tej metody następuje dzięki użyciu nazwy obiektu (`Mruczek`), po której zastosowano operator kropki (`.`) i nazwę metody (`SetAge()`). W podobny sposób wywoływane są wszystkie inne metody wszystkich klas.

Linia 47. wywołuje funkcję składową `Meow()`, zaś w linii 48. za pomocą akcesora `GetAge()`, wypisywany jest komunikat. Linia 50. ponownie wywołuje funkcję `Meow()`.

Konstruktory i destruktory

Istnieją dwa sposoby definiowania zmiennej całkowitej. Można zdefiniować zmienną, a następnie, w dalszej części programu, przypisać jej wartość. Na przykład:

```
int Weight;  // definiujemy zmienną
...          // tu inny kod
Weight = 7;  // przypisujemy jej wartość
```

Możemy też zdefiniować zmienną i jednocześnie zainicjalizować ją. Na przykład:

```
int Weight = 7;  // definiujemy i inicjalizujemy wartością 7
```

Inicjalizacja łączy w sobie definiowanie zmiennej oraz początkowe przypisanie wartości. Nic nie stoi na przeszkodzie temu, by zmienić później wartość zmiennej. Inicjalizacja powoduje tylko, że zmienna nigdy nie będzie pozbawiona sensownej wartości.

W jaki sposób zainicjalizować składowe klasy? Klasy posiadają specjalne funkcje składowe, zwane konstruktorami. *Konstruktor* (ang. *constructor*) może w razie potrzeby posiadać parametry, ale nie może zwracać wartości — nawet typu `void`. Konstruktor jest metodą klasy o takiej samej nazwie, jak nazwa klasy.

Gdy zadeklarujesz konstruktor, powinieneś także zadeklarować *destruktor* (ang. *destructor*). Konstruktor tworzy i inicjalizuje obiekt danej klasy, zaś destruktor porządkuje obiekt i zwalnia pamięć, którą mogłeś w niej zaalokować. Destruktor zawsze nosi nazwę klasy, poprzedzoną znakiem tyldy (~). Destruktory nie mają argumentów i nie zwracają wartości. Dlatego deklaracja destruktora klasy `Cat` ma następującą postać:

```
~Cat();
```

Domyślne konstruktory i destruktory

Jeśli nie zadeklarujesz konstruktora lub destruktora, zrobi to za ciebie kompilator.

Istnieje wiele rodzajów konstruktorów; niektóre z nich posiadają argumenty, inne nie. Konstruktor, którego można wywołać bez żadnych argumentów, jest nazywany konstruktorem domyślnym. Istnieje tylko jeden rodzaj destruktora. On także nie posiada argumentów.

Jeśli nie stworzysz konstruktora lub destruktora, kompilator stworzy je za ciebie. Konstruktor dostarczany przez kompilator jest konstruktorem domyślnym — czyli konstruktorem bez argumentów. Taki konstruktor domyślny możesz stworzyć samodzielnie.

Stworzone przez kompilator domyślny konstruktor i destruktor nie mają żadnych argumentów, a na dodatek w ogóle nic nie robią!

Użycie domyślnego konstruktora

Do czego może przydać się konstruktor, który nic nie robi? Jest to problem techniczny: wszystkie obiekty muszą być konstruowane i niszczone, dlatego w odpowiednich momentach wywoływane są te nic nie robiące funkcje. Aby móc zadeklarować obiekt bez przekazywania parametrów, na przykład

```
Cat Filemon;  // Filemon nie ma parametrów
```

musisz posiadać konstruktor w postaci

```
Cat();
```

Gdy definiujesz obiekt klasy, wywoływany jest konstruktor. Gdyby konstruktor klasy `Cat` miał dwa parametry, mógłbyś zdefiniować obiekt `Cat`, pisząc

```
Cat Mruczek (5, 7);
```

Gdyby konstruktor miał jeden parametr, napisałbyś

```
Cat Mruczek (3);
```

W przypadku, gdy konstruktor nie ma żadnych parametrów (gdy jest konstruktorem *domyślnym*), możesz opuścić nawiasy i napisać

```
Cat Mruczek;
```

Jest to wyjątek od reguły, zgodnie z którą wszystkie funkcje wymagają zastosowania nawiasów, nawet jeśli nie mają parametrów. Właśnie dlatego możesz napisać:

```
Cat Mruczek;
```

Jest to interpretowane jako wywołanie konstruktora domyślnego. Nie dostarczamy mu parametrów i pomijamy nawiasy.

Zwróć uwagę, że nie musisz używać domyślnego konstruktora dostarczanego przez kompilator. Zawsze możesz napisać własny konstruktor domyślny — tj. konstruktor bez parametrów. Możesz zastosować w nim ciało funkcji, w którym możesz zainicjalizować obiekt.

Zgodnie z konwencją, gdy deklarujesz konstruktor, powinieneś także zadeklarować destruktor, nawet jeśli nie robi on niczego. Nawet jeśli destruktor domyślny będzie działał poprawnie, nie zaszkodzi zadeklarować własnego. Dzięki niemu kod staje się bardziej przejrzysty.

Listing 6.4 zawiera nową wersję klasy `Cat`, w której do zainicjalizowania obiektu kota użyto konstruktora. Wiek kota został ustawiony zgodnie z wartością otrzymaną jako parametr konstruktora.

Listing 6.4. *Użycie konstruktora i destruktora*

```
// Demonstruje deklarowanie konstruktora
// i destruktora dla klasy Cat
// Domyślny konstruktor został stworzony przez programistę

#include <iostream>           // dla cout

class Cat                     // początek deklaracji klasy
{
public:                       // początek sekcji publicznej
    Cat(int initialAge);      // konstruktor
    ~Cat();                   // destruktor
    int GetAge();             // akcesor
    void SetAge(int age);     // akcesor
    void Meow();
private:                      // początek sekcji prywatnej
    int itsAge;               // zmienna składowa
};

// konstruktor klasy Cat
Cat::Cat(int initialAge)
{
   itsAge = initialAge;
```

```
22:  }
23:
24:  Cat::~Cat()                    // destruktor, nic nie robi
25:  {
26:  }
27:
28:  // GetAge, publiczny akcesor
29:  // zwraca wartość składowej itsAge
30:  int Cat::GetAge()
31:  {
32:      return itsAge;
33:  }
34:
35:  // definicja SetAge, akcesora
36:  // publicznego
37:
38:  void Cat::SetAge(int age)
39:  {
40:      // ustawia zmienną składową itsAge
41:      // zgodnie z wartością przekazaną w parametrze age
42:      itsAge = age;
43:  }
44:
45:  // definicja metody Meow
46:  // zwraca: void
47:  // parametery: brak
48:  // działanie: wypisuje na ekranie słowo "miauczy"
49:  void Cat::Meow()
50:  {
51:      std::cout << "Miauczy.\n";
52:  }
53:
54:  // tworzy kota, ustawia jego wiek, sprawia,
55:  // że miauczy, wypisuje jego wiek i ponownie miauczy.
56:  int main()
57:  {
58:      Cat Mruczek(5);
59:      Mruczek.Meow();
60:      std::cout << "Mruczek jest kotem i ma " ;
61:      std::cout << Mruczek.GetAge() << " lat.\n";
62:      Mruczek.Meow();
63:      Mruczek.SetAge(7);
64:      std::cout << "Teraz Mruczek ma " ;
65:      std::cout << Mruczek.GetAge() << " lat.\n";
66:      return 0;
67:  }
```

wynik

```
Miauczy.
Mruczek jest kotem i ma 5 lat.
Miauczy.
Teraz Mruczek ma 7 lat.
```

analiza

Listing 6.4 przypomina listing 6.3, jednak w linii 9. dodano konstruktor, posiadający argument w postaci wartości całkowitej. Linia 10. deklaruje destruktor, który nie posiada parametrów. Destruktory nigdy nie mają parametrów, zaś destruktory i konstruktory nie zwracają żadnych wartości — nawet typu `void`.

Linie od 19. do 22. zawierają implementację konstruktora. Jest ona podobna do implementacji akcesora `SetAge()`. Konstruktor nie zwraca wartości.

Linie od 24. do 26. przedstawiają implementację destruktora `~Cat()`. Ta funkcja nie robi nic, ale jeśli deklaracja klasy zawiera deklarację destruktora, zdefiniowany musi zostać wtedy także ten destruktor.

Linia 58. zawiera definicję obiektu `Mruczek`, stanowiącego egzemplarz klasy `Cat`. Do konstruktora obiektu `Mruczek` przekazywana jest wartość `5`. Nie ma potrzeby wywoływania funkcji `SetAge()`, gdyż `Mruczek` został stworzony z wartością `5` znajdującą się w zmiennej składowej `itsAge`, tak jak pokazano w linii 61. W linii 63. zmiennej `itsAge` obiektu `Mruczek` jest przypisywana wartość 7. Tę nową wartość wypisuje linia 65.

TAK	NIE
W celu zainicjalizowania obiektów używaj konstruktorów.	Pamiętaj, że konstruktory i destruktory nie mogą zwracać wartości. Pamiętaj, że destruktory nie mogą mieć parametrów.

Funkcje składowe const

Jeśli zadeklarujesz metodę klasy jako `const`, obiecujesz w ten sposób, że metoda ta nie zmieni wartości żadnej ze składowych klasy. Aby zadeklarować metodę w ten sposób, umieść słowo kluczowe `const` za nawiasami, lecz przed średnikiem. Pokazana poniżej deklaracja funkcji składowej `const` o nazwie `SomeFunction()` nie posiada argumentów i zwraca typ `void`:

```
void SomeFunction() const;
```

Wraz z modyfikatorem `const` często deklarowane są akcesory. Klasa `Cat` posiada dwa akcesory:

```
void SetAge(int anAge);
int GetAge();
```

Funkcja `SetAge()` nie może być funkcją `const`, gdyż modyfikuje wartość zmiennej składowej `itsAge`. Natomiast funkcja `GetAge()` może być `const`, gdyż nie modyfikuje wartości żadnej ze składowych klasy. Funkcja `GetAge()` po prostu zwraca bieżącą wartość składowej `itsAge`. Zatem deklaracje tych funkcji można przepisać następująco:

```
void SetAge(int anAge);
int GetAge() const;
```

Gdy zadeklarujesz funkcję jako `const`, zaś implementacja tej funkcji modyfikuje obiekt poprzez modyfikację wartości którejkolwiek ze składowych, kompilator zgłosi

błąd. Na przykład, gdy napiszesz funkcję `GetAge()` w taki sposób, że będziesz zapamiętywał ilość zapytań o wiek kota, spowodujesz błąd kompilacji. Jest to spowodowane tym, że wywołując tę metodę, modyfikujesz zawartość obiektu `Cat`.



Deklaruj funkcje jako `const` wszędzie, gdzie to jest możliwe. Deklaruj je tam, gdzie nie przewidujesz modyfikowania obiektu. Kompilator może w ten sposób pomóc ci w wykryciu błędów w programie; tak jest szybciej i dokładniej.

Deklarowanie funkcji jako `const` wszędzie tam, gdzie jest to możliwe, należy do tradycji programistycznej. Za każdym razem, gdy to zrobisz, umożliwisz kompilatorowi wykrycie pomyłki, zanim stanie się ona błędem, który ujawni się już podczas działania programu.

Interfejs a implementacja

Jak wiesz, klienty są tymi elementami programu, które tworzą i wykorzystują obiekty twojej klasy. Publiczny interfejs swojej klasy — deklarację klasy — możesz traktować jako kontrakt z tymi klientami. Ten kontrakt informuje, jak zachowuje się dana klasa.

Na przykład, w deklaracji klasy `Cat`, stworzyłeś kontrakt informujący, że wiek każdego kota może być zainicjalizowany w jego konstruktorze, modyfikowany za pomocą akcesora `SetAge()` oraz odczytywany za pomocą akcesora `GetAge()`. Oprócz tego obiecujesz, że każdy kot może miauczeć (funkcją `Meow()`). Zwróć uwagę, że w publicznym interfejsie nie ma ani słowa o zmiennej składowej `itsAge`; jest to szczegół implementacji, który nie stanowi elementu kontraktu. Na żądanie dostarczysz wieku (`GetAge()`) i ustawisz go (`SetAge()`), ale sam mechanizm (`itsAge`) jest niewidoczny.

Gdy uczynisz funkcję `GetAge()`funkcją `const` — a powinieneś to zrobić — kontrakt obiecuje także, że funkcja `GetAge()` nie modyfikuje obiektu `Cat`, dla którego jest wywołana.

C++ jest językiem zapewniającym silną kontrolę typów, co oznacza, że kompilator wymusza przestrzeganie kontraktu, zgłaszając błędy kompilacji za każdym razem, gdy naruszysz reguły tego kontraktu. Listing 6.5 przedstawia program, który nie skompiluje się z powodu naruszenia ustaleń takiego kontraktu.



Listing 6.5 nie skompiluje się!

Listing 6.5. *Przykład naruszenia ustaleń interfejsu*

```
0:  // Demonstruje błędy kompilacji
1:  // Ten program się nie kompiluje!
2:
3:  #include <iostream>          // dla cout
4:
5:  class Cat
```

```
 6: {
 7: public:
 8:     Cat(int initialAge);
 9:     ~Cat();
10:     int GetAge() const;          // akcesor typu const
11:     void SetAge (int age);
12:     void Meow();
13: private:
14:     int itsAge;
15: };
16:
17: // konstruktor klasy Cat,
18: Cat::Cat(int initialAge)
19: {
20:     itsAge = initialAge;
21:     std::cout << "Konstruktor klasy Cat\n";
22: }
23:
24: Cat::~Cat()                  // destruktor, nic nie robi
25: {
26:     std::cout << "Destruktor klasy Cat\n";
27: }
28: // GetAge, funkcja const,
29: // ale narusza zasadę const!
30: int Cat::GetAge() const
31: {
32:     return (itsAge++);         // narusza const!
33: }
34:
35: // definicja SetAge, publicznego
36: // akcesora
37:
38: void Cat::SetAge(int age)
39: {
40:     // ustawia zmienną składową itsAge
41:     // zgodnie z wartością przekazaną w parametrze age
42:     itsAge = age;
43: }
44:
45: // definicja metody Meow
46: // zwraca: void
47: // parametery: brak
48: // działanie: wypisuje na ekranie słowo "miauczy"
49: void Cat::Meow()
50: {
51:     std::cout << "Miauczy.\n";
52: }
53:
54: // demonstruje różne naruszenia reguł interfejsu
55: // oraz wynikające z tego błędy kompilatora
56: int main()
57: {
58:     Cat Mruczek;          // nie pasuje do deklaracji
59:     Mruczek.Meow();
60:     Mruczek.Bark();       // Nie, głuptasie, koty nie szczekają.
61:     Mruczek.itsAge = 7;   // itsAge jest składową prywatną
62:     return 0;
63: }
```



Program w przedstawionej powyżej postaci się nie kompiluje, więc nie ma wyników działania.

Pisanie go było dość zabawne, ponieważ zawiera tak dużo błędów.

Linia 10 deklaruje funkcję `GetAge()` jako akcesor typu `const` — i tak powinno być. Jednak w ciele funkcji `GetAge()`, w linii 32., inkrementowana jest zmienna składowa `itsAge`. Ponieważ ta metoda została zadeklarowana jako `const`, nie może zmieniać wartości tej zmiennej. Dlatego podczas kompilacji programu zostanie to zgłoszone jako błąd.

W linii 12., funkcja `Meow()` nie jest zadeklarowana jako `const`. Choć nie jest to błędem, stanowi zły obyczaj. Należałoby wziąć pod uwagę, że ta metoda nie modyfikuje zmiennych składowych klasy. Dlatego funkcja `Meow()` powinna być funkcją `const`.

Linia 58. pokazuje definicję obiektu `Mruczek` klasy `Cat`. W tym programie klasa `Cat` posiada konstruktor, który wymaga podania argumentu, będącego wartością całkowitą. Oznacza to, że musisz taki argument przekazać. Ponieważ w linii 58. nie występuje argument konstruktora, kompilator zgłosi błąd.



Jeśli stworzysz *jakikolwiek* konstruktor, kompilator zrezygnuje z dostarczenia swojego konstruktora domyślnego. Gdy stworzysz konstruktor wymagający parametru, nie będziesz miał konstruktora domyślnego, chyba że stworzysz go sam.

Linia 60. zawiera wywołanie metody `Bark()` dla obiektu `Mruczek`. Metoda `Bark()` nie została zadeklarowana, więc jest niedozwolona.

Linia 61. zawiera przypisanie wartości `7` do zmiennej `itsAge`. Ponieważ `itsAge` jest składową prywatną, kompilator zgłosi błąd kompilacji.

Po co używać kompilatora do wykrywania błędów?

Gdyby można było tworzyć programy w stu procentach pozbawione błędów, byłoby cudowanie, jednak tylko bardzo niewielu programistów jest w stanie tego dokonać. Wielu programistów opracowało jednak system pozwalający zminimalizować ilość błędów przez wczesne ich wykrycie i poprawienie.

Choć błędy kompilatora są irytujące i stanowią dla programisty przekleństwo, jednak są czymś dużo lepszym niż opisana dalej alternatywa. Język o słabej kontroli typów umożliwia naruszanie zasad kontraktu bez słowa sprzeciwu ze strony kompilatora, jednak program może załamać się w trakcie działania — na przykład wtedy, gdy pracuje z nim twój szef.

Błędy czasu kompilacji — tj. błędy wykryte podczas kompilowania programu — są zdecydowanie lepsze niż błędy czasu działania — tj. błędy wykryte podczas działania programu. Są lepsze, gdyż dużo łatwiej i precyzyjniej można określić ich przyczynę. Może się zdarzyć, że program zostanie wykonany wielokrotnie bez wykonania wszystkich istniejących ścieżek wykonania kodu. Dlatego błąd czasu działania może przez dłuższy czas pozostać niezauważony. Błędy kompilacji są wykrywane podczas każdej kompilacji, są więc dużo łatwiejsze do zidentyfikowania i poprawienia. Celem dobrego programowania jest ochrona przed pojawianiem się błędów czasu działania. Jedną ze znanych i sprawdzonych technik jest wykorzystanie kompilatora do wykrycia pomyłek już na wczesnym etapie tworzenia programu.

Gdzie umieszczać deklaracje klasy i definicje metod

Każda funkcja, którą zadeklarujesz dla klasy, musi posiadać definicję. Definicja jest nazywana także implementacją funkcji. Podobnie jak w przypadku innych funkcji, definicja metody klasy posiada nagłówek i ciało.

Definicja musi znajdować się w pliku, który może zostać znaleziony przez kompilator. Większość kompilatorów C++ wymaga, by taki plik miał rozszerzenie *.c* lub *.cpp*. W tej książce korzystamy z rozszerzenia *.cpp*, ale aby mieć pewność, sprawdź, czego oczekuje twój kompilator.



Wiele kompilatorów zakłada, że pliki z rozszerzeniem *.c* są programami C, zaś pliki z rozszerzeniem *.cpp* są programami C++. Możesz używać dowolnego rozszerzenia, ale rozszerzenie *.cpp* wyeliminuje ewentualne nieporozumienia.

W pliku, w którym umieszczasz implementację funkcji, możesz umieścić również jej deklarację, ale nie należy to do dobrych obyczajów. Zgodnie z konwencją zaadoptowaną przez większość programistów, deklaracje umieszcza się w tak zwanych plikach nagłówkowych, zwykle posiadających tę samą nazwę, lecz z rozszerzeniem *.h*, *.hp* lub *.hpp*. W tej książce dla plików nagłówkowych stosujemy rozszerzenie *.hpp*, ale sprawdź w swoim kompilatorze, jakie rozszerzenie powinieneś stosować.

Na przykład, deklarację klasy `Cat` powinieneś umieścić w pliku o nazwie *CAT.hpp*, zaś definicję metod tej klasy w pliku o nazwie *CAT.cpp*. Następnie powinieneś dołączyć do pliku *.cpp* plik nagłówkowy, poprzez umieszczenie na początku pliku *CAT.cpp* następującej dyrektywy:

```
#include "Cat.hpp"
```

Informuje ona kompilator, by wstawił w tym miejscu zawartość pliku *CAT.hpp* tak, jakbyś ją wpisał ręcznie. Uwaga: niektóre kompilatory nalegają, by wielkość liter w nazwie pliku w dyrektywie `#include` zgadzała się z wielkością liter w nazwie pliku na dysku.



Deklaracja klasy mówi kompilatorowi, czym jest ta klasa, jakie dane zawiera oraz jakie funkcje posiada. Deklaracja klasy jest nazywana jej interfejsem, gdyż informuje kompilator w jaki sposób ma z nią współdziałać. Ten interfejs jest zwykle przechowywany w pliku *.hpp*, często nazywanym plikiem nagłówkowym.

Definicja funkcji mówi kompilatorowi, jak działa dana funkcja. Definicja funkcji jest nazywana implementacją metody klasy i jest przechowywana w pliku *.cpp*. Szczegóły dotyczące implementacji klasy należą wyłącznie do jej autora. Klienty klasy — tj. części programu używające tej klasy — nie muszą, ani nie powinny wiedzieć, jak zaimplementowane zostały funkcje.

Dlaczego masz się trudzić, rozdzielając program na pliki *.hpp* i *.cpp*, skoro i tak plik *.hpp* jest wstawiany do pliku *.cpp*? W większości przypadków klienty klasy nie dbają o szczegóły jej implementacji. Odczytanie pliku nagłówkowego daje im wystarczającą ilość informacji, by zignorować plik implementacji. Poza tym, ten sam plik *.hpp* możesz dołączać do wielu różnych plików *.cpp*.

Implementacja inline

Możesz poprosić kompilator, by uczynił zwykłą funkcję funkcją `inline`, funkcjami `inline` mogą stać się również metody klasy. W tym celu należy umieścić słowo kluczowe `inline` przed typem zwracanej wartości. Na przykład, implementacja `inline` funkcji `GetWeight()` wygląda następująco:

```
inline int Cat::GetWeight()
{
    return itsWeight;  // zwraca daną składową itsWeight
}
```

Definicję funkcji można także umieścić w deklaracji klasy, co automatycznie sprawia, że ta funkcja staje się funkcją `inline`. Na przykład:

```
class Cat
{
public:
    int GetWeight() { return itsWeight; }  // inline
    void SetWeight(int aWeight);
};
```

Zwróć uwagę na składnię definicji funkcji `GetWeight()`. Ciało funkcji `inline` zaczyna się natychmiast po deklaracji metody klasy; po nawiasach nie występuje średnik. Podobnie jak w innych funkcjach, definicja zaczyna się od otwierającego nawiasu klamrowego i kończy zamykającym nawiasem klamrowym. Jak zwykle, białe spacje nie mają znaczenia; możesz zapisać tę deklarację jako:

```
class Cat
{
public:
    int GetWeight() const
    {
        return itsWeight;
    }  // inline
    void SetWeight(int aWeight);
};
```

Listingi 6.6 i 6.7 odtwarzają klasę `Cat`, tym razem jednak deklaracja klasy została umieszczona w pliku *CAT.hpp*, zaś jej definicja w pliku *CAT.cpp*. Oprócz tego, na listingu 6.7 akcesor `Meow()` został zadeklarowany jako funkcja `inline`.

Listing 6.6. *Deklaracja klasy Cat w pliku CAT.hpp*

```
0:  #include <iostream>
1:  class Cat
2:  {
3:  public:
4:      Cat (int initialAge);
5:      ~Cat();
6:      int GetAge() const { return itsAge;}        // inline!
7:      void SetAge (int age) { itsAge = age;}        // inline!
8:      void Meow() const { std::cout << "Miauczy.\n";}  // inline!
9:  private:
10:     int itsAge;
11: };
```

Listing 6.7. *Implementacja klasy Cat w pliku CAT.cpp*

```
0:  // Demonstruje funkcje inline
1:  // oraz dołączanie pliku nagłówkowego
2:
3:  // pamiętaj o włączeniu plików nagłówkowych!
4:  #include "cat.hpp"
5:
6:
7:  Cat::Cat(int initialAge)   //konstruktor
8:  {
9:      itsAge = initialAge;
10: }
11:
12: Cat::~Cat()             //destruktor, nic nie robi
13: {
14: }
15:
16: // tworzy kota, ustawia jego wiek, sprawia
17: // że miauczy, wypisuje jego wiek i ponownie miauczy.
18: int main()
19: {
20:     Cat Mruczek(5);
21:     Mruczek.Meow();
22:     std::cout << "Mruczek jest kotem i ma " ;
23:     std::cout << Mruczek.GetAge() << " lat.\n";
24:     Mruczek.Meow();
25:     Mruczek.SetAge(7);
26:     std::cout << "Teraz Mruczek ma " ;
27:     std::cout << Mruczek.GetAge() << " lat.\n";
28:     return 0;
29: }
```

wynik

```
Miauczy.
Mruczek jest kotem i ma 5 lat.
Miauczy.
Teraz Mruczek ma 7 lat.
```

analiza Kod zaprezentowany na listingach 6.6 i 6.7 jest podobny do kodu z listingu 6.4, trzy metody zostały zadeklarowane w pliku deklaracji jako `inline`, a deklaracja została przeniesiona do pliku *CAT.hpp* (listing 6.6).

Funkcja `GetAge()` jest deklarowana w linii 6., gdzie znajduje się także jej implementacja. Linie 7. i 8. zawierają kolejne funkcje `inline`, jednak w stosunku do poprzednich, „zwykłych" implementacji, działanie tych funkcji nie zmienia się.

Linia 4. listingu 6.7 zawiera dyrektywę `#include "cat.hpp"`, która powoduje wstawienie do pliku zawartości pliku *CAT.hpp*. Dołączając plik *CAT.hpp*, informujesz prekompilator, by odczytał zawartość tego pliku i wstawił ją w miejscu wystąpienia dyrektywy `#include` (tak jakbyś, począwszy od linii 5, sam wpisał tę zawartość).

Ta technika umożliwia umieszczenie deklaracji w pliku innym niż implementacja, a jednocześnie zapewnienie kompilatorowi dostępu do niej. W programach C++ technika ta jest powszechnie wykorzystywana. Zwykle deklaracje klas znajdują się w plikach *.hpp*, które są dołączane do powiązanych z nimi plików *.cpp* za pomocą dyrektyw `#include`.

Linie od 18. do 29. stanowią powtórzenie funkcji `main()` z listingu 6.4. Oznacza to, że funkcje `inline` działają tak samo jak zwykłe funkcje.

Klasy, których danymi składowymi są inne klasy

Budowanie złożonych klas przez deklarowanie prostszych klas i dołączanie ich do deklaracji bardziej skomplikowanej klasy nie jest niczym niezwykłym. Na przykład, możesz zadeklarować klasę koła, klasę silnika, klasę skrzyni biegów itd., a następnie połączyć je w klasę „samochód". Deklaruje to relację posiadania. Samochód posiada silnik, koła i skrzynię biegów.

Weźmy inny przykład. Prostokąt składa się z odcinków. Odcinek jest zdefiniowany przez dwa punkty. Punkt jest zdefiniowany przez współrzędną x i współrzędną y. Listing 6.8 przedstawia pełną deklarację klasy `Rectangle` (prostokąt), która może wystąpić w pliku *RECTANGLE.hpp*. Ponieważ prostokąt jest zdefiniowany jako cztery odcinki łączące cztery punkty, zaś każdy punkt odnosi się do współrzędnej w układzie, najpierw zadeklarujemy klasę `Point` (punkt) jako przechowującą współrzędne x oraz y punktu. Listing 6.9 zawiera implementacje obu klas.

Listing 6.8. *Deklarowanie kompletnej klasy*

```
0:  // początek Rect.hpp
1:
2:  #include <iostream>
3:  class Point     // przechowuje współrzędne x,y
4:  {
5:  // bez konstruktora, używa domyślnego
6:  public:
7:      void SetX(int x) { itsX = x; }
```

```
 8:      void SetY(int y) { itsY = y; }
 9:      int GetX()const { return itsX;}
10:      int GetY()const { return itsY;}
11:  private:
12:      int itsX;
13:      int itsY;
14:  };    // koniec deklaracji klasy Point
15:
16:
17:  class  Rectangle
18:  {
19:  public:
20:      Rectangle (int top, int left, int bottom, int right);
21:      ~Rectangle () {}
22:
23:      int GetTop() const { return itsTop; }
24:      int GetLeft() const { return itsLeft; }
25:      int GetBottom() const { return itsBottom; }
26:      int GetRight() const { return itsRight; }
27:
28:      Point  GetUpperLeft() const { return itsUpperLeft; }
29:      Point  GetLowerLeft() const { return itsLowerLeft; }
30:      Point  GetUpperRight() const { return itsUpperRight; }
31:      Point  GetLowerRight() const { return itsLowerRight; }
32:
33:      void SetUpperLeft(Point Location)  {itsUpperLeft = Location;}
34:      void SetLowerLeft(Point Location)  {itsLowerLeft = Location;}
35:      void SetUpperRight(Point Location)  {itsUpperRight = Location;}
36:      void SetLowerRight(Point Location)  {itsLowerRight = Location;}
37:
38:      void SetTop(int top) { itsTop = top; }
39:      void SetLeft (int left) { itsLeft = left; }
40:      void SetBottom (int bottom) { itsBottom = bottom; }
41:      void SetRight (int right) { itsRight = right; }
42:
43:      int GetArea() const;
44:
45:  private:
46:      Point  itsUpperLeft;
47:      Point  itsUpperRight;
48:      Point  itsLowerLeft;
49:      Point  itsLowerRight;
50:      int    itsTop;
51:      int    itsLeft;
52:      int    itsBottom;
53:      int    itsRight;
54:  };
55:  // koniec Rect.hpp
```

Listing 6.9. *RECTANGLE.cpp*

```
 0:  // początek rect.cpp
 1:
 2:  #include "rect.hpp"
 3:  Rectangle::Rectangle(int top, int left, int bottom, int right)
```

```
 4:  {
 5:      itsTop = top;
 6:      itsLeft = left;
 7:      itsBottom = bottom;
 8:      itsRight = right;
 9:
10:      itsUpperLeft.SetX(left);
11:      itsUpperLeft.SetY(top);
12:
13:      itsUpperRight.SetX(right);
14:      itsUpperRight.SetY(top);
15:
16:      itsLowerLeft.SetX(left);
17:      itsLowerLeft.SetY(bottom);
18:
19:      itsLowerRight.SetX(right);
20:      itsLowerRight.SetY(bottom);
21:  }
22:
23:
24:  // oblicza obszar prostokąta przez obliczenie
25:  // i pomnożenie szerokości i wysokości
26:  int Rectangle::GetArea() const
27:  {
28:      int Width = itsRight-itsLeft;
29:      int Height = itsTop - itsBottom;
30:      return (Width * Height);
31:  }
32:
33:  int main()
34:  {
35:      //inicjalizuje lokalną zmienną typu Rectangle
36:      Rectangle MyRectangle (100, 20, 50, 80 );
37:
38:      int Area = MyRectangle.GetArea();
39:
40:      std::cout << "Obszar: " << Area << "\n";
41:      std::cout << "Wsp. X lewego gornego rogu: ";
42:      std::cout << MyRectangle.GetUpperLeft().GetX();
43:      return 0;
44:  }
```




```
Obszar: 3000
Wsp. X lewego gornego rogu: 20
```



Linie od 3. do 14. listingu 6.8 deklarują klasę `Point` (punkt), która służy do przechowywania współrzędnych `x` i `y` określonego punktu rysunku. W tym programie nie wykorzystujemy należycie klasy `Point`. Jej zastosowania wymagają jednak inne metody rysunkowe.



Gdy nadasz klasie nazwę `Rectangle`, niektóre kompilatory zgłoszą błąd. W takim przypadku po prostu zmień nazwę klasy na `myRectangle`.

W deklaracji klasy Point, w liniach 12. i 13., zadeklarowaliśmy dwie zmienne składowe (itsX oraz itsY). Te zmienne przechowują współrzędne punktu. Zakładamy, że współrzędna x rośnie w prawo, a współrzędna y w górę. Istnieją także inne systemy. W niektórych programach okienkowych współrzędna y rośnie „w dół" okna.

Klasa Point używa akcesorów inline, zwracających i ustawiających współrzędne X i Y punktu. Te akcesory zostały zadeklarowane w liniach od 7. do 10. Punkty używają konstruktora i destruktora domyślnego. W związku z tym ich współrzędne trzeba ustawiać jawnie.

Linia 17. rozpoczyna deklarację klasy Rectangle (prostokąt). Klasa ta kłada się z czterech punktów reprezentujących cztery narożniki prostokąta.

Konstruktor klasy Rectangle (linia 20.) otrzymuje cztery wartości całkowite, top (górna), left (lewa), bottom (dolna) oraz right (prawa). Do czterech zmiennych składowych (listing 6.9) kopiowane są cztery parametry konstruktora i tworzone są cztery punkty.

Oprócz standardowych akcesorów, klasa Rectangle posiada funkcję GetArea() (pobierz obszar), zadeklarowaną w linii 43. Zamiast przechowywać obszar w zmiennej, funkcja GetArea() oblicza go w liniach od 28. do 30. listingu 6.9. W tym celu oblicza szerokość i wysokość prostokąta, następnie mnoży je przez siebie.

Uzyskanie współrzędnej x lewego górnego wierzchołka prostokąta wymaga dostępu do punktu UpperLeft (lewy górny) i zapytania o jego współrzędną X. Ponieważ funkcja GetUpperLeft() jest funkcją klasy Rectangle, może ona bezpośrednio odwoływać się do prywatnych danych tej klasy, włącznie ze zmienną (itsUpperLeft). Ponieważ itsUpperLeft jest obiektem klasy Point, a zmienna itsX tej klasy jest prywatna, funkcja GetUpperLeft() nie może odwoływać się do niej bezpośrednio. Zamiast tego, w celu uzyskania tej wartości musi użyć publicznego akcesora GetX().

Linia 33. listingu 6.9 stanowi początek ciała programu. Pamięć nie jest alokowana aż do linii 36.; w obszarze tym nic się nie dzieje. Jedyna rzecz, jaką zrobiliśmy, to poinformowanie kompilatora, jak ma stworzyć punkt i prostokąt (gdyby były potrzebne w przyszłości).

W linii 36. definiujemy obiekt typu Rectangle, przekazując mu wartości Top, Left, Bottom oraz Right.

W linii 38. tworzymy lokalną zmienną Area (obszar) typu int. Ta zmienna przechowuje obszar stworzonego przez nas prostokąta. Zmienną Area inicjalizujemy za pomocą wartości zwróconej przez funkcję GetArea() klasy Rectangle.

Klient klasy Rectangle może stworzyć obiekt tej klasy i uzyskać jego obszar, nie znając nawet implementacji funkcji GetArea().

Plik *RECT.hpp* został przedstawiony na listingu 6.8. Obserwując plik nagłówkowy, który zawiera deklarację klasy Rectangle, programista może wysnuć wniosek, że funkcja

`GetArea()` zwraca wartość typu `int`. Sposób, w jaki funkcja `GetArea()` uzyskuje tę wartość, nie interesuje klientów klasy `Rectangle`. Autor klasy `Rectangle` mógłby zmienić funkcję `GetArea()`; nie wpłynęłoby to na programy, które z niej korzystają.

Często zadawane pytanie

Jaka jest różnica pomiędzy deklaracją a definicją?

Odpowiedź

Deklaracja wprowadza nową nazwę, lecz nie alokuje pamięci; dokonuje tego definicja.

Wszystkie deklaracje (z kilkoma wyjątkami) są także definicjami. Najważniejszym wyjątkiem jest deklaracja funkcji globalnej (prototyp) oraz deklaracja klasy (zwykle w pliku nagłówkowym).

Struktury

Bardzo bliskim kuzynem słowa kluczowego `class` jest słowo kluczowe `struct`, używane do deklarowania struktur. W C++ struktura jest odpowiednikiem klasy, ale wszystkie jej składowe są domyślnie publiczne. Możesz zadeklarować strukturę dokładnie tak, jak klasę; możesz zastosować w niej te same zmienne i funkcje składowe. Gdy przestrzegasz jawnego deklarowania publicznych i prywatnych sekcji klasy, nie ma żadnej różnicy pomiędzy klasą a strukturą.

Spróbuj wprowadzić do listingu 6.8 następujące zmiany:

- w linii 3., zmień `class Point` na `struct Point`,
- w linii 17., zmień `class Rectangle` na `struct Rectangle`.

Następnie skompiluj i uruchom program. Otrzymane wyniki nie powinny się od siebie różnić.

Dlaczego dwa słowa kluczowe spełniają tę samą funkcję

Prawdopodobnie zastanawiasz się, dlaczego dwa słowa kluczowe spełniają tę samą funkcję. Przyczyn należy szukać w historii języka. Język C++ powstawał jako rozszerzenie języka C. Język C posiada struktury, ale nie posiadają one metod. Bjarne Stroustrup, twórca języka C++, rozbudował struktury, ale zmienił ich nazwę na klasy, odzwierciedlając w ten sposób ich nowe, rozszerzone możliwości.

TAK
Umieszczaj deklarację klasy w pliku *.hpp*, zaś funkcje składowe definiuj w pliku *.cpp*.
Używaj `const` wszędzie tam, gdzie jest to możliwe.
Zanim przejdziesz dalej, postaraj się dokładnie zrozumieć zasady działania klasy.