

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

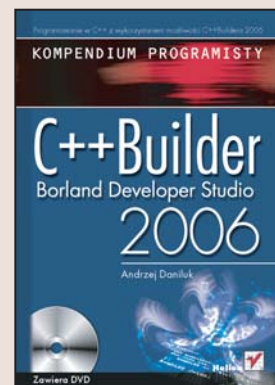
FRAGMENTY KSIĄŻEK ONLINE

C++Builder Borland Developer Studio 2006. Kompendium programisty

Autor: Andrzej Daniluk

ISBN: 83-246-0494-4

Format: B5, stron: 744



Jeden z najnowszych produktów firmy Borland, C++Builder Borland Developer Studio 2006, to połączenie nowoczesnego języka programowania, jakim jest C++, biblioteki komponentów wizualnych, zintegrowanego środowiska programistycznego oraz narzędzi służących do modelowania oprogramowania. Pomimo że zaimplementowana w C++Builder wersja języka C++ nie jest dokładnym odzwierciedleniem standardu ANSI, środowisko to zyskało duże uznanie wśród najlepszych programistów, doceniających jego uniwersalność i stabilność.

Książka „C++Builder Borland Developer Studio 2006. Kompendium programisty” przedstawia zasady programowania w języku C++ z wykorzystaniem narzędzia C++Builder 2006. Opisuje zarówno samo środowisko, jak i poszczególne elementy języka. Dzięki niej nauczysz się korzystać z języka UML używanego do projektowania aplikacji oraz dowiesz się, jak realizować projekty, wykorzystując język C++. Poznasz także nowoczesne metodologie tworzenia oprogramowania za pomocą narzędzi typu RAD.

- środowisko C++Builder Borland Developer Studio 2006
- Podstawy języka UML
- Korzystanie z biblioteki STL
- Obsługa wyjątków
- Operacje na systemie plików
- Programowanie wielowątkowe
- Komponenty
- Programowanie grafiki

Poznaj potęgę języka C++ i zdobądź szczególne umiejętności programowania



Spis treści

	Wstęp	11
Rozdział 1.	Środowisko programisty IDE C++Builder Borland Developer Studio 2006	15
	Struktura głównego menu	18
	Pasek narzędzi — Speed Bar	47
	Inspektor obiektów — Object Inspector	48
	Widok struktury obiektów	49
	Ogólna postać programu pisanego w C++	50
	Funkcja main()	52
	Dyrektywa #include i prekompilacja	53
	Konsolidacja	54
	Konfiguracja opcji projektu	54
	Uruchamiamy program	57
	Ogólna postać programu pisanego w C++Builderze BDS 2006	58
	Formularz	59
	Zdarzenia	61
	Konsola czy formularz?	64
	Podsumowanie	64
Rozdział 2.	Język modelowania, model dojrzałości i proces projektowania	65
	UML jako język modelowania	65
	Zawartość UML	67
	Diagramy klas	67
	Diagramy obiektów	72
	Diagramy komponentów	73
	Diagramy elementów zawierających strukturę wewnętrzną	74
	Diagramy pakietów	75
	Diagramy wdrożenia	76
	Diagramy czynności	76
	Diagramy przypadków użycia	79
	Diagramy stanów	80
	Diagramy przebiegu	82

Diagramy współpracy	84
Diagramy komunikacji	84
Diagramy harmonogramowania zadań	85
Mechanizmy rozszerzania	85
Strategia modelowania i dokumentowania	87
Model dojrzałości	88
Proces projektowania	88
Nie tylko IID	90
Podsumowanie	90
Rozdział 3. Podstawy języka C++	91
Dyrektywy preprocesora	91
Dyrektywa #include	91
Dyrektywa #define	92
Dyrektywa #undef	92
Dyrektywa #pragma hdrstop	92
Dyrektywa #pragma argsused	92
Dyrektywa #pragma inline	93
Dyrektywa #pragma option	93
Dyrektywa #pragma message	93
Dyrektywa #pragma package(smart_init)	93
Dyrektywa #pragma resource "nazwa_pliku"	93
Dyrektywa #error	94
Dyrektywy kompilacji warunkowej	94
Kategorie typów danych	96
Podstawowe proste typy całkowite i rzeczywiste	96
Typ int	97
Typ double	97
Modyfikatory typów	98
Typy danych Windows	100
Typ Currency	101
Typ void	101
Typy logiczne	101
Typy znakowe	102
Typy łańcuchowe	102
Modyfikatory dostępu const i volatile	103
Specyfikatory klas pamięci	104
Specyfikator extern	104
Specyfikator static	105
Specyfikator register	105
Operatory	105
Typ wyliczeniowy	109
Słowo kluczowe typedef	109
Typ zbiorowy	110
Deklarowanie tablic	111
Instrukcje sterujące przebiegiem programu	113
Instrukcja warunkowa if...else	113
Instrukcja wyboru switch...case...break	115
Instrukcja for	116
Nieskończona pętla for	118
Instrukcja iteracyjna do...while	118
Instrukcja iteracyjna while	119
Instrukcja przerywania wykonywania pętli break	120

Funkcja przerwania programu exit()	121
Funkcja przerwania programu abort()	122
Instrukcja kontynuacji programu continue	122
Funkcje w C++	122
Rekurencja	124
Przedefiniowywanie funkcji	125
Niejednoznaczność	128
Konwencje wywoływania funkcji	129
Specyfikatory konsolidacji funkcji	129
Wskazania i adresy	129
Operatory wskaźnikowe	133
Wskaźniki i tablice	133
Wielokrotne operacje pośrednie	135
Operatory new i delete	138
Dereferencja wskaźnika	138
Wskaźniki ze słowem kluczowym const	139
Wskaźniki do funkcji	140
Odwołania	145
Parametry odwołań	147
Zwracanie odwołań przez funkcje	149
Struktury	150
Przekazywanie struktur funkcjom	152
Struktury zagnieżdżone	153
Wskaźniki do struktur	155
Samodzielne tworzenie plików nagłówkowych	156
Unie	158
Klasy w C++	159
Przedstawienie w UML	163
Konstruktor i destruktor	164
Konstruktory kopiowania	166
Inne spojrzenie na klasy. Własności	167
Funkcje ogólne	170
Funkcje z dwoma typami ogólnymi	172
Przedefiniowywanie funkcji ogólnych	172
Klasy ogólne	174
Wzorce klas z wieloma ogólnymi typami danych	176
Wzorzec auto_ptr	177
Dziedziczenie	178
Powiązania	183
Funkcje składowe klas ze specyfikatorami const i volatile	186
Funkcje wewnętrzne	188
Realizacja przekazywania egzemplarzy klas funkcjom	190
Wskaźniki do egzemplarzy klas	191
Operatory (.*) oraz (->*)	193
Wskaźnik this	194
Przeładowywanie operatorów	195
Przeładowywanie jednoargumentowych operatorów ++ oraz --	196
Przeładowywanie operatorów (!) oraz (!=)	199
Przeładowywanie dwuargumentowego operatora arytmetycznego +	202
Przeładowywanie operatora &	204
Przeładowywanie operatora indeksowania tablic []	205

Funkcje i klasy zaprzyjaźnione	208
Klasy wejścia-wyjścia języka C++	213
Obsługa plików z wykorzystaniem klasy ios	215
Tablicowe operacje wejścia-wyjścia	217
Modele programistyczne	220
Programowanie sekwencyjne	221
Programowanie strukturalne	221
Programowanie proceduralne	222
Programowanie obiektowe	224
Programowanie zorientowane obiektowo	226
Programowanie generyczne	231
Programowanie aspektowe	231
Narzędzia metaprogramowania w C++	232
Programowanie oparte na skompilowanych modułach	234
Programowanie wielowątkowe	235
Programowanie komponentowe	235
Podsumowanie	235
Rozdział 4. Wczesne oraz późne wiązanie	237
Odwołania i wskaźniki do klas pochodnych	237
Funkcje wirtualne w C++	240
Wirtualne klasy bazowe	243
Funkcje wirtualne w C++Builderze	247
Klasy abstrakcyjne w stylu biblioteki VCL	251
Interfejsy	254
Zliczanie odwołań do interfejsu	255
Identyfikator interfejsu	256
Specyfikator <code>__closure</code>	262
Obszary nazw	265
Operator <code>__classid</code>	265
Funkcja <code>Register()</code>	266
Podsumowanie	266
Rozdział 5. Tablice	267
Tablice dynamicznie alokowane w pamięci	267
Tablice dynamiczne	270
Tablice otwarte	274
Tablice struktur	276
Tablice wskaźników do struktur	279
Odwołania do elementów tablicy wskaźników do struktur	281
Podsumowanie	284
Rozdział 6. Biblioteka STL	285
Iteratory	285
Kontenery	286
vector	286
deque	289
list	289
slist	290
set	290
map	291
multiset	292
multimap	293

hash_set	293
hash_map	294
hash_multiset	295
hash_multimap	296
Obiekty funkcyjne	297
Adaptory	300
Adaptory iteratorów	300
Adaptory kontenerów	300
Adaptory obiektów funkcyjnych	302
Algorytmy	303
Alokatory	305
Valarray	306
Podsumowanie	308
Rozdział 7. Zaawansowane operatory rzutowania typów	309
Operator static_cast	309
Operator dynamic_cast	310
Operator const_cast	313
Operator reinterpret_cast	316
Podsumowanie	317
Rozdział 8. Informacja czasu wykonania	319
Klasa TObject	319
Hierarchia własności komponentów VCL	323
Czas życia komponentów	324
Moduł typeinfo.h	326
Identyfikacja typów czasu wykonywania	328
Tablica metod wirtualnych	330
Klasa TControl	331
Modyfikator __rtti	333
Podstawowe elementy GUI	334
Przydatne techniki modelowania GUI	335
Agregacje, kompozycje i powiązania klas	339
Podsumowanie	341
Rozdział 9. Obsługa wyjątków	343
Standardowa obsługa wyjątków	343
Strukturalna obsługa wyjątków	348
Klasy wyjątków	350
Zmienne globalne __throwExceptionName, __throwFileName, __throwLineNumber	354
Zapisywanie nieobsłużonych wyjątków	356
Transformowanie wyjątków Windows	359
Podsumowanie	361
Rozdział 10. Obsługa plików	363
Klasy TDirectoryListBox, TFileListBox, TDriveComboBox	363
Klasy TActionList, TOpenDialog i TSaveDialog	365
Własność Options klas TOpenDialog i TSaveDialog	372
Klasy TOpenTextFileDialog i TSaveTextFileDialog	374
Klasy TOpenPictureDialog i TSavePictureDialog	374
Klasy TActionManager i TActionMainMenuBar	377

Moduł sysutils	381
Operacje na plikach	382
Atrybuty pliku	392
Określanie rozmiaru katalogu	397
Usuwanie katalogu	397
Operacje na dyskach	398
Operacje na nazwach plików	399
Wzajemne przeszukiwanie katalogów i plików	400
Windows API	403
Klasa TMemoryStream	408
Klasa TFileStream	411
Tworzenie logów wyjątków podczas kopiowania plików	414
Przesyłanie plików przez sieć	415
Funkcja WinExec()	419
Funkcja ShellExecuteEx()	420
Pliki wymiany danych	420
Kompilowanie plików zasobów	422
Klasa TIniFile	424
Drukowanie	428
Klasa TPrintDialog	431
Dokumenty XML	432
Pliki i katalogi w Linuksie	434
Podsumowanie	436
Rozdział 11. Łańcuchy ANSI	437
Makrodefinicja VCL_Iostream	445
Znaki wielobajtowe	448
Podsumowanie	449
Rozdział 12. Zmienne o typie modyfikowalnym w czasie wykonywania programu	451
Struktura TVarData	451
Klasa TCustomVariantType	455
Moduł variants	457
Tablice wariantowe	459
Wariantowe tablice otwarte	465
Klient OLE	468
Moduł VarCmplx	471
Liczby zespolone, płaszczyzna liczbowa, moduł i argument liczby	471
Podsumowanie	478
Rozdział 13. Funkcje FPU i systemowe	479
Funkcje FPU	479
Struktura SYSTEM_INFO	485
Klasa THeapStatus	488
Identyfikator GUID	491
Klasa TRegistry	493
Klasa TRegistryIniFile	496
Podsumowanie	497
Rozdział 14. Elementy wielowątkowości	499
Wątki i procesy	499
Funkcja _beginthread()	501
Funkcja _beginthreadNT()	504

Funkcja <code>_beginthreadex()</code>	509
Funkcja <code>BeginThread()</code>	509
Zmienne lokalne wątku	514
Klasa <code>TThread</code>	516
Synchronizacja wątków	522
Sekcje krytyczne	527
Wzajemne wykluczenia	529
Uruchamianie zewnętrznych plików wykonywalnych	531
Wielowątkowe operacje na dyskach, katalogach i plikach	532
Zmienna <code>IsMultiThread</code>	535
Podsumowanie	536
Rozdział 15. Liczby pseudolosowe	537
Funkcje <code>randomize()</code> i <code>random()</code>	538
Losowanie z powtórzeniami	543
Losowanie bez powtórzeń	546
Generatory częściowo uporządkowane	552
Szyfrowanie	560
Podsumowanie	561
Rozdział 16. Konwersje wielkości liczbowych	563
Podsumowanie	587
Rozdział 17. Wprowadzenie do grafiki	589
Barwne modele	590
Płótno	593
Mapy bitowe	598
JPEG	603
JPEG 2000	606
Obraz video	607
Drukowanie grafiki	611
Komponent <code>TChart</code>	612
Podsumowanie	615
Rozdział 18. Komponentowy model C++Buildera	617
Tworzymy nowy komponent	617
Modyfikacja istniejącego komponentu z biblioteki <code>VCL</code>	626
Komponenty graficzne	631
Harmonogramowanie zadań	637
Usuwanie komponentów z pakietu	644
Komponenty generyczne	645
Podsumowanie	648
Rozdział 19. Biblioteki DLL	649
Łączenie statyczne. Część I	650
Łączenie statyczne. Część II	652
Ładowanie i zwalnianie bibliotek w czasie działania programu	655
Funkcja <code>DllEntryPoint()</code>	658
Klasy jako elementy bibliotek DLL	661
Bazowe adresy ładowania	664
Określanie adresów funkcji	664
Komponenty i biblioteki DLL	667

	Pakiety	670
	Zmienna IsLibrary	675
	Podsumowanie	675
Dodatek A	GRAPPLE	677
	Zbieranie wymagań	678
	Modelowanie urządzenia	678
	Protokół	678
	Interakcje	679
	Identyfikacja współpracujących systemów	679
	Analiza	680
	Przypadki użycia	680
	Zmiany stanu systemu	681
	Wstępny model klas programu kontrolującego zewnętrzny system wbudowany ...	682
	Projektowanie	682
	Stacyczny model klas programu kontrolującego zewnętrzny system wbudowany	683
	Diagram artefaktów	684
	Diagramy czynności	684
	Kodowanie	685
	Wdrożenie	690
	Graficzny interfejs użytkownika	691
	Kodowanie	693
	Podsumowanie	693
Dodatek B	Together	695
	Literatura	707
	Skorowidz	711

4.

Wczesne oraz późne wiązanie

W analizie i projektowaniu zorientowanym obiektowo występują dwa bardzo ważne pojęcia: *wczesne* oraz *późne wiązanie*. Z wczesnym wiązaniem (ang. *early binding*) mamy do czynienia w sytuacjach, gdy funkcje w trakcie kompilacji programu wiąże się z określonymi obiektami. Przykładem wczesnego wiązania będą np. sytuacje, w których w głównej funkcji `main()` wywołujemy funkcje standardowe i przeładowane oraz funkcje przeddefiniowywanych standardowych operatorów. Sytuacje, gdy wywoływane funkcje wiązane są z określonymi obiektami w trakcie działania programu, określamy mianem późnego wiązania (ang. *late binding*). Przykładem późnego wiązania są klasy pochodne, funkcje wirtualne, klasy polimorficzne i abstrakcyjne. Wielką zaletą technik związanych z późnym wiązaniem jest możliwość stworzenia prawdziwego interfejsu użytkownika wraz z odpowiednią biblioteką klas, którą można niemal swobodnie uzupełniać i modyfikować.

Odwołania i wskaźniki do klas pochodnych

Na podstawie wiadomości przedstawionych w poprzednich rozdziałach śmiało możemy wywnioskować, iż wskaźnik określonego typu nie może wskazywać na dane odmiennych typów. Niemniej jednak od tej reguły istnieje pewne bardzo ważne odstępstwo. Rozpatrzmy sytuację, w której zaimplementowaliśmy w programie pewną klasę zwaną klasą bazową oraz klasę z niej dziedziczącą — czyli klasę pochodną (potomną). Okazuje się, że wskaźniki do klasy bazowej mogą również w określonych sytuacjach wskazywać na reprezentantów lub elementy klasy pochodnej.

Założmy, iż w programie zaimplementowaliśmy klasę bazową `TStudent` z publiczną funkcją składową przechowującą nazwisko pewnego studenta. Następnie stworzymy klasę pochodną `TEgzamin` z publiczną funkcją składową przechowującą ocenę, jaką otrzymała dana osoba z egzaminu z wybranego przedmiotu. W funkcji `main()` zadeklarujemy zmienną `infoStudent` jako wskaźnik do klasy `TStudent`:

```
TStudent *infoStudent;
```

Zadeklarujmy również po jednym egzemplarzu klas TStudent i TEgzamin:

```
TStudent student;
TEgzamin egzamin;
```

Okazuje się, że zmienna deklarowana jako wskaźnik do typu bazowego TStudent może wskazywać nie tylko na obiekty klasy bazowej, ale również i pochodnej:

```
infoStudent=&student;
infoStudent=&egzamin;
```

Za pomocą tak określonego wskaźnika infoStudent można uzyskać dostęp do wszystkich elementów klasy TEgzamin odziedziczonych po klasie TStudent, tak jak pokazano to na listingu 4.1.

Listing 4.1. Kod głównego modułu Unit_R4_01.cpp projektu Projekt_R4_01.bdsproj wykorzystującego wskaźniki i odwołania do typów pochodnych

```
#include <iostream>
#pragma hdrstop

using namespace std;

class TStudent // klasa bazowa
{
    char nazwisko[40];
public:
    void __fastcall jakiStudent(char *s)
        {strcpy(nazwisko, s);}
    void __fastcall pokazN() {cout << nazwisko << endl;}
};
//-----
class TEgzamin: public TStudent // klasa pochodna
{
    char ocena[5];
public:
    void __fastcall egzaminInformatyka(char *e)
        {strcpy(ocena, e);}
    void __fastcall pokazE() {cout << ocena << endl;}
};
//-----
int main()
{
    // wskaźnik do klasy TStudent (bazowej)
    TStudent *infoStudent;
    // student-egzemplarz klasy TStudent
    TStudent student;
    // wskaźnik do klasy TEgzamin (pochodnej)
    TEgzamin *eInfoStudent;
    // egzamin-egzemplarz klasy TEgzamin
    TEgzamin egzamin;
    // wskaźnik infoStudent wskazuje na egzemplarz
    // klasy TStudent
    infoStudent=&student;
```

```

infoStudent->jakiStudent("Wacek Jankowski");

// wskaźnik infoStudent wskazuje na egzemplarz
// klasy TEgzamin, będącej klasą pochodną względem
// klasy bazowej TStudent
infoStudent=&egzamin;

infoStudent->jakiStudent("Janek Wackowski");

// sprawdzenie poprawności przypisań
student.pokazN();
egzamin.pokazN();
cout << endl;

// funkcje egzaminInformatyka() i pokazE() są
// elementami klasy pochodnej. Dostęp do nich uzyskujemy
// za pomocą wskaźnika eInfoStudent
eInfoStudent = &egzamin;
eInfoStudent->egzaminInformatyka("Egz. Informatyka 2.5");
infoStudent->pokazN();
eInfoStudent->pokazE();
cout << endl;
// uzyskanie dostępu do funkcji składowej klasy pochodnej
// za pomocą wskaźnika do klasy bazowej
((TEgzamin *)infoStudent)->pokazE();

cin.get();
return 0;
}
//-----

```

Śledząc powyższe zapisy, z łatwością przekonamy się, iż wskaźnik do klasy bazowej może równie dobrze wskazywać na te elementy klasy pochodnej, które są zdefiniowane również w klasie bazowej, z tego względu, że klasa TEgzamin dziedziczy publiczne elementy klasy TStudent. Jednak, używając w prosty sposób wskaźnika do klasy bazowej, nie można uzyskać dostępu do tych elementów, które występują jedynie w klasie pochodnej. W przypadku, gdy zażądali-byśmy uzyskania dostępu np. do funkcji składowej pokazE() klasy pochodnej, należałoby wy-korzystać zmienną studentInfo będącą jawnym wskaźnikiem do klasy TEgzamin. Jeżeli mimo wszystko ktoś zdecydowałby się, aby za pomocą wskaźnika do klasy bazowej uzyskać dostęp do jakiegoś elementu klasy pochodnej, będzie musiał wykonać w odpowiedni sposób operację rzutowania typów:

```
((TEgzamin *)infoStudent)->pokazE();
```

Poprzez wykorzystanie zewnętrznej pary nawiasów informujemy kompilator, iż rzutowanie łączone jest ze wskaźnikiem infoStudent, a nie z wartością funkcji pokazE(). Występowanie we-wnętrznej pary nawiasów określa sytuację, w której wskaźnik infoStudent do klasy bazowej rzutowany jest na typ klasy pochodnej TEgzamin.

Funkcje wirtualne w C++

Funkcją wirtualną (ang. *virtual function*) nazywamy taką funkcję, która jest zadeklarowana w klasie bazowej za pomocą słowa kluczowego `virtual`, a następnie w takiej samej postaci definiowana również w klasach pochodnych. Funkcje takie bardzo często określa się mianem *funkcji kategorii virtual*. Ponownie definiując funkcję wirtualną w klasie pochodnej, możemy (ale nie musimy) powtórnie umieszczać słowo `virtual` przed jej nazwą. Funkcje wirtualne mają bardzo ciekawą właściwość. Charakteryzują się mianowicie tym, iż podczas wywoływania dowolnej z nich za pomocą odwołania lub wskaźnika do klasy bazowej wskazującego na egzemplarz klasy pochodnej, *aktualna wersja wywoływanej funkcji każdorazowo ustalana jest w trakcie wykonywania programu z rozróżnieniem typu wskazywanej klasy*. Klasy, w których zdefiniowano jedną lub więcej funkcji wirtualnych, nazywamy *klasami polimorficznymi*.

Jako praktyczny sposób wykorzystania klas polimorficznych rozpatrzmy przykład, gdzie zadeklarowano nieskomplikowaną klasę bazową `TBazowa` z funkcją `pokazB()` kategorii `virtual`, której jedynym zadaniem jest wyświetlenie odpowiedniego tekstu. Ponieważ funkcja jest rzeczywiście funkcją wirtualną, możemy ją z powodzeniem powtórnie zdefiniować (tzn. zdefiniować jej kolejną wersję) w klasie pochodnej `TPochodna`, dziedziczącej publiczne elementy klasy `TBazowa`. Sytuację tę ilustruje listing 4.2.

Listing 4.2. Kod głównego modułu `Unit_R4_02.cpp` projektu `Projekt_R4_02.bdsproj` jako przykład wykorzystania klas polimorficznych

```
#include <iostream>
#pragma hdrstop

using namespace std;

class TBazowa {
public:
    __fastcall TBazowa() {pokazB();}           // konstruktor
    virtual void __fastcall pokazB()
    {cout << "Jestem klasa bazowa" << endl; }
};
//-----
class TPochodna : public TBazowa {
public:
    __fastcall TPochodna() {pokazB();}         // konstruktor
    /*virtual*/ void __fastcall pokazB()
    {cout << "Jestem klasa pochodna" << endl;}
};
//-----
int main()
{
    TPochodna pochodna;                       // wywołanie funkcji pokazB()
    cin.get();
    return 0;
}
//-----
```

Jak łatwo zauważyć, w celu wywołania funkcji `pokazB()` w głównej funkcji `main()` zawarto jedynie deklarację egzemplarza klasy pochodnej. Wynika to z faktu, iż zarówno klasa bazowa, jak i pochodna zawierają odpowiednio zaimplementowane konstruktory.

Na listingu 4.3 zamieszczono przykład ilustrujący ideę posługiwania się wskaźnikami do klas polimorficznych, co w efekcie pozwala na pominięcie jawnych deklaracji konstruktorów odpowiednich klas.

Listing 4.3. Kod głównego modułu *Unit_R4_03.cpp* projektu *Projekt_R4_03.bdsproj* wykorzystującego wskaźniki do klas polimorficznych

```
#include <iostream>
#pragma hdrstop

using namespace std;

class TBazowa {
public:
    virtual void __fastcall pokazB()
    {cout << "Jestem klasa bazowa" << endl; }
};
//-----
class TPochodna : public TBazowa {
public:
    /*virtual*/ void __fastcall pokazB()
    {cout << "Jestem klasa pochodna" << endl;}
};
//-----
int main()
{
    TBazowa bazowa;
    TBazowa *ptrBazowa;
    TPochodna pochodna;
    ptrBazowa = &bazowa;
    ptrBazowa->pokazB();    // wywołanie funkcji pokazB() klasy TBazowa
    ptrBazowa=&pochodna;
    ptrBazowa->pokazB();    // wywołanie funkcji pokazB() klasy TPochodna
    cin.get();
    return 0;
}
//-----
```

W podanym przykładzie w klasie bazowej definiowana jest funkcja wirtualna `pokazB()`, po czym jej kolejna wersja zdefiniowana jest względem klasy pochodnej. W głównej funkcji `main()` zawarto w kolejności deklarację egzemplarza bazowa klasy bazowej, wskaźnika `ptrBazowa` do klasy bazowej i egzemplarza pochodna klasy pochodnej. Dzięki instrukcjom:

```
ptrBazowa = &bazowa;
```

zmienna `ptrBazowa` uzyskuje adres egzemplarza klasy bazowej, co w konsekwencji pozwala na wykorzystanie jej do wywołania funkcji `pokazB()` z klasy bazowej:

```
ptrBazowa->pokazB();
```

W analogiczny sposób można dokonać wywołania funkcji `pokazB()` klasy pochodnej, posługując się adresem egzemplarza klasy pochodnej. Ponieważ funkcja `pokazB()` jest w swoich klasach funkcją wirtualną, zatem w trakcie działania programu decyzja o tym, która wersja tej funkcji jest aktualnie wywoływana, zapada na podstawie określenia typu egzemplarza klasy aktualnie wskazywanego przez wskaźnik `ptrBazowa`.

Podczas pracy z funkcjami wirtualnymi możliwe jest również wykorzystywanie parametru jako odwołania do klasy bazowej. Odpowiednio konstruowane odwołania do klasy bazowej umożliwiają wywołanie funkcji wirtualnej z jednoczesnym przekazaniem jej argumentu. Przedstawiony na listingu 4.4 program jest modyfikacją algorytmu z poprzedniego ćwiczenia. Zadeklarowano w nim klasę bazową, dwie klasy pochodne oraz funkcję przeładowaną, zawierającą — poprzez parametr formalny `x` — odwołanie do klasy bazowej:

```
//-----
void __fastcall pokazB(TBazowa &x)    // odwołanie do klasy bazowej
{
    x.pokazB();
    return;
}
//-----
```

Dzięki tak skonstruowanemu odwołaniu aktualna wersji funkcji `pokazB()`, która powinna być w danym momencie działania programu wywołana, ustalana jest w głównej funkcji `main()` na podstawie typu, do którego odwołuje się jej parametr aktualny.

Listing 4.4. Kod głównego modułu *Unit_R4_04.cpp* projektu *Projekt_R4_04.bdsproj* wykorzystującego odwołanie do klasy polimorficznej

```
#include <iostream>
#pragma hdrstop

using namespace std;

class TBazowa {
public:
    virtual void __fastcall pokazB()
    {cout << "Jestem klasa bazowa" << endl; }
};
//-----
class TPochodna1 : public TBazowa {
public:
    /*virtual*/ void __fastcall pokazB()
    {cout << "Jestem 1 klasa pochodna" << endl;}
};
//-----
class TPochodna2 : public TBazowa {
public:
    /*virtual*/ void __fastcall pokazB()
    {cout << "Jestem 2 klasa pochodna" << endl;}
};
//-----
```

```
void __fastcall pokazB(TBazowa &x)    // odwołanie do klasy bazowej
{
    x.pokazB();
    return;
}
//-----
int main()
{
    TBazowa bazowa;
    TPochodna1 pochodna1;
    TPochodna2 pochodna2;
    pokazB(bazowa);                  // wywołanie funkcji pokazB() klasy TBazowa
    pokazB(pochodna1);               // wywołanie funkcji pokazB() klasy TPochodna1
    pokazB(pochodna2);               // wywołanie funkcji pokazB() klasy TPochodna2
    cin.get();
    return 0;
}
//-----
```



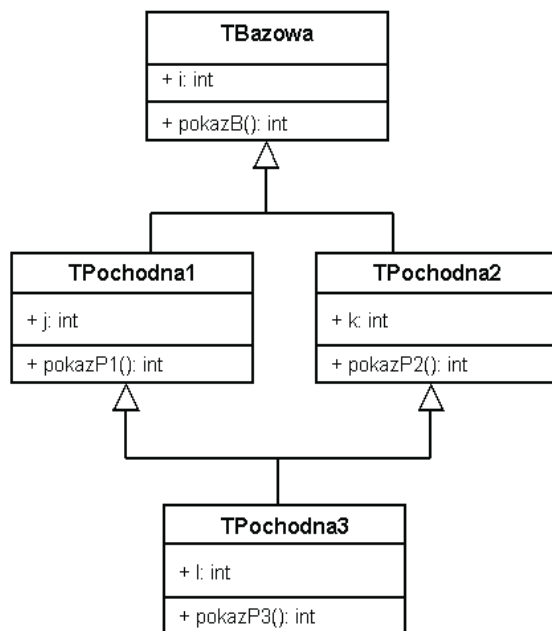
Wskazówka

Bardzo często funkcje zawierające odwołania do klas polimorficznych mają (choć niekoniecznie) takie same nazwy, jak funkcje wirtualne względem danej klasy. Choć funkcje te mogą mieć takie same nazwy, nie należy utożsamiać ich z *funkcjami przeładowanymi*. Pomiędzy konstrukcją funkcji przeładowywanych i ponownym definiowaniem funkcji wirtualnych istnieją poważne różnice, np. prototypy funkcji wirtualnych muszą być identyczne, funkcje przeładowane zaś mają różną liczbę lub typ parametrów. Z tego powodu ponowne definiowanie funkcji wirtualnych nazywa się *przykrywaniem* lub *nadpisywaniem* funkcji.

Wirtualne klasy bazowe

Tematem poprzedniego podrozdziału były funkcje wirtualne, czyli funkcje deklarowane ze słowem kluczowym `virtual`. Z przedstawionych przykładów łatwo wywnioskujemy, iż wielką ich zaletą jest to, że są one odpowiednio przykrywane w klasach pochodnych. Jednak w języku C++ słowo `virtual` posiada jeszcze jedno znaczenie, służy mianowicie do deklarowania tzw. *wirtualnych klas bazowych*.

Rozpatrzmy sytuację, w której potrzebujemy zdefiniować w programie pewną klasę bazową `TBazowa`, dwie klasy pochodne `TPochodna1` i `TPochodna2` dziedziczące po klasie bazowej i dodatkowo trzecią klasę pochodną `TPochodna3`, dziedziczącą elementy publiczne klas `TPochodna1` i `TPochodna2`. W każdej z klas zdefiniujemy po jednej funkcji zwracającej pewną wartość całkowitą. Przyjęte założenia ilustruje rysunek 4.1 oraz listing 4.5.



Rysunek 4.1. Idea poziomego dziedziczenia klas. Klasa *TPochodna3* dziedziczy poziomo (wielokrotnie) po klasach *TPochodna1* i *TPochodna2*

Listing 4.5. Kod głównego modułu *Unit_R4_05.cpp* projektu *Projekt_R4_05.bdsproj* wykorzystującego standardowe klasy bazowe

```

// Program nie zostanie skompilowany !
#include <iostream>
#include <vc1>
#pragma hdrstop

using namespace std;

class TBazowa {
public:
    int i;
    int __fastcall pokazB()
    {cout << "Jestem klasa bazowa" << endl;
    return i; }
};
//-----
class TPochodna1 : public TBazowa {
public:
    int j;
    int __fastcall pokazP1()
    {cout << "Jestem 1 klasa pochodna" << endl;
    return j;}
};
//-----

```

```

class TPochodna2 : public TBazowa {
public:
    int k;
    int __fastcall pokazP2()
    {cout << "Jestem 2 klasa pochodna" << endl;
    return k;}
};
//-----
// klasa TPochodna3 dziedziczy klasy TPochodna1 i TPochodna2,
// i zawiera dwie kopie klasy TBazowa
class TPochodna3 : public TPochodna1, public TPochodna2 {
public:
    int l;
    int __fastcall pokazP3()
    {cout << "Jestem 3 klasa pochodna" << endl;
    return l;}
};
//-----
int main()
{
    TPochodna3 klasa;
    klasa.i = 100;
    klasa.j = 200;
    klasa.k = 300;
    klasa.l = 400;
    cout << klasa.pokazP1() << endl;
    cout << klasa.pokazP2() << endl;
    cout << klasa.pokazP3() << endl;
    cin.get();
    return 0;
}
//-----

```

Podczas próby uruchomienia powyższego programu spotka nas przykra niespodzianka polegająca na tym, że program się po prostu nie skompiluje! Wynika to z faktu, iż jego konstrukcja jest niejednoznaczna, ponieważ wywołanie:

```
klasa.i = 100;
```

jest dla kompilatora niejednoznaczne:

```
[C++ Error] Unit_R4_05.cpp(44): E2014 Member is ambiguous:
'TBazowa::i' and 'TBazowa::i'
```

z tego powodu, że każdy egzemplarz klasy TPochodna3 zawiera dwie kopie elementów składowych klasy TBazowa. Ponieważ w tej sytuacji istnieją dwie kopie zmiennej *i* (deklarowanej w klasie bazowej), kompilator nie ma najmniejszej wiedzy na temat, którą kopię zmiennej ma wykorzystać — tę odziedziczoną przez klasę TPochodna1 czy tę z klasy TPochodna2.

Jeżeli dwie lub większa liczba klas dziedziczy z tej samej klasy bazowej, możemy zapobiec sytuacji, w której kopia klasy bazowej jest powielana w klasach potomnych w sposób niekontrolowany. Istnieje kilka sposobów, aby przeciwdziałać takiemu „samopowielaniu się” klasy bazowej. Najprostszym rozwiązaniem jest zadeklarowanie klas pochodnych jako klas kategorii `virtual`, tak jak pokazano to na listingu 4.6.

Listing 4.6. Kod głównego modułu *Unit_R4_06.cpp* projektu *Projekt_R4_06.bdsproj* wykorzystującego *wirtualne klasy bazowe*

```

#include <iostream>
#include <vc1>
#pragma hdrstop

using namespace std;

class TBazowa {
public:
    int i;
    int __fastcall pokazB()
    {cout << "Jestem klasa bazowa" << endl;
    return i; }
};
//-----
// klasa TPochodna1 dziedziczy klasę TBazowa jako wirtualną
class TPochodna1 : virtual public TBazowa {
public:
    int j;
    int __fastcall pokazP1()
    {cout << "Jestem 1 klasa pochodna" << endl;
    return j;}
};
//-----
// klasa TPochodna2 dziedziczy klasę TBazowa jako wirtualną
class TPochodna2 : virtual public TBazowa {
public:
    int k;
    int __fastcall pokazP2()
    {cout << "Jestem 2 klasa pochodna" << endl;
    return k;}
};
//-----
// klasa TPochodna3 dziedziczy klasy TPochodna1 i TPochodna2,
// ale zawiera jedną kopię klasy TBazowa
class TPochodna3 : public TPochodna1, public TPochodna2 {
public:
    int l;
    int __fastcall pokazP3()
    {cout << "Jestem 3 klasa pochodna" << endl;
    return l;}
};
//-----
int main()
{
    TPochodna3 klasa;
    klasa.i = 100;
    klasa.j = 200;
    klasa.k = 300;
    klasa.l = 400;
    cout << klasa.pokazP1() << endl;
    cout << klasa.pokazP2() << endl;
}

```

```
cout << klasa.pokazP3() << endl;  
cin.get();  
return 0;  
}  
//-----
```

Testując przedstawiony algorytm, natychmiast zauważymy, iż w klasie `TPochodna3` istnieje teraz już tylko jedna kopia klasy bazowej, gdyż klasy `TPochodna1` i `TPochodna2` dziedziczą klasę bazową jako klasę wirtualną, co skutkuje utworzeniem tylko jednej kopii klasy `TBazowa`.



Wskazówka

Klasy wirtualne należy wykorzystywać tylko wtedy, gdy w programie istnieje konieczność wielokrotnego dziedziczenia klas. Gdy klasa bazowa dziedziczona jest jako wirtualna, w programie tworzona jest tylko jedna jej kopia. Przez pojęcie wielokrotnego dziedziczenia (dziedziczenia poziomego) rozumiemy sytuację, w której jedna klasa jednocześnie dziedziczy po większej liczbie klas.

Funkcje wirtualne w C++Builderze

W C++Builderze, traktowanym jako kompletne środowisko programistyczne, istnieje klasa `TObject`, będąca podstawową klasą, po której dziedziczą wszystkie inne. Często mówimy, że klasa ta jest przodkiem wszystkich typów obiektowych C++Buildera, co między innymi oznacza, że na przykład zmienna wskazująca na typ `TObject` może wskazywać na obiekt dowolnej innej klasy dziedziczącej po `TObject`. Klasa `TObject` nie zawiera żadnych jawnych elementów składowych. Posiada natomiast elementy przechowujące wskaźniki do tzw. tablicy metod wirtualnych VMT.

Jak zapewne wywnioskowaliśmy z lektury niniejszego rozdziału, klasa bazowa (najbardziej ogólna) i klasy pochodne (klasy szczegółowe) tworzą pewną hierarchiczną strukturę danych. Z tego względu klasa bazowa powinna mieć te wszystkie elementy, z których korzystać będą klasy szczegółowe. W strukturze polimorficznej klasa bazowa powinna dodatkowo zawierać podstawowe wersje tych funkcji, które będą przykrywane w klasach pochodnych. Jeżeli tylko powyższe założenia zostaną spełnione, będziemy mogli posługiwać się pewnym spójnym interfejsem, zawierającym wiele implementacji różnych obiektów. Dobrą ilustracją tworzenia i posługiwania się prostą biblioteką klas będzie przykład, w którym, na podstawie pewnej klasy bazowej umożliwiającej przechowywanie dwu wymiarów pewnych figur geometrycznych przy wykorzystaniu standardowej funkcji składowej `dane()`, będziemy w stanie skonstruować klasy pochodne umożliwiające obliczanie pola prostokąta (klasa `TPochodna1`) i trójkąta (klasa `TPochodna2`). Funkcja `dane()` jest funkcją standardową, gdyż służy jedynie do przechowywania podstawowych wymiarów odpowiednich figur geometrycznych, natomiast funkcja `pokazPole()` powinna być kategorii `virtual`. Wynika to z faktu, iż pola powierzchni różnych figur oblicza się przy wykorzystaniu różnych wzorów. Na listingu 4.7 pokazano przykład implementacji biblioteki klas w C++Builderze.

Listing 4.7. Kod głównego modułu *Unit_R4_07.cpp* projektu *Projekt_R4_07.bdsproj*, wykorzystującego klasy polimorficzne konstruowane w stylu biblioteki klas VCL

```
#include <vcl>
#include <sysutils.hpp>
#include <iostream>
// Klasy polimorficzne w stylu VCL

using namespace std;

class TBazowa : public TObject
{
protected:
    double a,b;
public:
    void __fastcall dane(double x, double y)
    {a = x; b = y;}
    virtual void __fastcall pokazPole()
    {cout << "Jestem klasa bazowa i nic nie obliczam ";
      cout << "\n\n";}
};
//-----
class TPochodna1 : public TBazowa
{
public:
    virtual void __fastcall pokazPole()
    {cout << "Jestem 1 klasa pochodna i obliczam" << endl;
      cout << "pole prostokąta = " << a*b << "\n\n";}
      // a, b: długości boków prostokąta
};
//-----
class TPochodna2 : public TBazowa
{
public:
    virtual void __fastcall pokazPole()
    {cout << "Jestem 2 klasa pochodna i obliczam" << endl;
      cout << "pole trójkąta = " << 0.5*a*b << endl;}
      // a: długość podstawy trójkąta, b: wysokość trójkąta
};
//-----
int main()
{
    // utworzenie i zainicjowanie wskaźników do
    // typu bazowego i klas pochodnych
    TBazowa *ptrB = new TBazowa;
    TPochodna1 *ptrP1 = new TPochodna1;
    TPochodna2 *ptrP2 = new TPochodna2;

    cout << endl << "Jawne wywołanie funkcji pokazPole()"
         << " klas TBazowa, TPochodna1 i TPochodna2" << "\n\n";
    ptrB->pokazPole();

    ptrP1->dane(2,2);
    ptrP1->pokazPole();
}
```

```

ptrP2->dane(5,5);
ptrP2->pokazPole();

// wywołanie destruktorów klas
delete ptrB;
delete ptrP1;
delete ptrP2;
cout << endl;
system("PAUSE");
return 0;
}
//-----

```

Analizując powyższy program, natychmiast zauważymy, iż zainicjowanie kolejnych egzemplarzy klas wymaga utworzenia odpowiednich wskaźników oraz wykorzystania operatora `new` alokującego je na stacku. Ze względu na to, że typowi bazowemu oraz poszczególnym typom pochodnym został dynamicznie przydzielony pewien obszar pamięci, ma sens mówienie o utworzeniu *obiektów* poszczególnych klas, zwłaszcza że interfejs (sposób wywołania elementów):

```

ptrP1->dane(2,2);
ptrP1->pokazPole();
ptrP2->dane(5,5);
ptrP2->pokazPole();

```

obu klas pochodnych jest taki sam pomimo pewnych różnic w wykonywanych przez nie obliczeniach.

➔ Wskazówka

W dalszej części książki pojęcia *obiekt* będziemy używać w odniesieniu do elementów programistycznych, do których w trakcie działania programu zawsze można wysłać komunikaty oznaczone stereotypami `<<create>>` i `<<destroy>>`, to znaczy wielokrotnie wywołać ich konstruktor i destruktor. Oznacza to, iż czasem życia obiektów stworzonych na bazie odpowiednich klas można niemal dowolnie zarządzać.

Posługując się funkcjami wirtualnymi w C++, należy zwrócić baczną uwagę na sposób inicjowania elementów klas. Dane takich typów, jak odwołania oraz zmienne, muszą być zawsze we właściwy sposób inicjowane. Jednak niezależnie od tego, w jaki sposób są inicjowane, będą dalej pozostawać niezdefiniowane przed wywołaniem konstruktora klasy bazowej. W C++Builderze odpowiednie elementy biblioteki klas pisanych na wzór VCL są zawsze inicjowane wartością zero w momencie wywołania ich konstruktorów (listing 4.8).

Listing 4.8. Kod głównego modułu *Unit_R4_08.cpp* projektu *Projekt_R4_08.bdsproj*

```

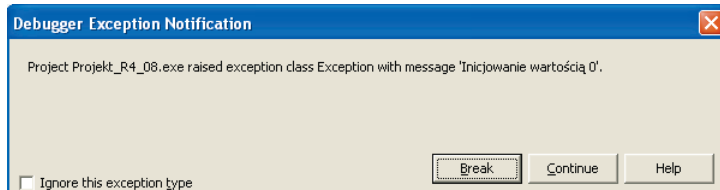
#include <vcl>
#include <sysutils.hpp>
#pragma hdrstop

class TBazowa : public TObject
{
public:
    __fastcall TBazowa() {init();}
    virtual void __fastcall init() { }
};
//-----

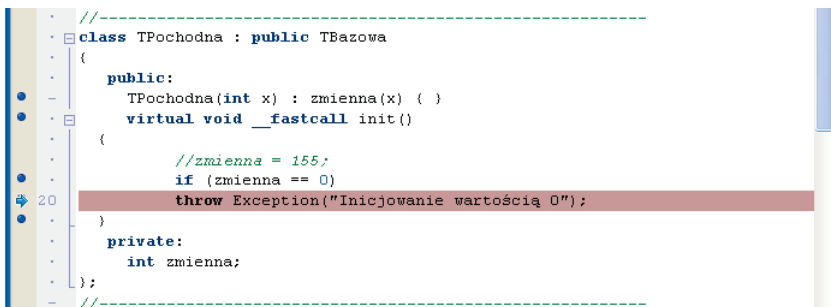
```

```
class TPochodna : public TBazowa
{
public:
    TPochodna(int x) : zmienna(x) { }
    virtual void __fastcall init()
    {
        // zmienna = 155;
        if (zmienna == 0)
            throw Exception("Inicjowanie wartością 0");
    }
private:
    int zmienna;
};
//-----
int main()
{
    TPochodna *ptrP = new TPochodna(155);
    delete ptrP;
    system("PAUSE");
    return 0;
}
//-----
```

W podanym przykładzie wyjątek przechwytywany jest ostatecznie przez konstruktor klasy bazowej, tak jak pokazano to sekwencją rysunków 4.2 – 4.5.

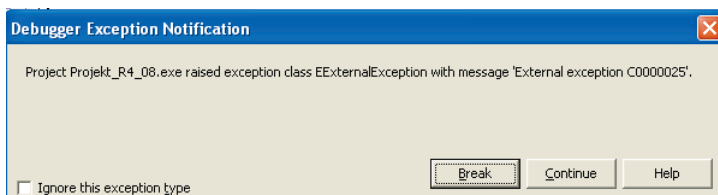


Rysunek 4.2. Prosty komunikat wyjątku

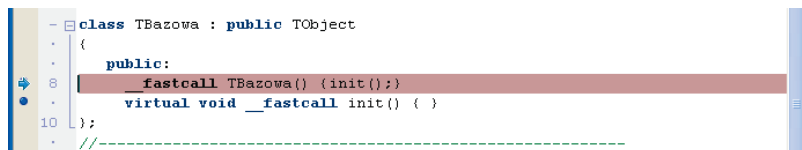


Rysunek 4.3. Miejsce generowania wyjątku

Ponieważ klasa bazowa jest tworzona w pierwszej kolejności, zmienna zero nie może w tym momencie być zainicjowana jakąkolwiek wartością, gdyż jest elementem typu pochodnego. Jak widzimy, nawet jawne wpisanie wartości do konstruktora klasy pochodnej nie przyniesie



Rysunek 4.4. Przetworzony wyjątek



Rysunek 4.5. Wyjątek przechwytywany przez konstruktor klasy bazowej

oczekiwanych efektów. Zawsze należy być świadomym faktu, iż nie można inicjować elementów składowych klas pochodnych pisanych w stylu VCL, zanim nie zostanie wywołany konstruktor klasy bazowej.

Klasy abstrakcyjne w stylu biblioteki VCL

Klasę, która zawiera przynajmniej jeden element w postaci tzw. *funkcji czysto wirtualnej* (ang. *pure virtual function*), nazywamy *klasą abstrakcyjną* (ang. *abstract class*). W ogólnej postaci funkcję czysto wirtualną możemy zapisać następująco:

```
virtual typ __fastcall nazwa_funkcji(<lista_parametrów>) = 0;
```

Funkcje takie są zawsze deklarowane w klasie bazowej, jednak nigdy nie są definiowane względem niej. Funkcja czysto wirtualna połączona z klasą abstrakcyjną oznacza, że funkcja taka nie ma swojej implementacji w macierzystej klasie. Funkcja czysto wirtualna zawsze powinna zostać zaimplementowana w klasach pochodnych.

Jako przykład praktycznego wykorzystania klasy abstrakcyjnej rozpatrzmy sytuację, w której klasa bazowa zawiera deklarację funkcji `pokazPole()`, traktowanej jako funkcja czysto wirtualna.

Listing 4.9. Moduł `Unit_R4_09.cpp` projektu `Projekt_R4_09.bdsproj` wykorzystującego klasę abstrakcyjną w stylu biblioteki klas VCL

```
#include <vcl>
#include <sysutils.hpp>
#include <iostream>

using namespace std;

class TBazowa : public TObject
{
```

```

protected:
    double a,b;
public:
    void __fastcall dane(double x, double y)
    {a = x; b = y;}
    // funkcja czysto wirtualna
    virtual void __fastcall pokazPole() = 0;
};
//-----
class TPochodna1 : public TBazowa
{
public:
    virtual void __fastcall pokazPole()
    {cout << "Jestem 1 klasa pochodna i obliczam" << endl;
     cout << "pole prostokata = " << a*b << "\n\n";}
};
//-----
class TPochodna2 : public TBazowa
{
public:
    virtual void __fastcall pokazPole()
    {cout << "Jestem 2 klasa pochodna i obliczam" << endl;
     cout << "pole trójkata = " << 0.5*a*b << endl;}
};
//-----
int main()
{
    // utworzenie i zainicjowanie obiektów klas z wykorzystaniem
    // konstruktorów domyślnych
    TPochodna1 *ptrP1 = new TPochodna1;
    TPochodna2 *ptrP2 = new TPochodna2;

    cout << endl << "Jawne wywołania funkcji pokazPole()"
         << " klas TPochodna1 i TPochodna2" << "\n\n";

    ptrP1->dane(2,2);
    ptrP1->pokazPole();

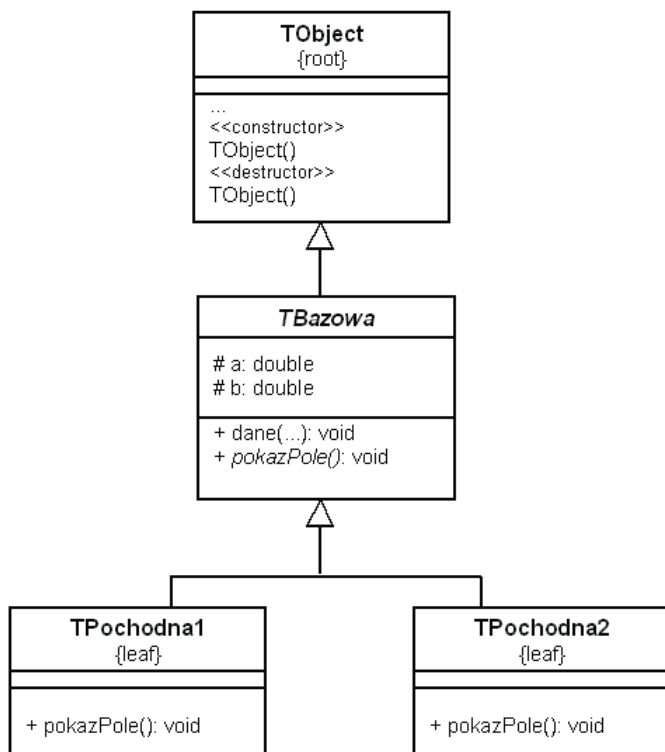
    ptrP2->dane(5,5);
    ptrP2->pokazPole();

    // wywołanie destruktorów klas i zniszczenie obiektów
    delete ptrP1;
    delete ptrP2;

    cout << endl;
    system("PAUSE");
    return 0;
}
//-----

```

Na rysunku 4.6 przedstawiono statyczny diagram klas dla przykład z listingu 4.9.



Rysunek 4.6. Wymodelowanie przykładu 4.9 w postaci statycznego diagramu klas

Analizując przykład 4.9, na pewno zauważymy, iż nie został utworzony i zainicjowany wskaźnik do typu bazowego. Wynika to z faktu, że klasa `TBazowa` zawiera jedną funkcję czysto wirtualną, co z kolei skutkuje niemożnością utworzenia jej obiektu. Jeżeli w programie głównym nastąpiłaby próba jawnego utworzenia i zainicjowania obiektu na bazie klasy abstrakcyjnej:

```
TBazowa *ptrB = new TBazowa;
```

pojawiający się komunikat kompilatora:

```
[C++ Error] Unit1.cpp(35): E2352 Cannot create instance of abstract class 'TBazowa'
```

nie pozostawia nam cienia wątpliwości, gdyż nie można skonstruować obiektu na podstawie klasy zawierającej funkcję czysto wirtualną.



Wskazówka

Nie można tworzyć obiektów na bazie klas abstrakcyjnych. Klas takich używamy wyłącznie w charakterze klas bazowych dziedziczonych przez inne klasy pochodne. Możliwe jest jednak tworzenie wskaźników nietraktowanych jako dynamiczne obiekty w stylu języka C++ (tzw. *wskaźników tradycyjnych*) do tego typu klas, np.:

```
TBazowa *ptrB;
```