helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++Builder 6. Ćwiczenia



Autor: Andrzej Daniluk
ISBN: 83-7197-986-X
Format: B5, stron: 128
Przykłady na ftp: 1391 kB

Borland C++ Builder to jedno z najwygodniejszych środowisk programistycznych dla programistów C++, platforma ciesząca się dużą popularnością i mająca za sobą długą historię. W języku C++ napisano wiele aplikacji dla Windows, z których znaczna część powstała właśnie w Builderze.

„C++ Builder. Ćwiczenia" to uzupełnienie poprzedniej publikacji Wydawnictwa Helion, zatytułowanej „C++ Builder 5. Ćwiczenia praktyczne". Książka omawia zmiany, jakie wprowadzono w nowej, szóstej już wersji C++ Buildera, a także porusza wiele zagadnień, które nie znalazły się w książce traktującej o poprzedniej edycji tego programu. Informacje zostały przekazane w formie ćwiczeń z dokładnym omówieniem prezentowanego kodu źródłowego.

Znajdziesz w niej między innymi:

- Zagadnienia związane z kompatybilnością pomiędzy wersjami piątą i szóstą C++ Buildera
- Serię ćwiczeń przybliżających język C++
- Omówienie środowiska IDE C++ Builder
- Ćwiczenia z programowania w C++ z wykorzystaniem C++ Builder
- Ćwiczenia z pisania aplikacji wielowątkowych
- Sposoby tworzenia własnych komponentów

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

Spis treści

Wstęp 5

Rozdział 1. Konfiguracja projektu 7

Kompatybilność wersji Buildera 8
Podsumowanie 9

Rozdział 2. C++ w pigułce 11

Pliki nagłówkowe 11
Przestrzenie nazw standardowych 14
Klasy wejścia-wyjścia języka C++ 14
Obsługa plików z wykorzystaniem klasy ios 17
Struktury w C++ 18
Samodzielne tworzenie plików nagłówkowych 21
Klasy w C++ 22
Konstruktor i destruktor 27
Inne spojrzenie na klasy. Własności 29
Funkcje przeładowywane 31
Niejednoznaczność 33
Funkcje ogólne 34
Przeładowywanie funkcji ogólnych 36
Typ wyliczeniowy 37
Dziedziczenie 38
Funkcje wewnętrzne 42
Realizacja przekazywania egzemplarzy klas funkcjom 43
Tablice dynamicznie alokowane w pamięci 45
Tablice otwarte 48
Wskaźniki do egzemplarzy klas 50
Wskaźnik this 51
Obsługa wyjątków 52
Podsumowanie 56

Rozdział 3. Środowisko programisty — IDE 57

Biblioteka VCL 59
Karta Standard 59
Karta Additional 61
Karta Win32 63
Karta System 65
Karta Dialogs 66

Biblioteka CLX 67
Karta Additional 67
Karta Dialogs 68
Podsumowanie 68

Rozdział 4. C++ w wydaniu Buildera 6 69

Formularz 69
Zdarzenia 71
Rzutowanie typów danych 78
Klasa TObject 78
Wykorzystujemy własne funkcje 79
Wykorzystujemy własny, nowy typ danych 81
Widok drzewa obiektów — Object Tree View 84
Komponenty TActionManager i TActionMainMenuBar 84
Typy wariantowe 89
Tablice wariantowe 91
Klasy wyjątków 93
Więcej o wskaźniku this 96
Podsumowanie 98

Rozdział 5. Biblioteka CLX 99

Komponenty TTimer i TLCDNumber 99
Podsumowanie 103

Rozdział 6. Tworzymy własne komponenty 105

Podsumowanie 110

Rozdział 7. Aplikacje wielowątkowe 111

Funkcja BeginThread() 111
Komponent TChart 114
Podsumowanie 116

Rozdział 8. C++Builder jako wydajne narzędzie obliczeniowe 119

Obliczenia finansowe 119
Podsumowanie 126

Rozdział 2.

C++ w pigułce

Ten rozdział poświęcony jest skrótowemu omówieniu podstawowych pojęć, którymi posługujemy się tworząc programy w C++. Zagadnienia tutaj poruszane posiadają kluczowe znaczenie dla zrozumienia idei programowania zorientowanego obiektowo w środowisku C++. Używana terminologia oraz zamieszczone w dalszej części rozdziału przykłady (o ile nie zostały użyte elementy z biblioteki VCL) zgodne są ze standardem ANSI X3J16/ISO WG21 języka C++[1].

Pliki nagłówkowe

W odróżnieniu od programów pisanych w standardowym języku C, gdzie nie musimy zbytnio przejmować się dołączaniem do kodu źródłowego wielu plików nagłówkowych, w programach C++ nie można ich pominąć. Wynika to z faktu, iż bardzo wiele funkcji bibliotecznych korzysta z własnych struktur oraz typów danych. W C++ ich definicje znajdują się właśnie w plikach nagłówkowych (ang. *header files*), które posiadają standardowe rozszerzenie *.h*. C++ jest językiem bazującym na funkcjach, dlatego do pliku źródłowego programu musimy włączyć przy pomocy dyrektywy `#include` odpowiednie pliki zawierające wywoływane funkcje wraz z ich prototypami. Większość standardowych plików nagłówkowych znajduje się w katalogu instalacyjnym *\INCLUDE*. W dalszej części rozdziału wiadomości na temat praktycznego wykorzystania zarówno standardowych, jak i samodzielnie tworzonych plików nagłówkowych znacznie rozszerzymy.

Prosty przykład wykorzystania standardowych plików nagłówkowych *iostream.h* (zawiera definicje klas umożliwiające wykonywanie różnorodnych operacji wejścia-wyjścia na strumieniach) oraz *conio.h* (zawiera funkcje obsługi ekranu) zawiera poniższe ćwiczenie.

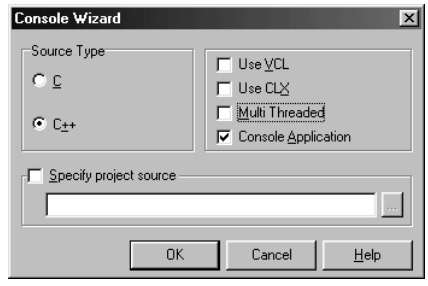
[1] Musimy zauważyć, iż Borland C++Builder traktowany jako kompletne środowisko programistyczne z pewnych względów nie spełnia wymogów ISO.

Ćwiczenie 2.1.

1. Stwórzmy na dysku odrębny katalog (folder) nazywając go po prostu `\01`. W katalogu tym przechowywane będą wszystkie pliki wykorzystywane przez aktualnie pisany program.

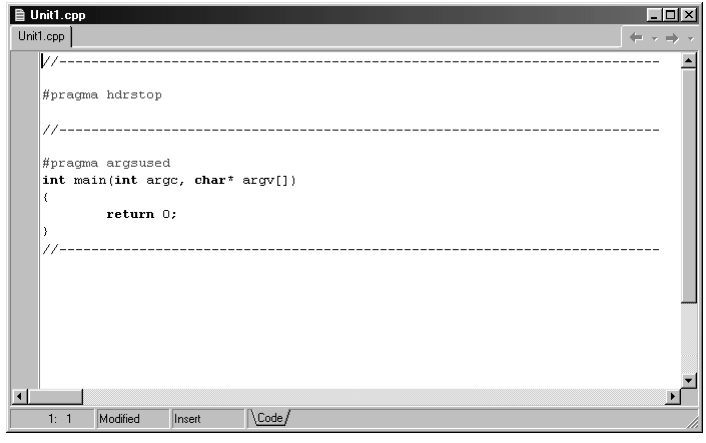
2. Uruchamiamy C++Buildera 6. Poleceniem menu *File\New\Other\Console Wizard* otwórzmy nowy moduł. W okienku dialogowym *Console Wizard* w opcji *Source Type* wybierzmy *C++*, zaś w drugim panelu odznaczmy *Use VCL*, *Use CLX*, *Multi Thread* oraz wybierzmy *Console Application*, tak jak pokazuje to rysunek 2.1. Zaznaczenie tej opcji powoduje, że program będzie traktował główny formularz tak, jakby był normalnym okienkiem tekstowym. Pozostawienie aktywnej opcji *Use CLX* (CLX jest skrótem od angielskiego terminu, określającego pewną klasę bibliotek wspomagających proces projektowania aplikacji przenośnych pomiędzy Windows a Linux: *Cluster software runing under LinuX*) spowoduje automatyczne dołączenie do programu pliku nagłówkowego *clx.h* wspomagającego tworzenie aplikacji międzyplatformowych.

Rysunek 2.1.
Okno Console Wizard



3. Potwierdzając przyciskiem *OK* przechodzimy do okna zawierającego szkielet kodu przyszłego programu, tak jak pokazuje to rysunek 2.2.

Rysunek 2.2.
Kod modułu Unit1.cpp



4. Programów naszych nie będziemy uruchamiać z linii poleceń, dlatego okno edycji kodu wypełnimy tekstem pokazanym na wydruku 2.1. Następnie poleceniem *File\Save As...* zapiszemy nasz moduł w katalogu *01* jako `Unit01.cpp`. Projekt modułu zapiszemy poleceniem *File\Save Project As...* w tym samym katalogu: `\01\Projekt_01.bpr`.

Wydruk 2.1. *Kod modułu Unit01.cpp projektu Projekt_01.bpr*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
int main()
{
   clrscr();
   cout << "Dzień dobry !" << endl;
   getch();
   return 0;
}
```

5. Po uruchomieniu programu poleceniem *Run\Run (F9)*, na ekranie w okienku udającym tryb tekstowy Windows pojawi się napis powitania. Program opuszczamy naciskając klawisz *Enter*.

Jak widać, w skład tego programu wchodzą dwa pliki nagłówkowe: *iostream.h* oraz *conio.h*. Pierwszy z nich zdefiniowany jest w C++ i umożliwia wykonywanie szeregu operacji wejścia-wyjścia. Powodem włączenia pliku *conio.h* jest zastosowanie funkcji `clrscr()` (ang. *clear screen*) czyszczącej ekran tekstowy oraz funkcji `getch()` (ang. *get character*) oczekującej na naciśnięcie dowolnego klawisza (wprowadzenie dowolnego znaku). Użycie dyrektywy `#pragma hdrstop` (ang. *header stop*) informuje kompilator o końcu listy plików nagłówkowych.

Każdy program pisany w C++ musi zawierać przynajmniej jedną funkcję. Główna funkcja `main()` jest tą, która zawsze musi istnieć w programie i zawsze wywoływana jest jako pierwsza. Zestaw instrukcji właściwych danej funkcji musi być zawarty w nawiasach klamrowych `{...}`, będących swojego rodzaju odpowiednikiem `begin...end` w Pascalu.



Instrukcje zawarte w nawiasach klamrowych nazywamy blokiem instrukcji (ang. *code block*); jest on grupą logicznie powiązanych ze sobą elementów traktowanych jako niepodzielny fragment programu.

Każda funkcja określonego typu powinna zwracać wartość tego samego typu. W powyższym przykładzie funkcja `main()` jest typu całkowitego `int`, zatem musi zwrócić do systemu taką samą wartość. Tutaj wykonaliśmy tę operację używając instrukcji `return 0`, która jest niczym innym, jak jedną z możliwych wartości powrotnych udostępnianych w następstwie wywołania funkcji `int main()`. Jeżeli funkcja byłaby typu nieokreślonego, czyli `void` (tzw. typ pusty, pusta lista parametrów), wówczas nie musielibyśmy zwracać do systemu żadnej wartości, tak jak ilustruje to poniższy przykład:

```
void main()
{
   clrscr();
   cout << "Dzień dobry !" << endl;
   getch();
   return;
}
```

Brak wartości zwracanej przez funkcję `void main()` w powyższym przykładzie wynika z faktu, że w programach C++ nieokreślona wartość funkcji (lub pusta lista parametrów) równoznaczna jest ze słowem `void`.

W języku C++ słowo `cout` identyfikuje ekran (ale nie formularz !), zaś wraz z operatorem << pozwala wyprowadzić zarówno łańcuchy znaków, jak i zmienne wszystkich typów. Słowo `endl` (ang. *end of line*) odpowiada wysłaniu kombinacji znaków `CR-LF` ustawiających kursor na początku następnego wiersza.

Przestrzenie nazw standardowych

Studiując niektóre programy C++ możemy zauważyć, iż przed słowami `cout` i `endl` może występować słowo `std::`. Informuje ono kompilator o potrzebie korzystania z tzw. wyznacznika przestrzeni nazw. Chodzi o to, aby kompilator wiedział, że używamy strumieni `cout` i `endl` z biblioteki standardowej:

```
#include <iostream.h>
int main()
{
  using std::cout;
  using std::endl;
  cout << "tekst " << endl;
  ...
}
```

Często też programiści, w celu uniknięcia niedogodności pisania `std::` przed każdym `cout` i `endl`, po prostu informują kompilator o potrzebie używania całej przestrzeni nazw standardowych, tj. że każdy obiekt, który nie zostanie oznaczony, z założenia będzie pochodził z przestrzeni nazw standardowych. W tym przypadku, zamiast konstrukcji `using std::cout;` piszemy po prostu `using namespace std;`.

```
#include <iostream.h>
int main()
{
   using namespace std;
   cout << "tekst " << endl;
   ...
}
```

W książce tej nie będziemy jawnie rozróżniać przestrzeni nazw standardowych.

Klasy wejścia-wyjścia języka C++

Język C++ posługuje się kilkoma predefiniowanymi łańcuchami wejścia-wyjścia. Dwa najbardziej podstawowe z nich, związane ze standardowym wejściem-wyjściem, to `cout` oraz `cin`. Z pierwszym z nich zapoznaliśmy się już wcześniej. Słowo `cin` wraz z operatorem >> pozwala wprowadzać zarówno łańcuchy znaków, jak i różnego rodzaju zmienne.

Oprócz opisanych predefiniowanych łańcuchów, w C++ zdefiniowany jest jeszcze szereg tzw. klas strumieni wejścia-wyjścia. Trzy podstawowe klasy wejścia-wyjścia to:

- `ofstream` (ang. *output file stream*) — wyprowadzanie danych,
- `ifstream` (ang. *input file stream*) — wprowadzanie danych,
- `fstream` (ang. *file stream*) — wprowadzanie i wyprowadzanie danych.

Dwa proste ćwiczenia pozwolą nam zapoznać się z właściwościami wymienionych klas.

Ćwiczenie 2.2.

Zaprojektujemy program, którego jedynym zadaniem będzie odczytanie swojego własnego tekstu źródłowego zawartego w pliku *.cpp* i wyświetlenie go na ekranie. Kod tego programu, korzystającego z uniwersalnej klasy `fstream`, pokazany jest na wydruku 2.2.

Wydruk 2.2. *Moduł Unit02.cpp projektu Projekt_02.bpr*

```
1: #include <iostream.h>
2: #include <fstream.h>
3: #include <conio.h>
4: #pragma hdrstop

5: int main()
6: {
7:   char bufor[100];
8:   fstream InFile;

9:   clrscr();
10:  InFile.open("Unit02.cpp");
11:  while (!InFile.eof()) {
12:     InFile.getline(bufor, sizeof(bufor));
13:     cout << bufor << endl;
14:  }
15:   InFile.close();
16:   getch();
17:   return 0;
18: }
```

Wykorzystanie omawianych na początku niniejszego podrozdziału klas wejścia-wyjścia wymaga włączenia do programu pliku nagłówkowego *fstream.h*, tak jak wykonaliśmy to w linii 2. programu.

Dane z pliku będziemy wczytywać znak po znaku, zatem wymagane jest zadeklarowanie w linii 7. programu bufora danych typu znakowego `char` o odpowiednim rozmiarze.

Chociaż budowę klas i różne aspekty ich wykorzystywania omówimy dokładniej w dalszej części rozdziału, jednak już w tym miejscu zaczniemy posługiwać się odpowiednią terminologią. Omawiając deklarację z linii 8. w sposób tradycyjny, powiedzielibyśmy, iż została tam zadeklarowana zmienna `InFile` typu `fstream`. Jednak w odniesieniu do klas wygodniej jest posługiwać się innym sformułowaniem — powiemy, że w linii 8. programu zadeklarowaliśmy egzemplarz `InFile` klasy `fstream` (bardzo często używa się też sformułowania, że został utworzony strumień wejściowy `InFile`).

Jak zapewne wiesz, każdy plik, którego zawartość chcemy w odpowiedni sposób wyświetlić, musi być najpierw otwarty do czytania. Czynność tę wykonujemy w linii 9. poprzez odwołanie się do egzemplarza (obiektu) klasy `fstream` z jawnym wskazaniem

funkcji, jakiej ma używać do otwarcia pliku. W terminologii stosowanej w programowaniu zorientowanym obiektowo powiemy, iż strumień wejściowy `InFile` połączyliśmy z plikiem dzięki wywołaniu funkcji składowej klasy (metody) `open()`, której argumentem jest nazwa pliku umieszczona w podwójnych apostrofach.

W przypadku, gdy czytany plik nie znajduje się w aktualnym katalogu, należy podać pełną ścieżkę dostępu według następującego przepisu:

```
InFile.open("c:\\cwbild_6\\kody\\01\\Unit01.cpp");
```

Proces czytania i wyświetlania na ekranie zawartości pliku wykonywany jest w liniach 11. – 14. przy pomocy instrukcji iteracyjnej `while`, która będzie wykonywana do czasu napotkania znacznika końca pliku. Osiągnięcie końca pliku wykrywamy za pomocą funkcji składowej `eof()` (ang. *end of file*).

W linii 12. przy pomocy dwuparametrowej funkcji składowej `getline()` wczytujemy zawartość pliku do bufora. Pierwszym parametrem tej funkcji jest zmienna `bufor` identyfikująca bufor danych wejściowych. Drugim parametrem jest jednoargumentowy operator czasu kompilacji `sizeof()` udostępniający długość zmiennej `bufor`. Operator ten wykorzystujemy w przypadku, gdy wymagamy, aby kod źródłowy programu był w dużej mierze przenośny, tzn. by można było wykorzystywać go na różnego rodzaju komputerach. Bardzo często warunek przenośności kodu wymaga, aby ustalić rzeczywistą długość danych, np. typu `char`.

W linii 15. wywołujemy funkcję składową `close()` zamykającą otwarty uprzednio plik.

Ćwiczenie 2.3.

Obecne ćwiczenie ilustruje sposób posługiwania się klasami `ofstream` oraz `ifstream`. Poprzedni program został zmodyfikowany w taki sposób, aby po utworzeniu w aktualnym katalogu nowego pliku można było zapisać w nim odpowiednie dane, którymi w tym przypadku są wartości funkcji `sin()` (sinus), której prototyp znajduje się w pliku nagłówkowym *math.h*.

Wydruk 2.3. *Moduł Unit03.cpp projektu Projekt_03.bpr*

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <conio.h>
#pragma hdrstop

int main()
{
   char bufor[100];
   clrscr();
//--- tworzenie nowego pliku i zapis danych na dysk--------
   ofstream OutFile("nowy.dat");
   if (! OutFile)
     return 0;
   for (int i=0; i<=15; i++)
     OutFile << sin(i) << endl;
   OutFile.close();
```

```
//--- odczyt danych -------------------------------------
   ifstream InFile("nowy.dat");
   while (!InFile.eof()) {
     InFile.getline(bufor, sizeof(bufor));
     cout << bufor << endl;
   }
   InFile.close();

   getch();
   return 0;
}
```

Analizując powyższe zapisy łatwo zauważymy, iż zaraz po utworzeniu nowego pliku, przy pomocy instrukcji warunkowej `if` sprawdzamy, czy czynność ta została wykonana pomyślnie. W przypadku niemożności stworzenia na dysku pliku o podanej nazwie, przy pomocy instrukcji `return 0` wywołujemy wartość powrotną funkcji `main()`, co jest równoznaczne z opuszczeniem programu.

W omawianym programie nie została użyta funkcja `open()`. Funkcji tej nie używa się zbyt często korzystając z klas strumieni, z tego względu, iż klasy `fstrem`, `ifstrem` oraz `ofstrem` posiadają odpowiednie konstruktory (inne funkcje składowe), które otwierają pliki automatycznie.

Obsługa plików z wykorzystaniem klasy ios

Jak zapewne pamiętasz, istnieją dwa podstawowe rodzaje plików, którymi w praktyce posługujemy się najczęściej. Są to pliki tekstowe oraz binarne. Pliki tekstowe, stanowiące zbiór znaków ASCII (lub Unicode), przechowują zawarte w nich informacje w kolejnych wierszach, z których każdy zakończony jest parą znaków `CR LF`. Pliki binarne zawierające kodowane dane różnią się od tekstowych tym, iż informacje w nich zawarte nie są przeznaczone do bezpośredniego oglądania — są zrozumiałe jedynie dla określonych programów. Bardzo wygodny dostęp do różnego rodzaju plików dają nam elementy klasy `ios`. Wspomniano wcześniej, że jednym ze sposobów połączenia uprzednio utworzonego strumienia z plikiem jest wywołanie funkcji:

```
void open(const char* s, ios_base::openmode = ios_base::out |
          ios_base::in, long protection = 0666);
```

gdzie `s` oznacza nazwę pliku, która może zawierać pełną ścieżkę dostępu. Tryb otwarcia pliku określa parametr `openmode`, który musi być przedstawiony w postaci jednej lub większej liczby określonych wartości, połączonych przy pomocy operatora binarnej sumy logicznej `|`. W tabeli 2.1 zebrano dostępne wartości, jakie może przyjmować parametr `openmode`.

Natomiast parametr `protection` opcjonalnie określa otwarcie zwykłego pliku.

Jeżeli zechcemy, aby plik został otworzony np. w trybie do dopisywania, wystarczy posłużyć się bardzo prostą konstrukcją:

```
ofstream OutFile("nowy.dat", ios::app);
```

A dopisywania w trybie binarnym:

```
ofstream OutFile("nowy.dat", ios::app | ios::binary);
```

Tabela 2.1. *Dopuszczalne tryby otwarcia pliku*

Wartości openmode	Opis
`ios::app`	Otwarcie pliku w trybie do dopisywania (dołączenie wprowadzanych danych na końcu pliku).
`ios::ate`	Ustawienie wskaźnika pliku na jego końcu.
`ios::in`	Otwarcie pliku w tzw. trybie wejściowym (w trybie do czytania). Wartość domyślna dla strumienia `ifstream`.
`ios::out`	Otwarcie pliku w tzw. trybie wyjściowym (w trybie do zapisywania). Wartość domyślna dla strumienia `ofstream`.
`ios::binary`	Otwarcie pliku binarnego.
`ios::trunc`	Po otwarciu zawartość pliku zostanie usunięta.

Ćwiczenie 2.4.

Korzystając z tabeli 2.1 zmodyfikuj projekt *Projekt_03.bpr* (wydruk 2.3) w ten sposób, aby sprawdzić działanie pozostałych metod otwarcia pliku.

Struktury w C++

Struktury, tworzące zbiór zmiennych złożonych z jednej lub z logicznie powiązanych kilku danych różnych typów, zgrupowanych pod jedną nazwą, są bardzo wygodnym typem danych, często definiowanym przez programistów C++. Na potrzeby obecnych ćwiczeń przedstawimy dwa bardzo często stosowane sposoby deklaracji oraz używania struktur. Słowo kluczowe `struct` (struktura) ułatwia logiczne pogrupowanie danych różnych typów. Po słowie `struct` w nawiasach klamrowych deklarujemy elementy składowe struktury wraz z ich typami bazowymi.

Ćwiczenie 2.5.

Zbudujemy naprawdę prostą bazę danych, w której przechowywać będziemy pewne informacje o wybranych studentach, tak jak pokazano to na wydruku 2.4.

Wydruk 2.4. *Moduł Unit04.cpp projektu Projekt_04.bpr ilustrującego prosty przykład wykorzystania informacji zawartej w pewnej strukturze*

```
1: #include <iostream.h>
2: #include <conio.h>
3: #include <stdlib.h>
4: #pragma hdrstop

5: struct  Student{
6:   char  Imie[15];
7:   char  Nazwisko[20];
8:   float EgzaminMatematyka;
9:   float EgzaminFizyka;
10:  float EgzaminInformatyka;
```

```
11:  char JakiStudent[40];
12:};

13: void StudentInfo(int, Student Info);
//-------------------------------------------------------
14: int main()
15: {
16:   Student Dane[3];
17:   int index=0;
18:   char buforM[5],buforF[5],buforI[5];
19:   clrscr();
20:   do {
21:     cout << "Imie: ";
22:     cin.getline(Dane[index].Imie, sizeof(Dane[index].Imie)-1);

23:     cout << "Nazwisko: ";
24:     cin.getline(Dane[index].Nazwisko,
                sizeof(Dane[index].Nazwisko)-1);

25:     cout << "Egzamin Matematyka: ";
26:     cin.getline(buforM, sizeof(buforM)-1);
27:     Dane[index].EgzaminMatematyka = atof(buforM); //atof!!!


28:     cout << "Egzamin Fizyka: ";
29:     cin.getline(buforF, sizeof(buforF)-1);
30:     Dane[index].EgzaminFizyka = atof(buforF);

31:     cout << "Egzamin Informatyka: ";
32:     cin.getline(buforI, sizeof(buforI)-1);
33:     Dane[index].EgzaminInformatyka = atof(buforI);

34:     cout << "Opinia: ";
35:     cin.getline(Dane[index].JakiStudent,
                sizeof(Dane[index].JakiStudent)-1);

36:     cout << endl;
37:     index++;
38:   } while (index<2);

39:   for (int i=0; i<2; i++)
40:      StudentInfo(i, Dane[i]);
41:   getch();
42:   return 0;
43: }
//-------------------------------------------------------
44: void StudentInfo(int numer, Student Info)
45: {
46:  cout << "Osoba (rekord) nr " << (numer+1) << " : " << endl;
47:  cout << "Imię i Nazwisko: "<< Info.Imie <<" ";
48:  cout << Info.Nazwisko << endl;
49:  cout << "Egzamin Matematyka: "
       << Info.EgzaminMatematyka << endl;
50:  cout << "Egzamin Fizyka: " << Info.EgzaminFizyka << endl;
51:  cout << "Egzamin Informatyka: "
       << Info.EgzaminInformatyka << endl;
52:  cout << "Opinia: " << Info.JakiStudent << endl;
53: }
```

W programie głównym w liniach 5. – 12. definiujemy strukturę `Student`, w której polach zapisywać będziemy interesujące nas informacje, czyli imię i nazwisko danej osoby, oceny z egzaminów z wybranych przedmiotów i na koniec naszą opinię o studencie.

Ponieważ chcemy mieć możliwość przechowywania w polach struktury informacji o większej liczbie osób, dlatego w linii 16. deklarujemy tablicę `Dane` typu strukturalnego `Student`. Informacje przechowywane w polach struktury będą indeksowane, musimy zatem zadeklarować w linii 17. zmienną `index` typu całkowitego.

Oceny z poszczególnych egzaminów deklarowane są w strukturze `Student` jako prosty typ zmiennopozycyjny `float`. W trakcie działania programu oceny te wpisywać będziemy z klawiatury w postaci ciągu znaków, które w dalszym etapie powinny zostać przekonwertowane na konkretne liczby. Z tego powodu łańcuchy znaków reprezentujących poszczególne oceny będą przechowywane w odpowiednich buforach deklarowanych w linii 18. programu.

Proces wypełniania poszczególnych pól tablicy struktur `Dane` odbywa się w liniach 20. – 38. programu w pętli `do...while`. I tak, w pierwszej kolejności za pomocą funkcji składowej `getline()` strumienia wejściowego `cin` wypełniamy pola `Imie` oraz `Nazwisko` tablicy struktur `Dane` (linie 22. – 24.). Elementy tablicy struktur (osoby) są indeksowane począwszy od wartości `0`.

W linii 26. wczytujemy do bufora `buforM` ciąg znaków reprezentujących ocenę z wybranego przedmiotu. W linii 27. ciąg znaków znajdujący się w buforze zostanie zamieniony na liczbę typu zmiennopozycyjnego za pomocą zdefiniowanej w pliku nagłówkowym *stdlib.h* funkcji `atof()`. W tej samej linii programu liczba ta zostanie wpisana do pola `EgzaminMatematyka` tablicy struktur `Dane` z odpowiednim indeksem określającym konkretną osobę. W podobny sposób wypełniamy pozostałe pola tablicy struktur `Dane` do momentu, kiedy zmienna `index` nie przekroczy wartości określającej liczbę „zapamiętanych" osób.

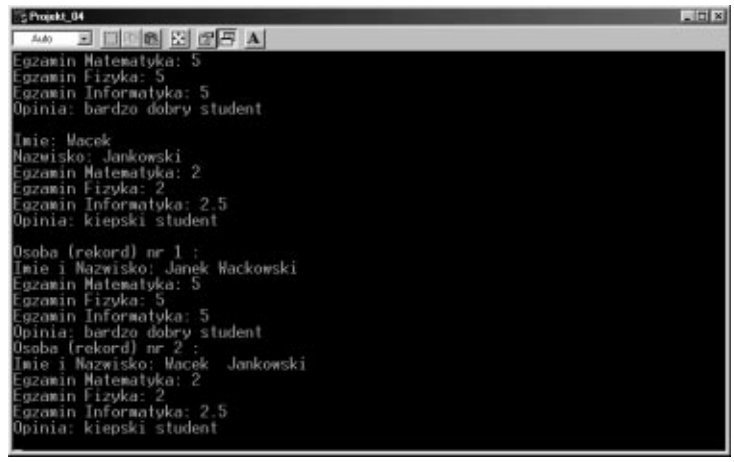
W dalszym etapie rozwoju naszego programu zajdzie prawdopodobnie potrzeba ponownego wyświetlenia (lub wyszukania i wyświetlenia) informacji o wybranych osobach (lub konkretnej osobie). Musimy zatem zaprojektować prostą funkcję rozwiązującą ten problem. W C++ istnieje konieczność deklarowania prototypów samodzielnie definiowanych funkcji. Prototyp naszej funkcji `StudentInfo()` typu `void` (funkcja nie zwraca żadnej wartości) umieszczony jest w linii 13. programu. Funkcja posiada dwa parametry: pierwszy typu całkowitego `int`, określający numer osoby, drugi `Info` typu strukturalnego `Student`, umożliwiający odwoływanie się do konkretnych pól struktury (informacji zawartych w strukturze).

Zapis funkcji `StudentInfo()` znajduje się w liniach 44. – 53. naszego programu. Łatwo zauważymy, iż odpowiednie dane pobierane są z określonych pól struktury poprzez strumień wyjściowy `cout`.

Proces wyświetlania danych na ekranie odbywa się w liniach 39. – 40., gdzie zmienna sterująca `i` w pętli `for()` przebiega już tylko indeksy tablicy struktur `Dane`.

Na rysunku 2.3 pokazano omawiany program w trakcie działania.

Rysunek 2.3.
Projekt Projekt_04.bpr po uruchomieniu



Samodzielne tworzenie plików nagłówkowych

Wspominaliśmy wcześniej, iż w C++ bardzo wiele funkcji bibliotecznych korzysta z własnych struktur oraz typów danych, których definicje znajdują się w plikach nagłówkowych. Korzystając z faktu, iż stworzyliśmy przed chwilą własny strukturalny typ danych, zmodyfikujemy *Projekt_04.bpr* w ten sposób, aby posługiwał się plikiem nagłówkowym zawierającym definicję struktury `Student`.

Ćwiczenie 2.6.

1. Poleceniem menu *File\New\Other\Header File* uaktywniamy okno edycji ułatwiające tworzenie i zapisywanie na dysku własnych, nowo tworzonych plików nagłówkowych.
2. Kod pliku zaczyna się od dyrektywy kompilacji warunkowej `#ifndef` (ang. *if not defined* — jeśli nie zdefiniowano). Za dyrektywą następuje pisana dużymi literami nazwa makra z reguły rozpoczynająca się od znaku podkreślenia.
3. Następnie, przy pomocy dyrektywy `#define` definiujemy nazwę makra w ten sposób, aby zaczynała się od znaku podkreślenia. Nazwa makra powinna odpowiadać nazwie, którą później nadamy plikowi nagłówkowemu.
4. Po zdefiniowaniu makra, należy zdefiniować własny typ strukturalny.
5. Definicja makra kończy się dyrektywą kompilacji warunkowej `#endif`.

```
#ifndef _STUDENT_H   //ale nie ifdef!!!
#define _STUDENT.H
struct  Student{
  char  Imie[15];
  char  Nazwisko[20];
  float EgzaminMatematyka;
  float EgzaminFizyka;
  float EgzaminInformatyka;
  char JakiStudent[40];
};
#endif
```

6. Tak określony plik nagłówkowy zapiszemy w aktualnym katalogu pod nazwą `student.h`.



Bardzo często (nawet przez nieuwagę) osoby rozpoczynające naukę C++ popełniają pewien błąd. Mianowicie zamiast dyrektywy `#ifndef` używają bardzo podobnej w zapisie `#ifdef` (ang. *if defined* — jeśli zdefiniowano). Różnica pomiędzy tymi dyrektywami jest dosyć wyraźna.

Jeżeli za pomocą dyrektywy `#define` została wcześniej zdefiniowana pewna makrodefinicja, wówczas sekwencja instrukcji występująca pomiędzy dyrektywami `#ifdef` oraz `#endif` będzie zawsze kompilowana.

Sekwencja instrukcji występująca pomiędzy dyrektywami `#ifndef` oraz `#endif` będzie kompilowana jedynie wówczas, gdy dana makrodefinicja nie została wcześniej zdefiniowana.

7. Tak przygotowany plik nagłówkowy zawierający definicję struktury `Student` bez problemu wykorzystamy w naszym programie. W tym celu wystarczy dołączyć go do kodu zaraz po dyrektywie `#pragma hdrstop`:

```
...
#pragma hdrstop
#include "student.h"
void StudentInfo(int, Student Info);
...
```

Zmodyfikowany *Projekt_04.bpr* możemy już samodzielnie przetestować.

Klasy w C++

Klasa jest jednym z podstawowych pojęć obiektowego języka C++. Przy pomocy słowa kluczowego `class` definiujemy nowy typ danych, będący w istocie połączeniem danych i instrukcji, które wykonują na nich działania. Umożliwia on tworzenie nowych (lub wykorzystanie istniejących) obiektów będących reprezentantami klasy. Konstrukcja klasy umożliwia deklarowanie elementów *prywatnych* (ang. *private*), *publicznych* (ang. *public*) oraz *chronionych* (ang. *protected*). Domyślnie, w standardowym języku C++ wszystkie elementy klasy traktowane są jako prywatne, co oznacza, że dostęp do nich jest ściśle kontrolowany i żadna funkcja nie należąca do klasy nie może z nich korzystać. Jeżeli w definicji klasy pojawią się elementy publiczne, oznacza to, że mogą one uzyskiwać dostęp do innych części programu. Chronione elementy klasy dostępne są jedynie w danej klasie lub w klasach potomnych. Ogólną postać definicji klasy można przedstawić w sposób następujący:

```
class NazwaKlasy
{
  private:
    //prywatne dane i funkcje
  public:
    //publiczne dane i funkcje
  protected:
    //chronione dane i funkcje

} EgzemplarzeKlasy; // lista egzemplarzy klasy
```

Definicja klasy jest zawsze źródłem definicji jej obiektów. Jeszcze kilkanaście lat temu przed pojawieniem się wizualnych środowisk programistycznych słowo *obiekt* (ang. *object*) było jednoznacznie utożsamiane z klasą[2]. Obecnie sytuacja nieco się skomplikowała, gdyż słowo *obiekt* otrzymało o wiele szersze znaczenie. Z tego względu wygodniej jest posługiwać się szeroko stosowanym w anglojęzycznej literaturze sformułowaniem *egzemplarz klasy* (ang. *class instance*). Otóż, po zdefiniowaniu klasy tworzymy obiekt jej typu o nazwie takiej samej, jak nazwa klasy występująca po słowie `class`. Jednak klasy tworzymy po to, by stały się specyfikatorami typów danych. Jeżeli w instrukcji definicji klasy po zamykającym nawiasie klamrowym podamy pewną nazwę, utworzymy tym samym określony egzemplarz klasy, który od tej chwili traktowany jest jako nowy, pełnoprawny typ danych. Oczywiście jedna klasa może być źródłem definicji wielu jej egzemplarzy. Innym sposobem utworzenia egzemplarza danej klasy będzie następująca konstrukcja:

```
...
class NazwaKlasy
{
  private:
    //prywatne dane i funkcje
  public:
    //publiczne dane i funkcje
  protected:
    //chronione dane i funkcje
};
....
int main()
...
EgzemplarzKlasy NazwaKlasy;
....
```

Ćwiczenie 2.7.

Jako przykład stworzymy bardzo prostą w swojej budowie klasę `Student`, przy pomocy której będziemy mogli odczytać wybrane informacje o pewnej osobie.

1. Deklaracja klasy `Student` składającej się z części publicznej i prywatnej może wyglądać następująco:

```
class Student
{
  public:
    void JakiStudent(char *);
    void Inicjalizacja();
    void Finalizacja();
  private:
    char *Nazwisko;
};
```

W sekcji publicznej (rozpoczynającej się od słowa kluczowego `public`) deklaracji klasy `Student` umieściliśmy prototypy trzech funkcji składowych klasy. W sekcji prywatnej (rozpoczynającej się od słowa kluczowego `private`) umieściliśmy deklarację

[2] Jako ciekawostkę podajmy, iż w latach 70. XX wieku słowo *obiekt* utożsamiano z pewnym wydzielonym obszarem pamięci w dużych maszynach cyfrowych oraz innych maszynach zwanych mini- i mikrokomputerami. W takim znaczeniu słowa *obiekt* używali Brian Kernighan i Dennis Ritchie — twórcy języka C.

jednej zmiennej (dokładniej — wskaźnika) składowej klasy. Mimo że istnieje możliwość deklarowania zmiennych publicznych w klasie, to jednak zalecane jest, aby ich deklaracje były umieszczane w sekcji prywatnej, zaś dostęp do nich był możliwy poprzez funkcje z sekcji publicznej. Jak już się zapewne domyślasz, dostęp do wskaźnika `Nazwisko` z sekcji prywatnej będzie możliwy właśnie poprzez funkcję `JakiStudent()`, której jedynym parametrem jest wskaźnik. Tego typu technika programowania nosi nazwę *enkapsulacji danych*, co oznacza, że dostęp do prywatnych danych jest zawsze ściśle kontrolowany.

2. Jak się zapewne domyślasz, rolą bezparametrowej funkcji `Inicjalizacja()` będzie zainicjowanie danych. Ponieważ omawiana funkcja jest częścią klasy `Student`, to przy jej zapisie w programie musimy poinformować kompilator, iż właśnie do niej należy. Operator rozróżniania zakresu `::` (ang. *scope resolution operator*) służy temu celowi.

```
void Student::Inicjalizacja()
{
 Nazwisko = new char[40];
 cout <<"Przydzielono pamięć'";
 cout << endl << "Naciśnij Enter" << endl;
 getchar();
 return;
}
```

Jedyną rolą funkcji `Inicjalizacja()` jest dynamiczne przydzielenie pamięci do wskaźnika `Nazwisko` przy pomocy operatora `new`. Tekst wyświetlany jest jedynie w celach poglądowych.

3. Funkcja `Finalizacja()` zajmuje się zwolnieniem przy pomocy operatora `delete` uprzednio przydzielonej pamięci.

```
void Student::Finalizacja()
{
 delete[] Nazwisko;
 cout <<"Zwolniono pamięć";
 cout << endl << "Naciśnij Enter" << endl;
 return;
}
```

4. W funkcji `JakiStudent()` dokonujemy porównania dwóch ciągów znaków. Przy pomocy funkcji `strcmp()` z modułu `string.h` porównujemy dwa łańcuchy znaków, czyli łańcuch wskazywany przez `Nazwisko` oraz drugi, odpowiadający już konkretnemu nazwisku danej osoby. Jeżeli oba łańcuchy są identyczne, funkcja `strcmp()` zwraca wartość `0`, zaś następnie przy pomocy strumienia wyjściowego `cout` wyświetlany jest odpowiedni komunikat.

```
void Student::JakiStudent(char *s)
{
  Nazwisko = s;
  if(strcmp(Nazwisko, "Wackowski")==0)
    strcpy(Nazwisko, "bardzo dobry student");
  if(strcmp(Nazwisko, "Jankowski")==0)
     strcpy(Nazwisko, "kiepski student");
  cout << Nazwisko << endl << endl;
  return;
}
```

5. W głównej funkcji `main()` wywołamy opisane wcześniej elementy klasy `Student`.

```
1: int main()
2: {
3:   char *KtoryStudent;
4:   KtoryStudent = new char[40];
5:   Student InfoStudent;
6:   clrscr();
7:   cout << "Podaj nazwisko: ";
8:   gets(KtoryStudent);
9:   InfoStudent.Inicjalizacja();
10:  InfoStudent.JakiStudent(KtoryStudent);
11:  InfoStudent.Finalizacja();
12:  getch();
13:  delete[] KtoryStudent;
14:  return 0;
15:}
```

I tak, w linii 3. deklarujemy wskaźnik `KtoryStudent` przechowujący wpisywane z klawiatury nazwisko (dokładniej — wskazujący na pierwszą literę nazwiska), zaś w linii 4. przydzielamy mu przy pomocy operatora `new` odpowiedni obszar pamięci. W linii 5. deklarujemy egzemplarz `InfoStudent` klasy `Student`. W liniach 6. – 8. wczytujemy nazwisko studenta. Ponieważ używamy wskaźnika, wygodniej jest w tym celu wykorzystać funkcję `gets()`, której prototyp znajduje się w pliku *stdio.h*. W liniach 10. i 11. w odpowiedniej kolejności wywołujemy funkcje składowe egzemplarza `InfoStudent` klasy `Student`. Aby wywołać funkcję składową egzemplarza klasy z fragmentu programu nie należącego do klasy (w naszym przypadku z głównej funkcji `main()`), należy w kolejności podać: nazwę egzemplarza klasy, operator w postaci kropki, zaś po nim nazwę właściwej funkcji. W linii 13. zwalniamy pamięć przydzieloną wskaźnikowi `KtoryStudent`.

Kompletny kod modułu *Unit05.cpp* projektu *Projekt_05.bpr* korzystającego z omawianej klasy przedstawiono na wydruku 2.5, zaś na rysunku 2.4 pokazano efekt naszej pracy.

Wydruk 2.5. *Kod modułu Unit05.cpp*

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
#include <stdio.h>
#pragma hdrstop
class Student
{
  private:
    char *Nazwisko;
  public:
    void JakiStudent(char *);
    void Inicjalizacja();
    void Finalizacja();
};// InfoStudent;
//---------------------------------------------------------
void Student::Inicjalizacja()
{
 Nazwisko = new char[40];
 cout <<"Przydzielono pamięć";
 cout << endl << "Naciśnij Enter" << endl;
```

```
 getchar();
 return;
}
//---------------------------------------------------------
void Student::Finalizacja()
{
 delete[] Nazwisko;
 cout <<"Zwolniono pamięć";
 cout << endl << "Naciśnij Enter" << endl;
 return;
}
//---------------------------------------------------------
void Student::JakiStudent(char *s)
{
  Nazwisko = s;
  if(strcmp(Nazwisko, "Wackowski")==0)
    strcpy(Nazwisko, "bardzo dobry student");
  if(strcmp(Nazwisko, "Jankowski")==0)
     strcpy(Nazwisko, "kiepski student");
  cout << Nazwisko << endl << endl;
  return;
}
//---------------------------------------------------------
int main()
{
  char *KtoryStudent;
  KtoryStudent = new char[40];
  Student InfoStudent;
  clrscr();
  cout << "Podaj nazwisko: ";
  gets(KtoryStudent);
  InfoStudent.Inicjalizacja();
  InfoStudent.JakiStudent(KtoryStudent);
  InfoStudent.Finalizacja();
  getch();
  delete[] KtoryStudent;
  return 0;
}
```

Rysunek 2.4.
Projekt Projekt_05.bpr w trakcie działania



Konstruktor i destruktor

Przedstawiony na przykładzie projektu *Projekt_05.bpr* sposób inicjowania i finalizowania działania różnych obiektów (w tym egzemplarzy klas) jest obecnie rzadko wykorzystywany w praktyce (poza zupełnie szczególnymi przypadkami, w których mamy jakieś konkretne wymagania odnośnie działania programu). Ponieważ konieczność inicjalizowania (i ew. finalizowania) np. egzemplarzy klas w C++ występuje bardzo często, dlatego w praktyce wykorzystujemy automatyczną ich inicjalizację (i ew. finalizację). Tego rodzaju automatyzacja możliwa jest dzięki wykorzystaniu specjalnych funkcji składowych klasy zwanych *konstruktorami*. Konstruktor i destruktor zawsze posiadają taką samą nazwę jak klasa, do której należą. Różnica w ich zapisie polega na tym, że nazwa destruktora poprzedzona jest znakiem ~.

Ćwiczenie 2.8.

Zmodyfikujemy poprzedni program w taki sposób, aby klasa `Student` posiadała swój konstruktor i destruktor.

1. Po wprowadzeniu konstruktora i destruktora do klasy `Student`, jej definicja przybierze następującą postać:

```
class Student
{
  public:
    void JakiStudent(char *);
    Student(); // konstruktor
    ~Student();// destruktor
  private:
    char *Nazwisko;
};
```

   Natychmiast zauważymy, iż w C++ nie określa się typów powrotnych konstruktorów i destruktorów, gdyż z założenia nie mogą zwracać żadnych wartości.

2. Zrozumienie idei używania konstruktorów i destruktorów ułatwi nam prześledzenie poniższego wydruku.

Wydruk 2.6. *Kod modułu Unit06.cpp projektu Projekt_06.bpr wykorzystującego funkcje składowe w postaci konstruktora i destruktora klasy*

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
#include <stdio.h>
#pragma hdrstop

class Student
{
  public:
    void JakiStudent(char *);
    Student(); // konstruktor
    ~Student();// destruktor
  private:
    char *Nazwisko;
};
```

```
//-----konstruktor klasy Student--------------------------
Student::Student()
{
 Nazwisko = new char[40];
 cout <<"Konstruktor klasy został zainicjowany.";
 cout << endl << "Naciśnij Enter" << endl;
 getchar();
}
//------------------------------------------------------
void Student::JakiStudent(char *s)
{
  Nazwisko = s;
  if(strcmp(Nazwisko, "Wackowski")==0)
    strcpy(Nazwisko, "bardzo dobry student");
  if(strcmp(Nazwisko, "Jankowski")==0)
     strcpy(Nazwisko, "kiepski student");
  cout << Nazwisko << endl;

  cout << endl<< "Naciśnij klawisz..." << endl;
  return;
}
//-------destruktor klasy Student-------------------------
Student::~Student()
{
  delete[] Nazwisko;
  cout <<"Konstruktor został zniszczony."
         " Powtórnie naciśnij klawisz..." << endl;
  getch();
}
//------------------------------------------------------
int main()
{
  char *KtoryStudent;
  KtoryStudent = new char[40];
  Student InfoStudent;
  clrscr();
  cout << "Podaj nazwisko: ";
  gets(KtoryStudent);
  InfoStudent.JakiStudent(KtoryStudent);
  getch();
  delete[] KtoryStudent;
  return 0;
}
```

Uruchamiając program bez trudu przekonamy się, iż konstruktor `Student()` jest automatycznie uruchamiany podczas tworzenia egzemplarza klasy, czyli w momencie wykonywania instrukcji deklarującej egzemplarz klasy. Jest to jedna z cech odróżniających język C++ od zwykłego C. W C++ deklaracje zmiennych wykonywane są w trakcie działania programu.

Ze względu na to, że każdy egzemplarz klasy jest tworzony w trakcie wykonywania instrukcji jego deklaracji, musi on być niszczony automatycznie w trakcie opuszczania bloku programu, w którym powyższa deklaracja się znajduje. Testując projekt *Projekt_06.bpr* bez trudu przekonamy się, iż mimo że destruktor `~Student()` nie został jawnie wywołany, to jednak kod jego zostanie wykonany, zwalniając tym samym pamięć przydzieloną uprzednio wskaźnikowi `Nazwisko` przez konstruktor `Student()`.

Ćwiczenie 2.9.

Samodzielnie przetestuj projekty *Projekt_05.bpr* oraz *Projekt_06.bpr* pod kątem określenia różnic w działaniu zwykłych funkcji składowych `Inicjalizacja()` i `Finalizacja()` oraz konstruktora `Student()` i destruktora `~Student()`.

Ćwiczenie 2.10.

Często, w celu uczynienia programu bardziej przejrzystym, definicje klas umieszczamy w oddzielnym pliku nagłówkowym. Postaraj się samodzielnie stworzyć taki plik i włączyć go do programu.

Inne spojrzenie na klasy. Własności

C++Builder, jako kompletne środowisko programowania, wprowadza bardziej ogólne pojęcie klasy. Otóż w Builderze definicję klasy można znacznie wzbogacić, tzn. w skład tej definicji oprócz sekcji *prywatnej* i *publicznej* może wchodzić również sekcja *publikowana* (ang. *published*) rozpoczynająca się od słowa kluczowego `__published`. W tej sekcji umieszcza się z reguły deklaracje funkcji, czyli deklaracje metod związane z komponentami pochodzącymi z biblioteki VCL. Dodatkowo klasa może zawierać też definicje *własności* rozpoczynające się od słowa `__property`. Warto w tym miejscu zauważyć, iż definicje w klasie związane z biblioteką VCL będą rozpoczynać się od pewnych słów kluczowych, poprzedzonych podwójnym znakiem podkreślenia. Chociaż do klas związanych z biblioteką komponentów wizualnych powrócimy jeszcze w dalszej części książki, to jednak w tym miejscu pokażemy, w jaki sposób można zbudować prostą własność w klasie.

Ćwiczenie 2.11.

Zbudujemy prostą klasę, której wykorzystanie w programie umożliwi w sposób bardzo elegancki i przejrzysty odczytywanie i zapisywanie danych w postaci nazwisk wybranych osób. W tym celu skorzystamy z definicji własności. Ponieważ ogólnie przyjętą konwencją jest to, aby w tego typu programach posługiwać się pewnymi standardowymi przedrostkami dla zmiennych oraz funkcji, w dalszej części opisu będziemy wykorzystywać nazewnictwo angielskie — po to, by nie tworzyć mieszanki nazw polskich i angielskich (np. `SetNazwisko`).

1. Na potrzeby naszego ćwiczenia samodzielnie zbudujemy własność, przy pomocy której będziemy w stanie odczytywać i przypisywać odpowiednie wartości (w tym przypadku łańcuchy znaków reprezentujące nazwiska i imiona osób). Każda własność służy do przechowywania wartości, zatem należy zadeklarować związaną z nią zmienną (tzw. pole w klasie). Ogólnie przyjętą konwencją jest to, że zmienne mają takie same nazwy, jak związane z nimi własności, ale poprzedzone są literą `F`. W naszym programie w sekcji prywatnej definicji klasy `Students` zadeklarujemy jedną taką zmienną typu `AnsiString`, reprezentującą tablicę indeksującą nazwiska studentów. Dodatkowo w tej samej sekcji zadeklarujemy dwie funkcje (metody): `GetName()`, która w przyszłości będzie odczytywała przy pomocy indeksu imię i nazwisko wybranej osoby oraz `SetName()`, za pomocą której będzie można przypisać odpowiedni łańcuch znaków (imię i nazwisko) do odpowiedniego indeksu w tablicy.

```
private:
   AnsiString FName[4];
   AnsiString GetName(int index);
   void SetName(int, AnsiString);
```

2. Przechodzimy teraz do deklaracji samej własności. W tym celu należy użyć słowa kluczowego `__property`. Dla naszych potrzeb wystarczy, by własność służyła wyłącznie do przekazywania danych (imion i nazwisk osób) przy pomocy indeksu. Własność zadeklarujemy w sekcji publicznej definicji klasy. Zdefiniowana przez nas własność `Name` będzie odczytywać aktualny stan tablicy `FName` przy pomocy dyrektywy `read`, a następnie przekazywać (zapisywać) ją do funkcji `SetName()` korzystając z dyrektywy `write`.

```
public:
   Students(){}   // konstruktor
   ~Students(){}  // destruktor
   __property AnsiString Name[int index] =
                              {read=GetName, write=SetName};
```

3. Jednoparametrowa funkcja (metoda) `GetName()` ma bardzo prostą budowę i służyć będzie do odpowiedniego indeksowania nazwisk:

```
AnsiString Students::GetName(int i)
{
  return FName[i];
};
```

4. Dwuparametrowa funkcja (metoda) `SetName()` również nie jest skomplikowana i przypisuje odpowiedniemu indeksowi tablicy ciąg znaków określony przez `names`.

```
void Students::SetName(int i, const AnsiString names)
{
   FName[i]=names;
}
```

Kompletny kod modułu *Unit_07.cpp* projektu *Projekt_07.bpr* wykorzystującego własność w klasie pokazany jest na wydruku 2.7.

Wydruk 2.7. *Moduł Unit07.cpp*

```
#include <vcl.h>
#include <stdio.h>
#include <conio.h>
#pragma hdrstop

class Students
{
 public:
   Students(){}   // konstruktor klasy
   ~Students(){}  // destruktor  klasy
   __property AnsiString Name[int index] =
                             {read=GetName, write=SetName};

 private:
   AnsiString FName[4];
   AnsiString GetName(int index);
   void SetName(int, AnsiString);
} Person; //egzemplarz Person (osoba) klasy Students
//---------------------------------------------------------
```

```
AnsiString Students::GetName(int i)
{
  return FName[i];
};
//---------------------------------------------------------
void Students::SetName(int i, const AnsiString names)
{
   FName[i]=names;
}
//---------------------------------------------------------
int main()
{
  clrscr();
  Person.Name[0]="Wacek Jankowski"; // wywołuje Person::SetName()
  Person.Name[1]="Janek Wackowski";
  Person.Name[2]="Jola Lobuzinska";

  for (int i = 0; i <= 2; i++)
    puts(Person.Name[i].c_str());  // wywołuje Person::GetName()

 getch();
 return 0;
}
```

W głównej funkcji programu, w celu wyświetlenia odpowiednich łańcuchów znaków, użyliśmy metody `c_str()` zwracającej wskaźnik (`char *`) do pierwszego znaku łańcucha identyfikującego własność tablicową `Name` egzemplarza `Person` klasy `Students`. Można też zauważyć, iż użycie w programie słowa `__property` oraz typu `AnsiString` (elementy te nie są zdefiniowane w standardowym C++) wymaga włączenia do kodu modułu pliku nagłówkowego *vcl.h* (patrz rysunek 2.1).

Funkcje przeładowywane

Stosowanie w programie funkcji przeładowywanych (często też nazywanych funkcjami przeciążanymi) w bardzo wielu wypadkach może znacznie ułatwić pisanie rozbudowanych programów. Używanie funkcji przeładowywanych nierozerwalnie wiąże się z tzw. *polimorfizmem* w C++. Polega on na zdefiniowaniu większej liczby funkcji posiadających taką samą nazwę, ale o różnych postaciach listy parametrów. Możliwość przeciążania funkcji jest w C++ czymś bardzo naturalnym i nie wymaga stosowania w ich deklaracjach specjalnych dyrektyw. Dla porównania przypomnijmy, że w Delphi funkcje czy procedury przeciążane zawsze należy deklarować z dyrektywą `overload`. Pod względem używania funkcji przeładowywanych C++ nie generuje żadnego nadmiarowego kodu w porównaniu z Object Pascalem.

Ćwiczenie 2.12.

Stworzymy program obliczający kolejne całkowite potęgi liczb różnych typów.

1. Program zawierać będzie dwie podobne (ale nie takie same) funkcje o nazwie `power()` (potęga), zwracające kolejne potęgi pobranego argumentu. Prototypy tych funkcji możemy zapisać w sposób następujący:

```
unsigned int power(unsigned int x, unsigned int y);
double power(double dx, unsigned long dy);
```

2. Następnie zapiszmy ciała tych funkcji:

```
unsigned int power(unsigned int x, unsigned int y)
{
  unsigned int i, z=1;
  for(i=1; i<=y; i++)
    z = z*x;
  return z;
}
//---------------------------------------------------------
double power(double dx, unsigned long dy)
{
  double z=1.0;
  for(unsigned int i=1; i<=dy; i++)
    z = z*dx;
  return z;
}
```

Jak widać, nazwa `power()` nie oznacza żadnej konkretnej funkcji, a jedynie ogólny rodzaj działania wykonywanego na konkretnym typie liczb.

3. Wybór wersji funkcji odpowiadającej określonym potrzebom programisty jest wykonywany automatyczne przez kompilator, dzięki podaniu typu parametrów aktualnych wywoływanej funkcji `power()`. Czynność tę wykonujemy wywołując odpowiednie funkcje w programie głównym.

```
int main()
{
  clrscr();
  for(unsigned int i=1; i<=10; i++)
    cout<< endl << "2^"<<i<<"="<< power(2, i);
  cout<< endl;
  for(unsigned int i=1; i<=10; i++)
    cout<< endl << "2.5^"<<i<<"="<< power(2.5, i+0.0);

  cout << endl << "Naciśnij klawisz...";
  getch();
  return 0;
}
```

Jak łatwo zauważyć, główną zaletą przeładowywania funkcji jest to, że bez problemu można wywoływać powiązane ze sobą zestawy instrukcji za pomocą tej samej nazwy, co pozwala, tak jak w pokazanym przypadku, zredukować dwie nazwy do jednej.

Kompletny kod źródłowy głównego modułu projektu *Projekt_08.bpr*, wykorzystującego przeładowywane funkcje, zamieszczono na wydruku 2.8.

Wydruk 2.8. *Moduł Unit08.cpp*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
//prototypy przeładowywanej funkcji power()
unsigned int power(unsigned int x, unsigned int y);
double power(double dx, unsigned long dy);
```

```
int main()
{
  clrscr();
  for(unsigned int i=1; i<=10; i++)
    cout<< endl << "2^"<<i<<"="<< power(2, i);
  cout<< endl;
  for(unsigned int i=1; i<=10; i++)
    cout<< endl << "2.5^"<<i<<"="<< power(2.5, i+0.0);

  cout << endl << "Naciśnij klawisz...";
  getch();
  return 0;
}
//-------------------------------------------------------
unsigned int power(unsigned int x, unsigned int y)
{
  unsigned int i, z=1;
  for(i=1; i<=y; i++)
    z = z*x;
  return z;
}
//-------------------------------------------------------
double power(double dx, unsigned long dy)
{
  double z=1.0;
  for(unsigned int i=1; i<=dy; i++)
    z = z*dx;
  return z;
}
```

Kończąc rozważania o funkcjach przeładowywanych należy zauważyć, iż teoretycznie możemy nadawać identyczne nazwy funkcjom wykonującym różne działania, np. mnożenie, potęgowanie i logarytmowanie. Jednak z praktycznego punktu widzenia nie powinniśmy postępować w ten sposób. Zawsze należy dążyć do tego, aby przeładowywać funkcje wykonujące te same operacje.

Ćwiczenie 2.13.

Postaraj się zmodyfikować *Projekt_08.bpr* w taki sposób, aby bazował na pewnej klasie wykorzystującej omawiane funkcje.

Niejednoznaczność

Polimorfizm, którego przykładem jest możliwość przeładowywania funkcji, stanowi jedną z wielkich zalet obiektowego języka C++, jednak również niesie ze sobą (szczególnie dla mniej doświadczonych programistów) pewne pułapki. Jedną z nich jest *niejednoznaczność*. Głównym źródłem niejednoznaczności w C++ są automatyczne konwersje typów. Uważny Czytelnik na pewno spostrzegł, w jaki sposób w programie przedstawionym na wydruku 2.8 wywoływane są funkcje należące do swoich prototypów:

```
unsigned int power(unsigned int x, unsigned int y); // prototyp
for(unsigned int i=1; i<=10; i++)
    cout<< endl << "2^"<<i<<"="<< power(2, i);
                                // wywołanie w programie
```

oraz:

```
double power(double dx,  unsigned long dy); // prototyp
for(unsigned int i=1; i<=10; i++)
    cout<< endl << "2.5^"<<i<<"="<< power(2.5, i+0.0);
                                // wywołanie w programie
```

Funkcja `power()` została przeładowana w taki sposób, że może pobierać argumenty w postaci par liczb typu `unsigned int`, `unsigned int` oraz `double`, `unsigned long`. Pierwsza instrukcja wywołania funkcji `power()` będzie jednoznaczna, gdyż nie musi występować żadna konwersja danych pomiędzy jej parametrami formalnymi i aktualnymi. Rozpatrzmy przypadek, gdy funkcja `power()` wywoływana jest z parametrem typu całkowitego `unsigned int`, będącego typem zmiennej sterującej `i` w pętli `for()`, zaś jednym z jej parametrów formalnych (zdeklarowanym w jej prototypie) jest zmienna typu `unsigned long`. W takiej sytuacji kompilator „nie wie", czy zmienną taką przekształcić na typ `long int`, czy `long double`. Jeżeli wywołalibyśmy funkcję w programie w sposób następujący:

```
for(unsigned int i=1; i<=10; i++)
    cout<< endl << "2.5^"<<i<<"="<< power(2.5, i);
                                // błędne wywołanie w programie
```

możemy spodziewać się wystąpienia przykrego komunikatu kompilatora:

```
[C++ Error] Unit08.cpp(15): E2015 Ambiguity between 'power(unsigned int,unsigned int)'
and 'power(double,unsigned long)'
```

Aby uniknąć tego typu dwuznaczności, często uciekamy się do pewnego fortelu. Mianowicie wystarczy do liczby deklarowanej jako całkowita dodać wartość `0.0`, aby kompilator dokonał jej automatycznej konwersji na typ zmiennopozycyjny.

Funkcje ogólne

Funkcjami ogólnymi posługujemy się w sytuacjach, w których wymagamy, aby definicja danej funkcji zawierała całość operacji wykonywanych na danych różnych typów. Funkcję ogólną tworzymy przy pomocy słowa kluczowego `template` (szablon). Jego intuicyjne znaczenie jest bardzo trafne — szablon służy do ogólnego opisu działań wykonywanych przez daną funkcję, zaś dalszymi szczegółami powinien zająć się już sam kompilator C++. Szkielet definicji funkcji ogólnej rozpoczyna się właśnie od słowa `template`:

```
template <class TypDanych> TypZwracany NazwaFunkcji(lista parametrów)
{
  // instrukcje
};
```

Ćwiczenie 2.14.

Jak zapewne wiesz, jedną z właściwości języków C i C++ jest to, iż wszystkie argumenty funkcji przekazywane są przez wartość. Wynika z tego, że wywoływana funkcja otrzymuje wartości swoich argumentów, a nie ich adresy. Można oczywiście spowodować, aby funkcja zmieniała wartości zmiennych w funkcji wywołującej. W tym celu funkcja wywołująca musi przekazać adresy swoich zmiennych. Zbudujemy prostą funkcję ogólną, która zmieniać będzie wartości dwóch zmiennych przekazywanych jej w instrukcji wywołania. Funkcja będzie porównywać dwie zmienne i jako wartość powrotną zwracać większą z liczb.

1. Zadeklarujemy prostą funkcję ogólną o nazwie `max()`.

```
template <class T> T max(T &x, T &y)
{
  return (x > y) ? x : y;
};
```

   Litera `T` oznacza tutaj nazwę zastępczą typu danych wykorzystywanych przez dwuparametrową funkcją `max()`. Zaletą podawania nazwy zastępczej jest to, że kompilator zawsze automatycznie zastąpi ją rzeczywistym typem danych w trakcie tworzenia konkretnej wersji funkcji. W funkcji tej wykorzystaliśmy operator warunkowy `?:` (pytajnik i dwukropek). Znaczenie jego jest następujące:

```
E1 ? E2 : E3
```

   Jeżeli wartość logiczna wyrażenia (lub warunku) `E1` występującego po lewej stronie znaku `?` jest prawdą (wartość różna od zera), wówczas funkcja zwróci wartość wyrażenia `E2` (w naszym przypadku `x`), w przeciwnym razie wartością powrotną funkcji będzie wartość wyrażenia `E3` występującego po znaku `:` (w naszym przypadku `y`).

2. Wywołanie funkcji ogólnej `max()` w programie głównym nie powinno sprawić nam najmniejszych trudności. Na wydruku 2.9 pokazano kompletny kod źródłowy modułu *Unit09.cpp* projektu *Projekt_09.bpr*.

Wydruk 2.9. *Moduł Unit09.cpp*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
template <class T> T max(T &x, T &y)
{
  return (x > y) ? x : y;
};
//---------------------------------------------------------
int main()
{
  double x=12222.333365, y=14444.4467677;
  cout.setf(ios::fixed);
  cout << max(x, y) << endl;
  getch();
  return 0;
}
//---------------------------------------------------------
```

W programie występuje dodatkowy szczegół, na który warto zwrócić uwagę. Mianowicie sposób formatowania liczb. W celu wyświetlenia wartości z większą liczbą cyfr po kropce, oddzielającej część całkowitą od części ułamkowej, wykorzystaliśmy metodę `setf` strumienia `cout` oraz metodę `fixed` (zwaną tutaj *manipulatorem*) klasy `ios`. Z innymi sposobami przedstawiania liczb z użyciem funkcji składowych klasy `ios` możemy zapoznać się studiując pliki pomocy.

Ćwiczenie 2.15.

Programiści C przyzwyczajeni są do używania makrodefinicji. Definicję funkcji ogólnej `max()` możemy zastąpić następującą makrodefinicją:

```
#define max(x, y) ((x > y) ? x : y)
```

Warto jednak zwrócić uwagę, iż makrodefinicje nie mogą zmieniać wartości swoich parametrów. Z tego względu pokazany sposób tworzenia makrodefinicji `max()` uważa się obecnie za przestarzały. Programy tworzone w „czystym" C++ powinny (o ile jest to konieczne) używać funkcji ogólnych, umożliwiających operowanie na różnych typach parametrów. Przetestuj samodzielnie projekt *Projekt_09.bpr* wykorzystując opisaną makrodefinicję.

Przeładowywanie funkcji ogólnych

Z potrzebą przeładowywania funkcji ogólnych możemy spotkać się w sytuacji, w której istnieje konieczność zbudowania funkcji przeładowującej funkcję ogólną do określonego zestawu typów parametrów formalnych.

Ćwiczenie 2.16.

Jako prostą ilustrację metody przeładowania omawianej wcześniej funkcji ogólnej `max()` niech nam posłuży przykład, w którym zażądamy, aby wartością powrotną funkcji przeładowującej była mniejsza z dwu liczb całkowitych będących jej argumentami.

1. Jawne przeładowanie funkcji ogólnej `max()` może przybrać następującą postać:

```
template <class T> T max(T &x, T &y)
{
  return (x > y) ? x : y;
};
//----przeładowana wersja funkcji ogólnej max()------------
max(int &x, int &y)
{
  return (x < y) ? x : y;
}
```

Cechą charakterystyczną przeładowywanych funkcji ogólnych jest to, że wszystkie ich wersje muszą wykonywać identyczne działania, zaś różnić się mogą jedynie typami danych. Mogłoby się wydawać, że funkcja ogólna `max()` oraz jej wersja przeładowana wykonują różne działania, jednak z numerycznego punktu widzenia tak nie jest. Wynika to z faktu, iż zarówno obliczanie większej, jak i mniejszej wartości

dwu wprowadzonych liczb jest z numerycznego punktu widzenia czynnością identyczną, gdyż tak naprawdę wykonujemy jedynie operację porównania dwóch liczb, zaś jej „kierunek" nie ma najmniejszego znaczenia.

2. Funkcję ogólną oraz jej wersję przeładowaną wywołamy w programie tak, jak pokazano to na wydruku 2.10, gdzie wyraźnie zaznaczono z jakimi parametrami są one używane.

Wydruk 2.10. *Kod modułu Unit_10.cpp projektu Projekt_10.bpr*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
template <class T> T max(T &x, T &y)
{
  return (x > y) ? x : y;
};
//----przeładowana wersja funkcji ogólnej max()------------
max(int &x, int &y)
{
  return (x < y) ? x : y;
}

//---------------------------------------------------------
int main()
{
  double x=12222.333365, y=14444.4467677;
  int i=-102, j=105;
  char a='a', z='z';
  cout.setf(ios::fixed);
  // Wywołanie funkcji ogólnej max() z parametrami typu double
  cout << max(x, y) << endl;
  // Jawne wywołanie przeładowanej funkcji max() z parametrami
  // typu int. Mimo, iż funkcja posiada taką samą nazwę, to jednak
  // zwraca mniejszy liczbowo parametr aktualny
  cout << max(i, j) << endl;
  // Wywołanie funkcji ogólnej max() z parametrami typu char
  cout << max(a, z) << endl;
  getch();
  return 0;
}
```

Śledząc kod modułu bez trudu zauważymy, iż w czasie kompilacji nie jest generowana wersja funkcji ogólnej `max()` posiadająca parametry typu całkowitego `int` i zwracająca mniejszy liczbowo parametr aktualny. Wynika to z bardzo prostego faktu, mianowicie w programie została ona przedefiniowana przez zwykłą funkcję `max(int &x, int &y)`.

Typ wyliczeniowy

Typ wyliczeniowy pozwala programiście na zdefiniowanie nowego typu danych, gdzie każdemu elementowi zbioru zostanie przyporządkowana liczba odpowiadająca jego pozycji w zbiorze.

Typy wyliczeniowe definiuje się bardzo prosto. Po zarezerwowanym słowie `enum` podajemy nazwę nowego typu danych i znak równości, po którym następuje lista nazw nowego typu:

```
enum type {Fiat, Audi, Opel};
```

Elementy tak zdefiniowanego typu są uporządkowane zgodnie z kolejnością ich wyliczania.

Mamy też możliwość jawnego przypisania wartości poszczególnym identyfikatorom zadeklarowanego typu:

```
int main()
{
  enum type {Fiat=5, Audi=3, Opel=10, Suma = Fiat+Audi+Opel};
  cout <<"Liczba aut w salonie = " << Suma;
  getch();
  return 0;
}
```

Dziedziczenie

Dziedziczenie jest jednym z najważniejszych mechanizmów programowania zorientowanego obiektowo. Pozwala na przekazywanie właściwości klasy bazowej (ang. *base class*) klasom pochodnym (ang. *derived classes*). Oznacza to, że w prosty sposób można zbudować pewną hierarchię klas, uporządkowaną od najbardziej ogólnej do najbardziej szczegółowej. Na takiej właśnie hierarchii klas zbudowana jest w C++Builderze 6 zarówno biblioteka komponentów wizualnych VCL, jak i biblioteka międzyplatformowa CLX.

Ogólną postać definicji *klasy pochodnej* zapisujemy z reguły w sposób następujący:

```
class NazwaNowejKlasy: specyfikator_dostępu NazwaKlasyDziedziczonej
{
 // deklaracje sekcji w nowej klasie
} NazwaEgzemplarzaNowejKlasy; //opcjonalnie
```

Ćwiczenie 2.17.

1. Jako przykład zdefiniujemy klasę bazową o nazwie `Vehicle` (pojazd). Jak wiadomo, przeznaczenie danego pojazdu można szybko odgadnąć patrząc m.in. na liczbę i rodzaj jego kół. Zatem omawiana klasa zawierać będzie zmienną prywatną `FWeehls` określającą liczbę kół w pojeździe. Publiczna właściwość `Wheels` (koła) służy do odczytu i zapisu liczby kół.

```
class Vehicle
{
 public:
   __property int Wheels = {read=GetWheels, write=SetWheels};

 private:
   int FWheels;  // liczba kół
   int GetWheels();
   void SetWheels(int num);
};
```

2. Następnie wykorzystamy powyższą klasę bazową do definicji klasy reprezentującej pojazdy popularnie zwane „tirami". Tiry będziemy rozróżniać pod względem ich ładowności związanej z własnością `Cargo` (ładowność).

```
 class Tir: public Vehicle
{
 public:
   __property int Cargo = {read=GetCargo, write=SetCargo};
 private:
   int FCargo; //ładowność
   int GetCargo();
   void SetCargo(int num);
};
```

3. Z kolei prywatna część definicji klasy `Tir` posłuży nam do określenia klasy reprezentującej już określone marki ciężarówek `Lorry`.

```
enum type {Steyer, MAN, Volvo};
class Lorry: private Tir
{
 public:
   __property enum type LorryType = {read=GetLorryType,
                                     write=SetLorryType};
   void Show();
   enum type GetLorryType();
   void SetLorryType(enum type V);
  private:
   enum type FLorryType; // marka pojazdu
} L; //egzemplarz L klasy Lorry dziedziczącej po klasie Tir
```

W klasie tej własność `LorryType` musi być typu wyliczeniowego `enum type`, jeżeli pojazdy te zechcemy w przyszłości rozróżniać podając nazwę ich marek.

4. Specyfikator dostępu w klasie `Tir` został użyty ze słowem `public`. Oznacza to, iż wszystkie publiczne elementy klasy dziedziczonej są elementami publicznymi w klasie pochodnej. W naszym przykładzie funkcje z klasy `Tir` będą miały bezpośredni dostęp do funkcji składowych klasy `Vehicle`. Jednak funkcje klasy `Tir` nie będą miały dostępu do zmiennej `Wheels` z klasy `Vehicle`, gdyż z oczywistych względów nie ma takiej potrzeby.

5. Specyfikator dostępu w klasie `Lorry` został użyty ze słowem `private`. Oznacza to, iż wszystkie prywatne elementy klasy dziedziczonej są elementami prywatnymi w klasie pochodnej. W naszym przykładzie funkcje z klasy `Lorry` będą miały bezpośredni dostęp do zmiennej `Cargo` klasy `Tir`. Postąpiliśmy w ten sposób, aby bez problemu w funkcji składowej `Show()` (pokaż) klasy `Lorry` odwołać się bezpośrednio do zmiennej `Wheels` z klasy `Vehicle` oraz `Cargo` klasy `Tir`. W ten oto sposób wszystkie interesujące nas informacje uzyskamy wywołując w głównym programie jedynie funkcje składowe egzemplarza `L` klasy `Lorry`, tak jak pokazano to na wydruku 2.11. Chociaż program nasz składa się z trzech różnych klas, zawierających szereg odrębnych funkcji składowych, to jedynymi obiektami, do których jawnie odwołujemy się w głównej funkcji `main()`, są funkcje składowe `Show()` oraz `SetLorryType()` egzemplarza `L` klasy `Lorry`.

Wydruk 2.11. *Kod modułu Unit_11.cpp projektu Projekt_11.bpr*

```
#include <vcl.h>
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#pragma hdrstop

class Vehicle
{
 public:
   __property int Wheels = {read=GetWheels, write=SetWheels};

 private:
   int FWheels;  // liczba kół
   int GetWheels();
   void SetWheels(int num);
};
//---------------------------------------------------------
class Tir: public Vehicle
{
 public:
   __property int Cargo = {read=GetCargo, write=SetCargo};
 private:
   int FCargo; //ładowność
   int GetCargo();
   void SetCargo(int num);
};
//---------------------------------------------------------
enum type {Steyer, MAN, Volvo};
class Lorry: private Tir
{
 public:
   __property enum type LorryType = {read=GetLorryType,
                                     write=SetLorryType};
   void Show();
   enum type GetLorryType();
   void SetLorryType(enum type V);
  private:
   enum type FLorryType; // marka pojazdu
} L; //egzemplarz L klasy Lorry dziedziczącej po klasie Tir
//---------------------------------------------------------
int Vehicle::GetWheels()
{
  return FWheels;
};
//---------------------------------------------------------
void Vehicle::SetWheels(int num)
{
   FWheels=num;
   return;
}
//---------------------------------------------------------
int Tir::GetCargo()
{
   return FCargo;
};
//---------------------------------------------------------
```

```
void Tir::SetCargo(int num)
{
   FCargo=num;
   return;
}
//---------------------------------------------------------
enum type Lorry::GetLorryType()
{
  return FLorryType;
}
//---------------------------------------------------------
void Lorry::SetLorryType(enum type V)
{
  FLorryType=V;
}
//---------------------------------------------------------
void Lorry::Show()
{
  switch(GetLorryType()){
    case MAN:    {
         cout << " Ciężarówka marki MAN"<< endl;
         Wheels=6;
         cout << " Liczba kół: "<< Wheels << endl;
         Cargo = 20;
         cout << " Ładowność: "<< Cargo << "ton"<< endl;
         break;
    }
    case Steyer: {
         cout << " Ciężarówka marki Steyer"<< endl;
         Wheels=10;
         cout << " Liczba kół: "<< Wheels << endl;
         Cargo = 40;
         cout << " Ładowność: "<< Cargo << "ton"<< endl;
         break;
    }
    case Volvo: {
         cout << " Ciężarówka marki Volvo"<< endl;
         Wheels=8;
         cout << " Liczba kół: "<< Wheels << endl;
         Cargo = 35;
         cout << " Ładowność: "<< Cargo << "ton"<< endl;
         break;
    }
  }
  return;
}
//---------------------------------------------------------
int main()
{
  clrscr();
  L.SetLorryType(Steyer);
  L.Show();
  getch();
  return 0;
}
```

Analizując treść przedstawionego programu, natychmiast zauważymy, iż główną zaletą stosowania zasady dziedziczenia klas jest możliwość utworzenia jednej klasy bazowej wykorzystywanej następnie w definicjach klas pochodnych. Pokazaliśmy więc, w jaki sposób zdefiniować kilka bardzo podobnych klas uporządkowanych według określonej hierarchii.

Funkcje wewnętrzne

Jedną z zalet języka C++ jest możliwość definiowania *funkcji wewnętrznych*, często też nazywanych po prostu *funkcjami inline* (ang. *inline functions*). Definicja funkcji wewnętrznej rozpoczyna się od słowa kluczowego `inline`:

```
inline TypPowrotny DeklaracjaFunkcji(ListaParametrów)
```

Cechą charakterystyczną funkcji wewnętrznej jest to, iż nie jest ona bezpośrednio wywoływana w programie, natomiast jej kod umieszczany jest w miejscu, w którym ma być wykonany. Jeżeli funkcje wywołujemy w programie w sposób tradycyjny, zawsze należy liczyć się z tym, iż czynność ta będzie wymagała wykonania szeregu różnych instrukcji, takich jak umieszczenie jej argumentów w rejestrach procesora (lub na stosie), czy chociażby określenie wartości zwracanej. Inaczej cała sprawa przedstawia się w przypadku, gdy używamy funkcji wewnętrznych. Otóż ich kod umieszczany jest bezpośrednio w funkcji wywołującej, przez co czynności takie jak odpowiednie umieszczenie w pamięci jej argumentów i określenie wartości powrotnej nie są wykonywane.

Ćwiczenie 2.18.

Jako prosty przykład wykorzystania funkcji wewnętrznych niech nam posłuży omawiane już zagadnienie określania liczby kół w pojeździe.

Zmodyfikujemy fragment kodu programu przedstawionego na wydruku 2.11 w ten sposób, aby w klasie `Vehicle` zdefiniowane zostały dwie funkcje wewnętrzne: `GetWheels()` oraz `SetWheels()`, tak jak pokazano na wydruku 2.12.

Wydruk 2.12. *Kod modułu Unit_12.cpp projektu Projekt_12.bpr*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

class Vehicle
{
 public:
   int GetWheels();
   void SetWheels(int numWheels);
 private:
   int Wheels;  // liczba kół
}V;
//---------------------------------------------------------
inline int Vehicle::GetWheels()
```

```
{
  return Wheels;
};
//---------------------------------------------------------
inline void Vehicle::SetWheels(int numWheels)
{
  Wheels=numWheels;
}
//---------------------------------------------------------
int main()
{
  V.SetWheels(12);
  cout << V.GetWheels();
  getch();
  return 0;
}
```

Natychmiast zauważymy, iż w pewnych sytuacjach definicję własności w klasie możemy zastąpić funkcjami ogólnymi. Należy jednak pamiętać, iż tego typu zabiegi stosujemy głównie do funkcji o niewielkim kodzie. Ponadto, korzystając z funkcji wewnętrznych należy liczyć się z możliwością błędnego ich obsłużenia przez kompilator, gdy zechcemy skorzystać np. z klas wyjątków.

Ćwiczenie 2.19.

Definiując funkcje wewnętrzne bezpośrednio w deklaracji klasy możemy pominąć słowo `inline`. Jednak w tym wypadku należy jawnie w odpowiednim miejscu i w odpowiedni sposób wpisać ich kod źródłowy. Wynika to z faktu, iż każda funkcja zdefiniowana w obrębie definicji klasy (jeżeli jest to oczywiście możliwe) traktowana jest jako funkcja wewnętrzna.

```
class Vehicle
{
 public:
   int GetWheels() {return Wheels;}
   void SetWheels(int numWheels) {Wheels=numWheels;}
 private:
   int Wheels;  // liczba kół
}V;
```

Przetestuj *Projekt_12.bpr* z tak określonymi funkcjami wewnętrznymi. Zwróć szczególną uwagę na sposób umieszczenia znaków ; (średnik), oznaczających koniec wykonywanych instrukcji.

Realizacja przekazywania egzemplarzy klas funkcjom

Egzemplarze klas (lub klasy) przekazujemy funkcjom w sposób standardowy, czyli przez wartość. Pociąga to za sobą pewną dość poważną konsekwencję, mianowicie w trakcie przekazywania egzemplarza klasy do funkcji tworzona jest jego kopia, zaś w pamięci komputera powstaje zupełnie nowy obiekt.

Ćwiczenie 2.20.

Wykorzystamy zmodyfikowany *Projekt_07.bpr* w celu zilustrowania ciekawej właściwości, jaką możemy zaobserwować w momencie, gdy egzemplarz danej klasy staje się parametrem aktualnym wywoływanej funkcji. W tym celu przedefiniujemy nieco znaną już nam klasę `Students` oraz zaprojektujemy dwie nowe funkcje `Show1()` i `Show2()`, których parametrami formalnymi będą egzemplarze wspomnianej klasy. Jedynym zadaniem naszych funkcji będzie pobieranie i wyświetlanie na ekranie imion wybranych osób. Na wydruku 2.13 pokazano kompletny kod źródłowy modułu *Unit_13.cpp*, zaś na rysunku 2.5 wynik działania programu.

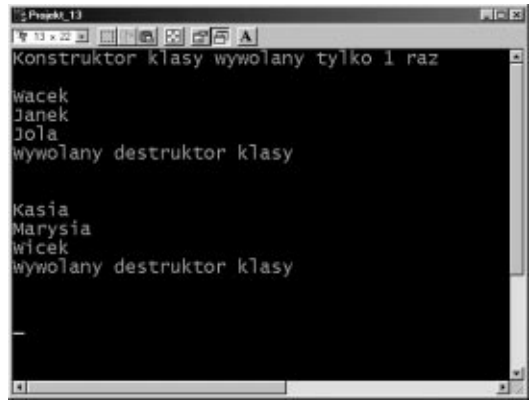
Wydruk 2.13. *Kod modułu Unit13.cpp projektu Projekt_13.bpr*

```
#include <vcl.h>
#include <conio.h>
#include <stdio.h>
#pragma hdrstop

class Students
{
 public:
    Students(){puts("Konstruktor klasy wywołany tylko 1 raz\n");}
   // konstruktor klasy
   ~Students(){puts("Wywołany destruktor klasy\n\n") ;}
   // destruktor  klasy

   AnsiString GetName(int i){return Name[i];}
   void SetName(int i, const AnsiString names){Name[i]=names;}
   AnsiString Name[4];
} Person; //egzemplarz Person (osoba) klasy Students
//---------------------------------------------------------
void Show1(Students ClassInstance); // prototyp funkcji show1()
void Show2(Students ClassInstance); // prototyp funkcji show2()
//---------------------------------------------------------
int main()
{
  Show1(Person);
  Show2(Person);
  getch();
  return 0;
}
//---------------------------------------------------------
void Show1(Students ClassInstance)
{
  ClassInstance.Name[0]="Wacek";
  ClassInstance.Name[1]="Janek";
  ClassInstance.Name[2]="Jola";
  for (int i = 0; i <= 2; i++)
    puts(ClassInstance.Name[i].c_str());
  return;
}
//---------------------------------------------------------
void Show2(Students ClassInstance)
{
  ClassInstance.Name[0]="Kasia";
  ClassInstance.Name[1]="Marysia";
```

```
  ClassInstance.Name[2]="Wicek";
  for (int i = 0; i <= 2; i++)
    puts(ClassInstance.Name[i].c_str());
  return;
}
```



Rysunek 2.5.
Projekt_13.bpr w trakcie działania

Testując przedstawiony program natychmiast zauważymy, iż podczas przekazywania egzemplarza klasy do funkcji nie jest wywoływany konstruktor klasy. To, co zobaczymy na ekranie, należy interpretować w ten sposób, że konstruktor wywoływany jest tylko raz, na początku programu, podczas inicjowania klasy. Konstruktor wywoływany jest tylko raz, natomiast destruktor dwukrotnie, podczas każdorazowego niszczenia kopii egzemplarza klasy przekazywanej funkcjom `Show1()` i `Show2()`. Możemy obrazowo powiedzieć, że tworzenie kopii egzemplarza klasy jest konsekwencją tego, iż jest on przekazywany funkcji, co absolutnie nie skutkuje wywołaniem jego konstruktora. Jednak kończenie działania funkcji musi być związane z jego zniszczeniem, co w konsekwencji pociąga za sobą wywołanie jego destruktora. Na marginesie dodajmy, iż tak naprawdę destruktor wywoływany jest jeszcze po raz trzeci, ale jest to już związane z określeniem wartości powrotnej głównej funkcji `main()`.

Tablice dynamicznie alokowane w pamięci

Jak wiemy, tablice służą do zapamiętywania danych tego samego typu. Deklarując tablicę informujemy nasz komputer o potrzebie przydzielenia odpowiedniej ilości pamięci oraz o kolejności rozmieszczenia elementów tablicy. W niniejszym podrozdziale opiszemy jedną z metod dynamicznego przydzielania pamięci tablicom jednowymiarowym (popularnie zwane wektorami) oraz dwuwymiarowym (zwane macierzami). W C++ do dynamicznego przydzielania i zwalniania pamięci służą operatory `new` i `delete`, których używa się następująco:

```
ZmiennaWskaźnikowa = new TypZmiennej;
delete ZmiennaWskaźnikowa;
```

Należy zauważyć, iż operator `new` przydziela tyle pamięci, ile potrzeba do zapisania wartości danego typu, oraz zwraca adres tej wartości. Ponadto do jednej z wielkich zalet operatora `new` należy zaliczyć to, iż automatycznie oblicza on rozmiar typu danych oraz zapobiega możliwości przydzielenia nieodpowiedniej ilości pamięci.

Korzystając z pary operatorów `new` i `delete` możemy też bez problemu przydzielać pamięć tablicom oraz odpowiednio ją zwalniać. W przypadku tablic jednowymiarowych stosujemy bardzo prostą instrukcję:

```
ZmiennaWskaźnikowa = new TypZmiennej[rozmiar];
delete[] ZmiennaWskaźnikowa;
```

W przypadku tablic dwuwymiarowych sytuacja nieco się komplikuje. Wynika to z faktu, iż musimy zainicjować zarówno wiersze tablicy, jak i jej kolumny. Dla przykładu rozpatrzmy dwuwymiarową tablicę `A[m][n]`, gdzie `m` jest liczbą wierszy, zaś `n` liczbą kolumn:

$$A = \begin{pmatrix} a_{11} & a_{12} \ldots a_{1n} \\ a_{21} & a_{22} \ldots \\ \ldots & \\ a_{m1} & a_{m2} \ldots a_{mn} \end{pmatrix}$$

Wówczas należy przydzielić pamięć najpierw dla określonej liczby wierszy, następnie zaś dla każdej z kolumn. Musimy też pamiętać, iż podczas deklaracji zmiennej wskaźnikowej reprezentującej macierz powinniśmy dwukrotnie użyć operatora wyłuskiwania *. Wynika to z faktu, iż należy poinformować kompilator, że będziemy dynamicznie przydzielać pamięć tworowi (mówiąc językiem matematyki) dwuwymiarowemu.

```
long double **A; // deklaracja macierzy kwadratowej A
...
A = new long double*[m];
for (int j = 1; j < m; j++)
   A[j] = new long double[n];
```

Zwolnienie tak przydzielonej pamięci odbywa się już w sposób znacznie prostszy:

```
for (int j = 1; j < m;  j++)
   delete[] A[j];
delete[] A;
```

W C++ poszczególne elementy tablicy ponumerowane są za pomocą indeksu rozpoczynającego się od wartości 0, jednak istnieją sytuacje, w których wygodniej jest przeindeksować je tak, aby rozpoczynały się od liczby 1, gdyż wówczas łatwiej koduje się wiele nie tylko algebraicznych zależności.

Ćwiczenie 2.21.

Jako przykład zrealizujmy prosty algorytm wykonujący mnożenie macierzy `A[m][n]` przez wektor `B[n]` dające w wyniku wektor `C[m]`. Przy wykonywaniu tego rodzaju działań

algebraicznych musimy pamiętać, iż liczba wierszy macierzy musi być równa liczbie elementów wektora (lub w przypadku mnożenia dwóch macierzy — liczba wierszy macierzy pierwszej musi być równa liczbie kolumn drugiej macierzy).

Wydruk 2.14. *Kod modułu Unit14.cpp projektu Projekt_14.bpr realizującego działania na tablicach, dla których pamięć została przydzielona dynamicznie*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

void wynik(long double **, long double*, long double *);
void zwolnij_pamiec(long double **, long double* ,
                    long double *);

int m = 3;
// liczba wierszy (w przypadku przeindeksowania zawsze +1)
int n = 3;
// liczba kolumn  (w przypadku przeindeksowania zawsze +1)

int main()
{
  long double **A; // macierz kwadratowa A
  long double *B;  // wektor B
  long double *C; // wektor wynikowy C

//- inicjowanie pamięci przydzielanej dynamicznie-wektorom-
  C = new long double[n];
  B = new long double[n];
//inicjowanie pamięci przydzielanej dynamicznie tablicy 2D-
  A = new long double*[m];
  for (int j = 1; j < m; j++)
     A[j] = new long double[n];
//-------------------------------------------------------

  for (int i = 1; i < m; i++)
    for (int j = 1; j < n; j++)
        A[i][j] = i+j-1;  //elementy macierzy kwadratowej (2x2)

  cout << "macierz A\n";
  for (int i = 1; i < m; i++)
    {
      for (int j = 1; j < n; j++)
        cout << A[i][j] << " ";
        cout << "\n" << endl;
    }

  cout << "wektor B\n";
  for (int j = 1; j < n; j++)
    {
     B[j]=j;  // dwuelementowy wektor
     cout << B[j] << " ";
     cout << "\n" << endl;
    }
  wynik(A, B, C);
  zwolnij_pamiec(A, B, C);
  getch();
  return 0;

}
```

```
//------Obliczanie wyniku A*B=C-------------------------
void wynik(long double **A, long double *B, long double *C)
{
 int sum;
 cout << "wektor C\n";
 for (int i = 1; i < m; i++)
   {
     sum=0;
     for (int j = 1; j < n; j++)
       {
         sum += A[i][j]*B[j];
         C[i]=sum; // dwuelementowy wektor wynikowy
       }
     cout << C[i] << " \n\n";
   }
  return;
}
//--------- Zwolnienie przydzielonej pamięci---------------
void zwolnij_pamiec(long double **A, long double *B,
                    long double *C)
{
  for (int j = 1; j < m;  j++)
     delete[] A[j];

  delete[] A;
  delete[] B;
  delete[] C;
  return;
}
```

Podsumowując, zwróćmy uwagę na jeszcze jeden ważny szczegół. Otóż, zawsze posługujemy się parą operatorów `new` i `delete`. Pamiętajmy, iż `delete` można używać jedynie ze wskaźnikami do obszarów pamięci, które zostały uprzednio przydzielone za pomocą operatora `new`. Używając `delete` z innym adresem bardzo łatwo zakończymy działanie aplikacji w sposób nie kontrolowany i jest to najmniej przykra niespodzianka, jaka może nas spotkać.

Tablice otwarte

Język C++ znacznie rozszerza pojęcie tablicy. Jednym z takich rozszerzeń są tablice otwarte, które występują w roli argumentów funkcji i mogą posiadać dowolne rozmiary. Wywołując funkcję, można przekazać jej jako argument tablicę dowolnego rozmiaru i o tym samym typie bazowym. W module *math.hpp* zdefiniowanych jest wiele funkcji posiadających argumenty w postaci tablic otwartych. Rozpatrzmy prototyp jednej z nich, obliczającej średnią arytmetyczną z elementów tablicy otwartej `Data` będącej jednym z jej argumentów:

```
extern PACKAGE Extended __fastcall Mean(const double * Data,
                                        const int Data_Size);
```

`Data_Size` określa rozmiar tablicy, czyli liczbę jej elementów pomniejszoną o jeden (pamiętamy, że elementy tablic w C++ indeksowane są począwszy od wartości 0). Po zadeklarowaniu jednowymiarowej tablicy, zawierającej wybrane elementy, dla których chcemy obliczyć średnią arytmetyczną:

```
double d[] = {1, 2, 3, 4, 5}; // jednowymiarowa tablica składająca
                              // się z 5 elementów
```

funkcję `Mean()` możemy w programie wywołać korzystając z następujących sposobów:

1. jawnie określamy rozmiar tablicy (pomniejszony o jeden):

```
long double x = Mean(d, 4);
```

2. do określenia rozmiaru tablicy wykorzystujemy operatory czasu kompilacji `sizeof()`:

```
long double y = Mean(d, (sizeof(d) / sizeof(d[0])) - 1);
```

3. korzystamy z makrodefinicji (makra) `ARRAYSIZE()`, określającej rozmiar tablicy będącej jej argumentem:

```
long double z = Mean(d, ARRAYSIZE(d) - 1);
```

Ćwiczenie 2.22.

C++ nie posiada funkcji bibliotecznej obliczającej średnią harmoniczną wyrazów tablicy otwartej.

Istnieją eksperymenty, w których określa się np. średni czas $x_i > 0$ życia zwierząt poddawanych działaniu różnego rodzaju nowych środków farmakologicznych. W analizie takiego rodzaju eksperymentów nie wylicza się średniej arytmetycznej czasu życia zwierząt biorących udział w doświadczeniu, gdyż czas taki jest trudny do określenia. Zamiast wyliczania średniej arytmetycznej, do obliczeń stosuje się średnią harmoniczną, będącą ilorazem liczby obserwacji i sumy odwrotności danych liczbowych:

$$\langle x_h \rangle = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Pokazana na wydruku 2.15 funkcja `HarmonicMean()` oblicza średnią harmoniczną niezerowych i nieujemnych elementów tablicy otwartej `Data`.

Wydruk 2.15. *Kod modułu Unit15.cpp projektu Projekt_15.bpr realizującego obliczanie średniej harmonicznej wyrazów tablicy otwartej Data*

```
#include <iostream.h>
#include <conio.h>
#include <vcl.h>
#pragma hdrstop
// prototyp funkcji HarmonicMean()
long double HarmonicMean(const double *Data,
                         const int Data_Size);
int main()
{
  double d[] = {1, 2, 3.9, 4.9, 5, 6, 7};
  long double HM = HarmonicMean(d, ARRAYSIZE(d) - 1);
  cout << "Średnia harmoniczna= " << HM << endl;
```

```
  getch();
  return 0;
}
//---------------------------------------------------------
long double HarmonicMean(const double *Data,
                         const int Data_Size)
{
  long double sum = 0, HM;
  for (int i=0; i<= Data_Size; i++)
    {
      if (Data[i] > 0)
        sum += 1.0/Data[i];
    };
  if (sum != 0)
    HM = (Data_Size+1.0)/sum;
    else
      cout << "Niedozwolone parametry funkcji" << endl;
  return HM;
}
```

Przyglądając się powyższym zapisom musimy stwierdzić, iż posługiwanie się tablicami otwartymi w C++ nie jest czynnością zbyt skomplikowaną, chociaż należy przyznać, iż w porównaniu z Object Pascalem (gdzie nie musimy martwić się ustalaniem ich rozmiaru) może wywołać wrażenie występowania pewnej nadmiarowości kodu.

Wskaźniki do egzemplarzy klas

Do tej pory, w celu uzyskania dostępu do elementów egzemplarza wybranej klasy, odwoływaliśmy się do niego w sposób bezpośredni. Pamiętamy, że po to, aby uzyskać bezpośredni dostęp do elementu egzemplarza klasy, wystarczy podać nazwę egzemplarza klasy, zaś po zastosowaniu operatora w postaci kropki (.) — nazwę wybranego elementu (np. funkcji składowej). Z deklaracją wskaźnika do egzemplarza klasy wiąże się możliwość uzyskiwania pośredniego dostępu do elementów tej klasy, co z kolei skutkuje koniecznością posługiwania się pewnym bardzo ważnym operatorem `->`.

Ćwiczenie 2.23.

Powróćmy do projektu *Projekt_12.bpr*. Zmodyfikujemy go w ten sposób, aby było możliwe uzyskanie zarówno bezpośredniego, jak i pośredniego dostępu do egzemplarza wybranej klasy. Na początku przepiszemy znaną już nam niezwykle prostą klasę `Vehicle`. Następnie zdefiniujemy egzemplarz tej klasy o nazwie `V` oraz wskaźnik do niego o nazwie `pV` (*pointer to V*), czyli `*pV`.

Wydruk 2.16. *Kod modułu Unit16.cpp projektu Projekt_16.bpr*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
class Vehicle
```

```
{
 public:
   int GetWheels() {return Wheels;}
   void SetWheels(int numWheels) {Wheels=numWheels;}
 private:
   int Wheels;
}V, *pV;// deklaracja egzemplarza klasy i wskaźnika do niego
//-------------------------------------------------------
int main()
{
  cout <<" bezpośrednie uzyskanie dostępu do egz.klasy V"
       << endl;
  V.SetWheels(12);
  cout << V.GetWheels() << endl;

  pV = &V; // przypisanie wskaźnikowi pV adresu egzemplarza V
  cout <<" pośrednie uzyskanie dostępu do egz.klasy V"
       << endl;
  cout << pV->GetWheels();

  getch();
  return 0;
}
```

Analizując przedstawiony na powyższym wydruku kod, bez trudu zauważymy, iż operację przypisania wskaźnikowi `pV` adresu egzemplarza `V` klasy `Vehicle` wykonaliśmy korzystając z operatora adresowego `&`. W konsekwencji skorzystanie ze wskaźnika do egzemplarza wybranej klasy możliwe jest dzięki zastosowaniu operatora `->` (strzałka). Zapis:

```
pV->GetWheels();
```

oznacza, że jeżeli `pV` jest wskaźnikiem do egzemplarza klasy, wówczas

```
pV->SkładowaKlasy;
```

odwołuje się do konkretnej składowej klasy. Zauważmy, że `pV` wskazuje na konkretny egzemplarz klasy, zatem do wybranego elementu klasy można odwołać się również w następujący sposób:

```
(*pV).GetWheels();
```

Jednak (jak się już niebawem przekonamy) wskaźniki do egzemplarzy klas (i nie tylko, również np. do struktur) są tak często używane, iż dla wygody w dalszej części książki będziemy posługiwać się operatorem `->`.

Wskaźnik this

W języku C++ istnieje słowo kluczowe `this`, będące ważnym elementem wielu tzw. „przeładowywanych operatorów". Każda funkcja składowa aplikacji lub ogólnie — obiektu, w momencie wywołania uzyskuje automatycznie wskaźnik do obiektu, który ją wywołał. Dostęp do tego wskaźnika uzyskuje się dzięki słowu (wskaźnikowi) `this`, który jest *niejawnym parametrem* wszystkich funkcji wchodzących w skład obiektu, a w szczególności egzemplarza klasy.

Ćwiczenie 2.24.

Funkcje składowe egzemplarza klasy mogą uzyskiwać bezpośredni dostęp do danych (zmiennych) zdefiniowanych w macierzystej klasie. I tak instrukcja przypisania:

```
Wheels=12;
```

jest tak naprawdę skróconą wersją następującej instrukcji:

```
this->Wheels=12;
```

O tym, że wskaźnik `this` jest w rzeczywistości niejawnym parametrem funkcji wchodzących w skład klasy, przekona nas zmodyfikowana wersja projektu *Projekt_16.bpr* przedstawiona na wydruku 2.17.

Wydruk 2.17. *Kod modułu Unit17.cpp projektu Projekt_17.bpr*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
class Vehicle
{
 public:
   int GetWheels() {return this->Wheels;}
   void SetWheels(int numWheels) {this->Wheels=numWheels;}
 private:
   int Wheels;
}V;
//---------------------------------------------------------
int main()
{
  V.SetWheels(12);
  cout << V.GetWheels() << endl;
  getch();
  return 0;
}
```

Przedstawiony algorytm należy potraktować jako bardzo poglądowy, gdyż na co dzień nie używamy jawnie wskaźnika `this`. Wskaźnik ten odgrywa bardzo wielką rolę w przypadku przeładowywania operatorów. Jednak zagadnienie to wykracza poza ramy naszej książki, również z tego powodu, iż w dalszej jej części nie będziemy posługiwać się jawnie przeładowywanymi operatorami.

Obsługa wyjątków

Możliwość programowej obsługi wyjątków jest bardzo ważnym mechanizmem języka C++, pozwalającym w odpowiedni sposób kontrolować błędy powstające w trakcie działania programu. Wyjątki pozwalają w sposób automatyczny tworzyć procedury obsługi błędów.

Obsługa wyjątków w C++ opiera się na trzech słowach kluczowych: `try`, `catch` oraz `throw`. Ogólna postać bloku instrukcji `try...catch()` wygląda następująco:

```
try // „próbuj" wykonać operację
  {
     // ciąg wykonywanych operacji
  }
catch(lista argumentów)  // jeżeli operacja nie powiodła się
                         // „przechwyć wyjątek"
  {
      // przetwarzanie wyjątku
      // jeżeli nastąpił wyjątek, pokaż komunikat
  }
```

Każdy pojawiający się wyjątek zostanie przechwycony i odpowiednio przetworzony przez instrukcję `catch()` występującą bezpośrednio po `try`.

Wyjątki możemy też przechwytywać za pomocą instrukcji `throw` rzutującej (modyfikującej) wyjątki na wybrany obiekt wyjątku, czyli jego wartość:

```
throw wyjątek;
```

Każda taka instrukcja powinna znajdować się w bloku `try...catch()` lub w jednej z wywoływanych tam funkcji.

Ćwiczenie 2.25.

Jako przykład użycia bloku instrukcji `try` i bezparametrowej `catch()` rozpatrzmy sytuację, w której chcemy obliczyć średnią harmoniczną z elementów pewnej tablicy otwartej (patrz ćwiczenie 2.22). Jak wiemy, operacja dzielenia przez zero jest niewykonalna.

Wydruk 2.18. *Zapis funkcji HarmonicMean() z wykorzystaniem bloku instrukcji try...catch()*

```
long double HarmonicMean(const double *Data,
                         const int Data_Size)
{
  long double sum = 0, HM;
  for (int i=0; i<= Data_Size; i++)
    {
      try {
            sum += 1.0/Data[i];
      }
      catch(...){
        cout << "Niedozwolone parametry funkcji HM()" << endl;
      }

    };
    HM = (Data_Size+1.0)/sum;
  return HM;
}
```

Jeżeli jednym z elementów tablicy otwartej będzie liczba zero:

```
double d[] = {1, 2, 3.9, 4.9, 5, 9, 0};
```

wówczas w trakcie działania programu kompilator poinformuje nas o przykrym fakcie, iż nastąpiła próba dzielenia przez zero, tak jak pokazano to na rysunku 2.6.

Rysunek 2.6. *Przechwytywanie wyjątku za pomocą bloku try...catch()*



Ćwiczenie 2.26.

Argumentem funkcji obliczającej średnią harmoniczną nie powinna być też tablica otwarta, zawierająca elementy w postaci kombinacji liczb ujemnych i dodatnich, gdyż w takim przypadku mianownik we wzorze na średnią harmoniczną może się zerować. Jeżeli funkcję `HarmonicMean()` pokazaną na wydruku 2.18 wywołamy z argumentem w postaci tablicy:

```
double d[] = {1, 2, 3.9, 4.9, 5, 9, -7, 0 };
```

wynik działania programu jako całości będzie identyczny z pokazanym na rysunku 2.6. Przetestujmy ten wariant funkcji, a przekonamy się, iż nie otrzymamy dostatecznej informacji o błędach w deklaracji elementów tablicy otwartej `d`.

Aby otrzymać pełniejszą informację o występujących nieprawidłowościach, zastosujemy instrukcję `throw`, tak jak pokazuje to wydruk 2.19.

Wydruk 2.19. *Zapis funkcji HarmonicMean() z wykorzystaniem bloku instrukcji try...catch() oraz instrukcji throw*

```
 long double HarmonicMean(const double *Data,
                          const int Data_Size)
{
  long double sum = 0, HM;
  for (int i=0; i<= Data_Size; i++)
    {
      try {
        if ((! Data[i]) || (Data[i]<0)) throw Data[i];
        sum += 1.0/Data[i];
      }
      catch(...){
        cout << "Niedozwolone parametry funkcji HM()" << endl;
      }

    };
    HM = (Data_Size+1.0)/sum;
  return HM;
}
```

W tym przypadku w trakcie działania programu otrzymamy tyle informacji o błędach, ile jest błędnie zadeklarowanych elementów tablicy, będącej argumentem funkcji `HarmonicMean()`. Sytuację tę ilustruje rysunek 2.7.

Rysunek 2.7. *Przechwytywanie wyjątku za pomocą bloku try...catch() oraz instrukcji throw*



Ćwiczenie 2.27.

Dla bardziej wymagającego użytkownika pokazany wcześniej sposób przechwytywania błędów może okazać się niewystarczający. Pójdźmy krok dalej. Zażądajmy mianowicie, aby działanie programu zawierającego np. tablicę otwartą z błędnie określonymi elementami, okazało się niemożliwe! W tym celu skorzystajmy z bloku `try` oraz instrukcji `catch()` z parametrem będącym zarazem pierwszym argumentem funkcji `HarmonicMean()`. Zapis:

```
...
catch(const double *Data)
...
```

spowoduje jawne wskazanie na przechwytywany obiekt wyjątku. W przypadku błędnej deklaracji jakiegokolwiek elementu tablicy otwartej wskazywanej przez `Data`, program wykonywalny *.exe* nie powinien w ogóle działać!

Wydruk 2.20. *Zapis funkcji HarmonicMean() z wykorzystaniem bloku sparametryzowanej instrukcji try...catch() oraz instrukcji throw. Tak określoną funkcję wykorzystuje Projekt_18.bpr*

```
long double HarmonicMean(const double *Data,
                         const int Data_Size)
{
  long double sum = 0, HM;
  for (int i=0; i<= Data_Size; i++)
    {
      try {
        if ((! Data[i]) || (Data[i]<0)) throw Data[i];
        sum += 1.0/Data[i];
      }
      catch(const double *Data){
        cout << "Niedozwolone parametry funkcji HM()" << endl;
      }

    };
    HM = (Data_Size+1.0)/sum;
  return HM;
}
```

Podsumowując, musimy przyznać, iż możliwość obsługi wyjątków daje do dyspozycji programistom C++ niezwykle wydajne narzędzie, pozwalające kontrolować błędy występujące podczas wykonywania programu w jego najbardziej newralgicznych punktach. Niemniej jednak należy zdawać sobie sprawę z faktu, iż wyjątki w żadnym wypadku nie mogą być używane jako prosta metoda zwracania komunikatów przez aplikację w zupełnie niegroźnej sytuacji. Wynika to z faktu, iż wygenerowanie i obsłużenie wyjątku pociąga za sobą konieczność przydzielenia określonych zasobów procesora, który w tym czasie może być zajęty przez zupełnie inne zadania. Wyjątki należy traktować jako zło konieczne i w żadnym wypadku nie można ich używać jako normalnej drogi zwracania informacji.

Podsumowanie

W niniejszym rozdziale zostały przedstawione podstawowe pojęcia związane z techniką obiektowego programowania w C++. Omówiliśmy podstawowe elementy języka odnoszące się do struktur, funkcji, klas i wyjątków. Przypomnienie wiadomości na temat wskazań, adresów oraz dynamicznego alokowania w pamięci różnego typu danych bardzo nam w przyszłości pomoże w zrozumieniu mechanizmu obsługi zdarzeń już z poziomu Borland C++Buildera 6. Przedstawienie szeregu pożytecznych przykładów praktycznego wykorzystania elementów języka C++ ułatwi nam samodzielne wykonanie zamieszczonych w tym rozdziale ćwiczeń.