

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

Wyjątkowy język C++. 40 nowych łamigłówek, zadań programistycznych i rozwiązań

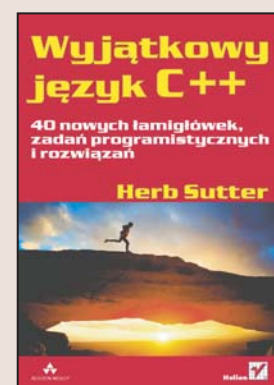
Autor: Herb Sutter

Tłumaczenie: Marcin Miklas

ISBN: 83-7361-712-4

Tytuł oryginału: [More Exceptional C++](#)

Format: B5, stron: 280



Poznaj skuteczne rozwiązania problemów,
jakie napotkasz podczas codziennej pracy programisty

- Prawidłowo zoptymalizuj kod źródłowy
- Zaprojektuj efektywną obsługę wyjątków
- Naucz się odpowiednio korzystać z przestrzeni nazw

Nauka każdego nowego zagadnienia przebiega zdecydowanie szybciej i efektywniej pod kierunkiem osoby znającej temat. Nie inaczej jest w przypadku nauki programowania w języku C++. Programowanie w tym języku wymaga dogłębnego opanowania możliwości biblioteki standardowej, inżynierii oprogramowania i wielu innych tematów. Rozwiązując samodzielnie zagadki, jakie często pojawiają się podczas poznawania języka C++ lepiej zapamiętujemy ich rozwiązania, dzięki czemu łatwiej możemy zastosować je w codziennej pracy.

Książka „Wyjątkowy język C++. 40 nowych łamigłówek, zadań programistycznych i rozwiązań” to zbiór zagadnień związanych z niemal wszystkimi aspektami programowania w C++, przedstawionych w formie zagadek z rozwiązaniami. Opisuje zarówno podstawy stosowania szablonów, biblioteki standardowej i projektowanie klas, jak i tematy zaawansowane – przestrzenie nazw, wyjątki, dziedziczenie i zarządzanie zasobami. Czytając ją, przekonasz się, w jaki sposób można użyć pozornie niezwiązanych ze sobą elementów do opracowania nowych i oryginalnych rozwiązań częstych problemów.

- Poprawne stosowanie predykatów
- Kontenery i wskaźniki
- Korzystanie z szablonów
- Optymalizacja kodu
- Bezpieczna obsługa wyjątków
- Mechanizmy dziedziczenia i polimorfizmu
- Zarządzanie zasobami i pamięcią
- Makrodefinicje
- Przestrzenie nazw



Spis treści

Słowo wstępne	7
Przedmowa.....	9
Rozdział 1. Programowanie uogólnione i biblioteka standardowa języka C++	13
Zagadnienie 1. Przełączanie strumieni	14
Zagadnienie 2. Predykaty. Część 1 — co usuwa funkcja remove()?	18
Zagadnienie 3. Predykaty. Część 2 — pamięć stanu	22
Zagadnienie 4. Szablony rozszerzalne — dziedziczenie czy cechy charakterystyczne? ..	29
Zagadnienie 5. Słowo kluczowe typename.....	42
Zagadnienie 6. Kontenery, wskaźniki i kontenery, które nie są kontenerami.....	46
Zagadnienie 7. Stosowanie kontenerów vector i deque.....	55
Zagadnienie 8. Stosowanie kontenerów set i map.....	62
Zagadnienie 9. Równoważny kod?.....	67
Zagadnienie 10. Specjalizacja i przeciążanie szablonów.....	72
Zagadnienie 11. Gra Mastermind	77
Rozdział 2. Optymalizacja a efektywność	89
Zagadnienie 12. Słowo kluczowe inline.....	89
Zagadnienie 13. Opieszała optymalizacja. Część 1 — klasa String	93
Zagadnienie 14. Opieszała optymalizacja. Część 2 — wprowadzamy opieszałość.....	96
Zagadnienie 15. Opieszała optymalizacja. Część 3 — iteratory i referencje.....	100
Zagadnienie 16. Opieszała optymalizacja. Część 4 — środowiska wielowątkowe	108
Rozdział 3. Zagadnienia i techniki związane z bezpieczną obsługą wyjątków	121
Zagadnienie 17. Błędy konstruktora. Część 1 — czas życia obiektu.....	121
Zagadnienie 18. Błędy konstruktora. Część 2 — wchłanianie	125
Zagadnienie 19. Wyjątki nieprzechwycone.....	132
Zagadnienie 20. Problem niezarządzanego wskaźnika. Część 1 — obliczanie parametrów.....	137
Zagadnienie 21. Problem niezarządzanego wskaźnika. Część 2 — co z szablonem auto_ptr?.....	140

Zagadnienie 22. Projektowanie klas zapewniających bezpieczną obsługę wyjątków. Część 1 — operator przypisania	146
Zagadnienie 23. Projektowanie klas zapewniających bezpieczną obsługę wyjątków. Część 2 — dziedziczenie	154
Rozdział 4. Dziedziczenie i polimorfizm	159
Zagadnienie 24. Dlaczego dziedziczenie wielobazowe?	159
Zagadnienie 25. Emulowanie dziedziczenia wielobazowego	163
Zagadnienie 26. Dziedziczenie wielobazowe i problem bliźniąt syjamskich	166
Zagadnienie 27. Metody (nie)czysto wirtualne	170
Zagadnienie 28. Polimorfizm kontrolowany	175
Rozdział 5. Zarządzanie zasobami i pamięcią.....	179
Zagadnienie 29. Stosowanie szablonu auto_ptr	179
Zagadnienie 30. Inteligentne wskaźniki jako składowe. Część 1 — problem z auto_ptr	186
Zagadnienie 31. Inteligentne wskaźniki jako składowe. Część 2 — kurs na ValuePtr	191
Rozdział 6. Funkcje swobodne i makrodefinicje	203
Zagadnienie 32. Rekurencyjne deklaracje	203
Zagadnienie 33. Imitowanie funkcji zagnieżdżonych.....	208
Zagadnienie 34. Dyrektywy preprocesora	216
Zagadnienie 35. Makrodefinicje.....	219
Rozdział 7. Zagadnienia różne	225
Zagadnienie 36. Inicjalizacja.....	225
Zagadnienie 37. Deklaracje zapowiadające.....	228
Zagadnienie 38. Instrukcja typedef	229
Zagadnienie 39. Przestrzenie nazw. Część 1 — deklaracje i dyrektywy using	232
Zagadnienie 40. Przestrzenie nazw. Część 2 — migracja do przestrzeni nazw	235
Postówie	245
Dodatek A Optymalizacje, które nie są optymalizacjami	247
Dodatek B Wyniki testów	263
Bibliografia	271
Skorowidz	273

Rozdział 4.

Dziedziczenie i polimorfizm

Czym jest programowanie obiektowe bez odrobiny dziedziczenia i polimorfizmu?

Dziedziczenie, chociaż często nadużywane, jest nadal ważnym narzędziem — i dotyczy to także dziedziczenia wielobazowego. Dziedziczenie wielobazowe jest szczególnie przydatne, gdy zachodzi potrzeba połączenia bibliotek różnych producentów. Przy takim łączeniu daje się we znaki problem tzw. bliźniąt syjamskich — pokażemy tu, jak go uniknąć. Zaprezentujemy także wiele uprawnionych (i kilka nieuprawnionych) zastosowań funkcji wirtualnych, kodowanie rozwiązań alternatywnych wobec dziedziczenia wielobazowego i kontrolowanie możliwości używania związków dziedziczenia.

Zagadnienie 24. Dlaczego dziedziczenie wielobazowe?

Stopień trudności: 6

Niektóre języki, łącznie ze standardem języka SQL99, zmagają się z pytaniem, czy obsługiwać dziedziczenie wyłącznie pojedyncze czy również wielobazowe. Zagadnienie to zachęca do rozpatrzenia tych kwestii.

1. Co to jest dziedziczenie wielobazowe i jakie dodatkowe możliwości lub komplikacje wprowadza włączenie takiego dziedziczenia do języka C++?
2. Czy dziedziczenie wielobazowe jest kiedykolwiek konieczne? Jeśli tak, pokaż jak najczęściej przykładowych sytuacji i uzasadnij, dlaczego język powinien obsługiwać dziedziczenie wielobazowe. Jeśli nie, uzasadnij dlaczego dziedziczenie pojedyncze (DP), być może połączone z interfejsami w stylu języka Java, dorównuje lub przewyższa wielobazowe i dlaczego język nie powinien zawierać takiego dziedziczenia.



Rozwiązanie

1. Co to jest dziedziczenie wielobazowe i jakie dodatkowe możliwości lub komplikacje wprowadza włączenie takiego dziedziczenia do języka C++?

W skrócie, dziedziczenie wielobazowe oznacza możliwość dziedziczenia po więcej niż jednej bezpośredniej klasie bazowej.

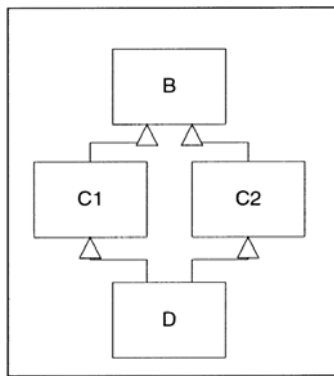
Na przykład:

```
class Derived : public Base1, private Base2
{
    //...
};
```

Dopuszczenie dziedziczenia wielobazowego wprowadza możliwość, że klasa może mieć tę samą (bezpośrednią lub pośrednią) klasę bazową występującą jako przodek więcej niż jeden raz. Prosty przykład to klasyczny rombkształtny diagram dziedziczenia pokazany na rysunku 4.

Rysunek 4.

*Zgubny romb
(jeśli dziedziczenie
po klasie B
jest wirtualne)*



Tutaj klasa B jest pośrednią klasą bazową klasy D dwukrotnie, raz przez klasę C1, drugi raz przez klasę C2.

Sytuacja ta wprowadza potrzebę dodatkowej własności języka C++ — dziedziczenia wirtualnego. Pytanie brzmi: Czy programista chce, aby klasa D posiadała jeden podobiekt bazowy typu B czy dwa? Jeśli jeden, to wtedy klasa B powinna być wirtualną klasą bazową i rysunek 4. staje się rombem zgubnym. Jeśli dwa, to klasa B powinna być normalną (niewirtualną) klasą bazową.

Wreszcie, główną komplikacją wirtualnych klas bazowych jest to, że muszą być inicjalizowane bezpośrednio przez najdalej dziedziczącą klasę. Więcej informacji o tym i innych aspektach dziedziczenia wielobazowego znajdziesz w książkach [Stroustrup00] oraz w Zagadnieniu 43. w książce [Meyers97].



Unikaj wielobazowego dziedziczenia po więcej niż jednej nieprotokolowej klasie (klasa protokolowa to abstrakcyjna klasa bazowa niezawierająca danych i składająca się jedynie z czysto wirtualnych metod).

2. Czy dziedziczenie wielobazowe jest kiedykolwiek konieczne?

Krótką odpowiedź: żadna własność języka nie jest „bezwzględnie konieczna”, albowiem każdy program można napisać w assemblerze. Jednakże tak jak większość ludzi wolałaby nie kodować swojego własnego mechanizmu funkcji wirtualnych w zwykłym

języku C, tak w niektórych przypadkach brak dziedziczenia wielobazowego wymaga stosowania kłopotliwych obejść.

Mamy więc cudowną właściwość języka zwaną dziedziczeniem wielobazowym tudzież wielodziedziczeniem. Pytanie brzmi, lub co najmniej brzmiało, czy jest to aby na pewno zaleta?¹

Są programiści, którzy traktują dziedziczenie wielobazowe jako zmore, unikając go za wszelką cenę. To błąd. Co prawda, jeśli zostanie użyte bezmyślnie, dziedziczenie wielobazowe może powodować niepotrzebne sprzężenia i potęgować złożoność programu, ale taką samą wadę posiada każda źle użyta postać dziedziczenia (zobacz *Exceptional C++*² [Sutter00], zagadnienie 24.) i myślę, że możemy się zgodzić, że jeśli wielodziedziczenie miałyby być „złe”, to nie z tego właśnie powodu. Co prawda każdy program można napisać bez uciekania się do wielodziedziczenia, ale jeśli tak stawiać sprawę, to można również stwierdzić, że każdy program można napisać, nie używając wcale dziedziczenia. Więcej, każdy program da się napisać w assemblerze, tyle że nie zawsze jest to wygodne i efektywne.

Jeśli tak, pokaż jak najwięcej przykładowych sytuacji i uzasadnij, dlaczego język powinien zawierać dziedziczenie wielobazowe. Jeśli nie, uzasadnij, dlaczego dziedziczenie pojedyncze, być może połączone z interfejsami w stylu języka Java, dorównuje (lub przewyższa) dziedziczeniu wielobazowemu i dlaczego język nie powinien zawierać takiego dziedziczenia.

A więc kiedy użycie wielodziedziczenia jest właściwe? Krótko mówiąc, jest ono właściwe tylko wtedy, gdy właściwe jest również każde dziedziczenie z osobna. Dość wyczerpująca lista sytuacji uzasadniających zastosowanie dziedziczenia zawarta jest w zagadnieniu 24. z książki *Exceptional C++*. Sytuacje, w których należy użyć dziedziczenia wielobazowego, należą do jednej z trzech kategorii:

1. *Łączenie modułów lub bibliotek.* Wymieniam ten punkt na pierwszym miejscu z powodu, który przedstawiam poniżej. Wiele klas zostało zaprojektowanych jako klasy bazowe — to znaczy, że ich użycie wymaga dziedziczenia. Nasuwa się więc naturalne pytanie o sposób zdefiniowania klasy, która ma rozszerzać dwie biblioteki, a każda biblioteka wymaga, aby dziedziczyć po jednej z jej klas?

¹ Częściowo temat tego zagadnienia został zainspirowany wydarzeniami spotkania standaryzującego języka SQL, które miało miejsce w czerwcu 1998, kiedy dziedziczenie wielobazowe zostało usunięte z próbnego standardu ANSI SQL99 (zainteresowani bazami danych będą mogli zobaczyć zrewidowaną postać DW w standardzie SQL4 jeśli, bądź gdy, dotrzemy tak daleko). Stało się tak głównie dlatego, że proponowana specyfikacja dziedziczenia wielobazowego sprawiała trudności techniczne i dlatego, że chciano dopasować język SQL do języków takich jak Java, które nie obsługują prawdziwego dziedziczenia wielobazowego. Niemniej jednak samo siedzenie na tym spotkaniu i słuchanie ludzi dyskutujących o zaletach i wadach dziedziczenia wielobazowego w tak relatywnie późnym terminie było intrygujące. Czegoś takiego nie robiliśmy w świecie języka C++ od czasów, kiedy powstawały podstawy języka i przypomniało mi to przewlekle wojny emailowe toczące się na grupach dyskusyjnych wiele lat temu (i kilka trochę późniejszych) zawierające tematy podobne do „DW to szatański wynalazek!!!”.

² Wydanie polskie *Wyjątkowy język C++. 47 lamigłówek, zadań programistycznych i rozwiązań*, WNT 2002 — *przyp. tłum.*

Gdy stykasz się z taką sytuacją, nie możesz raczej uniknąć dziedziczenia przez zmianę kodu biblioteki. Najprawdopodobniej biblioteka taka została zakupiona od niezależnego producenta, albo jest dziełem niezależnej grupy projektowej w Twojej firmie. W każdym z tych przypadków zmiana kodu jest często niemożliwa również z powodu braku dostępu do kodu! Nie ma wtedy alternatywy dla wielodziedziczenia — nie ma bowiem innego (naturalnego) sposobu wykonania założonego zadania, a użycie dziedziczenia wielobazowego jest jak najbardziej uzasadnione.

W praktyce da się zauważyć, że łączenie bibliotek przez wielodziedziczenie to jedna z podstawowych technik programistycznych, trzymana na podorędziu każdego niemal programisty, a więc bez względu na częstotliwość jej wykorzystywania warto ją poznać i zrozumieć.

- 2. Klasy protokołowe (klasy interfejsowe).** Najlepszym i najbezpieczniejszym sposobem użycia dziedziczenia wielobazowego w języku C++ jest definiowanie klas protokołowych, to jest klas złożonych wyłącznie z metod czysto wirtualnych. Nieobecność składowych danych w klasie bazowej pozwala na unikanie największych komplikacji wynikających z wielodziedziczenia.

Co ciekawe, niektóre języki (modele) obsługują ten rodzaj dziedziczenia wielobazowego bynajmniej nie za pośrednictwem mechanizmu dziedziczenia. Dwoma przykładami są język Java i model COM. Mówiąc precyzyjnie, języka Java posiada dziedziczenie wielobazowe, ale jedynie w sferze interfejsu — dziedziczenie implementacji ogranicza do dziedziczenia pojedynczego. Klasa w języku Java może implementować wiele „interfejsów”, gdzie interfejs jest bardzo podobny do czysto abstrakcyjnej klasy bazowej niezawierającej danych składowych. Model COM sam w sobie nie zawiera pojęcia dziedziczenia (choć jest to częsta technika implementacji obiektów COM napisanych w języku C++), ale i on definiuje pojęcie składania interfejsów, a interfejsy COM przypominają kombinację interfejsów języka Java i szablonów języka C++.

- 3. Łatwość (polimorficznego) użycia.** Pozyteczną koncepcją jest użycie dziedziczenia celem umożliwienia innemu kodowi zastosowania obiektu pochodnego wszędzie tam, gdzie spodziewana jest klasa bazowa. W niektórych przypadkach możliwość użycia tego samego obiektu pochodnego zamiast kilku rodzajów klas bazowych może okazać się bardzo przydatna — to znakomite pole do popisu dla wielodziedziczenia. Dobry przykład znajduje się w punkcie 14.2.2 w [Stroustrup00]; widzimy tam bazujący na dziedziczeniu wielobazowym projekt klas wyjątków, w którym najdalej dziedzicząca klasa wyjątku może mieć polimorficzny związek „jest” z wieloma bezpośrednimi klasami bazowymi.

Zwróć uwagę, że punkt 3. w dużej części pokrywa się z punktami 1. i 2. Często praktyczne jest zastosowanie punktu 3. w tym samym czasie i z tych samych powodów co jednego z pozostałych punktów.

Jeszcze jedno: nie zapominaj, że czasem nie chodzi po prostu o dziedziczenie po dwóch klasach bazowych; czasem dziedziczy się po każdej z nich z innego powodu. Polimorficzna zależność „jest” nie jest jedynym powodem użycia dziedziczenia. Na przykład klasa musi dziedziczyć prywatnie po klasie bazowej A, aby otrzymać dostęp do składowych chronionych klasy A, ale równocześnie dziedziczyć publicznie po klasie bazowej B, aby polimorficznie zaimplementować funkcje wirtualne klasy B.

Zagadnienie 25. Emulowanie dziedziczenia wielobazowego**Stopień trudności: 5**

Jeśli nie mógłbyś użyć dziedziczenia wielobazowego, jak byś je emulował? Ćwiczenie to ma pomóc Ci zrozumieć powody, dla których dziedziczenie w języku C++ działa tak, a nie inaczej. W odpowiedzi nie zapomnij emulować jak najbardziej naturalnej składni wywołania.

Weź pod uwagę następujący przykład:

```
class A
{
public:
    virtual ~A();
    string Name();
private:
    virtual string DoName();
};

class B1 : virtual public A
{
    string DoName();
};

class B2 : virtual public A
{
    string DoName();
};

A::~A() {}
string A::Name() { return DoName(); }
string A::DoName() { return "A"; }
string B1::DoName() { return "B1"; }
string B2::DoName() { return "B2"; }

class D : public B1, public B2
{
    string DoName() { return "D"; }
};
```

Pokaż najlepszy sposób uniknięcia dziedziczenia wielobazowego, pisząc równoważną (lub jak najbardziej podobną) klasę D bez użycia takiego dziedziczenia. Jak można uzyskać podobną użyteczność klasy D, wprowadzając minimum zmian w składni kodu wywołującego?

* * * * *

Punkt startowy: możesz zacząć od rozważenia przypadków w następującej uprzejmej testowej.

```
void f1( A& x ) { cout << "f1:" << x.Name() << endl; }
void f2( B1& x ) { cout << "f2:" << x.Name() << endl; }
void f3( B2& x ) { cout << "f3:" << x.Name() << endl; }

void g1( A x ) { cout << "g1:" << x.Name() << endl; }
void g2( B1 x ) { cout << "g2:" << x.Name() << endl; }
void g3( B2 x ) { cout << "g3:" << x.Name() << endl; }
```

```

int main()
{
    D d;
    B1* pb1 = &d; //konwersja D* -> B*
    B2* pb2 = &d;
    B1& rb1 = d; //konwersja D& -> B&
    B2& rb2 = d;

    f1( d ); //polimorfizm
    f2( d );
    f3( d );

    g1( d ); //automatyczne rzutowanie do klasy bazowej
    g2( d );
    g3( d );

    //dynamic_cast/RTTI
    cout << ( (dynamic_cast<D*>(pb1) != 0) ? "dobrze " : "źle " );
    cout << ( (dynamic_cast<D*>(pb2) != 0) ? "dobrze " : "źle " );

    try
    {
        dynamic_cast<D&>(rb1);
        cout << "dobrze ";
    }
    catch(...)
    {
        cout << "źle ";
    }

    try
    {
        dynamic_cast<D&>(rb2);
        cout << "dobrze ";
    }
    catch(...)
    {
        cout << "źle ";
    }
}

```



Rozwiązanie

```

class D : public B1, public B2
{
    string DoName() { return "D"; }
};

```

Pokaż najlepszy sposób „obejścia” dziedziczenia wielobazowego, pisząc równoważną (lub jak najbardziej podobną) klasę D bez użycia takiego dziedziczenia. Jak uzyskać podobną użyteczność klasy D, wprowadzając minimum zmian w składni kodu wywołującego?

Istnieje kilka strategii, każda ma słabe strony, ale ta tutaj jest całkiem bliska ideału.

```

class D : public B1
{
public:
    class D2 : public B2
    {
    public:
        void Set ( D* d ) { d_ = d; }
    private:
        string DoName();
        D* d_;
    } d2_;

    D()                { d2_.Set( this ); }

    D( const D& other ) : B1( other ), d2_( other.d2_ )
                        { d2_.Set( this ); }

    D& operator=( const D& other )
    {
        B1::operator=( other );
        d2_ = other.d2_;
        return *this;
    }

    operator B2&()     { return d2_; }

    B2& AsB2()        { return d2_; }

private:
    string DoName()   { return "D"; }
};

string D::D2::DoName(){ return d_->DoName(); }

```

Zanim będziesz czytać dalej, rozważ przeznaczenie każdej klasy i funkcji.

Wady

Obejście to całkiem dobrze implementuje wielodziedziczenie, automatyzuje większość zachowań takiego dziedziczenia i umożliwia wszystkie jego zastosowania, pod warunkiem zachowania pewnej dyscypliny w uzupełnianiu części, które nie są w pełni zautomatyzowane. Brak tej automatyzacji objawia się w następujących elementach:

- ◆ Dostarczenie funkcji `operator B2&()` sprawia bezsprzecznie, że referencje są traktowane inaczej niż wskaźniki, co wprowadza niespójność.
- ◆ Wywołujący kod musi jawnie wywołać funkcję `D::AsB2()`, aby użyć obiektów klasy `D` jako obiektów klasy `B2` (w uprząży testującej oznacza to zmianę `B2* pb2 = &d;` na `B2* pb2 = &d.AsB2();`).
- ◆ Operator `dynamic_cast` z typu `D*` do `B2*` nadal nie działa (można to obejść, jeśli jesteś gotów użyć preprocesora do przededefiniowania wywołania `dynamic_cast`, to byłoby jednak rozwiązanie ekstremalne).

Interesujące jest, że — co łatwo zauważyć — rozmieszczenie obiektu D w pamięci jest podobne do tego, które dałoby dziedziczenie wielobazowe, a to dlatego, że próbujemy zasymulować takie dziedziczenie, rezygnując z jego udogodnień składniowych i wygody, którą dałaby obsługa wbudowana w język.

Wielodziedziczenie być może nie jest potrzebne często, ale kiedy już jest, jest potrzebne *koniecznie*. Niniejsze zagadnienie ma na celu pokazanie, że obsługa tej użytecznej techniki wbudowana w język jest znacznie lepsza niż próba samodzielnych implementacji tego mechanizmu, nawet jeśli dałaby ona — przy zachowaniu pewnej dyscypliny kodowania — dość dokładny duplikat wielodziedziczenia.

Zagadnienie 26. Dziedziczenie wielobazowe i problem bliźniąt syjamskich

Stopień trudności: 4

Przesłanianie odziedziczonych metod wirtualnych jest proste, dopóki nie próbujesz przesłonić metody wirtualnej, która ma taką samą sygnaturę w dwóch klasach bazowych. Przypadek taki może wystąpić bynajmniej nie tylko wtedy, kiedy obie klasy bazowe pochodzą od różnych dostawców! Jaki jest najlepszy sposób rozróżnienia takich „bliźniąt syjamskich”?

Weź pod uwagę dwie następujące klasy:

```
class BaseA
{
    virtual int ReadBuf( const char* );
    // ...
};

class BaseB
{
    virtual int ReadBuf( const char* );
    // ...
};
```

Obie klasy BaseA i BaseB są oczywiście przeznaczone do użycia jako klasy bazowe, poza tym są od siebie zupełnie niezależne; klasy pochodzą od różnych producentów bibliotek, a ich funkcje ReadBuf() są przeznaczone do wykonywania zupełnie odmiennych czynności.

Pokaż, jak zdefiniować klasę Derived dziedziczącą publicznie po obu klasach (BaseA i BaseB), która niezależnie przesłaniałaby obie metody ReadBuf(), tak by wykonywały różne czynności.



Rozwiązanie

Celem tego zagadnienia jest ukazanie drugorzędnej pułapki czyhającej na użytkowników dziedziczenia wielobazowego i przedstawienie sposobu jej uniknięcia. Załóżmy, że w danym projekcie mamy do dyspozycji dwie biblioteki niezależnych od siebie producentów. Producent A zdefiniował klasę bazową BaseA następująco:

```
class BaseA
{
public:
    virtual int ReadBuf( const char* );
    // ...
};
```

Chodzi o to, że zamierzamy odziedziczyć po klasie BaseA, przesłaniając niektóre funkcje wirtualne, ponieważ inne części biblioteki producenta A umożliwiają polimorficzne zastosowania obiektów typów pochodnych wobec typu BaseA. Jest to praktyka częsta i normalna, szczególnie w rozszerzalnych szkieletach aplikacji — nie ma w tym nic złego.

Nic do momentu, gdy zaczniesz używać bibliotek producenta B i skonsternowany odkryjesz następującą klasę:

```
class BaseB
{
public:
    virtual int ReadBuf( const char* );
    // ...
};
```

„To raczej przypadek” — pomyślisz. Nie dość, że producent B ma także klasę bazową, po której trzeba dziedziczyć, ale zdarzyło się, że ta klasa zawiera metodę wirtualną z dokładnie taką samą sygnaturą jak jedna z metod wirtualnych w klasie BaseA. Sęk w tym, że klasa BaseB przeznaczona jest do zupełnie czegoś innego niż klasa A.

Obie klasy BaseA i BaseB są oczywiście przeznaczone do użycia jako klasy bazowe, poza tym są od siebie niezależne. Klasy pochodzą od różnych producentów bibliotek, a ich funkcje ReadBuf() są przeznaczone do wykonywania zupełnie różnych czynności.

Pokaż, jak zdefiniować klasę Derived dziedziczącą publicznie po obu klasach (BaseA i BaseB), która niezależnie przesłaniałaby obie metody ReadBuf(), tak by wykonywały różne czynności.

Problem staje się oczywisty, gdy przychodzi do zdefiniowania klasy dziedziczącej jednocześnie po klasach BaseA i BaseB, kiedy potrzebny jest obiekt, który może zostać użyty polimorficznie przez funkcje z bibliotek obu producentów. Oto naiwna próba przesłonięcia takiej klasy:

```
// Przykład 26.1. Próba 1., nie działa
//
class Derived : public BaseA, public BaseB
{
    // ...

    int ReadBuf( const char* );
    // przesłania obie funkcje: BaseA::ReadBuf()
    // i BaseB::ReadBuf()
};
```

Tutaj metoda `Derived::ReadBuf()` przesłania obie metody `BaseA::ReadBuf()` i `BaseB::ReadBuf()`. Aby zobaczyć, dlaczego w naszych warunkach nie jest to odpowiednie, weź pod uwagę następujący kod:

```
// Przykład 26.1(a). Kontrprzykład,
// dlaczego próba 1. nie działa
//
Derived d;
BaseA* pba = d;
BaseB* pbb = d;

pba->ReadBuf( "przykładowy bufor" );
// wywołuje funkcję Derived::ReadBuf

pbb->ReadBuf( "przykładowy bufor" );
// wywołuje funkcję Derived::ReadBuf
```

Czy widzisz tu problem? Metoda `ReadBuf()` jest wirtualna w obu interfejsach i działa polimorficznie, zgodnie z oczekiwaniami. Ale jest to *ta sama* metoda — bez względu na to, który interfejs został użyty, wywoływana jest metoda `Derived::ReadBuf()`. Tymczasem metody `BaseA::ReadBuf()` i `BaseB::ReadBuf()` mają inne znaczenie i przeznaczone są do wykonywania różnych, a nie tych samych czynności. Dodatkowo nie ma sposobu zasygnalizowania metodzie `Derived::ReadBuf()`, czy jest wywoływana za pośrednictwem interfejsu klasy `BaseA` bądź interfejsu klasy `BaseB`, nie da się więc rozwiązać problemu wykonywania różnych funkcji instrukcją `if` w ciele metody `Derived::ReadBuf()`. Zdaje się, że utknęliśmy.

Można pomyśleć: „Bez przesady, to przykład wymyślony złośliwie”. Otóż nie. Na przykład John Kdlin z firmy Microsoft donosi, że tworzenie klas dziedziczących po dwóch interfejsach `COM IOleObject` i `IConnectionPoint` (traktuj je jako abstrakcyjne klasy bazowe złożone wyłącznie z publicznych funkcji wirtualnych) staje się problematyczne, ponieważ (a) oba interfejsy mają metodę zadeklarowaną jako `virtual HRESULT Unadvise(unsigned long)` i (b) zazwyczaj zachodzi potrzeba przesłonięcia tych metod, tak aby wykonywały różne czynności.

Zastanówmy się przez chwilę. Jak można by rozwiązać ten problem? Czy istnieje sposób na przesłonięcie dwóch odziedziczonych metod `ReadBuf` oddzielnie, tak aby dało się w każdej z nich wykonywać różne czynności w zależności od tego, czy zewnętrzny kod wywołuje metodę przez interfejs klasy `BaseA` czy klasy `BaseB`? Słowem, jak rozdzielić te bliźnięta?

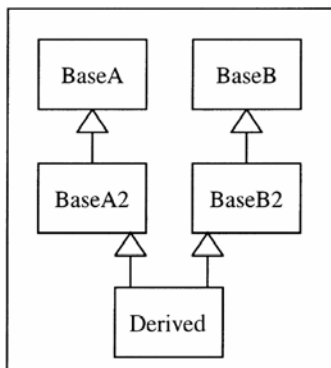
Jak rozdzielić bliźnięta syjamskie?

Na szczęście istnieje całkiem solidne rozwiązanie. Kluczowym problemem jest to, że dwie dające się przesłonić metody mają dokładnie tę samą nazwę i sygnaturę. Zatem rozwiązanie musi polegać na zmianie sygnatury przynajmniej jednej z tych metod, a najprostszą do zmiany częścią sygnatury jest nazwa.

Jak zmienić nazwę funkcji? Przez dziedziczenie oczywiście! Potrzebna jest po prostu pośrednia klasa, która dziedziczy po klasie bazowej, deklaruje nową funkcję wirtualną i przesłania odziedziczoną wersję, aby wywoływała nową funkcję. Rzeczona hierarchia dziedziczenia wygląda tak jak na rysunku 5.

Rysunek 5.

*Użycie klas
pośrednich
do zmiany nazwy
odziedziczonych
funkcji wirtualnych*



Kod zaś wygląda jak poniżej:

```

// Przykład 26.2. Próba 2., prawidłowa
//
class BaseA2 : public BaseA
{
public:
    virtual int BaseAReadBuf( const char* p ) = 0;
private:
    int ReadBuf( const char* p )    // przesłoni wersję odziedziczoną,
    {
        return BaseAReadBuf( p );  // aby wywoływała nową metodę
    }
};

class BaseB2 : public BaseB
{
public:
    virtual int BaseBReadBuf( const char* p ) = 0;
private:
    int ReadBuf( const char* p )    // przesłoni wersję odziedziczoną,
    {
        return BaseBReadBuf( p );  // aby wywoływała nową metodę
    }
};

class Derived : public BaseA2, public BaseB2
{
    /* ... */

public: // lub "private:", w zależności od tego, czy metody te
        // mają być dostępne na zewnątrz

    int BaseAReadBuf( const char* );
        // pośrednio przesłania metodę BaseA::ReadBuf
        // przez metodę BaseA2::BaseAReadBuf

    int BaseBReadBuf( const char* );
        // pośrednio przesłania metodę BaseB::ReadBuf
        // przez metodę BaseB2::BaseBReadBuf
};

```

Może wystąpić konieczność powielenia konstruktorów klasy BaseA i BaseB w klasach BaseA2 i BaseB2, tak aby można było je wywołać w klasie Derived. Ale to wszystko (często prostszym sposobem niż powielanie konstruktorów w kodzie jest dziedziczenie wirtualne w klasach BaseA2 i BaseB2, żeby klasa Derived miała bezpośredni dostęp do konstruktorów bazowych). BaseA2 i BaseB2 są klasami abstrakcyjnymi, więc nie muszą powielać innych funkcji i operatorów klas BaseA i BaseB, takich jak operatorów przypisania.

Teraz wszystko działa tak jak powinno.

```
// Przykład 26.2(a). Dlaczego próba 2. działa
//
Derived d;
BaseA* pba = d;
BaseB* pbb = d;

pba->ReadBuf( "bufor przykładowy" );
// wywołuje metodę Derived::BaseAReadBuf

pbb->ReadBuf( "bufor przykładowy" );
// wywołuje metodę Derived::BaseBReadBuf
```

Należy tylko pamiętać, aby w kolejnych klasach pochodnych nie przesłaniać metody ReadBuf. Jej przesłonięcie wyłączy pośrednictwo metod zmieniających nazwy zainstalowanych w klasie pośredniej.

Zagadnienie 27. Metody (nie)czysto wirtualne

Stopień trudności: 7

Czy ma kiedykolwiek sens zadeklarowanie metody czysto wirtualnej i zdefiniowanie jej ciała?

1. Co to jest metoda czysto wirtualna? Podaj przykład.
2. Po cóż mielibyśmy zadeklarować metodę czysto wirtualną i zdefiniować również jej ciało? Podaj jak najwięcej możliwych powodów i sytuacji.



Rozwiązanie

1. Co to jest metoda czysto wirtualna? Podaj przykład.

Metoda czysto wirtualna to taka metoda wirtualna, którą trzeba przesłonić w skonkretyzowanych klasach pochodnych. Jeśli klasa posiada nieprzesłonięte metody czysto wirtualne, jest nadal klasą „abstrakcyjną”, co uniemożliwia tworzenie obiektów tej klasy.

```
// Przykład 27.1
//
class AbstractClass
{
    // deklaracja metody czysto wirtualnej:
    // klasa jest teraz abstrakcyjna
    virtual void f(int) = 0;
};
```

```

class StillAbstract : public AbstractClass
{
    // nie przesłania metody f(int),
    // zatem ta klasa jest nadal abstrakcyjna
};

class Concrete : public StillAbstract
{
public:
    // wreszcie przesłania metodę f(int),
    // zatem ta klasa jest konkretna
    void f(int) { /*...*/ }
};

AbstractClass a;    // błąd, klasa abstrakcyjna
StillAbstract b;   // błąd, klasa abstrakcyjna
Concrete c;        // ok, klasa konkretna

```

2. Dlaczego mielibyśmy zadeklarować metodę czysto wirtualną i zdefiniować również jej ciało? Podaj jak najwięcej możliwych powodów i sytuacji.

Rozważmy trzy główne powody takiego postępowania. Przypadek 1. jest powszechny, 2. i 3. są użyteczne, ale występują raczej rzadko, a 4. jest używany okazjonalnie przez zaawansowanych programistów, którzy pracują ze słabszymi kompilatorami (typowe: trzy powody i cztery pozycje na liście powodów — cóż, wiadomo powszechnie, że żadna trylogia nie jest kompletna bez czwartej części³).

1. Destruktory czysto wirtualne

Destruktory wszystkich klas bazowych powinny być albo wirtualne i publiczne, albo niewirtualne i chronione. Oto powód tego wymagania: po pierwsze, trzeba zawsze unikać dziedziczenia po klasach konkretnych. Zakładając zatem, że mamy daną klasę niekonkretną, która niekoniecznie potrzebuje publicznego destruktora, gdyż sama nigdy nie będzie konkretyzowana, pozostają dwie możliwe sytuacje. Albo (a) zachodzi potrzeba dopuszczenia polimorficznego usuwania obiektu przez wskaźnik bazowy (w tym przypadku destruktorem musi być wirtualny i publiczny), albo (b) nie zachodzi taka potrzeba (destruktorem powinien być niewirtualny i chroniony — to drugie po to, aby zapobiec niechcianemu użyciu; po szczegóły odsyłam do [Sutter01]).

Jeśli klasa powinna być abstrakcyjna (chcesz zapobiec jej konkretyzowaniu), ale nie ma żadnych metod czysto wirtualnych i ma publiczny destruktorem, to popularną techniką jest zadeklarowanie destruktora jako czysto wirtualnego (i tak powinien być czysto wirtualny):

```

// Przykład 27.2(a)
//
// plik b.h
//
class B

```

³ „Naszą główną bronią jest zaskoczenie. Strach i zaskoczenie są naszymi dwiema głównymi broniąmi” (Latający Cyrk Monty Pythona).

```
{
public: /*...inne funkcje...*/
    virtual ~B() = 0; //destruktor czysto wirtualny
};
```

Oczywiście każdy destruktor klasy pochodnej musi niejawnie wywołać destruktor klasy bazowej, więc destruktor musi mimo wszystko zostać zdefiniowany (nawet jeśli miałby być pusty).

```
// Przykład 27.2(a), kontynuacja
//
// plik b.cpp
//
B::~B() { /* może być pusty */ }
```

Jeśli definicja nie zostanie dostarczona, będzie można nadal wyprowadzać klasy pochodne z klasy B, ale nigdy nie będą one mogły zostać skonkretyzowane, co znakomicie zredukuję ich przydatność.



Destruktry klasy bazowych powinny zawsze być albo wirtualne i publiczne albo niewirtualne i chronione.

2. Wymuszanie świadomej akceptacji zachowania domyślnego

Jeśli klasa pochodna nie przesłoni zwykłych metod wirtualnych, odziedziczy po prostu domyślnie zachowanie klasy bazowej. Chcąc udostępnić zachowanie domyślne, ale bez możliwości takiego „cichego” dziedziczenia, można zadeklarować metody jako czysto wirtualne i jednocześnie dostarczyć ich implementację. Autor klasy pochodnej musi — jeśli chce odwołać się do zachowania domyślnego — wywołać te metody celowo.

```
// Przykład 27.2(b)
//
class B
{
protected:
    virtual bool f() = 0;
};

bool B::f()
{
    return true; // jest to dobre zachowanie domyślne,
                // ale nie powinno być ślepo używane
}

class D : public B
{
    bool f()
    {
        return B::f(); // jeśli klasa D chce domyślnego
    }
    // zachowania, musi to "powiedzieć"
};
```

Dobrym przykładem użycia tej techniki jest wzorzec stanu w książce „bandy czworga” *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma95].

3. Dostarczenie zachowania częściowego

Czasem przydatne jest zaimplementowanie w klasie bazowej częściowego zachowania klas pochodnych, które muszą owo zachowanie na swój sposób uzupełnić. Chodzi o to, by klasa pochodna w ramach własnej implementacji metody wywołała wersję z klasy bazowej:

```
// Przykład 27.2(c)
//
class B
{
    // ...
protected
    virtual bool f() = 0;
};

bool B::f()
{
    // wykonaj czynności ogólne
}

class D : public B
{
    bool f()
    {
        // najpierw użyj implementacji z klasy bazowej
        B::f();

        // ... teraz wykonaj kolejne czynności ...
    }
};
```

I tym razem przykład znajdziemy w książce [Gamma95] w postaci wzorca dekoratora.

4. Obejście słabej diagnostyki błędów w kompilatorze

Istnieją sytuacje, w których przypadkowo zdarza się wywołać funkcję czysto wirtualną (pośrednio z konstruktora lub destruktoru klasy bazowej; przykładów szukaj w dowolnej książce o zaawansowanym programowaniu w języku C++). Oczywiście problemy tego typu nie występują w dobrze napisanym kodzie, ale — jako że nikt nie jest doskonały — czasem się jednak zdarzają.

Niestety, nie wszystkie kompilatory⁴ potrafią taki problem sygnalizować. Często efektem słabości diagnostycznej kompilatora są fałszywe, niezwiązane ze źródłem problemu komunikaty o błędach. Kiedy wreszcie (w co najmniej kilka godzin później) dojdzie się od sedna problemu, zaciskając zęby przeklina się kompilator, zadając sobie pytanie: „dlaczego kompilator nie poinformował mnie o tym?” (odpowiedź: milczenie kompilatora jest jednym z przejawów „zachowania niezdefiniowanego”).

⁴ Z technicznego punktu widzenia tego typu błędy wyłapuje środowisko uruchomieniowe. Mówię tu o kompilatorze, ponieważ generalnie to kompilator powinien dodać kod, który w czasie wykonywania wykrywa błąd wywołania metody czysto wirtualnej.

Jednym ze sposobów zabezpieczenia się przed marnowaniem czasu na szukanie błędów jest dostarczenie definicji metodom czysto wirtualnym (które z definicji nie powinny zostać nigdy wywołane) i wstawienie do ich ciał takiego kodu, który uniemożliwi przeoczenie przypadkowego wywołania metody.

Na przykład:

```
// Przykład 27.2(d)
//
class B
{
public:
    bool f();           // wywołuje metodę do_f()
                       // (wzorzec niewirtualnego interfejsu)

private:
    virtual bool do_f() = 0;
};

bool B::do_f() // ta metoda NIGDY nie powinna zostać wywołana
{
    if( PromptUser( "wywołanie metody czysto wirtualnej B::f -- "
                    "przerwać czy zignorować?" ) == Abort )
        DieDieDie();
}
```

W funkcji DieDieDie() zrób wszystko co konieczne, aby w danym systemie program przeszedł do debugera albo zrzucił stos wywołań czy też w jakikolwiek inny sposób wypisał informacje diagnostyczne. Oto kilka powszechnych metod, które w większości systemów wywołają debuger:

```
void DieDieDie() // sposób w stylu języka C na pisanie do pamięci
{
    // przez wskaźnik zerowy... sprawdzony sposób na wywołanie zamieszania
    memset( 0, 1, 1 );
}

void DieDieDie() // kolejna metoda w stylu języka C
{
    abort();
}

void DieDieDie() // sposób w stylu języka C++ na pisanie do pamięci
// przez wskaźnik zerowy
*static_cast<char*>(0) = 0;
}

void DieDieDie() // sposób w stylu języka C++ na wywołanie przez
// wskaźnik zerowy funkcji, która nie istnieje
static_cast<void(*)>(0)();
}

void DieDieDie() // wróćmy do ostatniej instrukcji "catch(...)"
{
    class LocalClass {};
    throw LocalClass();
}

void DieDieDie() // alternatywna metoda dla zagorzałych zwolenników standardu
{
```

```
    throw std::logic_error();
}
// dla zagorzałych zwolenników standardu używających dobrych kompilatorów
void DieDieDie() throw()
{
    throw 0;
}
```

Chyba już wiadomo, o co chodzi. Wykaż się inwencją twórczą. Możliwości rozmyślnego zawieszenia programu są nieograniczone, ale chodzi o to, żeby zrobić to w taki sposób, aby debugger umiejscowił się jak najbliżej miejsca błędu. Najpewniejszą metodą jest chyba pisanie do pamięci przez wskaźnik zerowy.

Zagadnienie 28. Polimorfizm kontrolowany**Stopień trudności: 3**

Polimorfizm „jest” jest bardzo użytecznym narzędziem w modelowaniu obiektowym. Czasem jednak chcemy ograniczyć możliwości polimorficznego stosowania pewnych klas. Niniejsze zagadnienie przedstawia stosowne przykłady i pokazuje, jak uzyskać zamierzony efekt.

Weź pod uwagę następujący kod:

```
class Base
{
public:
    virtual void VirtFunc();
    // ...
};

class Derived : public Base
{
public:
    void VirtFunc();
    // ...
};

void SomeFunc( const Base& );
```

Mamy też dwie inne funkcje. Zadanie polega na umożliwieniu funkcji f1() polimorficznego użycia obiektów klasy Derived w miejsce obiektów klasy Base i jednocześnie zablokowaniu tej możliwości wszystkim innym funkcjom (w tym funkcji f2()).

```
void f1()
{
    Derived d;
    SomeFunc( d ); // działa, OK
}

void f2()
{
    Derived d;
    SomeFunc( d ); // chcesz temu zapobiec
}
```

Pokaż, jak uzyskać założony efekt.



Rozwiązanie

Weź pod uwagę następujący kod:

```
class Base
{
public:
    virtual void VirtFunc();
    // ...
};

class Derived : public Base
{
public:
    void VirtFunc();
    // ...
};

void SomeFunc( const Base& );
```

Powodem, dla którego każdy kod może polimorficznie zamiast obiektów klasy Base używać obiektów klasy Derived, jest to, że klasa Derived dziedziczy publicznie po klasie Base — to było nam wiadome i nie jest żadną niespodzianką.

Gdyby klasa Derived dziedziczyła prywatnie po klasie Base, wtedy prawie żaden kod nie mógłby używać obiektów klasy Derived polimorficznie w roli obiektów klasy Base. Użyłem słowa „prawie”, ponieważ kod, który ma dostęp do prywatnych składowych klasy Derived może mimo wszystko odwoływać się do prywatnych klas bazowych klasy Derived, a zatem może używać obiektów klasy Derived polimorficznie zamiast obiektów klasy Base. Normalnie taki dostęp mają tylko składowe klasy Derived, jednak dostęp możemy rozszerzyć na wybrany kod zewnętrzny, stosując „za-przyjaźnienie”.

Podsumowując:

Mamy dwie inne funkcje. Zadanie polega na umożliwieniu funkcji f1() polimorficznego użycia obiektów klasy Derived w miejsce obiektów klasy Base i jednocześnie zablokowaniu tej możliwości wszystkim innym funkcjom (w tym funkcji f2()).

```
void f1()
{
    Derived d;
    SomeFunc( d ); // działa, OK
}

void f2()
{
    Derived d;
    SomeFunc( d ); // chcesz temu zapobiec
}
```

Pokaż, jak uzyskać założony efekt.

Odpowiedzią jest kod:

```
class Derived : private Base
{
public:
    void VirtFunc();
    //...
    friend void f1();
};
```

Jest to przejrzyste rozwiązanie, chociaż daje funkcji `f1()` większy dostęp do klasy `Derived`, niż go miała w oryginalnej wersji.