

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

CZYTELNIĄ

FRAGMENTY KSIĄŻEK ONLINE

C++. Projektowanie systemów informatycznych. Vademecum profesjonalisty

Autor: John Lakos

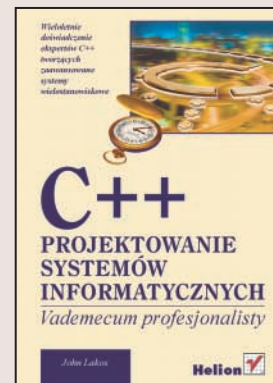
Tłumaczenie: Wojciech Moch (rozdz. 1 – 4), Michał Dadan (rozdz. 5, 6), Radosław Meryk (rozdz. 7 – 10, dod. A – C)

ISBN: 83-7361-173-8

Tytuł oryginału: [Large-Scale C++ Software Design](#)

Format: B5, stron: 688

[Przykłady na ftp: 52 kB](#)



C++ nie jest tylko rozszerzeniem języka C, ale wprowadza zupełnie nowy model programowania. Stopień skomplikowania C++ może być przytłaczający nawet dla doświadczonych programistów C, jednak zazwyczaj nie sprawia im problemów napisanie i uruchomienie małego, niebanalnego programu w C++. Niestety, brak dyscypliny dopuszczalny przy tworzeniu małych programów, zupełnie nie sprawdza się w dużych projektach. Podstawowe użycie technologii C++ nie wystarczy do budowy dużych projektów. Na niezorientowanych czeka wiele pułapek.

Książka ta opisuje metody projektowania dużych systemów wysokiej jakości. Adresowana jest do doświadczonych programistów C++ próbujących stworzyć architekturę łatwą w obsłudze i możliwą do ponownego wykorzystania. Nie zawarto w niej teoretycznego podejścia do programowania. W tej książce znajdują się praktyczne wskazówki wyływające z wieloletnich doświadczeń ekspertów C++ tworzących ogromne systemy wielostanowiskowe. Autor pokazuje, jak należy projektować systemy, nad którymi pracują setki programistów, składające się z tysięcy klas i prawdopodobnie milionów linii kodu.

W książce opisano:

- Tworzenie programów wieloplukowych w C++
- Konstruowanie komponentów
- Podział projektu fizycznego na poziomy
- Całkowitą i częściową izolację, reguły jej stosowania
- Tworzenie pakietów i ich podział na poziomy
- Projektowanie funkcji
- Implementowanie metod

Dodatki do książki opisują przydatny wzorzec projektowy – hierarchię protokołów, implementowanie interfejsu C++ zgodnego ze standardem ANSI C oraz pakiet służący do określania i analizowania zależności.



Spis treści

Przedmowa	11
Wprowadzenie.....	15
W.1. Od C do C++	15
W.2. Jak używać C++ do tworzenia dużych projektów	16
W.2.1. Zależności cykliczne.....	16
W.2.2. Nadmierne zależności na etapie konsolidacji	18
W.2.3. Nadmierne zależności na etapie kompilacji.....	20
W.2.4. Globalna przestrzeń nazw	22
W.2.5. Projekt logiczny i fizyczny	23
W.3. Ponowne użycie.....	25
W.4. Jakość	25
W.4.1. Kontrola jakości.....	27
W.4.2. Zapewnienie jakości	27
W.5. Narzędzia.....	27
W.6. Podsumowanie	28

Część I Podstawy

29

Rozdział 1. Wstęp	31
1.1. Programy wieloplikowe w C++	31
1.1.1. Deklaracja a definicja.....	31
1.1.2. Konsolidacja (łączenie) wewnętrzna a zewnętrzna	33
1.1.3. Pliki nagłówkowe (.h)	36
1.1.4. Pliki implementacji (.c)	37
1.2. Deklaracje typedef	38
1.3. Instrukcje sprawdzające	39
1.4. Kilka słów na temat stylu.....	40
1.4.1. Identyfikatory.....	41
1.4.1.1. Nazwy typów.....	41
1.4.1.2. Nazwy identyfikatorów składające się z wielu słów.....	42
1.4.1.3. Nazwy składowych klas	42
1.4.2. Kolejność ułożenia składowych w klasie	44
1.5. Iteratory.....	46
1.6. Logiczna notacja projektu	52
1.6.1. Relacja Jest.....	53
1.6.2. Relacja Używa-W-Interfejsie	54
1.6.3. Relacja Używa-W-Implementacji	56
1.6.3.1. Relacja Używa.....	58
1.6.3.2. Relacje Ma i Trzyma	58
1.6.3.3. Relacja Był.....	59

1.7. Dziedziczenie i warstwy.....	60
1.8. Minimalizacja.....	61
1.9. Podsumowanie	62
Rozdział 2. Podstawowe reguły.....	65
2.1. Przegląd	65
2.2. Dostęp do pól klasy.....	66
2.3. Globalna przestrzeń nazw	70
2.3.1. Dane globalne.....	70
2.3.2. Wolne funkcje	72
2.3.3. Wyliczenia, stałe i deklaracje typedef	73
2.3.4. Makra preprocesora.....	74
2.3.5. Nazwy w plikach nagłówkowych.....	75
2.4. Kontrola dołączeń	77
2.5. Dodatkowa kontrola dołączeń	79
2.6. Dokumentacja	84
2.7. Sposoby nazywania identyfikatorów.....	86
2.8. Podsumowanie	87
Część II Projekt fizyczny	91
Rozdział 3. Komponenty	93
3.1. Komponenty a klasy.....	93
3.2. Reguły projektów fizycznych.....	100
3.3. Relacja ZależyOd	108
3.4. Zależności implikowane.....	112
3.5. Wydobywanie rzeczywistych zależności	117
3.6. Przyjaźń	119
3.6.1. Przyjaźń na odległość i zależności implikowane.....	122
3.6.2. Przyjaźń i oszustwo	124
3.7. Podsumowanie	126
Rozdział 4. Hierarchia fizyczna.....	129
4.1. Metafora dla testowania oprogramowania	129
4.2. Złożony podsystem	130
4.3. Problemy z testowaniem „dobrych” interfejsów	134
4.4. Projektowanie zorientowane na testowalność	136
4.5. Testowanie pojedynczych modułów	138
4.6. Acykliczne zależności fizyczne.....	140
4.7. Numery poziomów.....	142
4.7.1. Źródła numeracji poziomów.....	142
4.7.2. Używanie numerów poziomów w oprogramowaniu	144
4.8. Testowanie hierarchiczne i przyrostowe	147
4.9. Testowanie złożonych podsystemów	153
4.10. Testowalność kontra testowanie.....	154
4.11. Cykliczne zależności fizyczne.....	155
4.12. Suma zależności komponentów	156
4.13. Jakość projektu fizycznego	161
4.14. Podsumowanie	167
Rozdział 5. Podział na poziomy	169
5.1. Niektóre przyczyny cyklicznych zależności fizycznych	169
5.1.1. Rozszerzenie	170
5.1.2. Wygoda	172
5.1.3. Wewnętrzna zależność	176

5.2. Wyniesienie.....	178
5.3. Obniżenie.....	187
5.4. Nieprzezroczyste wskaźniki.....	199
5.5. Głupie dane.....	206
5.6. Redundancja.....	216
5.7. Wywołania zwrotne.....	219
5.8. Klasa-menedżer.....	231
5.9. Faktoring.....	235
5.10. Wynoszenie enkapsulacji.....	249
5.11. Podsumowanie.....	260

Rozdział 6. Izolacja261

6.1. Od enkapsulacji do izolacji.....	262
6.1.1. Koszty powiązań na etapie kompilacji.....	266
6.2. Konstrukcje w języku C++ a zależności na etapie kompilacji.....	267
6.2.1. Dziedziczenie (Jest) a zależności na etapie kompilacji.....	268
6.2.2. Podział na warstwy (Ma/Trzyma) a zależności na etapie kompilacji.....	269
6.2.3. Funkcje inline a zależności na etapie kompilacji.....	270
6.2.4. Składowe prywatne a zależności na etapie kompilacji.....	271
6.2.5. Składowe chronione a zależności na etapie kompilacji.....	273
6.2.6. Funkcje składowe generowane przez kompilator a zależności na etapie kompilacji.....	274
6.2.7. Dyrektywy include a zależności na etapie kompilacji.....	275
6.2.8. Argumenty domyślne a zależności na etapie kompilacji.....	277
6.2.9. Wyliczenia a zależności na etapie kompilacji.....	277
6.3. Techniki częściowej izolacji.....	279
6.3.1. Rezygnacja z dziedziczenia prywatnego.....	279
6.3.2. Usuwanie osadzonych danych składowych.....	281
6.3.3. Usuwanie prywatnych funkcji składowych.....	282
6.3.4. Usuwanie składowych chronionych.....	290
6.3.5. Usuwanie prywatnych danych składowych.....	299
6.3.6. Usuwanie funkcji generowanych przez kompilator.....	302
6.3.7. Usuwanie dyrektyw include.....	302
6.3.8. Usuwanie argumentów domyślnych.....	303
6.3.9. Usuwanie wyliczeń.....	305
6.4. Techniki całkowitej izolacji.....	307
6.4.1. Klasa protokołu.....	308
6.4.2. W pełni izolująca klasa konkretna.....	317
6.4.3. Izolujące komponenty otaczające.....	322
6.4.3.1. Pojedyncze komponenty otaczające.....	323
6.4.3.2. Wielokomponentowe warstwy otaczające.....	331
6.5. Interfejs proceduralny.....	338
6.5.1. Architektura interfejsu proceduralnego.....	339
6.5.2. Tworzenie i usuwanie nieprzezroczystych obiektów.....	341
6.5.3. Uchwyty.....	342
6.5.4. Uzyskiwanie dostępu do nieprzezroczystych obiektów i manipulowanie nimi.....	346
6.5.5. Dziedziczenie a nieprzezroczyste obiekty.....	351
6.6. Izolować czy nie izolować.....	354
6.6.1. Koszt izolacji.....	354
6.6.2. Kiedy nie należy izolować.....	356
6.6.3. Jak izolować.....	360
6.6.4. Do jakiego stopnia należy izolować.....	366
6.7. Podsumowanie.....	373

Rozdział 7. Pakiety	377
7.1. Od komponentów do pakietów	378
7.2. Zarejestrowane prefiksy pakietów.....	385
7.2.1. Potrzeba stosowania prefiksów	385
7.2.2. Przestrzenie nazw	387
7.2.3. Zachowanie integralności pakietu	391
7.3. Podział pakietów na poziomy	393
7.3.1. Znaczenie podziału pakietów na poziomy	393
7.3.2. Techniki podziału pakietów na poziomy	394
7.3.3. Podział systemu.....	396
7.3.4. Wytwarzanie oprogramowania w wielu ośrodkach	398
7.4. Izolacja pakietów	399
7.5. Grupy pakietów	402
7.6. Proces wydawania oprogramowania	406
7.6.1. Struktura wydania	408
7.6.2. Łaty	413
7.7. Program main.....	415
7.8. Faza startu	421
7.8.1. Strategie inicjalizacji	423
7.8.1.1. Technika „przebudzenia” w stanie zainicjowania	424
7.8.1.2. Technika jawnego wywoływania funkcji init.....	424
7.8.1.3. Technika wykorzystania specjalnego licznika	426
7.8.1.4. Technika sprawdzania za każdym razem.....	431
7.8.2. Porządkowanie	432
7.8.3. Przegląd.....	433
7.9. Podsumowanie	434
 Część III Zagadnienia dotyczące projektu logicznego	 437
Rozdział 8. Projektowanie komponentów.....	439
8.1. Abstrakcje i komponenty	439
8.2. Projekt interfejsu komponentu	440
8.3. Poziomy enkapsulacji.....	444
8.4. Pomocnicze klasy implementacyjne.....	454
8.5. Podsumowanie	459
 Rozdział 9. Projektowanie funkcji	 461
9.1. Specyfikacja interfejsu funkcji.....	462
9.1.1. Operator czy metoda?.....	462
9.1.2. Wolny operator czy składowa klasy?	468
9.1.3. Metoda wirtualna czy niewirtualna?.....	472
9.1.4. Metoda czysto wirtualna czy nie czysto wirtualna?	476
9.1.5. Metoda statyczna czy niestatyczna?	477
9.1.6. Metody stałe czy modyfikowalne?.....	478
9.1.7. Metody publiczne, chronione czy prywatne	483
9.1.8. Zwracanie wyniku przez wartość, referencję czy wskaźnik?	484
9.1.9. Zwracanie wartości typu const czy nie-const?	487
9.1.10. Argument opcjonalny czy obowiązkowy?.....	488
9.1.11. Przekazywanie argumentów przez wartość, referencję lub wskaźnik	490
9.1.12. Przekazywanie argumentów jako const lub nie-const	495
9.1.13. Funkcja zaprzyjaźniona czy niezaprzyjaźniona?	496
9.1.14. Funkcja inline czy nie inline?.....	497

9.2. Typy podstawowe użyte w interfejsie	498
9.2.1. Użycie typu short w interfejsie.....	498
9.2.2. Użycie kwalifikatora unsigned w interfejsie	501
9.2.3. Zastosowanie typu long w interfejsie	505
9.2.4. Zastosowanie typów float, double oraz long double w interfejsie.....	507
9.3. Funkcje specjalnego przeznaczenia.....	508
9.3.1. Operatory konwersji.....	508
9.3.2. Semantyka wartości generowanych przez kompilator.....	512
9.3.3. Destruktor.....	513
9.4. Podsumowanie	515
Rozdział 10. Implementowanie obiektów.....	521
10.1. Pola	521
10.1.1. Wyrównanie naturalne.....	522
10.1.2. Użycie typów podstawowych w implementacji.....	524
10.1.3. Użycie konstrukcji typedef w implementacji.....	526
10.2. Definicje funkcji	527
10.2.1. Samokontrola	527
10.2.2. Unikanie przypadków szczególnych	528
10.2.3. Podział zamiast powielania	530
10.2.4. Zbytня przebiegłość nie popłaca	533
10.3. Zarządzanie pamięcią.....	533
10.3.1. Wartości stanu logicznego i fizycznego	538
10.3.2. Parametry fizyczne	541
10.3.3. Systemy przydziału pamięci	545
10.3.4. Zarządzanie pamięcią na poziomie klasy	551
10.3.4.1. Dodawanie własnych mechanizmów zarządzania pamięcią	555
10.3.4.2. Zablockowana pamięć	558
10.3.5. Zarządzanie pamięcią na poziomie obiektu.....	562
10.4. Użycie szablonów w dużych projektach	567
10.4.1. Implementacje szablonów	567
10.4.2. Zarządzanie pamięcią w szablonach.....	568
10.4.3. Wzorce a szablony.....	577
10.5. Podsumowanie	579
Dodatki.....	583
Dodatek A Wzorzec projektowy — Protocol Hierarchy.....	585
Protocol Hierarchy — struktury klas.....	585
Cel	585
Znany też jako	585
Motywacja.....	585
Zakres zastosowania.....	589
Struktura.....	590
Elementy składowe.....	590
Współpraca.....	591
Konsekwencje	591
Implementacja	592
Przykładowy kod.....	603
Znane zastosowania.....	607
Pokrewne wzorce	608

Dodatek B	Implementowanie interfejsu C++ zgodnego ze standardem ANSI C.....	611
	B.1. Wykrywanie błędu alokacji pamięci.....	611
	B.2. Definiowanie procedury main (tylko ANSI C).....	618
Dodatek C	Pakiet określania i analizowania zależności	621
	C.1. Korzystanie z poleceń adept, cdep i ldep.....	622
	C.2. Dokumentacja wiersza polecenia	633
	C.3. Architektura pakietu idep	643
	C.4. Kod źródłowy	646
Dodatek D	Leksykon.....	647
	D.1. Definicje.....	647
	D.2. Główne reguły projektowe	652
	D.3. Poboczne reguły projektowe	653
	D.4. Wskazówki.....	654
	D.5. Zasady	657
	Bibliografia	667
	Skorowidz	671

Rozdział 9.

Projektowanie funkcji

Celem projektowania funkcji jest zapewnienie łatwego i wydajnego dostępu do operacji zdefiniowanych przez abstrakcję. Język C++ zapewnia swobodę definiowania interfejsu na poziomie funkcji. Czy funkcja ma być operatorem, metodą lub wolnym operatorem, w jaki sposób będą przekazywane argumenty oraz w jaki sposób będą przekazywane wyniki — to elementy należące do tego etapu procesu projektowania. Styl programowania to tylko jeden z elementów, które odgrywają rolę w podejmowaniu tego typu decyzji projektowych. Wiele z nich zostanie omówionych w niniejszym rozdziale.

W języku C++ mamy do dyspozycji wiele odmian podstawowych typów reprezentujących liczby całkowite (jak np. `short`, `unsigned`, `long` itp.). Typy te zapewniają dodatkową swobodę. Nierozważne ich wykorzystanie może skomplikować lub nawet osłabić interfejs.

Operator konwersji dla typów definiowanych przez użytkownika umożliwia kompilatorowi wykonywanie niejawnej konwersji na lub z typu definiowanego przez użytkownika. Uważne projektowanie wymaga przeanalizowania możliwych zalet niejawnej konwersji w zestawieniu z niejednoznacznościami oraz możliwością powstania błędów w związku z obniżeniem bezpieczeństwa typów. Niektóre inne funkcje, w przypadku gdy nie zostaną określone jawnie, gdy znajdzie taka potrzeba mogą być zdefiniowane automatycznie przez kompilator. Podjęcie decyzji dotyczących dopuszczalności generowania definicji funkcji przez kompilator wymaga uważnej analizy.

W tym rozdziale opiszemy szkielet projektowania interfejsu komponentu na poziomie pojedynczej funkcji. Omówimy obszerną przestrzeń projektową dostępną dla autorów komponentów i wskażemy decyzje projektowe, które są korzystne lub niekorzystne. Przekonamy się, ile poziomów swobody w przestrzeni projektu interfejsu funkcji można wyeliminować bez strat w skuteczności. Uzyskany szkielet pomoże nam opracowywać interfejsy prostsze, bardziej jednolite i łatwiejsze do utrzymania.

9.1. Specyfikacja interfejsu funkcji

Zgodnie z podstawowymi zasadami przedstawionymi w rozdziale 2., podczas określania interfejsu funkcji w języku C++ należy zwrócić uwagę na kilka aspektów:

1. Czy funkcja jest operatorem, czy nim nie jest?
2. Czy jest wolnym operatorem, czy elementem składowym klasy?
3. Czy jest to metoda wirtualna, czy niewirtualna?
4. Czy jest to metoda czysto wirtualna, czy też nie czysto wirtualna?
5. Metoda stała czy modyfikowalna?
6. Metoda typu `const` czy `nie-const`?
7. Metoda publiczna (`public`), chroniona (`protected`) czy prywatna (`private`)?
8. Wynik zwracany przez wartość, referencję czy wskaźnik?
9. Zwracany wynik typu `const` czy `nie-const`?
10. Argument opcjonalny czy obowiązkowy?
11. Argumenty przekazywane przez wartość, referencję czy wskaźnik?
12. Przekazywane argumenty typu `const` czy `nie-const`?

Istnieją także dwa aspekty dotyczące organizacji, które należy wziąć pod uwagę, pomimo tego, że nie są one częścią logicznego interfejsu:

13. Czy jest to funkcja zaprzyjaźniona, czy też nie?
14. Funkcja typu `inline`, czy nie typu `inline`?

Pomiędzy tymi aspektami istnieje wiele wzajemnych zależności. Zazwyczaj odpowiedź na jedno z pytań ma wpływ na odpowiedź na inne. W dalszej części niniejszego rozdziału omówimy wymienione zagadnienia osobno i podamy wskazówki umożliwiające podjęcie optymalnych decyzji projektowych¹.

9.1.1. Operator czy metoda?

Oprócz operatorów generowanych przez kompilator (np. przypisania), jedynym powodem utworzenia z funkcji operatora jest wygodna notacja wewnątrz klientów. Zwróćmy uwagę, że inaczej niż w przypadku notacji typowej dla funkcji, notacja operatorowa nie zależy od kontekstu — wywołanie funkcji w wyniku interpretacji operatora wywołanego przez metodę będzie takie samo jak w przypadku wywołania w zasięgu pliku². Rozważne wykorzystywanie przeciążania operatorów ma naturalną i oczywistą przewagę nad notacją funkcyjną — w szczególności w przypadku typów logicznych i arytmetycznych definiowanych przez użytkownika.

¹ Zobacz też **meyers**, pozycja 19, str. 70.

² **ellis**, punkt 13.4.1, str. 332.

Przeanalizujemy dwa różne modele składni pokazane na listingu 9.1 odpowiadające dwóm różnym interfejsom dla komponentu zbioru liczb całkowitych `pub_intset`. Na listingu 9.1a pokazano skuteczny sposób zastosowania notacji operatorowej. Natura abstrakcji zbioru powoduje, że znaczenie tych operatorów staje się intuicyjne nawet dla tych programistów, którzy nie znają pokazawanego komponentu. Na listingu 9.1b pokazano odpowiednik składni z zastosowaniem bardziej nieporęcznej notacji funkcyjnej³.

Listing 9.1. Dwa modele składni dla abstrakcji zbioru liczb całkowitych

<pre>#include "pub_intset.h" #include <iostream.h> int main() { pub_IntSet a, b, c, d, e, f; a += 1; a += 3; a += 5; a += 7; b += 1; b += 2; b += 3; b += 4; c = a + b; d = a*b; e = a - b; f = a*b*c + b*c*d + c*d*e; cout << f << endl; return 0; }</pre>	<pre>#include "pub_intset.h" #include <iostream.h> int main() { pub_IntSet a, b, c, d, e, f; a.add(1); a.add(3); a.add(5); a.add(7); b.add(1); b.add(2); b.add(3); b.add(4); c = pub_IntSet::or(a, b); d = pub_IntSet::and(a, b); e = pub_IntSet::sub(a, b); f = pub_IntSet::or(pub_IntSet::or(pub_IntSet::and(pub_IntSet::and(a, b), c), pub_IntSet::and(pub_IntSet::and(b, c), d)), pub_IntSet::and(pub_IntSet::and(c, d), e)); pub_IntSet::print(cout, f) << endl; return 0; }</pre>
--	---

(a) z przeciążaniem operatorów

(b) bez przeciążania operatorów



Podstawowym powodem stosowania mechanizmu przeciążania operatorów powinna być czytelność (w większym stopniu niż łatwość używania).

³ Niektóre metody zdefiniowano jako statyczne, aby umożliwić tę samą symetryczną niejawną konwersję argumentów, jak w przypadku odpowiadających im operatorów (patrz punkt 9.1.5). Styl akapitów głęboko zagnieżdżonych wywołań funkcji na rysunku 9.1b zapożyczono z takich języków, jak LISP i CLOS, gdzie takie konstrukcje występują bardzo często.

W omawianej aplikacji obsługi zbioru liczb całkowitych notacja operatorowa w czytywisty sposób poprawia zarówno czytelność, jak też łatwość używania. Przez *czytelność* rozumiemy zdolność inżyniera oprogramowania do rozróżniania w szybki i precyzyjny sposób znanego kodu treści funkcji od nieznanego kodu źródłowego. *Łatwość używania* dotyczy tego, jak łatwo programista może skutecznie wykorzystać obiekt w celu utworzenia nowego oprogramowania. Zazwyczaj kod źródłowy odczytuje się więcej razy, niż się go zapisuje („w przypadku większości dużych, wykorzystywanych przez długi okres czasu systemów oprogramowania, koszty utrzymania przekraczają koszty wytwarzania od 2 do 4 razy⁴”).



Semantyka przeciążonych operatorów powinna być naturalna, oczywista i intuicyjna dla klientów.

Podczas projektowania często można otrzymać zgrabne i łatwe w użyciu aplikacje operatorów, które nie mają intuicyjnego znaczenia dla programistów, nie znających naszego komponentu. Nierozsądne stare przyzwyczajenia, jak np. zdefiniowanie jednoargumentowego operatora ~ jako składowej klasy `String` w celu odwrócenia kolejności znaków w ciągu, jest nie na miejscu w środowisku projektowym dużej skali. Papierkiem lakmusowym odpowiadającym na pytanie czy zastosować notację operatorową powinno być stwierdzenie, czy istnieje naturalne i intuicyjne znaczenie — natychmiast zrozumiałe dla nowych klientów, które poprawia poziom czytelności (lub przynajmniej go nie pogarsza)⁵.



Syntaktyczne właściwości przeciążonych operatorów dla typów definiowanych przez użytkownika powinny być lustrzaną kopią właściwości zdefiniowanych dla typów podstawowych.

Na poziomie semantycznym dość trudno dostarczyć szczegółowych wskazówek odnośnie tego co jest, a co nie jest intuicyjne. Jednak na poziomie syntaktycznym, biorąc za podstawę implementację podstawowych typów języka, można sformułować kilka zdecydowanych i dobrze zdefiniowanych stwierdzeń.



Wzorowanie syntaktycznych właściwości operatorów zdefiniowanych przez użytkownika na predefiniowanych operatorach C++ pozwala na uniknięcie niespodzianek i sprawia, że sposób ich używania jest łatwiejszy do przewidzenia.

W języku C++ każde wyrażenie ma wartość. Istnieją dwa podstawowe typy wartości — tzw. *lwartości* (ang. *lvalue*) oraz *pwartości* (ang. *rvalue*)⁶. *Lwartość* to taka wartość, dla której można wyznaczyć adres. Jeżeli *lwartość* może się znaleźć po lewej stronie wyrażenia przypisania, mówi się o niej, że jest modyfikowalna, w innym przypadku określa się ją jako *lwartość niemodyfikowalną*⁷. Do *pwartości* nie można przypisać wartości, ani

⁴ sommerville, punkt 1.2.1, str. 10.

⁵ Patrz też **cargill**, rozdział 5., str. 91.

⁶ Pojęcia te pochodzą z klasycznego języka C: pojęcie *lwartość* oznacza, że wartość wyrażenia może się znaleźć po lewej stronie instrukcji przypisania, natomiast *pwartość* może się znaleźć wyłącznie po jej prawej stronie. Wraz z pojawieniem się konstrukcji `const` w języku C++ i ANSI C, *lwartości* dzieli się teraz na dwie odmiany: modyfikowalne i niemodyfikowalne (patrz **stroustrup**, punkt 2.1.2, str. 46 – 47).

⁷ **ellis**, podrozdział 3.7, str. 25 – 26.

nie można pobrać jej adresu⁸. Najprostszą wartością jest identyfikator zmiennej. Jeżeli zmienna nie jest zadeklarowana jako `const`, jest to wartość modyfikowalna. Niektóre operatory, jak np. operator przypisania (`=`) i jego odmiany (`+=`, `-=`, `*=`, `/=`, `^=`, `&=`, `|=`, `~=`, `%=`, `>>=`, `<<=`), preinkrementacji (`++x`) i predekrementacji (`--x`) zastosowane do typów podstawowych, zwracają modyfikowalne *wartości*. Operatory te zawsze zwracają zapisywalną referencję do modyfikowanego argumentu. Np. hipotetyczna definicja operatorów dla podstawowego typu `double` (w przypadku jego implementacji jako klasy C++) może mieć postać taką, jak pokazano na listingu 9.2.

Listing 9.2. Hipotetyczna implementacja podstawowego typu `double`

```
class double {                                // Uwaga: niepoprawne w języku C++
    //...
public:
    double() {}
    double(int);
    double(const double&);
    ~double() {}

    double& operator=(const double& d);
    double& operator+=(const double& d);
    double& operator-=(const double& d);
    double& operator*=(const double& d);
    double& operator/=(const double& d);

    double& operator++();                      // preinkrementacja ++x
    double& operator--();                      // predekrementacja --x
    double operator++(int);                    // postinkrementacja x++
    double operator--(int);                    // postdekrementacja x--

    double *operator&();                      // jednoargumentowy operator adresu
    const double *operator&() const;          // jednoargumentowy operator adresu
};

double operator+(const double& d);            // jednoargumentowy +
double operator-(const double& d);            // jednoargumentowy -

int operator!(const double& d);               // jednoargumentowy logiczny operator "not"

int operator&&(const double& left, const double& right);
int operator||(const double& left, const double& right);

double operator+(const double& left, const double& right);
double operator-(const double& left, const double& right);
double operator*(const double& left, const double& right);
double operator/(const double& left, const double& right);

int operator==(const double& left, const double& right);
int operator!=(const double& left, const double& right);
int operator<(const double& left, const double& right);
int operator<=(const double& left, const double& right);
int operator>(const double& left, const double& right);
int operator>=(const double& left, const double& right);
```

⁸ Pola bitowe są wyjątkiem w tym sensie, że mogą się znaleźć po lewej stronie wyrażenia przypisania, ale zgodnie z ARM (ellis, podrozdział 9.6, str. 184) nie można wyznaczyć ich adresu. Zasada ta dotyczy także zmiennych tymczasowych typów zdefiniowanych przez użytkownika, które nie posiadają nazwy (patrz punkt 9.1.2).

Inne operatory pokazane na listingu 9.2 zwracają wartość, ponieważ nie ma możliwości zwrócenia odpowiedniej lwartości. W przypadku symetrycznych operatorów binarnych (jak np. + oraz *) wartość do zwrócenia nie jest ani argumentem lewym, ani prawym, ale nową wartością uzyskaną na podstawie obydwu, a zatem wynik musi być zwrócony przez wartość⁹. Operatory równości (==, !=) oraz operatory relacyjne (<, <=, >, >=) zawsze zwracają wartość typu `int` o wartości 0 lub 1, a zatem i w tym przypadku żaden z argumentów wejściowych nie jest odpowiednią wartością do zwrócenia. Operatory postinkrementacji i postdekrementacji to interesujący przypadek specjalny w tym sensie, że są to jedyne operatory, które modyfikują obiekt, a zatem nie ma odpowiedniej lwartości do zwrócenia:

```
double double::operator++(int)           double double::operator--(int)
{
    double tmp = *this;
    ++*this;
    return tmp;
}
{
    double tmp = *this;
    --*this;
    return tmp;
}
```

Jako bardziej subtelny przykład przeanalizujemy dwa modele składni odpowiadające abstrakcji ogólnej tabeli symboli, pokazane na listingu 9.3. W obu przypadkach na podstawie parametru typu `int` tworzona jest tabela symboli, dodawane są dwa symbole, a następnie poszukuje się parametru *foo* według nazwy. Ponieważ możliwe jest, że w tabeli nie ma symbolu o określonej nazwie, zatem funkcja wyszukująca nie powinna zwracać wyniku przez wartość, ani przez referencję — tak więc wynik jest zwracany przez wskaźnik (zwróćmy uwagę na to, w jaki sposób i do jakiego stopnia wykorzystano zalety enkapsulacji). Porównajmy tę sytuację z zastosowaniem operatora `[]` w odniesieniu do tablicy wartości `int`. Spodziewamy się uzyskać referencję do poindeksowanej wartości, nie zaś wskaźnik, który może mieć wartość `null`. Ta różnica w składni pomiędzy zastosowaniem operatora `[]` na listingu 9.3a oraz zastosowaniem operatora `[]` dla typów podstawowych powoduje, że notacja z wywołaniem funkcji pokazana na listingu 9.3b jest w tym przypadku lepsza. Zarezerwowanie notacji operatorowej dla tych przypadków, kiedy składnia stanowi lustrzane odbicie odpowiadającej jej składni dla typów podstawowych, wzmacnia skuteczność przeciążania operatorów.

Listing 9.3. Dwa modele składni dla abstrakcji ogólnej tabeli symboli

<pre>#include "gen_symtab.h" int main() { gen_SymTab<int> s; s("foo", 1); // operator() s("bar", 2); //(zly pomysl) const int *val = s["foo"]; // ... }</pre>	<pre>#include "gen_symtab.h" int main() { gen_SymTab<int> s; s.add("foo", 1); s.add("bar", 2); const int *val = s.lookup("foo"); // ... }</pre>
--	--

(a) z przeciążaniem operatorów

(b) bez przeciążania operatorów

Na listingu 9.4. zestawiono deklaracje większości operatorów języka C++ w postaci, w jakiej użyto by ich w odniesieniu do podstawowych typów języka (podstawowe operatory `->`, `->*`, `()` nie dają zbyt wielu informacji).

⁹ Bardziej szczegółowe objaśnienie znajduje się w **meyers**, pozycja 23, str. 82 – 84.

Listing 9.4. Podsumowanie właściwości niektórych podstawowych operatorów

```

// operatory o podobnych deklaracjach
class T {
    T& operator++();           // ++x --x (prefiks)
    T operator++(int);        // x++ x-- (postfiks)

    T* operator&();           // &x (jednoargumentowy)
    const T* operator&() const; // &x (jednoargumentowy)

    T& operator=(const T&);    // = += -= *= /= %= <=> &= ^= |=

};

T operator-(const T&);        // - + ~ (jednoargumentowe)
int operator!(const T&);      // ! (jednoargumentowy)

T operator+(const T&, const T&); // + - * / << >> % & ^ |
int operator==(const T&, const T&); // == != < <= > >=
int operator&&(const T&, const T&); // && ||

// w przypadku typów w postaci wskaźników (np. P = T*)

class P {
    T& operator[](int) const; // dostęp do indeksowanej tablicy (dwuargumentowy)
    T& operator*() const;     // uzyskanie wskazywanej danej (jednoargumentowy)
};

// w przypadku typów w postaci wskaźników na stałe (np. PC = const T*)

class PC {
    const T& operator[](int) const; // dostęp do indeksowanej tablicy (dwuargumentowy)
    const T& operator*()const;      // uzyskanie wskazywanej danej (dwuargumentowy)
};

```

Zwróćmy uwagę, że operatory jednoargumentowe, które nie modyfikują argumentów, nie muszą być składowymi. Przykładowo, jednoargumentowy operator `!` z powodzeniem działa dla typu zdefiniowanego przez użytkownika, jak np. `ostream`, pomimo tego, że dla tego typu nie zdefiniowano operatora `!`:

```

#include "iostream.h"
void g(ostream& out)
{
    if (!out) {
        cerr << "strumień wyjściowy jest niewłaściwy" << endl ;
        return;
    }
    // ...
}

```

Kod pokazany powyżej działa dlatego, ponieważ typ `ostream` „wie”, w jaki sposób ma się przekształcić na podstawowy typ (`void *`), dla którego zdefiniowano operator `!`. Gdyby operator `!` potraktowano jako składową hipotetycznej definicji klasy `void *`, nie można byłoby wykonać zdefiniowanej przez użytkownika konwersji, a wykonanie pokazanego powyżej kodu zakończyłoby się błędem kompilacji.

Aby wyłączyć możliwość niejawnej konwersji argumentów „wolnego” jednoargumentowego operatora `!`, należy zdefiniować operację `!` jako metodę — np. `obj.not()` zamiast operatora¹⁰.

¹⁰Patrz też **murray**, podrozdział 2.5, str. 44.

9.1.2. Wolny operator czy składowa klasy?

Decyzja dotycząca tego, czy funkcja definiująca operator ma być składową klasy, czy wolną funkcją, zależy od tego, czy pożądana jest niejawna konwersja typu skrajnego lewego operandu. Jeżeli operator modyfikuje ten operand, wówczas taka konwersja nie jest pożądana.



Język C++ sam w sobie jest obiektywnym i właściwym standardem, według którego należy modelować operatory definiowane przez użytkownika.

Zastanówmy się nad tym, co mogłoby się zdarzyć, gdybyśmy zdefiniowali operator konkatencji (`+=`) dla klasy `String` jako wolną funkcję, zamiast emulacji podejścia zapożyczonego z typów podstawowych. Zgodnie z tym, co pokazano na rysunku 9.1, utworzenie operatora `+=` jako wolnej funkcji umożliwiło niejawną konwersję jego lewego operandu `const char*` na tymczasowy obiekt `pub_String` (oznaczony tu jako `t005`) o wartości `foo`. Choć w przypadku typów podstawowych ta zmienna tymczasowa byłaby pwarością, to właśnie do tymczasowego obiektu `pub_String` dołączono wartość „bar” (bez powstania błędu kompilacji)¹¹. Ponieważ takie działanie zaskoczyłoby i zdenerwowało użytkowników, dobrze byłoby, aby je wyłączyć.

Rysunek 9.1.
Efekt implementacji
operator `+=`
jako wolnej funkcji

```
// pub_String.h
// ...
class pub_String {
    // ...
public:
    pub_String(const char *str);
};

pub String& operator+=(pub_String& left, const pub_String& right);
// definicja wolnej funkcji konkatencji lancuchow znakow

// ...
```

```
#include "pub_string.h"

void f()
{
    pub_String a("tar");
    const char *b = "foo";
    pub_String c("bar");
    b += c; // brak efektu

    (pub_String) t005
    +=
    c
    (pub_String) t005 ← („bar” połączony z tymczasową kopią obiektu pub_String)

    a += b += c; // a zawiera teraz "tarfoobar", ale
                // zawartosc b pozostaje bez zmian
```

¹¹ Obecnie w języku C++ dozwolone jest modyfikowanie nienazwanych tymczasowych zmiennych typu zdefiniowanego przez użytkownika nieposiadających nazwy. Patrz **murray**, punkt 2.7.3, str. 53 – 55.

Z drugiej strony spodziewamy się, że niektóre operatory (np. + oraz ==) będą działać bez względu na kolejność ich argumentów. Przeanalizujmy operator +, który służy do konkatencji dwóch ciągów znaków i zwraca wynik swojego działania przez wartość. Język C++ pozwala na zdefiniowanie operatora + jako składowej lub nieskładowej. To samo dotyczy operatora ==. Jeżeli zdecydujemy się na zdefiniowanie tych operatorów jako składowych, wówczas narazimy się na możliwość następującego, nienormalnego działania naszych klientów:

```
void f()
{
    pub_String s("foo"), t("");
    int i;

    t = s + "bar";           // dobrze
    t = "bar" + s;          // błąd
    i = s == "bar";         // dobrze
    i = "bar" == s;         // błąd
}
```

Problem polega na tym, że deklaracje:

```
pub_String::operator+(const String& right)
```

oraz

```
pub_String::operator==(const String& right)
```

umożliwiają niejawną konwersję typu `char *` na `pub_String` po prawej stronie za pomocą konstruktora następującej postaci:

```
pub_String::pub_String(const char *)
```

natomiast taka konwersja dla argumentu po lewej stronie nie jest możliwa¹². Jeżeli operatory te będą wolne, problem symetrii dla klasy `pub_String` zostanie rozwiązany, do czasu dodania operatora konwersji:

```
pub_String::operator const char *() const
```

Na listingu 9.5 pokazano problem powstały w wyniku dodania operatora konwersji (rzutowania) z typu `pub_String` na `const char *`. Co jest dość dziwne, dwa niewątpliwie podobne operatory `==` oraz `+` nie są identyczne pod względem przeciążania, w co (niestety naiwnie) chcielibyśmy wierzyć. Różnica polega na tym, że teraz istnieją dwa sposoby interpretacji operatora `==`:

1. Jawna konwersja typu `char *` na `pub_String` i porównanie za pomocą `operator==(const String&, const String&)`.
2. Niejawna konwersja typu `pub_String` na `const char *` i porównanie za pomocą wbudowanego operatora `==` dla typów wskaźnikowych.

Taki problem nie istnieje dla operatora +, ponieważ w języku C++ nie ma sposobu „dodania” dwóch typów wskaźnikowych i dlatego w tym przypadku nie ma niejednoznaczności.

¹²ellis, punkt 13.4.2, str. 333.

Listing 9.5. *Niejednoznaczności wynikające z zastosowania dwóch operatorów konwersji*

```
//pub_String.h
// ...
class pub_String {
    //...
public:
    pub_String(const char *pcc);
    //...
    operator const char *() const;    // <== nowy operator konwersji
};

int operator==(const String& left, const String& right);
String operator+(const String& left, const String& right);

// ...
```

```
#include "pub_String.h"

void f()
{
    pub_String s("foo"), t("");
    int i;

    t = s + "bar";           // dobrze
    t = "bar" + s;          // dobrze
    i = s == "bar";          // błąd (niejednoznaczne)
    i = "bar" == s;          // błąd (niejednoznaczne)
    i = strlen(s);           // dobrze
}
```

W przypadku klasy `String` wykorzystywanej w praktyce nie będziemy polegać na niejawniej konwersji w celu uzyskania wartości znakowej, ze względu na obawę, że taka dodatkowa konstrukcja i destrukcja spowoduje zbyt ni spadek wydajności. Zamiast tego zdefiniujemy osobne przeciążone wersje operatora `+`, których zadaniem będzie jak najbardziej wydajna obsługa każdej z trzech możliwości, a tym samym rozwiązanie problemów jednoznaczności.



Niespójności powstałe w wyniku przeciążania operatorów mogą być oczywiste, denerwujące i kosztowne dla użytkowników.

Zgodnie z tym, co pokazano na listingu 9.6, dla umożliwienia zaakceptowania wartości `const char *` po lewej stronie operatora `==` jesteśmy zmuszeni do zdefiniowania co najmniej jednej z funkcji operatorów porównania jako wolnej funkcji.

Listing 9.6. *Wynik zaimplementowania operatora `==` jako funkcji składowej*

```
class pub_String {
    // ...
public:
    pub_String(const char *pcc);
    operator const char *() const;
    int operator==(const char *pcc) const; // zły pomysł: (asymetryczny)
    // Umożliwia konwersje zdefiniowane przez użytkownika tylko
    // dla argumentów po prawej stronie operatora.
};
```

```

};

int operator==(const pub_String& left, const pub_String& right);
int operator==(const char *left, const pub_String& right);
    // Umożliwia symetryczne konwersje zdefiniowane przez użytkownika
    // dla argumentów zarówno po prawej, jak po lewej stronie operatora.

struct Foo {
    Foo();
    operator const pub_String& () const;
    // Niejawna konwersja typu Foo na pub_String.
};

struct Bar {
    Bar();
    operator const char *() const;
    // Niejawna konwersja typu Bar na (const char *).
};

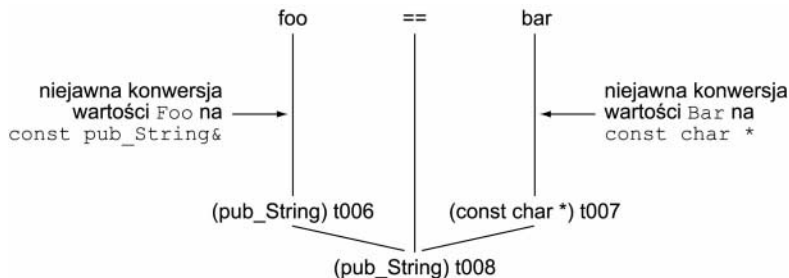
void g()
{
    Foo foo;
    Bar bar;
    if (bar == foo) {                // dobrze: Bar =na=> (const char *)
        // ...                      Foo =na=> (const pub_String&)
    }
    if (foo == bar) {                // błąd: Foo =NIE=> (const pub_String&)
        // ...                      Bar =na=> (const char *)
    }
}

```

Na czym polega problem zdefiniowania jednej wersji funkcji `operator==` jako składowej w przypadku, gdy zdefiniowano wszystkie trzy wersje tego operatora? Otóż problem polega na tym, że brak symetrii mógłby zaskoczyć użytkowników. W przypadku gdy jeden obiekt może być niejawnie przekształcony na `pub_String`, a drugi na `const char*`, spodziewamy się, że porządek porównania jest nieistotny. Jeżeli zatem konstrukcja `bar==foo` kompiluje się bez problemów, podobnie powinno być w przypadku zapisu `foo==bar` (wyniki wykonania obu konstrukcji powinny być identyczne). Jednak jeżeli wersja:

```
int operator==(const pub_String&, const char *);
```

nie będzie dostępna jako wolna funkcja, wówczas nie będzie sposobu przeprowadzenia następującej konwersji:



Wniosek jest taki, że operator `==` zawsze powinien być zdefiniowany jako wolna funkcja niezależnie od zastosowania innych funkcji. Te same powody dotyczą innych operatorów dwuargumentowych, które nie modyfikują żadnego z operandów i zwracają swój wynik przez wartość.

Przykład, jaki daje sam język, jest bezstronnym i użytecznym modelem, który może służyć klientom do tworzenia podstawowych syntaktycznych i aksjomatycznych właściwości operatorów. Celem modelowania podstawowych operacji nie jest umożliwianie niejawnych konwersji samo w sobie, ale raczej zapewnienie symetrii po to, by uniknąć niespodzianek. W przypadku zastosowania przeciążania operatorów w szerokim zakresie należy się spodziewać, że abstrakcja może być wykorzystywana wielokrotnie w wielu sytuacjach. Użytkownicy komponentów wielokrotnego użytku docenią spójny i profesjonalny interfejs niezależnie od syntaktycznych niespodzianek. Zwróćmy uwagę na to, że w języku C++ wymagane jest, aby zdefiniować jako składowe następujące operatory¹³:

<code>=</code>	przypisania,
<code>[]</code>	indeksu,
<code>-></code>	dostępu do składowej klasy,
<code>()</code>	wywołania funkcji,
<code>(T)</code>	konwersji,
<code>new</code>	przydział pamięci (statyczny),
<code>delete</code>	zwolnienie pamięci (statyczne).

9.1.3. Metoda wirtualna czy niewirtualna?

Dynamiczne wiązanie umożliwia definiowanie metod, do których dostęp odbywa się za pomocą klasy podstawowej, przez rzeczywisty podtyp obiektu, w odróżnieniu do typu wskaźnika lub referencji użytej w wywołaniu. W celu zapewnienia dynamicznego wiązania funkcję należy zadeklarować jako wirtualną (`virtual`). W języku C++ wirtualne mogą być tylko metody. Jednak wniosek, że polimorficzne działanie operatora wymaga, aby stał się funkcją składową, gdy w innym przypadku byłby wolną funkcją, jest błędny.

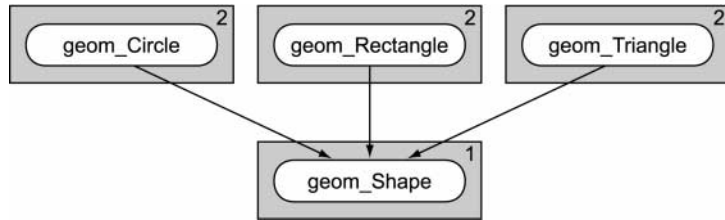


W celu osiągnięcia polimorficznego działania operatorów nie trzeba naruszać zagadnień syntaktycznych, jak np. symetrycznej niejawnej konwersji operatorów dwuargumentowych.

Na rysunku 9.2 pokazano, w jaki sposób operatory symetryczne mogą i powinny pozostać operatorami wolnymi pomimo zastosowania poliformizmu. Zamiast przekształcania każdego z sześciu operatorów porównań i relacji na wirtualne metody klasy, opracowano jedną wirtualną metodę porównawczą. Te sześć operatorów w dalszym ciągu będzie działać symetrycznie bez względu na niejawną konwersję dowolnego typu.

¹³ ellis, podrozdział 12.3c, str. 306; stroustrup⁹⁴, punkt 3.6.2, str. 82 – 83.

Rysunek 9.2.
*Polimorficzne
 porównanie figur
 geometrycznych
 za pomocą wolnych
 operatorów*



```
// geom_shape.h
#ifndef INCLUDED_GEOM_SHAPE
#define INCLUDED_GEOM_SHAPE

class geom_Shape {
public:
    virtual ~geom_Shape();
    virtual const void *classId() const = 0;
    virtual int compare(const geom_Shape& shape) const = 0;
    // Zwraca liczbę ujemną, zero lub dodatnią zależnie od tego,
    // czy obiekt geom_Shape jest odpowiednio mniejszy, równy, czy
    // większy od
    // określonego innego obiektu geom_Shape.
};

inline int operator==(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) == 0; }

inline int operator!=(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) != 0; }
inline int operator<=(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) <= 0; }
inline int operator<(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) < 0; }
inline int operator>=(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) >= 0; }
inline int operator>(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) > 0; }

#endif
```

Operatory porównań często mają sens nawet wtedy, gdy operatory relacji nie mają sensu (weźmy pod uwagę abstrakcję punktu). Czasami sortowanie heterogenicznych kolekcji pozwala na uzyskanie wydajniejszego dostępu. W takich przypadkach przydaje się wykorzystanie porządkowania (dowolnego typu). Wirtualna metoda `classId()` pokazana na rysunku 9.2 pozwala na zdefiniowanie własnego identyfikatora typu fazy wykonania¹⁴. Dzięki wykorzystaniu tego identyfikatora można sortować figury tego samego typu wykorzystując zdefiniowany dla nich indywidualny porządek, natomiast sortowanie innych typów można zdefiniować za pomocą innego (zupełnie dowolnego) porównania. Implementację klasy `geom_Circle` wykorzystywanej do porządkowania figur pokazano na liście 9.7.



Metody wirtualne implementują różnice w działaniu, natomiast pola — różnice w wartościach.

¹⁴ Patrz **ellis**, podrozdział 10.2, str. 212 – 213.

Listing 9.7. Implementacja polimorficznego porównania dla klasy *geom_Circle*

```

// geom_circle.h
#ifndef INCLUDED_GEOM_CIRCLE
#define INCLUDED_GEOM_CIRCLE

#ifndef INCLUDED_GEOM_SHAPE
#include "geom_shape.h"
#endif

class geom_Circle : public geom_Shape {
    static const void *d_classId_p;
    double d_radius;

public:
    geom_Circle(double radius) : d_radius(radius) {}
    geom_Circle(const geom_Circle& circle) : d_radius(circle.d_radius) {}
    ~geom_Circle(const geom_Circle& circle);
    geom_Circle& operator=(const geom_Circle& circle) {
        d_radius = circle.d_radius; return *this; }
    const void *classId() const { return d_classId_p; }
    int compare(const geom_Shape& shape) const;    // wirtualna
    int compare(const geom_Circle& circle) const;  // niewirtualna
};

inline int operator<(const geom_Circle& left, const geom_Circle& right) {
    return left.compare(right) < 0; }

// ... (definicje pozostałych 5 operatorów symetrycznych pominięto)

#endif

// geom_circle.c
#include "geom_circle.h"
const void *geom_Circle::d_classId_p = &d_classId_p; // typ id fazy
wykonania
geom_Circle::~geom_Circle() {} // pusta i nie inline (patrz punkt 9.3.3)

int geom_Circle::compare(const geom_Shape& shape) const
{
    return shape.classId() == d_classId_p ?
        compare((const geom_Circle&) shape) : // porównanie instancji
        d_classId_p < shape.classId() ? -1 : 1; // porównanie typów
}

int geom_Circle::compare(const geom_Circle& circle) const
{
    return d_radius < circle.d_radius ? -1 : d_radius > circle.d_radius;
}

```

Ogólnie rzecz biorąc, metody wirtualne służą do opisanía różnic w działaniu pomiędzy typami pochodzącymi od wspólnej klasy bazowej, natomiast wirtualne pola służą do opisanía różnic wartości i pozwalają na uniknięcie stosowania dziedziczenia¹⁵. Np. nie będziemy definiować klasy-protokołu *art_Color*, by potem utworzyć klasy pochodne *art_Red*, *art_Blue* i *art_Yellow*. Lepszym rozwiązaniem w tej sytuacji będzie zdefiniowanie

¹⁵Patrz **cargill**, rozdział 1, str. 16 – 19.

jednej (w pełni odizolowanej) klasy `art_Color`, w której zapisano by jeden z kilku wymienionych kolorów w postaci typu wyliczeniowego. Zastosowanie metod wirtualnych jest jednak skuteczną techniką rozwiązywania problemu zależności zarówno fazy kompilacji, jak i konsolidacji (patrz punkt 6.4.1). Z tego powodu można zdecydować się na utworzenie konkretnej klasy pochodnej na podstawie ogólnej klasy `art_Color`.



Ukrywanie: Metoda ukrywa funkcję o tej samej nazwie zadeklarowaną w ramach klasy podstawowej lub w zasięgu pliku.

Przeciążanie: Funkcja przeciąża inną funkcję o tej samej nazwie zdefiniowaną w tym samym zasięgu.

Przesłanianie: Metoda przesłania identyczną metodę, którą w klasie bazowej zadeklarowano jako wirtualną.

Przedefiniowywanie: Domyślna definicja funkcji jest nieodwracalnie zastępowana inną definicją.

Na koniec zdefiniujemy cztery powszechnie używane (często błędnie), podobne do siebie, pojęcia służące do opisu funkcji i efektów ich działania na inne funkcje (*ukrywanie*, *przeciążanie*, *przesłanianie* i *przedefiniowywanie*). O różnych funkcjach o tej samej nazwie mówi się, że są *przeciążone* tylko wtedy, gdy zadeklarowano je w tym samym zasięgu. Jeżeli metodę klasy pochodnej zadeklarowano z identycznym interfejsem do metody zadeklarowanej w klasie bazowej jako wirtualną, wówczas mówi się, że ta funkcja *przesłania* metodę klasy bazowej. We wszystkich pozostałych przypadkach, nazwa funkcji *ukrywa* wszystkie inne funkcje o identycznej nazwie w tym samym zasięgu, niezależnie od ich sygnatur argumentów. Do funkcji ukrytych w danym zasięgu nie ma bezpośredniego dostępu, chociaż można uzyskać do nich dostęp za pomocą operatora zasięgu (`::`). Jeżeli jednak przedefiniujemy funkcję (np. globalny operator `new` lub jednoargumentowy operator klasy `&`), czyli zamienimy jej definicję, wówczas jej poprzednia definicja nie będzie już dostępna¹⁶.



Należy unikać ukrywania metod klasy bazowej w klasie pochodnej.

Należy uważać, aby nie ukrywać definicji metod klasy bazowej w obrębie klas pochodnych. Nie należy zwłaszcza tworzyć nowych definicji dla niewirtualnych metod w klasie pochodnej, ponieważ w takim przypadku takie metody będą wrażliwe na typy wskaźników oraz adresów, spod których będą wywoływane¹⁷. Umożliwienie powstania zależności typu wskaźników lub adresów od tego, która funkcja będzie wywołana, jest działaniem antyintuicyjnym, subtelnym i powodującym powstanie błędów. Ukrywanie metod zdefiniowanych w klasach bazowych nie wyklucza możliwości ich użycia, a jedynie utrudnia ich wykorzystanie. Aby wywołać ukrytą metodę, zawsze można wykonać pewne działania ze wskaźnikiem lub posłużyć się operatorem zasięgu. Najlepiej starać się unikać ukrywania metod. Przykłady wzorców projektowych obejmujących wykorzystanie metod wirtualnych, wielokrotnego dziedziczenia oraz identyfikacji typu w fazie wykonania zaprezentowano w dodatku C.

¹⁶ellis, podrozdział 10.2, str. 210 oraz podrozdział 13.1, str. 310.

¹⁷Patrz meyers, pozycja 37, str. 130 – 132.