helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

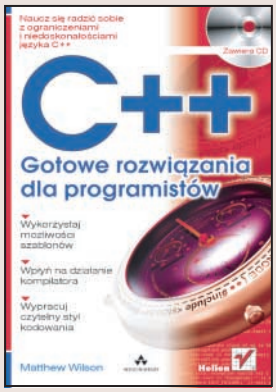
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Język C++. Gotowe rozwiązania dla programistów



Autor: Matthew Wilson
Tłumaczenie: Zbigniew Banach, Michał Dadan, Tomasz Walczak
ISBN: 83-7361-841-4
Tytuł oryginału: Imperfect C++: Practical Solutions for Real-Life Programming
Format: B5, stron: 696

C++ to popularny i uniwersalny język programowania. Jednak po dłuższym stosowaniu programiści zaczynają zauważać pewne jego niedoskonałości i ograniczenia. System typów, sposób działania niektórych kompilatorów, związki pomiędzy wskaźnikami i tablicami, nieprzewidziane w standardzie zachowania obiektów statycznych i bibliotek dynamicznych to tylko niektóre z nich. Aby je obejść, należy wykorzystywać wiele bardzo zaawansowanych i nieznanych wielu programistom metod.

Książka „Język C++. Gotowe rozwiązania dla programistów" to podręcznik dla tych programistów C++, którzy zaczęli już dostrzegać ograniczenia tego języka i zastanawiają się, jak sobie z nimi poradzić. Autor pokazuje sposoby ujarzmienia złożoności języka i uzyskania pełnej kontroli nad kodem. Przedstawia najpoważniejsze wady C++ i sposoby rozwiązywania powodowanych przez nie problemów. Opisuje również metody tworzenia stabilniejszego, bardziej uniwersalnego, wydajniejszego i łatwiejszego w pielęgnacji kodu.

- Wymuszanie założeń projektowych
- Cykl życia obiektów
- Hermetyzacja zasobów, danych i typów
- Modele dostępu do obiektów
- Obsługa wątków
- Korzystanie z obiektów statycznych
- Konwersja danych i typów
- Zarządzanie pamięcią
- Sterowanie działaniem kompilatora

Wszyscy programiści, niezależnie od stopnia zaawansowania, znajdą w tej książce wiadomości, które usprawnią i przyspieszą ich pracę.

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

Spis treści

Przedmowa **11**

Prolog. Filozofia praktyka niedoskonałego **19**

Niedoskonałości, ograniczenia, definicje i zalecenia **29**

Część I Podstawy **37**

Rozdział 1. Wymuszanie założeń projektowych: ograniczenia, kontrakty i asercje **39**

1.1. Kilka oczywistych mądrości 40
1.2. Kontrakty kompilacji — ograniczenia 41
1.3. Kontrakty wykonawcze: warunki wejściowe, końcowe i niezmienniki 49
1.4. Asercje 56

Rozdział 2. Życie obiektów **67**

2.1. Cykl życia obiektu 67
2.2. Kontrola klientów 68
2.3. Dobrodziejstwa list inicjalizacji 73

Rozdział 3. Hermetyzacja zasobów **81**

3.1. Poziomy hermetyzacji zasobów 81
3.2. Typy POD 82
3.3. Pośrednie typy opakowujące 84
3.4. Typy RRID 87
3.5. Typy RAII 92
3.6. RAII — podsumowanie 95

Rozdział 4. Hermetyzacja danych i typy wartości **97**

4.1. Poziomy hermetyzacji danych 98
4.2. Typy wartości a typy egzystencjalne 98
4.3. Klasyfikacja typów wartości 99
4.4. Typy otwarte 101
4.5. Typy hermetyzowane 103
4.6. Typy wartości 104
4.7. Arytmetyczne typy wartości 106
4.8. Typy wartości — podsumowanie 107
4.9. Hermetyzacja — podsumowanie 107

Rozdział 5. Modele dostępu do obiektów 113
5.1. Gwarantowany czas życia 113
5.2. Kopia dla wywołującego 115
5.3. Oryginał dla wywołującego 116
5.4. Obiekty współdzielone 116

Rozdział 6. Zasięg klas 119
6.1. Wartość 119
6.2. Stan 124
6.3. API i usługi 128
6.4. Mechanizmy języka 132

Część II Przetrwanie w świecie rzeczywistym 135

Rozdział 7. ABI 137
7.1. Udostępnianie kodu 137
7.2. Wymagania ABI C 139
7.3. Wymagania ABI C++ 144
7.4. C — i wszystko jasne 148

Rozdział 8. Obiekty bez granic 157
8.1. Czyżby przenośne tabele funkcji wirtualnych? 157
8.2. Przenośne tabele vtable 161
8.3. Przenośność — podsumowanie 169

Rozdział 9. Biblioteki dynamiczne 171
9.1. Jawne wywołania funkcji 171
9.2. Tożsamość — jednostki i przestrzeń konsolidacji 174
9.3. Czas życia 175
9.4. Wersjonowanie 176
9.5. Własność zasobów 179
9.6. Biblioteki dynamiczne — podsumowanie 180

Rozdział 10. Wątki 181
10.1. Synchronizacja dostępu do wartości całkowitych 182
10.2. Synchronizacja dostępu do bloków kodu — regiony krytyczne 186
10.3. Wydajność operacji atomowych 190
10.4. Rozszerzenia wielowątkowe 195
10.5. TSS — składowanie danych w wątkach 199

Rozdział 11. Obiekty statyczne 207
11.1. Globalne obiekty statyczne 209
11.2. Singletony 214
11.3. Lokalne obiekty statyczne funkcji 222
11.4. Składowe statyczne 224
11.5. Obiekty statyczne — podsumowanie 227

Rozdział 12. Optymalizacja 229
12.1. Funkcje inline 229
12.2. Optymalizacja wartości zwracanej 231
12.3. Optymalizacja pustych klas bazowych 234
12.4. Optymalizacja pustych klas potomnych 237
12.5. Zapobieganie optymalizacji 239

Część III Kwestie językowe 243

Rozdział 13. Typy podstawowe 245
13.1. Komu bajt? 246
13.2. Typy całkowitoliczbowe o stałym rozmiarze 249
13.3. Duże typy całkowitoliczbowe 255
13.4. Typy niebezpieczne 257

Rozdział 14. Tablice i wskaźniki 263
14.1. Nie powtarzaj się 263
14.2. Degeneracja tablic do wskaźników 265
14.3. dimensionof() 268
14.4. Nie można przekazywać tablic do funkcji 270
14.5. Tablice są zawsze przekazywane przez adres 273
14.6. Tablice typów dziedziczonych 274
14.7. Brak tablic wielowymiarowych 281

Rozdział 15. Wartości 285
15.1. NULL — słowo kluczowe, którego nie było 285
15.2. Spadek do zera 292
15.3. Naginanie prawdy 294
15.4. Literały 296
15.5. Stałe 302

Rozdział 16. Słowa kluczowe 311
16.1. interface 311
16.2. temporary 314
16.3. owner 317
16.4. explicit(_cast) 321
16.5. unique 326
16.6. final 327
16.7. Nieobsługiwane słowa kluczowe 328

Rozdział 17. Składnia 331
17.1. Układ klasy 331
17.2. Wyrażenia warunkowe 334
17.3. for 338
17.4. Zapis zmiennych 341

Rozdział 18. Definicja typów za pomocą typedef 345
18.1. Definicje typu dla wskaźników 347
18.2. Co wchodzi w skład definicji? 349
18.3. Nowe nazwy 354
18.4. Prawdziwe definicje typu 356
18.5. Dobre, złe i brzydkie 361

Część IV Świadome konwersje 369

Rozdział 19. Rzutowanie 371
19.1. Niejawna konwersja 371
19.2. Rzutowanie w C++ 372
19.3. Przypadek rzutowania w stylu C 373
19.4. Rzutowanie na sterydach 375
19.5. explicit_cast 377
19.6. literal_cast 382
19.7. union_cast 385

19.8. comstl::interface_cast ... 388
19.9. boost::polymorphic_cast ... 399
19.10. Rzutowanie — podsumowanie ... 401

Rozdział 20. Podkładki ... 403
20.1. Ogarnąć zmiany i zwiększyć elastyczność ... 404
20.2. Podkładki atrybutów ... 406
20.3. Podkładki logiczne ... 408
20.4. Podkładki sterujące ... 409
20.5. Podkładki konwertujące ... 411
20.6. Podkładki złożone ... 414
20.7. Przestrzenie nazw a sprawdzenie Koeniga ... 420
20.8. Dlaczego nie typy cechujące? ... 423
20.9. Dopasowanie strukturalne ... 424
20.10. Przełamywanie monolitu ... 426
20.11. Podkładki — podsumowanie ... 428

Rozdział 21. Forniry ... 429
21.1. Lekkie RAII ... 430
21.2. Wiązanie danych z operacjami ... 431
21.3. Przypomnienie założeń ... 438
21.4. Forniry — podsumowanie ... 440

Rozdział 22. Sworznie ... 441
22.1. Dodawanie nowej funkcjonalności ... 442
22.2. Wybór skóry ... 442
22.3. Przesłanianie metod niewirtualnych ... 443
22.4. Wykorzystywanie zasięgu ... 445
22.5. Symulowany polimorfizm czasu kompilacji — sworznie wsteczne ... 447
22.6. Parametryzowalne pakowanie polimorficzne ... 449
22.7. Sworznie — podsumowanie ... 451

Rozdział 23. Konstruktory szablonów ... 453
23.1. Ukryte koszty ... 455
23.2. Wiszące referencje ... 455
23.3. Specjalizacja konstruktorów szablonów ... 457
23.4. Pośrednictwo argumentów ... 458
23.5. Ukierunkowywanie na konkretne rodzaje argumentów ... 460
23.6. Konstruktory szablonów — podsumowanie ... 461

Część V Operatory ... 463

Rozdział 24. operator bool() ... 465
24.1. operator int() const ... 465
24.2. operator void *() const ... 466
24.3. operator bool() const ... 467
24.4. operator !() — not! ... 468
24.5. operator boolean const *() const ... 468
24.6. operator int boolean::*() const ... 469
24.7. Stosowanie operatorów w praktyce ... 469
24.8. operator! ... 473

Rozdział 25. Szybkie i nieinwazyjne łączenie ciągów znaków ... 475
25.1. fast_string_concatenator<> ... 476
25.2. Wydajność ... 484
25.3. Praca z innymi klasami obsługującymi ciągi znaków ... 487

25.4. Inicjalizacja łączenia ... 488
25.5. Patologiczne nawiasowanie ... 489
25.6. Standaryzacja ... 490

Rozdział 26. Jaki jest Twój adres? ... 491
26.1. Nie można pobrać rzeczywistego adresu ... 491
26.2. Co dzieje się w czasie konwersji? ... 494
26.3. Co zwracać? ... 496
26.4. Jaki jest Twój adres — podsumowanie ... 498

Rozdział 27. Operatory indeksowania ... 501
27.1. Przekształcanie wskaźników i operatory indeksowania ... 501
27.2. Obsługa błędów ... 504
27.3. Zwracana wartość ... 506

Rozdział 28. Operatory inkrementacji ... 509
28.1. Brakujące operatory przyrostkowe ... 510
28.2. Wydajność ... 511

Rozdział 29. Typy arytmetyczne ... 515
29.1. Definicja klasy ... 515
29.2. Konstruktor domyślny ... 516
29.3. Inicjalizacja (nadawanie wartości) ... 517
29.4. Konstruktor kopiujący ... 519
29.5. Przypisanie ... 519
29.6. Operatory arytmetyczne ... 520
29.7. Operatory porównania ... 520
29.8. Dostęp do wartości ... 521
29.9. Klasa SInteger64 ... 521
29.10. Obcinanie, promocja do większego typu i testowanie warunków ... 522
29.11. Typy arytmetyczne — podsumowanie ... 525

Rozdział 30. Skrócona ewaluacja ... 527

Część VI Rozszerzanie możliwości języka C++ ... 529

Rozdział 31. Czas życia wartości zwracanej ... 531
31.1. Klasyfikacja problemów związanych z czasem życia wartości zwracanej ... 532
31.2. Do czego służy zwracanie przez referencję? ... 533
31.3. Rozwiązanie pierwsze — szablon integer_to_string<> ... 533
31.4. Rozwiązanie drugie — mechanizm TSS ... 536
31.5. Rozwiązanie trzecie — rozszerzanie RVL ... 541
31.6. Rozwiązanie czwarte — statyczne określanie rozmiaru tablicy ... 543
31.7. Rozwiązanie piąte — podkładki konwertujące ... 545
31.8. Wydajność ... 547
31.9. RVL — wielkie zwycięstwo automatycznego przywracania pamięci ... 548
31.10. Potencjalne zastosowania ... 549
31.11. RVL — podsumowanie ... 549

Rozdział 32. Pamięć ... 551
32.1. Rodzaje pamięci ... 551
32.2. Najlepsze w obu rodzajach ... 553
32.3. Alokatory ... 565
32.4. Pamięć — podsumowanie ... 569

Rozdział 33. Tablice wielowymiarowe 571
33.1. Udostępnianie składni indeksowania 572
33.2. Określanie rozmiaru w czasie wykonywania programu 573
33.3. Określanie rozmiaru na etapie kompilacji 579
33.4. Dostęp blokowy 582
33.5. Wydajność 586
33.6. Tablice wielowymiarowe — podsumowanie 589

Rozdział 34. Funktory i zakresy 591
34.1. Bałagan składniowy 591
34.2. Funkcja for_all() 592
34.3. Funktory lokalne 595
34.4. Zakresy 604
34.5. Funktory i zakresy — podsumowanie 612

Rozdział 35. Właściwości 613
35.1. Rozszerzenia kompilatora 615
35.2. Sposoby implementacji 616
35.3. Właściwości pola 617
35.4. Właściwości metody 624
35.5. Właściwości statyczne 638
35.6. Właściwości wirtualne 642
35.7. Zastosowania właściwości 642
35.8. Właściwości — podsumowanie 645

Dodatki 647

Dodatek A Kompilatory i biblioteki 649
A.1. Kompilatory 649
A.2. Biblioteki 651
A.3. Inne źródła 653

Dodatek B Pycha w końcu Cię zgubi! 655
B.1. Przeciążanie operatora 656
B.2. Jak zawiodło DRY 657
B.3. Programowanie paranoiczne 657
B.4. Do szaleństwa i jeszcze dalej 658

Dodatek C Arturius 661

Dodatek D Płyta CD 663

Epilog 665

Bibliografia 667

Skorowidz 675

Rozdział 4.

Hermetyzacja danych i typy wartości

W poprzednim rozdziale omawialiśmy hermetyzację zasobów jako proces odmienny od hermetyzacji danych. Hermetyzacja zasobów dotyczy bardziej samego mechanizmu niż zawartości zasobów, natomiast hermetyzację danych można w uproszczeniu zdefiniować odwrotnie (choć w konkretnych przypadkach to rozróżnienie często ulega rozmyciu).

Hermetyzacja danych niesie ze sobą tradycyjne zalety klasycznej, obiektowej hermetyzacji:

1. Spójność danych. Stan instancji obiektu można zainicjalizować tak, by utworzyć poprawną całość. Wszelkie modyfikacje instancji za pośrednictwem metod interfejsu są atomowe, co oznacza, że stan składowych będzie spójny przed wywołaniem metody i po jego zakończeniu.
2. Zmniejszenie złożoności. Kod kliencki operuje na instancjach obiektów za pośrednictwem dokładnie zdefiniowanego publicznego interfejsu i nie musi (a w dodatku nie potrzebuje) znać poziomu wewnętrznej złożoności typu.
3. Odporność na zmiany. Kod kliencki jest niezależny od wewnętrznych zmian wprowadzanych w implementacji typu. Możliwe jest też działanie na kilku różnych typach o podobnych interfejsach publicznych, ale różnych postaciach wewnętrznych.

Klasy implementujące hermetyzację danych mogą również implementować hermetyzację zasobów (dobrym przykładem są wszelkie klasy napisów), ale w przypadku zasobów chodzi przede wszystkim o sposób hermetyzacji, podczas gdy my zajmiemy się teraz samymi danymi.

Dalszym rozwinięciem hermetyzacji danych są typy wartości. W rozdziale omówimy różnicę między typami wartości (ang. *value types*) a typami egzystencjalnymi (ang. *entity types*) i przyjrzymy się dokładnie cechom wyróżniającym typy wartości. Zobaczymy też, jak różne poziomy hermetyzacji wpływają na definicje typów wartości.

4.1. Poziomy hermetyzacji danych

W poprzednim rozdziale poznaliśmy różne poziomy hermetyzacji zasobów, od otwartych struktur z dostępem poprzez funkcje API po klasy całkowicie hermetyczne.

Poziomom hermetyzacji danych odpowiadają dostępne w C++ specyfikatory dostępu. Dane niehermetyzowane są dostępne dla dowolnego innego kontekstu i są definiowane w bloku `public` klasy (lub w bloku domyślnym struktury lub unii). W pełni hermetyczne dane są definiowane w bloku `private` lub `protected` typu klasowego.

Przy niewielkiej ilości przykładów może się wydawać, że istnieje ścisły związek między poziomem hermetyzacji a stopniem, w jakim dany typ jest typem wartości. Nie jest to jednak regułą i nie musi istnieć żadna bezpośrednia relacja między tymi własnościami. Jest to tylko kolejna myląca kombinacja cech i dobrze mieć świadomość różnicy między tymi dwoma pojęciami.

4.2. Typy wartości a typy egzystencjalne

Ujmując rzecz możliwie najprościej — i jest to moja prywatna, robocza definicja — typy wartości można opisać jako rzeczy, które po prostu są, a typy egzystencjalne jako rzeczy, które coś robią.

Bjarne Stroustrup podaje świetną definicję typów wartości, nazywając je typami konkretnymi[1]: „Ich celem jest robienie jednej małej rzeczy dobrze i wydajnie. Na ogół nie dają możliwości modyfikacji swego zachowania".

Langer i Kreft [Lang2002] podają bardziej precyzyjne definicje. Typami wartości są „typy posiadające zawartość, których zachowanie nieodłącznie zależy od tej zawartości. Na przykład dwa łańcuchy znakowe zachowują się różnie, jeśli różnią się zawartością, a zachowują się tak samo przy takiej samej zawartości (a ich porównywanie wykazuje równość). Zawartość jest ich najważniejszą cechą". Istotne jest podkreślenie, że równość jest ważniejsza od tożsamości, co moim zdaniem stanowi jeden z najważniejszych wyróżników typów wartości.

Langer i Kreft definiują typy egzystencjalne jako takie typy, „których zachowanie jest w dużej mierze niezależne od zawartości. Zachowanie jest ich najważniejszą cechą". Tym samym sprawdzanie równości typów egzystencjalnych jest, ogólnie rzecz biorąc, działaniem bezcelowym. Osobiście wolę prostotę mojej własnej definicji (cóż za niespodzianka), ale kwalifikacja typów zawarta w definicji Langera i Krefta jest istotna.

[1] Dla mnie typ konkretny to taki, dla którego można tworzyć instancje, czyli typ kompletny (bo widać definicję) i nieabstrakcyjny (nie ma żadnych metod czysto wirtualnych do wypełnienia). Żeby było śmieszniej, nawet w przypadku typu abstrakcyjnego istnieją różne definicje. Znowu ból mózgu.

W kontekście tego rozdziału będę traktował pojęcie typu egzystencjalnego jako jednolite, choć w rzeczywistości obejmuje ono wielką różnorodność typów, w tym co najmniej typy konkretne, abstrakcyjne, polimorficzne i niepolimorficzne. Wiele z tych pojęć zostanie omówionych osobno w dalszych częściach książki.

W pozostałej części tego rozdziału przyjrzymy się typom wartości i spróbujemy ustalić, czy można mówić o tylko jednym ich rodzaju. Jak zwykle zamierzam twierdzić, że jest inaczej.

4.3. Klasyfikacja typów wartości

Bjarne Stroustrup [Stro1997] definiuje semantykę wartości (w przeciwieństwie do semantyki wskaźników) jako niezależność skopiowanych typów. Daje to nam świetną podstawę, ale nie wystarczy do definicji.

Eugene Gershnik, jeden z recenzentów tej książki, ma własną, niezależną od języka definicję typów wartości. Dany typ jest typem wartości, jeśli:

1. Instancję można utworzyć jako kopię innej instancji lub później ją taką kopią uczynić.
2. Każda instancja ma własną tożsamość. Zmiana jednej instancji nie powoduje wprowadzenia zmian w innej instancji.
3. Instancja nie może polimorficznie zastąpić instancji innego typu w czasie wykonania ani być przez taką instancję zastąpiona.

Powyższa definicja jest atrakcyjna, ale zdecydowanie zbyt szeroka jak na mój gust. W dalszej części tego rozdziału nieco ją zawęzimy.

Jednym z możliwych podejść do typów wartości jest ocena, czy i w jakim stopniu zachowują się rozsądnie. Czego na przykład moglibyśmy się spodziewać po poniższych wyrażeniach?

```
String      str1("Pierwotny napis");
String      str2("Niedoskonały");
String      str3("C++");
char const  *cs1 = str1.c_str();

str1 = str2 + " " + str3; // wyrażenie 1
if(!str3) { . . . }       // wyrażenie 2
str2.Empty();             // wyrażenie 3
++str;                    // wyrażenie 4
```

Moim zdaniem wyrażenie 1. spowodowałoby sklejenie `str2`, `" "` i `str3` (w tej kolejności). Wynik zostałby umieszczony w `str1`, powodując nadpisanie, rozszerzenie lub zastąpienie zakresu pamięci, w którym w chwili utworzenia `str1` został zapisany ciąg

`"Pierwotny napis"`[2]. Stwierdziłbym też, że z chwilą zakończenia wyrażenia 1. wskaźnik `cs1` stałby się niepoprawny i dalsze korzystanie z niego prowadziłoby do zachowania nieokreślonego (oczywiście nie byłoby tego problemu, gdyby metoda `String::c_str()` była zadeklarowana jako `temporary` [patrz punkt 16.2], gdyż przypisanie byłoby wtedy niedopuszczalne).

Typowym rozumieniem wyrażenia 2. byłoby wykonanie bloku pod warunkiem, że `str3` „nie ma". Problem polega na ustaleniu, co dokładnie oznacza to „niebycie" — może chodzi o brak zawartości, może o zawieranie pustego ciągu `""`, a może o jedno i drugie? Zresztą równie dobrze może chodzić o zawierania ciągu `"false"`! Takie wyrażenie jest niejednoznaczne, a wieloznaczność jest najgorszym wrogiem poprawności i łatwości pielęgnacji. Niestety, nieraz widywałem identyczne konstrukcje w kodzie produkcyjnym.

Trzecie polecenie może oznaczać „opróżnij `str2`", ale równie dobrze może znaczyć „zwróć wartość wskazującą, czy ciąg `str2` jest pusty"[3]. Osobiście wybrałbym zawsze pierwsze znaczenie wyrażenia, rozumując zgodnie z zasadą, że nazwami typów i zmiennych są rzeczowniki, a nazwami metod i funkcji czasowniki. Niestety, biblioteka standardowa nie zgadza się ze mną w tej kwestii, a trudno się nie zgadzać z biblioteką standardową[4]. Wyrażenie 4. jest pozbawione sensu, gdyż nie przychodzi mi do głowy żaden rozsądny sposób inkrementacji ciągu znaków w C++[5] (dodatek B przytacza historię z zamierzchłych czasów, gdy takich zahamowań nie miałem).

W przypadku typów wbudowanych nie ma problemu z przewidywalnością zachowania, gdyż jest ono z góry określone i nienaruszalne. Tworząc własne typy, mamy zatem obowiązek zapewnienia przewidywalności ich działania ze zdefiniowanymi operatorami. Każdy, kto napisze na przykład typ całkowitoliczbowy o zwiększonej precyzji i zdefiniuje `operator -=()` jako dzielenie modulo, zostanie nieuchronnie zlinczowany. Typy z założenia traktowane jak wartości powinny w miarę możliwości zachowywać się „tak jak `int`-y" [Meye1996].

W pozostałej części rozdziału przyjrzymy się całemu spektrum typów wartości. Wyróżniam tu cztery poziomy:

1. Typy otwarte — zwykłe struktury z funkcjami API.
2. Typy hermetyzowane — częściowo lub całkowicie hermetyzowane typy klasowe, obsługiwane za pośrednictwem metod.

---

[2] Jest to wprawdzie wygodne, ale stosowanie operatora + do napisów jest pewnym nadużyciem. Dodawanie jest operacją arytmetyczną i stosowanie jej do ciągów znakowych jest pierwszym krokiem na krętej i niebezpiecznej ścieżce (patrz dodatek B). W rozdziale 25. zignoruję jednak moje własne zastrzeżenia w pogoni za chwałą zwycięzcy.

[3] Problem polega tu na niejednoznaczności słowa *empty* w nazwie metody. W pierwszym rozumieniu może to być forma rozkazująca czasownika, czyli polecenie *opróżnij*, natomiast w drugim rozumieniu jest to przymiotnik *pusty — przyp. tłum.*

[4] W przypadku bibliotek STLSoft musiałem przełknąć swoje zasady i dostosować się do większości: małe litery, podkreślenia, `empty()` itd.

[5] W Perlu i niektórych innych językach skryptowych występuje inkrementacja ciągów znakowych polegająca na dokonaniu interpretacji numerycznej ciągu, zwiększeniu wyniku o jeden i konwersji z powrotem na ciąg znaków. W przypadku Perla jest to jak najbardziej na miejscu, ale kod C++ takich rzeczy wyprawiać nie powinien.

3. Typy wartości — całkowicie hermetyzowane typy klasowe, w tym operatory przypisania, równości i nierówności.
4. Arytmetyczne typy wartości — używane do wartości numerycznych, w tym wszystkie operatory arytmetyczne.

W zależności od punktu widzenia, za typy wartości można uznać wszystkie powyższe typy lub tylko dwa ostatnie. W klasyfikacji umieściłem jednak wszystkie, gdyż odpowiadają one odrębnym i stosowanym w praktyce przedziałom spektrum.

4.4. Typy otwarte

4.4.1. Otwarte typy POD

Typami otwartymi nazwiemy typy proste o publicznie dostępnych składowych, najczęściej pozbawione metod — innymi słowy, są to agregaty (C++98: 8.5.1; 1). Rozważmy typ `uinteger64`:

```
struct uinteger64
{
   uint32_t  lowerVal;
   uint32_t  upperVal;
};
```

Jest to struktura bardzo prosta i niemal całkowicie bezużyteczna. Jako taka stanowi świetny przykład otwartego typu wartości, gdyż, ogólnie rzecz biorąc, typy takie niespecjalnie nadają się do użytku.

Korzystanie z typów otwartych w charakterze typów wartości wymaga niezwykłej cierpliwości. Nie mogą one brać udziału w prostych porównaniach, a wykonywanie na nich działań arytmetycznych wymaga ręcznego operowania na ich składowych.

```
uinteger64  i1 = . . .;
uinteger64  i2 = . . .;
bool        bLess   = i1 < i2;  // błąd kompilacji!
bool        bEqual  = i1 == i2; // błąd kompilacji!
uinteger64  i3      = i1 + i2;  // błąd kompilacji!
```

Powinniśmy jednak być wdzięczni językowi, że z marszu odrzuca wszystkie te operatory, gdyż w ten sposób chroni nas już podczas kompilacji. Porównywanie według składowych byłoby bardzo niebezpieczne — prawdopodobnie dałoby się w wielu przypadkach określić poprawną równość czy też nierówność, ale skąd kompilator miałby znać priorytety operatorów w sprawdzaniu mniejszości? (Warto wiedzieć, że dla zachowania zgodności wstecznej ze strukturami kompilator dopuszcza konstruktory kopiujące i przypisanie kopiujące [patrz podrozdział 2.2]).

Pomimo poważnych problemów z praktycznym korzystaniem z typów otwartych, czasami praca z nimi jest nieunikniona, najczęściej podczas łączenia się z API systemu operacyjnego lub biblioteki, na przykład w celu uzyskania dostępu do operacji arytmetycznych

o zwiększonej precyzji [Hans1997]. W celach instruktażowych załóżmy, że z jakichś względów musimy jednak korzystać ze zdefiniowanych wcześniej 64-bitowych liczb całkowitych. W takiej sytuacji trzeba utworzyć API do wykonywania operacji na tych typach.

```
void   UI64_Assign( uinteger64 *lhs, uint32_t higher
                  , uint32_t lower);
void   UI64_Add( uinteger64 *result, uinteger64 const *lhs
               , uinteger64 const *rhs);
void   UI64_Divide( uinteger64 *result, uinteger64 const *lhs
                  , uinteger64 const *rhs);
int    UI64_Compare( uinteger64 const *lhs, uinteger64 const *rhs);
#define UI64_IsLessThan(pi1, pi2)     (UI64_Compare(pi1, pi2) < 0)
#define UI64_IsEqual(pi1, pi2)        (0 == UI64_Compare(pi1, pi2))
#define UI64_IsGreaterThan(pi1, pi2)  (0 < UI64_Compare(pi1, pi2))
```

Z pomocą takiego API, możemy zaimplementować pierwotny kod w całkowicie legalny sposób:

```
uinteger64  i1 = . . .;
uinteger64  i2 = . . .;
bool        bLess  = UI64_IsLessThan(i1, i2);
bool        bEqual = UI64_IsEqual(i1, i2);
uinteger64  i3;

UI64_Add(&i3, &i1, &i2);
```

Straszne rzeczy. Na szczęście C++ umożliwia nam znaczne ulepszenie tego rozwiązania.

4.4.2. Struktury danych C++

Zanim wszyscy pobiegniemy do naszego kodu źródłowego i w obiektowym amoku zaczniemy nerwowo zamieniać każde `struct` na `class`, musimy sobie uświadomić, że dotychczas omówiliśmy tylko te otwarte typy danych, w których poszczególne składowe stanowią logiczną całość, a wprowadzanie zmian w indywidualnych składowych stanowi ewidentne zagrożenie dla stanu logicznego całej struktury.

Istnieją jednak typy otwarte, na których można wykonywać takie operacje i które tym samym nie stanowią żadnego zagrożenia. Odróżnianie typów bezpiecznych od niebezpiecznych opiera się na stwierdzeniu, czy wprowadzanie zmian w poszczególnych składowych prowadzi do naruszenia znaczenia całości.

Spójrzmy na następujący typ określający walutę:

```
struct Currency
{
   int majorUnit; // dolary, złotówki
   int minorUnit; // centy, grosze
};
```

Jest to przykład niebezpiecznego typu otwartego, gdyż istnieje możliwość podania wartości składowej `minorUnit`, która uczyni całą instancję typu `Currency` logicznie błędną. Z kolei poniższy typ jest idealnym przykładem całkowicie bezpiecznego typu otwartego:

```
struct Patron
{
   String   name;   // nazwisko
   Currency wallet; // zasobność portfela
};
```

Nie ma nierozerwalnego związku między nazwiskiem a zawartością portfela, stąd też wprowadzenie dowolnej zmiany w polu `name` lub `wallet` nie prowadzi do żadnego oczywistego naruszenia spójności logicznej typu `Patron`. Takie typy nazywane są strukturami danych C++ [Stro1997].

Jak zawsze w przypadku rozgraniczeń istnieje tu pewna szara strefa, ale powyższe dwa przykłady są całkowicie jednoznaczne: jeden jest czarny, a drugi biały.

4.5. Typy hermetyzowane

Otwarte typy wartości POD są kruchymi tworami, a ich zastosowanie ogranicza się do przypadków, w których korzystanie z pojęć wyższego poziomu jest ewidentnie szkodliwe dla innych wymagań, na przykład wydajności, współpracy między językami czy łączenia typów. Wydaje się oczywistym, że większość typów nie jest użyteczna, dopóki nie odpowiadają one co najmniej kolejnemu poziomowi, jakim są typy hermetyzowane (poniższą definicję można w równym stopniu stosować do typów egzystencjalnych).

Definicja: *Typy hermetyzowane* umożliwiają odczyt i modyfikację stanu instancji obiektu za pośrednictwem odpowiedniego interfejsu publicznego. Kod kliencki nie powinien (i nie potrzebuje) mieć bezpośredniego dostępu do składowych takich typów. Typy hermetyzowane pozwalają pojęciowo odgraniczyć stan logiczny od fizycznego.

Ogólnie rzecz biorąc, typy hermetyzowane pozwalają w większym stopniu ukryć implementację (niekiedy całkowicie), co zapewnia bezpieczeństwo, oraz dostarczają metody bezpiecznego dostępu do odpowiedniej reprezentacji wartości. Operacje na składowych `lowerVal` i `upperVal` naszego przykładowego typu `uinteger64` mogłyby się odbywać za pośrednictwem metod klasy w rodzaju `Add()`, `Assign()` itp. Korzystanie z metod daje kontrolę dostępu i modyfikacji wewnętrznego stanu instancji, stąd też możliwe jest narzucanie niezmienników klas (podrozdział 1.3), co pozwala znacznie podnieść jakość kodu.

Typy mogą też udostępniać metody obsługi typowych lub spodziewanych operacji, by uniknąć ręcznego ich tworzenia w kodzie klienckim. Niestety, zbiór takich operacji jest zawsze otwarty, nawet jeśli (jako dobrzy obywatele) zachowujemy ortogonalność typu. Efekt ten można nieco złagodzić, stosując wolne funkcje w miejsce metod klas wszędzie tam, gdzie jest to możliwe [Meye2000].

Klasowa implementacja naszego przykładowego typu będzie wyglądać mniej więcej tak jak pokazana na listingu 4.1 klasa `UInteger64`. Zawiera ona zmienne składowe typu `uinteger64`, co pozwala wykorzystać gotowe funkcje zdefiniowanego wcześniej API).

Listing 4.1.

```
class UInteger64
{
public:
   UInteger64();
   UInteger64(uint32_t low);
   UInteger64(uint32_t high, uint32_t low);
#ifdef ACMELIB_COMPILER_SUPPORTS_64BIT_INT
   UInteger64(uint64_t low);
#endif /* ACMELIB_COMPILER_SUPPORTS_64BIT_INT */
   UInteger64(UInteger64 const &rhs);
   . . .

// porównania
public:
   static bool IsLessThan(UInteger64 const &i1, UInteger64 const &i2);
   static bool IsEqual(UInteger64 const &i1, UInteger64 const &i2);
   static bool IsGreaterThan(UInteger64 const &i1, UInteger64 const &i2);

// operacje arytmetyczne
public:
   static UInteger64 Multiply(UInteger64 const &i1, UInteger64 const &i2);
   static UInteger64 Divide(UInteger64 const &i1, UInteger64 const &i2);

private:
   uinteger64 m_value;
};
```

4.6. Typy wartości

Po typach hermetyzowanych już tylko mały krok dzieli nas od typów, które uważam za prawdziwe typy wartości. Głównym wyróżnikiem typu wartości jest moim zdaniem możliwość porównywania, czyli posiadanie cechy *EqualityComparable* [Aust1999, Muss2001]. Oznacza to zwracanie sensownych wyników porównań pod względem równości i nierówności, zgodnie z definicją Langera-Krefta [Lang2002] (patrz podrozdział 4.2). Dla typów klasowych kompilator nie udostępnia domyślnie operatorów porównania, więc musimy je zdefiniować samodzielnie.

Przede wszystkim potrzebujemy typów, które zachowują się rozsądnie. Główny problem typów hermetyzowanych polega na niemożności ich użycia w kodzie wykorzystującym porównania pod względem równości. Szczególnie istotna jest możliwość umieszczania typów wartości w kontenerach składujących według wartości, w tym w kontenerach biblioteki standardowej. Możemy wprawdzie tworzyć instancje kontenera `std::vector <UInteger64>` i operować nimi, gdyż kontener nie nakłada ograniczeń co do unikalności składowanych elementów, jednak nie możemy wyszukiwać instancji naszego typu za pomocą standardowego algorytmu `std::find<>()`:

```
std::vector<UInteger64>  vi;

vi.push_back(i1);
```

```
vi.push_back(i2);
vi.push_back(i3);

std::find( vi.start(), vi.end()
         , UInteger64(1, 2)); // błąd! brak operatora == dla UInteger64
```

(`std::vector<>` przechowuje elementy bez porządkowania, więc nie ma potrzeby definiowania operatora < do porównania porządkującego).

Przekształcenie naszego typu w typ wartości będzie bardzo proste. W poprzedniej implementacji `UInteger64` zdefiniowaliśmy metodę `IsEqual()`, którą możemy teraz wykorzystać do definicji operatorów równości i nierówności:

```
inline bool operator ==(UInteger64 const &i1, UInteger64 const &i2)
{
   return UInteger64::IsEqual(i1, i2);
}
inline bool operator !=(UInteger64 const &i1, UInteger64 const &i2)
{
   return !operator ==(i1, i2);
}
```

Dodatkową zaletą takiego podejścia jest fakt, że uczyniliśmy typ pełnym typem wartości przez dodanie funkcji nieskładowych [Meye2000]. Funkcje te poprzednio nie istniały, stąd też na pewno nie istnieje żaden kod wymagający ich obecności (lub nieobecności) i tym samym osiągnęliśmy cel bez tworzenia jakichkolwiek problemów pielęgnacyjnych (przytyk odautorski: to właśnie bezmyślne wciskanie wszystkiego do klas jest powodem niekontrolowanego puchnięcia wielu środowisk programistycznych i niesprawiedliwych opinii na temat braku wydajności C++).

Możemy zatem powrócić do naszej definicji typu wartości i zaproponować następującą jej postać[6]:

Definicja: *Typ wartości*

Instancje obiektów nie mogą polimorficzne zastępować instancji innego typu w czasie uruchamiania ani być przez nie zastępowane.

Instancje można tworzyć jako kopie innych instancji lub je takimi kopiami później uczynić.

Każda instancja ma samodzielną tożsamość logiczną. Zmiana stanu logicznego jednej instancji nie wpływa na stan logiczny innej (stan fizyczny może być współdzielony, jeśli tak wskazują decyzje projektowe dla danej implementacji, ale tylko pod warunkiem, że nie wpływa to na niezależność logiczną).

Instancje można porównywać z dowolnymi innymi instancjami (w tym z samymi sobą) pod względem równości i nierówności. Obie te relacje są zwrotne, symetryczne i przechodnie.

---

[6] Eugene twierdzi, że wymaganie sprawdzania równości nie jest konieczne, więc nie mogę ochrzcić definicji mianem definicji typu wartości Gershnika-Wilsona, jak by to gładko nie spływało z języka.

4.7. Arytmetyczne typy wartości

Ostatni krok naszej wspinaczki po drabinie typów wartości polega na dodaniu obsługi typowych operatorów arytmetycznych, co pozwoli naszemu przykładowemu typowi uczestniczyć w tradycyjnych działaniach, na przykład:

```
uinteger64  i1 = . . .;
uinteger64  i2 = . . .;
bool        bLess   = i1 < i2;  // w porządku
bool        bEqual  = i1 == i2; // w porządku
uinteger64  i3      = i1 + i2;  // w porządku

i3 %= i1;
i1 = i2 *= i3;
```

Najpierw zajmiemy się operatorem mniejszości `operator<()`, gdyż dzięki niemu stanie się możliwe porządkowanie instancji typu. Możliwość porównywania pod względem mniejszości jest określana mianem własności *LessThanComparable* [Aust1999] i jest nieodzowną cechą typów obsługiwanych przez STL — zgodnie ze standardem, jest to jedyna cecha wymagana. Przyznaję, że mam osobistą skłonność do korzystania wyłącznie z tego porównania, nawet jeśli oznacza to pisanie mniej czytelnego kodu, na przykład pisanie `assert(!(size() < index))` zamiast `assert(index <= size())`. Nie jestem jednak wcale pewien, czy jest to praktyka godna polecenia.

Możemy zatem dodać operator mniejszości do dotychczasowej definicji naszego typu, implementując go (podobnie jak w przypadku operatorów równości i nierówności) jako funkcję nieskładową. Instancje tak zdefiniowanego typu `UInteger64` możemy już umieszczać w kontenerach biblioteki standardowej, na przykład `std::set` czy `std::map`.

W ten sam sposób można dodać wiele innych operatorów arytmetycznych, ale ich wybór zależy od konkretnego typu. Tworzony przez nas typ `UInteger64` jest typem całkowitoliczbowym, stąd też wskazane byłoby zdefiniowanie wszystkich operatorów, a więc +, -, *, /, %, ~, <<, >>, &, |, ^ i odpowiadających im operatorów przypisania (w rozdziale 29. przyjrzymy się sposobom optymalnego implementowania takich operatorów). Świetnym przykładem definicji wolnych operatorów jest klasa szablonowa `true_typedef` (podrozdział 18.4), w której wszystkie operatory arytmetyczne dla klasy są zdefiniowane jako wolne funkcje. Tak samo możemy postępować w przypadku innych typów.

Najważniejszą kwestią jest zdefiniowanie tylko tych operatorów, które mają sens. Powracając do przykładu typu `Currency` określającego walutę, z pewnością chcielibyśmy mieć możliwość dodawania i odejmowania instancji `Currency` za pomocą operatorów + i -, ale już mnożenie nie miałoby większego sensu (na przykład $6.50 razy 2.93 PLN). Z kolei przydałaby się możliwość przemnożenia instancji `Currency` przez liczbę, ale już dodawanie i odejmowanie liczb nie byłoby potrzebne.

Oczywiście C++ pozwala zdefiniować dowolny operator dla dowolnej klasy i przypisać mu dowolne działanie, co może stanowić przedmiot karygodnych nadużyć (patrz dodatek B). W pierwszym przykładzie tego rozdziału widzieliśmy klasę `String` ze zdefiniowanymi operatorami + i ++. Używanie operatora + do łączenia łańcuchów znako-

wych jest błędem,[7] gdyż nie oblicza on sumy arytmetycznej argumentów. Jest to jednak tak przydatne zastosowanie, że niemal wszyscy przymykamy oko na jego niemoralność i rozkoszujemy się wygodą (może i jest ono wygodne, ale na pewno nie wydajne; w rozdziale 25. zobaczymy, że można tę operację znacząco przyspieszyć).

Nadużywanie operatorów jest wprawdzie szeroko akceptowaną praktyką, ale mimo to jest paskudnym zwyczajem i należy unikać tej pokusy. Wiem, bo sam niegdyś zgrzeszyłem. Bardzo zgrzeszyłem (patrz dodatek B).

4.8. Typy wartości — podsumowanie

Gdy przyglądam się spektrum typów wartości, nasuwa mi się ciekawa analogia z iteratorami STL. Najtrudniejszym do naśladowania iteratorem jest iterator dostępu bezpośredniego, ale obsługują go wskaźniki, w których implementacji wyręcza nas sam język.

Podobnie najtrudniejszym w implementacji pojęciem spośród możliwych typów wartości jest typ arytmetyczny, który z kolei jest domyślnym typem dla podstawowych typów całkowitoliczbowych. Potęga C++ przejawia się w zezwoleniu nam na własnoręczne definiowanie takich pojęć. Można też zadumać się nad faktem, że naszym szczytowym osiągnięciem jest określenie składni najbardziej podstawowych z typów.

4.9. Hermetyzacja — podsumowanie

Dużo mówiliśmy o teoretycznych aspektach typów wartości, ale pozostaje jeszcze kwestia hermetyzacji takich typów. Z podręczników dowiemy się, że jako sumienni, obiektowi obywatele powinniśmy zawsze całkowicie hermetyzować typy. Niestety, rzeczywistość rzadko bywa taka prosta.

Zaproponuję siedem możliwych implementacji typu `UInteger64`, z których tylko trzy są w pełni hermetyzowane. Wybór odpowiedniej w danym zastosowaniu postaci zależy od odpowiedzi na trzy podstawowe pytania.

Czy potrzebujemy implementacji bazującej na istniejącym API w C lub z nim współpracującej? Jeśli tak, to nasza implementacja będzie musiała zawierać istniejącą strukturę (postać 2.) lub po niej dziedziczyć (postać 3.), gdyż tylko w ten sposób możemy przekazywać odpowiednie elementy klasy do API w C[8]. Jeśli nie, to wystarczy zawieranie składowych struktury (postać 1.).

---

[7] Jest to kolejne moje kontrowersyjne stwierdzenie, za które prawdopodobnie nieźle mi się oberwie. Musimy jednak pamiętać, że popularność ani nawet przydatność nie są równoznaczne z poprawnością, a ta książka dostarcza licznych tego przykładów.

[8] Przynajmniej bez uciekania się do dyrektyw pakujących, rzutowania i innych paskudnych sztuczek. Lepiej skorzystać z samej struktury C.

Listing 4.2.

```
// postać #1
class UInteger64
{
   . . . // metody i operatory typu wartości

private:
   uint32_t  lowerVal;
   uint32_t  upperVal;
};

// postać #2
class UInteger64
{
   . . . // metody i operatory typu wartości

private:
   uinteger64  m_value;
};

// postać #3
class UInteger64
   : private uinteger64
{
   . . . // metody i operatory typu wartości
};
```

Z premedytacją wybrałem jako przykład 64-bitowe liczby całkowite, gdyż w rzeczywistej implementacji mogę oszukiwać, korzystając z konwersji z i na prawdziwe 64-bitowe liczby całkowite, które są używane do faktycznej realizacji operacji arytmetycznych. Gdybyśmy chcieli obsłużyć dowolnych rozmiarów liczby całkowite korzystające z API C (opisane na przykład w [Hans1997]), to musielibyśmy pracować na strukturach C.

Czy wszystkie operacje można hermetyzować w ramach typu? Jeśli tak, to do zapewnienia pełnej hermetyzacji wystarczy prawdopodobnie zadeklarowanie wewnętrznej implementacji jako `private` (postacie 1 – 3). Jeśli nie, to interakcja z innymi funkcjami czy typami będzie wymagała ujawnienia jakiejś części stanu wewnętrznego (w przypadku typów możemy skorzystać z deklaracji typów zaprzyjaźnionych za pomocą modyfikatora `friend` — patrz punkt 2.2.9). W ustaleniu, czy konieczne jest ujawnianie szczegółów implementacji na zewnątrz, pomoże kilka pytań pomocniczych.

Czy będziemy tworzyć typ unikalny, czy jeden z wielu podobnych? Klasycznym przykładem może tu być obsługa czasu. W samym C mamy `time_t`, `struct tm`, `struct timeval`, `DATE`, `FILETIME`, `SYSTEMTIME` i wiele innych typów czasowych, a jeśli dodatkowo uwzględnić typy C++, lista będzie praktycznie nieograniczona. Któż nie implementował własnego typu czasu czy daty? Jeśli nasz typ jest takim właśnie jednym z wielu, to musimy uwzględnić ewentualną potrzebę interakcji i wzajemnej konwersji z typami już istniejącymi.

Czy potrzebujemy interakcji z interfejsem w stylu C? W świecie idealnym odpowiedź brzmiałaby zawsze „nie”, ale w świecie rzeczywistym jest nią często „tak”. Każda chyba implementacja klasy daty będzie bazować na jednym z typów C, gdyż w przeciwnym

razie musielibyśmy ręcznie przepisywać cały skomplikowany, żmudny i podatny na błędy kod do obsługi kalendarza. Może komuś lata przestępne, kalendarz gregoriański albo poprawki orbitalne? Dziękuję bardzo. Tylko jak się upewnić, że nasza hermetyzacja objęła wszystkie potrzebne funkcje?

Społeczność C++, ogólnie rzecz biorąc, ignoruje wszelkie informacje związane z tą kwestią. Moim zdaniem z wielką szkodą dla całej społeczności twórców oprogramowania.[9] Producenci narzędzi skwapliwie unikają wyprowadzania programistów z błędu, bo im bardziej programista przyzwyczai się do pracy z jakąś „naprawdę przydatną klasą", tym mniej będzie się interesować alternatywnymi rozwiązaniami. Implikacje takiego przyzwyczajenia wykraczają daleko poza preferowanie jednego, wybranego środowiska. Z pewnością każdy miał do czynienia z projektami nieodwracalnie związanymi z określonym systemem operacyjnym tylko dlatego, że programiści nie chcieli (lub nie potrafili) wykroczyć poza ramy konkretnego środowiska tworzenia aplikacji. Jest to potężna niedoskonałość C++, z którą rozprawimy się bezlitośnie w części drugiej.

W wielu zastosowaniach praktycznych ewidentnie nie mamy wyboru i musimy się zadowolić częściową hermetyzacją. Można ją osiągnąć na kilka sposobów. Najprościej byłoby zmienić specyfikator dostępu z `private` na `public` (postacie 4 – 6), ale spowodowałoby to ujawnienie całej zawartości klasy.

Listing 4.3.

```
// postać #4
class UInteger64
{
   . . . // metody i operatory typu wartości

public:
   uint32_t  lowerVal;
   uint32_t  upperVal;
};

// postać #5
class UInteger64
{
   . . . // metody i operatory typu wartości

public:
   uinteger64  m_value;
};

// postać #6
class UInteger64
   : public uinteger64
{
   . . . // metody i operatory typu wartości
};
```

[9] Niektórzy recenzenci tej książki narzekali nawet, że w publikacji ewidentnie poświęconej C++ nie powinienem omawiać przykładów kodu zgodnego z C.

Istnieją sposoby nieco bardziej dyskretnego ujawniania informacji. Dobrze sprawdzają się tu funkcje dostępowe, ale są one wyjątkowo nieestetyczne:

```
// postać #6
class UInteger64
   : private uinteger64
{
   . . . // metody i operatory typu wartości
public:
   uinteger64 &get_uinteger64();
   uinteger64 const &get_uinteger64() const; // brzydkie!
};
```

Można te paskudztwa nieco ugładzić, odpowiednio definiując operator `explicit_cast`, któremu przyjrzymy się dokładniej w podrozdziałach 16.4 i 19.5.

Listing 4.4.

```
// postać #7
class UInteger64
   : private uinteger64
{
   . . . // metody i operatory typu wartości
public:
   operator explicit_cast<uinteger64 &>();
   operator explicit_cast<uinteger64 const &>() const;
};
```

W ten sposób zapobiegamy niejawnemu dostępowi do wnętrza klasy, zapewniając jednocześnie dostęp jawny. Inną możliwością jest dodanie do klasy osobnej metody dla każdej operacji API, którą chcemy udostępnić użytkownikom klasy nadrzędnej.

Z zagadnieniem tym są ściśle związane kolejne dwa pytania, pozwalające dodatkowo sprecyzować kryteria wyboru rozwiązania:

Czy względy wydajności będą wymagać naruszenia ortogonalności? Może się zdarzyć, że niektóre przydatne operacje na naszych typach będą stanowić logiczną kombinację dwóch prostszych operacji, ale połączenie ich w jedną całość da znaczny przyrost wydajności. Co gorsza, takie łączenie jest niekiedy dokonywane wyłącznie dla wygody. Jak by na to nie patrzeć, jest to pierwszy krok na drodze do opasłych klas o przerażającej ilości metod (patrz `std::basic_string<>`). Prawidłowy wybór w tej kwestii zależy od konkretnego sytuacji, ale trzeba pamiętać, by faworyzować wydajność tylko w przypadku metod stanowiących autentyczne wąskie gardła. Wszelkie decyzje należy podejmować na podstawie faktycznych pomiarów, a nie instynktu (w tej kwestii dość często omylnego).

Czy możemy być pewni uniknięcia dołączania plików podczas konsolidacji i kompilacji? Jest to kolejny niedostatecznie omówiony aspekt projektowania klas. Jeśli całą implementację typu można wydajnie zapisać w postaci definicji wbudowanej, to w ogóle nie musimy się przejmować dołączaniem podczas konsolidacji. Dołączanie takie obejmuje implementacje funkcji skompilowane osobno w ramach tej samej jednostki konsolidacji lub umieszczone w zewnętrznych bibliotekach (statycznych bądź dynamicznych).

Tak czy inaczej musimy jeszcze pomyśleć o plikach dołączanych podczas kompilacji, czyli wszystkich plikach niezbędnych do kompilacji danej klasy. Ironią losu jest to, że zmniejszenie ilości plików dołączanych podczas konsolidacji często zwiększa ilość plików dołączanych podczas kompilacji.

Niestety, trudno jest uniknąć dołączania podczas kompilacji, a im bardziej staramy się zapewnić przenośność, tym więcej plików trzeba dołączać. Powodem tego jest konieczność obsłużenia niestandardowych typów (podrozdział 13.2), funkcji kompilatorów, konwencji wywołania (rozdział 7.) i tym podobnych. Wszelkie tego typu moduły są (zupełnie rozsądnie) umieszczane w osobnych, dokładnie przetestowanych plikach nagłówkowych, które rozrastają się w miarę dojrzewania poszczególnych rozwiązań. Pozwala to scentralizować obsługę różnych architektur, kompilatorów, systemów operacyjnych i bibliotek.

Jak widać, hermetyzacja danych nie jest sprawą oczywistą, stąd też (podobnie jak w przypadku wielu innych zagadnień omawianych w tej książce) jedyną uniwersalną zasadą jest pamiętanie o wszystkich poruszonych kwestiach. Zakres działania tworzonych przez nas klas nie powinien być ograniczony do środowiska, w którym je piszemy, a ponieważ klasy nie myślą, więc to my powinniśmy myśleć za nie.