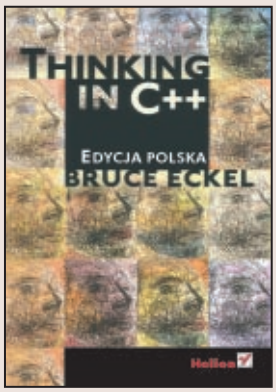
Thinking in C++. Edycja polska



Autor: Bruce Eckel
Tłumaczenie: Piotr Imiela
ISBN: 83-7197-709-3
Tytuł oryginału: Thinking in C++
Format: B5, stron: 642
Przykłady na ftp: 247 kB

C++ to jeden z najpopularniejszych języków programowania, w którym napisano szereg profesjonalnych aplikacji, a nawet kilka systemów operacyjnych. Nie bez przyczyny uważany jest on za trudny do opanowana, stanowiąc poważne wyzwanie zarówno dla programistów, jak dla autorów podręczników.

Wieloletnie doświadczenie w nauczaniu programowania Bruce'a Eckela gwarantuje, że po przeczytaniu tej książki będziesz posługiwał się C++ tak sprawnie, jak językiem polskim. Bruce Eckel to nie tylko autor bestsellerowych podręczników takich jak: Thinking in Java, ale również członek komitetu standaryzującego C++ i szef firmy zajmujący się szkoleniem programistów. Tworzone przez niego kursy programowania uznawane są za jedne z najlepszych na świecie.

- poznasz podstawowe i zaawansowane techniki programowania w C++
- krok po kroku prześledzisz konstrukcję języka
- nauczysz się diagnozować i rozwiązywać problemy w C++
- zwięzłe, łatwe do zrozumienia przykłady zilustrują przedstawiane zagadnienia
- ćwiczenia utrwalą zdobyte umiejętności na każdym etapie nauki
- kod źródłowy zawarty w książce zgodnie z wieloma kompilorami (w tym z darmowym kompilatorem GNU C++)

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

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 obiektówprzy 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 kompilacji ... 69
Narzędzia do rozłącznej kompilacji ... 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 za 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
Używanie 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 pułapkizwiązane 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 programówza 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 kompilacją ... 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
Tworzenieabstrakcyjnych typów danych 195
Szczegóły dotyczące obiektów 196
Zasady używania plików nagłówkowych 197
Znaczenie plików nagłówkowych 198
Problem wielokrotnych deklaracji 199
Dyrektywy preprocesora #define, #ifdef i #endif 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 protected 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 kompilacji 224
Podsumowanie 226
Ćwiczenia 227

Rozdział 6. Inicjalizacjai końcowe porządki 229
Konstruktor gwarantuje inicjalizację 230
Destruktor gwarantuje sprzątanie 232
Eliminacja bloku definicji 233
Pętle for 235
Przydzielanie pamięci 236
Klasa Stash z konstruktorami i destruktorami 237
Klasa Stack z konstruktorami i destruktorami 240
Inicjalizacja agregatowa 242
Konstruktory 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 kompilacji, 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 wewnątrz 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 destruktorach ... 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 pamięcidla 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 kopiujący ... 359
Wskaźniki w C++ ... 359
Referencje w C++ ... 360
Wykorzystanie referencji w funkcjach ... 361
Wskazówki dotyczące przekazywania argumentów ... 363
Konstruktor kopiujący ... 363
Przekazywanie i zwracanie przez wartość ... 364
Konstrukcja za pomocą konstruktora kopiującego ... 369
Domyślny konstruktor kopiujący ... 374
Możliwości zastąpienia konstruktora kopiującego ... 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 klasy ... 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 dziedziczone ... 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 górę ... 492
Dlaczego „rzutowanie w górę”? ... 494
Rzutowanie w górę a konstruktor kopiujący ... 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 górę ... 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 wirtualnychwewnątrz konstruktorów ... 532
Destruktory i wirtualne destruktory ... 533
Czysto wirtualne destruktory ... 535
Wirtualne wywołania w destruktorach ... 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
Rozwiązanie 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 odbieranieprawa własności ... 570
Przechowywanie obiektówjako 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

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 wywoływane w 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 rozwiązanie 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 charakter jawny.

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 mogą łatwo 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 kompilacji. 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 składową bieżącej 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 **Bob**a!" i spodziewając 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 ją 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 jej, zanim kompilator napotka pełną specyfikację klasy **Y**, nastąpiłoby zgłoszenie błędu. Jest to cecha zapewniająca bezpieczeństwo i spójność, a także zapobiegająca błędom.

Zwróć 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**, nadającą 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 całą pamięć odpowiednią wartością (podaną jako drugi argument), wypełniając **n** 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 widnieć 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. Odnosi się to przede wszystkim do układu pamięci obiektów tej struktury, to znaczy określenia, jak w pamięci przydzielonej obiektowi 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 takiej 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 języka nie 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 kapsułkowaniem[1]) 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 martwiąc 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 mogą jedynie 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 Stroustrupa 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 zaopatrzonymi 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 wstecznej zgodności języka C++ z językiem C.

Wiele osób preferuje styl tworzenia klas bliższy strukturom niż klasom. Nie przywiązują one wagi do „domyślnie prywatnego" zachowania klas, rozpoczynając deklaracje klas od ich elementów publicznych:

```
class X {
public:
  void funkcja_interfejsu();
private:
  void funkcja_prywatna();
  int wewnetrzna_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 dlatego, 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 wewnetrzna_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 są danymi globalnymi, a które — składowymi klasy.

Modyfikacja programu Stash, wykorzystująca kontrolę dostępu

Modyfikacja programu z rozdziału 4., dokonana w taki sposób, by używał on klas oraz kontroli dostępu, wydaje się racjonalna. Zwróć uwagę na to, 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* data;
    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.

Zwróć uwagę na to, że interfejs prezentowany klientowi-programiście 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 kompilacji), 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 powtórną kompilację 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 wewnątrz pliku, zawierającego jej implementację, niż ujawniać ją w pliku nagłówkowym.

Ograniczanie powtórnych kompilacji

Menedżer projektu, dostępny w używanym przez ciebie środowisku programistycznym, spowoduje powtórną kompilację 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 powtórnej kompilacji 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 kompilacje 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 kompilacji 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 {O}
// 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 zagnieżdżoną, 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 **Ceshire**, 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(1);
  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 powtórnej kompilacji. 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 programiście 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 jakiejkolwiek części kodu tworzącego wewnętrzną implementację klasy. Dzięki temu twórca klasy może zmienić 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 jej „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, a następnie je 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 kompilując 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. Zaimplementuj dwie wersje klasy **StackImp** — wykorzystującą tablicę liczb całkowitych o stałym rozmiarze i stosującą typ **vector<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**.