Język C++ – podstawy programowania

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Język C++ – podstawy programowania

Paweł Mikołajczak

Lublin 2011

Instytut Informatyki UMCS
Lublin 2011

Paweł Mikołajczak
JĘZYK C++ – PODSTAWY PROGRAMOWANIA

Recenzent: Marek Stabrowski

Opracowanie techniczne: Karol Kuczyński, Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
ESUS Agencja Reklamowo-Wydawnicza Tomasz Przybylak
ul. Ratajczaka 26/8
61-815 Poznań
www: www.esus.pl

ISBN: 978-83-62773-11-4

Spis treści

Przedmowa ix

1 Specyficzne elementy języka C++ 1
1.1. Wstęp 2
1.2. Rys historyczny 2
1.3. Strumienie wejścia/wyjścia w C++ 4
1.4. Nowa postać komentarzy 6
1.5. Dowolne rozmieszczenie deklaracji 7
1.6. Przekazywanie argumentów przez referencje 7
1.7. Argumenty domniemane 11
1.8. Przeciążanie funkcji 13
1.9. Wzorce funkcji 14
1.10. Funkcje inline 16
1.11. Dynamiczne zarządzanie pamięcią 17
1.12. Słowa kluczowe języka C++ 23
1.13. Schemat programu w języku C++ 24

2 Wprowadzenie do programowania obiektowego 27
2.1. Wstęp 28
2.2. Paradygmaty programowania 28
2.3. Obiekty i klasy 30
2.4. Hermetyzacja danych 34
2.5. Dziedziczenie 35
2.6. Polimorfizm 36
2.7. Podsumowanie terminologii 37
2.8. Środowisko programistyczne 38

3 Klasy i obiekty 41
3.1. Wstęp 42
3.2. Deklaracja i definicja klasy 42
3.3. Wystąpienie klasy (definiowanie obiektu) 43
3.4. Dostęp do elementów klasy 43
3.5. Metody klasy 47

3.6. Klasa z akcesorami 50
3.7. Funkcje składowe const 53

4 Konstruktory i destruktory **57**
4.1. Wstęp 58
4.2. Inicjalizacja obiektu klasy 58
4.3. Konstruktory i destruktory domyślne 59
4.4. Konstruktor jawny 61
4.5. Wywoływanie konstruktorów i destruktorów 63
4.6. Rozdzielenie interfejsu od implementacji 65
4.7. Wskaźnik do obiektu this 67
4.8. Wskaźnik this – kaskadowe wywołania funkcji 68
4.9. Tablice obiektów 71
4.10. Inicjalizacja tablic obiektów nie będących agregatami 73
4.11. Tablice obiektów tworzone dynamicznie 75
4.12. Kopiowanie obiektów 77
4.13. Klasa z obiektami innych klas 78

5 Dziedziczenie i hierarchia klas **83**
5.1. Wstęp 84
5.2. Hierarchiczna struktura dziedziczenia 84
5.3. Notacja UML (Unified Modeling Language) 88
5.4. Proste klasy pochodne 92
5.5. Konstruktory w klasach pochodnych 98
5.6. Dziedziczenie kaskadowe 104
5.7. Dziedziczenie wielokrotne bezpośrednie 117

6 Funkcje zaprzyjaźnione **121**
6.1. Wstęp 122
6.2. Funkcja niezależna zaprzyjaźniona z klasą 123
6.3. Funkcja składowa zaprzyjaźniona z inną klasą 134
6.4. Klasy zaprzyjaźnione 141

7 Przeciążanie operatorów **143**
7.1. Wstęp 144
7.2. Definicje 144
7.3. Przeciążony operator dodawania (+) 147
7.4. Przeciążony operator mnożenia (*) 150
7.5. Funkcja operatorowa w postaci niezależnej funkcji 154
7.6. Przeciążanie operatorów równości i nierówności 161
7.7. Przeciążanie operatora przypisania (=) 165
7.8. Przeciążanie operatora wstawiania do strumienia («) 173

8 Funkcje statyczne i wirtualne **177**
8.1. Wstęp 178
8.2. Metody i dane statyczne 178
8.3. Polimorfizm 183
8.4. Funkcje wirtualne 187
8.5. Funkcje abstrakcyjne 194

9 Klasy wirtualne i zagnieżdżone **203**
9.1. Wstęp 204
9.2. Klasy wirtualne 204
9.3. Klasy zagnieżdżone 212

10 Wskaźniki do klas **217**
10.1. Wstęp 218
10.2. Wskaźniki klasowe 218
10.3. Wskaźniki składowych klas 224

11 Techniki obsługi błędów **229**
11.1. Wstęp 230
11.2. Funkcje walidacyjne 230
11.3. Graniczne wartości – plik limits.h 234
11.4. Narzędzie assert() i funkcja abort() 237
11.5. Przechwytywanie wyjątków 239

12 Szablony w C++ **265**
12.1. Wstęp 266
12.2. Przeciążanie funkcji 266
12.3. Szablony funkcji 276
12.4. Szablony klas 292

Słownik angielsko-polski **307**

Skorowidz **325**

Przedmowa

Język C++ jest uznawany za najlepszy język do programowania obiektowego. Jest on nadzbiorem języka C. Powstał w Bell Labs (USA), gdzie Bjarne Stroustrup stworzył go na początku lat osiemdziesiątych. Potem język C++ był wiele lat rozwijany. Prace nad standardem języka zaczęły się w roku 1990. Ostatecznie, po rozszerzeniu języka o wyjątki, RTTI, szablony oraz bibliotekę STL w 1998 roku przyjęto międzynarodowy standard (ISO-/IEC 14882:1998). W roku 2003 pojawiła się druga wersja standardu, ale zawierała ona tylko drobne korekty.

Trudno sobie obecnie wyobrazić zawodowego programistę, który nie zna przynajmniej podstaw języka C++.

Język C++ jest językiem skomplikowanym i trudnym, niemniej możliwym do opanowanie. Jak pisze twórca języka, B. Stroustrup w swoim najnowszym podręczniku opublikowanym w 2009 roku, z ponad tysiąca studentów pierwszego roku, których uczył w Texas A&M University większość z nich odniosła sukces (zdała egzaminy) mimo, że 40 procent z nich nie miało nigdy do czynienia z programowaniem. To wyznanie wybitnego fachowca jest optymistyczne. Należy pamiętać, że klasyczne i kompletne podręczniki programowania są bardzo obszerne (podręcznik B. Stroustrupa liczy sobie 1106 stron, a podręcznik S. Prata – 1303 strony). Skrypt akademicki z natury rzeczy jest tylko wprowadzeniem do tematyki przedmiotu, ma za zadnie przedstawić wstępne i elementarne zagadnienia związane z omawianą dziedziną, stąd jego objętość nie może być zbyt duża.

Na potrzeby studentów kierunku informatyka opracowaliśmy ten podręcznik wybierając dość arbitralnie prezentowany materiał. Prezentując język C++, zakładamy, że studenci znają język C.

Niniejszy podręcznik jest przeznaczony głównie dla studentów 3-letnich studiów zawodowych (licencjatów) a także dla studentów 2-letnich studiów uzupełniających. Podręcznik powstał na podstawie notatek wykorzystywanych przeze mnie w trakcie prowadzenia wykładów z przedmiotów „Języki programowania", oraz „Język C i C++" dla studentów UMCS w latach 1995 – 2005 i dalszych.

W podręczniku przyjąłem zasadę, że nauka programowania oparta jest

na licznych przykładach, które ilustrują praktyczne zastosowania paradygmatów programowania i składni języka C++.

Podręcznik ma być pomocą dydaktyczną wspierającą naukę programowania realizowana w ramach 30-godzinnego wykładu i ćwiczeń laboratoryjnych. Należy pamiętać, że podręcznik nie stanowi pełnego kompendium wiedzy o języku C++, jest jedynie prostym i przystępnym wprowadzeniem w fascynujący świat pisania efektywnych i efektownych programów komputerowych. Wszystkie przykłady programów, które zamieściłem w tym podręczniku zostały przetestowane za pomocą kompilatora C++ Builder 6 firmy Borland, na platformie Windows. Większość programów sprawdzana także była na platformie Linuksowej oraz QT.

Rozdział 1

Specyficzne elementy języka C++

1.1. Wstęp **2**
1.2. Rys historyczny **2**
1.3. Strumienie wejścia/wyjścia w C++ **4**
1.4. Nowa postać komentarzy **6**
1.5. Dowolne rozmieszczenie deklaracji **7**
1.6. Przekazywanie argumentów przez referencje **7**
1.7. Argumenty domniemane **11**
1.8. Przeciążanie funkcji **13**
1.9. Wzorce funkcji **14**
1.10. Funkcje inline **16**
1.11. Dynamiczne zarządzanie pamięcią **17**
1.12. Słowa kluczowe języka C++ **23**
1.13. Schemat programu w języku C++ **24**

1.1. Wstęp

Język C oraz C++ są najbardziej popularnymi językami programowania. W ośrodkach akademickich, a także w dużych korporacjach komputerowych, tworzone są co jakiś czas nowe języki programowania wysokiego poziomu, ale jak uczy doświadczenie, wszystkie te pomysły są z początku entuzjastycznie przyjmowane przez programistów, nowe języki mają swoich zagorzałych wielbicieli, a po jakimś czasie powoli odchodzą w niepamięć, lub są marginalizowane. Moda na dany język przemija. Jedynie język C trwa i ma się bardzo dobrze, dzięki także swoim modyfikacjom (język C++ jest nadzbiorem języka C). Należy przypomnieć, że język C jest integralnie związany z systemem Unix. Język C++ został zaprojektowany jako obiektowa wersja języka C. W języku C++ powstało i powstaje wiele znaczącego oprogramowania: systemy operacyjne, programy narzędziowe, programy użytkowe, programy do obsługi sieci komputerowych. Na rynku pracy, programista bez znajomości języka C/C++ i systemu Unix, ma nikłe szanse. Obecnie od programisty wymaga się znajomości przynajmniej dwóch języków programowania i znajomości minimum dwóch systemów operacyjnych, język C/C++ i Unix są bezwzględnie wymagane, znajomość innych języków programowania oraz innych systemów (systemy operacyjne rodziny Windows) jest dodatkowym atutem. Obecnie (rok 2010) dominującym jest język C++ integralnie związany z programowaniem obiektowym. Język C++ jest nadzbiorem języka C. Zwykle podręczniki dydaktyczne do nauki programowania w języku C++ zawierają obszerną część poświęconą programowaniu w języku C. Aby przyspieszyć projektowanie i implementowanie programów, zostały opracowane specjalne systemy wspomagające prace programistów. Doskonałymi narzędziami do tworzenia aplikacji są pakiety takie jak np. C++ Builder firmy Borland czy Visual C++ firmy Microsoft. Te produkty należą do systemów szybkiego projektowania aplikacji (ang. RAD - Rapid Application Development). Korzystając z tych pakietów możemy efektywnie konstruować 32-bitowe programy pracujące w systemie Windows.

Wydaje się rozsądne, aby dobrze wykształcony informatyk opanował następujące narzędzia programistyczne:
— język C
— język C++
— język Java

1.2. Rys historyczny

Historia powstawania języka C a następnie C++ jest długa, rozwojem tych języków zajmowało się wielu wspaniałych fachowców. Język C powstał

dzięki przejęciu podstawowych zasad z dwóch innych języków: BCPL i języka B. BCPL opracował w 1967 roku Martin Richards. Ken Thompson tworzył język B w oparciu o BCPL. W 1970 roku w Bell Laboratories na podstawie języka B opracowano system operacyjny UNIX.

Twórcą języka C jest Dennis M. Ritchie, który także pracował w Bell Laboratories. W 1972 roku język C był implementowany na komputerze DEC PDP-11. Jedną z najważniejszych cech języka C jest to, że programy pisane w tym języku na konkretnym typie komputera, można bez większych kłopotów przenosić na inne typy komputerów. Język C był intensywnie rozwijany w latach siedemdziesiątych. Za pierwszy standard przyjęto opis języka zamieszczony w dodatku pt. "C Reference Manual" podręcznika „The C Programming Language". Publikacja ukazała się w 1978 roku.

Opublikowany podręcznik definiował standard języka C, reguły opisane w tym podręczniku nazywamy standardem K&R języka C. W 1983 roku Bell Laboratories wydało dokument pt. "The C Programming Language - Reference Manual", którego autorem był D. M. Ritchie. W 1988 Kernighan i Ritchie opublikowali drugie wydanie "The C Programming Language". Na podstawie tej publikacji, w 1989 roku Amerykański Narodowy Instytut Normalizacji ustalił standard zwany standardem ANSI języka C. Za twórcę języka C++ (przyjmuje się, że jest to nadzbiór języka C) uważany jest Bjarne Stroustrup, który w latach osiemdziesiątych pracując w Bell Laboratories rozwijał ten język, tak aby zrealizować programowanie obiektowe. Bjarne Stroustrup wspólnie z Margaret Ellis opublikował podręcznik „The Annotated C++ Reference Manual". W 1994 roku Amerykański Narodowy Instytut Normalizacji opublikował zarys standardu C++.

Nowym językiem, silnie rozwijanym od 1995 roku jest język programowania Java. Język Java opracowany w firmie Sun Microsystem, jest oparty na języku C i C++. Java zawiera biblioteki klas ze składnikami oprogramowania do tworzenia multimediów, sieci, wielowątkowości, grafiki, dostępu do baz danych i wiele innych. Jak już to zasygnalizowano, język C jest podstawą języka C++ i Javy. Wobec tego należy doskonale znać ten język programowania.

W 1997 roku ukazało się trzecie wydanie książki Bjarne'a Stroustrupa „Język programowania C++". Oczywiście język C++ jest nadal rozwijany, ale uznano, że trzecie wydanie podręcznika Stroustrupa wyczerpująco ujmuje nowe elementy języka i de facto może być traktowane jako standard. Ostateczne standard języka C++ został przyjęty przez komitet ANSI/ISO w 1998 roku, została wtedy ustalona jednolita specyfikacja języka. W 2002 roku przyjęto kolejną poprawioną wersję tego standardu. Dokument opisujący ten standard jest dostępny za pośrednictwem sieci Internet pod numerem 14882 na stronie http://www.ansi.org. Język C++ jest dzisiaj najpopularniejszym językiem programowania stosowanym przez zawodowców. Język

ten jest także podstawą wykształcenia informatyków. Na uniwersytetach, gdzie przez wiele lat język Pascal był głównym językiem programowania, obecnie podstawą nauczania staje się język C/C++.

Istotnym elementem rozwoju języka C++ było dołączenie do standardu bardzo elastycznej i o dużych możliwościach biblioteki szablonów, znanej pod nazwą Standardowej Biblioteki Szablonów (ang. Standard Template Library, STL). Przyjęcie tej biblioteki do standardu języka nastąpiło w 1997 roku. Faktycznie biblioteka szablonów umożliwiła realizowanie nowego paradygmatu programowania – programowanie uogólnionego (ang. generic programming).

Zasadnicza różnica pomiędzy językami C i C++ polega na tym, że C++ posiada specyficzne elementy związane bezpośrednio z programowaniem obiektowy oraz kilka istotnych usprawnień. W języku C++ występują też specyficzne elementy nie związane z programowaniem obiektowym. Są to:
— Nowe mechanizmy wejścia/wyjścia
— Nowa postać komentarzy
— Dowolne rozmieszczenie deklaracji
— Przekazywanie argumentów przez referencje
— Argumenty domniemane
— Przeciążanie funkcji
— Dynamiczne zarządzanie pamięcią
— Funkcje inline

1.3. Strumienie wejścia/wyjścia w C++

W C++ pojawiły się nowe mechanizmy wejścia/wyjścia. Standardowe strumienie języka C to:
— stdin
— stdout
— stderr
— stdlog

W języku C++ występują dodatkowe, predefiniowane strumienie:
— cin
— cout
— cerr
— clog

Tego typu mechanizmy obsługi wejścia/wyjścia są bardzo wygodne i korzystnie zastępują funkcje printf() i scanf(). Na przykład:

```
cout << wyrazenie_1 << wyrazenie_2 <<wyrazenie_3
```

wysyła do strumienia cout (oznaczającego normalne wyjście stdout) kolejne wartości wyrażeń. Podobnie

```
cin >> wartosc_1 >> wartosc_2 >>wartosc_3
```

odczytuje ze strumienia cin (standardowe wejście stdin) wartość jednego z typów: char, short, int, long, float, double lub char *.

W celu zilustrowania typowej dla C++ obsługi wejścia/wyjścia napiszemy pokazany na wydruku 1.1. program w konwencji języka C używając funkcji printf() i scanf() i dla kontrastu pokażemy program 1.2 typowy dla języka C++ wykorzystujący strumienie cin i cout.

Listing 1.1. Program napisany w konwencji języka C

```
1 #include <stdio.h>
  #include <conio.h>
3
  int main()
5 {
   int x;
7  float y;
   printf("\nPodaj liczbe calkowita: ");
9  scanf("%d",&x);
   printf("\nPodaj liczbe rzeczywista: ");
11 scanf("%f",&y);
   printf("\n%d razy %f = %f",x, y, x*y);
13 getch();
   return 0;
15 }
```

W pokazanym klasycznym programie napisanym w języku C należy wprowadzić liczbę całkowitą i liczbę rzeczywistą. Program wyświetli iloczyn tych liczb. Taki sam program, ale napisany w konwencji C++ pokazuje wydruk 1.2.

Listing 1.2. Program napisany w konwencji C++

```
1 #include <iostream.h>
  #include <conio.h>
3
  int main()
5 {
  int x;
7 float y;
   cout << "Podaj liczbe calkowita: ";
9  cin >> x;
   cout << "Podaj liczbe rzeczywista: ";
11 cin >> y;
   cout << x << " razy " << y << " = " << x*y;
```

```
13     getch();
       return 0;
15 }
```

Należy zwrócić uwagę na istotne różnicę pomiędzy pokazanymi programami. Po pierwsze, w programie w konwencji języka C++ należy włączyć inny plik nagłówkowy:

```
#include <iostream.h>
```

Zamiast instrukcji:

```
printf("\nPodaj liczbe calkowita: ");
```

mamy instrukcję:

```
cout << "Podaj liczbe calkowita: ";
```

Łańcuch "Podaj liczbe calkowita" został przesłany do strumienia przy pomocy operatora «. W zwykłym C jest to operator przesunięcia bitów w lewą stronę, który działa na wartościach całkowitych przesuwając ich bity o podaną ilość miejsc. Podobnie działa operator przesunięcia bitów w prawą stronę », który przesuwa bity w prawo. W C++ operatory « i » dalej funkcjonują jako operatory przesuwania bitów, ale mogą mieć także inne znaczenie, zależne od kontekstu. W pokazanym przykładzie, znak « może także być użyty do wysłania danych wyjściowych do strumienia cout i z tego powodu nazywany jest operatorem umieszczającym (ang. insertor), tzn. umieszcza dane w strumieniu cout. Podobnie znak » może być wykorzystany do odczytania danych wejściowych pochodzących ze strumienia cin, dlatego zwany jest operatorem pobierającym (ang. extractor). Ta zdolność nadawania nowego znaczenia typowym operatorom nazywana jest przeciążeniem operatorów (ang. operator overloading) jest charakterystyczną cechą języka C++.

1.4. Nowa postać komentarzy

W języku C++ wprowadzono dodatkowo symbol jednoliniowego komentarza //. C++ ignoruje zawartość linii następującej po symbolu //. Pierwotnie standard ANSI języka C ustalał, że tylko zawartość umieszczona pomiędzy znakami /* i */ jest komentarzem. W tym przypadku, nie ma ograniczeń, co do ilości linii. Jeżeli mamy krótki komentarz to zazwyczaj używamy symbolu //, dla długich komentarzy dalej używamy symboli /* i */. Komentarze są istotna częścią programów komputerowych. Nie wpły-

wają na działanie programu, dokumentują podejmowane akcje i specyfikują używane zmienne i stałe. Komentarzy nie można zagnieżdżać. W fazie wstępnej, analizy tekstu programu komentarz zostaje przekształcony w spacje. W tej sytuacji napis:

```
z=x*/*iloczyn*/ y
```

zostanie przekształcony w napis:

```
z = x * y
```

1.5. Dowolne rozmieszczenie deklaracji

Język C++ zezwala na umieszczanie deklaracji w dowolnym miejscu, pod warunkiem, że wykorzystanie jej nastąpi przed użyciem deklarowanej zmiennej. Można także używać wyrażeń w inicjalizacji zmiennych. Jak pamiętamy, standard języka C wymagał aby deklaracje były umieszczone na początku funkcji lub bloku. Przykładowy fragment kodu może mieć postać:

```
int a;
a = 10;
int b;
b = 100;
int y = a * b;
```

1.6. Przekazywanie argumentów przez referencje

W C++ argument może być przekazywany przez referencję. W tym celu należy poprzedzić jego nazwę symbolem & w nagłówku i prototypie funkcji. Używając standardowo funkcji mamy na uwadze dwa ograniczenia: argumenty są przekazywane przez wartość, funkcja może zwrócić tylko jedną wartość. W języku C to ograniczenie jest usuwane gdy zastosujemy wskaźniki. W języku C++ przekazywanie argumentów dodatkowo poprzez referencję może usunąć dodatkowo wymienione ograniczenia. W C++ przekazywanie argumentów, oprócz przekazywanie przez wartość możemy realizować dodatkowo jeszcze na dwa sposoby: z wykorzystaniem wskaźników i z wykorzystaniem referencji. Należy pamiętać, że w tym przypadku funkcja nie pracuje na kopii, program przekazuje funkcji oryginalny obiekt. Omówimy trzy krótkie programy, w których wykorzystana będzie funkcja zamiana(). Funkcja zamiana() ma otrzymać dwie zmienne, zainicjalizowane w funkcji main() i zmienić ich wartości. Do funkcji zamiana() przekazane będą

wartości argumentów, wskaźniki i referencje. Struktura trzech programów pokazanych na wydrukach 1.3, 1.4 i 1.5 jest bardzo podobna. Zanalizujemy program pokazany na wydruku 1.3. Wewnątrz funkcji main() program inicjalizuje dwie zmienne całkowite, zmienna x otrzymuje wartość 1 a zmienna y - wartość 100. Następnie wywoływana jest funkcja zamiana() z argumentami x i y. Następuje klasyczne przekazanie argumentów przez wartość. Funkcja zamiana() zmienia wartości zmiennych (co widać na wydruku). Ale gdy ponownie sprawdzane są wartości zmiennych w funkcji main() okazuje się, że wartości nie uległy zmianie.

Podczas przekazywania argumentów przez wartość, tworzone są lokalne kopie tych zmiennych wewnątrz funkcji zamiana(). Po skończeniu zadania kopie są niszczone i wartości zmiennych w funkcji main() pozostają niezmienione.

Listing 1.3. Argumenty przekazywane do funkcji przez wartość

```
 1 #include <iostream.h>
   #include <conio.h>
 3 void zamiana(int, int);
   int main()
 5 {
     int x = 1, y = 100;
 7   cout << "Funkcja main, wynik przed zamiana x: "
       << x << " y: " << y << "\n";
 9   zamiana(x,y);
     cout << "Funkcja main, wynik po zamiana x: "
11     << x << " y: " << y << "\n";
         getch();
13       return 0;
   }
15 void zamiana(int x, int y)
   {
17  int temp;
    cout << "Funkcja zamiana, wynik przed zamiana x: "
19   << x << " y: " << y << "\n";
    temp = x;
21  x = y;
    y = temp;
23  cout << "Funkcja zamiana, wynik po zamiana x: "
     << x << " y: " << y << "\n";
25 }
```

Po uruchomieniu programu mamy następujący wynik:

```
Funkcja main, wynik przed zamiana x: 1 y: 100
Funkcja zamiana, wynik przed zamiana x: 1 : y: 100
Funkcja zamiana, wynik po zamiana x: 100 y: 1
Funkcja main, wynik po zamiana x: 1 y: 100
```

Omówimy podobny program, ale w tym przykładzie posłużymy się wskaźnikami. Program pokazany jest na wydruku 1.4. W zmiennej wskaźnikowej umieszczony jest adres obiektu, przekazując wskaźnik przekazujemy adres obiektu i dlatego funkcja może bezpośrednio operować na wartości zmiennej znajdującej się pod wskazanym adresem, żadne kopie nie są potrzebne. Za pomocą wskaźników umożliwimy funkcji zamiana() rzeczywistą zamianę wartości zmiennych x i y. W prototypie funkcji zamiana() deklarujemy, że argumentami funkcji będą dwa wskaźniki do zmiennych całkowitych:

```
void zamiana(int *, int *);
```

Listing 1.4. Przekazanie przez referencję z wykorzystaniem wskaźników

```
1 #include <iostream.h>
  #include <conio.h>
3 void zamiana(int *, int *);
  int main()
5 {
    int x = 1, y = 100;
7   cout << "Funkcja main, wynik przed zamiana x: "
      << x << " y: " << y << "\n";
9   zamiana(&x, &y);
    cout << "Funkcja main, wynik po zamiana x: "
11    << x << " y: " << y << "\n";
      getch();
13    return 0;
  }
15 void zamiana(int *px, int *py)
  {
17  int temp;
    cout<<"Funkcja zamiana, wynik przed zamiana *px: "
19   <<*px <<" *py: " << *py << "\n";
    temp = *px;
21  *px = *py;
    *py = temp;
23  cout << "Funkcja zamiana, wynik po zamiana *px: "
     << *px << " *py: " << *py << "\n";
25 }
```

Funkcja main() wywołuje funkcję zamiana(), przekazywanymi argumentami są adresy zmiennych x i y:

```
zamiana(&x, &y);
```

W funkcji zamiana() zmiennej temp nadawana jest wartość zmiennej x:

```
temp = *px;
```

W tej instrukcji stosujemy operator wyłuskania (czyli dereferencję). Pod adresem zawartym w zmiennej wskaźnikowej px umieszczona jest wartość x. W następnej linii:

```
*px = *py;
```

zmiennej wskazywanej przez px przypisana jest wartość wskazywana przez py. W następnej linii:

```
*py = temp;
```

wartość przechowywana w zmiennej temp jest przypisana zmiennej wskazywanej przez py. Dzięki takim mechanizmom dokonywana jest prawdziwa zamiana wartości zmiennych x i y co widać po uruchomieniu programu:

```
Funkcja main, wynik przed zamiana x: 1 y: 100
Funkcja zamiana, wynik przed zamiana *px: 1  *py: 100
Funkcja zamiana, wynik po zamiana *px: 100 *py: 1
Funkcja main, wynik po zamiana x: 100 y: 1
```

Program pokazany na wydruku 1.4 działa ale styl programowania daleko odbiega od prostoty. Istnieje inny sposób wykonania tego samego zadania. W kolejnym programie pokazanym na wydruku 1.5. zastosowano referencję.

Listing 1.5. Zastosowanie referencji

```
  #include <iostream.h>
2 #include <conio.h>
  void zamiana(int &, int &);
4 int main()
  {
6   int x = 1, y = 100;
    cout << "Funkcja main, wynik przed zamiana x: "
8      << x << " y: " << y << "\n";
    zamiana(x,y);
10  cout << "Funkcja main, wynik po zamiana x: "
       << x << " y: " << y << "\n";
12     getch();
       return 0;
14 }
  void zamiana(int &rx, int &ry)
16 {
   int temp;
18 cout << "Funkcja zamiana, wynik przed zamiana rx: "
       << rx << " ry: " << ry << "\n";
20 temp = rx;
   rx = ry;
22 ry = temp;
   cout << "Funkcja zamiana, wynik po zamiana rx: "
```

```
24      << rx << " ry: " << ry << "\n";
    }
```

Prototyp funkcji zamiana() ma postać:

```
void zamiana(int &, int &);
```

Wywołanie funkcji zamiana() ma postać:

```
zamiana(x,y);
```

należy zauważyć, że teraz nie są przekazywane adresy zmiennych x i y ale same zmienne. Teraz sterowanie zostaje przekazane do linii:

```
void zamiana(int &rx, int &ry)
```

gdzie zmienne zostaną zidentyfikowane jako referencje. W dalszych instrukcjach funkcji zamiana() następuje rzeczywista zamiana wartości zmiennych. Ponieważ parametry funkcji zamiana() zostały zadeklarowane jako referencje, wartości w funkcji main() zostały przekazane przez referencję, dlatego są zamienione również w tej funkcji. Wynikiem działania programu jest komunikat:

```
Funkcja main, wynik przed zamiana x: 1 y: 100
Funkcja zamiana, wynik przed zamiana rx: 1 ry: 100
Funkcja zamiana, wynik po zamiana rx: 100 ry: 1
Funkcja main, wynik po zamiana x: 100 y: 1
```

1.7. Argumenty domniemane

Dość często mamy do czynienia z sytuacją, gdy przekazujemy funkcji kilka argumentów. Może się okazać, że w wielokrotnie wywoływanej funkcji zmianie ulega tylko jeden parametr. W języku C++ programista może określić, że dany argument jest argumentem domniemanym (domyślnym) i wybrane argumenty. W wywołaniu funkcji możemy pominąć argument domniemany. Pominięty argument domniemany jest automatycznie wstawiany przez kompilator. Argumenty domniemane muszą być umieszczone na liście argumentów najdalej z prawej strony. Wartości domyślne mogą być stałymi, zmiennymi globalnymi lub wywołaniami funkcji. Argumenty domyślne powinny być określone wraz z pierwszym wystąpieniem nazwy funkcji. W praktyce wartości argumentów domyślnych umieszcza się w prototypie funkcji. Na kolejnym wydruku przedstawiono wykorzystanie argumentów domniemanych. W pokazanym na wydruku 1.6 programie należy obliczyć

objętość (praktycznie jest to mnożenie trzech liczb). Prototyp funkcji obliczającej iloczyn trzech liczb ma postać:

```
void objetosc(int x = 1, int y =1, int z = 1);
```

Listing 1.6. Użycie argumentów domniemanych

```
1 #include <iostream.h>
  #include <conio.h>
3
  int main()
5 {
    objetosc();
7   objetosc(10);
    objetosc(10, 10);
9   objetosc(10, 10, 10);
       getch();
11     return 0;
  }
13 void objetosc(int x, int y, int z)
  {
15  int vol;
    vol = x * y * z;
17  cout << "x= " << x << " y= " << y << " z= " << z << endl;
    cout << "objetosc = " << vol << endl;
19 }
```

W funkcji main() wywoływana jest funkcja objetosc() z różną ilością argumentów. Pierwsze wywołanie ma postać:

```
objetosc();
```

Użytkownik nie określił argumentów, wszystkie trzy argumenty są domniemane, wobec tego w tym wywołaniu argumenty mają wartości:

```
x = 1;      y = 1;      z = 1;
```

W kolejnym wywołaniu:

```
objetosc(10);
```

pierwszy argument jest określony i ma wartość 10, pozostałe są domniemane wobec tego wartości argumentów są następujące:

```
x = 10; y = 1;      z = 1;
```

Ostatnie wywołanie przekazuje argumenty nie używając wartości domyślnych.

1.8. Przeciążanie funkcji

Załóżmy, że w programie musimy obliczać kwadraty liczb. Definicja funkcji obliczającą kwadrat liczby całkowitej może mieć postać:

```
int kwadrat ( int x ) { return x * x ; }
```

a wywołanie tej funkcji może mieć postać:

```
kwadrat( 3 )
```

Funkcja kwadrat() może obsłużyć jedynie liczby całkowite (zmienne typu int). Gdy chcemy obliczyć kwadrat liczby rzeczywistej (zmiennej typu double) w języku C musimy zdefiniować nową funkcję. W języki C++ istnieje specjalny mechanizm noszący nazwę przeciążanie funkcji, który pozwala na legalne wywołania typu:

```
kwadrat( 3 )
kwadrat( 3.3 )
```

Możemy zdefiniować dwie wersje funkcji kwadrat() i w zależności od typu przesyłanego argumentu (int lub double) C++ wybierze odpowiednią wersję funkcji. Oczywiście wersji funkcji może być więcej niż dwie. Przeciążanie funkcji jest proste; w naszym przypadku musimy tylko utworzyć dwie definicje funkcji. Program pokazany na wydruku 1.7. używa przeciążonej funkcji kwadrat() do obliczania kwadratu liczby typu int i typu double. Funkcje przeciążane są odróżniane przez swoje sygnatury – sygnatura jest połączeniem nazwy funkcji i typów jej parametrów.

Listing 1.7. Przeciążanie funkcji

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

int kwadrat ( int x ) { return x * x ; }
double kwadrat ( double y ) { return y * y ; }

int main()
{
   cout << "kwadrat liczby calkowitej     3 = "
     << kwadrat( 3 );
   cout << "\nkwadrat liczby rzeczywistej 3.3 = "
     << kwadrat ( 3.3);
```

```
    getch();
14  return 0;
}
```

1.9. Wzorce funkcji

W języku C++ mamy jeszcze jeden mechanizm pozwalający na użycie funkcji do operowania na różnych typach zmiennych – są to wzorce funkcji. Programista pisze pojedynczą definicję funkcji w oparciu o typy argumentów w wywoływanej funkcji, kompilator C++ automatycznie generuje oddzielne funkcje wzorcowe, aby poprawnie obsłużyć żądany typ danych. Program pokazany na wydruku 1.8. ilustruje użycie wzorca funkcji wieksza() do określenia większej z dwóch wartości typu int, typu double i typu char. W tworzonych szablonach (wzorcach) wszystkie definicje wzorców funkcji rozpoczynają się od słowa kluczowego template, po którym następuje lista formalnych typów parametrów do tego wzorca, ujęta w nawiasy ostre. Każdy parametr formalnego typu jest poprzedzony słowem kluczowym class. Definicja funkcji wzorcowej użytej w naszym przykładzie ma postać:

```
template <class T>
T wieksza ( T war1, T war2)
 {
  T war;
  war = (war1 > war2) ? war1 : war2;
  return war;
 }
```

Pokazany wzorzec funkcji deklaruje parametr typu formalnego T (nazwa jest dowolna) jako typ danych do sprawdzania przez funkcję wieksza(). Podczas kompilacji typ danych przekazywanych do funkcji wieksza() jest zastępowany przez konkretny typ za pomocą pełnej definicji wzorca, C++ tworzy kompletną funkcję w celu określenia większej z dwóch wartości konkretnego typu. Następnie, nowo utworzona funkcja jest kompilowana. Realizacja funkcji z żądanymi wartościami np. typu int ma postać:

```
int wieksza ( int war1, int war2)
 {
  int war;
  war = (war1 > war2) ? war1 : war2;
  return war;
 }
```

Podczas realizacji, typ uogólniony (w naszym przykładzie T) zostaje zastąpiony konkretnym typem (w naszym przykładzie int). Każdy parametr typu w definicji wzorca musi pojawić się na liście parametrów funkcji przynajmniej jeden raz.

Listing 1.8. Wzorce funkcji

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

template <class T>
T wieksza ( T war1, T war2)
 {
  T war;
  war = (war1 > war2) ? war1 : war2;
  return war;
 }

int main()
{
 int x1,y1;
 cout << "\npodaj dwie liczby calkowite: ";
 cin >> x1 >> y1;
 cout << "wieksza to: " << wieksza(x1,y1);
 double x2,y2;
 cout << "\npodaj dwie liczby rzeczywiste: ";
 cin >> x2 >> y2;
 cout << "wieksza to: " << wieksza(x2,y2);
 char x3,y3;
 cout << "\npodaj dwa znaki: ";
 cin >> x3 >> y3;
 cout << "wieksza to: " << wieksza(x3,y3);
   getch();
   return 0;
}
```

Po uruchomieniu programu mamy następujący komunikat:

```
podaj dwie liczby calkowite: 1  5
wieksza to:  5
podaj dwie liczby rzeczywiste: 1.1  5.5
wieksza to:  5.5
podaj dwa znaki:  a  z
wieksza to:  z
```

Wzorce stały się integralną częścią języka C++, są one elastyczne, a przy tym bezpieczne pod względem obsługi różnych typów.

1.10. Funkcje inline

Kolejnym nowym elementem w C++ są funkcje typu inline (rozwijalne, tzn. wplecione w kod programu). Funkcje rozwijalne mogą znacznie przyspieszyć wykonanie programu (ale w pewnych przypadkach wcale nie muszą). Dla definiowanej funkcji kompilator tworzy w pamięci osobny zestaw instrukcji. Podczas wywoływania funkcji wykonywany jest skok do tego zestawu. Gdy funkcja kończy działanie, wykonanie wraca do instrukcji następnej po instrukcji wywołania funkcji.

Listing 1.9.

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

inline int Kwadrat ( int ) ;

int main()
{
 int kw;
 cout << "\nPodaj liczbe: ";
 cin >> kw ;
 kw = Kwadrat(kw);
 cout << "Kwadrat tej liczby to " << kw;
   getch();
   return 0;
}

int Kwadrat ( int kw )
{
 return kw * kw;
}
```

Obsługa wywołania funkcji jest bardzo kosztowna – w szczególności potrzebny jest kod niezbędny do umieszczania wartości na stosie, wywołania funkcji, pobrania parametru ze stosu i zakończenia działania funkcji. Jeżeli funkcja zostanie zadeklarowana ze słowem kluczowym inline, kompilator nie tworzy prawdziwej funkcji tylko kopiuje kod z funkcji rozwijalnej bezpośrednio do kodu funkcji wywołującej (w miejscu wywołania funkcji inline).Nie jest wykonywany żaden skok, program działa tak, jakby w tym miejscu były instrukcje funkcji. Na wydruku 1.9 pokazano użycie funkcji typu inline. Funkcja Kwadrat() jest deklarowana jako funkcja typu inline, otrzymująca parametr typu int i zwracająca wartość typu int. Program kompiluje się do kodu, który ma postać taką, jakby w każdym miejscu wystąpienia instrukcji:

```
kw = Kwadrat ( kw ) ;
```

znajdowała się instrukcja:

```
kw = kw * kw;
```

Należy pamiętać, że gdy funkcja będzie wywoływana w kilku miejscach, jej kod będzie kopiowany w tych miejscach. Może to spowodować znaczny wzrost objętości pliku wykonywalnego, co w efekcie może spowolnić wykonywanie programu a nie zwiększenie szybkości jak można by się spodziewać. Należy zdawać sobie sprawę, że słowo kluczowe inline jest w rzeczywistości życzeniem postawionym kompilatorowi a nie jego dyrektywą. Oznacza to, że funkcja będzie rozwinięta, jeśli będzie to możliwe, gdy funkcji nie można rozwinąć tworzona jest zwyczajna wywoływana funkcja.

1.11. Dynamiczne zarządzanie pamięcią

Do przechowywania danych program potrzebuje odpowiedniej ilości pamięci. Przydzielanie pamięci może odbywać się automatycznie, gdy np. wystąpi deklaracja:

```
char nazw[ ] ="Jan Kowalski";
```

Taka deklaracja powoduje zarezerwowanie obszaru pamięci potrzebnego do przechowywania tablicy znaków "Jan Kowalski". Można również zarezerwować określoną ilość pamięci, tak jak w tej deklaracji:

```
int liczby[100];
```

Tego typu deklaracja powoduje zarezerwowanie dla tablicy liczby 100 jednostek pamięci, z których każda jest w stanie przechowywać wartość typu int. W języku C można rezerwować pamięć w trakcie działania programu. Służy do tego celu funkcja malloc(), która pobiera tylko jeden argument – ilość potrzebnej pamięci w bajtach. Znajduje ona odpowiedni obszar wolnej pamięci i zwraca adres jego pierwszego bajtu. Rozpatrzmy następujące instrukcje wykorzystujące malloc() do utworzenia tablicy:

```
double *wsk;
wsk = ( double * ) malloc ( 30 * sizeof(double ) );
```

Powyższy kod rezerwuje pamięć dla trzydziestu wartości typu double. Jak pamiętamy, nazwa tablicy jest adresem jej pierwszego elementu, stąd przypisanie wskaźnikowi wsk adresu pierwszej wartości double sprawia, że można z niego korzystać tak jak ze zwykłej nazwy tablicy wsk[]. Schemat postępowania jest następujący:

— deklarujemy wskaźnik
— wywołujemy funkcję malloc()
— odwołujemy się do elementów tablicy za pomocą nazwy wskaźnika

Ta metoda pozwala na tworzenie tablic dynamicznych, czyli takich, których rozmiar jest określany w trakcie działania programu. Taką możliwość ilustruje program pokazany na wydruku 1.10.

Listing 1.10. Dynamicznie alokowana pamięć

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
 double *wsk;
 int max, liczba;
 int i = 0;
 puts ("Podaj ilosc elementow: ");
 scanf("%d", &max);
 wsk = (double *) malloc ( max  * sizeof( double ));
 if (wsk ==NULL)      {
        puts("Blad przydzialu pamieci, koniec");
        exit(EXIT_FAILURE);
     }
 puts("Podaj elemnty tablicy. q konczy program");
 while (i < max && scanf("%lf", &wsk[i]) == 1)   ++i;
 printf("Lista %d elementow:\n", liczba = i);
 for (i=0; i<liczba; i++)      {
       printf("%7.2f ", wsk[i]);
       if (i % 7 == 6)
          putchar('\n');
     }
  if (i % 7 != 0)    putchar('\n');
  puts("Koniec");
  free(wsk);
    getch();
    return 0;
}
```

Przebieg działania programu może mieć postać:

```
Podaj ilosc elementow:
3
Podaj elementy tablicy. q konczy program
20 30 50 50 70 q
lista 3 elementow
    20.00     30.00     50.00
Koniec
```

Wpisaliśmy 5 liczb, ale program zaakceptował tylko 3 liczby. Spowodowane jest to faktem, że rozmiar tablicy został ustalony jako 3. Pobranie rozmiaru tablicy realizują instrukcje:

```
puts ("Podaj ilosc elementow: ");
scanf("%d", &max);
```

W instrukcji:

```
wsk = (double *) malloc(max * sizeof(double));
```

rezerwujemy miejsce w pamięci dla żądanej ilości elementów i przypisujemy adres bloku pamięci wskaźnikowi wsk. Gdy nie można przydzielić pamięci, funkcja malloc() zwraca wskaźnik zerowy i program kończy działanie:

```
if (wsk ==NULL)
    {
     puts("Blad przydzialu pamieci, koniec");
     exit(EXIT_FAILURE);
    }
```

Funkcja free() zwalnia zarezerwowaną pamięć. Funkcje malloc() i free() zarządzają razem pamięcią komputera. Dzięki tablicom dynamicznym wykorzystujemy optymalnie pamięć. Jeżeli wiemy, że program większość czasu będzie potrzebował tylko 100 elementów a od czasu do czasu albo tylko jeden raz będzie potrzebował 10000 elementów to należy stosować tablice dynamiczne. Bez możliwości korzystania z tablic dynamicznych należałoby utworzyć tablicę o rozmiarze 10000 elementów. Taka tablica cały czas zajmowałaby niepotrzebnie pamięć. Jest to zwykłe marnowanie pamięci.

Ponieważ dynamiczne przydzielanie pamięci jest istotne dla tworzenia wydajnych programów C++ oferuje dodatkowe, bardzo wygodne narzędzia jakim są operatory new (przydział) i delete (zwalnianie). Składnia wyrażeń z tymi operatorami ma postać:

```
new typ_obiektu
delete wskaznik_obiektu
```

Użycie operatora new ma dwie zalety w stosunku do funkcji malloc():

— nie wymaga użycia operatora sizeof
— operator new zwraca wskaźnik żądanego typu, nie ma potrzeby wykonywania konwersji typu wskaźnika

Instrukcja:

```
new unsigned short int
```

alokuje na stercie dwa bajty(ta ilość zależy od konkretnego kompilatora), a zwracaną wartością jest adres pamięci. Musi on zostać przypisany do wskaźnika. Aby stworzyć na stercie obiekt typu unsigned short, możemy użyć instrukcji:

```
unsigned short int *wsk;
wsk = new unsigned short int;
```

Można też zainicjalizować wskaźnik w trakcie tworzenia:

```
unsigned short int * wsk = new unsigned short int;
```

Tak określony wskaźnik zachowuje się jak każdy inny wskaźnik, wobec tego możemy np. przypisać obiektowi na stercie dowolną wartość:

```
*wsk = 99;
```

Ta instrukcja oznacza: „Umieść 99 jako wartość obiektu wskazywanego przez wsk” lub „Przypisz obszarowi sterty wskazywanemu przez wskaźnik wsk wartość 99”. Gdy operator new nie jest w stanie przydzielić pamięci, zgłasza wyjątek. Gdy przydzielona pamięć nie jest już potrzebna, należy ją zwolnić. Do tego celu należy użyć słowa kluczowego delete (ang. usuń) z właściwym wskaźnikiem. Instrukcja delete zwalnia pamięć zaalokowaną na stercie, ta pamięć może być wykorzystana ponownie do innych zadań. Należy pamiętać, że pamięć zaalokowana operatorem new nie jest zwalniana automatycznie, staje się niedostępna, taką sytuację nazywamy wyciekiem pamięci (ang. memory leak). Pamięć możemy zwrócić w następujący sposób:

```
delete wsk;
```

Najczęściej operatora new używa się do obsługi tablic. Dziesięcioelementowa tablica liczb całkowitych może być utworzona i przypisana w następujący sposób:

```
int *wsk = new int[ 10 ];
```

Zwolnienie tak przydzielonej pamięci można zrealizować za pomocą instrukcji:

```
delete [ ] wsk;
```

Jak widać obsługa dynamicznego przydziału pamięci dzięki operatorom jest bardzo wygodna. Należy wiedzieć, że operator delete zwalnia pamięć, na którą on wskazuje, wskaźnik nadal pozostaje wskaźnikiem. Ponowne wywołanie delete dla tego wskaźnika spowoduje załamanie programu. Zaleca się,

aby podczas zwalniania wskaźnika ustawić go na zero (NULL). Kompilator gwarantuje, że wywołanie delete z pustym wskaźnikiem jest bezpieczne. Program pokazany na wydruku 1.11 ilustruje zastosowanie operatorów new i delete. Pamięć jest dwukrotnie przydzielona i dwukrotnie zwalniana.

Listing 1.11. Dynamiczny przydział pamięci. Operatory new i delete

```
 1 #include <iostream.h>
   #include <conio.h>
 3
   int main()
 5 {
    int zmienna = 55;
 7  int *zm = &zmienna;
    int *sterta = new int;
 9  *sterta = 155;
    cout << "zmienna: " << zmienna << '\n';
11  cout << "*zm: " << *zm << '\n';
    cout << "*sterta: " << *sterta << '\n';
13  delete sterta;
    sterta = new int;
15  *sterta = 111;
    cout << "*sterta: " << *sterta << '\n';
17  delete sterta;
       getch();
19     return 0;
   }
```

Po uruchomieniu programu mamy następujący komunikat:

```
zmienna: 55
*zm: 55
*sterta: 155
*sterta: 111
```

W instrukcjach

```
int zmienna = 55;
int *zm = &zmienna;
```

program deklaruje i inicjalizuje lokalną zmienną wartością 55 oraz deklaruje i inicjalizuje wskaźnik, przypisując mu adres tej zmiennej. W instrukcji:

```
int *sterta = new int;
```

deklarowany jest wskaźnik sterta inicjalizowany wartością uzyskaną w wyniku wywołania operatora new. Powoduje to zaalokowanie na stercie miejsca dla wartości typu int. W instrukcji:

```
*sterta = 155;
```

przypisano wartość 155 do nowo zaalokowanej pamięci.
Instrukcje:

```
cout << "zmienna: " << zmienna << '\n';
cout << "*zm: " << *zm << '\n';
cout << "*sterta: " << *sterta << '\n';
```

wypisują odpowiednie wartości. Wydruki wskazują, że rzeczywiście mamy dostęp do zaalokowanej pamięci. W kolejnej instrukcji:

```
delete sterta;
```

pamięć zaalokowana jest zwracana na stertę. Czynność ta zwalnia pamięć i odłącza od niej wskaźnik. Teraz sterta może wskazywać na inny obszar pamięci. Nowy adres i wartość przypisana jest w kolejnych instrukcjach:

```
sterta = new int;
*sterta = 111;
```

a w końcu wypisujemy wynik i zwalniamy pamięć:

```
cout << "*sterta: " << *sterta << '\n';
delete sterta;
```

Wspominaliśmy już o zjawisku wycieku pamięci. Taka sytuacja może nastąpić, gdy ponownie przypiszemy wskaźnikowi adres bez wcześniejszego zwolnienia pamięci, na którą on wskazuje. Rozważmy fragment programu:

```
int * wsk = new int;
*wsk = 111;
wsk = new int;  //error
wsk = 999;
```

Na początku tworzymy wskaźnik wsk i przypisujemy mu adres rezerwowanego na stercie obszaru pamięci. W tym obszarze umieszczamy wartość 111. W trzeciej instrukcji przypisujemy wskaźnikowi wsk adres innego obszaru pamięci a czwarta instrukcja umieszcza w tym obszarze wartość 999. Nie ma sposobu na odzyskanie pierwszego obszaru pamięci. Poprawna postać tego kodu może być następująca:

```
int * wsk = new int;
*wsk = 111;
delete wsk;
wsk = new int;
wsk = 999;
```

Zaleca się, aby za każdym razem, gdy użyto słowa kluczowego new, powinno się użyć słowa kluczowego delete.

1.12. Słowa kluczowe języka C++

Obecnie (rok 2010, cytujemy B. Strostrupa) w standardzie języka C++ zdefiniowano 74 słowa kluczowe. Słowa kluczowe są identyfikatorami wykorzystywanymi w programach do nazywania konstrukcji języka C++. Listę słów kluczowych umieszczono tabeli 1.1.Pamietamy, że słowa kluczowe pisane są zawsze małą literą. Nie wolno ich używać do innych celów niż do nazywania konstrukcji języka.

Tabela 1.1. Słowa kluczowe języka C++

bool	break	case	catch	char
class	compl	const	const_cas	continue
default	delete	do	double	dynamic_cast
else	enum	explicit	export	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	not	not_eq	operator
or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t
while	xor	xor_eq		

wracamy uwagę, że w standardzie ANSI/ISO C++ do języka C++ dodano nowy typ – bool. Zmienna typu bool (zmienna logiczna) może przyjmować jedną z dwóch wartości : true (prawda) lub false (fałsz). Pierwotnie język C++ tak jak i język C nie posiadał typu logicznego. Jak pamiętamy, wartość zero interpretowana była jako fałsz, każda wartość różna od zera była interpretowana jako prawda. Postać instrukcji z typem bool może mieć postać:

```
int cena = true;
int odpowiedz = false;
```

1.13. Schemat programu w języku C++

Ogólna postać programu w języku C++ nie różni się zbytnio od ogólnego schematu programu napisanego w języku C i zazwyczaj konstruowana jest następująco:

— nagłówek
— deklaracje klas bazowych
— deklaracje klas pochodnych
— prototypy funkcji ogólnych
— prototypy funkcji
— ciało funkcji main()
— definicje funkcji ogólnych

Należy zauważyć, że z powodu żądania kompatybilności wstecznej, język C++ jako nadzbiór języka C musi zawierać biblioteki tego języka. Ponadto rozwój języka C++ spowodował stworzenie nowej biblioteki standardowej. Prowadzi to do pewnego zamieszania ponieważ w programach języka C++ możemy używać jednocześnie aż trzech różnych bibliotek. Stosując różne biblioteki musimy nadawać plikom nagłówkowym różne nazwy. Stosowane konwencje pokazane są w tabeli 1.2

Tabela 1.2. Konwencje nazw bibliotek języka C++

biblioteka	nazewnictwo	przykład
Język C	<nazwa_pliku.h>	<stdio.h>
Język C++, stara wersja	<nazwa_pliku.h>	<iostream.h>
Język C++, nowa wersja	<nazwa_pliku>	<iostream>

Jest jeszcze jedna istotna różnica, gdy chcemy stosować nową bibliotekę języka C++ (zamiast np. pliku iostream.h, włączamy plik nagłówkowy iostream). Stosowanie nowej biblioteki wymaga udostępnienia dyrektywy przestrzeni nazw, aby udostępnić elementy np. strumienia iostream w programach:

```
using namespace std:
```

Dzięki dyrektywie using włączamy standardową przestrzeń nazw i możemy bez kłopotu używać zmiennych cout czy cin. Można pominąć dyrektywę using, ale wtedy aby korzystać z elementów nowej biblioteki iostream musimy napisać wprost typ przestrzeni nazw, tak jak w następującym przykładzie:

```
std :: cout << "kolosalne zamieszanie" << std:: endl;
```

Ponieważ przy każdym wystąpieniu cout, cin itp. należy pisać typ przestrzeni nazw, w praktyce zawsze używamy dyrektywy using namespace std.

Poniżej pokazujemy programy wykorzystujące starą bibliotekę oraz z nową

```
//Program C++, stara biblioteka
#inlcude <iostream.h>
int main()
{
   return 0;
}
```

```
//Program C++, nowa biblioteka
#inlcude <iostream>
using namespace std;
int main()
{
   return 0;
}
```

Rozdział 2

Wprowadzenie do programowania obiektowego

2.1. Wstęp **28**
2.2. Paradygmaty programowania **28**
2.3. Obiekty i klasy **30**
2.4. Hermetyzacja danych **34**
2.5. Dziedziczenie **35**
2.6. Polimorfizm **36**
2.7. Podsumowanie terminologii **37**
2.8. Środowisko programistyczne **38**

2.1. Wstęp

Ponad trzydzieści lat temu (w latach osiemdziesiątych) Bjarne Stroustrup tworzy język C++ jako ulepszona wersje języka C. Zasadnicza nowością języka C++ było:
— eliminacja istotnych wad języka C
— Wprowadzenie obsługi obiektów i możliwość
— realizacji programowania obiektowego

Programowanie obiektowe było nowością w latach osiemdziesiątych, stosunkowo niedawno powstały propozycje teoretyczne nowego paradygmatu programowania i powstał pierwszy język realizujący ten paradygmat - język Smalltalk. W programowaniu obiektowym analizuje się zadanie do rozwiązania a następnie opracowuje się reprezentacje tego problemu w postaci klasy. Konstrukcja klasy (wywodząca się pierwotnie ze struktury języka C) umożliwiła realizacje paradygmatu programowania obiektowego i jest centralnym elementem programów obiektowych pisanych w języku C++.

2.2. Paradygmaty programowania

Mamy wiele metod programowania w języku C++, wydaje się jednak, że decydujące znaczenie mają trzy paradygmaty programowania:
— Programowanie proceduralne
— Programowanie strukturalne
— Programowanie zorientowane obiektowo

Powstanie i rozwój języków i stylów programowania ma swój początek w pracach matematyków z pierwszej połowy XX wieku. Wprowadzono wtedy pojęcie funkcji obliczalnej, oznaczające funkcję, której wartość dla dowolnych wskazanych argumentów można obliczyć w sposób efektywny (w skończonym czasie). Program w języku proceduralnym jest zestawem instrukcji definiujących algorytm działania. Do grupy języków proceduralnych zaliczają się: assemblery oraz języki wysokiego poziomu, takie jak Fortran, Basic, Pascal, język C. Każdy algorytm składa się z opisu podmiotów oraz opisu czynności, które mają być na tych podmiotach wykonane. W zapisanym w języku proceduralnym programie opisom tym odpowiadają, odpowiednio: dane (struktury danych) i instrukcje.

Modyfikując stylu programowania proceduralnego wypracowano koncepcję tzw. programowania strukturalnego. Zakłada ona grupowanie fragmentów kodu w podprogramy (procedury i funkcje) i definiowanie zasad komunikowania się tych podprogramów między sobą. Istotne zalety programowania strukturalnego polegały na poprawie czytelności programu i możliwości tworzenia i użytkowania bibliotek podprogramów. Styl programowania

strukturalnego doskonale nadawał się do pisania małych i średnio dużych programów przenośnych, nie sprawdzał się (był kosztowny i generował dużo błędów) przy realizacji dużych i bardzo zaawansowanych programów. Coraz większe wymagania stawiane programistom spowodowały powstanie nowego stylu programowania – programowania obiektowego.

Na jego gruncie wyrosła szeroka klasa języków obiektowych: tak specjalizowanych, np. O++, jak i ogólnego przeznaczenia, np. C++, Object Pascal, Java. W programowaniu proceduralnym i strukturalnym program jest traktowany jako seria procedur (funkcji) działających na danych. Dane są całkowicie odseparowane od procedur, programista musi pamiętać, które funkcje były wywołane i jakie dane zostały zmienione. Przy realizacji dużych programów, dominuje programowanie obiektowe. Programowanie zorientowane obiektowo dostarcza technik zarządzania złożonymi elementami, umożliwia ponowne wykorzystanie komponentów i łączy w logiczną całość dane oraz manipulujące nimi funkcje. Zadaniem programowania zorientowanego obiektowo jest modelowanie „obiektów” a nie danych. Język C++ został stworzony dla programowania zorientowanego obiektowo. Zasadniczymi elementami stylu programowania obiektowego są:
— klasy (abstrakcyjne typy danych),
— hermetyzacja danych (kapsułkowanie),
— dziedziczenie,
— polimorfizm.

Należy pamiętać, że język C++ nie jest czystym językiem programowania obiektowego. Swoją popularność język C++ zawdzięcza przede wszystkim zgodności z językiem C. W języku C++ można programować obiektowo, ale także i strukturalnie. Jednym z najlepszych języków zorientowanych obiektowo jest Smalltalk, rozwinięty we wczesnych latach 70-tych w Palo Alto Research Center amerykańskiej firmy Xerox. Smalltalk jest idealnie zaprojektowanym obiektowo językiem – dosłownie wszystko jest obiektem. Niestety język Smalltalk nie zdobył takiej popularności, jaką ma język C++. Język C++ jest językiem hybrydowym – umożliwia programowanie albo w stylu strukturalnym (podobnym do C) albo w stylu zorientowanym obiektowo. Decyzja o wykorzystaniu języka C++ jako języka programowania obiektowego należy do programisty. Bardzo trudno jest zdefiniować precyzyjnie pojęcie „język obiektowy”, a także ustalić czy dany jest językiem zorientowanym obiektowo czy nie jest. Bruce Eckel w podręczniku „Thinking in C++” przytoczył pięć podstawowych cech języka Smalltalk (był to jeden z pierwszych języków obiektowych).

1. Wszystko jest obiektem. Obiekt jest szczególnym rodzajem zmiennej, przechowuje ona dane, ale może także wykonać operacje na sobie samej
2. Program jest grupą obiektów, przekazujących sobie wzajemnie informacje o tym, co należy zrobić za pomocą komunikatów. Komunikat może

być traktowany jako żądanie wywołania funkcji należącej do tego obiektu.
3. Każdy obiekt posiada własną pamięć, złożoną z innych obiektów. Oznacza to, że nowy rodzaj obiektu jest tworzony przez utworzenie pakietu złożonego z już istniejących obiektów.
4. Każdy obiekt posiada typ. Mówimy, że każdy obiekt jest egzemplarzem jakiejś klasy. Klasa jest synonimem słowa typ.
5. Wszystkie obiekty określonego typu mogą odbierać te same komunikaty.

Język C++ ma wszystkie cechy, które dla wielu specjalistów są charakterystyczne dla języka programowania obiektowego.

Większość zrealizowanych dużych komercyjnych projektów oprogramowania albo działa zawodnie, albo nie tak jak powinno. Są one zbyt drogie i za mało niezawodne. Jest to rezultatem ogromnej złożoności programów, które są obecnie tworzone. Ocenia się, że prom kosmiczny NASA, samolot Boeing 747 i system operacyjny Windows są w historii człowieka najbardziej skomplikowanymi wynalazkami. Widzimy, że nowoczesne programy zaliczane są do najbardziej skomplikowanych rzeczy, jakie wymyślił człowiek. Wielu specjalistów widzi w programowaniu obiektowym nadzieję na tworzenie produktów, które będą działać tak jak powinny.

2.3. Obiekty i klasy

Kluczowe znaczenie w technologii programowania obiektowego mają obiekty i klasy. Wydaje się, że człowiek postrzega świat jako zbiór obiektów. Oglądając obraz na ekranie monitora, widzimy np. raczej twarz aktora niż zbiór kolorowych pikseli. Aktor jest obiektem, który ma odpowiednie cechy i może wykonywać różnorodne akcje. Innym przykładem obiektu jest np. samochód. Również samochód ma odpowiednie cechy i może wykonywać różne działania. Otaczająca nas rzeczywistość złożona jest z obiektów. Obiekty mają atrybuty takie jak np. wielkość, kolor, kształt. Obiekty mogą działać, np. aktor biegnie samochód hamuje, itp. Obserwujemy i rozumiemy świat badając cechy obiektów i ich zachowanie. Różne obiekty mogą mieć podobne atrybuty i podobne działania. Np. aktor ma wagę, tak samo jak samochód, aktor może się poruszać z odpowiednią prędkością, podobnie samochód. Obiekty takie jak człowiek czy samochód w istocie są bardzo skomplikowanymi i złożonymi obiektami, jest prawie niemożliwe dokładne ich opisanie. Abstrakcja umożliwia nam zredukowanie złożoności problemu. Właściwości (cechy) i procesy (działania, akcje) zostają zredukowane do niezbędnych cech i akcji zgodnie z celami, jakie chcemy osiągnąć. Inaczej będzie przeprowadzona redukcja cech i działań samochodu, gdy chcemy go sprzedawać (istotny wtedy jest typ samochodu, producent, kolor, cena, moc

silnika, itp.) a inaczej będzie wyglądał proces abstrakcji, gdy zechcemy modelować opór powietrza samochodu (istotne są kształty, rozmiary np. lusterek bocznych, kąt pochylenia szyby przedniej, itp.). Dzięki abstrakcji uzyskujemy możliwość rozsądnego opisania wybranych obiektów i zarządzania złożonymi systemami. Różne obiekty mogą mieć podobne atrybuty. Obiekty takie jak samochód, samolot i okręt mają wiele wspólnego (mają wagę, przewożą pasażerów, bilet na dany pojazd ma cenę, itp.). Obiekt nie musi reprezentować realnie istniejącego bytu. Może być całkowicie abstrakcyjny lub reprezentować jakiś proces. Obiekt może reprezentować rozdawanie kart do pokera lub mecz piłki nożnej. W tym ostatnim przykładzie atrybutami obiektu mogą być nazwy drużyn, ilość strzelonych goli, nazwiska trenerów, itp.

W języku C/C++ istnieje typ danych zwany strukturą, który umożliwia łączenie różnych typów danych. Wprowadzenie do języków programowania struktury, było milowym krokiem w rozwoju języków programowania, ponieważ umożliwiło posługiwanie się zestawami różnych cech przy pomocy jednego typu danych. Struktura stanowi jeden ze sposobów zastosowania abstrakcji w programach. Np. struktura pracownik może mieć postać:

```
struct pracownik  {
    char nazwisko[MAXN];
    char imie{MAXI}
    int rok_urodzenia;
};
```

Deklaracja ta opisuje strukturę złożoną z dwóch tablic i jednej zmiennej typu int. Nie tworzy ona rzeczywistego obiektu w pamięci, a jedynie określa, z czego składa się taki obiekt. Opcjonalna etykieta (znacznik, ang. structure tag) pracownik jest nazwą przyporządkowaną strukturze. Ta nazwa jest wykorzystywana przy deklaracji zmiennej:

```
struct pracownik kierownik;
```

Deklaracja ta stwierdza, że kierownik jest zmienną strukturalną o budowie pracownik. Nazwy zdefiniowane wewnątrz nawiasów klamrowych definicji struktury są składowymi struktury (elementami struktury, polami struktury). Definicja struktury pracownik zawiera trzy składowe, dwie typu char – nazwisko i imie oraz jedna typu int – rok_urodzenia. Dostęp do składowych struktur jest możliwy dzięki operatorom dostępu do składowych. Mamy dwa takie operatory: operator kropki (.) oraz operator strzałki (->). Za pomocą operatora kropki możliwy jest dostęp do składowych struktury przez nazwę lub referencję do obiektu. Np. aby wydrukować składową rok_urodzenia możemy posłużyć się wyrażeniem:

```
cout << kierownik.rok_urodzenia;
```

Składowe tej samej struktury muszą mieć różne nazwy, jednak dwie różne struktury mogą zawierać składowe o tej samej nazwie. Składowe struktury mogą być dowolnego typu. Struktury danych umożliwiają przechowywanie cech (właściwości, atrybutów) obiektów. Ale jak wiemy z obiektem związane są również najróżniejsze działania. Działania te tworzą interfejs obiektu. W języku C nie ma możliwości umieszczenia w strukturze atrybutów obiektu i operacji, jakich można na obiekcie wykonać. W języku C++ wprowadzono nowy abstrakcyjny typ danych, który umożliwia przechowywanie atrybutów i operacji. Tworzenie abstrakcyjnych typów danych odbywa się w języku C++ (tak samo jak w innych językach programowania obiektowego) za pomocą klas. Klasa stanowi implementację abstrakcyjnego typu danych.

Na każdy projektowany obiekt składają się dane (atrybuty) i dobrze określone operacje (działania). Do danych nie można dotrzeć bezpośrednio, należy do tego celu wywołać odpowiednią metodę. Dzięki temu chronimy dane przed niepowołanym dostępem. Komunikacja użytkownika z obiektem (a także komunikacja wybranego obiektu z innym obiektem) zaczyna się od wysłania do niego odpowiedniego żądania (ang. request). Po odebraniu żądania (inaczej komunikatu) obiekt reaguje wywołaniem odpowiedniej metody lub wysyła komunikat, że nie może żądania obsłużyć.

Klasa opisuje obiekt. Z formalnego punktu widzenia klasa stanowi typ. W programie możemy tworzyć zmienne typu określonego przez klasę. Zmienne te nazywamy instancjami (wcieleniami). Instancje stanowią realizację obiektów opisanych przez klasę.

Jak już wiemy, operacje wykonywane na obiektach noszą nazwę metod. Aby wykonać konkretną operację, obiekt musi otrzymać komunikat, który przetwarzany jest przez odpowiednią metodę. Cechą charakterystyczną programów obiektowych jest przesyłanie komunikatów pomiędzy obiektami.

Ponieważ język C++ jest w istocie językiem hybrydowym, panuje pewne zamieszanie w stosowanej terminologii. Metody noszą nazwę funkcji (procedury i podprogramy w języku C++ noszą też nazwy funkcji), a instancje nazywana są obiektami konkretnymi. W specyfikacji języka C++ pojęcie instancji nie jest w ogóle używane. Dla określania instancji klasy używa się po prostu terminu obiekt. Również istnieje duża różnorodność, jeżeli chodzi o terminologię stosowaną w opisie elementów klasy. Mamy takie terminy jak: elementy, składowe, atrybuty, dane. Operacje nazywane są metodami, funkcjami składowymi lub po prostu funkcjami.

W języku C++ klasa jest strukturą, której składowe mogą także być funkcjami (metodami).

Deklaracja klasy precyzuje, jakie dane i funkcje publiczne są z nią zwią-

zane, czyli do jakich danych ma dostęp użytkownik klasy. Deklaracja klasy punkt może mieć postać:

```
class punkt
{
private:
 int x;
 int y;
public:
 void init (int, int) ;
 void przesun (int, int)
};
```

Do deklarowania klasy służy słowo kluczowe class. Po nim podajemy nazwę tworzonej klasy, a następnie w nawiasach klamrowych umieszczamy zmienne wewnętrzne (dane) i metody (funkcje składowe). Deklarację kończy się średnikiem. W tym przykładzie klasa punkt zawiera dwie prywatne dane składowe x i y oraz dwie publiczne funkcje składowe init() i przesun(). Deklaracja tej klasy nie rezerwuje pamięci na nią. Mówi ona kompilatorowi, co to jest punkt, jakie dane zawiera i co może wykonać. Obiekt nowego typu definiuje się tak, jak każdą inną zmienną, np. typu int:

```
int radian; //definicja int
punkt srodek;   //definicja punktu
```

W tym przykładzie definiujemy zmienną o nazwie radian typu int oraz srodek , który jest typu punkt. Definicja klasy składa się z definicji wszystkich funkcji składowych. Definiując funkcje składowe podajemy nazwę klasy bazowej przy użyciu operatora zakresu(::). Definicja funkcji składowej init() może mieć postać:

```
void punkt :: init (int xp, init yp)
{
    x = xp;
    y = yp;
}
```

W tej definicji x i y reprezentują dane składowe x i y obiektu klasy punkt. Aby skorzystać z klasy punkt, powinniśmy zadeklarować obiekty klasy punkt:

```
punkt p1, p2;
```

Zostały utworzone dwa obiekty klasy punkt. Dostęp do publicznej funkcji składowej init() uzyskujemy przy pomocy operatora kropki:

```
p1.init( 10,10 );
```

Została wywołana publiczna funkcja składowa init() klasy, do której należy obiekt p1, to znaczy do klasy punkt.

2.4. Hermetyzacja danych

Hermetyzacja (ang. encapsulation) oznacza połączenie danych i instrukcji programu w jednostkę programową, jakim jest obiekt. Hermetyzacja obejmuje interfejs i definicję klasy. Podstawową zaletą hermetyzacji jest możliwość zabezpieczenia danych przed równoczesnym dostępem ze strony różnych fragmentów kodu programowego. W tym celu wszystkie dane (pola w obiekcie, atrybuty) i zapisy instrukcji (metody w obiekcie, funkcje składowe) dzieli się na ogólnodostępne (interfejs obiektowy) i wewnętrzne (implementacja obiektu). Dostęp do pól i metod wewnętrznych jest możliwy tylko za pośrednictwem "łącza obiektowego" - pól i metod ogólnodostępnych. Wybrane pola i metody można więc ukryć przed określonymi (w tym - wszystkimi) obiektami zewnętrznymi. Hermetyzacja umożliwia separację interfejsu od implementacji klasy. Jak pamiętamy, w klasycznej strukturze danych w języku C mamy swobodny dostęp do składowych struktury. Oznacza to, że z dowolnego miejsca w programie możemy dokonać zmiany tych danych. Nie jest to dobra cecha. Zastosowanie takiego modelu dostępu do danych obiektu grozi wystąpieniem niespójności danych, co prowadzi zwykle do generacji błędów. W języku C++ rozwiązano problem panowania nad dostępem do danych przy pomocy hermetyzacji danych. W literaturze przedmiotu spotkamy się z zamiennie stosowanymi terminami takimi jak: ukrywanie danych, kapsułkowanie czy zgoła całkiem egzotycznym terminem enkapsulacja.

W celu uniemożliwienia programistom wykonywania niekontrolowanych operacji na danych obiektu silnie ograniczono dostęp do jego składowych za pomocą dobrze zdefiniowanego interfejsu. Programista może na obiekcie wykonać tylko te operacja, na które pozwolił mu projektant klasy. W języku C++ dostęp do składowych klasy kontrolowany jest przez trzy specyfikatory:

— private
— public
— protected

Składowe zadeklarowane po słowie kluczowym private nie są dostępne dla programisty aplikacji. Posiada on dostęp do składowych oraz może wywoływać funkcje składowe zadeklarowane po słowie kluczowym public. Dostęp do atrybutów obiektu z reguły jest możliwy za pośrednictwem funkcji. Jeżeli definiując klasę pominiemy specyfikator dostępu, to żadna składowa klasy nie będzie dostępna na zewnątrz obiektu, domyślnie przyjmowa-

ny jest sposób dostępu określony specyfikatorem private. Hermetyzacja ma ogromne znaczenie dla przenośności programów i optymalizowania nakładów potrzebnych na ich modyfikacje. Wpływa także dodatnio na osiąganie niezawodności w projektach programistycznych.

2.5. Dziedziczenie

Jedna z najistotniejszych cech programowania zorientowanego obiektowo jest dziedziczenie (ang. inheritance). Mechanizm dziedziczenia służy w językach obiektowych do odwzorowania występujących często w naturze powiązań typu generalizacja - specjalizacja. Umożliwia programiście definiowanie potomków istniejących obiektów. Każdy potomek dziedziczy przy tym (wszystkie lub wybrane) pola i metody obiektu bazowego, lecz dodatkowo uzyskuje pewne pola i własności unikatowe, nadające mu nowy charakter. Typ takiego obiektu potomnego może stać się z kolei typem bazowym do zdefiniowania kolejnego typu potomnego. Bjarne Stroustrup w podręczniku „Język C++” rozważając zależności występujące pomiędzy klasą figura i klasą okrąg, tak zdefiniował paradygmat programowania obiektowego:
— zdecyduj, jakie chcesz mieć klasy;
— dla każdej klasy dostarcz pełny zbiór operacji;
— korzystając z mechanizmu dziedziczenia, jawnie wskaż to, co jest wspólne

Typowe aplikacje operują na wielu obiektach, programiści zmuszeni są do projektowania wielu klas, które powstają dzięki dużemu nakładowi czasu i kosztów. Stosunkowo szybko zorientowano się, że nie zawsze trzeba projektować klasę od początku, można wykorzystać istniejące już inne, przetestowane i sprawdzone klasy. W praktyce okazało się także, że wiele klas ma zazwyczaj kilka wspólnych cech. Naturalnym jest więc dążenie, aby analizując podobne klasy wyodrębnić wszystkie wspólne cechy i stworzyć uogólniona klasę, która będzie zwierać tylko te atrybuty i metody, które są wspólne dla rozważanych klas. W językach obiektowych taką uogólnioną klasę nazywamy superklasą lub klasą rodzicielską, a każda z analizowanych klas nazywa się podklasą lub klasą potomną.

W języku C++ wprowadzono koncepcje klasy bazowej (ang. base class) i klasy pochodnej (ang. derived class). Klasa bazowa (klasa podstawowa) zawiera tylko te elementy składowe, które są wspólne dla wyprowadzonych z niej klas pochodnych. Podczas tworzenia nowej klasy, zamiast pisania całkowicie nowych danych oraz metod, programista może określić, że nowa klas odziedziczy je z pewnej, uprzednio zdefiniowanej klasy podstawowej. Każda taka klasa może w przyszłości stać się klasą podstawową. W przypadku dziedziczenia jednokrotnego klasa tworzona jest na podstawie jednej klasy podstawowej. W sytuacji, gdy nowa klasa tworzona jest w oparciu o wiele

klas podstawowych mówimy o dziedziczeniu wielokrotnym. W języku C++ projektując nową klasę pochodną mamy następujące możliwości:
— w klasie pochodnej można dodawać nowe zmienne i metody
— w klasie pochodnej można redefiniować metody klasy bazowej

W języku C++ możliwe są trzy rodzaje dziedziczenia: publiczne, chronione oraz prywatne. Klasy pochodne nie mają dostępu do tych składowych klasy podstawowej, które zostały zadeklarowane jako private. Oczywiście klasa pochodna może się posługiwać tymi składowymi, które zostały zadeklarowane jako public lub protected. Jeżeli chcemy aby jakieś składowe klasy podstawowej były niedostępne dla klasy pochodnej, to deklarujemy je jako private. Z takich prywatnych danych, klasa pochodna może korzystać wyłącznie za pomocą funkcji dostępu znajdujących się w publicznym oraz zabezpieczonym interfejsie klasy podstawowej.

Mechanizm dziedziczenia obiektów ma znaczenie dla optymalizowania nakładów pracy potrzebnych na powstanie programu (optymalizacja kodu, możliwość zrównoleglenia prac nad fragmentami kodu programowego) i jego późniejsze modyfikacje (przeprowadzenie zmian w klasie obiektów wymaga przeprogramowania samego obiektu bazowego).

2.6. Polimorfizm

Drugą istotną cechą (obok dziedziczenia) programowania zorientowanego obiektowo jest polimorfizm (ang. polimorphism). Słowo polimorfizm oznacza dosłownie wiele form. Polimorfizm, stanowiący uzupełnienie dziedziczenia sprawia, że możliwe jest pisanie kodu, który w przyszłości będzie wykorzystywany w warunkach nie dających się jeszcze przewidzieć. Mechanizm polimorfizmu wykorzystuje się też do realizacji pewnych metod w trybie nakazowym, abstrahującym od szczegółowego typu obiektu. Zachowanie polimorficzne obiektu zależy od jego pozycji w hierarchii dziedziczenia. Jeśli dwa lub więcej obiektów mają ten sam interfejs, ale zachowują się w odmienny sposób, są polimorficzne. Jest to bardzo istotna cecha języków obiektowych, gdyż pozwala na zróżnicowanie działania tej samej funkcji w zależności od rodzaju obiektu. Zagadnienie polimorfizmu jest trudne pojęciowo, w klasycznych podręcznikach programowania jest omawiane zazwyczaj razem z funkcjami wirtualnymi (ang. virtual function). Przy zastosowaniu funkcji wirtualnych i polimorfizmu możliwe jest zaprojektowanie aplikacji, która może być w przyszłości prosto rozbudowywana. Jako przykład rozważmy zbiór klas modelujących figury geometryczne takie jak koło, trójkąt, prostokąt, kwadrat, itp. Wszystkie są klasami pochodnymi klasy bazowej figura. Każda z klas pochodnych ma możliwość narysowania siebie, dzięki metodzie rysuj. Ponieważ mamy różne figury, funkcja jest inna w

każdej klasie pochodnej. Z drugiej strony, ponieważ mamy klasę bazową figura, to dobrze by było, aby niezależnie, jaką figurę rysujemy, powinniśmy mieć możliwość wywołania metody rysuj klasy bazowej figura i pozwolić programowi dynamicznie określić, którą z funkcji rysuj z klas pochodnych ma zastosować. Język C++ dostarcza narzędzi umożliwiających stosowanie takiej koncepcji. W tym celu należy zadeklarować metodę rysuj w klasie bazowej, jako funkcję wirtualną. Funkcja wirtualna może mieć następującą deklarację:

```
virtual void rysuj () const ;
```

i powinna być umieszczona w klasie bazowej figura. Powyższy prototyp deklaruje funkcję rysuj jako stałą, nie zawierającą argumentów, nie zwracającą żadnej wartości i wirtualną. Jeżeli funkcja rysuj została zadeklarowana w klasie bazowej jako wirtualna, to gdy następnie zastosujemy wskaźnik w klasie bazowej lub referencję do obiektu w klasie pochodnej i wywołamy funkcje rysuj stosując ten wskaźnik to program powinien wybrać dynamicznie właściwą funkcje rysuj.

Polimorfizm umożliwia tworzenie w typach potomnych tzw. metod wirtualnych, nazywających się identycznie jak w typach bazowych, lecz różniących się od swych odpowiedników pod względem znaczeniowym.

2.7. Podsumowanie terminologii

W powyższych rozważaniach wprowadziliśmy dużo nowych terminów związanych z programowaniem zorientowanym obiektowo. Wydaje się celowe sporządzenie krótkiego spisu nowych terminów z krótkimi objaśnieniami.

- Atrybuty. Są to dane klasy, inaczej składowe klasy, które opisują bieżący stan obiektu. Atrybuty powinny być ukryte przed użytkownikami obiektu, a dostęp do nich powinien być określony i zdefiniowany w interfejsie.
- Obiekt. Obiekt jest bytem istnieje i może być opisany. Obiekt charakteryzują atrybuty i operacje. Obiekt jest instancją klasy (egzemplarzem klasy).
- Klasa. Klasa jest formalnie w języku C++ typem. Określa ona cechy i zachowanie obiektu. Klasa jest jedynie opisem obiektu. Należy traktować klasę jako szablon do tworzenia obiektów.
- Dziedziczenie. Dziedziczenie jest rodzajem relacji pomiędzy klasami. Klasy bardziej wyspecjalizowane dziedziczą po klasach ogólniejszych.
- Hermetyzacja. Hermetyzacja obejmuje interfejs i abstrakcję klasy, Hermetyzacja polega na ukrywaniu danych i ścisłej kontroli dostępu do pól i metod.

— Polimorfizm. Dosłownie oznacza wiele form. Dwa obiekty są polimorficzne, jeżeli mają ten sam interfejs, ale zachowują się w odmienny sposób.
— Interfejs. Jest to widoczna funkcjonalność klasy. Interfejs tworzy pomost pomiędzy obiektem i użytkownikiem obiektu. Użytkownicy posługują się obiektami poprzez interfejs.
— Implementacja. Jest to wewnętrzna funkcjonalność i atrybuty klasy. Implementacja klasy jest ukryta przed użytkownikiem klasy. Użytkownicy operują obiektami, poprzez interfejs, nie muszą wiedzieć jak obiekt jest implementowany.

2.8. Środowisko programistyczne

Podczas nauki podstaw języka C++ bardzo często ilustruje się elementy języka na przykładzie prostych programów. Aby programować w języku C++ potrzebny jest edytor, kompilator i konsolidator. Kompilator kupuje się wraz z konsolidatorem i zbiorem funkcji bibliotecznych. Istnieje wiele wersji kompilatorów języka C++. Niektóre kompilatory języka C++ sprzedawane są razem z tzw. zintegrowanym środowiskiem do tworzenia aplikacji (ang. integrated development environment - IDE). Jest to bardzo wygodne narzędzie. Przykładem takiego kompilatora jest pakiet Borland C++ (niestety firma Borland zawiesiła działalność). Bardzo wiele programów uruchamianych jest w środowisku Windows. Pisanie przyjaznych dla użytkownika programów (takie programy charakteryzują się dużą ilością okienek i ikon) jest zagadnieniem dość skomplikowanym i uciążliwym technicznie. Aby usprawnić proces pisania programów dla środowiska Windows powstały systemy błyskawicznego projektowania aplikacji (ang. RAD – Rapid Application Development).

Centralne miejsce w RAD zajmuje edytor kodu. W okienku edytora wygenerowany jest szkielet programu, możemy go przystosować do naszych celów. W szkielecie znajdują się zazwyczaj specyficzne dla danego RAD dyrektywy (tak jak pokazano na przykładzie z RAD Borland Bulider 6)

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main( int argc, char* argv[])
{
    return 0;
}
//---------------------------------------------------------------------------
```

Jest to poprawny program komputerowy, który nie robi nic. W programie mamy trzy dyrektywy preprocesora.

Dyrektywa:

```
#include <vcl.h>
```

nakazuje dołączenie pliku nagłówkowego vcl.h. Przy pomocy dyrektywy pragma:

```
#pragma hdrstop
#pragma argsused
```

przekazujemy kompilatorowi dodatkowe informacje. Pierwsza (ang. header stop) mówi, że właśnie nastąpił koniec listy plików nagłówkowych. Użycie drugiej (ang. argument used) zapobiega wyświetleniu ostrzeżenia, że argument funkcji main() nie został użyty. Mając tak przygotowane środowisko możemy przystąpić to napisania pierwszego programu. Chcemy aby na ekranie monitora był wyświetlony napis „Zaczynamy programowanie w C++ Builder". Aby wyświetlić taki komunikat musimy napisać instrukcję:

```
cout << "Zaczynamy programowanie w C++ Builder";
```

Wyświetlaniem napisów na ekranie zajmuje się klasa iostream. Klasa iostream obsługuje podstawowe operacje wejścia/wyjścia wykorzystując mechanizm strumieni (ang. stream). Do wysyłania danych do standardowego wyjścia (ekran) służy klasa cout, do pobierania danych wejściowych ze standardowego wejścia (klawiatura) służy klasa cin. Do obsługi strumieni potrzebne są dwa operatory: wstawiania « i pobierania ». Należy zaznaczyć, że użycie strumienia cout ma sens tylko w aplikacjach tekstowych. W aplikacjach graficznych do wyprowadzania informacji potrzebne są inne mechanizmy. Gdy odwołujemy się do strumienia cout, należy włączyć plik nagłówkowy <iostream.h>. Wykorzystując edytor tekstowy piszemy tekst naszego programu. Zazwyczaj usuwamy zbędne dyrektywy sugerowane przez RAD oraz zbędne argumenty. Program po poprawkach przedstawiony jest na wydruku 2.1.

Listing 2.1. Pierwszy program testowy

```
1 #include <iostream.h>
  #include <conio.h>
3
  int main()
5 {
    cout << "Zaczynamy programowanie w C++ Builder";
7   getch();
    return 0;
```

```
9 }
```

Po napisaniu tekstu programu możemy go wykonać. W systemach RAD mamy do dyspozycji odpowiednia opcje menu (obsługa myszką) lub skrót klawiszowy). Występująca w programie instrukcja

```
getch();
```

realizuje tzw. „przytrzymanie ekranu". Bez tej instrukcji (albo innej np. PAUSE) w większości RAD nie jesteśmy w stanie zobaczyć wyniku działania programu. Funkcja getch() odczytuje naciśnięcie pojedynczego klawisza. Dopóki nie naciśniemy dowolnego klawisza, program jest wstrzymany, dzięki temu możemy oglądać wynik działania programu na ekranie.

Tak napisany program powinien być zapisany na dysku w wybranym katalogu. Podczas zapisywania projektu RAD tworzy różne pliki. Plik projektu (ang. project file) zawierają informacje potrzebne do generowania kodu wynikowego. Plik źródłowy (ang. source file) o rozszerzeniu zazwyczaj .cpp zawiera tekst źródłowy programu. Widzimy, że tworzenie aplikacji w języku C++ tworzy się w zintegrowanym środowisku programistycznym (IDE) stosunkowo łatwo. Programowanie w takim środowisku ma wiele zalet:
— Środowisko generuje szkielet programu
— Zazwyczaj wbudowany edytor podświetla słowa kluczowe,
— Zazwyczaj kompilacja i uruchomienie programu realizowane są w jednym kroku
— Środowisko ma wbudowany debuger, wykrywanie prostych błędów jest efektywne

W Internecie można znaleźć wiele bezpłatnych środowisk programistycznych. Wybór środowiska jest kwestią gustu. Jeżeli weźmiemy pod uwagę wielkość pamięci potrzebnej do zainstalowania takiego środowiska to rekomendujemy doskonałe środowisko Dev-C++ przeznaczone dla systemu operacyjnego Windows. Dev-C++ jest tzw. oprogramowaniem otwartym (open-source). Jeżeli chcemy pracować z tym środowiskiem, możemy je pobrać nieodpłatnie z witryny Bloodshed Software (`www.bloodshed.net`).

Rozdział 3

Klasy i obiekty

3.1. Wstęp **42**
3.2. Deklaracja i definicja klasy **42**
3.3. Wystąpienie klasy (definiowanie obiektu) **43**
3.4. Dostęp do elementów klasy **43**
3.5. Metody klasy **47**
3.6. Klasa z akcesorami **50**
3.7. Funkcje składowe const **53**

3.1. Wstęp

Zasadniczą cechą języka C++ jest możliwość używania klas (ang. class). Niezbyt precyzyjnie mówiąc, klasa jest bardzo podobna do typu strukturalnego znanego z języka C. Zasadnicza różnica między klasą a strukturą polega na fakcie, że struktura przechowuje jedynie dane, natomiast klasa przechowuje zarówno dane jak i funkcje. Klasa jest typem definiowanym przez użytkownika. Gdy deklarowane są zmienne tego typu, są one obiektami. Mówiąc krótko: w języku C++ klasy są formalnymi typami, obiekty są specyficznymi zmiennymi tego typu. Klasa jest właściwie zbiorem zmiennych, często różnego typu i skojarzonymi z tymi danymi metodami, czyli funkcjami wykorzystywanymi wyłącznie dla ściśle określonych danych. Operowanie obiektami w programie jest istotą programowania obiektowego.

3.2. Deklaracja i definicja klasy

Zmienna obiektowa (obiekt) jest elementem klasy, klasa jest typem definiowanym przez użytkownika. Klasa tworzona jest przy pomocy słowa kluczowego class (podobnie jak struktura przy pomocy słowa kluczowego struct) i może zawierać dane i prototypy funkcji. W skład klasy wchodzą zmienne proste oraz zmienne innych klas. Zmienna wewnątrz klasy jest nazywana zmienną składową lub daną składową. Funkcja w danej klasie zwykle odnosi się do zmiennych składowych. Funkcje klasy nazywa się funkcjami składowymi lub metodami klasy. Klasa może być deklarowana:

- Na zewnątrz wszystkich funkcji programu. W tym przypadku jest ona widoczna przez wszystkie pliki programu.
- Wewnątrz definicji funkcji. Tego typu klasa nazywa się lokalną, ponieważ jest widzialna tylko wewnątrz funkcji.
- Wewnątrz innej klasy. Tego typu klasa jest nazywana klasą zagnieżdżoną

Deklaracja klasy o nazwie Kot może mieć następującą postać:

```
class Kot
{
     int wiek;
     int waga;
     void Glos();
  };
```

Została utworzona klasa o nazwie Kot, zawiera ona dane: wiek i waga a ponadto jedną funkcje Glos(). Taka deklaracja nie alokuje pamięci, informuje jedynie kompilator, czym jest typ Kot, jakie zawiera dane i funkcje.

Na podstawie tych informacji kompilator wie jak dużą potrzeba przygotować pamięć w przypadku tworzenia zmiennej typu Kot. W tym konkretnym przykładzie zmienna typu Kot zajmuje 8 bajtów (przy założeniu, że typ int potrzebuje 4 bajty). Dla funkcji składowych (w naszym przykładzie funkcja Glos()) miejsce w pamięci nie jest rezerwowane.

3.3. Wystąpienie klasy (definiowanie obiektu)

Wystąpienie klasy deklaruje się tak samo jak zmienne innych typów, np.

```
Kot Filemon;
```

W ten sposób definiujemy zmienną o nazwie Filemon, która jest obiektem klasy Kot. Mówimy, że ten obiekt jest indywidualnym egzemplarzem klasy Kot. Deklaracja obiektów możliwa jest także w trakcie deklaracji klasy, np.

```
class Kot
{
    int wiek;
    int waga;
    void Glos();
} Filemon;
```

Należy pamiętać o następujących ograniczeniach przy deklarowaniu składowych klasy:
— Deklarowana składowa nie może być inicjalizowana w deklaracji klasy
— Nazwy składowych nie mogą się powtarzać
— Deklaracje składowych nie mogą zawierać słów kluczowych auto, extern i register

Zamknięcie związanych ze sobą elementów klasy (danych i metod) w jedną całość nazywane jest hermetyzacją albo enkapsulacją (ang. encapsulation). Zdefiniowaną klasę traktujemy jako typ, który może być wykorzystany na różne sposoby. Poniższe deklaracje ilustrują to zagadnienie:

```
Kot Filemon;          //deklaracja obiekty typu  Kot
Kot Grupa_koty[10];   //deklaracja tablicy obiektow Kot
Kot *wskKot;          //wskaznik do obiektu Kot
```

3.4. Dostęp do elementów klasy

Dostęp do składowej obiektu klasy uzyskuje się przy pomocy operatora dostępu, zwanego też operatorem kropki (.). Gdy chcemy przypisać zmiennej składowej o nazwie waga wartość 10, należy napisać następującą instrukcję:

```
Filemon.waga = 10;
```

Podobnie w celu wywołania funkcji Glos(), należy napisać następującą instrukcję:

```
Filemon.Glos();
```

Dla zmiennych wskaźnikowych dostęp do elementów klasy realizuje się przy pomocy operatora dostępu (->).

W języku C++ są trzy kategorie dostępu do elementów klasy:
— Prywatny – etykieta private
— Publiczny – etykieta public
— Chroniony – etykieta protected

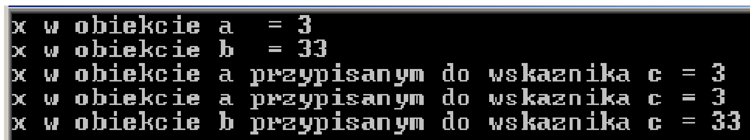
Poziomy dostępu określają sposób wykorzystania elementów klasy przez jej użytkowników. Specyfikatory dostępu (słowa kluczowe private, public i protected) odgrywają dużą rolę w programowaniu obiektowym. Ponieważ każda klasa powinna komunikować się z otoczeniem, powinna posiadać część publiczną (inaczej interfejs), czyli elementy, do których można odwoływać się z zewnątrz. Bardzo często chcemy, aby funkcje składowe klasy nie były dostępne z zewnątrz; funkcje te tworzą część prywatną, czyli implementację klasy. Ukrywanie wewnętrznych szczegółów implementacji przed dostępem z zewnątrz jest jednym z elementów dobrego projektowania klas i nosi nazwę abstrahowania danych. Elementy klasy objęte dostępem chronionym (słowo kluczowe protected) nie są dostępne z zewnątrz generalnie, jednak są dostępne dla klas od nich pochodnych. Domyślną kategorią dostępu dla wszystkich elementów klasy jest kategoria private. Oznacza to, gdy nie podano specyfikatora dostępu, kompilator przyjmie domyślnie etykietę private. W tym przypadku dane składowe są dostępne jedynie dla funkcji składowych i tzw. funkcji zaprzyjaźnionych. Zastosowanie kategorii public powoduje, że występujące po tej etykiecie nazwy deklarowanych składowych mogą być używane przez dowolne funkcje. Rozważmy następujący przykład.

Listing 3.1. Dostęp do składowych publicznych

```
1 #include <iostream.h>
  #include <conio.h>
3 class Liczba
  {
5   public:
          int x;
7 };

9 int main()
  {
11  Liczba a, b, *c;
```

```
      a.x = 3;
13    b.x = 33;
      c = &a;
15    cout << "x w obiekcie a  = " << a.x << endl;
      cout << "x w obiekcie b  = " << b.x << endl;
17    cout << "x w obiekcie a przypisanym do wskaznika c = "
         << c -> x << endl;
19    cout << "x w obiekcie a przypisanym do wskaznika c = "
         << (*c).x << endl;
21    c = &b;
      cout << "x w obiekcie b przypisanym do wskaznika c = "
23       <<(*c).x << endl;
        getch();
25      return 0;
    }
```

Wynik wykonania programu:



W programie występuje klasa Liczba, która ma zmienną składową x. Zmienna składowa jest składową publiczną i dlatego można w obiektach a i b bezpośrednio przypisywać jej wartość 3 i 33. W instrukcji:

```
Liczba a, b, *c;
```

zadeklarowano wskaźnik c do klasy Liczba. W instrukcji:

```
c = &a;      oraz      c = &b;
```

temu samemu wskaźnikowi c przypisano kolejno obiekty a i b klasy Liczba. W instrukcjach:

```
cout << "x w obiekcie a przypisanym do wskaznika c = "
    << c -> x << endl;
cout << "x w obiekcie a przypisanym do wskaznika c = "
    << (*c).x << endl;
```

zademonstrowano operator dostępu (operator kropki i operator strzałki) do składowej x w równoważnych postaciach. W kolejnym przykładzie wszystkie składowe klasy Kot są publiczne, typy zmiennych są różne, w naszym przypadku int, float i char.

Listing 3.2. Dostęp do publicznych składowych klasy

```
 1 #include <iostream.h>
 2 #include <conio.h>
 3 class Kot
 4  {
 5     public:
 6       int wiek;
 7       float waga;
 8       char *kolor;
 9  };
10 int main()
11 {
12     Kot Filemon;
13     Filemon.wiek = 3;
14     Filemon.waga = 3.5;
15     Filemon.kolor = "rudy";
16     cout << "Filemon to " << Filemon.kolor << " kot " << endl;
17     cout << "Wazy " << Filemon.waga << " kilogramow" ;
18     cout << " i ma " << Filemon.wiek << " lata" << endl;
19       getch();
20       return 0;
21 }
```

Wynik działania programu:

```
Filemon to rudy kot
Wazy 3.5 kilogramow i ma 3 lata
```

Deklaracja klasy Kot ma postać:

```
class Kot
 {
   public:
     int wiek;
     float waga;
     char *kolor;
 };
```

Ponieważ składowe klasy są publiczne to możemy utworzyć obiekt Filemon i przypisać im odpowiednie wartości:

```
Kot Filemon;
Filemon.wiek = 3;
Filemon.waga = 3.5;
Filemon.kolor = "rudy";
```

Jeżeli w deklaracji klasy Kot nie będzie kwantyfikatora public, to kompi-

lator uzna, że składowe klasy są prywatne i próba przypisania im wartości zakończy się niepowodzeniem.

3.5. Metody klasy

Klasę tworzą dane i funkcje składowe. Funkcje składowe (są to tzw. metody) mogą być deklarowane jako inline, static i virtual, nie mogą być deklarowane jako extern.

Listing 3.3. Dostęp do publicznych składowych klasy

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

class Kot
 {
   public:
     int wiek;
     int waga;
     char *kolor;
     void Glos();
 };

 void Kot :: Glos()
   { cout << " miauczy " ; }



int main()
{
   Kot Filemon;
   Filemon.wiek = 3;
   Filemon.waga = 5;
   Filemon.kolor = "rudy";
   cout << "Filemon to " << Filemon.kolor << " kot " << endl;
   cout << "Wazy " << Filemon.waga << " kilogramow" ;
   cout << " i ma " << Filemon.wiek << " lata" << endl;
   cout << "gdy jest glodny to " ;
   Filemon.Glos();
     getch();
     return 0;
}
```

Wynik wykonania programu:

```
Filemon to rudy kot
Wazy 5 kilogramow i ma 3 lata
gdy jest glodny to  miauczy
```

W definicji klasy Kot pojawiła się funkcja składowa (metoda) o nazwie Glos():

```
class Kot
 {
  public:
     int wiek;
     int waga;
     char *kolor;
     void Glos();
 };
```

Funkcja składowa Glos() jest zdefiniowana w następujący sposób:

```
void Kot :: Glos()
   { cout << " miauczy " ; }
```

Jak widzimy, ta metoda nie ma parametrów i nie zwraca żadnej wartości. Metoda zawiera nazwę klasy (tutaj Kot), dwa dwukropki (operator zasięgu) oraz nazwę funkcji (tutaj Glos). Ta składnia informuje kompilator, że ma do czynienia z funkcją składową zadeklarowaną w klasie Kot. Dwuargumentowy operator zasięgu '::' znajdujący się pomiędzy nazwą klasy i nazwą funkcji składowej ustala zasięg funkcji Glos(). Napis Kot:: informuje kompilator, że występująca po nim funkcja jest funkcją składową klasy Kot. Aby wywołać funkcję Glos() należy umieścić instrukcję:

```
Filemon.Glos();
```

Aby użyć metodę klasy należy ją wywołać. W naszym przykładzie została wywołana metoda Glos() obiektu Filemon. Można definiować funkcje wewnątrz klasy:

```
class Kot
 {
   public:
     int wiek;
     int waga;
     char *kolor;
     void Glos() { cout << " miauczy " ; };
 };
```

Jeżeli funkcja składowa (metoda) jest zdefiniowana wewnątrz klasy to jest traktowana jako funkcja o atrybucie inline. Metodę można także zadeklarować jako inline poza klasą:

```
class Kot
 {
```

```
    public:
        int wiek;
        int waga;
        char *kolor;
        void Glos();
};

    inline void Kot :: Glos()
    { cout << " miauczy " ; }
```

Elementami klasy są dane i metody, należą one do zasięgu klasy. Zasięg klasy oznacza, że wszystkie składowe klasy są dostępne dla funkcji składowych poprzez swoją nazwę. Poza zasięgiem klasy, do składowych odnosimy się najczęściej za pośrednictwem tzw. uchwytów – referencji i wskaźników. Jak już mówiliśmy, w języku C++ mamy dwa operatory dostępu do składowych klasy:

Operator kropki (.) stosowany z nazwą obiektu lub referencją do niego
Operator strzałki (->) stosowany w połączeniu ze wskaźnikiem do obiektu

Operator strzałki (ang. arrow member selection operator) jest nazywany w polskiej literaturze różnie, najczęściej jako skrótowy operator zasięgu lub krótko operator wskazywania. Aby przykładowo wydrukować na ekranie składową sekunda klasy obCzas, z wykorzystaniem wskaźnika czasWs, stosujemy instrukcję:

```
czasWs = &obCzas;
cout << czasWs -> sekunda;
```

Wyrażenie:

```
czasWs -> sekunda
```

jest odpowiednikiem

```
(*czasWs). sekunda
```

To wyrażenie dereferuje wskaźnik, a następnie udostępnia składową sekunda za pomocą operatora kropki. Konieczne jest użycie nawiasów okrągłych, ponieważ operator kropki ma wyższy priorytet niż operator dereferencji. Ponieważ taki zapis jest dość skomplikowany, w języku C++ wprowadzono skrótowy operator dostępu pośredniego – operator strzałki. Używanie operatora strzałki jest preferowane przez większość programistów. W prostym programie zademonstrujemy korzystanie z klasy Liczba dla zilustrowania sposobów dostępu do składowych klasy za pomocą omówionych operatorów wybierania składowych.

Listing 3.4. Dostęp do składowych: operator kropki i strzałki

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

class Liczba
{
  public:
    float x;
    void pokaz() { cout << x << endl; }
};

int main()
{
   Liczba nr;                      //obiekt nr typu Liczba
   Liczba *nrWsk;                  //wskaznik
   Liczba &nrRef = nr;             //referencja

nr.x = 1.01;
cout << " nazwa obiektu,      skladowa x ma wartosc : ";
nr.pokaz();

nrRef.x = 2.02;
cout << " referencja obiektu, skladowa x ma wartosc : ";
nrRef.pokaz();

nrWsk = &nr;
nrWsk -> x = 3.03;
cout << " wskaznik obiektu,   skladowa x ma wartosc : ";
nrWsk -> pokaz();
        getch();
        return 0;
}
```

3.6. Klasa z akcesorami

Dane składowe klasy powinny być kwalifikowane jako prywatne. Wobec tego klasa powinna posiadać publiczne funkcje składowe, dzięki którym możliwe będzie manipulowanie danymi składowymi. Publiczne funkcje składowe zwane są funkcjami dostępowymi lub akcesorami, ponieważ umożliwiają odczyt zmiennych składowych i przypisywanie im wartości. W kolejnym przykładzie pokażemy zastosowanie publicznych akcesorów.

Akcesory są funkcjami składowymi, które użyjemy do ustawiania wartości prywatnych zmiennych składowych klasy i do odczytu ich wartości.

Listing 3.5. Dostęp do prywatnych składowych klasy - akcesory

```
1 #include <iostream.h>
  #include <conio.h>
3 class Prost
   { private:
5          int a;
           int b;
7     public:
           int get_a();
9          int get_b();
           void set_a(int aw);
11         void set_b(int bw);
   };
13
   int Prost :: get_a() {return a ;}
15 int Prost :: get_b() {return b ;}
   void Prost :: set_a (int aw) {a = aw;}
17 void Prost :: set_b (int bw) {b = bw;}

19 int main()
  {  Prost p1;
21    int pole;
      p1.set_a(5);
23    p1.set_b(10);
      pole = p1.get_a() * p1.get_b();
25 //    pole = p1.a * p1.b;   //niepoprawna instrukcja
      cout << "Powierzchnia = " << pole << endl;
27         getch();
           return 0;
29 }
```

Wynik działania programu:

```
Powierzchnia = 50
```

W tym programie po wprowadzeniu długość i szerokość prostokąta obliczamy jego pole powierzchni. Klasa o nazwie Prost ma następującą deklarację:

```
class Prost
 {  private:
         int a;
         int b;
    public:
         int get_a();
         int get_b();
         void set_a(int aw);
         void set_b(int bw);
 };
```

Dane składowe a i b są danymi prywatnymi. Wszystkie funkcje składowe są akcesorami publicznymi. Dwa akcesory ustawiają zmienne składowe:

```
void set_a(int aw);
void set_b(int bw);
```

a dwa inne:

```
int get_a();
int get_b();
```

zwracają ich wartości. Akcesory są publicznym interfejsem do prywatnych danych klasy. Każdy akcesor (tak jak każda funkcja) musi mieć definicję. Definicja akcesora jest nazywana implementacją akcesora. W naszym przykładzie definicje akcesorów publicznych są następujące:

```
int Prost :: get_a() {return a ;}
int Prost :: get_b() {return b ;}
void Prost :: set_a (int aw) {a = aw;}
void Prost :: set_b (int bw) {b = bw;}
```

Postać definicji funkcji składowej jest dość prosta. Np. zapis:

```
int Prost :: get_a() {return a ;}
```

zawiera definicje funkcji get_a(). Ta metoda nie ma parametrów, zwraca wartość całkowitą. Metoda klasy zawiera nazwę klasy (w naszym przypadku jest to Prost), dwa dwukropki i nazwę funkcji. Tak składnia informuje kompilator, że definiowana funkcja get_a() jest funkcją składową klasy Prost. Bardzo prosta funkcja get_a() zwraca wartość zmiennej składowej a. Składowa a jest składową prywatną klasy Prost i funkcja main() nie ma do niej dostępu bezpośredniego. Z drugiej strony get_a() jest metodą publiczna, co oznacz, że funkcja main() ma do niej pełny dostęp. Widzimy, że funkcja main() za pośrednictwem metody get_a() ma dostęp do danej prywatnej a. Podobnie ma się sprawa z metodą set_a(). Kod wykonywalny zawarty jest w ciele funkcji main(). W instrukcji:

```
Prost p1;
```

zadeklarowany jest obiekt Prost o nazwie p1. W kolejnych instrukcjach:

```
p1.set_a(5);
p1.set_b(10);
```

zmiennym a i b przypisywane są wartości przy pomocy akcesorów set_a() i set_b(). Wywołanie tej metody polega na napisaniu nazwy obiektu (tu-

taj p1) a następnie użyciu operatora kropki (.) i nazwy metody (w tym przypadku set_a() i set_b()). W instrukcji:

```
pole = p1.get_a() * p1.get_b();
```

obliczane jest pole powierzchni prostokąta.

3.7. Funkcje składowe const

Dobrze napisane programy są zabezpieczone przed przypadkową modyfikacją obiektów. Dobrym sposobem określania czy obiekt może być zmieniony czy nie jest deklarowanie obiektów przy pomocy słowa kluczowego const. W wyrażeniu:

```
const Liczba nr ( 10.05 ) ;
```

zadeklarowano stały obiekt nr klasy Liczba z jednoczesną inicjacją. Podobnie deklarowane mogą być funkcje składowe. W prototypie i w definicji funkcji składowej należy użyć słowa kluczowego const. Na przykład następująca definicja funkcji składowej klasy Liczba

```
int Liczba :: pokazWartosc ( ) const { return x ; };
```

zwraca wartość jednej z prywatnych danych składowych funkcji. Prototyp tej funkcji może mieć postać:

```
int pokazWartosc ( ) const ;
```

Wraz z modyfikatorem const często deklarowane są akcesory. Deklarowanie funkcji jako const jest bardzo dobrym zwyczajem programistycznym, efektywnie pomaga wykrywać błędy przy przypadkowej próbie modyfikowania stałych obiektów. Funkcje składowe zadeklarowane jako const nie mogą modyfikować danych obiektu, ponieważ nie dopuści do tego kompilator. Obiekt stały nie może być modyfikowany za pomocą przypisań, ale może być zainicjalizowany. W takim przypadku można wykorzystać odpowiednio zbudowany konstruktor. Do konstruktora musi być dostarczony odpowiedni inicjator, który zostanie wykorzystany jako wartość początkowa stałej danej klasy. W kolejnym przykładzie klasa Punkt posiada trzy dane prywatne, w tym jedną typu const. Do celów dydaktycznych w programie umieszczono dwa konstruktory (w danym momencie może być czynny tylko jeden z nich). Jeżeli program uruchomimy z konstruktorem w postaci:

```
Punkt :: Punkt ( int xx, int yy, int k )
        { x = 1; y = 1; skok = k ; }          //ERROR !!!
```

zostanie wygenerowany następujący komunikat:

```
[C++ Warning] ftest1.cpp[23]: W8038 Constant member
    'Punkt::skok' is not initialized
[C++ Error] ftest1.cpp[23]: E2024 Cannot modify
    a const object
```

W definicji konstruktora jest próba zainicjowania danej składowej skok za pomocą przypisania a nie przy użyciu inicjatora składowej i to wywołuje błędy kompilacji.

Listing 3.6. Obiekty typu const

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

class Punkt                        // deklaracja klasy punkt
{
  public:                          //skladowe publiczne
     Punkt (int xx = 0, int yy = 0, int k = 1 );
     void przesunY() {y += skok;}
     void pokaz() const;
  private:                         //skladowe prywatne
     int x, y;
     const int skok;
};

Punkt :: Punkt(int xx, int yy, int k) : skok(k)
          { x = 1; y = 1;}     //konstruktor
/*----------------
Punkt :: Punkt (int xx, int yy, int k)
          { x = 1; y = 1; skok = k; }    //ERROR !!!
*/----------------

void Punkt :: pokaz () const     //definicja metody
 { cout << "skok = " << skok << "  x = "
       << x << " y = " << y << endl;
 }

int main()
{
  Punkt p1( 1, 1, 2);          //deklaracja obiektu
                      //p1 typu Punkt
  cout << "---- dane poczatkowe ---- " << endl;
  p1.pokaz();
  cout << "- zmiana wspolrzednej y - " << endl;
  for ( int j = 0; j <5; j++)
     {
      p1.przesunY();
      p1.pokaz();
     }
```

```
            getch();
40          return 0;
    }
```

Program zadziała poprawnie, gdy wykorzystane zostaną inicjatory do zainicjowania stałej danej składowej skok klasy Punkt. Poprawny konstruktor ma postać:

```
Punkt :: Punkt(int xx, int yy, int k) : skok(k)   //konstruktor
        { x = 1; y = 1;}
```

Wynik działania programu:

```
---- dane poczatkowe ----
skok = 2   x = 1 y = 1
- zmiana wspolrzednej y -
skok = 2   x = 1 y = 3
skok = 2   x = 1 y = 5
skok = 2   x = 1 y = 7
skok = 2   x = 1 y = 9
skok = 2   x = 1 y = 11
```

W definicji konstruktora widać notację zapożyczoną przez C++ z języka Simula. Inicjowanie jakiejś zmiennej x wartością np. 44 najczęściej ma postać:

```
int x = 44;
```

Dopuszczalne jest także równoważna postać:

```
int x(44);
```

Wykorzystując drugą notację otrzymujemy definicje konstruktora klasy Punkt. Wyrażenie

```
: skok(k)
```

powoduje zainicjowanie składowej skok wartością k. Możemy mieć listę zmiennych składowych z wartościami inicjalnymi :

```
Punkt :: Punkt (int a, int b) : xx(a), yy(b)  { }
```

Po dwukropku należy kolejno umieszczać inicjatory rozdzielając je przecinkami.

Rozdział 4

Konstruktory i destruktory

4.1. Wstęp **58**
4.2. Inicjalizacja obiektu klasy **58**
4.3. Konstruktory i destruktory domyślne **59**
4.4. Konstruktor jawny **61**
4.5. Wywoływanie konstruktorów i destruktorów **63**
4.6. Rozdzielenie interfejsu od implementacji **65**
4.7. Wskaźnik do obiektu this **67**
4.8. Wskaźnik this – kaskadowe wywołania funkcji **68**
4.9. Tablice obiektów **71**
4.10. Inicjalizacja tablic obiektów nie będących agregatami . **73**
4.11. Tablice obiektów tworzone dynamicznie **75**
4.12. Kopiowanie obiektów **77**
4.13. Klasa z obiektami innych klas **78**

4.1. Wstęp

Klasa jest typem definiowanym przez użytkownika. Gdy deklarowane są zmienne tego typu, są one obiektami. Mamy następujące definicje.
- Klasa jest zdefiniowanym przez użytkownika typem danych, zawiera pola danych oraz funkcje.
- Obiekt jest egzemplarzem utworzonego (na podstawie klasy) typu.
- Klasa ma charakter unikatowy i istnieje tylko jedna postać klasy
- Może istnieć wiele obiektów określonego typu (utworzonego na podstawie konkretnej klasy)

Sama definicja klasy nie definiuje żadnych obiektów. W definicji klasy, dane nie mogą być inicjalizowane (nie można przypisać im wartości). Dane można inicjalizować dla konkretnego obiektu (egzemplarza klasy). Po utworzeniu klasy można powołać do życia jej egzemplarz, czyli obiekt. Dobrą praktyką programistyczną jest tworzenie obiektów z zainicjalizowanymi danymi. Inicjalizacja obiektów nie jest zadaniem trywialnym.

4.2. Inicjalizacja obiektu klasy

Bardzo często chcemy, aby w momencie tworzenia obiektu, jego składowe były zainicjalizowane. Dla prostych typów definicja zmiennej i jednoczesna inicjalizacja ma postać:

```
int x1 = 133;
```

Inicjalizacja łączy w sobie definiowanie zmiennej i początkowe przypisanie wartości. Oczywiści później możemy zmienić wartości zmiennej. Inicjalizacja zapewnia, że zmienna będzie miała sensowną wartość początkową. Składowe klasy także możemy inicjalizować. Dane składowe klasy inicjalizowane są za pomocą funkcji składowej o nazwie konstruktor (ang. costructor). Konstruktor jest funkcją o nazwie identycznej z nazwą klasy. Konstruktor jest wywoływany za każdym razem, gdy tworzony jest obiekt danej klasy. Konstruktor może posiadać argumenty, ale nie może zwracać żadnej wartości.

Destruktor (ang. destructor) klasy jest specjalną funkcją składową klasy. Destruktor jest wywoływany przy usuwaniu obiektu. Destruktor nie otrzymuje żadnych parametrów i nie zwraca żadnej wartości. Klasa może posiadać tylko jeden destruktor. Nazwa destruktora jest taka sama jak klasy, poprzedzona znakiem tyldy (~). Gdy jest zadeklarowany jeden konstruktor, powinien być także zadeklarowany destruktor. Wyróżniamy konstruktory i dekonstruktory inicjujące, konstruktory i dekonstruktory domyślne i konstruktor kopiujący.

Zadaniem konstruktora jest konstruowanie obiektów. Po wywołaniu konstruktora następuje:
— Przydzielenie pamięci dla obiektu
— Przypisanie wartości do zmiennych składowych
— Wykonanie innych operacji (np. konwersja typów)

Podczas deklaracji obiektu, mogą być podane inicjatory (ang. initializers). Są one argumentami przekazywanymi do konstruktora. Programista nigdy jawnie nie wywołuje konstruktora, może za to wysłać do niego argumenty.

4.3. Konstruktory i destruktory domyślne

Każda klasa zawiera konstruktor i destruktor. Jeżeli nie zostały zadeklarowane jawnie, uczyni to w tle kompilator. Zagadnienie to ilustrujemy popularnym przykładem – realizacji klasy Punkt do obsługi punktów na płaszczyźnie.

Listing 4.1. Klasa Punkt (punkt na płaszczyźnie)

```
1  #include <iostream.h>
   #include <conio.h>
3
   class Punkt                           // deklaracja klasy punkt
5  { public:                             //skladowe publiczne
        void ustaw(int, int);
7       void przesun(int, int);
        void pokaz();
9    private:                            //skladowe prywatne
        int x, y;
11 };

13 void Punkt :: ustaw(int a, int b) //definicja metody
      { x = a;
15      y = b;
      }
17 void Punkt::przesun(int da, int db) //definicja metody
      { x = x + da;
19      y = y + db;
       }
21 void Punkt :: pokaz ()              //definicja metody
    { cout << "wspolrzedne: " << x << "  " << y << endl;
23  }

25 int main()              //program testujacy uzycie klasy Punkt
   { Punkt p1;             //deklaracja obiektu p1 typu Punkt
27    p1.ustaw (5, 10); //inicjalizacja
      p1.pokaz();
29    p1.przesun(10,10);
```

```
   p1.pokaz();
31   getch();
   return 0;
33 }
```

Wynik działania programu:

```
Wspolrzedne : 5   10
Wspolrzedne : 15   20
```

Klasa Punkt ma następującą postać:

```
class Punkt                    // deklaracja klasy punkt
{
  public:                      //skladowe publiczne
     void ustaw(int, int);
     void przesun(int, int);
     void pokaz();
  private:                     //skladowe prywatne
     int x, y;
};
```

Deklaracja klasy składa się z prywatnych danych (współrzędne kartezjańskie punktu x i y) oraz z publicznych funkcje składowych: ustaw(), przesun() i pokaz(). Definicja klasy składa się z definicji wszystkich funkcji składowych (metod) i ma postać:

```
void Punkt :: ustaw(int a, int b)   //definicja metody
{ x = a;
  y = b;
}

void Punkt::przesun(int da, int db) //definicja metody
 { x = x + da;
   y = y + db;
 }

void Punkt :: pokaz ()              //definicja metody
 { cout << "wspolrzedne: " << x << "  " << y << endl;
 }
```

Wewnątrz definicji wszystkie dane i funkcje składowe są bezpośrednio dostępne (bez względu na to czy są publiczne czy prywatne). Dane składowe należące do klasy pamięci typu static są dostępne dla wszystkich obiektów danej klasy. Przez domniemanie , dane statyczne są inicjalizowane wartością zerową. Jak już mówiliśmy, jeżeli nie stworzymy konstruktora, kompilator stworzy konstruktor domyślny – bezparametrowy. Potrzeba tworzenia kon-

struktora jest natury technicznej – wszystkie obiekty muszą być konstruowane i niszczone, dlatego często tworzone są nic nie robiące funkcje. Przypuśćmy, że chcemy zadeklarować obiekt bez przekazywania parametrów:

```
Punkt p1;
```

to wtedy musimy posiadać konstruktor w postaci:

```
Punkt();
```

Gdy definiowany jest obiekt klasy, wywoływany jest konstruktor. Załóżmy, że konstruktor klasy Punkt ma dwa parametry, można zdefiniować obiekt Punkt pisząc:

```
Punkt p1( 5, 10);
```

Dla konstruktora z jednym parametrem mamy instrukcję:

```
Punkt p1(5);
```

Jeżeli konstruktor jest domyślny (nie ma żadnych parametrów) można opuścić nawiasy i napisać po prostu:

```
Punkt p1;
```

Jest to wyjątek od reguły, która mówi, że funkcje muszą mieć nawiasy nawet wtedy, gdy nie mają parametrów. Dlatego zapis:

```
Punkt p1;
```

jest interpretowany przez kompilator jako wywołanie konstruktora domyślnego – w tym przypadku nie wysyłamy parametrów i nie piszemy nawiasów. Gdy deklarowany jest konstruktor, powinien być deklarowany destruktor, nawet gdy nic nie robi.

4.4. Konstruktor jawny

Zmienimy teraz klasę Punkt w ten sposób, że do inicjalizacji obiektu użyjemy konstruktora jawnego, pokażemy także jawną postać destruktora.

Listing 4.2. Klasa Punkt z konstruktorem

```
1 #include <iostream.h>
  #include <conio.h>
3 class Punkt
  { public:
```

```
5       Punkt(int, int);                //konstruktor
        ~Punkt();                       //destruktor
7       void przesun(int, int);
        void pokaz();
9     private:
        int x, y;
11  };
    Punkt :: Punkt(int a, int b)     //implementacja konstruktora
13                    //klasy Punkt
       { x = a;        y = b;      }
15  Punkt :: ~Punkt()     {   }        // implementacja destruktora
    void Punkt::przesun(int da, int db)
17     { x = x + da;          y = y + db;        }
    void Punkt :: pokaz ()
19   { cout << "wspolrzedne: " << x << "  " << y << endl;    }

21  int main()
    {     Punkt p1(5, 10);            //ustawia punkt
23          p1.pokaz();                //pokazuje wspolrzedne
            p1.przesun(15,15);         //przesuwa punkt
25          p1.pokaz();                //pokazuje nowe wspolrzedne
            getch();
27          return 0;
    }
```

Wynik wykonania programu ma postać:

```
wspolrzedne: 5  10
wspolrzedne: 20  25
```

Klasa Punkt z konstruktorem jawnym ma postać:

```
class Punkt
{
  public:
     Punkt(int, int);               // konstruktor
     ~Punkt();                      // destruktor
     void przesun(int, int);
     void pokaz();
  private:
     int x, y;
};
```

Konstruktor ma dwa parametry, dzięki którym ustawiamy wartości współrzędnych punktu:

```
Punkt :: Punkt(int a, int b)      // konstruktor klasy Punkt
 { x = a;        y = b;
 }
```

Destruktor w tym programie nic nie robi. Ponieważ deklaracja klasy zawiera deklarację destruktora musimy zamieścić implementację destruktora:

```
Punkt :: ~Punkt()          // implementacja destruktora
 {
 }
```

W funkcji main() należy zwrócić uwagę na instrukcję:

```
Punkt p1(5, 10);           //ustawia punkt
```

Mamy tutaj definicję obiektu p1, stanowiącego egzemplarz klasy Punkt. Do konstruktora obiektu p1 przekazywane są wartości 5 i 10.

4.5. Wywoływanie konstruktorów i destruktorów

Konstruktory i destruktory wywoływane są automatycznie. Kolejność ich wywoływania jest dość skomplikowanym zagadnieniem – wszystko zależy od kolejności w jakiej wykonywany program przetwarza obiekty globalne i lokalne. Gdy obiekt jest zdefiniowany w zasięgu globalnym, jego konstruktor jest wywoływany jako pierwszy a destruktor, gdy funkcja main() kończy pracę. Dla automatycznych obiektów lokalnych konstruktor jest wywoływany w miejscu, w którym zostały one zdefiniowane, a destruktor jest wywołany wtedy, gdy program opuszcza blok, w którym obiekt ten był zdefiniowany. Dla obiektów typu static konstruktor jest wywoływany w miejscu, w którym obiekt został zdefiniowany a destruktor jest wywołany wtedy, gdy funkcja main() kończy pracę.

Do demonstracji kolejności wywoływania konstruktorów i destruktorów adaptowaliśmy program opisany w monografii H.Deitela i P.Deitela „Arkana C++”.

Listing 4.3. Kolejność wywoływania konstruktorów i destruktorów

```
 1 #include <iostream.h>
   #include <conio.h>
 3
   class Kon_Des
 5 { public:
        Kon_Des(int);          //konstruktor
 7      ~Kon_Des();            // destruktor
     private:
 9      int nr;
   };
11
   Kon_Des::Kon_Des(int numer)
13  { nr = numer;
```

```
      cout << "konstruktor obiektu " << nr;
   }

Kon_Des :: ~Kon_Des()
  { cout << "destruktor obiektu  " << nr << endl;
  }

void fun_ob(void);        //prototyp funkcji tworzacej
                    //nowe obiekty
Kon_Des ob1(1);           //obiekt globalny

void fun_ob(void)
  {Kon_Des ob5(5);                  // obiekt lokalny
   cout << "   (automatyczny obiekt lokalny w fun_ob() ) "
     << endl;
   static Kon_Des ob6(6);           //obiekt lokalny
   cout << "   (statyczny obiekt lokalny w fun_ob() ) "
     << endl;
   Kon_Des ob7(7);                  // obiekt lokalny
   cout << "   (automatyczny obiekt lokalny w fun_ob() ) "
     << endl;
  }

int main()
{   cout << "   (globany utworzony przed funkcja main() ) "
     << endl;
    Kon_Des ob2(2);                  // obiekt lokalny
    cout << "   (automatyczny obiekt lokalny w main() ) "
     << endl;
    static Kon_Des ob3(3);          // obiekt lokalny
    cout << "   (statyczny obiekt lokalny w main() ) "
     << endl;
    fun_ob();       //wywolanie funkcji tworzacej obiekty
    Kon_Des ob4(4);                 //obiekt lokalny
    cout << "   (automatyczny obiekt lokalny w main() "
     << endl;
         getch();
         return 0;
}
```

Wynik wykonania programu:

```
konstruktor obiektu 1   (globany utworzony przed funkcja main() )
konstruktor obiektu 2   (automatyczny obiekt lokalny w main() )
konstruktor obiektu 3   (statyczny obiekt lokalny w main() )
konstruktor obiektu 5   (automatyczny obiekt lokalny w fun_ob() )
konstruktor obiektu 6   (statyczny obiekt lokalny w fun_ob() )
konstruktor obiektu 7   (automatyczny obiekt lokalny w fun_ob() )
destruktor obiektu  7
destruktor obiektu  5
konstruktor obiektu 4   (automatyczny obiekt lokalny w main()
```

Dalsze wywołania powinny mieć postać:

```
Destruktor obiektu 4
Destruktor obiektu 2
Destruktor obiektu 6
Destruktor obiektu 3
Destruktor obiektu 1
```

4.6. Rozdzielenie interfejsu od implementacji

Zaleca się, aby rozdzielać interfejs od implementacji. W tej metodzie kod programu rozdzielamy na wiele plików. Deklarację klasy zaleca się umieszczać w pliku nagłówkowym. Zwyczajowo plik taki ma rozszerzenie .h . Definicje funkcji składowych klasy powinny być umieszczone w odrębnym pliku źródłowym, zwyczajowo plik taki ma rozszerzenie .cpp, a nazwę taką samą jak plik z deklaracją klasy. Do programu głównego włączamy wszystkie potrzebne, utworzone przez nas pliki nagłówkowe. Zintegrowane środowiska programistyczne mają osobne narzędzia do obsługi programów wieloplikowych.

Zagadnienie programów wieloplikowych ilustrujemy programem wykorzystującym klasę punkt obsługującą punkty na płaszczyźnie. Mamy trzy pliki:
1. punkt1.h – deklaracja klasy punkt
2. punkt1.cpp – definicje funkcji składowych
3. ftest1.cpp – definicja funkcji main()

Listing 4.4. Program wieloplikowy - ftest1.cpp

```
1  //plik ftest1.cpp, program wykorzystuje klase punkt

3  #include <iostream.h>
   #include <conio.h>
5  #include "punkt1.h"
   int main()
7  {
       punkt p1 (0.5, 1.5);
9      p1.wyswietl();
       p1.przesun(0.5,-0.5);
11     p1.wyswietl();
             getch();
13           return 0;
   }
```

Listing 4.5. Program wieloplikowy - punkt1.h

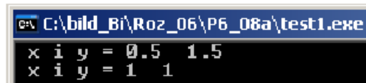
```
   // klasa punkt
```

```
2 #ifndef _PUNKT1_H
  #define _PUNKT1_H
4 class punkt
  {  private:
6     float x, y;
    public:
8      punkt (float, float);
       void przesun(float, float);
10     void wyswietl();
  };
12 #endif
```

Listing 4.6. Program wieloplikowy - punkt1.cpp

```
  //definicje klasy punkt
2 #include "punkt1.h"
  #include <iostream.h>
4
  punkt :: punkt (float xx, float yy)
6 { x = xx;  y = yy;
  }
8
  void punkt :: przesun (float dx, float dy)
10 { x = x + dx;   y = y + dy;
  }
12
  void punkt :: wyswietl ()
14 { cout << " x i y = " << x << "  " << y << endl;
  }
```

Wynik wykonania programu pokazanego na wydruku 4.4a ma postać:



Umieszczone w pliku pokazanym na wydruku 4.4b dyrektywy preprocesora :

```
#ifndef _PUNKT1_H
#define _PUNKT1_H
.......................
.......................
.......................
#endif
```

zapobiegają wielokrotnemu dołączaniu tego pliku do innego.

4.7. Wskaźnik do obiektu this

W języku C++ ze względu na oszczędność pamięci istnieje tylko jedna kopia funkcji danej klasy. Obiekty danej klasy korzystają wspólnie z tej funkcji, natomiast każdy obiekt ma swoją własną kopię danych składowych. Funkcje składowe są związane z definicją klasy a nie z deklaracjami obiektów tej klasy. Cechą charakterystyczną funkcji składowych klasy jest fakt, że w każdej takiej funkcji jest zadeklarowany wskaźnik specjalny this jako:

```
X_klas *const this;
```

gdzie X_klas jest nazwą klasy. Ten wskaźnik jest inicjalizowany wskaźnikiem do obiektu, dla którego wywołano funkcję składową. Wskaźnik this zadeklarowany jest jako *const, co oznacza, że nie można go zmieniać. Wskaźnik this jest niejawnie używany podczas odwoływania się zarówno do danych jak i funkcji składowych obiektu. Wskaźnik this jest przekazywany do obiektu jako niejawny argument każdej niestatycznej funkcji składowej wywoływanej na rzecz obiektu. Używanie wskaźnika this w odwołaniach nie jest konieczne, ale często stosuje się go do pisania metod, które działają bezpośrednio na wskaźnikach. Wskaźnik this może być wykorzystywany także w sposób jawny, co jest pokazane w kolejnym przykładzie.

Listing 4.7. Wskaznik this

```
1 #include <iostream.h>
  #include <conio.h>
3
  class Liczba
5 { public:
      Liczba(float = 0);      //konstruktor domyslny
7     void pokaz();
    private:
9     float nr;
  };
11
   Liczba :: Liczba(float xx) { nr = xx; } //konstruktor
13
   void Liczba :: pokaz()
15   {
      cout << "         nr = " << nr << endl;
17    cout << "    this -> = " << this -> nr << endl;
      cout << " (*this).nr = " << (*this).nr << endl;
19   }

21 int main()
  {  Liczba obiekt(1.01);
23   obiekt.pokaz();
          getch();
```

```
25          return 0;
   }
```

Elementami klasy Liczba są dwie metody publiczne i prywatna dana float nr. Wskaźnik this został wykorzystany przez funkcję składową pokaz() do wydrukowania prywatnej danej składowej nr egzemplarza klasy:

```
void Liczba :: pokaz()
  {
   cout << "          nr = " << nr << endl;
   cout << "    this -> = " << this -> nr << endl;
   cout << " (*this).nr = " << (*this).nr << endl;
  }
```

Funkcja składowa pokaz() trzy razy wyświetla wartość danej składowej nr. W pierwszej instrukcji mamy klasyczną metodę – bezpośredni dostęp do wartość nr. Funkcja składowa ma dostęp nawet do prywatnych danych swojej klasy. W drugiej instrukcji:

```
cout << "    this -> = " << this -> nr << endl;
```

wykorzystano operator strzałki do wskaźnika. Jest to popularna jawna metoda stosowania wskaźnika this. W następnej instrukcji zastosowano operator kropki dla wskaźnika po dereferencji:

```
cout << " (*this).nr = " << (*this).nr << endl;
```

Jak już mówiono nawias w zapisie (*this) jest konieczny.

4.8. Wskaźnik this – kaskadowe wywołania funkcji

Wskaźnik this umożliwia tzw. kaskadowe wywołania funkcji składowych. W programie przedstawione zostało zwracanie referencji do obiektu klasy Punkt3D. W pliku punkt3D.cpp, który zawiera definicje funkcji składowych Punkt3D każda z funkcji:

```
ustawPunkt ( int, int, int);     //ustawia caly punkt
ustawX (int );                   //ustawia X
ustawY (int );                   //ustawia Y
ustawZ (int );                   //ustawia Z
```

zwraca *this typu Punkt3D &. Biorąc po uwagę, że operator kropki wiąże od strony lewej do prawej, wiemy, że w wyrażeniu:

```
p1.ustawX(10).ustawY(20).ustawZ(30);
```

najpierw będzie wywołana funkcja ustawX(10), która zwróci referencję do obiektu p1, dzięki czemu pozostała część przybiera postać:

```
p1.ustawY(20).ustawZ(30);
```

Kolejno wywołana funkcja ustawY (20) zwróci ponownie referencje do obiektu p1 i mamy ostatecznie:

```
p1.ustawZ(30);
```

W ten sam sposób realizowane jest wywołanie kaskadowe w wyrażeniu:

```
p1.ustawPunkt( 50, 50, 50).pokaz();
```

W tym wyrażeniu należy zwrócić uwagę na zachowanie kolejności. Funkcja pokaz() nie zwraca referencji do p1, wobec tego musi być umieszczona na końcu.

Listing 4.8. Kaskadowe wywołanie metod - deklaracja klasy - plik punkt3D.h

```
1  //klasa Punkt3D

3  #ifndef _PUNKT3D_H
   #define _PUNKT3D_H
5
   class Punkt3D
7  {
     public:
9      Punkt3D (int = 0, int = 0, int = 0); //konstruktor
       Punkt3D &ustawPunkt ( int, int, int);//ustawia
11                           //caly punkt
       Punkt3D &ustawX (int );        //ustawia X
13     Punkt3D &ustawY (int );        //ustawia Y
       Punkt3D &ustawZ (int );        //ustawia Z
15
       int pobierzX () const;   //pobiera X
17     int pobierzY () const;   //pobiera Y
       int pobierzZ () const;   //pobiera Z
19     void pokaz () const;     //pokazuje wspolrzedne
                      //punktu X,Y,Z
21
   private:
23     int x;
       int y;
25     int z;
   };
27 #endif
```

Listing 4.9. Kaskadowe wywołanie metod - definicja klasy - plik punkt3D.cpp

```
1 // definicje klasy Punkt3D

3 #include "punkt3D.h"
  #include <iostream.h>
5
  Punkt3D :: Punkt3D (int xx, int yy, int zz)
7             { ustawPunkt ( xx, yy, zz); }

9 Punkt3D &Punkt3D :: ustawPunkt (int x1, int y1, int z1)
   {
11  ustawX ( x1 );
    ustawY ( y1 );
13  ustawZ ( z1 );
    return *this;
15 }

17 Punkt3D &Punkt3D :: ustawX ( int x1)
   { x = x1; return *this;}
19 Punkt3D &Punkt3D :: ustawY ( int y1)
   { y = y1; return *this;}
21 Punkt3D &Punkt3D :: ustawZ ( int z1)
   { z = z1; return *this;}
23
  int Punkt3D :: pobierzX () const  {return x; }
25 int Punkt3D :: pobierzY () const  {return y; }
  int Punkt3D :: pobierzZ () const  {return z; }
27
  void Punkt3D :: pokaz () const
29 {cout << " x = " << x << endl;
    cout << " y = " << y << endl;
31  cout << " z = " << z << endl;
   }
```

Listing 4.10. Kaskadowe wywołanie metod - testowanie - plik ftest1.cpp

```
  //Kaskadowe wywolanie metod
2 #include <iostream.h>
  #include <conio.h>
4 #include "punkt3D.h"

6 int main()
  { Punkt3D p1, p2;
8   p1.ustawX(10).ustawY(20).ustawZ(30);
    cout << " punkt startowy p1" << endl;
10  p1.pokaz();
    cout << "   nowy punkt p1  " << endl;
12  p1.ustawPunkt( 50, 50, 50).pokaz();
```

```
    cout << "   inny punkt p2  " << endl;
14  p2.ustawPunkt (100, 100, 100).pokaz();
        getch();
16      return 0;
  }
```

Wynikiem działania programu jest następujący wydruk:

```
punkt startowy p1
x = 10
y = 20
z = 30
   nowy punkt p1
x = 50
y = 50
z = 50
   inny punkt p2
x = 100
y = 100
z = 100
```

4.9. Tablice obiektów

Obiekty klas są zmiennymi definiowanymi przez użytkownika i działają analogicznie jak zmienne typów wbudowanych. Zgodnie z tym możemy grupować obiekty w tablicach. Deklaracja tablicy obiektów jest identyczna jak deklaracja tablicy zmiennych innych typów. Dla przypomnienia, w języku C++ tablice obiektów wbudowanych mają postać:

```
int temp[50];         // 50 elementowa tablica typu int
float waga [100];     // 100 elementowa tablica typu float
char *tabWsk [10];    //10 elementowa tablica wskaznikow do char
```

W podobny sposób tworzona jest tablica obiektów jakiejś klasy. Niech klasa punkt ma bardzo prostą postać;

```
class punkt
{ public:
    int x;  int y;
};
```

Tablica obiektów klasy punkt może mieć postać;

```
punkt ptab[10];
```

Powyższą definicję możemy czytać w następujący sposób: ptab jest 10 elementową tablicą obiektów klasy punkt. W tej prostej klasie wszystkie dane są publiczne, więc dostęp do nich jest następujący:

```
cout << ptab[1].x ;
cout << ptab[5].y;
```

Dostęp do elementu tablicy realizowany jest przy pomocy dwóch operatorów. Właściwy element tablicy jest wskazywany za pomocą operatora indeksu [], po czym stosujemy operator kropki (.) wydzielający określoną zmienną składową obiektu. Możemy także zdefiniować wskaźnik, który będzie wskazywał na obiekty klasy punkt:

```
punkt *pWsk ;
```

W instrukcji:

```
pWsk = & ptab[5];
```

ustawiono wskaźnik tak aby wskazywał na konkretny element tablicy. Przy pomocy wskaźnika możemy odwołać się do danych klasy:

```
pWsk -> x;
```

W pokazanym programie tworzona jest tablica obiektów prostej klasy punkt:

```
class punkt
{ public:
      int x; int y;
};
```

Zasadniczym problemem jest inicjalizacja, czyli nadawanie wartości początkowych w momencie definicji obiektu. Mogą wystąpić trzy przypadki:
— tablica jest agregatem
— tablica nie jest agregatem
— tablica jest tworzona dynamicznie (operator new)

Agregatem nazywamy tablicę obiektów będących egzemplarzami klasy, która nie ma danych prywatnych i nie ma konstruktorów. W naszym przykładzie mamy do czynienia z agregatem. W takim przypadku tablica obiektów jest inicjalizowana dość prosto: listę inicjalizatorów umieszczamy w nawiasach klamrowych i oddzielamy przecinkami:

```
punkt ptab[5] =     //inicjalizacja tablicy
{ 0, 0,             //x i y dla ptab[0]
1, 1,               //x i y dla ptab[1]
2, 2,               //x i y dla ptab[2]
3, 3,               //x i y dla ptab[3]
10, 20              //x i y dla ptab[4]
};
```

Oto przykład tworzenia tablicy obiektów i jej inicjalizowanie. W programie zademonstrowano także użycie wskaźników do elementów obiektu.

Listing 4.11. Tablica obiektów - agregaty

```
#include <iostream.h>
#include <conio.h>
class punkt
{public:
   int x; int y;
};

int main()
{punkt *pWsk;                 //wskaznik na obiekt typu punkt
 punkt ptab[5] =              //inicjalizacja tablicy
          { 0, 0,             //x i y dla ptab[0]
            1, 1,             //x i y dla ptab[1]
            2, 2,             //x i y dla ptab[2]
            3, 3,             //x i y dla ptab[3]
            10, 20            //x i y dla ptab[4]
          };
 cout << "x       y " << endl;
 for (int i =0; i<5; i++)
  cout << ptab[i].x << "      " << ptab[i].y << endl;
 pWsk = &ptab[4];      //wskaznik inicjalizowany adresem
 cout << "dostep przez wskaznik  x = " << pWsk -> x ;
           getch();
          return 0;
}
```

Efektem działania programu jest następujący wydruk:

```
x       y
0       0
1       1
2       2
3       3
10       20
dostep przez wskaznik  x = 10
```

4.10. Inicjalizacja tablic obiektów nie będących agregatami

W przypadku, gdy mamy do czynienia z danymi prywatnymi w danej klasie, aby zainicjalizować tablicę obiektów musimy posłużyć się konstruktorem. Program ilustruje zagadnienie inicjalizacji tablicy obiektów w takim przypadku. Należy zwrócić uwagę na deklarację klasy – widzimy tam zwykły konstruktor i konstruktor domniemany.

Listing 4.12. Tablica obiektów - tablica nie jest agregatem

```
#include <iostream.h>
#include <conio.h>
class punkt
{private:
        int x; int y;
  public:
        punkt ( int xx, int yy );      //konstruktor
        punkt ( );                     //konstruktor domyslny
        void pokaz ();
};
punkt :: punkt (int xx, int yy) : x(xx), y(yy) {};
punkt :: punkt ( ) { x = 0;  y = 0; }
void punkt :: pokaz()
   { cout << x << "   " << y << endl; }
int main()
{ const int nr = 5;        //ilosc punktow,rozmiar tablicy
  punkt ptab[nr] =        //tablica obiektow
    {
     punkt (15, 15),        //konstruktor
     punkt (10, 10),
     punkt (20, 20),
     punkt (),                   //konstruktor domyslny
     punkt (1, 1 )
    } ;
 cout << "x      y " << endl;
 for (int i =0; i<nr; i++)
            ptab[i].pokaz();
  getch();
  return 0;
}
```

Efektem działania programu jest wydruk:

```
x      y
15     15
10     10
20     20
0   0
1   1
```

Definicje konstruktorów mają postać:

```
punkt :: punkt (int xx, int yy) : x(xx), y(yy) {};
punkt :: punkt ( ) { x = 0;  y = 0; }   //konstruktor domyslny
```

W definicji konstruktora należy zwrócić uwagę na inicjalizację składowych x i y – wykorzystano listę inicjalizacyjną poprzedzoną dwukropkiem. Definicja 5 elementowej tablicy obiektów klasy punkt ma postać:

```
const int nr = 5;        //ilosc punktow,rozmiar tablicy
punkt ptab[nr] =     //tablica obiektow
  {
   punkt (15, 15),       //konstruktor
   punkt (10, 10),
   punkt (20, 20),
   punkt (),             //konstruktor domyslny
   punkt (1, 1 )
  } ;
```

Widzimy, że pomiędzy nawiasami klamrowymi umieszczona jest lista inicjalizatorów, oddzielonych przecinkami. Poszczególne wywołania konstruktorów inicjalizują tablicę obiektów. Konstruktor domniemany nie jest konieczny, ale jest dobrym zwyczajem umieszczać go. W sytuacji, gdy na przykład tablica ma 10 elementów a na liście inicjalizatorów umieszczono jedynie 5 konstruktorów, wtedy kompilator niejawnie wywołuje dla pozostałych elementów tablicy obiektów konstruktor domyślny. Gdy w takiej sytuacji nie będzie konstruktora domyślnego, kompilator zasygnalizuje błąd.

4.11. Tablice obiektów tworzone dynamicznie

W przypadku tworzenia tablic obiektów w pamięci swobodnej (tablice dynamiczne) należy pamiętać, że nie można ich jawnie inicjalizować. Obowiązują zasady:
— klasa nie ma żadnego konstruktora
— klasa ma konstruktor domniemany
Inicjowanie tablic obiektów w takich przypadkach odbywa się za pomocą funkcji składowych. Pokazany program tworzy tablicę 5 obiektów typu punkt w pamięci swobodnej:

```
const int nr = 5; //rozmiar tablicy
punkt *twsk;      //wskaznik
twsk = new punkt[nr]; //tablica obiektow
```

W powyższym fragmencie mamy deklarację tablicy twsk zawierającą 5 elementów typu punkt. Cała tablica jest tworzona na stercie, za pomocą wywołania new punkt[nr]. Usunięcie tablicy twsk automatycznie zwróci całą przydzieloną pamięć gdy zostanie użyty operator delete:

```
delete [] twsk;
```

Listing 4.13. Tablica obiektów - alokacja dynamiczna

```
1 #include <iostream.h>
```

```
#include <conio.h>

class punkt
{
  private:
     int x,y;
  public:
     punkt () {};            //konstruktor domyslny
     void set (int xx, int yy)      //funkcja skladowa
            { x = xx; y = yy; }
     int getX () { return x; }      //akcesor
     int getY () { return y; }
};

int main()
{
  const int nr = 5;
  punkt *twsk;
  twsk = new punkt[nr];
  if (!twsk)
    { cerr << "nieudana alokacja\n";
      return 1;
     }
  for (int i = 0; i < nr; i++)
   {
    twsk[i].set(i+1, i+3);
    cout << twsk[i].getX()<< "   " << twsk[i].getY() <<endl;
    }
  delete [] twsk;
        getch();
        return 0;
}
```

Efektem działania programu jest wydruk:

```
1     3
2     4
3     5
4     6
5     7
```

Inicjowanie tablicy obiektów realizowane jest w pętli for przy pomocy funkcji składowej set(int, int):

```
twsk[i].set(i+1, i+3);
```

Wypisywanie wartości danych realizowane jest przy pomocy akcesorów getX() i getY():

```
    cout << twsk[i].getX()<< "   " << twsk[i].getY() <<endl;
```

4.12. Kopiowanie obiektów

Obiekty tej samej klasy mogą być przypisywane sobie za pomocą domyślnego kopiowania składowych. W tym celu wykorzystywany jest operator przypisania (=). Kopiowanie obiektów przez wywołanie domyślnego konstruktora kopiującego daje poprawne wyniki jedynie dla obiektów, które nie zawierają wskazań na inne obiekty, na przykład, gdy klasa wykorzystuje dane składowe, którym dynamicznie przydzielona jest pamięć operacyjna. W programie tworzone są dwa obiekty klasy Data o nazwach d1 i d2 :

```
Data d1(31, 1, 2002), d2;
```

Obiekt d1 jest inicjalizowany jawnym konstruktorem, zaś obiekt d2 inicjalizowany jest konstruktorem domyślnym.

Listing 4.14. Kopiowanie obiektów

```
1 #include <iostream.h>
  #include <conio.h>
3 class Data
  { public:
5   Data (int = 1, int = 1, int = 2000);
    void pokaz ();
7   private:
      int dzien;
9     int miesiac;
      int rok;
11 };
  Data::Data(int kd, int km, int kr)
13 {dzien = kd;
   miesiac = km;
15 rok = kr;
  }
17 void Data:: pokaz()
  { cout << dzien << "." << miesiac << "." << rok << endl;
19 }

21 int main()
  {  Data d1(31, 1, 2002), d2;
23    cout << " d1 = ";
      d1.pokaz();
25    cout << " d2 = " ;
      d2.pokaz();
27    d2 = d1;
      cout << "przypisanie, d2 = ";
```

```
29      d2.pokaz();
        cout << "tworzy obiekt d3 = ";
31      Data d3 = d1;
        d3.pokaz();
33              getch();
                return 0;
35  }
```

Efektem działania programu jest następujący wydruk:

```
 d1  = 31.1.2002
 d2  = 1.1.2000
przypisanie, d2  = 31.1.2002
```

W instrukcji:

```
d2 = d1;
```

mamy proste przypisanie, obiektowi d2 przypisane są składowe obiektu d1. W kolejnej instrukcji:

```
Data d3 = d1;
```

Tworzony jest nowy obiekt klasy Data o nazwie d3 i jemu przypisane są składowe obiektu d1.

4.13. Klasa z obiektami innych klas

Bardzo często klasy korzystają z obiektów innych klas. Tego typu konstrukcje nazywamy złożeniem (ang. composition). Tworzymy wieloplikowy program do rejestracji osób. Tworzymy dwie klasy Data i Osoba. Klasa Osoba zawiera składowe: imie, nazwisko, miasto oraz dataUrodzenia. Konstruktor ma postać:

```
Osoba::Osoba (char *wsimie, char *wsnazwisko,
              char *wsmiasto, int urDzien,
              int urMiesiac, int urRok)
               : dataUrodzenia (urDzien, urMiesiac, urRok)
```

Konstruktor ma sześć parametrów, znak dwukropka rozdziela listę parametrów od listy inicjatorów składowych. Inicjatory składowych przekazują argumenty konstruktora Osoba do konstruktorów obiektów składowych. Na kolejnych wydrukach pokazano deklaracje klas Osoba i Data, ich definicje i w piątym pliku pokazano program testujący. W klasie Osoba:

```
class Osoba
{ public:
    Osoba (char *, char *, char *, int , int , int);
    void pokaz() const;
  private:
    char imie[30];
    char nazwisko[30];
    char miasto[30];
    const Data dataUrodzenia;
};
```

widzimy składową dataUrodzenia, która jest obiektem klasy Data.

Listing 4.15. Klasa wykorzystuje obiekt innej klasy - plik pracow.h

```
   //pracow.h
 2 // deklaracja klasy Pracownik
   #ifndef PRACOW1_H
 4 #define PRACOW1_H
   #include "data1.h"
 6 class Osoba
   { public:
 8     Osoba (char *, char *, char *, int , int , int);
       void pokaz() const;
10   private:
       char imie[30];
12     char nazwisko[30];
       char miasto[30];
14     const Data dataUrodzenia;
   };
16 #endif
```

Występujące w kodzie dyrektywy preprocesora:

```
#ifndef PRACOW1_H
#define PRACOW1_H
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
#endif
```

zapobiegają wielokrotnego włączania tych samych plików do programu.

Listing 4.16. Klasa wykorzystuje obiekt innej klasy - plik pracow.cpp

```
   //pracow.cpp
 2 //definicje funkcji skladowych klasy Pracownik
   #include <iostream.h>
 4 #include <string.h>
   #include "pracow.h"
 6 #include "data1.h"
```

```
 8 Osoba::Osoba (char *wsimie, char *wsnazwisko,
             char *wsmiasto,int urDzien, int urMiesiac, int urRok)
10           : dataUrodzenia (urDzien, urMiesiac, urRok)
   { int dlugosc = strlen(wsimie);
12  strncpy ( imie, wsimie, dlugosc);
    imie[dlugosc] = '\0';
14  dlugosc = strlen(wsnazwisko);
    strncpy ( nazwisko, wsnazwisko, dlugosc);
16  imie[dlugosc] = '\0';
    dlugosc = strlen(wsmiasto);
18  strncpy ( miasto, wsmiasto, dlugosc);
    imie[dlugosc] = '\0';
20 }
   void Osoba::pokaz() const
22 {cout << nazwisko << ", " << imie << endl;
    cout << " miejsce urodzenia: " << miasto << endl;
24  cout << " Data urodzenia: ";
    dataUrodzenia.pokaz();
26  cout << endl;
   }
```

Do obsługi łańcuchów wykorzystano funkcje biblioteczne z pliku <string.h>:

```
int dlugosc = strlen(wsimie);
strncpy ( imie, wsimie, dlugosc);
imie[dlugosc] = '\0';
```

Funkcja strlen() podaje długość łańcucha, a funkcja strcpy() kopiuje ten łańcuch. W trzeciej linii instrukcja kończy tablicę znaków znakiem końca łańcucha `'\0'`. Nie przeprowadzono kontroli długości napisów. Dla wszystkich trzech napisów zarezerwowano tablice 30 elementowe, tzn. łańcuchy nie mogą zawierać więcej niż 29 znaków. Gdy imię lub nazwisko jest krótsze, wtedy zarezerwowane elementy tablicy niepotrzebnie będą zajmować pamięć. W zasadzie należałoby zastosować dynamiczny przydział pamięci.

Listing 4.17. Klasa wykorzystuje obiekt innej klasy - plik data1.h

```
 1 //data1.h, deklaracja klasy Data
  #ifndef DATA1_H
 3 #define DATA1_H
   class Data
 5 { public:
        Data (int = 1, int = 1, int = 1900);
 7      void pokaz() const;
     private:
 9    int dzien;
      int miesiac;
11    int rok;
   };
```

```
13 #endif
```

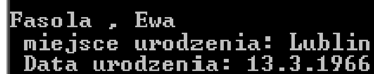
Listing 4.18. Klasa wykorzystuje obiekt innej klasy - plik data1.cpp

```
1  //data1.cpp, definicje funkcji skladowych klasy Data
   #include <iostream.h>
3  #include "data1.h"
   Data::Data(int d, int m, int r)
5  {dzien = d;
    miesiac = m;
7   rok = r;
   }
9
   void Data::pokaz() const
11 { cout << dzien << "." << miesiac << "." << rok;
   }
```

Listing 4.19. Klasa wykorzystuje obiekt innej klasy - plik pracow.cpp

```
   //klasa z obiektem innej klasy
2  #include <vcl.h>
   #include <iostream.h>
4  #include <conio.h>
   #include "pracow.h"
6
   int main()
8  {  Osoba os ("Ewa", "Fasola", "Lublin", 13,3,1966);
      cout << '\n';
10    os.pokaz();
          getch();
12        return 0;
      }
```

Efektem wykonania tego programu jest wydruk: Należy zwrócić uwa-



gę na dość skomplikowany sposób obsługi łańcuchów. Być może lepszym wyjściem byłoby wykorzystanie możliwości obiektów biblioteki string.

Rozdział 5

Dziedziczenie i hierarchia klas

5.1. Wstęp **84**
5.2. Hierarchiczna struktura dziedziczenia **84**
5.3. Notacja UML (Unified Modeling Language) **88**
5.4. Proste klasy pochodne **92**
5.5. Konstruktory w klasach pochodnych **98**
5.6. Dziedziczenie kaskadowe **104**
5.7. Dziedziczenie wielokrotne bezpośrednie **117**

5.1. Wstęp

Najistotniejszymi cechami programowania zorientowanego obiektowo są dziedziczenie (ang. inheritance), polimorfizm (ang. polimorphism) i dynamiczne związywanie (ang. dynamic binding). Dziedziczenie jest formą ponownego używania kodu uprzednio opracowanej klasy w nowotworzonej klasie pochodnej. Definiowana na nowo klasa przejmuje pewne cechy od innych zdefiniowanych już klas, dziedzicząc po nich wybrane metody i pola. W procesie tworzenia nowej klasy, programista może wykorzystać istniejącą już klasę i określić, że nowa klasa odziedziczy pewne dane i funkcje składowe. Istniejącą starą klasę nazywamy klasą podstawową lub bazową (ang. base class, parent class, or superclass), a nowa klasa dziedzicząca z klasy podstawowej dane i funkcje nazywana jest klasą pochodną (ang. derived class, child class, or subclass). Mówimy, że klasa pochodna dziedziczy wszystkie cechy swojej klasy bazowej. Dzięki mechanizmowi dziedziczenie w klasie pochodnej możemy:
— dodać nowe zmienne i funkcje składowe
— przedefiniować funkcje składowe klasy bazowej

Wynika z tego, że klasa pochodna jest bardziej szczegółowa od bazowej i jest większa, w tym sensie, że może mieć więcej danych i funkcji składowych niż klasa bazowa.

W języku C++ możliwe są trzy rodzaje dziedziczenia: publiczne, chronione oraz prywatne. Nas najbardziej będzie interesować dziedziczenie publiczne. Klasy pochodne nie mają dostępu do tych składowych klasy bazowej, które zostały zadeklarowane jako prywatne (private). W przypadku dziedziczenia publicznego dane i metody klasy bazowej zadeklarowane jako public i private stają się odpowiednio składowymi publicznymi oraz prywatnymi klasy pochodnej. Należy zwrócić także uwagę na fakt, że funkcje zaprzyjaźnione z klasą nie są dziedziczone, także konstruktory i destruktory nie są dziedziczone. Obiekty klas pochodnych mogą być traktowane jako obiekty klasy podstawowej.

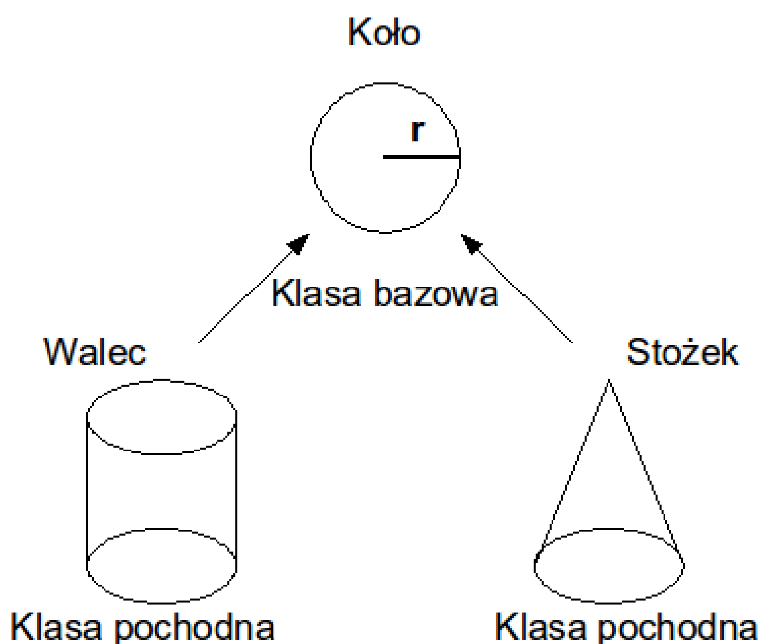
5.2. Hierarchiczna struktura dziedziczenia

Klasy pochodne mogą być tworzone na różne sposoby: mogą dziedziczyć cechy jednej klasy, mogą dziedziczyć cechy wielu klas. Klasa C może dziedziczyć cechy klasy B, która dziedziczy cechy klasy A. Możemy mieć także sytuację, gdy klasa C dziedziczy jednocześnie (bezpośrednio) cechy klasy A i B. Wykorzystując schemat dziedziczenia możemy budować hierarchiczne, wielopoziomowe struktury klas. Jeżeli w hierarchicznej strukturze dziedziczenia, każda klasa dziedziczy tylko cechy jednej klasy, to otrzymu-

jemy strukturę drzewiastą. Jeżeli klasa pochodna dziedziczy cechy wielu innych klas to otrzymujemy strukturę zwaną grafem acyklicznym.

Podstawowa klasyfikacja ma postać:
— Dziedziczenie pojedyncze (jednokrotne) (ang. simple inheritance)
— Dziedziczenie mnogie (wielokrotne) (ang. multiple inheritance)

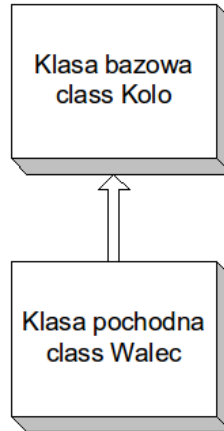
Rozpatrzmy teraz prostą strukturę dziedziczenia. Załóżmy, że nasz program musi obsłużyć figury geometryczne takie jak np.: koło, walec, stożek. Te figury mają wspólną cechę – promień. Relacje pomiędzy tymi figurami pokazuje rysunek. Cechy klasy bazowej opisującej koło, może przejąć klas pochodna opisująca walec oraz klasa pochodna opisująca stożek.



Rysunek 5.1. Relacje pomiędzy obiektami i struktura dziedziczenia

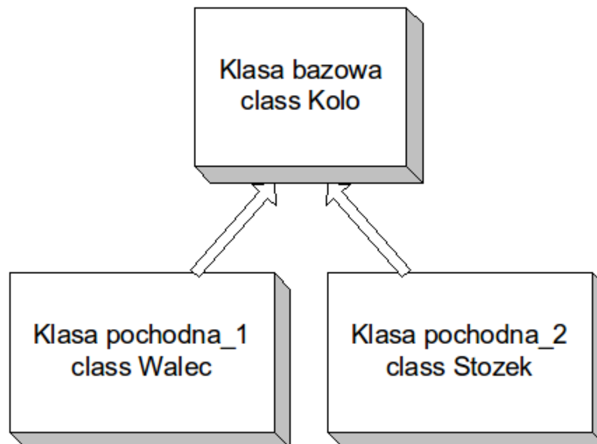
Rozpatrzmy program obsługujący koło i walec. Można przyjąć, że klasa modelująca koło będzie klasą bazową dla klasy opisującej walec. Schemat dziedziczenia pokazany jest na rysunku.

Zgodnie z konwencją, strzałka zawsze jest skierowana od klasy pochodnej do klasy bazowej. Relacja ukazana na rysunku 5.2 jest przykładem prostego dziedziczenia, każda klasa pochodna ma tylko jedną, bezpośrednią klasę bazową. Jeżeli w naszym programie zechcemy obsłużyć kolejną figurę, jaką jest stożek, to ponownie możemy skorzystać z klasy Kolo, która będzie klasą bazową dla nowej klasy pochodnej Stozek. Nowa hierarchia dziedzi-



Rysunek 5.2. Hierarchiczna struktura dziedziczenia – dziedziczenie proste

czenia pokazana jest na rysunku, w dalszym ciągu mamy do czynienia z dziedziczeniem prostym.



Rysunek 5.3. Hierarchiczna struktura dziedziczenia – dziedziczenie proste

Możemy opracować bardziej złożony program, np. program rysujący walec. Elementem figury geometrycznej takiej jak walec jest koło. Rysując koło na ekranie, musimy znać położenie środka koła. Wygodnie jest wiec opracować klasę Punkt. Taka klasa będzie obsługiwała współrzędne kartezjańskie punktu. Tworząc klasę obsługującą koło (nazwijmy tą klasę Kolo)możemy wykorzystać klasę Punkt. Klasa Punkt będzie klasą bazową klasy pochodnej Kolo. W tym przypadku mamy do czynienia z dziedziczeniem prostym (klasa Kolo dziedziczy bezpośrednio cechy klasy Punkt). Mając opracowana

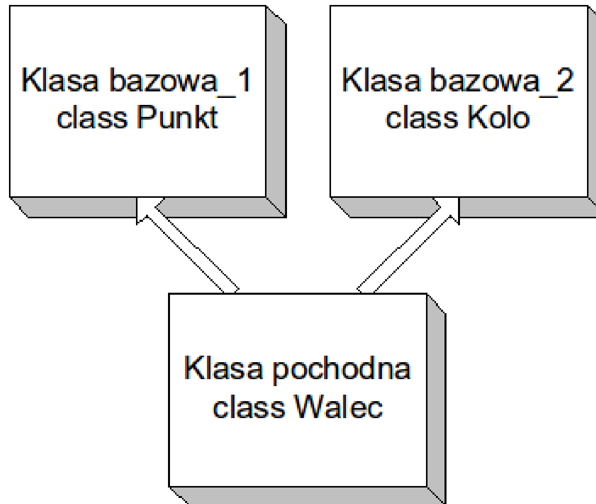
klasę Kolo, możemy opracować klasę obsługującą walec (nazwijmy ta klasę Walec). Klasa Walec dziedziczy bezpośrednio cechy klasy Kolo, klasa Kolo dziedziczy bezpośredni cechy klasy Punkt. Klasa Walec dziedziczy pośrednio cechy klasy Punkt. Mamy tu do czynienia z dziedziczeniem kaskadowym, typ dziedziczenia to dziedziczenie proste. Opisana hierarchiczna struktura dziedziczenia pokazana jest na rysunku 5.4.

Rysunek 5.4. Hierarchiczna struktura dziedziczenia – dziedziczenie proste, kaskadowe

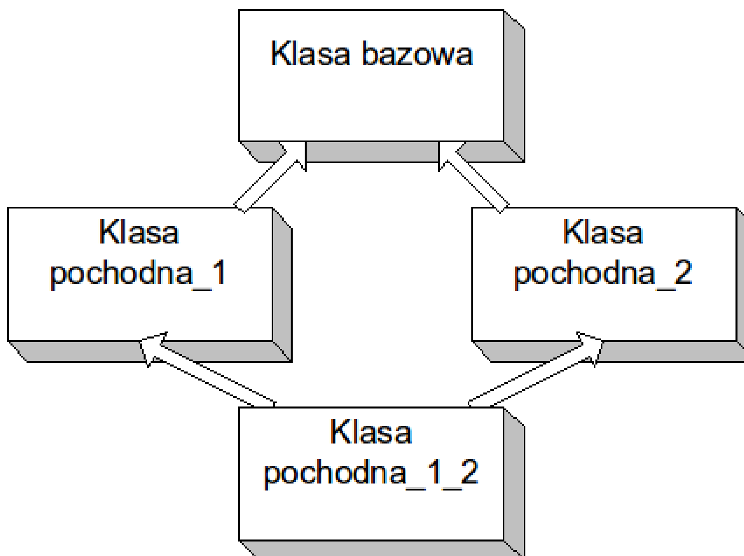
Opracowując program obsługi walca, mamy także możliwość innego zaprojektowania klasy Walec. Klasa Walec może jednocześnie dziedziczyć cechy klasy Punkt i cechy klasy Kolo. W takim przypadku mamy do czynienia z dziedziczeniem mnogim. Tego typu hierarchia dziedziczenia pokazana jest na rysunku 5.5

Możliwe są bardziej skomplikowane struktury dziedziczenia mnogiego. Przykład takiej rozbudowanej hierarchii pokazany jest na rysunku 5.6.

W praktyce hierarchiczne struktury dziedziczenia mogą być bardzo skomplikowane. Rozważmy program obsługujący osoby związane z uczelnią. Najbardziej ogólny podział to pracownicy uczelni i studenci. Pracownicy uczelni mogą dzielić się na pracowników administracji i nauczycieli akademickich. Tworząc model osób związanych z uczelnią (opracowując odpowiednie klasy) możemy zaprojektować strukturę dziedziczenia pokazaną na rysunku 5.7. Należy zwrócić uwagę na klasę P_Funkcyjny. Ta klasa ma za zadanie obsłużyć pracowników uczelni, którzy sprawują funkcje takie jak np. rektor czy dziekan. Są oni bardzo często jednocześnie pracownikami dydaktycznymi i pracownikami administracji. W tym przypadku projektując

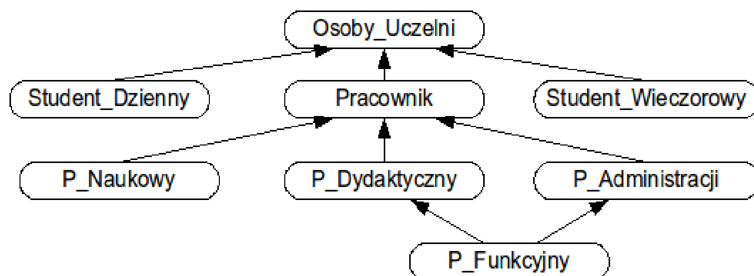


Rysunek 5.5. Hierarchiczna struktura dziedziczenia – dziedziczenie mnogie, dwu-bazowe



Rysunek 5.6. Hierarchiczna struktura dziedziczenia – dziedziczenie mnogie (klasa pochodna_1_2), skierowany graf acykliczny

klasę P_Funkcyjny korzystamy z dziedziczenia mnogiego (klasa P_Funkcyjny dziedziczy jednocześnie cechy klasy P_Dydaktyczny i P_Administracji).

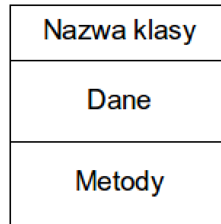


Rysunek 5.7. Hierarchiczna struktura dziedziczenia – model osób związanych z uczelnią

5.3. Notacja UML (Unified Modeling Language)

Jak widać z przytoczonych przykładów, hierarchia struktury klas może być całkiem skomplikowana, szczególnie gdy wzrasta ilość klas oraz ich wzajemne relacje związane z dziedziczeniem. Programy obiektowe są tworzone w postaci systemów współpracujących klas, które obejmują zarówno dane jak i metody. Bardzo często do uwidocznienia wzajemnych relacji pomiędzy klasami oraz danymi i metodami klas stosowany jest język UML (ujednolicony język programowania, ang. unified modeling language). UML jest językiem o szerokim zakresie zastosowań, można go używać w różnych typach systemów, dziedzin i procesów. Formalnie rzecz biorąc, UML jest językiem służącym do specyfikacji, wizualizacji, konstrukcji i dokumentacji artefaktów procesu zorientowanego na system. Specyfikacja obejmuje tworzenie modelu opisującego system. Model jest opisywany za pomocą diagramów. Dla naszych celów wykorzystamy notacje UML do opisu klas i obiektów. Dzięki użyciu odpowiednich diagramów skomplikowane relacje pomiędzy klasami bazowymi i pochodnymi staną się bardziej jasne. Obiekty w języku UML traktowane są jako egzemplarze klasy. Klasa jest formalnie typem obiektu. Klasa opisuje atrybuty oraz zachowanie obiektu. W języku UML klasa może być przedstawiona w postaci graficznej – tworzony jest diagram klasy. Jest to prostokąt podzielony na trzy części, każda część przechowuje inną informację:
— nazwa klasy
— atrybuty (dane)
— lista operacji (metody)
Ogólny sposób przedstawiania klasy w języku UML pokazany jest na rysunku.

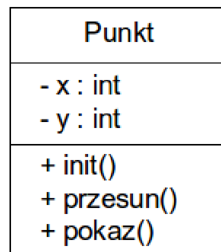
Jako przykład przedstawimy diagram klasy Punkt. Niech deklaracja klasy Punkt ma postać:



Rysunek 5.8. Ogólny diagram klasy

```
class Punkt
{
   private:
    int x;
    int y;
    public:
    void init(int, int);
    void przesun (int, int);
    void pokaz();
};
```

Diagram tej klasy pokazany jest na rysunku.



Rysunek 5.9. Diagram klasy Punkt z danymi i metodami

W pierwszej części widnieje nazwa klasy, w tym przypadku Punkt. W drugiej części przedstawione są dane (atrybuty). W klasie mamy dwa atrybuty: x i y. Zapis:

```
- x : int
```

informuje, że w klasie mamy atrybut o nazwie x, jest on typu całkowitego (int) oraz jego widoczność jest prywatna (znak "-"). W trzeciej części pokazane zostały operacje (metody). W klasie Punkt mamy trzy metody: init(), przesun() i pokaz(). Zapis:

```
+ init()
```

mówi, że w przedstawianej klasie mamy operację init() oraz widoczność tej operacji jest publiczna (znak "+"). W języku UML składnia atrybutu jest następująca:

```
widocznosc nazwa [licznosc porzadkowanie]
      : typ = wartosc_poczatkowa
```

Przykładem składni atrybutu może być następujący zapis:

```
− num_telefon [1..∗ ordered] : String
```

Składnik widoczność jest opcjonalny, informuje on o dostępności atrybutu. Mamy trzy możliwości oznaczenia: znak minus (-), znak plus (+) oraz znak hasz (#).

Znak minus (-) oznacza widoczność prywatną; atrybut jest niedostępny spoza swojej klasy.

Znak plus (+) oznacza widoczność publiczną; atrybut jest dostępny spoza klasy.

Znak hasz (#) oznacza widoczność chronioną; atrybut jest dostępny dla klas, które mają relacje generalizacji z jego klasą.

Typ jest opcjonalny, wskazuje typ danych, które może zawierać atrybut. W języku UML mamy następujące typy danych:
— Boolean – wartość logiczna
— Integer – liczba całkowita
— Real – liczba rzeczywista
— String – łańcuch (ciąg znaków)

W praktyce możemy w diagramach klas używać typów specyficznych dla danego języka. Np. w C++ możemy stosować typy podstawowe takie jaki int, double, float, char itp. Liczność jest opcjonalna, oznacza liczbę wartości, które mogą być przechowywane w atrybucie. Należy podać granicę dolną i granicę górną. Gdy granica górna jest nieskończona umieszczamy znak gwiazdki. Zapis 0..* oznacza od zera do nieskończonej liczby wartości. Gdy liczność jest równa 1 to jej nie podajemy (jest to wartość domyślna). Porządkowanie jest opcjonalne, mamy dwie możliwości:
— unordered – oznacza, że wartości nie są uporządkowane
— ordered – oznacza, że wartości są uporządkowane

Wartością domyślną jest unordered. Wartość początkowa jest opcjonalna. Gdy atrybut posiada wartość początkową piszemy ją po znaku =. Domyślnie atrybut nie ma wartości początkowej. Podobnie jak atrybuty opisywane są operacje (w trzeciej części diagramu klas). Przypominamy, że operacja jest tym, co obiekt z danej klasy może zrobić. Jest to specyfikacja usługi udostępnianej przez obiekt. Metoda oznacza sposób, w jaki obiekt z

danej klasy wykonuje swoje działanie. W języku UML składnia operacji jest następująca:

```
widocznosc nazwa (lista_parametrow) : typ_zwracany
```

Przykładem składni opisu operacji mogą być następujące zapisy:

```
+ ustawNazwisko (in Nazwisko:String)
+ pokazNazwisko () : String
```

Składnik widoczność jest opcjonalny, informuje on o dostępności operacji. Mamy trzy możliwości oznaczenia: znak minus (-), znak plus (+) oraz znak hasz (#).

Znak minus (-) oznacza widoczność prywatną; operacja jest niedostępna spoza swojej klasy. Znak plus (+) oznacza widoczność publiczną; operacja jest dostępna spoza klasy. Znak hasz (#) oznacza widoczność chronioną; operacja jest dostępna dla klas, które mają relacje generalizacji z jej klasą. Typ zwracany jest opcjonalny, wskazuje typ danych, zwracanych przez operację do tego, kto ją wywołał. Lista parametrów jest opcjonalna. Na liście umieszczamy parametry oddzielone przecinkami. Lista wyszczególnia parametry przekazywane do operacji i otrzymywane z operacji. Parametr ma rozbudowaną składnię:

```
rodzaj nazwa : typ = wartosc_poczatkowa
```

W tym zapisie rodzaj wskazuje kierunek przesyłania, mamy trzy możliwości: in, out i inout. Znaczenie symboli jest następujące:
- in oznacza, że parametr jest wejściowy i nie może być modyfikowany przez operację
- out oznacza, że parametr jest tylko wyjściowy, i może być modyfikowany przez operację
- inout oznacza, że parametr jest wejściowy i może być zmodyfikowany przez operację

5.4. Proste klasy pochodne

Zajmiemy się teraz deklaracją klasy pochodnej, która dziedziczy cechy klasy podstawowej. Omówimy podstawowe zagadnienia związane z dziedziczeniem prostym (dziedziczenie pojedyncze). Niech klasa bazowa nazywa się Punkt a pochodna Kwadrat. Deklaracja klasy pochodnej ma następującą postać:

```
class Kwadrat : specyfikator_dostepu Punkt
{
```

```
        //.......................
    };
```

Specyfikator_dostępu oznacza rodzaj dziedziczenia: public, private lub protected. Jeżeli mamy do czynienia z dziedziczeniem publicznym to piszemy:

```
    class Kwadrat : public Punkt
    {
        //.......................
    };
```

Podamy przykład zastosowania klasy pochodnej. Niech klasa bazowa o nazwie punkt ma dwie składowe prywatne x i y, które są współrzędnymi punktu na płaszczyźnie oraz funkcje składowe do obsługi tego punktu. Należy utworzyć nową klasę pochodną punktB od klasy bazowej punkt. Klasa pochodna punktB powinna zawierać nową funkcję, dzięki której będzie można obliczyć odległość punktu od początku układu współrzędnych. Jeżeli współrzędne punktu wynoszą x i y to odległość punktu od początku układu współrzędnych wyraża się wzorem:

$$r = \sqrt{x^2 + y^2} \tag{5.1}$$

Deklaracja klasy punkt zawarta w pliku "punkt.h" ma postać:

Listing 5.1. Dziedziczenie publiczne - plik "punkt.h"

```
//deklaracja klasy punkt

class punkt
{ double x, y;
  public:
      void init(double xx=0.0, double yy=0.0)
        { x = xx;
          y = yy;
        }
      void pokaz()
        { cout << "wspolrzedne: " << x << "  " << y << endl;
        }
      double wsX() { return x; }
      double wsY() { return y; }
};
```

Klasa punkt zawiera: dane prywatne x i y (są to współrzędne punktu), funkcję składową init(), funkcję składową pokaz() do wypisywania współrzędnych kartezjańskich punktu i dwie funkcje dostępu wsX() i wsY(). Na podstawie klasy bazowej punkt tworzymy klasę pochodną, która zawiera

dodatkową funkcję promien(). Na wydruku 5.2 pokazano klasę pochodną o nazwie punktB oraz funkcję testującą.

Listing 5.2. Dziedziczenie publiczne - klasa pochodna i funkcja testująca

```
1 //dziedziczenie publiczne ---
  #include <iostream.h>
3 #include <conio.h>
  #include <math.h>
5 #include "punkt.h"

7 class punktB : public punkt
  {
9  public :
     double promien()
11     { return sqrt ( wsX( ) * wsX( ) + wsY( )*wsY( ) );    }
  };
13
  int main()
15 {
          punktB a;
17        a.init ( 3, 4 );
          a.pokaz ( );
19        cout << "promien␣␣␣␣␣:␣" << a.promien ( );
          getch();
21        return 0;
  }
```

Po uruchomieniu programu mamy następujący komunikat:

```
wspolrzedne: 3  4
promien     : 5
```

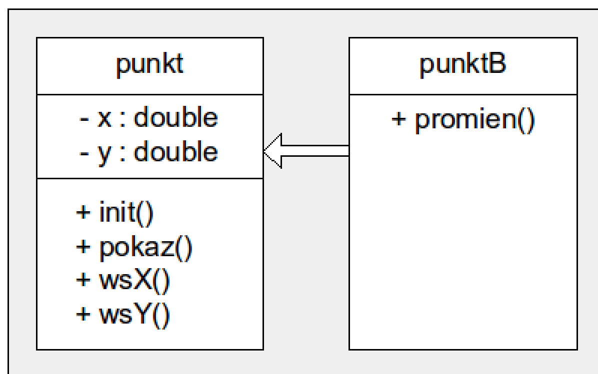
Klasa pochodna punktB ma postać:

```
class punktB : public punkt
{
 public :
   double promien()
     { return sqrt(wsX()*wsX() + wsY()*wsY());
     }
};
```

Zawiera ona funkcję składową promien (), która oblicza pierwiastek kwadratowy z sumy kwadratów współrzędnych punku. Jest to odległość punktu od początku układu współrzędnych.

W liniach:

```
punktB a;
```



Rysunek 5.10. Diagram klasy punkt i punktB w notacji UML. Klasa punktB dziedziczy składowe klasy punkt.

```
a.init ( 3, 4 );
a.pokaz ( );
```

tworzony jest obiekt a klasy punktB, współrzędnym x i y nadawane są wartości 3 i 4 oraz wywoływana jest funkcja składowa klasy bazowej pokaz() do wyświetlania współrzędnych punktu (obiektu a). W linii:

```
cout << "promien     : " << a.promien();
```

wywoływana jest funkcja składowa klasy pochodnej do obliczania odległości punktu od początku układu współrzędnych. Klasa punktB nie ogranicza, jak widać dostępu do dziedziczonych składowych klasy punkt. Oznacza to, że odziedziczone publiczne funkcje składowe takie jak na przykład init() mogą być wywoływane także dla obiektów klasy punktB.

Klasy pochodne mogą być specyfikowane jako publiczne, prywatne i chronione. Każda taka specyfikacja rodzi określone konsekwencje:

— Jeżeli specyfikatorem jest public, to wszystkie składowe publiczne w klasie bazowej są także składowymi publicznymi w klasie pochodnej. Klasa pochodna dziedziczy wszystkie składowe klasy bazowej.
— Jeżeli specyfikatorem jest private, to wszystkie składowe publiczne klasy bazowej są prywatnymi składowymi klasy pochodnej. Składowe prywatne klasy bazowej są niedostępne dla klasy pochodnej.
— Jeżeli specyfikatorem jest protected to klasa pochodna ma dostęp do składowych chronionych.

Składowe klasy bazowej mogą być typu private, protected i public. Z kolei klasa bazowa może być także zadeklarowana dla klasy pochodnej jako

private, protected i public. Wymaga to omówienia. Rozpatrzmy następującą klasę bazową punkt:

```
class  punkt
{
private:
  int x;
protected:
  int y;
public:
  int z;
};
```

Mamy następujące przypadki:
— Klasa bazowa jest publiczna. Dostępne publiczne składowe klasy bazowej nadal dostępne są publicznie w klasie pochodnej. Składowe o dostępie chronionym i prywatnym w klasie bazowej, zachowują odpowiedni rodzaj dostępu w klasie pochodnej. W naszym przykładzie zmienna y jest składową chronioną w klasie pochodnej (jest składową prywatną, ale dostępną dla klas pochodnych)
— Klasa bazowa jest prywatna. W tym przypadku dostęp do wszystkich składowych dziedziczonych po klasie bazowej staje się prywatny. Składowe te przestają być dostępne zarówno dla użytkownika klasy pochodnej, jak i dla jej klas pochodnych. W naszym przykładzie zmienne y i z stają się prywatnymi składowymi klasy pochodnej.
— Klasa bazowa jest chroniona. W takim przypadku wszystkie publiczne i chronione składowe klasy bazowej stają się chronionymi składowymi klasy pochodnej. W naszym przykładzie zmienne y i z staną się chronionymi składowymi klasy pochodnej.

Aby zilustrować powyższe rozważania utworzymy klasę bazową punkt, w której składowe x i y są chronione:

Listing 5.3. Dziedziczenie publiczne - plik "punkt.h" - dane typu protected

```
1  //deklaracja klasy punkt

3  class punkt
   { protected:
5        double x, y;
     public:
7        void init(double xx=0.0, double yy=0.0)
           {  x = xx;
9             y = yy;
           }
11       void pokaz()
           { cout << "wspolrzedne: " << x << "  " << y << endl;
13         }
```

```
         double wsX() { return x; }
15       double wsY() { return y; }
};
```

Klasa pochodna punktB ma dostęp do zmiennych x i y i wtedy funkcja składowa promien() istotnie się upraszcza:

```
class punktB : public punkt
{public :
   double promien()
     { return sqrt(x*x + y*y);
     }
};
```

Funkcja testująca razem z klasą pochodną punktB ma postać:

Listing 5.4. Klasa pochodna i funkcja testująca - dane typu protected

```
   //dziedziczenie publiczne dane protected ---
 2
   #include <iostream.h>
 4 #include <conio.h>
   #include <math.h>
 6 #include "punkt.h"

 8 class punktB : public punkt
   {
10  public :
      double promien()
12      { return sqrt(x*x + y*y);
        }
14 };

16 int main()
   {
18          punktB a;
            a.init(3,4);
20          a.pokaz();
            cout << "promien    : " << a.promien();
22          getch();
            return 0;
24 }
```

W tym przykładzie widzimy, że funkcja składowa promien() klasy pochodnej punktB ma prostą postać:

```
double promien()
{ return sqrt ( x * x + y * y ) ;
}
```

Ponieważ dane x i y są zadeklarowane jako protected, klasa pochodna ma bezpośredni dostęp do nich. W poprzednim przykładzie, dane x i y były zadeklarowane jako private, wobec czego funkcja składowa klasy pochodnej musiała mieć dość skomplikowaną postać.

Definicja tej funkcji (wydruk 5.2) miała postać:

```
double promien()
{ return sqrt ( wsX() * wsX() + wsY() * wsY() ) ;
}
```

5.5. Konstruktory w klasach pochodnych

Konstruktory i destruktory nie są dziedziczone. Konstruktory oraz destruktory można tworzyć dla klas bazowych, dla klas pochodnych oraz dla obu z nich jednocześnie. Gdy tworzony jest obiekt klasy pochodnej, wywoływany jest najpierw konstruktor klasy bazowej (o ile istnieje), a następnie konstruktor klasy pochodnej. Podczas tworzenia obiektu klasy pochodnej, aby zainicjować dane składowe klasy podstawowej musi zostać wywołany jej konstruktor. Jednak możliwe jest ich wywołanie odpowiednio w konstruktorach i operatorach przypisania klasy pochodnej. Wywoływanie konstruktora klasy bazowej zilustrujemy przykładem.

Listing 5.5. klasa pochodna i konstruktory

```
1  #include <iostream.h>
   #include <conio.h>
3
   class punkt                          //klasa bazowa
5  { public:
       punkt(double xx, double yy) {x = xx; y = yy;}
7      void pokazXY();
     private:
9      double x, y;
   };
11
   void punkt::pokazXY()
13      {cout << "x = " << "  " << x << "  y = " << y << endl;}

15 class punktB : public punkt          //klasa pochodna
   { public:
17      punktB(double k, double m, double n) : punkt(m, n)
         { z = k; }
19      void pokaz()
         { pokazXY();
21          cout << "z =   " << z << endl;
         }
```

```
23   private:
         double z;
25 };

27 int main()
   {punktB p1(5, 50, 50);
29   p1. pokaz();
           getch();
31         return 0;
   }
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
x = 50   y = 50
z =  5
```

Klasa bazowa punkt ma postać:

```
class punkt                        //klasa bazowa
{
  public:
    punkt(double xx, double yy) {x = xx; y = yy;}
    void pokazXY();
  private:
    double x, y;
};
```

Konstruktor klasy bazowej ma postać:

```
punkt ( double xx, double yy ) { x = xx; y = yy; }
```

i wymaga dwóch argumentów dla inicjacji składowych x i y. Klasa pochodna punktB zawiera dodatkową daną z i ma postać:

```
class punktB : public punkt        //klasa pochodna
{
  public:
     punktB(double k, double m, double n) : punkt(m, n)
      { z = k; }

     void pokaz()
      { pokazXY();
        cout << "z =   " << z << endl;
      }
  private:
     double z;
};
```

Konstruktor klasy pochodnej ma postać:

```
punktB ( double k, double m, double n ) : punkt ( m, n )
{ z = k; }
```

Konstruktor punktB musi wywołać konstruktor punkt, który jest odpowiedzialny za zainicjowanie tej części obiektu klasy punktB, która pochodzi z klasy podstawowej punkt. W tym celu został wykorzystany tzw. inicjator składowej. Zastosowano rozszerzoną postać deklaracji konstruktora klasy pochodnej, która pozwala przekazywać argumenty do jednego lub kilku konstruktorów klasy bazowej.

Formalna postać rozszerzonej deklaracji jest następująca:

```
konstruktor_klasa_pochodna ( lista_parametrow)
: nazwa_klasy_pochodnej_1(lista argumentow),
: nazwa_klasy_pochodnej_2(lista argumentow),
......................................
: nazwa_klasy_pochodnej_N(lista argumentow),
{
//cialo konstruktora klasy pochodnej
}
```

Przy pomocy znaku dwukropka oddzielamy deklarację konstruktora klasy pochodnej od specyfikacji klasy bazowej. Gdy mamy więcej klas bazowych dziedziczonych przez klasę pochodną, używamy przecinka, do oddzielenia ich specyfikacji. W naszym przykładzie mamy tylko jedną klasę bazową, która ma dwa parametry, wobec czego mamy napis:

```
: punkt(m,n)
```

Jak wiec widzimy, nasz konstruktor klasy pochodnej jest funkcją o trzech argumentach, dwa argumenty muszą być przesłane do konstruktora klasy bazowej. W momencie utworzenia obiektu p1:

```
punktB  p1 ( 5, 50, 50 );
```

następuje przekazanie argumentów aktualnych 50 i 50 do konstruktora klasy bazowej i wykonanie konstruktora:

```
punkt ( xx, yy )
```

a następnie wykonanie konstruktora klasy pochodnej. Bardzo często zachodzi sytuacja, gdy tworzone są obiekty klasy pochodnej a klasa bazowa i klasa pochodna zawierają różne składowe. Jeżeli tworzony jest obiekt klasy pochodnej to kolejność wywoływania konstruktorów jest następująca:
— Wywoływane są konstruktory obiektów klasy bazowej
— Wywoływany jest konstruktor klasy bazowej

— Wywoływany jest konstruktor klasy pochodnej
— Destruktory wywoływane są w odwrotnej kolejności

Zagadnienie kolejności wywoływania konstruktorów zilustrujemy kolejnym przykładem, w którym dwie klasy – bazowa i pochodna posiadają własne konstruktory i destruktory. Klasą bazową jest klasa punkt, której danymi chronionymi są współrzędne punktu, klasa pochodna kolo ma jedną daną prywatną, jest nią promień koła. Moment wywoływania konstruktorów i destruktorów będzie wypisywany na ekranie w trakcie wykonywania funkcji testującej. Dla celów dydaktycznych, program składa się z 5 plików: deklaracji klasy punkt, definicja klasy punkt, deklaracja klasy pochodnej kolo, definicja klasy kolo oraz funkcji testującej.

Zmienne x i y są danymi chronionymi klasy bazowej punkt. Definicja klasy pokazana jest na wydruku.

Listing 5.6. klasa pochodna - konstruktory

```
1  //plik "punkt.h"
   //deklaracja klasy punkt
3
   #ifndef _PUNKT_H
5  #define _PUNKT_H

7  class punkt
    {
9    public :
       punkt(double = 0.0, double = 0.0); //konstruktor domyslny
11     ~punkt();                          //destruktor
     protected:
13     double x, y;
    };
15
   #endif
```

Klasa posiada konstruktor domyślny i destruktor:

```
punkt ( double = 0.0, double = 0.0 );// konstruktor domyslny
~punkt();                            // destruktor
```

W definicji klasy bazowej punkt, konstruktor jak i destruktor wyświetlają komunikat o obiekcie, na rzecz którego zostały wywołane. Deklaracja klasy punkt umieszczona jest umieszczona w odrębnym pliku i pokazana na wydruku.

Listing 5.7. klasa pochodna - konstruktory

```
   //plik punkt.cpp
2  //definicja klasy punkt
```

```
  #include <iostream.h>
4 #include "punkt.h"

6 punkt::punkt(double xx, double yy)
  {
8   x = xx;        y = yy;
    cout << "konstruktor obiektu klasy punkt ";
10  cout << "x= " << x << "  y= " << y << endl;
  }
12
  punkt::~punkt()
14 {
    cout << " destruktor obiektu klasy punkt ";
16  cout << "x= " << x << "  y= " << y << endl;
  }
```

Klasa pochodna kolo dziedziczy od klasy bazowej punkt i jej deklaracja pokazana jest na wydruku. Składowa klasy pr zadeklarowana została jako prywatna.

Listing 5.8. klasa pochodna - konstruktory

```
1 //plik "kolo.h"
  //deklaracja klasy kolo
3
  #ifndef _KOLO_H
5 #define _KOLO_H

7 #include "punkt.h"

9 class kolo : public punkt
  {
11  public :
      kolo(double r=0.0, double xx=0.0, double yy=0.0);
13          //konstruktor domyslny
      ~kolo();      //destruktor
15  private:
      double pr;
17 };

19 #endif
```

Klasa kolo posiada konstruktor i destruktor:

```
kolo ( double r=0.0, double xx=0.0, double yy=0.0);
          //konstruktor domyslny
~kolo();      //destruktor
```

Na kolejnym wydruku pokazana jest definicja klasy pochodnej kolo.

Listing 5.9. klasa pochodna - konstruktory

```
1 //plik "kolo.cpp"
  //definicja klasy kolo
3 #include <iostream.h>
  #include "kolo.h"
5
  kolo::kolo(double r, double xx, double yy) : punkt(xx,yy)
7  {
    pr = r;
9   cout << "konstruktor obiektu klasy kolo ";
    cout << "pr= " << pr << " x= " << x <<
11          "  y= " << y << endl;
   }
13
   kolo::~kolo()
15 {
    cout << " destruktor obiektu klasy kolo ";
17  cout << "pr= " << pr << " x= " << x <<
            "  y= " << y << endl;
19 }
```

Konstruktor klasy kolo wywołuje równocześnie konstruktor klasy punkt :

```
kolo :: kolo ( double r, double xx, double yy )
      // konstruktor klasy kolo
      : punkt ( xx, yy ) // konstruktor klasy punkt
 {pr = r;
  cout << "konstruktor obiektu klasy kolo ";
  cout << "pr= " << pr << " x= " << x << "  y= "
      << y << endl;
 }
```

Listing 5.10. klasa pochodna; konstruktory; funkcja testująca

```
  #include <iostream.h>
2 #include <conio.h>
  #include "punkt.h"
4 #include "kolo.h"
  int main()
6 {
   {     cout <<"obiekt p1" << endl;
8          punkt p1(10, 20);
   }
10 {    cout << "obiekt k1" <<endl;
        kolo k1 (5, 100, 200);
12 }
   {    cout << "obiekt k2" <<endl;
14       kolo k2 (50, 150, 250);
```

```
        }
16          getch();
            return 0;
18 }
```

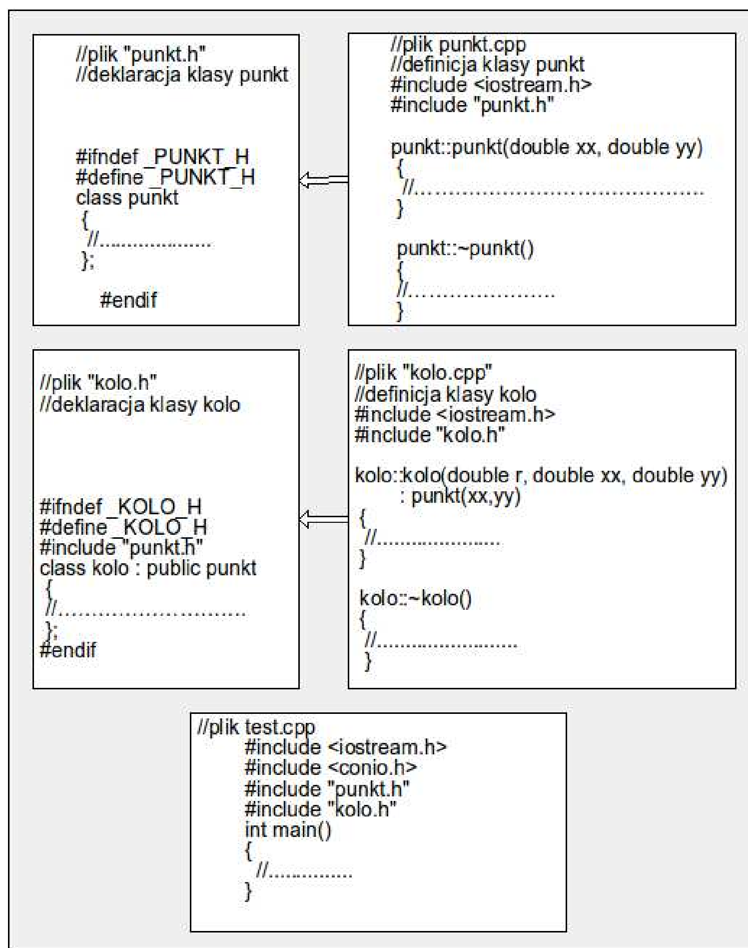
Przykładowy program pokazany na wydrukach 5.6 – 5.10 składa się z 5 plików, jego schemat fizyczny pokazano na rysunku. Fizyczną strukturę każdego programu C++ można określić jako zbiór plików. Niektóre z nich będą plikami nagłówkowymi(.h), a inne plikami implementacji (.cpp). Przyjmuje się, że komponent jest najmniejszym elementem projektu fizycznego. Strukturalnie komponent stanowi niepodzielną jednostkę fizyczną, której części nie mogą być użyte niezależnie od pozostałych. Komponent składa się z jednego pliku nagłówkowego i jednego pliku implementacji. Graficzne przedstawienie komponentów znacznie ułatwia zbadanie wzajemnych relacji pomiędzy plikami i bibliotekami. W omawianym przykładzie, w funkcji main() tworzony jest obiekt p1 klasy bazowej punkt. Następnie tworzone są kolejno dwa obiekty k1 i k2 klasy pochodnej kolo. Po uruchomieniu funkcji testującej mamy następujący wynik:

```
obiekt p1
konstruktor obiektu klasy punkt x= 10  y= 20
 destruktor obiektu klasy punkt x= 10  y= 20
obiekt k1
konstruktor obiektu klasy punkt x= 100  y= 200
konstruktor obiektu klasy kolo pr= 5 x= 100  y= 200
 destruktor obiektu klasy kolo pr= 5 x= 100  y= 200
 destruktor obiektu klasy punkt x= 100  y= 200
obiekt k2
konstruktor obiektu klasy punkt x= 150  y= 250
konstruktor obiektu klasy kolo pr= 50 x= 150  y= 250
 destruktor obiektu klasy kolo pr= 50 x= 150  y= 250
 destruktor obiektu klasy punkt x= 150  y= 250
```

Na początku tworzony jest egzemplarz obiektu klasy punkt. Wywoływany jest konstruktor a potem destruktor. Następnie tworzony jest obiekt k1 klasy pochodnej kolo. Wywoływany jest najpierw konstruktor klasy punkt, który pokazuje przekazane wartości a następnie wywołany jest konstruktor klasy kolo, który też pokazuje przekazane wartości. Kolejno wywoływany jest destruktor klasy koło i destruktor klasy punkt (wywoływanie destruktorów odbywa się w odwrotnej kolejności). Następnie utworzony zostaje kolejny obiekt k2 klasy kolo.

5.6. Dziedziczenie kaskadowe

Rozpatrywaliśmy dotychczas proste przypadki, gdy klasa pochodna miała tylko jedną klasę bazową. W bardziej rozbudowanych projektach może



Rysunek 5.11. Fizyczne diagramy komponentów (wydruki 5.6 -5.9) i programu testującego (wydruk 5.10)

się okazać, że potrzebne będzie utworzenie klasy, która będzie dziedziczyła cechy wielu klas. Możliwe są dwa schematy dziedziczenia:

— Klasa C dziedziczy od klasy B, klasa B dziedziczy od klasy A. Klasa C ma bezpośrednią klasę bazową B i pośrednią klasę bazową A. W ogólnym schemacie klasa pochodna może w ten sposób dziedziczyć cechy wielu klas, ale ma tyko jednego bezpośredniego przodka. W tym przypadku mówimy o dziedziczeniu kaskadowym.
— Proces bezpośredni. Klasa pochodna C ma dwie bezpośrednie klasy bazowe: A i B. W ogólnym przypadku, klasa pochodna może dziedziczyć jednocześnie cechy wielu klas. W tym przypadku mówimy o dziedziczeniu wielokrotnym (mnogim).

Dziedziczenie kaskadowe zilustrujemy przykładem. Najpierw utworzymy klasę punkt. Na jej podstawie tworzymy klasę pochodną kolo. Z klasy kolo tworzymy kolejną klasę pochodną walec.

W naszym przykładzie najpierw utworzymy klasę punkt i plik z programem testującym. Utworzymy trzy pliki: punkt.h, punkt.cpp, ftest1.cpp.

Listing 5.11. klasa punkt

```
// plik punkt.h
//definicja klasy punkt
#ifndef PUNKT_H
#define PUNKT_H
class punkt
{   public:
      punkt(double = 0.0, double = 0.0);//konstruktor domyslny
      void pokaz();                    //pokazuje wspolrzedne
    protected:
      double wX, wY;
};
#endif
```

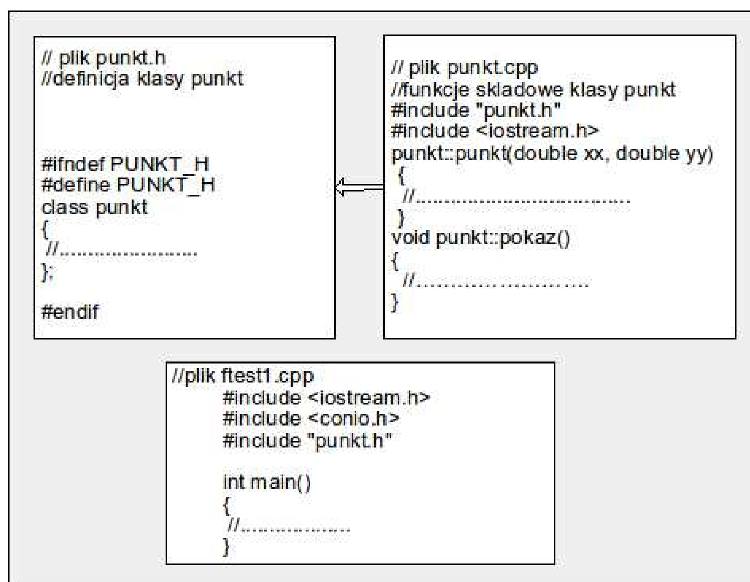
Listing 5.12. klasa punkt

```
// plik punkt.cpp
//funkcje skladowe klasy punkt
#include "punkt.h"
#include <iostream.h>
punkt::punkt(double xx, double yy) //konstruktor klasy punkt
 { wX = xx;  wY = yy;
 }
void punkt::pokaz()
{   cout << "x= " << wX << " y= " << wY << endl;
}
```

Listing 5.13. klasa punkt; funkcja main()

```
//plik ftest1.cpp
#include <iostream.h>
#include <conio.h>
#include "punkt.h"
int main()
{
 double wX,wY;
 cout << "Podaj X= " ;
 cin >> wX;
 cout << "Podaj Y= ";
 cin >> wY;
 punkt p1(wX, wY);
```

```
    p1.pokaz();
14          getch();
            return 0;
16  }
```

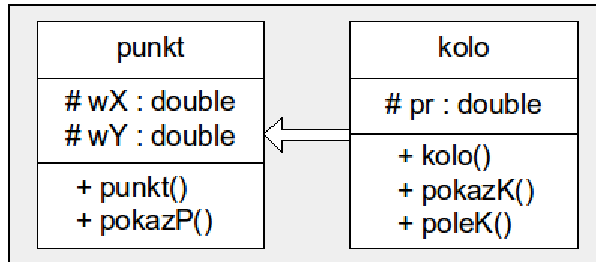


Rysunek 5.12. Fizyczne diagramy komponentów (wydruki 5.11 i 5.12) i programu testującego (wydruk 5.13). Program bez dziedziczenia klas.

W wyniku uruchomienia programu testującego mamy następujący wydruk:

```
Podaj X= 2
Podaj Y= 2
x= 2   y= 2
```

Rozbudujemy nasz projekt o możliwość obsłużenia obiektu typu koło. Najpierw tworzymy klasę punkt a potem na podstawie tej klasy tworzymy klasę kolo. Realizujemy model dziedziczenia jednokrotnego (prostego).



Rysunek 5.13. Diagram klasy punkt i kolo w notacji UML. Klasa kolo dziedziczy składowe klasy punkt.

Nasz projekt zbudowany jest z dwóch komponentów i pliku z funkcją main(). Fizycznie mamy pięć plików: punkt.h, punkt.cpp, kolo.h, kolo.cpp i ftest2.cpp. Klasa pochodna kolo dziedziczy cechy klasy bazowej punkt (rys.5.13).

Listing 5.14. dziedziczenie proste - klasa bazowa punkt

```
1  // plik punkt.h
   //definicja klasy punkt
3  #ifndef PUNKT_H
   #define PUNKT_H
5
   class punkt
7  {  public:
        punkt(double = 0.0, double = 0.0);
9             //konstruktor domyslny
        void pokazP();     //pokazuje wspolrzedne
11    protected:
        double wX, wY;
13 };
   #endif
```

Listing 5.15. dziedziczenie proste - klasa bazowa punkt

```
// plik punkt.cpp
//funkcje skladowe klasy punkt

#include "punkt.h"
#include <iostream.h>

                    //konstruktor klasy punkt
punkt::punkt(double xx, double yy)
 { wX = xx;   wY = yy;
 }

void punkt::pokazP()
{   cout << "x= " << wX << " y= " << wY << endl;
}
```

Listing 5.16. dziedziczenie proste - klasa pochodna kolo

```
// plik kolo.h
//definicja klasy kolo

#ifndef KOLO_H
#define KOLO_H

#include "punkt.h"

class kolo : public punkt
{
    public:
     kolo(double r = 0.0, double wX = 0.0, double wY = 0.0);
             //konstruktor
     void pokazK();        //pokazuje wspolrzedne
     double poleK() const;
   protected:
     double pr;
};

#endif
```

Listing 5.17. dziedziczenie proste - klasa pochodna kolo

```
//plik kolo.cpp
//definicje funkcji skaldowych klasy kolo

#include <iostream.h>
#include "kolo.h"

kolo::kolo(double r, double wX, double wY) : punkt(wX, wY)
{    pr = r;
```

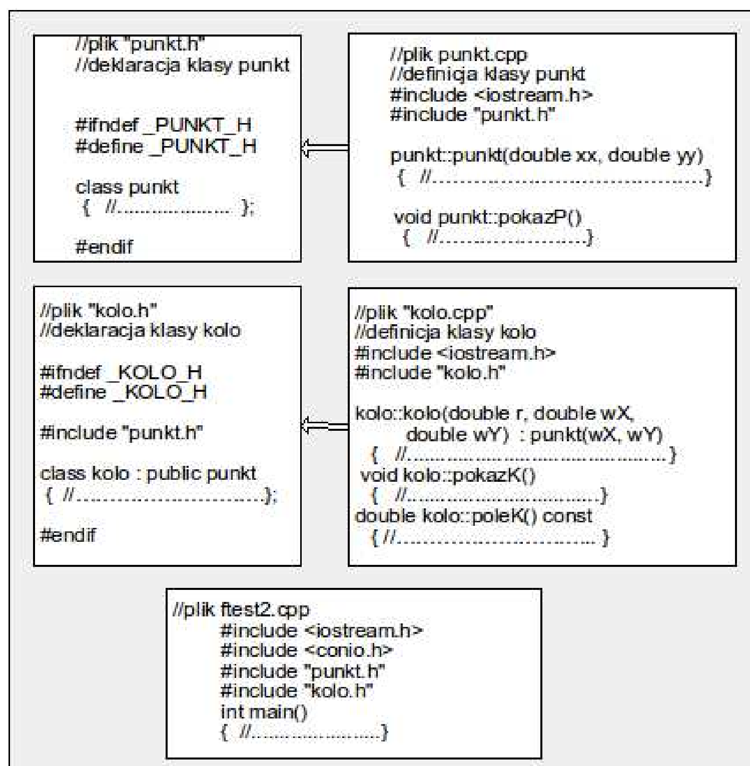
```
}

void kolo::pokazK()
{   cout << "x= " << wX << " y= " << wY << " r= "
        << pr << endl;
}

double kolo::poleK() const
 { return 3.1415926*pr*pr;
 }
```

Listing 5.18. dziedziczenie proste; klasa punkt i kolo; funkcja main()

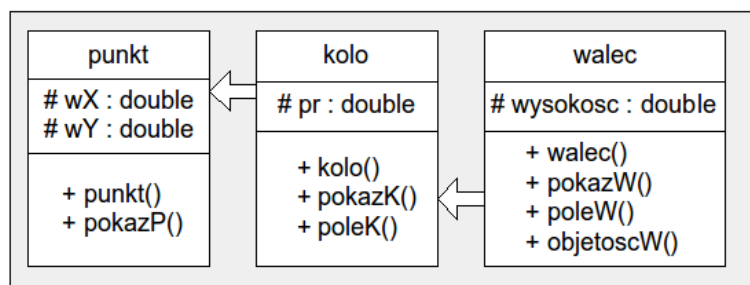
```
//program testujacy klase kolo
#include <iostream.h>
#include <conio.h>
#include "punkt.h"
#include "kolo.h"
int main()
{ double wX, wY, pr;
 cout << "Podaj X= " ;
 cin >> wX;
 cout << "Podaj Y= ";
 cin >> wY;
 cout << "Podaj promien, pr= ";
 cin >> pr;
 kolo k1(pr, wX, wY);
 k1.pokazK();
 cout <<"powierzchnia kola = " << k1.poleK() << endl;
          getch();
          return 0;
}
```

Komponenty projektu pokazano na rysunku 5.14. W wyniku wykonania programu testującego otrzymujemy następujący komunikat:

```
Podaj X= 1
Podaj Y= 1
Podaj promien, pr= 1
x= 1 y= 1 r= 1
powierzchnia kola = 3.14159
```



Rysunek 5.14. Fizyczne diagramy komponentów (wydruki 5.14, 5.15, 5.16 i 5.17) i pliku ftest2.cpp (wydruk 5.18). Program z dziedziczeniem prostym.



Rysunek 5.15. Diagram klasy punkt, kolo i walec w notacji UML. Jest to przykład dziedziczenia kaskadowego. Klasa walec dziedziczy składowe klasy kolo, klasa kolo dziedziczy składowe klasy punkt.

Obecnie zbudujemy projekt wykorzystujący dziedziczenie kaskadowe. Rozbudujemy poprzedni projekt przyjmując hierarchię punkt, kolo, walec. Najpierw utworzymy klasę punkt. Następnie tworzymy klasę kolo. Klasa

kolo dziedziczy cechy klasy punkt. W końcu tworzymy klasę walec. Klasa walec dziedziczy cechy klasy kolo. Aby sprawdzić działanie utworzonych klas tworzymy program testujący ftest3.cpp. Powstał projekt wieloplikowy, otrzymaliśmy następujące pliki: pliki punkt.h, punkt.cpp, kolo.h, kolo.cpp, walec.h, walec.cpp, ftest3.cpp. W naszym projekcie klasa walec jest klasą pochodną od klasy kolo, klasa kolo jest klasą pochodną od klasy punkt, jednocześnie jest klasą bazową dla klasy walec. Klasa punkt jest klasą bazową dla klasy kolo.

Listing 5.19. dziedziczenie kaskadowe; klasa bazowa punkt

```
1  // plik punkt.h, definicja klasy punkt
   #ifndef PUNKT_H
3  #define PUNKT_H

5  class punkt
   {
7    public:
       punkt(double = 0.0, double = 0.0);//konstruktor domyslny
9      void pokazP();                    //pokazuje wspolrzedne
     protected:
11     double wX, wY;
   };
13
   #endif
```

Listing 5.20. dziedziczenie kaskadowe; klasa bazowa punkt

```
   // plik punkt.cpp, funkcje skladowe klasy punkt
2
   #include "punkt.h"
4  #include <iostream.h>
   punkt::punkt(double xx, double yy)//konstruktor klasy punkt
6
    { wX = xx;  wY = yy;
8   }

10 void punkt::pokazP()
   {  cout << "x= " << wX << " y= " << wY << endl;
12 }
```

Listing 5.21. dziedziczenie kaskadowe; klasa pochodna kolo

```
   // plik kolo.h, definicja klasy kolo
2
   #ifndef KOLO_H
4  #define KOLO_H
   #include "punkt.h"
```

```
 6
   class kolo : public punkt
 8 {
     public:
10     kolo(double r = 0.0, double wX = 0.0, double wY = 0.0);
                                   //konstruktor
12     void pokazK();              //pokazuje wspolrzedne
       double poleK() const;
14   protected:
       double pr;
16 };
  #endif
```

Listing 5.22. dziedziczenie kaskadowe; klasa pochodna kolo

```
 1 //plik kolo.cpp, definicje funkcji skladowych klasy kolo

 3 #include <iostream.h>
   #include "kolo.h"
 5
   kolo::kolo(double r, double wX, double wY) : punkt(wX, wY)
 7 {    pr = r;
   }
 9
   void kolo::pokazK()
11 {   cout << "x= " << wX << " y= " << wY << " r= " << pr
          << endl;
13 }

15 double kolo::poleK() const
    { return 3.1415926*pr*pr;
17  }
```

Listing 5.23. dziedziczenie kaskadowe; klasa pochodna walec

```
 1 //plik walec.h, definicja klasy walec

 3 #ifndef WALEC_H
   #define WALEC_H
 5
   #include "kolo.h"
 7
   class walec : public kolo
 9  {
      public:
11      walec( double h = 0.0, double r = 0.0,
               double wX = 0.0, double wY = 0.0);
13             //konstruktor domyslny
        void pokazW();
```

```
15       double poleW() const ;   // oblicza powierzchnie walca
         double objetoscW() const; //oblicza objetowsc walca
17     protected:
         double wysokosc;        // wysokosc walca
19   };

21 #endif
```

Listing 5.24. dziedziczenie kaskadowe; klasa pochodna walec

```
1 //plik walec.cpp, definicje funkcji skladowych klasy walec

3 #include <iostream.h>
  #include "walec.h"
5
  walec::walec(double h, double r, double wX, double wY)
7         : kolo(r, wX, wY)
   {wysokosc = h ;   }
9
  void walec::pokazW()
11 {
   cout << "x= " << wX << " y= " << wY << endl;
13 cout << "promien= " << pr << " wysokosc= " << wysokosc
       << endl;
15 }

17 double walec::poleW() const
   {
19   return 2 * kolo::poleK() +
            2 * 3.1415926 * pr * wysokosc;
21 }

23 double walec::objetoscW() const
   {
25   return kolo::poleK() * wysokosc;
   }
```

Listing 5.25. dziedziczenie kaskadowe; klasa punkt; kolo; walec; funkcja main()

```
  //Program testujacy klase walec———
2 #include <iostream.h>
  #include <conio.h>
4
  #include "punkt.h"
6 #include "kolo.h"
  #include "walec.h"
8
  int main()
```

```
10 {
   double wX, wY, pr, wysokosc;
12 cout << "Podaj␣X=␣" ;
   cin >> wX;
14 cout << "Podaj␣Y=␣";
   cin >> wY;
16 cout << "Podaj␣promien,␣pr=␣";
   cin >> pr;
18 cout << "Podaj␣wysokosc,␣wysokosc=␣";
   cin >> wysokosc;
20 walec w1(wysokosc, pr, wX, wY);
   w1.pokazW();
22 cout << "powierzchnia␣walca=␣" << w1.poleW() << endl;
   cout << "objetosc␣walca=␣" << w1.objetoscW() << endl;
24
         getch();
26       return 0;
  }
```

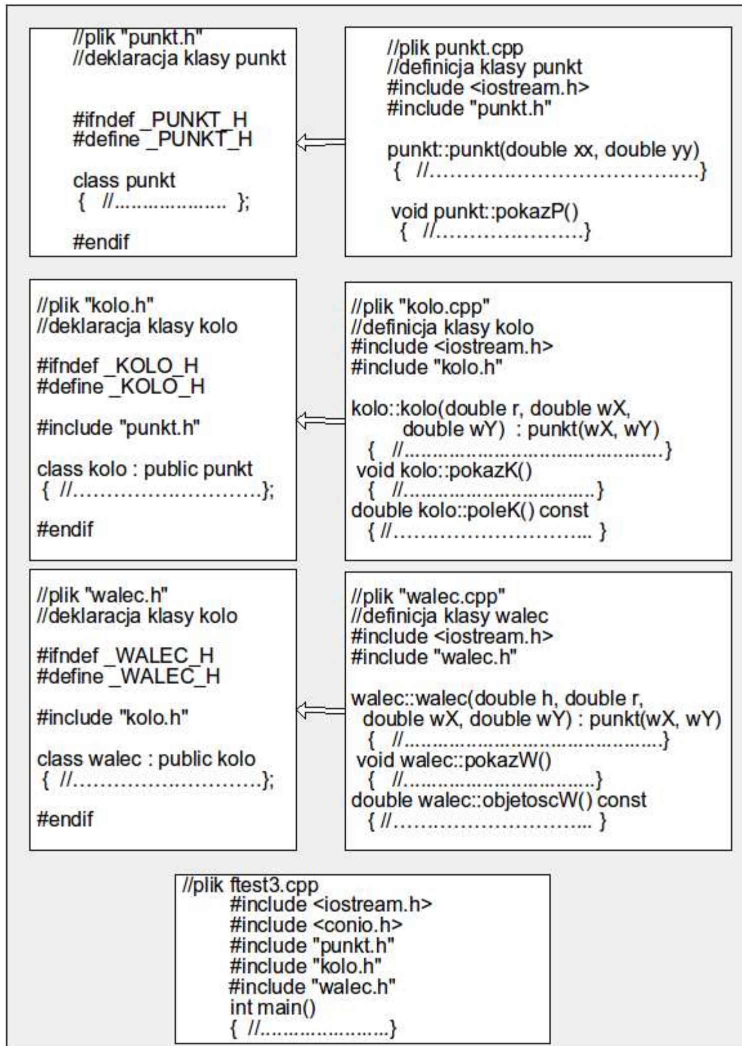
Program testujący tworzy obiekt w1 klasy walec:

```
walec w1(wysokosc, pr, wX, wY);
```

a następnie posługując się funkcjami pokazW(), poleW() i objetoscW() pokazuje dane i oblicza pole i objętość utworzonego walca. Kompletny wydruk otrzymany po uruchomieniu programu testującego ma postać:

```
Podaj X= 1
Podaj Y= 1
Podaj promien, pr= 1
Podaj wysokosc, wysokosc= 1
x= 1 y= 1
promien= 1 wysokosc= 1
powierzchnia walca= 12.5664
objetosc walca= 3.14159
```

Projekt fizyczny jest rozbudowany, składa się z 3 komponentów (każdy komponent to dwa pliki: plik nagłówkowy i plik implementacji) i pliku z programem testującym, w sumie mamy siedem plików. Diagramy komponentów i program testujący zostały pokazane na rysunku.



Rysunek 5.16. Fizyczne diagramy komponentów (wydruki 5.19 - 5.24) i pliku ftest3.cpp (wydruk 5.25). Program z dziedziczeniem kaskadowym.

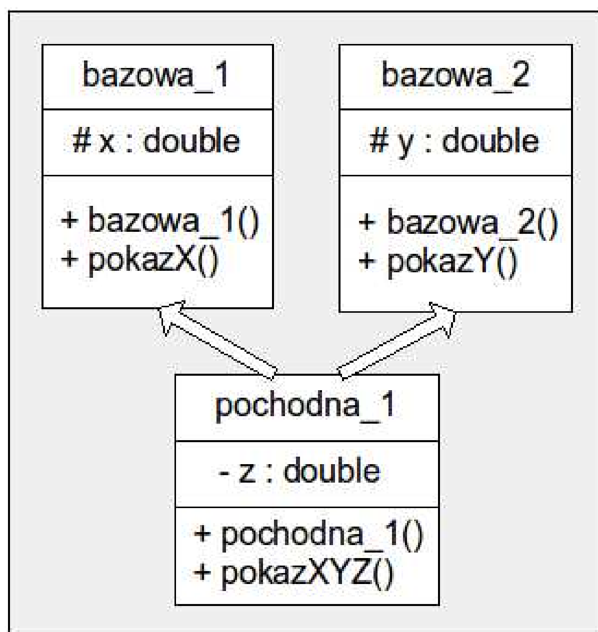
5.7. Dziedziczenie wielokrotne bezpośrednie

Klasa pochodna może być utworzona z kilku klas bazowych. Gdy klasa pochodna dziedziczy od np. dwóch klas bazowych to instrukcja deklaracji klasy pochodnej ma postać:

```
class pochodna_1 : public bazowa_1,  public bazowa_2
{
//cialo klasy pochodna
};
```

W tym przykładzie klasa pochodna nosi nazwę pochodna_1, nazwy publicznych klas bazowych to bazowa_1 i bazowa_2. Widzimy, że w przypadku dziedziczenia mnogiego, występuje lista klas bazowych, kolejne klasy bazowe poprzedzone są swoimi specyfikatorami dostępu i oddzielone są przecinkami.

Krótki przykład zilustruje zasadę dziedziczenia mnogiego. Utworzymy dwie klasy bazowe o nazwach bazowa_1 oraz bazowa_2, a następnie klasę pochodna o nazwie pochodna_1. Klasa pochodna_1 dziedziczy za pomocą dziedziczenia wielokrotnego z klas bazowa_1 i bazowa_2. Diagram klas pokazany jest na rysunku 5.17.



Rysunek 5.17. Diagram klasy bazowa_1, bazowa_2 i pochodna_1 w notacji UML. Jest to przykład dziedziczenia wielokrotnego. Klasa pochodna_1 dziedziczy jednocześnie składowe klasy bazowa_1 i klasy bazowa_2.

Klasa bazowa_1 zawiera jedną daną składową x typu protected, klasa bazowa_2 zawiera jedną daną składową y także typu protected. Klasa pochodna pochodna_1 zawiera jedna daną składową z typu private.

Listing 5.26. dziedziczenie mnogie; 2 klasy bazowe

```
//dziedziczenie mnogie———
#include <iostream.h>
#include <conio.h>

class bazowa_1
{   protected:
      double x;
    public:
      bazowa_1(double xx)                    //konstruktor  1
         { x = xx; }
      double pokazX() {return x; }               //zwraca x
};

class bazowa_2
{ protected:
      double y;
  public:
      bazowa_2(double yy)                    //konstruktor 2
        { y = yy; }
      double pokazY() { return y; }              //zwraca y
};

class pochodna_1 : public bazowa_1, public bazowa_2
{ private:
      double z;
  public:
      pochodna_1(double, double, double); //konstruktor 3
      void pokazXYZ()
        {   cout << " x= " << pokazX();
            cout << " y= " << pokazY();
            cout << " z= " << z << endl;
        };
  };

pochodna_1::pochodna_1(double a, double b, double c)
             : bazowa_1(a), bazowa_2(b) //konstruktor klasy
                       //pochodnej
        { z = c; }

int main()
{
  pochodna_1 ob1(1,11,111);
  ob1.pokazXYZ();
  cout << "x= " << ob1.pokazX() << endl;
  cout << "y= " << ob1.pokazY() << endl;
```

```
46          getch();
            return 0;
48 }
```

Po uruchomieniu tego programu otrzymujemy następujący wydruk:

```
x= 1   y=11   z=111
x= 1
y= 11
```

W funkcji testującej tworzony jest obiekt o nazwie ob1 z parametrami 1,11 i 111 :

```
pochodna_1 ob1(1,11,111);
```

Wyspecyfikowane parametry zostają przekazane w odpowiedniej kolejności do konstruktorów klas bazowych:

```
public:
 bazowa_1 ( double xx )          // konstruktor 1
 { x = xx; }
public:
 bazowa_2 ( double yy )          //konstruktor 2
 { y = yy; }
```

Zostają utworzone obiekty klas bazowa_1 i bazowa_2. Następnie wykonany zostanie konstruktor klasy pochodnej:

```
public:
  pochodna_1 ( double, double, double ) ;      // konstruktor 3
```

a konkretnie konstruktor

```
public:
  pochodna_1 ( 1, 11, 111 );              // konstruktor 3
```

W rezultacie zostaje utworzony i zainicjalizowany obiekt ob1 klasy pochodna_1 z pod-obiektami klas bazowych. Stosowanie dziedziczenia wielokrotnego w praktyce jest dość skomplikowane. Wielu autorów zaleca umiar w tworzeniu klas z dziedziczeniem wielokrotnym, wielu też zaleca unikanie takich konstrukcji.

Rozdział 6

Funkcje zaprzyjaźnione

6.1. Wstęp **122**
6.2. Funkcja niezależna zaprzyjaźniona z klasą **123**
6.3. Funkcja składowa zaprzyjaźniona z inną klasą **134**
6.4. Klasy zaprzyjaźnione **141**

6.1. Wstęp

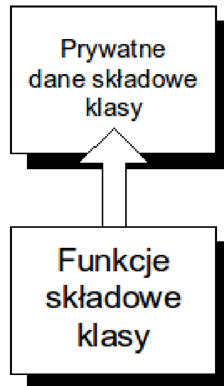
W języku C++ to klasa jest chroniona przed nieuprawnionym dostępem a nie obiekt. W takiej sytuacji funkcja składowa konkretnej klasy może używać wszystkich składowych prywatnych dowolnego obiektu tej samej klasy. W językach czysto obiektowych mamy do czynienia z mechanizmem odbioru komunikatu przez obiekt. Wywołując funkcję obsługującą dany obiekt, wysyłamy odpowiedni komunikat, obiekt pełni rolę odbiorcy komunikatu. Treścią komunikatu są wartości zmiennych, adresy, itp. przekazywane jako argumenty aktualne wywoływanej funkcji. Jeżeli mamy klasę o nazwie punkt, oraz funkcję składową klasy punkt, której prototyp ma postać int ustaw(int, int), to deklaracja obiektu a może mieć postać:

```
punkt a;
```

Dostęp do publicznej składowej funkcji klasy uzyskujemy przy użyciu operatora dostępu, np.

```
a.ustaw( 5, 5);
```

W ten sposób wywołujemy funkcję składową ustaw() klasy, do której należy obiekt a, to znaczy klasy punkt. Z formalnego punktu widzenia, to wywołanie możemy traktować jako zaadresowany do obiektu a komunikat o nazwie ustaw, którego treścią są wartości dwóch zmiennych (5 i 5).



Rysunek 6.1. Do danych prywatnych klasy dostęp mają jedynie funkcje składowe (metody) klasy

Z drugiej strony wiemy, że do składowych prywatnych utworzonego obiektu nie mają dostępu funkcje niezależne ani funkcje innej klasy. W wielu jednak przypadkach istnieje potrzeba dostępu do chronionych danych obiektu. Język C++ znany ze swojej elastyczności ma specjalny mechanizm pozwala-

jący na dostęp, w sposób kontrolowany, do formalnie niedostępnych danych. Możemy, wykorzystując odpowiednie konstrukcje języka C++ spowodować, że „zwykła" funkcja będzie mogła wykonywać operacje na obiekcie w taki sposób, jak czyni to funkcja składowa klasy. W tym przypadku, obiekt a może być traktowany jako jeden z argumentów formalnych funkcji. W takim przypadku wywołanie może mieć postać:

```
ustaw( a, 5, 5);
```

W tym wywołaniu, obiekt a jest traktowany tak samo jak pozostałe argumenty funkcji. Tym specjalnym mechanizmem w języku C++ jest mechanizm deklaracji zaprzyjaźnienia. Deklaracja zaprzyjaźnienia (ang. friend) pozwala zadeklarować w danej klasie funkcje, które będą miały dostęp do składowych prywatnych, ale nie są funkcjami składowymi klasy. Wyróżniamy następujące sytuacje:
— Funkcja niezależna zaprzyjaźniona z klasą X
— Funkcja składowa klasy Y zaprzyjaźniona z klasą X
— Wszystkie funkcje klasy Y są zaprzyjaźnione z klasą X (klasa zaprzyjaźniona)

Funkcje zaprzyjaźnione mają dostęp do wszystkich składowych klasy zadeklarowanych jako private lub protected. Z formalnego punktu widzenia, funkcje zaprzyjaźnione łamią zasadę hermetyzacji (ukrywania) danych, ale czasami ich użycie może być korzystne.

6.2. Funkcja niezależna zaprzyjaźniona z klasą

Funkcja zaprzyjaźniona z klasą jest zdefiniowana na zewnątrz jej zasięgu i ma dostęp do składowych prywatnych klasy. Aby zadeklarować funkcję jako zaprzyjaźnioną (mającą dostęp do danych prywatnych) z jakąś klasą, prototyp funkcji w jej definicji należy poprzedzić słowem kluczowym friend. Mimo, że prototyp funkcji umieszczony jest w definicji klasy, nie jest ona jej funkcją składową. Etykiety definiujące sposób dostępu do składowych, takie jak private, public i protected nie mają nic wspólnego z definicją funkcji zaprzyjaźnionych. Dobrym zwyczajem programistycznym jest umieszczanie prototypów wszystkich funkcji zaprzyjaźnionych z daną klasą na jej początku, ale nie jest to konieczne. Rozważmy program, pokazany na wydruku 6.1, w którym wykorzystamy funkcje zaprzyjaźnioną. W pokazanym przykładzie mamy prostą klasę punkt:

```
class punkt
{
    int x, y;
  public:
```

```
    punkt ( int xx = 0, int yy = 0 )
      {x = xx ;     y = yy ; }
 friend void pokaz (punkt );
};
```

Dwie dane składowe x i y są prywatne, dostęp do nich możliwy jest jedynie poprzez funkcje składowe klasy punkt. Klasa posiada konstruktor. Jeżeli chcemy, aby funkcja zewnętrzna miała dostęp do danych składowych klasy punkt, musimy jawnie zadeklarować taką funkcję jako zaprzyjaźnioną. W naszym przykładzie tworzymy funkcję zaprzyjaźnioną o nazwie pokaz():

```
friend void pokaz (punkt );
```

Listing 6.1. Dostęp do składowych prywatnych; funkcje zaprzyjaźnione

```
 1 #include <iostream.h>
   #include <conio.h>
 3 class punkt
   {private:
 5     int x;
       int y;
 7  public:
       punkt ( int xx = 0, int yy = 0 )
 9       {x = xx ;     y = yy ; }
    friend void pokaz (punkt );
11 };
   void pokaz (punkt p1)
13  { cout << "Pozycja X = " << p1.x << endl;
      cout << "Pozycja Y = " << p1.y << endl;
15  }
   int main()
17 {  punkt a1( 2, 4 ) ;
      pokaz ( a1 ) ;
19    getche();
      return 0;
21 }
```

Dzięki specyfikacji friend funkcja pokaz() staje się funkcją zaprzyjaźnioną, ma dostęp do prywatnych danych klasy punkt. Funkcja pokaz() wyświetla aktualne wartości danych prywatnych x i y. Funkcja pokaz() jest samodzielną funkcją w stylu języka C – nie jest ona funkcją składową klasy punkt:

```
void pokaz (punkt p1)
 { cout << "Pozycja X = " << p1.x << endl;
   cout << "Pozycja Y = " << p1.y << endl;
 }
```

W programie testującym:

```
int main()
{  punkt a1( 2, 4 ) ;
   pokaz ( a1 ) ;
   return 0;
}
```

tworzymy obiekt a1 klasy punkt i inicjujemy go wartościami 2 i 4. Aby wyświetlić na ekranie monitora wartości x i y musimy wywołać funkcje pokaz(). Ponieważ jest to funkcja zaprzyjaźniona, możemy wywoływać tą funkcje tak, jak zwykłą funkcję w języku C:

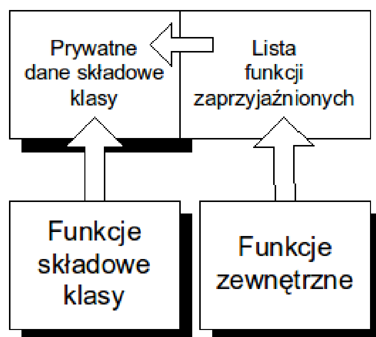
```
pokaz ( a1 ) ;
```

W naszym przykładzie funkcja pokaz() pobiera jako argument a1. Pamiętamy, że gdyby funkcja pokaz() była funkcją składową klasy punkt, to wywołanie jej miałoby postać:

```
a1.pokaz();
```

Pomimo, że funkcja pokaz() nie jest składową klasy punkt, ze względu na deklarację zaprzyjaźnienia ma ona pełny dostęp do prywatnych składowych tej klasy. Funkcja zaprzyjaźniona pokaz() jest wywoływana bez użycia operatora kropka. Nie jest funkcją składową, nie może być poprzedzana nazwą obiektu.

W omawianym przykładzie, zamiany funkcji składowej klasy na funkcję zaprzyjaźnioną z klasą niczego nie wnosi, a faktycznie nawet osłabia podstawową zasadę programowania obiektowego, jaką jest zasada ukrywania danych, to zdarzają się sytuacje gdzie stosowanie funkcji zaprzyjaźnionych jest korzystne a nawet konieczne.



Rysunek 6.2. Funkcja zaprzyjaźniona z klasą definiowana jest na zewnątrz jej zasięgu, a mimo tego ma dostęp do jej składowych prywatnych

Na pokazanym diagramie widzimy dwie metody dostępu do prywatnych danych klasy – przy pomocy funkcji składowych klasy lub przy pomocy funkcji zewnętrznych, ale zaprzyjaźnionych z klasą. W kolejnym przykładzie opracujemy klasę punkt, która ma jedną funkcję składową inicjującą dane klasy (ustalimy współrzędne punktu na płaszczyźnie) i jedną funkcję zaprzyjaźnioną (obliczy ona odległość punktu od początku układu współrzędnych). W omówionym przykładzie (program z wydruku 6.2), tworzymy obiekt p1 a następnie przy pomocy funkcji składowej ustaw() przypisujemy wartości prywatnym zmiennym składowym:

```
p1.ustaw(1.0, 1.0);
```

Funkcja zaprzyjaźniona promien() nie może odwołać się do zmiennych składowych bezpośrednio, wykorzystuje do tego celu obiekt n klasy punkt. W wywołaniu:

```
cout << "Odleglosc = " << promien(p1) << endl;
```

funkcji zaprzyjaźnionej promien() przekazywany jest argument aktualny p1.

Listing 6.2. Dostęp do składowych prywatnych; funkcje zaprzyjaźnione

```
1 #include <iostream>
  #include <math>
3 #include <conio>
  using namespace std;
5 class punkt
  { float x, y;
7   public:
     void ustaw(float a, float b);
9    friend float promien(punkt n);
  };
11 void punkt :: ustaw(float a, float b)
    { x = a;   y = b;
13  }
  float promien ( punkt n)
15  { return sqrt(n.x * n.x + n.y * n.y);
    }
17 int main()
  { punkt p1;
19  p1.ustaw(1.0, 1.0);
    cout << "Odleglosc = " << promien(p1) << endl;
21    getche();
       return 0;
23 }
```

Jeszcze bardziej uproszczona wersje poprzedniego przykładu pokazano na kolejnym przykładzie (wydruk 6.3).

Listing 6.3. Dostęp do składowych prywatnych; funkcje zaprzyjaźnione

```
1 #include <iostream>
  #include <math>
3 #include <conio>
  using namespace std;
5 class punkt
  { float x, y;
7   friend float promien(punkt n, float a, float b);
  };
9     float promien ( punkt n, float a, float b)
    { n.x = a;
11      n.y = b;
      return sqrt(n.x * n.x + n.y * n.y);
13  }
      int main()
15 { punkt p1;
    cout << "odleglosc = " << promien(p1, 1.0, 1.0) << endl;
17    getche();
      return 0;
19 }
```

Zmodyfikowana klasa punkt ma postać:

```
class punkt
{ float x, y;
  friend float promien(punkt n, float a, float b);
};
```

Składowe x i y są danymi prywatnymi klasy (gdy nie podamy specyfikatora private, składowe domyślnie są zadeklarowane jako prywatne). Ponieważ klasa nie posiada żadnych funkcji składowych, nie ma potrzeby stosowania specyfikatora public (funkcje zaprzyjaźnione nie podlegają specyfikacji dostępu, możemy je umieszczać w dowolnej sekcji zarówno prywatnej jak i publicznej). Funkcja zaprzyjaźniona:

```
float promien ( punkt n, float a, float b)
  { n.x = a;
    n.y = b;
    return sqrt(n.x * n.x + n.y * n.y);
  }
```

otrzymuje potrzebne argumenty i ustawia zmienne x i y, a następnie oblicza odległość punktu od początku układu współrzędnych.

W języku C++ argumenty w wywoływanej funkcji przekazywane są przez wartość lub przez referencję. Przypomnimy krótko zasady posługiwania się referencją. Zasadniczo referencja jest to niejawny wskaźnik. Kiedy argument przekazywany jest przez wartość, tworzona jest kopia wartości

argumentu i ta kopia przekazywana jest do wywoływanej funkcji. Możemy dokonywać dowolnych zmian na kopii, oryginalna wartość jest bezpieczna – nie ulega zmianom. Przekazywanie argumentów przez wartość jest bezpieczne, ale bardzo wolne. Przekazywanie parametrów funkcji przez referencje jest korzystne ze względu na wydajność – jest po prostu procesem bardzo szybkim. Za każdym razem, gdy przekazywany jest obiekt do funkcji przez wartość, tworzona jest kopia tego obiektu. Za każdym razem, gdy zwracany jest obiekt z funkcji tworzona jest kolejna kopia. Obiekty kopiowane są na stos. Wymaga to sporej ilości czasu i pamięci. W przypadku zdefiniowanych przez programistę dużych obiektów, ten koszt staje się bardzo duży. Rozmiar obiektu umieszczonego na stosie jest sumą rozmiarów wszystkich jego zmiennych składowych. Stosowanie referencji jest korzystne, ponieważ eliminuje koszty związane z kopiowaniem dużych ilości danych.

Parametr referencji jest aliasem (synonimem) do odpowiadającego mu argumentu. Referencja jest specjalnym, niejawnym wskaźnikiem, który działa jak alternatywna nazwa dla zmiennej. Zmienną o podanej po nazwie typu z przyrostkiem &, np.

```
T& nazwa
```

gdzie T jest nazwą typu, nazywamy referencją do typu T. Na przykład, deklaracja

```
int &liczba
```

umieszczona w nagłówku funkcji oznacza „liczba jest referencją do int". W krótkim przykładzie przypomnimy przekazywanie argumentów przez referencję.

Listing 6.4. Przekazywanie parametrów przez referencję

```
1 #include <iostream>
  #include <conio>
3  using namespace std;
   void Kwadrat( int & );
5 int main()
  {  int x = 2;
7     cout << "zmienna x zmodyfikowana:" <<endl;
      cout << "x = " << x << endl;
9     Kwadrat(x);
      cout << "x = " << x << endl;
11    int a = x;
      cout << "zmienna x nie zmodyfikowana:" <<endl;
13    cout << "x = " << x << endl;
      Kwadrat(a);
15    cout << "x = " << x << endl;
      getche();
```

```
17    return 0;
  }
19  void Kwadrat( int &xref)
    { xref *= xref; }              //argument zmodyfikowany
```

Po uruchomieniu programu otrzymujemy wydruk:

```
zmienna x zmodyfikowana:
x = 2
x = 4
zmienna x nie zmodyfikowana:
x = 4
x = 4
```

W programie parametr funkcji Kwadrat() jest przekazywany przez referencję, co widać wyraźnie w prototypie:

```
void Kwadrat( int & );
```

Programista analizując wywołanie funkcji:

```
Kwadrat(x); lub          Kwadrat(a);
```

nie jest wstanie zorientować się, czy parametry przekazywane są przez wartość czy przez referencję (taką pewność ma tylko wtedy, gdy zanalizuje prototyp funkcji). Może to prowadzić do nieoczekiwanych skutków ubocznych, co demonstruje nasz program. Po pierwszym wywołaniu funkcji Kwadrat() zmienna x zmieniała wartość. Aby uniknąć takich skutków wielu programistów przesyła niemodyfikowalne argumenty stosując referencję do stałych.

Ponieważ zagadnienia wydajności są istotne, w kolejnym programie demonstrującym wykorzystanie funkcji zaprzyjaźnionych, zastosujemy przekazywanie argumentów przez referencję.

Mamy następującą klasę:

```
class Demo_1
{   friend void ustawA ( Demo_1 &, int  );
          // funkcja zaprzyjazniona
  public:
     Demo_1( ) { a = 0; }
     void pokaz( ) const { cout << a << endl ; }
  private:
     int a;
};
```

W tej klasie jest zadeklarowana dana składowa a, która jest prywatna, czyli

jest niedostępna dla funkcji nie będących składowymi klasy. Aby funkcja zewnętrzna ustawA() miała dostęp do tej danej, zadeklarowano ją jako friend:

```
friend void ustawA ( Demo_1 &, int ) ;
        // funkcja zaprzyjazniona
```

Definicja funkcji ustawA() ma postać:

```
void ustawA ( Demo_1 &x, int ile)
  {   x.a = ile;
  }
```

Funkcja ustawA() ma możliwość modyfikacji danej a ponieważ jest zaprzyjaźniona z klasą Demo_1. Cały program ma postać przedstawiona na wydruku 6.1.

Listing 6.5. Dostęp do składowych prywatnyc; funkcje zaprzyjaźnione

```
1 #include <iostream.h>
  #include <conio.h>
3 class Demo_1
  {   friend void ustawA ( Demo_1 &, int );
5           // funkcja zaprzyjazniona
    public:
7       Demo_1() { a = 0; }
        void pokaz ()  const { cout << a << endl; }
9   private:
        int a;
11 };
  void ustawA ( Demo_1 &x, int ile )
13  {  x.a = ile;
    }
15 int main()
  { Demo_1 zm;
17  cout << "zm.a po utworzeniu obiektu: ";
    zm.pokaz();
19  cout << "zm.a po wywolaniu funkcji friend ustawA: ";
    ustawA ( zm, 10);
21  zm.pokaz();
        getch();
23       return 0;
  }
```

Po uruchomieniu otrzymujemy następujący wynik:

```
zm.a po utworzeniu obiektu : 0
zm.a po wywolaniu funkcji friend ustawA: 10
```

W trakcie wykonywania programu zostaje utworzony obiekt zm i danej a jest

przypisana wartość 0. Następnie wywołana zostaje funkcja zaprzyjaźniona ustawA() i danej a zostaje przypisana wartość 10. Funkcja składowa pokaz():

```
void pokaz ()  const { cout << a << endl; }
```

została zadeklarowana jako const, co oznacza, że nie może ona modyfikować danych obiektów. Funkcja deklarowana jest jako const zarówno w prototypie jak i w definicji przez wstawienie słowa kluczowego po liście jej parametrów i, w przypadku definicji przed nawiasem klamrowym rozpoczynającym ciało funkcji. Programista może określić, które obiekty wymagają modyfikacji, a które pod żadnym pozorem nie mogą być zmieniane. Gdy obiekt nie może być zmieniony, programista może posłużyć się słowem kluczowym const. Wszystkie próby późniejszej modyfikacji takiego obiektu znajdowane są już w trakcie kompilacji programu. W kolejnym przykładzie, pokażemy jak napisać niezależną funkcję pokaz(), zaprzyjaźnioną z klasą punkt, wypisującą na ekranie współrzędne punktu. Deklaracja wyjściowa klasy punkt ma postać:

```
class punkt
 {
   int x, y;
   public:
     punkt (int xx = 0, int yy = 0)
          { x = xx ;    y = yy ;
          }
 };
```

Aby mieć dostęp do prywatnych danych klasy punkt, co jest nam potrzebne, aby wyświetlić współrzędne punktu x i y, musimy dysponować zewnętrzną funkcją pokaz(), zaprzyjaźnioną z klasą punkt. Argumenty do tej funkcji muszą być przekazane jawnie. Wobec tego prototyp tej funkcji może mieć postać:

```
void pokaz ( punkt );
```

gdy chcemy przekazywać argumenty przez wartość, lub:

```
void pokaz  (punkt & );
```

gdy chcemy przekazać argumenty przez referencję. Możemy usprawnić funkcję pokaz() wiedząc, że ta funkcja nie zmienia współrzędnych i zastosować modyfikator const:

```
void pokaz ( const punkt & ) ;
```

Modyfikacja pierwotnej klasy punkt może mieć postać pokazaną na wydruku 6.6.

Listing 6.6. Dostęp do składowych prywatnych; funkcje zaprzyjaźnione

```
1 // deklaracja klasy punkt,  plik punkt1.h
   #ifndef _PUNKT1_H
3  #define _PUNKT1_H
  class punkt
5 {  int x, y;
     public:
7      friend void pokaz ( const punkt & ) ;
       punkt (int xx = 0, int yy = 0)
9          { x = xx ;    y = yy;
           }
11   };
   #endif
```

Funkcja służąca do wyświetlenia współrzędnych ma postać:

```
friend void pokaz( const punkt &);
```

Funkcja pokaz() jest zaprzyjaźniona z klasą punkt i ma dostęp do prywatnych danych x i y. Jeżeli powstanie obiekt klasy punkt o nazwie p to możemy uzyskać dostęp do jego składowych klasycznie:

```
p.x  i  p.y
```

Definicję funkcji pokaz() umieszczamy w oddzielnym pliku (wydruk 6.7).

Listing 6.7. Dostęp do składowych prywatnych; funkcje zaprzyjaźnione

```
1 //definicja klasy punkt , plik punkt1.cpp
  #include "punkt1.h"
3 #include <iostream.h>

5 void pokaz (const punkt &p)
    {  cout << " wspolrzedne punktu: " << p.x << " " << p.y
7       << "\n";
    }
```

Funkcja testująca pokazana jest na wydruku 6.8:

Listing 6.8. Dostęp do składowych prywatnych; funkcje zaprzyjaźnione

```
  //program testujacy funkcje zaprzyjaznione
2 #include <iostream.h>
  #include <conio.h>
4 #include <punkt1.h>
```

```
     int main()
  6  {   cout << "zmienna automatyczna : " << endl;
       punkt p1(10,20);
  8    pokaz (p1);
       cout << "zmienna dynamiczna : " << endl;
 10    punkt *wsk;
       wsk = new punkt (20, 40);
 12    pokaz (*wsk);
            getch();
 14         return 0;
     }
```

Po uruchomieniu otrzymujemy następujący wydruk;

```
zmienna automatyczna :
wspolrzedne punktu: 10 20

zmienna dynamiczna :
wspolrzedne punktu: 20 40
```

W pokazanym programie przypomniano także tworzenie obiektów dynamicznych. W języku C++ dla każdego programu przydzielany jest pewien obszar pamięci dla alokacji obiektów tworzonych dynamicznie. Obszar ten jest zorganizowany w postaci kopca (ang. heap). Na kopcu alokowane są obiekty dynamiczne. Obiekty dynamiczne tworzone są przy pomocy operatora new. Operator new alokuje (przydziela) pamięć na kopcu. Gdy obiekt nie jest już potrzebny należy go zniszczyć przy pomocy operatora delete. Aby można było zastosować operator new należy najpierw zadeklarować zmienną wskaźnikową, dla przykładu:

```
int *wsk;
```

Tworzenie zmiennej dynamicznej ma postać:

```
wsk = new typ;
```

lub

```
wsk = new typ (wartosc);
```

gdzie typ oznacza typ zmiennej (np. int, float, itp.) Możemy deklarację zmiennej wskaźnikowej połączyć z tworzeniem zmiennej dynamicznej:

```
typ * wsk = newt typ;
```

Tak złożona instrukcja może mieć przykładową postać:

```
int *wsk = new int;
```

W pokazanym programie utworzono zmienną dynamiczną w następujący sposób:

```
cout << "zmienna dynamiczna : " << endl;
punkt *wsk;
wsk = new punkt (20, 40);
```

6.3. Funkcja składowa zaprzyjaźniona z inną klasą

Funkcja zaprzyjaźniona danej klasy może być też funkcją składową zupełnie innej klasy. Taka funkcja ma dostęp do prywatnych danych swojej klasy i do danych klasy, z którą się przyjaźni. Kolejny przykład ilustruje to zagadnienie. Mamy dwie klasy - klasę prostokat i klasę punkt. Klasa prostokat definiuje prostokąt przy pomocy współrzędnych dwóch punktów – lewego dolnego rogu prostokąta i prawego górnego rogu prostokąta. Klasa punkt opisuje punkt na podstawie jego współrzędnych kartezjańskich. Mając dany punkt i dany prostokąt należy określić czy punkt znajduje się wewnątrz prostokąta czy też leży poza nim.

W programie deklarujemy dwie klasy – punkt i prostokat z konstruktorami. Funkcja miejsce() jest funkcją składową klasy prostakat i jest zaprzyjaźniona z klasą punkt. W programie testującym wywołujemy funkcję miejsce(), aby ustalić położenie punktu względem prostokąta.

Listing 6.9. Dostęp do składowych prywatnych; funkcje zaprzyjaźnione

```
1 #include <iostream>
  #include <conio>
3       using namespace std;
  class punkt;          //deklaracja zapowiadajaca
5
  class prostokat
7 { int xp, yp, xk, yk;
   public:
9   prostokat( int xpo, int ypo, int xko, int yko);
    void miejsce ( punkt &p);
11 };
  class punkt
13 { int x1, y1;
   public:
15  punkt (int ax, int ay);
    friend void prostokat::miejsce ( punkt &p);
17 };
```

```
19 prostokat:: prostokat( int xpo, int ypo, int xko, int yko)
   { xp = xpo;  yp = ypo;
21   xk = xko;  yk = yko;
   }
23
   punkt ::  punkt (int ax, int ay)
25 {  x1 = ax;     y1 = ay;
   }
27
   void prostokat :: miejsce( punkt &pz)
29 {  if (    (pz.x1 >= xp) && (pz.x1 <= xk)
            &&
31         (pz.y1 >= yp) && (pz.y1 <= yk)
        )
33           cout << "punkt lezy w polu" << endl;
     else
35           cout << "punkt lezy poza polem" << endl;
   }
37
   int main()
39 {prostokat pr( 0, 0, 100, 100);
     punkt pu ( 10, 10);
41   pr.miejsce(pu);
         getche();
43       return 0;
   }
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
punkt lezy w polu
```

Jak widzimy deklaracja klasy prostokat ma postać:

```
class prostokat
{ int xp, yp, xk, yk;
 public:
  prostokat( int xpo, int ypo, int xko, int yko);
  void miejsce ( punkt &p);
};
```

Funkcja miejsce() jest zwykłą funkcją składową klasy prostokat. W deklaracji funkcji składowej miejsce() argumentem jest obiekt klasy punkt, dlatego konieczna jest deklaracja zapowiadająca klasę punkt. Deklaracja klasy punkt ma postać:

```
class punkt
{ int x1, y1;
 public:
  punkt (int ax, int ay);
```

```
    friend void prostokat::miejsce ( punkt &p);
};
```

Deklaracja funkcji zaprzyjaźnionej ma postać:

```
friend void prostokat::miejsce ( punkt &p);
```

W deklaracji zaprzyjaźnionej funkcji miejsce() musimy podać nazwę klasy, w której ta funkcja jest funkcją składową, w naszym przypadku jest to klasa prostokat.

Bardzo często argumentuje się, że nie należy nadużywać funkcji zaprzyjaźnionych, ponieważ istotą programowania obiektowego jest ukrywanie informacji. Funkcja zaprzyjaźniona nie jest składową klasy a mimo tego ma dostęp do danych prywatnych. W wielu jednak przypadkach zastosowanie funkcji zaprzyjaźnionych znacznie usprawnia algorytm. Klasycznym przykładem jest program do wykonania mnożenie wektora przez macierz. Załóżmy, że są zdefiniowane dwie klasy: wektor i macierz. Każda z nich ukrywa swoje dane i dostarcza odpowiedni zbiór operacji do działania na obiektach swojego typu. Należy zdefiniować funkcję mnożącą macierz przez wektor. Ustalmy konkretne warunki. Niech wektor ma cztery elementy indeksowane 0,1,2,3. Wektor zapamiętywany będzie w postaci tablicy jednowymiarowej. Macierz ma rozmiar 4x4 i będzie zapamiętywana w postaci tablicy dwuwymiarowej. Funkcja obliczająca iloczyn musi korzystać z danych pochodzących z dwóch klas, jest więc oczywiste, że musi być z nimi zaprzyjaźniona. W tym przypadku konieczne jest także użycie deklaracji referencyjnej(zwaną także deklaracją zapowiadającą, albo referencją zapowiadającą), w przypadku deklaracji klasy wekt musi wystąpić deklaracja klasy macierz i podobnie w deklaracji klasy macierz musi wystąpić deklaracja klasy wekt.

Jest to konieczne, gdyż w klasie wekt w deklaracji iloczyn() istnieje odwołanie do niezadeklarowanej jeszcze klasy macierz. Deklaracje poszczególnych klas i ich definicje zapisujemy w oddzielnych plikach. Musimy na początku opracować klasę wekt do obsługi wektorów. Deklaracja klasy wekt może mieć postać:

Listing 6.10. Iloczyn macierzy przez wektor; funkcje zaprzyjaźnione

```
1 //plik wektor1.h, deklaracja klasy wekt

3 #ifndef _WEKTOR1_H
  #define _WEKTOR1_H
5
   class macierz ;      // deklaracja zapowiadajaca
7
  class wekt
9 {
```

```
        double v[4];      //wektor o 4 skladowych
11    public:
        wekt(double v1=0, double v2=0,
13        double v3=0, double v4=0)
        {v[0] = v1; v[1] = v2; v[2] = v3; v[3] = v4;}
15    friend wekt iloczyn (const macierz &, const wekt &) ;
        void pokaz();
17  };
  #endif
```

Podobnie w deklaracji klasy macierz musimy zastosować deklaracje zapowiadającą klasy wekt. Deklaracja klasy macierz do obsługi macierzy może mieć postać:

Listing 6.11. Iloczyn macierzy przez wektor; funkcje zaprzyjaźnione

```
  //plik macierz.h, deklaracja klasy macierz
2
  #ifndef _MACIERZ_H
4 #define _MACIERZ_H

6 class wekt;

8 class macierz
  {
10   double mac[4][4] ;      //macierz 4x4
    public:
12     macierz();  //konstruktor z inicjacja na 0
       macierz(double t[4][4]); //konstruktor, dane z tablicy
14     friend wekt iloczyn (const macierz &, const wekt &);
  };
16
  #endif
```

W klasie wekt mamy deklarację funkcji składowej pokaz(), która służy do wyświetlania składowych wektora. Definicja funkcji pokaz() może mieć postać:

Listing 6.12. Iloczyn macierzy przez wektor; funkcje zaprzyjaźnione

```
1 //plik pokaz.cpp, definicja skladowej pokaz()
  #include <iostream.h>
3 #include "wektor1.h"

5 void wekt::pokaz()
  { int i;
7   for (i=0; i < 4; i++)      cout << v[i] << " ";
    cout << "\n";
9 }
```

W klasie macierz jest zadeklarowany konstruktor macierz(). Jego definicja może mieć postać:

Listing 6.13. Iloczyn macierzy przez wektor; funkcje zaprzyjaźnione

```
//plik konmac.cpp, definicja konstruktora  klasy macierz
#include <iostream.h>
#include "macierz.h"

macierz::macierz (double t [4][4])
{  int i;
   int j;
   for (i=0; i<4; i++)
      for (j=0; j<4; j++)
         mac[i][j] = t[i][j];
}
```

Należy opracować funkcje zaprzyjaźnioną iloczyn(). Definicja tej funkcji (wykonuje ona mnożenie macierzy przez wektor) może mieć postać:

Listing 6.14. Iloczyn macierzy przez wektor; funkcje zaprzyjaźnione

```
//plik iloczyn.cpp
//definicja funkcji iloczyn
#include "wektor1.h"
#include "macierz.h"

wekt iloczyn (const macierz & m, const wekt & x)
{  int i,j;
   double suma;
   wekt wynik;
   for (i=0; i<4; i++)
       {  for (j=0, suma=0; j<4; j++)
               suma += m.mac[i][j] * x.v[j];
          wynik.v[i] = suma;
        };
    return wynik;
}
```

Możemy zacząć testować mnożenie macierzy przez wektor. Funkcje iloczyn() możemy wykorzystać do realizacji przekształceń w przestrzeni 3D. W grafice komputerowej wykorzystuje się tzw. współrzędne jednorodne. W tych współrzędnych punkt (x,y,z) reprezentowany jest jako punkt w przestrzeni 4-wymiarowej (x,y,z,1). Poszczególne przekształcenia takie jak przesuniecie, skalowanie czy obroty reprezentowane są macierzami 4x4. Aby otrzymać położenie nowego punktu P' należy punkt początkowy P pomnożyć przez macierz przekształcenia M:

$$P' = M \bullet P \tag{6.1}$$

Przesunięcie w przestrzeni 3D ma postać:

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.2}$$

Operacja skalowania przedstawiana jest następująco:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.3}$$

Mamy trzy macierze reprezentujące obroty wokół osi x,y i z. Obrót wokół osi z przedstawia się następująco:

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.4}$$

Obrót wokół osi x ma postać:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.5}$$

Obrót wokół osi y ma postać:

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.6}$$

Te transformacje można łatwo zweryfikować: wynikiem obrotu o 90^o jednostkowego wektora osi x $\begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}^T$ powinien być jednostkowy wektor $\begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix}^T$ osi y.

Ogólnie mamy do czynienia z mnożeniem macierzy przez wektor:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.7}$$

W programie testującym wyliczymy współrzędne punktu po przekształceniach.

Listing 6.15. iloczyn macierzy przez wektor; funkcje zaprzyjaźnione

```
#include <iostream.h>
#include <conio.h>
#include "wektor1.h"
#include "macierz.h"
int main()
{ wekt w (1, 0, 0, 1);
  wekt wynik;
  wynik = w;
  cout << "punkt poczatkowy: ";
  wynik.pokaz();
  double trans[4][4] = {1, 0, 0, 2,
                        0, 1, 0, 3,
                        0, 0, 1, 1,
                        0, 0, 0, 1  };
  macierz a = trans;
  wynik = iloczyn (a, w);
  cout << "   po translacji: ";
  wynik.pokaz ();
  double skal[4][4] = {2, 0, 0, 0,
                       0, 2, 0, 0,
                       0, 0, 2, 0,
                       0, 0, 0, 1  };
  macierz b = skal;
  wynik = iloczyn (b, w);
  cout << "   po skalowaniu: ";
  wynik.pokaz ();
  double obrZ[4][4] = {0,-1, 0, 0,
                       1, 0, 0, 0,
                       0, 0, 1, 0,
                       0, 0, 0, 1      };
  macierz c = obrZ;
  wynik = iloczyn (c, w);
  cout << "po obrocie o PI/2 : ";
  wynik.pokaz ();
        getch();
        return 0;
}
```

Po uruchomieniu programu testującego otrzymamy następujący wynik:

```
punkt poczatkowy : 1 0 0 1
po translacji : 3 3 1 1
po skalowaniu : 2 0 0 1
po obrocie o PI/2 : 0 1 0 1
```

6.4. Klasy zaprzyjaźnione

Każda klasa może mieć wiele funkcji zaprzyjaźnionych, można nawet uczynić wszystkie funkcje składowe jednej klasy zaprzyjaźnione z inną klasą. Możemy przy pomocy słowa kluczowego friend uczynić daną klasę zaprzyjaźnioną z inną klasą. Jeżeli zadeklarujemy, że cała klasa A jest uznawana za przyjaciela klasy B to ten fakt zapisujemy w następujący sposób:

```
class B
{
friend class A ;
// .....................
};
```

Zapis:

```
friend class A ;
```

oznacza, że wszystkie funkcje składowe klasy A maja dostęp do danych prywatnych i chronionych klasy B. Jeżeli klasa A ma być zaprzyjaźniona z klasą B, to deklaracja klasy A musi poprzedzać deklarację klasy B. Zagadnienie to zilustrujemy przykładem.

Listing 6.16. Klasy zaprzyjaźnione

```
1 #include <iostream>
  #include <conio>
3
  using namespace std;
5
  class Dane
7 {
    int x1, x2;
9   public:
     Dane (int a, int b) { x1 = a;  x2 = b; }
11   friend class Test;
  };
13
  class Test
15 {
   public:
17   int min(Dane a) { return a.x1 < a.x2 ? a.x1 : a.x2; }
     int iloczyn (Dane a) { return a.x1 * a.x2 ; }
19 };

21 int main()
  {
23   Dane liczby(10, 20);
     Test t;
```

```
25     cout << "mniejsza to: " << t.min(liczby) << endl;
       cout << "   iloczyn = " << t.iloczyn(liczby) << endl;
27
       getche();
29     return 0;
    }
```

Po uruchomieniu programu mamy następujący wydruk:

```
mniejsza to: 10
iloczyn = 200
```

W pokazanym programie klasa Test ma dwie funkcje składowe (jedna wyznacza mniejszą z dwóch liczb, druga wylicza iloczyn dwóch liczb) i ma dostęp do danych prywatnych klasy Dane. Należy pamiętać, że klasy zaprzyjaźnione stosuje się stosunkowo rzadko w praktyce.

Rozdział 7

Przeciążanie operatorów

7.1. Wstęp **144**
7.2. Definicje **144**
7.3. Przeciążony operator dodawania (+) **147**
7.4. Przeciążony operator mnożenia (*) **150**
7.5. Funkcja operatorowa w postaci niezależnej funkcji . . . **154**
7.6. Przeciążanie operatorów równości i nierówności **161**
7.7. Przeciążanie operatora przypisania (=) **165**
7.8. Przeciążanie operatora wstawiania do strumienia («) . **173**

7.1. Wstęp

Programując w języku C++ programista może korzystać zarówno z typów wbudowanych jak i typów stworzonych przez siebie. W języku C++ został zdefiniowany zestaw operatorów, wszystkie one są używane w połączeniu z typami wbudowanymi. W języku C++ nie można tworzyć nowych operatorów. Często jednak zachodzi potrzeba zmienienia sposobu działania konkretnego operatora. Język C++ pozwala na przeciążanie istniejących operatorów, to znaczy możemy nadawać im nowe znaczenie dla operacji wykonywanych częściowo lub w całości na obiektach typu klasa. Być może przeciążanie operatorów na pierwszy rzut oka jest dość egzotyczną techniką, ale wielu programistów nie zdaje sobie sprawy z tego, że używa przeciążonych operatorów. Przykładem przeciążonego operatora jest operator dodawania (+). Operator dodawania działa inaczej na danych typu int i inaczej na danych typu double. Takie działanie jest możliwe, ponieważ operator dodawania został przeciążony w samym języku C++.

7.2. Definicje

Aby przeciążyć operator definiuje się funkcję (z nagłówkiem i ciałem) w zwykłej postaci z wyjątkiem tego, że nazwą funkcji jest słowo kluczowe operator poprzedzające symbol przeciążanego operatora. Na przykład nazwa funkcji operator+ mogłaby być użyta do przeciążenie operatora dodawania (+). Przeciążanie operatora realizowane jest:

- albo w postaci niezależnej funkcji (zazwyczaj zaprzyjaźnionej z jedną lub kilkoma klasami)
- albo w postaci funkcji składowej

W pierwszym przypadku, jeżeli oper jest operatorem dwuargumentowym, to zapis:

```
x oper y
```

jest równoznaczny wywołaniu:

```
operator oper (x, y)
```

w drugim przypadku ten sam zapis odpowiada wywołaniu:

```
x.operator oper (y)
```

Tabela 7.1. Tabela operatorów, które mogą być przeciążane

Operator	Opis
()	Wywołanie funkcji (Function call)
[ ]	Element tablicy (Array element)
− >	Operator dostepu (Structure memeber pointer reference)
New	Dynamicznie alokowana pamięć (Dynamically allocate memory)
Delete	Dynamicznie usuwana pamięć (Dynamically deallocated memory)
++	Inkrementacja (Increment)
–	Dekrementacja (Decrement)
-	Minus (Unary minus)
!	Logiczna negacja (Logical negation)
∼	Dopełnienie logiczne (One's complement)
*	Dereferencja (Indirection)
*	Mnożenie (Multiplication)
/	Dzielenie (Division)
%	Modulo (Modulus (remainder))
+	Dodawanie (Addition)
-	Odejmowanie (Subtraction)
<<	Przesunięcie (Left shift)
>>	Przesunięcie (Right shift)
<	Mniej niż (Less than)
<=	Mniej niz lub rowne (Less than or equel to)
>	Większe niż (Greater than)
>=	Większe niż lub równe (Greater than or equel to)
==	Równe (Equal to)
!=	Różne (Not equal to)
&&	Logiczne AND (Logical AND)
||	Logiczne OR (Logical OR)
&	Bitowe AND (Bitwise AND)
ˆ	Bitowe XOR (Bitwise exlusive OR)
|	Bitowe OR (Bitwise inclusive OR)
=	Przypisanie (assignment)
+= -= *=	Przypisanie (assignment)
/= %= &=	Przypisanie (assignment)
ˆ= | =	Przypisanie (assignment)
<<=>>=	Przypisanie (assignment)
,	Przecinek (Comma)

Przeciążony operator musi mieć jako jeden z operandów obiekt. W języku C++ można przeciążać prawie wszystkie operatory, a lista operatorów, które

mogą być przeciążane jest pokazana w tabeli. Każdy z tych operatorów może być przeciążony w dowolny sposób (np. operator dodawania po przeciążeniu wcale nie musi wykonywać operacji dodawania). Oczywiście są pewne ścisłe reguły. Operator binarny musi pozostać binarnym, a operator unarny musi pozostać unarnym.

Działanie symboli pokazanych w tabeli może być przedefiniowane tak, aby jak najlepiej obsługiwać swoją klasę.

Należy pamiętać o następujących ograniczeniach:

— Symbole, które nie są umieszczone w tabeli nie mogą być przeciążane. Odnosi się to takich symboli jak:
  — operator kropka(.)
  — operator wyłuskania wskaźnika do składowej (.*)
  — operator zasięgu (::)
  — operator warunkowy (?:)
  — operator rozmiaru (sizeof)
  — symbole # i ##
— Nie można kreować nowych symboli operatorów. Np. ^^ nie jest operatorem w C++, nie może być kreowany jako operator klasy. W języku C++ nie istnieje operator potęgowania, więc próba tworzenia własnego operatora potęgowania nie wydaje się konieczna, w języku istnieje odpowiednia funkcja, którą należy wywołać w celu wykonania potęgowania.
— Ani kolejność (ang. precedence) ani łączność (ang. associativity) operatorów C++ nie może być modyfikowana. Nie można np. operatorowi dodawania nadać wyższy priorytet niż ma operator dzielenia.
— Nie można redefiniować operatorów dla typów wbudowanych
— Operator unarny nie może być zmieniany na binarny, a binarny nie może być zmieniany na unarny.
— Operator musi być albo członkiem klasy albo być tak definiowany, aby przynajmniej jeden członek klasy był jego operandem.

Zazwyczaj konstruując jakąś klasę należy określić, czy potrzebne będą specjalne operatory. Na przykład, aby porównać dwa trójwymiarowe wektory, należy kolejno sprawdzić czy odpowiednie składowe (x, y oraz z) dwóch wektorów są równe. Zdefiniowana przez użytkownika operacja (operator) jest projektowana jako funkcja, która redefiniuje wbudowany w C++ symbol operatora i może być wykorzystana przez klasę. Funkcja, która definiuje operacje na obiektach klasy i wykorzystuje wbudowane w C++ symbole operatorów nosi nazwę funkcji operatorowej (operator function). Funkcje operatorowe są deklarowane i implementowane w taki sam sposób, jak wszystkie funkcje składowe, z jednym wyjątkiem – wszystkie funkcje operatorowe mają nazwę postaci:

```
operator op
```

gdzie op jest jednym z symboli pokazanych w tabeli przeciążanych operatorów. Na przykład, nazwa funkcji operator + jest nazwą funkcji dodawania, a nazwa funkcji operator == jest nazwą funkcji porównującej “równy z”.

7.3. Przeciążony operator dodawania (+)

Omówimy prosty przykład ilustrujący działanie przeciążonego operatora dodawania. Funkcja operatora jest funkcją składową klasy.

Listing 7.1. Przeciążanie operatora – funkcja operatorowa jest składową

```
#include <iostream>
#include <conio>
using namespace std;

class wek
{ int vx, vy;
  public:
   void ustaw( int ux, int uy);
   void pokaz();
   wek operator+ (wek v);
};

void wek :: ustaw(int ux, int uy)
{ vx = ux;     vy = uy;
}

void wek :: pokaz()
{ cout << "skladowe wektora: ";
  cout << vx << ", " << vy << endl;
}

wek wek :: operator+ ( wek v )
{ wek wektor;
  wektor.vx = vx + v.vx;
  wektor.vy = vy + v.vy;
  return wektor;
}

int main()
{wek a, b, suma_wek;
 a.ustaw(1, 1);
 b.ustaw(1, 2);
 a.pokaz();
 b.pokaz();
 cout << "po dodaniu ";
 suma_wek = a + b;
```

```
37          //   suma_wek = a.operator+(b);
     suma_wek.pokaz();
39     getche();
       return 0;
41  }
```

Po uruchomieniu tego programu otrzymujemy następujący wydruk:

```
skladowe wektora: 1, 1
skladowe wektora: 1, 2
po dodaniu skladowe wektora: 2, 3
```

Ten przykładowy program używa klasy wek i funkcji składowej, która jest przeciążonym operatorem dodawania. W omawianym programie wykonywane jest dodawanie dwóch wektorów (dwuwymiarowych), w wyniku otrzymuje trzeci wektor. Operacja dodawania wektorów polega na oddzielnym dodawaniu składowych poszczególnych wektorów:

$$a + b = c \tag{7.1}$$

$$a_x + b_x = c_x \tag{7.2}$$

$$a_y + b_y = c_y \tag{7.3}$$

Zadanie dodawania wektorów zrealizujemy za pomocą operatora +, przeciążonego w taki sposób, aby wykonać pokazaną metodą dodawanie składowych wektorów. W konwencji używanej przez ten program po dodaniu wektora a do wektora b otrzymamy wektor c. Opracowana klasa ma postać:

```
class wek
{ int vx, vy;
  public:
   void ustaw( int ux, int uy);
   void pokaz();
   wek operator+ (wek v);
};
```

W deklaracji klasy wek widzimy dwie dane prywatne vx i vy, oraz trzy funkcje składowe. Funkcja dostępu ustaw() służy do inicjowania składowych wektora, funkcja pokaz() służy do wyświetlania składowych wektora. W deklaracji klasy wek mamy także funkcję operatorową (funkcja przeciążonego operatora), która nadaje nowe znaczenie operatowi +:

```
wek operator+ (wek v);
```

Definicja funkcji operatorowej ma postać:

```
wek wek :: operator+ ( wek v )
{ wek wektor;
  wektor.vx = vx + v.vx;
  wektor.vy = vy + v.vy;
  return wektor;
}
```

Należy zauważyć, że w tej deklaracji pierwszym argumentem operatora jest wektor, (ponieważ operator jest składową klasy wek), a drugim argumentem jest także wektor (ponieważ typem parametru jest wek). Wek przed słowem kluczowym operator oznacza, że rezultatem użycia operatora (zwracanym przez niego typem) jest nowy wektor. W funkcji main() definiujemy trzy egzemplarze klasy wek, a, b i suma_wek. Dane składowe wektorów a i b inicjalizujemy za pomocą funkcji ustaw(). Do zmiennej suma_wek przypisujemy wynik wywołania przeciążonego operatora:

```
suma_wek = a + b;
```

Należy zauważyć, że funkcja operatorowa ma tylko jeden parametr, a wiemy, że przeciążamy operator dwuargumentowy. W języku C++ obowiązuje zasada, że jeżeli operator dwuargumentowy jest przeciążany za pomocą funkcji składowej klasy, to jawnie przekazywany jest mu tylko jeden argument. Drugi argument jest przekazywany niejawnie za pomocą wskaźnika this. Tak więc, zmienna vx występująca w instrukcji funkcji operator+():

```
wektor.vx = vx + v.vx;
```

oznacza this − > vx, czyli element vx obiektu, który spowodował wywołanie funkcji operatora. Należy zapamiętać następującą regułę:

Obiekt powodujący wywołanie funkcji operatora znajduje się zawsze po lewej stronie operacji. Obiekt występujący po prawej stronie jest przekazywany funkcji w postaci argumentu.

Jeżeli funkcja operatorowa jest składową klasy to przypadku przeciążenia operatora jednoargumentowego nie ma potrzeby przesyłania argumentów. W obu sytuacjach (przeciążanie operatorów jednoargumentowych i przeciążanie operatorów dwuargumentowych) obiekt powodujący wywołanie funkcji operatora jest jej niejawnie przekazywany za pomocą wskaźnika this. W definicji funkcji operatorowej w instrukcji:

```
wektor.vx = vx + v.vx;
```

vx jest prywatną daną klasy wek, udostępnioną za pomocą niejawnego wskaźnika this. Tą instrukcję możemy zapisać z jawnym wskaźnikiem this:

```
wektor.vx = this->vx + v.vx;
```

Z kolei w funkcji testującej instrukcję:

```
suma_wek = a + b;
```

możemy zastąpić ekwiwalentną instrukcją:

```
suma_wek = a.operator+(b);
```

Ten ostatni zapis pomaga zrozumieć, w jaki sposób funkcja operator+() otrzymuje niejawny wskaźnik this, odnoszący się do obiektu klasy a. Za pomocą instrukcji

```
suma_wek = a + b;
```

wektor suma_wek otrzymuje wartości składowych, które są sumą poszczególnych składowych wektorów a i b.

Wykonanie pokazanej instrukcji składa się z dwóch operacji:

1. Najpierw wywołany jest operator dodawania (+), którego dwoma argumentami są wektory a i b. Operator dodawania zwraca nowy tymczasowy obiekt wektora jako wynik swojego działania.
2. Nowy wektor zwrócony przez operator dodawania jest przypisany wektorowi suma_wek za pomocą domyślnego operatora przypisania, który kopiuje składowe wektorów.

Niektóre operatory posiadają specjalne właściwości, do takich należy operator przypisania (=). Domyślny operator przypisania zdefiniowany jest dla każdej klasy. Przypisuje on poszczególne składowe obiektów i zwraca obiekt, któremu przypisana została nowa wartość. Przeciążanie operatora przypisania wymaga rozwiązania kilku ważnych kwestii, szczególnie, gdy składowymi klas są wskaźniki. Istnieją także inne operatory automatycznie generowane dla każdej klasy. Są to:

— operator pobrania adresu (&), obiektu danej klasy
— operator przecinka (,)
— operator tworzenia obiektów dynamicznych (new)
— operator niszczenia obiektów dynamicznych (delete)

7.4. Przeciążony operator mnożenia (*)

Przeciążanie operatora mnożenia (*) zilustrujemy przykładem w którym funkcja operatorowa (operator* ( )) umożliwi mnożenie dwóch ułamków. Funkcja operatorowa jest funkcją składową klasy.

Listing 7.2. mnożenie ułamków – przeciążony operator *

```
1 #include <iostream>
```

```
#include <conio>

using namespace std;

class ulamek
{  int li, mia;
  public:
   ulamek();
   ulamek (int, int);
   ulamek operator * (ulamek);
   void pokaz();
};

ulamek :: ulamek(): li(0), mia(1) //inicjuje ulamek zerem
{ }



ulamek :: ulamek ( int a, int b)
{ if (b == 0)
     { cerr << "mianownik = 0 " << endl;
       exit(EXIT_FAILURE);
     }
  li = a;    mia = b;
}

void ulamek :: pokaz()
{ cout << li << "/" << mia << endl;
}

ulamek ulamek :: operator * ( ulamek x)
{  return ulamek(li * x.li, mia * x.mia);
}

int main()
{ ulamek u1(3, 4), u2(5, 6);
  cout << "ulamek 1 = ";
  u1.pokaz();
  cout << "ulamek 2 = ";
  u2.pokaz();
 ulamek wynik ;
 wynik = u1 * u2;
 cout << "iloczyn ulamkow = ";
 wynik.pokaz();
  getche();
  return 0;
}
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
ulamek 1 = 3/4
```

```
ulamek 2 = 5/6
iloczyn ulamkow = 15/24
```

W pokazanym przykładzie klasa ulamek ma postać:

```
class ulamek
{  int li , mia;
  public:
   ulamek();
   ulamek (int, int);
   ulamek operator * (ulamek);
   void pokaz();
};
```

Ułamek przedstawiamy w postaci licznik/mianownik, wobec czego klasa ulamek ma dwie dane prywatne li i mia (licznik i mianownik). Klasa ma dwa konstruktory oraz funkcję składową pokaz() do wyświetlania licznika i mianownika ułamka. Deklaracje konstruktorów są następujące:

```
ulamek();
ulamek (int, int);
```

Domyślny konstruktor ulamek() ma postać:

```
ulamek :: ulamek(): li(0), mia(1) //inicjuje ulamek zerem
{ }
```

i inicjuje ułamek 0/1 (licznik jest równy zeru, mianownik jest równy jeden). W definicji konstruktora wykorzystaliśmy konstrukcję z tak zwaną listą inicjacji. Po nazwie konstruktora umieszczamy znak dwukropka, po nim pojawia się lista inicjacji atrybutów. Powyższa definicja konstruktora równoważna jest klasycznej konstrukcji:

```
ulamek :: ulamek()
{
  li = 0;
  mia = 1;
}
```

Zauważmy, że w języku C++ możemy deklarować zmienne i inicjować je na trzy sposoby:

1. int x = 0;
2. int x(0);
3. int x;
   x = 0;
   Drugi konstruktor posiada dwa parametry, i wywoływany jest wtedy,

gdy tworzymy konkretny ułamek podając wartość licznika i mianownika. Konstruktor sprawdza także, czy mianownik nie jest zerem:

```
ulamek :: ulamek ( int a, int b)
{ if (b == 0)
    { cerr << "mianownik = 0 " << endl;
      exit(EXIT_FAILURE);
    }
  li = a;      mia = b;
}
```

W klasie ulamek mamy także deklarowana funkcję operatorową:

```
ulamek operator * (ulamek);
```

Widzimy, że pierwszym argumentem operatora mnożenia jest ułamek, ponieważ deklaracja funkcji operatorowej jest wewnątrz klasy ulamek. Drugim argumentem operatora jest także ułamek, ponieważ ten typ został umieszczony na liście argumentów operatora. Rezultatem zastosowania operatora mnożenia będzie nowy ułamek, ponieważ typ ulamek poprzedza słowo kluczowe operator. Implementacja operatora mnożenia ma postać:

```
ulamek ulamek :: operator * ( ulamek x )
{
      return ulamek ( li * x.li , mia * x.mia ) ;
}
```

Operator mnożenia zdefiniowany w klasie ulamek jest operatorem dwuargumentowym. Zastosowanie operatora zakresu postaci ulamek :: sprawia, że pierwszym argumentem operatora mnożenia jest obiekt klasy ulamek, na rzecz którego operator został wywołany. Drugi argument jest umieszczony na liście argumentów operatora mnożenia – jest to także obiekt klasy ulamek. W programie testującym tworzone są dwa ułamki, korzystamy z konstruktorów:

```
ulamek u1(3, 4), u2(5, 6);
```

Tworzone są dwa obiekty: u1 (ułamek 3/4) oraz u2 (ułamek 5/6). Deklarujemy jeszcze jeden ułamek i wykonujemy mnożenie ułamków:

```
ulamek wynik ;
wynik = u1 * u2;
```

W programowaniu obiektowym nie istnieje globalna funkcja mnożenia dwóch ułamków, możemy wykonać jedynie odpowiednią operację. Do istniejącego ułamka wysyłany jest komunikat: “wykonaj mnożenie z przekazanym za po-

mocą parametru ułamkiem". W naszym przypadku komunikatem tym jest funkcja operatorowa, a odbiorcą tego komunikatu jest ułamek. Wyrażenie postaci:

```
u1 * u2
```

interpretowane jest jak:

```
u1.operator*(u2)
```

W kontekście naszego programu operator mnożenia (*) mnoży dwa ułamki, jest to operator dwuargumentowy. Pierwszym argumentem jest ułamek u1 (jest to obiekt, na rzecz którego została wywołana funkcja operatorowa, umieszczony jest on po lewej stronie słowa kluczowego operator). Drugim argumentem jest u2, jest to argument funkcji operatorowej. Wywołana funkcja operatorowa zwraca ułamek, który jest iloczynem ułamków u1 i u2. Najprostsza definicja funkcji operatorowej może mieć postać:

```
ulamek ulamek :: operator* (ulamek x )
{ ulamek ulamek12;
  ulamek12.li = li + x.li;          //oblicza licznik
  ulamek12.mia = mia + x.mia;          //oblicza mianownik
  return ulamek12;          //zwraca wynik mnozenia
}
```

Ta definicja jest zrozumiała, ale możemy ją uprościć.

W bardziej wydajnej implementacji nie tworzymy lokalnego obiektu tymczasowego (ulamek ulamek12):

```
ulamek ulamek :: operator * ( ulamek x)
{  return ulamek(li * x.li , mia * x.mia);
}
```

7.5. Funkcja operatorowa w postaci niezależnej funkcji

Jak już mówiliśmy funkcja operatorowa może być implementowana jako funkcja składowa klasy albo jako zwykła funkcja globalna. Należy pamiętać, że zwykła funkcja nie ma dostępu bezpośredniego do danych prywatnych. Jeżeli chcemy, aby funkcja globalna miała dostęp do danych prywatnych klasy musimy funkcję operatorową zadeklarować jako funkcję zaprzyjaźnioną z klasą. Zanim pokażemy przykład z funkcją operatorową w postaci funkcji zaprzyjaźnionej, omówimy program, dzięki któremu wykonamy mnożenie ułamków korzystając z funkcji zaprzyjaźnionych (nie użyjemy przeciążenia operatora.). Klasa ulamek ma następującą postać:

```
class ulamek
{  int li , mia;
  public:
   ulamek (int , int);
   friend int iloczyn_Li ( ulamek &, ulamek &);
   friend int iloczyn_Mi  ( ulamek &, ulamek &);
   void pokaz();
};
```

Listing 7.3. mnożenie ułamkow – funkcje zaprzyjaźnione

```
#include <iostream>
#include <conio>

using namespace std;

class ulamek
{  int li , mia;
  public:
   ulamek (int , int);         //konstruktor
   friend int iloczyn_Li ( ulamek &, ulamek &);
   friend int iloczyn_Mi  ( ulamek &, ulamek &);
   void pokaz();
};

ulamek :: ulamek ( int a, int b)
{ if (b == 0)
    { cerr << "mianownik = 0 " << endl;
      exit(EXIT_FAILURE);
    }
  li = a;     mia = b;
}

void ulamek :: pokaz()
{ cout << li << "/" << mia << endl;
}
int iloczyn_Li (ulamek &x, ulamek &y)
{  return (x.li * y.li);
}
int iloczyn_Mi (ulamek &x, ulamek &y)
{  return (x.mia * y.mia);
}
int main()
{ ulamek u1(3, 4), u2(5, 6);
  int licznik, mianownik;
  cout << "ulamek 1 = ";
  u1.pokaz();
  cout << "ulamek 2 = ";
  u2.pokaz();
  licznik = iloczyn_Li(u1, u2);
```

```
40     mianownik = iloczyn_Mi (u1, u2);
       ulamek u3(licznik, mianownik);
42     cout << "iloczyn ulamkow = ";
       u3.pokaz();
44     getche();
       return 0;
46  }
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
ulamek 1 = 3/4
ulamek 2 = 5/6
iloczyn ulamkow = 15/24
```

W definicji klasy ulamek mamy dwie dane prywatne li i mia (są to wartości odpowiednio licznika i mianownika), dwie funkcje składowe (konstruktor i funkcja pokaz()) oraz dwie funkcje zaprzyjaźnione. Iloczyn dwóch ułamków jest nowym ułamkiem – jego licznik jest równy iloczynowi liczników a mianownik jest równy iloczynowi mianowników. Do obliczenia iloczynów użyjemy funkcji zaprzyjaźnionych:

```
friend int iloczyn_Li ( ulamek &, ulamek &);
friend int iloczyn_Mi  ( ulamek &, ulamek &);
```

Implementacja tych funkcji ma postać:

```
int iloczyn_Li (ulamek &x, ulamek &y)
{  return (x.li * y.li);
}

int iloczyn_Mi (ulamek &x, ulamek &y)
{  return (x.mia * y.mia);
}
```

Należy zwrócić uwagę, że argumentami funkcji są referencje obiektów. W funkcji testującej tworzymy dwie zmienne pomocnicze licznik i mianownik oraz dwa ułamki u1 i u2:

```
ulamek u1(3, 4), u2(5, 6);
 int licznik, mianownik;
```

a następnie wywołujemy funkcje zaprzyjaźnione:

```
licznik = iloczyn_Li(u1, u2);
mianownik = iloczyn_Mi (u1, u2);
```

dzięki którym obliczamy iloczyn liczników ułamków u1 i u2 oraz iloczyn

mianowników tych ułamków. Mając obliczony nowy licznik i nowy mianownik tworzymy ułamek u3 będący iloczynem ułamków u1 i u2:

```
ulamek u3(licznik, mianownik);
```

Analizując powyższy przykład widzimy, że obliczanie iloczynu ułamków przy pomocy funkcji zaprzyjaźnionych wymaga skomplikowanych wywołań tych funkcji. Pokażemy, że w znaczący sposób możemy uprościć program wprowadzając przeciążony operator mnożenia jako funkcję operatorową zaprzyjaźnioną z klasą.

Listing 7.4. mnożenie ułamków – funkcja operatorowa zaprzyjaźniona

```
1 #include <iostream>
  #include <conio>
3 using namespace std;
  class ulamek
5 { int li, mia;
   public:
7   ulamek ();                    //konstruktor
    ulamek (int, int);            //konstruktor
9   friend ulamek operator*(ulamek, ulamek);
    void pokaz();
11 };

13 ulamek :: ulamek ( int a, int b)
   { if (b == 0)
15     { cerr << "mianownik = 0 " << endl;
         exit(EXIT_FAILURE);
17     }
     li = a;      mia = b;
19 }

21 ulamek :: ulamek ():li(0),mia(1)
   { }
23
   void ulamek :: pokaz()
25 { cout << li << "/" << mia << endl;
   }
27
   ulamek operator*(ulamek x, ulamek y)
29 { ulamek u;
     u.li = x.li * y.li;
31   u.mia = x.mia * y.mia;
     return u;
33 }

35 int main()
   { ulamek u1(3, 4), u2(5, 6), u3;
37   cout << "ulamek 1 = ";
```

```
    u1.pokaz();
39  cout << "ulamek 2 = ";
    u2.pokaz();
41  u3 = u1 * u2;
    cout << "iloczyn ulamkow = ";
43  u3.pokaz();
      getche();
45    return 0;
  }
```

W powyższym przykładzie deklaracja klasy jest następująca:

```
class ulamek
{  int li , mia;
  public:
   ulamek ();                    //konstruktor
   ulamek (int , int);        //konstruktor
   friend ulamek operator*(ulamek , ulamek);
   void pokaz();
};
```

Deklaracja klasy ulamek zawiera deklaracje dwóch danych prywatnych li i mia, dwóch konstruktorów, jednej funkcji składowej pokaz() oraz operatorowej funkcji zaprzyjaźnionej. Należy pamiętać o konstruktorze bezparametrowym, bez jego jawnej definicji kompilacja naszego programu nie powiedzie się. Deklaracja funkcji zaprzyjaźnionej ma postać:

```
friend ulamek operator*(ulamek , ulamek);
```

W języku C++ operatory mogą być przeciążane także przy pomocy funkcji, które nie są składowymi klasy. Można stosować zwykłe funkcje globalne a także funkcje zaprzyjaźnione. Aby dana funkcję zadeklarować jako funkcję zaprzyjaźnioną z konkretną klasą, należy wstawić prototyp takiej funkcji do wnętrza definicji klasy (tak, jakby to była metoda danej klasy) i poprzedzić ten prototyp słowem kluczowym friend. W zastosowaniach taka funkcja będzie traktowana jakby była metodą należącą do danej klasy. Jak pamiętamy, funkcje zaprzyjaźnione mają dostęp do danych prywatnych klasy. W pokazanym przykładzie funkcja operatorowa jest funkcją zaprzyjaźnioną.

Ponieważ do funkcji zaprzyjaźnionych nie jest przekazywany wskaźnik this, zaprzyjaźniona funkcja operatorowa wymaga przekazania operandów w sposób jawny. Wobec tego funkcja operatorowa jednoargumentowa wymaga przekazania jednego parametru, funkcja operatorowa dwuargumentowa wymaga przekazania dwóch argumentów. Należy pamiętać o kolejności przekazywanych argumentów. W funkcji operatorowej zaprzyjaźnionej lewy operand jest przekazywany jako pierwszy, a prawy jako drugi. Do

przeciążania operatorów najczęściej wykorzystuje się funkcje zaprzyjaźnione, ponieważ funkcje zaprzyjaźnione są bardziej elastyczne w porównaniu z funkcjami składowymi. Składowe funkcje operatorowe wymagają zgodności typów operandów, natomiast w przypadku funkcji zaprzyjaźnionych nie jest wymagane, aby lewy operand koniecznie był obiektem klasy. Stąd wynika użyteczność zaprzyjaźnionych funkcji operatorowych - możemy mieszać typy operandów, co jest przydatne, jeżeli chcemy stosować w wyrażeniach obiekty i dane numeryczne. Implementacja przeciążenia operatora mnożenia przy pomocy funkcji zaprzyjaźnionej może mieć postać:

```
ulamek operator*(ulamek x, ulamek y)
{ ulamek u;
  u.li = x.li * y.li;
  u.mia = x.mia * y.mia;
  return u;
}
```

Przypominamy, że w definicji klasy zaprzyjaźnionej nie podajemy nazwy klasy bazowej (bo funkcja zaprzyjaźniona nie jest funkcją składową klasy) łącznie z operatorem zakresu. Funkcja operatorowa pobiera dwa argumenty typu ulamek i zwraca obiekt typu ulamek. W ciele funkcji tworzymy tymczasowy obiekt ulamek u. W funkcji testującej tworzymy trzy obiekty typu ulamek:

```
ulamek u1(3, 4), u2(5, 6), u3;
```

Przy pomocy prostej instrukcji:

```
u3 = u1 * u2;
```

mnożymy dwa ułamki, wykorzystując przeciążony operator mnożenia (*). Tą prostą instrukcję należy porównać z pokazana poprzednio metoda mnożenia ułamków:

```
  licznik = iloczyn_Li(u1, u2);
  mianownik = iloczyn_Mi (u1, u2);
  ulamek u3(licznik, mianownik);
```

aby docenić zalety używania przeciążonych operatorów. Bardziej wydajna wersja programu ilustrującego wykorzystanie przeciążonego operator mnożenia pokazana jest na kolejnym wydruku. Argumenty zaprzyjaźnionej funkcji operatorowej są przekazywane przez referencję. Deklaracja funkcji operatorowej ma postać:

```
friend ulamek operator*(ulamek &x, ulamek &y);
```

Implementacja może mieć postać:

```
ulamek operator * ( ulamek &x, ulamek &y )
{
   return ulamek ( x.li * y.li ,  x.mia * y.mia ) ;
}
```

Jeżeli tą implementację porównamy z implementacją, gdzie argumenty są przekazywane przez wartość:

```
ulamek operator*(ulamek x, ulamek y)
{ ulamek u;
  u.li = x.li * y.li;
  u.mia = x.mia * y.mia;
  return u;
}
```

to widzimy widoczne zalety gdy argumenty przekazywane są przez referencję. Przede wszystkim nie tworzymy kopii argumentów, nie tworzymy także tymczasowego obiektu, który musi być następnie zniszczony. Zagadnienie wydajności w naszym przykładzie nie odgrywa większej roli, gdy mnożymy dwa ułamki, staję się jednak istotne, gdy zechcemy dokonać mnożeń tysięcy ułamków.

Listing 7.5. mnożenie ułamków – funkcja operatorowa zaprzyjaźniona

```
 //funkcja operatorowa, zaprzyjazniona, referencje
2 #include <iostream>
 #include <conio>
4
          using namespace std;
6
 class ulamek
8 {  int li, mia;
   public:
10   ulamek ();                        //konstruktor
     ulamek (int, int);         //konstruktor
12   friend ulamek operator*(ulamek &x, ulamek &y);
     void pokaz();
14 };

16 ulamek :: ulamek ( int a, int b)
 { if (b == 0)
18     { cerr << "mianownik = 0 " << endl;
         exit(EXIT_FAILURE);
20     }
    li = a;     mia = b;
22 }

24 ulamek :: ulamek ():li(0),mia(1)
```

```
{ }
26
void ulamek :: pokaz()
28 { cout << li << "/" << mia << endl;
}
30 ulamek operator*(ulamek &x, ulamek &y)
{ return ulamek(x.li*y.li, x.mia*y.mia);
32 }
int main()
34 { ulamek u1(3, 4), u2(5, 6), u3;
    cout << "ulamek 1 = ";
36   u1.pokaz();
   cout << "ulamek 2 = ";
38   u2.pokaz();
   u3 = u1 * u2;
40   cout << "iloczyn ulamkow = ";
   u3.pokaz();
42   getche();
   return 0;
44 }
```

Mimo zalet związanych ze stosowaniem operatorowych funkcji zaprzyjaźnionych należy pamiętać o szeregu ograniczeń. Podczas przeciążania operatorów inkrementacji i dekrementacji konieczne jest stosowanie parametrów referencyjnych. Kolejnym ograniczenie jest fakt, że za pomocą funkcji zaprzyjaźnionych nie można przeciążać następujących operatorów:

— operator przypisani (=)
— operator odwołania do elementu tablicy ([ ])
— operator odniesienia do składowej klasy ($->$)
— operator ()

7.6. Przeciążanie operatorów równości i nierówności

Zagadnienie przeciążania operatorów równości i nierówności zilustrujemy przykładami. Załóżmy, że chcemy przeładować operatory równości (==) i nierówności (!=). Dla prostoty weźmy klasę reprezentującą trójwymiarowy wektor. Należy przeciążyć operator == oraz != w ten sposób, aby można było ustalić równość i nierówność dwóch wektorów. Przeciążanie należy zrealizować:

— jako funkcje składowe
— jako funkcje zaprzyjaźnion

Definicja klasy może mieć postać:

```
class wektor3d
 {   double x, y, z;
```

```
    public:
        wektor3d (double w1=0.0, double w2=0.0,
          double w3 = 0.0)
           {     x = w1;    y = w2;    z = w3;
           }
        int operator == ( wektor3d );
        int operator != ( wektor3d );
 };
```

W deklaracji klasy wektor3d mamy trzy dane prywatne (składowe wektora), konstruktor oraz dwie operatorowe funkcje składowe.

Listing 7.6. porównanie wektorów – przeciążanie operatorów

```
   //definicja z funkcjami skladowymi
 2 #include <iostream.h>
   #include <conio.h>
 4
   class wektor3d
 6  {   double x, y, z;
      public:
 8        wektor3d (double w1=0.0, double w2=0.0,
            double w3 = 0.0)
10           {     x = w1;    y = w2;    z = w3;
             }
12        int operator == ( wektor3d );
          int operator != ( wektor3d );
14  };

16  int wektor3d::operator == ( wektor3d v )
     {      if ( (v.x == x) && (v.y == y) && (v.z == z) )
18            return 1;
           else return 0;
20   }

22  int wektor3d::operator != (wektor3d v)
     {      return ! ( (*this) == v) ;
24   }

26 int main()
   {
28   wektor3d v1(10,10,20), v2(20,30,40), v3(10,10,20);
                                   // 3 obiekty
30   if (v1 == v2)
       cout << "\n wektory v1 i v2 sa identyczne" << endl;
32   else
       cout << "\n wektory v1 i v2 nie sa rowne" << endl;
34    if (v1 == v3)
       cout << "\n wektory v1 i v3 sa identyczne" << endl;
36   else
       cout << "\n wektory v1 i v3 nie sa rowne" << endl;
```

```
38
  if (v1 != v2)
40      cout << "\n wektory v1 i v2 nie sa rowne" << endl;
          getch();
42        return 0;
  }
```

Po uruchomieniu mamy następujący komunikat:

```
wektory v1 i v2 nie sa rowne
wektory v1 i v3 sa identyczne
wektory v1 i v2 nie sa rowne
```

Jak widać w klasie wektor3d zadeklarowane są dwie funkcje składowe:

```
int operator == (wektor3d);
int operator != (wektor3d);
```

Każda z nich ma jeden argument typu wektor3d i jeden argument domniemany – this. Implementacji funkcji operator== ma postać:

```
int wektor3d::operator == ( wektor3d v )
  {       if ( (v.x == x) && (v.y == y) && (v.z == z) )
                return 1;
          else return 0;
  }
```

Składowe dwóch wektorów są kolejno porównywane, gdy wszystkie są równe zwracana jest wartość 1, w przeciwnym przypadku zwracana jest wartość 0. Należy zwrócić uwagę, że w funkcji implementującej przeciążony operator nierówności wykorzystano przeciążony operator równości! Ta funkcja ma postać:

```
int wektor3d::operator != (wektor3d v)
 {      return ! ( (*this) == v) ;
 }
```

Kolejny wydruk pokazuje przeciążania operatorów równości i nierówności wykorzystując funkcje zaprzyjaźnione.

Listing 7.7. porównanie wektorów – przeciążanie operatorów

```
1 //definicja z funkcjami zaprzyjaznionymi
  #include <iostream.h>
3 #include <conio.h>

5 class wektor3d
   { double x, y, z;
```

```
  public:
        wektor3d (double w1=0.0, double w2=0.0,
          double w3 = 0.0)
              {    x = w1;  y = w2;  z = w3;       }
        friend int operator == (wektor3d, wektor3d);
        friend int operator != (wektor3d, wektor3d);
};

operator == (wektor3d v, wektor3d w)
 {   if ( (v.x == w.x) && (v.y == w.y) && (v.z == w.z) )
    return 1;
      else return 0;
 }

operator != (wektor3d v, wektor3d w)
    {        return ! ( v == w) ;       }

int main()
{
  wektor3d v1(10,10,20), v2(10,10,20), v3(10,20,20);

  cout << " v1 == v2 : " << (v1 == v2 ? "prawda" : "falsz" )
    << endl;
  cout << " v1 == v3 : " << (v1 == v3 ? "prawda" : "falsz" )
    << endl;

  cout << " v1 != v2 : " << (v1 != v2 ? "prawda" : "falsz" )
    << endl;
  cout << " v1 != v3 : " << (v1 != v3 ? "prawda" : "falsz" )
    << endl;
        getch();
        return 0;
}
```

Po uruchomieniu programu otrzymujemy następujący wynik:

```
v1 == v2 : prawda
v1 == v3 : falsz
v1 != v2 : falsz
v1 != v3 : prawda
```

W klasie wektor3d zadeklarowano dwie operatorowe funkcje zaprzyjaźnione o nazwach operator == i operator !=:

```
friend int operator == (wektor3d, wektor3d);
friend int operator != (wektor3d, wektor3d);
```

Będą one otrzymywać po dwa argumenty typu wektor3d. Implementacja tych funkcji ma postać:

```
operator == (wektor3d v, wektor3d w)
  {   if ( (v.x == w.x) && (v.y == w.y) && (v.z == w.z) )
    return 1;
      else return 0;
  }

 operator != (wektor3d v, wektor3d w)
     {       return ! ( v == w) ;
     }
```

Widzimy, że operator przeciążony może być zdefiniowany jako funkcja składowa i jako funkcja globalna (lub zaprzyjaźniona). Wobec tego musimy odpowiedzieć na pytanie, jakie są kryteria wyboru implementacji przeciążania operatora. Prawdę mówiąc nie ma generalnej zasady, wszystko zależy od zastosowania przeciążonego operatora.

Jeżeli operator modyfikuje operandy to powinien być zdefiniowany jako funkcja składowa klasy. Przykładem są tu takie operatory jak: =, +=, -=, *=, ++, itp. Jeżeli operator nie modyfikuje swoich operandów to należy go definiować jako funkcję globalną lub zaprzyjaźnioną. Przykładem są tu takie operatory jak: +, -, ==, &, itp.

7.7. Przeciążanie operatora przypisania (=)

Operator przypisania jest szczególnym operatorem, ponieważ w przypadku, gdy nie zostanie przeciążony, jest on definiowany przez kompilator. Tak więc, operacja przypisani jednego obiektu drugiemu jest zawsze wykonalna. Ilustruje ten fakt kolejny program. W naszym przykładzie została zaimplementowana klasa data:

```
class data
 {
  private:
     int dzien;
     int miesiac;
     int rok;
  public:
     data (int = 1, int = 1, int = 2004 ); //konstruktor
     void pokaz(void); // funkcja skladowa, pokazuje date
 };
```

Ta klasa nie zawiera funkcji operatorowej przypisania. W funkcji głównej main() mamy instrukcję:

```
d1 = d2;
```

co oznacz, że odpowiednie pola obiektu d2 są przypisane polom obiektu d1. Ten typ przypisania nosi nazwę przypisania danych składowych (memberwise assignment).

Listing 7.8. przypisanie bez przeciążania operatorów

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
class data
 { private:
     int dzien;
     int miesiac;
     int rok;
   public:
     data (int = 1, int = 1, int = 2004) ;      //konstruktor
     void pokaz(void);     //funkcja skladowa, pokazuje date
 };

data :: data (int dd, int mm, int rr)
  {     dzien = dd;
        miesiac = mm;
        rok = rr;
  }

 void data :: pokaz(void)
 {     cout << setfill ('0')
                << setw(2) << dzien << '/'
                << setw(2) << miesiac << '/'
                << setw(4) << rok ;
       return;
 }

int main()
{ data d1(13,3,2003), d2( 15, 5, 2004);
  cout << "\n pierwsza data, d1: " ;
  d1.pokaz();
  cout << "\n    druga data, d2: ";
  d2.pokaz();
  d1 = d2;
  cout << "\npo podstawieniu d1: ";
  d1.pokaz();
  cout << endl;
    getche();
    return 0;
}
```

Po uruchomieniu programu otrzymujemy komunikat:

```
pierwsza data, d1: 13/03/2003
druga data d2: 15/05/2004
```

```
po podstawieniu d1: 15/05/2005
```

Jeżeli mamy proste przypisanie, tak jak to pokazano powyżej, wygenerowane przez kompilator przeciążenie jest wystarczające, jednak w bardziej skomplikowanych przypadkach (np. gdy chcemy mieć wielokrotne przypisanie typu: d1 = d2 = d3) musimy zaprojektować odpowiednią funkcję operatorową. W pokazanym programie wykorzystano przeciążenie operatora przypisania wygenerowanego przez kompilator. Plik nagłówkowy <iomanip.h> jest potrzebny, ponieważ w programie wykonywane są formatowane operacje wyjścia z tak zwanymi parametryzowanymi manipulatorami strumienia. Do ustawienia szerokości pola wydruku zastosowano manipulator strumienia setw(), manipulator setfill() określa znak wypełnienia pola. Jak mówiliśmy, operator przypisania (=) jest szczególnym operatorem, podobnie jak operator pobrania adresu (&) i przecinkowy (,) mają predefiniowane znaczenie w odniesieniu do obiektów klas. Oczywiście możemy, przestrzegając odpowiednich reguł przeciążyć operator przypisania. Deklaracja prostego operatora przypisania może mieć postać:

```
void operator = (nazwa_klasy & )
```

Słowo kluczowe void wskazuje, że przypisanie nie zwróci żadnej wartości, napis operator = wskazuje, że przeciążamy operator przypisania, a nazwa klasy i znak & wewnątrz nawiasów okrągłych wskazują, że argumentem operatora jest referencja do klasy. Na przykład, aby zadeklarować operator przypisania dla naszej klasy data, możemy użyć deklaracji:

```
void operator = ( data &);
```

Implementacja funkcji operatorowej może mieć postać:

```
void Data :: operator= (Data & d)
{
 dzien = d.dzien;
 miesiac = d.miesiac;
 rok = d.rok;
}
```

W definicji operatora zastosowano referencję. W tej definicji d jest zdefiniowane jako referencja do klasy Data. W ciele funkcji operatorowej składowa dzien obiektu d jest przypisana składowej dzien aktualnego obiektu:

```
dzien = d.dzien;
```

Ta sama operacja powtórzona jest dla składowych miesiac i rok. Przypisanie typu:

```
a.operator=(b);
```

może być zastosowane do wywołania przeciążonego operatora przypisania i przypisania wartości składowych obiektu b do obiektu a. W tej sytuacji zapis a.operator=(b) może być zastąpiony wygodnym zapisem a = b;.

Listing 7.9. przypisanie; przeciążania operatorów

```
1 #include <iostream>
  #include <iomanip>
3 #include <conio>
  using namespace std;
5 class Data
  { int dzien, miesiac, rok;
7   public:
    Data (int =1, int = 1, int = 2000);   //konstruktor
9   void pokaz ();
    void operator= (Data &);
11 };
  Data :: Data ( int dd, int mm, int rr)
13  {    dzien = dd;
         miesiac = mm;
15       rok = rr;
    }
17 void Data :: pokaz ()
  {   cout << setfill ('0')
19        << setw(2) << dzien << '/'
          << setw(2) << miesiac << '/'
21        << setw(2) << rok % 100 << endl;
  }
23 void Data :: operator= (Data & d)
  {   dzien = d.dzien;
25    miesiac = d.miesiac;
      rok = d.rok;
27 }

29 int main()
  {   Data d1(31, 12, 2004), d2(1,1,2005);
31    cout << "data nr 1: " ;
      d1.pokaz();
33    cout << "data nr 2: " ;
      d2.pokaz();
35    d1 = d2;
      cout << "Przypisanie dat:" << endl;
37    cout << "data nr 1: " ;
      d1.pokaz();
39    getche();
      return 0;
41 }
```

Po uruchomieniu tego programu mamy wydruk:

```
data nr 1: 31/12/04
data nr 2: 01/01/05
Przypisanie dat:
data nr 1: 01/01/05
```

Funkcja operatora przypisania zaprezentowana w tym przykładzie, aczkolwiek poprawna, nie obsłuży poprawnie próby przypisania wielokrotnego postaci:

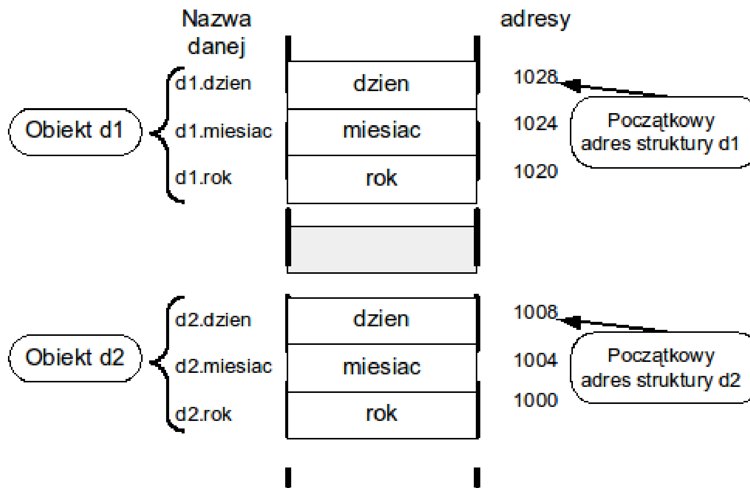
```
a = b = c;
```

Dzieje się tak, ponieważ powyższy zapis interpretowany jest jako:

```
a = ( b = c );
```

Zgodnie z definicją, nasza funkcja operatorowa nie zwraca żadnej wartości, wobec tego po wykonaniu przypisania b = c, żadna wartość nie będzie zwrócona i nic nie możemy przypisać do a. W celu wykonywania wielokrotnego przypisania, potrzebna jest funkcja operatorowa zwracająca referencję do klasy swojego typu. Zanim pokażemy implementację przeciążonego operatora przypisania wielokrotnego, przypomnimy znaczenie słowa kluczowego this. Funkcje składowe są związane z definicja klasy, a nie z deklaracjami obiektów tej klasy. Za każdym razem, gdy obiekt jest kreowany, odrębne obszary pamięci są przydzielane do przechowywania danych składowych. W naszym przykładzie kreowane są dwa obiekty klasy Data – d1 i d2. Organizacja przechowywanych w pamięci danych pokazana jest na rysunku 7.1.

Jak widać na rysunku, każdy zbiór danych ma inny początkowy adres w pamięci, który jest adresem pierwszej danej składowej obiektu. Tego typu kopiowanie danych składowych nie odnosi się do funkcji składowych. Istnieje tylko jeden egzemplarz kodu definicji danej funkcji składowej. Każdy obiekt używa tych samych funkcji. Ponieważ jedna funkcja składowa musi obsłużyć wiele obiektów, musi istnieć sposób identyfikacji danych poszczególnych obiektów. Rozwiązane jest to w ten sposób, że do funkcji przekazywany jest adres wskazujący gdzie w pamięci znajdują się dane składowe konkretnego obiektu. Taki adres jest przekazywany jest poprzez nazwę obiektu. Na przykład, jeżeli używamy naszej klasy Data i założymy, że a jest obiektem tej klasy, to instrukcja a.pokaz() przesyła adres obiektu a do funkcji składowej pokaz(). Możemy zapytać jak taki adres jest przesyłany i gdzie jest przechowywany. Adres jest przechowywany w specjalnej zmiennej wskaźnikowej o nazwie this, która jest automatycznie dostarczana jako ukryty parametr do każdej niestatycznej funkcji składowej, gdy funkcja jest wywoływana. W naszym przykładzie, gdzie klasa Data ma dwie funkcje składowe:



Rysunek 7.1. Przechowywanie dwóch obiektów typu Data w pamięci

```
Data (int =1, int = 1, int = 2000);    //konstruktor
void pokaz ();
```

Lista parametrów przekazywanych ma równoważną postać:

```
Data (Date *this,int =1,int = 1,int = 2000);//konstruktor
void pokaz ( Date *this );
```

W ten sposób, każda funkcja składowa otrzymuje aktualnie dodatkowy argument, który jest adresem struktury danych. Aby się o tym przekonać możemy używać tych wskaźników w sposób jawny. Zademonstrujemy użycie wskaźnika this w krótkim programie (oczywiście w praktyce nie korzysta się z tej techniki).

Listing 7.10. jawne użycie wskaźnika this

```
   #include <iostream>
 2 #include <conio>
   using namespace std;
 4 class punkt
   { int x, y;
 6   public:
      int ustaw_1(int, int);
 8    int ustaw_2(int, int);
   };
10 int punkt :: ustaw_1 (int a, int b)
   {  this->x = a;
12    this->y = b;
      return x;
```

```
14 }
   int punkt :: ustaw_2 (int a, int b)
16 {   this->x = this->x + a;
       this->y = this->y + b;
18     return x;
   }
20
   int main()
22 {   punkt p1;
       cout << " ustaw_1, x = ";
24     cout << p1.ustaw_1(10, 10 ) << endl;
       cout << " ustaw_2, x = ";
26     cout << p1.ustaw_2(20, 20 ) << endl;
       getche();
28     return 0;
   }
```

Po uruchomieniu programu mamy następujący wydruk:

```
Ustaw_1, x = 10
Ustaw_2, x = 30
```

Jak pamiętamy funkcja operatorowa postaci:

```
void Data :: operator= (Data & d)
{   dzien = d.dzien;
    miesiac = d.miesiac;
    rok = d.rok;
}
```

nie umożliwia użycia wielokrotnego przypisania postaci a = b = c. Przy pomocy wskaźnika this zmienimy implementację operatorowej funkcji przypisania tak, aby możliwe było wielokrotne przypisanie. Zasadniczą sprawą jest zaprojektowanie funkcji operator= tak, aby mogła zwrócić wartość typu Data. Prototyp takiej funkcji może mieć postać:

```
Data operator= ( const Data & ) ;
```

Zastosowaliśmy specyfikator const do parametru funkcji, aby mieć pewność, że ten operand nie będzie zmieniony przez funkcję. Implementacja nowej funkcji operatorowej może mieć postać:

```
Data Data :: operator=(const Data &d)
{   dzien = d.dzien;
    miesiac = d.miesiac;
    rok = d.rok;
    return *this;
}
```

Należy pamiętać, że funkcja operatorowa przypisania musi być funkcją składową klasy, nie może być funkcja zaprzyjaźnioną. W przypadku przypisania takiego jak b = c , (równoważna forma b.operator=(c)), wywołana funkcja zmienia dane składowe obiektu b na dane obiektu c i zwraca nową wartość obiektu b. Taka operacja umożliwia wielokrotne przypisanie typu a = b = c. Implementacja przeciążonego operatora przypisania i jego zastosowanie pokazano na kolejnym wydruku.

Listing 7.11. rozszerzona wersja operatora przypisania

```
 #include <iostream>
#include <iomanip>
#include <conio>
using namespace std;
class Data
{ int dzien, miesiac, rok;
  public:
   Data(int, int, int);     //konstruktor
   void pokaz();
   Data operator=(const Data &);
};

Data :: Data(int dd = 1, int mm = 1 , int rr = 2000)
{   dzien = dd;
    miesiac = mm;
    rok = rr;
}

void Data :: pokaz()
{   cout << setfill('0')
         << setw(2) << dzien << '/'
         << setw(2) << miesiac << '/'
         << setw(4) << rok << endl;
}

Data Data :: operator=(const Data &d)
{    dzien = d.dzien;
     miesiac = d.miesiac;
     rok = d.rok;
     return *this;
}
int main()
{   Data d1(2,2,2002), d2(3,3,2003), d3(4,4,2004);
    cout << " pierwsza data: ";
    d1.pokaz();
    cout << "    druga data: ";
    d2.pokaz();
    cout << "  trzecia data: ";
    d3.pokaz();
    d1 = d2 = d3;
```

```
    cout << " przypisanie wielokrotne , teraz ";
42  cout << " pierwsza data: ";
    d1.pokaz();
44  getche();
    return 0;
46 }
```

Po uruchomieniu programu mamy wydruk;

```
pierwsza data: 02/02/2002
druga data: 03.03.2003
trzecia data: 04/04/2004
przypisanie wielokrotne , teraz pierwsza data: 04/04/2004
```

7.8. Przeciążanie operatora wstawiania do strumienia («)

Wiele zastosowań praktycznych mają funkcje realizujące przeciążanie operatorów wstawiania danych («) do strumienia i operatorów pobierania danych (») ze strumienia. Jak wiemy, w języku C++ operator « jest operatorem powodującym przesuwanie bitów o żądaną liczbę pozycji.

Fakt, że możemy użyć tego operatora na przykład przy wyświetlaniu wartości:

```
int x = 13;
cout << x;
```

zawdzięczamy technice przeciążania operatorów. Powyższy zapis ma następującą interpretację:

```
cout.operator<<(x);
```

cout jest egzemplarzem obiektu klasy ostream, klasa ta jest zawarta w bibliotece standardowej. Operator « jest zdefiniowany w klasie ostream, a co więcej jest on przeciążony w ten sposób, że możemy go wykorzystywać dla wszystkich wbudowanych typów danych. Możliwe jest również takie przedefiniowanie tego operatora, aby można było wyświetlać dane typów zdefiniowanych przez użytkownika. Ma to duże znaczenie praktyczne. Rozważmy przykładową klasę punkt:

```
class punkt
{
  private:
    int x ;
    int y ;
  public:
```

```
    punkt(int a, int b) { x = a; y = b;}    //konstruktor
    int pokazX() { return x ; }
    int pokazY() { return y ; }
};
```

W klasie mamy dwie dane prywatne x i y oraz dwie funkcje składowe pokazX() i pokazY() dla uzyskania dostępu do zmiennych prywatnych. Jeżeli chcemy wyswietlić wartości danych x i y do musimy napisać dwie instrukcje strumienia cout z odpowiednimi argumentami dla operatora wstawiania «. To zagadnienie ilustruje pokazany poniżej przykład. Aby wyświetlić wartości danych x i y musimy napisać dwie instrukcje:

```
cout << " x = " << p1.pokazX() << endl;
cout << " y = " << p1.pokazY() << endl;
```

Listing 7.12. operator wstawiania «

```
#include <iostream>
#include <conio>
using namespace std;

class punkt
{ int x,y;
  public:
    punkt(int a, int b) { x = a; y = b;}    //konstruktor
    int pokazX() { return x ; }
    int pokazY() { return y ; }
};

     int main()
{
  punkt p1(5, 15);
    cout << " x = " << p1.pokazX() << endl;
    cout << " y = " << p1.pokazY() << endl;
    getche();
    return 0;
}
```

W kolejnym przykładzie pokażemy jak można przeciążyć operator wstawiania, aby można było wyświetlić dane obiektu przy pomocy jednej instrukcji. Klasyczna postać definicji operatora wstawiania («) jest następująca:

```
ostream & operator<< (ostream & os, nazwa_klasy & ob)
{
//instrukcje
return os;
}
```

Zdefiniowana funkcja jest klasy ostream &, czyli musi podawać referencję do obiektu klasy ostream &. Pierwszym argumentem funkcji operator«() jest referencja do obiektu typu ostream. Oznacza to, że os musi być strumieniem wyjściowym. Drugim argumentem także jest referencja, do argumentu ob przesyła się obiekt typu nazwa_klasy. Funkcja operator« zawsze zwraca referencję do swojego pierwszego argumentu, to znaczy do strumienia wyjściowego os. Zaprojektowana funkcja operatorowa musi być zaprzyjaźniona z klasą nazwa_klasy, jeżeli chce mieć dostęp do składowych chronionych klasy. Jeżeli mamy następującą klasę:

```
class punkt
{
    int x,y;
  public:
    punkt(int a, int b) { x = a; y = b;}    //konstruktor
    friend ostream& operator<< (ostream &, punkt & );
};
```

to funkcja operatorowa może mieć postać:

```
ostream & operator<< (ostream & os, punkt & ob)
{   os << "x = " << ob.x << endl;
    os << "y = " << ob.y << endl;
    return os;
}
```

Krótki program ilustrujący omawiane zagadnienie pokazany jest na wydruku 7.13.

Listing 7.13. przeciążony operator wstawiania «

```
1  #include <iostream>
   #include <conio>
3  using namespace std;

5  class punkt
   { int x,y;
7    public:
      punkt(int a, int b) { x = a; y = b;}    //konstruktor
9     friend ostream& operator<< (ostream &, punkt & );
   };
11
   ostream & operator<< (ostream & os, punkt & ob)
13 {   os << "x = " << ob.x << endl;
       os << "y = " << ob.y << endl;
15     return os;
   }
17
   int main()
```

```
19 {    punkt p1(5, 15);
        cout << p1;
21      cout << "wywolanie jawne: \n";
        operator<<(cout, p1);
23      getche();
        return 0;
25 }
```

Po wykonaniu tego programu mamy następujący wynik:

```
x = 5
y = 15
wywolanie jawne:
x = 5
y = 15
```

Jak pokazano w programie, wywołanie funkcji operatorowej może mieć dwie formy:

```
cout << p1;
```

albo

```
operator<<(cout, p1);
```

Rozdział 8

Funkcje statyczne i wirtualne

8.1. Wstęp **178**
8.2. Metody i dane statyczne **178**
8.3. Polimorfizm **183**
8.4. Funkcje wirtualne **187**
8.5. Funkcje abstrakcyjne **194**

8.1. Wstęp

Jedną z fundamentalnych cech programowania obiektowego jest polimorfizm (innymi ważnymi cechami, jak pamiętamy są abstrakcja danych, ukrywanie danych i dziedziczenie). Polimorfizm zapewnia tworzenie bardziej zrozumiałego kodu a w powiązaniu z wykorzystaniem funkcji wirtualnych umożliwia zaprojektowanie i wdrożenie programów, które są stosunkowo bardziej rozszerzalne. Na etapie kompilacji polimorfizm jest realizowany przy pomocy przeciążanych funkcji i operatorów, na etapie wykonania jest realizowany przy pomocy dziedziczenia i funkcji wirtualnych. Dla wielu początkujących programistów pojęcie funkcji wirtualnych może być początkowo niejasne, ponieważ nie posiadają one swoich odpowiedników w językach proceduralnych. Twierdzi się, że funkcje wirtualne są fundamentem programowania obiektowego.

8.2. Metody i dane statyczne

W języku C++ każdy obiekt klasy posiada swoją własną kopię danych składowych. Jeżeli tworzymy w programie wiele obiektów danej klasy, często żądamy istnienia tylko jednej kopii zmiennej. Taki przypadek występuje często w przypadku żądania informacji o liczbie obiektów. Oczywiście można zastosować zmienne globalne, ale nie jest to rozwiązanie elegancki i bezpieczne. W języku C++ dane składowe klasy mogą być zadeklarowane jako statyczne składowe klasy. Aby zadeklarować statyczne dane, należy umieścić przed deklaracją składowej słowo kluczowe static. W ten sposób powstają dane które należą do klasy jako całość a nie do poszczególnych obiektów pojedynczo. Składowe statyczne tworzone są tylko raz, każdy obiekt danej klasy posiada dostęp do składowych statycznych klasy. Dostęp do statycznych składowych klasy kontrolowany jest przy pomocy specyfikatorów public, protected i private. Zmienną statyczną należy definiować poza klasą, nie można używać (powtórnie) słowa kluczowego static. Składowe statyczne muszą być inicjalizowane tylko raz w zasięgu pliku. Co ciekawsze, statyczne składowe istnieją nawet wtedy, gdy nie ma żadnego obiektu. Dana składowa static inicjalizowana jest najczęściej wartością zero, gdy nie wykonamy jawnej inicjalizacji, dana składowa static przez domniemanie otrzyma wartość 0 (zero). Przykładowa klasa z danymi typu static może mieć postać:

```
class ST
{   private:
      int x;
      static int y;
      static int z;
};
```

```
int ST :: y;
int ST :: z = 1;
```

W pokazanej klasie mamy dwie składowe y i z zadeklarowane jako statyczne. Te zmienne mamy zdefiniowane poza klasą (co oznacza, że przydzielona została im pamięć). Do tak zdefiniowanych zmiennych statycznych możemy odwoływać się przez pełną kwalifikowaną nazwę:

```
ST :: y ;
```

lub przy pomocy operatora wyboru (kropki lub strzałki). W pokazanym przykładzie składowa statyczna y zostanie zainicjalizowana wartością zero. Poniższy przykład ilustruje zastosowanie danych statycznych do zliczania tworzonych obiektów danej klasy.

Listing 8.1. Dane składowe statyczne

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;
  class licznik
5 { public:
     static int n;
7    licznik () { n++ ; }
  };
9
    int licznik :: n;        // definicja danej statycznej
11
  int main()
13 {
    licznik obj1;
15  cout << " ilosc objektow: " << licznik ::n << endl;
    licznik obj2;
17  cout << " ilosc objektow: " << licznik ::n << endl;
     getche();
19   return 0;
   }
```

Po uruchomieniu programu mamy następujący wydruk:

```
ilosc objektow : 1
ilosc obiektow : 2
```

Możemy także deklarować metody klasy jako statyczne, pamiętając jednak o kilku ograniczeniach. Statyczne funkcje składowe mogą odwoływać się bezpośrednio jedynie do danych statycznych klasy (i oczywiście do danych globalnych). Składowe funkcje statyczne nie posiadają wskaźnika this,

nie można także tworzyć statycznych funkcji wirtualnych ani deklarować je jako const. W kolejnym przykładzie ilustrujemy wykorzystanie funkcji statycznych. W programie obsługujemy listę pacjentów. Dana składowa n jest inicjalizowana początkowo wartością 0.

```
int pacjent :: n = 0 ;
```

Listing 8.2. Statyczne dane składowe i metody

```
1  #include <iostream>
   #include <string.h>
3  #include <conio.h>
   using namespace std;
5  class pacjent
   { public:
7     pacjent ( char *,  char *);
      ~pacjent();
9      char *get_imie()  ;
       char *get_nazwisko();
11    static int liczba();           / /statyczna funkcja
    private:
13    char *imie;
      char *nazwisko;
15    static int n;                  // liczba pacjentow
   };
17
    int pacjent :: n = 0 ;      // inicjacja danej statycznej
19
   int pacjent :: liczba() { return n ; }
21
    pacjent :: pacjent ( char *ximie, char *xnazwisko)
23  { imie = new char[strlen(ximie) + 1 ];
      strcpy(imie, ximie);
25    nazwisko = new char[strlen(xnazwisko) + 1 ];
      strcpy(nazwisko, xnazwisko);
27    ++n;
      cout << " pacjent: " <<imie << "  "
29     << nazwisko<< endl;
    }
31
    pacjent :: ~pacjent()
33  { cout <<" zbadany pacjent "<< imie << "  "
       << nazwisko << endl;
35    delete imie;
      delete nazwisko;
37    --n;
    }
39
    char *pacjent :: get_imie() { return imie; }
41  char *pacjent :: get_nazwisko()  { return nazwisko; }
```

```
43 int main()
   {
45   cout << "liczba pacjentow : "
          << pacjent :: liczba() << endl;
47   pacjent *p1 = new pacjent ( "Jan", "Fasola");
     pacjent *p2 = new pacjent ( "Emil", "Burak");
49   pacjent *p3 = new pacjent ( "Borys", "Delfin");
     cout << "liczba pacjentow : " << p1->liczba()
51        << endl;
     delete p1;
53   delete p2;
     delete p3;
55   cout << "pozostala liczba pacjentow : "
          << pacjent::liczba() << endl;
57    getche();
      return 0;
59 }
```

Dana składowa n przechowuje liczbę obiektów klasy pacjent. Mamy także statyczną funkcje składową liczba(), która dostarcza informacji o liczbie utworzonych obiektów klasy pacjent:

```
int pacjent :: liczba() { return n ; }
```

Konstruktor dynamicznie przydziela pamięć dla danych pacjenta oraz inkrementuje statyczną daną n :

```
pacjent :: pacjent ( char *ximie, char *xnazwisko)
{ imie = new char[ strlen ( ximie ) + 1 ];
  strcpy( imie, ximie ) ;
  nazwisko = new char[strlen(xnazwisko) + 1 ];
  strcpy( nazwisko, xnazwisko) ;
  ++n;          // zwieksza licznik pacjentow
  cout << " pacjent: " <<imie << "  " << nazwisko<< endl;
}
```

Do zwalniania pamięci i dekrementacji statycznej danej n wykorzystujemy destruktor:

```
pacjent :: ~pacjent()
{ cout <<" zbadany pacjent "<< imie << "  "
   << nazwisko << endl;
  delete imie;
  delete nazwisko;
  --n;      // zmniejsza licznik pacjentow
}
```

W funkcji main() testujemy klasę pacjent. Najpierw tworzymy trzy obiekty klasy pacjent i przy pomocy instrukcji:

```
cout << "liczba pacjentow : " << p1->liczba() << endl;
```

wyświetlamy liczbę utworzonych obiektów. Następnie przy pomocy operatora delete zwalniamy pamięć. W naszym przypadku w tym momencie nie istnieje już żaden utworzony obiekt klasy pacjent. W takim przypadku informacja o ilości obiektów dostępna jest jedynie za pomocą statycznej metody liczba() :

```
cout << "pozostala liczba pacjentow : "
     << pacjent::liczba() << endl;
```

Metoda liczba() służy do informowania o liczbie aktualnie istniejących obiektów klasy pacjent. Należy zwrócić uwagę, że podczas zwalniania pamięci (operator delete) wywoływany jest destruktor obiektu i wtedy następuje dekrementacja statycznej danej składowej n.

Po uruchomieniu programu mamy następujący wynik:

```
liczba pacjentow : 0
 pacjent : Jan Fasola
 pacjent : Emil Burak
 pacjent  : Borys Delfin
liczba pacjentow : 3
 zbadany pacjent Jan Fasola
 zbadany pacjent Emil Burak
 zbadany pacjent Borys Delfin
pozostala liczba pacjentow : 0
```

Kolejny przykład ilustruje zastosowanie statycznej metody do obliczania iloczynu skalarnego dla dwóch wektorów 3D.

Listing 8.3. Statyczne metody – iloczyn skalarny

```
 1 #include <iostream>
   #include <conio.h>
 3 using namespace std;

 5 class wektor
   {   int x,y,z;
 7   public:
       void set (int, int, int);
 9     static int iloczyn (wektor &, wektor &);
   };
11 void wektor :: set(int xx, int yy, int zz)
   { x=xx ;  y=yy ;  z=zz ;
13 }
```

```
   int wektor :: iloczyn (wektor &w1, wektor &w2)
15  { return w1.x*w2.x + w1.y*w2.y + w1.z*w2.z ;
    }
17
   int main()
19 { wektor w1, w2;
     w1.set(1,1,1);
21   w2.set(2,2,2);
      cout <<"iloczyn = " << wektor :: iloczyn(w1,w2) << endl;
23   w1.set(5,5,5);
      cout <<"iloczyn = " << w1.iloczyn(w1,w2) << endl;
25    getche();
      return 0;
27 }
```

Wynik wykonania programy ma postać:

```
iloczyn = 6
iloczyn = 30
```

Klasa wektor ma dwie metody – set() do ustawiania wartości składowych wektora oraz statyczną metodę iloczyn() do obliczania iloczynu skalarnego dwóch wektorów:

```
int wektor :: iloczyn (wektor &w1, wektor &w2)
 { return w1.x*w2.x + w1.y*w2.y + w1.z*w2.z ;
 }
```

W funkcji testującej main() demonstrujemy dwa sposoby wywołania metody statycznej:

```
cout <<"iloczyn = " << wektor :: iloczyn(w1,w2) << endl;
cout <<"iloczyn = " << w1.iloczyn(w1,w2) << endl;
```

W pierwszym wywołaniu funkcja iloczyn() może być wywołana samodzielnie, niezależnie od obiektu. Musimy w wyrażeniu użyć nazwy klasy i operatora zakresu. W drugim przypadku funkcja iloczyn() jest wywołana na rzecz obiektu w1, co jest standardowym wywołaniem metody.

Należy zauważyć, że dostęp do zmiennych i wywołania funkcji statycznych danej klasy nie wymagają istnienia obiektów tej klasy.

8.3. Polimorfizm

Ważna cechą programowania obiektowego w C++ jest polimorfizm. Dzięki polimorfizmowi istnieje możliwość otrzymania różnego działania metod w odpowiedzi na ten sam komunikat . Taka technika jest wykorzystywana w

przypadku gdy metody są wywoływane przez obiekty różnych powiązanych klas, a chcemy żeby działanie ich zależało od konkretnej sytuacji. Ten sam komunikat przesłany do różnych typów obiektów przybiera różne formy – stąd nazwa polimorfizm. Idea polimorfizmu realizowana jest w C++ przy pomocy funkcji wirtualnych.

Najprostszą techniką polimorfizmu jest tzw. nadpisanie funkcji (ang. overriding of a function). Taki przypadek zachodzi, gdy funkcja zdefiniowana w klasie bazowej otrzymuje nową postać w klasie dziedziczącej. Polimorfizm pozwala funkcji o tej samej nazwie wywołanej na rzecz obiektu klasy bazowej dać inny wynik niż wywołanie tej funkcji na rzecz obiektu klasy pochodnej. W wielu przypadkach metoda nadpisywania nie działa tak jak oczekujemy. Poprawne działanie nadpisywania ilustruje kolejny przykład.

Listing 8.4. nadpisanie metody klasy bazowej

```
#include <iostream>
#include <conio.h>
#include <math>
using namespace std;

const double PI = 2.0*asin(1.0);

class kolo
{ protected:
     double r;
  public:
     kolo(double rr = 1);
     double oblicz();
};

kolo :: kolo(double rr) { r = rr ; }
double kolo :: oblicz() { return PI*r*r ;}

class walec : public kolo
{ protected:
     double h;
  public:
      walec(double rr = 1.0, double hh = 1.0) : kolo(rr),
                                 h(hh) { }
      double oblicz();
};
double walec :: oblicz()
{ return (h*kolo::oblicz()) ; }

int main()
{
   kolo k1, k2(2);
   walec walec1(3,4);
   cout << "pole kola k1 = " << k1.oblicz() << endl;
```

```
    cout << "pole␣kola␣k2␣=␣" << k2.oblicz() << endl;
36  cout << "objetosc␣walca␣=␣" << walec1.oblicz() << endl;
    getche();
38  return 0;
}
```

Wynikiem uruchomienia programu jest komunikat:

```
pole kola k1 = 3.14159
pole kola k2 = 12.5664
objetosc walca = 113.097
```

W programie należy zwrócić uwagę na definicję:

```
const double PI = 2.0 * asin(1.0) ;
```

Jest to sprytny sposób wymuszenia na kompilatorze zwrócenia wartości liczby Pi z maksymalną precyzją jaką oferuje nasz komputer.

W pokazanych klasach programu mamy funkcje o takiej samej nazwie – oblicz(). Metoda w klasie bazowej oblicza pole powierzchni koła, w klasie pochodne oblicza objętość walca. Nadpisanie bazowej funkcji składowej przez przeciążoną pochodną funkcję składową, tak jak to pokazano w programie jest przykładem polimorfizmu. Polimorfizm pozwala na różne działania funkcji składowej o takiej samej nazwie w zależności na rzecz jakiego obiektu została wywołana. Te różne wywołania widać w instrukcjach:

```
cout << "pole␣kola␣k2␣=␣" << k2.oblicz() << endl;
cout << "objetosc␣walca␣=␣" << walec1.oblicz() << endl;
```

Można użyć wskaźnika do obiektu klasy bazowej. Jeżeli użyjemy wskaźnika typu klasy bazowej do wywołania metody w hierarchii klas, zawsze wywoływana jest metoda z egzemplarza klasy bazowej. Zagadnienie to ilustrujemy przez zmodyfikowanie poprzedniego programu.

Listing 8.5. nadpisanie metody klasy bazowej; błędne użycie wskaźnika

```
  #include <iostream>
2 #include <conio.h>
  #include <math>
4 using namespace std;

6 const double PI = 2.0*asin(1.0);

8 class kolo
  { protected:
10     double r;
    public:
```

```
12        kolo(double rr = 1);
          double oblicz();
14  };
    kolo :: kolo(double rr) { r = rr ; }
16  double kolo :: oblicz() { return PI*r*r ;}

18  class walec : public kolo
    { protected:
20        double h;
      public:
22         walec(double rr = 1.0, double hh = 1.0) : kolo(rr),
                                        h(hh) { }
24         double oblicz();
    };
26  double walec :: oblicz()
    { return (h*kolo::oblicz()) ; }
28
    int main()
30  {
        kolo k1;
32      walec walec1(3,4);
        kolo *wsk;
34      wsk = & k1;
         cout << "wsk pole kola k1 = " << wsk->oblicz() << endl;
36      wsk = & walec1;
         cout << "wsk objetosc walca = " << wsk->oblicz()
38        << endl;
        getche();
40      return 0;
    }
```

Wynikiem działania programu jest komunikat:

```
wsk pole kola k1 = 3.14159
wsk objetosc walca = 28.2743
```

We fragmencie kodu:

```
kolo k1;
walec walec1(3,4);
kolo *wsk;
wsk = & k1;
 cout << "wsk pole kola k1 = " << wsk->oblicz() << endl;
wsk = & walec1;
 cout << "wsk objetosc walca = " << wsk->oblicz()
  << endl;
```

zdefiniowaliśmy egzemplarze obu klas oraz wskaźnik typu klasy bazowej wsk. Wywołanie:

```
wsk = & k1;
wsk->oblicz();
```

zadziała prawidłowo, natomiast ustawienia wskaźnika na walec1:

```
wsk = & walec1;
```

spowoduje, że kolejne wywołanie

```
wsk->oblicz()
```

nie zadziała tak jak się spodziewamy.

Otrzymany wynik nie jest prawidłowy, użycie wskaźnika w pokazany sposób jest kłopotliwe. Chcemy mieć możliwość używania wskaźnika klasy bazowej w celu dostępu do egzemplarzy dowolnej z klas pochodnych w tej samej hierarchii. Do osiągnięcia polimorfizmu musimy zastosować specjalny mechanizm – użyć funkcji wirtualnych.

8.4. Funkcje wirtualne

Funkcja wirtualne jest to funkcja składowa zadeklarowana w klasie bazowej przy pomocy słowa kluczowego virtual i zdefiniowana w klasie pochodnej:

```
virtual typ_zwracany nazwa_funkcji() ;
```

W klasie pochodnej, która dziedziczy metody klasy bazowej definiujemy na nowo funkcję wirtualną, tak aby uwzględnić specyfikę klasy pochodnej. Realizujemy więc zasadę polimorfizmu : „jeden interfejs, wiele metod”. Każda definicja funkcji wirtualnej w klasie pochodnej tworzy nową metodę. Wirtualność metody jest cechą dziedziczoną. Funkcja zdefiniowana w klasie bazowej jako wirtualna pozostaje taką do końca działania programu – nie ma możliwości zdjęcia wirtualności z funkcji składowej. W klasie pochodnej podczas definiowania funkcji wirtualnej nie ma potrzeby używania słowa kluczowego virtual, ale jest to zalecane ze względu na czytelność kodu. Pamiętać należy, że funkcje wirtualne spowalniają wykonanie kodu (nieznaczne). W zasadzie funkcja wirtualna w typowym działaniu zachowuje się jak zwykła funkcja składowa klasy. Cała różnica działania funkcji wirtualnych uwidacznia się gdy jest wywoływana przez wskaźnik.

Dokładnie omówimy prosty przykład (skorzystamy z koncepcji H.Schildta, “Programowanie C++”), aby zapoznać się z działaniem metod wirtualnych.

Listing 8.6. funkcje wirtualne; poprawne użycie wskaźnika

```
#include <iostream>
#include <conio.h>
using namespace std;

class bazowa
{ public:
    virtual void wirfun() {
     cout << "  wirtualna - klasa bazowa "<< endl;
    }
};

class pochodna : public bazowa
{ public:
   virtual void wirfun() {
     cout << "  wirtualna - klasa pochodna " << endl;
   }
};

//referencja funkcji bazowej jako parametr:
void reffun(bazowa &z ) { z.wirfun() ; }

int main()
{ bazowa *wsk;           //wskaznik klasy bazowej
   bazowa x;             //obiekt typu bazowa
   pochodna y;           //obiekt typu pochodna
   cout << "wywolanie funkcji wirtualnej
        przy pomocy wskaznika:"<< endl;
    wsk = &x;            //przypisanie adresu obiektu
              //bazowego x
    wsk -> wirfun();     // metod klasy bazowa
    wsk = &y;            // przypisanie adresu obiektu
              //pochodnego y
    wsk -> wirfun();     // metod klasy pochodna
   cout << "wywolanie funkcji wirtualnej
        przy pomocy referencji:"<< endl;
    reffun(x);           //przekazanie obiektu bazowego
            //do reffun()
    reffun(y);           //przekazanie obiektu pochodnego
            //do reffun()
   getche();
   return 0;
}
```

Po uruchomieniu programu uzyskujemy następujący komunikat:

```
wywolanie funkcji wirtualnej przy pomocy wskaznika
wirtualna - klasa bazowa
wirtualna - klasa pochodna
wywolanie funkcji wirtualnej przy pomocy referencji
```

```
wirtualna – klasa bazowa
wirtualna – klasa pochodna
```

W naszym programie mamy klasę bazową i pochodną, w każdej z nich zdefiniowana jest metoda wirfun(), zgodnie z oczekiwaniami klasy. Funkcja main() testuje nasze klasy. W programie mamy zadeklarowane cztery zmienne: wsk (wskaźnik klasy bazowej), x (obiekt klasy bazowej), y (obiekt klasy pochodnej) i z (referencja klasy bazowej).

W instrukcjach

```
wsk = &x;          //przypisanie adresu obiektu bazowego x
wsk -> wirfun(); // metod klasy bazowa
```

zmiennej wsk jest przypisany adres x i wywoływana jest funkcja wirfun(). Dzięki temu, że wsk jest wskaźnikiem do obiektu klasy bazowej, wykonana zostanie wersja tej funkcji zdefiniowana w klasie bazowej. W kolejnych instrukcjach:

```
wsk = &y;          // przypisanie adresu obiektu pochodnego y
wsk -> wirfun(); // metod klasy pochodna
```

zmiennej wsk przypisywany jest adres y (obiekt klasy pochodnej) a metoda wirfun() jest ponownie wywoływana. W takim przypadku wywoływana jest metoda wirfun() zdefiniowana w klasie pochodnej. Najważniejsze jest to, że wybór wersji funkcji wirfun() dokonywany jest na podstawie typu obiektu wskazywanego przez wskaźnik wsk. Mówimy, że realizacja wersji metody wykonywana jest w trakcie działania programu, co oznacza, że realizowany jest polimorfizm na etapie wykonania. Należy zwrócić uwagę, że klasyczne przeciążanie funkcji a nadpisywanie funkcji przy pomocy metod wirtualnych jest całkiem innym mechanizmem. Konieczne jest, aby prototyp funkcji nadpisywany w klasie pochodnej był identyczny jak w klasie bazowej. Pamiętamy, że podczas klasycznego nadpisywania, sygnatury funkcji muszą się różnić. Niesie to pewne niebezpieczeństwo. Jeżeli w trakcie stosowania metod wirtualnych zmienimy przypadkowo jakiś element prototypu, funkcji nadany zostanie status funkcji przeciążonej i może to spowodować błędne wykonanie programu. W instrukcji:

```
//referencja funkcji bazowej jako parametr:
void reffun(bazowa &z ) { z.wirfun() ; }
```

mamy prototyp funkcji, której argumentem jest referencja klasy bazowej. Jeżeli zgodzimy się, że referencja jest niejawnym wskaźnikiem, to jasne staje się, że z polimorficznych własności funkcji wirtualnych możemy skorzystać, wywołując je za pomocą referencji. W takim przypadku, podobnie jak przy

wykorzystaniu wskaźnika, o wyborze wersji funkcji decyduje rodzaj obiektu wskazywanego przez referencję w momencie wywoływania. W kolejnych instrukcjach mamy pokazane wywołanie odpowiednich metod przy pomocy referencji:

```
cout << "wywolanie funkcji wirtualnej
przy pomocy referencji:"<< endl;
reffun(x); //przekazanie obiektu bazowego do reffun()
reffun(y); //przekazanie obiektu pochodnego do reffun()
```

Mając możliwość użycia metod wirtualnych możemy poprawić źle działający program w którym ilustrowaliśmy poprzednio nadpisywanie funkcji. Poprawny kod pokazany jest na kolejnym listingu.

Listing 8.7. funkcje wirtualne; poprawne użycie wskaźnika

```
  #include <iostream>
2 #include <conio.h>
  #include <math>
4 using namespace std;

6 const double PI = 2.0*asin(1.0);

8 class kolo
  { protected:
10      double r;
    public:
12      kolo(double rr = 1);
        virtual double oblicz(); // wirtualna funkcja skladowa
14 };

16 kolo :: kolo(double rr) { r = rr ; }
   double kolo :: oblicz() { return PI*r*r ;}
18
   class walec : public kolo
20 { protected:
        double h;
22   public:
         walec(double rr = 1.0, double hh = 1.0) : kolo(rr),
24                                  h(hh) { }
         double oblicz();
26 };

28 double walec :: oblicz()
   { return (h*kolo::oblicz()) ; }
30
   int main()
32 {
      kolo k1;
34    walec walec1(3,4);
```

```
    kolo *wsk;
36  wsk = & k1;
     cout << "wskaznik, pole kola k1 = " << wsk->oblicz()
38        << endl;
    wsk = & walec1;
40   cout << "wskaznik, objetosc walca = " << wsk->oblicz()
          << endl;
42  getche();
    return 0;
44      }
```

Wynikiem działania tego programu są poprawne obliczenia:

```
wskaznik, pole kola k1 = 3.14159
wskaznik, objetosc walca = 113.097
```

Do osiągnięcia polimorfizmu musimy zastosować funkcję wirtualną. W naszym przypadku funkcja oblicz() deklarowana w klasie bazowej kolo musi być wyszczególniona jako virtual:

```
virtual double oblicz(); // wirtualna funkcja skladowa
```

Teraz użycie wskaźnika do obsługi wywołania metody oblicz() działa prawidłowo! Jeżeli w klasie pochodnej nadpisywana jest funkcja standardowa, zdefiniowana w klasie bazowej (nie wirtualna) mamy do czynienia z procesem nazywanym wiązaniem funkcji (ang. function binding). W typowym wywołaniu funkcji jest wykonane tzw. statyczne wiązanie (ang. static binding). W statycznym wiązaniu decyzja, jaką wersję funkcji należy zrealizować jest wykonana na etapie czasu kompilacji (ang. compile time). W wielu przypadkach, chcemy aby zamiast decyzji o wiązaniu statycznym, decyzja o wersji metody była podejmowana później, w czasie wykonania (ang. run time), na podstawie typu obiektu, który wykonał wywołanie funkcji. Oczywiście taki mechanizm zapewnia polimorfizm, konkretnie funkcje wirtualne. Ten typ wiązania funkcji nosi nazwę wiązania dynamicznego (ang. dynamic binding). Specyfikacja funkcji wirtualnej informuje kompilator języka C++ aby utworzył wskaźnik do funkcji lecz nie nadawał wartości wskaźnikowi dopóki funkcja nie będzie wywołana. W tej sytuacji w czasie wykonywania programu, na podstawie typu obiektu wykonującego wywołanie metody, odpowiedni adres jest wykorzystany i odpowiednia wersja funkcji jest zastosowana.

Jak już pisaliśmy, wykorzystanie funkcji wirtualnych nieznacznie spowalnia wykonanie programu w porównaniu z wykorzystaniem funkcji klasycznych. Jak już opisaliśmy, dla klas niepolimorficznych metoda jest wybierana w czasie kompilacji, mechanizm nosi nazwę wczesnego wiązania (ang. ear-

ly binding). Typ dynamiczny obiektu (zastosowanie wskaźnika) może być określony w czasie wykonania programu. Oznacza to, że w fazie kompilacji po stwierdzeniu wywołania metody z klasy polimorficznej, kompilator nie może umieścić kodu wykonywalnego odpowiedniej funkcji. Zamiast niego jest umieszczany kod sprawdzający i podejmujący decyzję później o wersji funkcji. Tego typu mechanizm nosi też nazwę późnego wiązania (ang. late binding). Wszystko to powoduje nieznaczne spowolnienie wykonania programu a także powiększenie niezbędnej pamięci.

Gdy mamy wybierać pomiędzy zwykłą funkcją składową a wirtualną – zaleca się wybór metody wirtualnej. Zaleca się także stosowanie wirtualnych destruktorów – zapobiega to wyciekowi pamięci, ponieważ gdy destruktor nie jest wirtualny to będzie wywołany destruktor właściwy dla typu obiektu w czasie kompilacji (wczesne wiązanie). Tworząc implementację hierarchii klas zaleca się aby wszystkie klasy w korzeniu drzewa dziedziczenia miały metody wirtualne. Zwracamy uwagę na fakt, że funkcje statyczne nie mogą być wirtualnie i tak samo metody wirtualne nie mogą być statyczne. Jeżeli w klasie pochodnej funkcja nie jest deklarowana jako wirtualna, klasa ta dziedziczy bezpośrednio definicję funkcji wirtualnej z klasy bazowej.

Projektowanie polimorficznych klas wymaga dużej uwagi. Jednym z problemów który może się pojawić jest działanie destruktora. W przetwarzaniu dynamicznym, w trakcie niszczenia obiektu przy zastosowaniu operatora delete do wskaźnika klasy bazowej, wywoływana jest funkcja destruktora tej klasy. Jest to niezależne od typu obiektu wskazywanego przez wskaźnik klasy bazowej. Poniższy program ilustruje fakt wywołania destruktora bazowego, mimo ustawienia wskaźnika na obiekt klasy pochodnej (spodziewamy się wywołania destruktora klasy pochodnej).

Listing 8.8. Desktruktor w hierarchii klas – błędne użycie

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;

5 class bazowa
  { public:
7   bazowa() { cout << "konstruktor klasy bazowej" << endl; }
    ~bazowa () {cout << "destruktor klasy bazowej" << endl; }
9 };

11 class pochodna : public bazowa
   { public:
13    pochodna() { cout << "konstruktor klasy pochodnej"
                 << endl; }
15   ~pochodna () {cout << "destruktor klasy pochodnej"
                 << endl; }
17 };
```

```
19 int main()
   { bazowa * wsk = new pochodna ;
21   cout << "przydzielony adres = " << wsk << endl;
     delete wsk;
23
      getche();
25    return 0;
   }
```

W wyniku uruchomienia tego programu mamy komunikat:

```
konstruktor klasy bazowej
konstruktor klasy pochodnej
przydzielony adres 10050348
destruktor klasy bazowej
```

W trakcie wykonywania programu, wywołania konstruktorów wykonały się prawidłowo, pamięć dynamiczna została przydzielona. Natomiast po wykonaniu instrukcji delete wsk otrzymaliśmy komunikat, że wywołany został destruktor klasy bazowej (chociaż można by było się spodziewać wywołania konstruktora klasy pochodnej). Dzieje się tak, ponieważ nie ma żadnej wskazówki, który destruktor ma być wywołany. Taka sytuacja jest niepożądana ponieważ prowadzi do tzw. wycieku pamięci. W kolejnym programie pokazujemy metodę, dzięki której operacja zwolnienia zasobów przydzielonych obiektowi klasy pochodnej przebiegnie prawidłowo. Osiągniemy ten efekt dzięki użyciu destruktora wirtualnego.

Listing 8.9. wirtualny desktruktor w hierarchii klas – poprawne użycie

```
   #include <iostream>
 2 #include <conio.h>
   using namespace std;
 4
   class bazowa
 6 { public:
     bazowa() { cout << "konstruktor klasy bazowej" << endl; }
 8   virtual ~bazowa () {cout << "destruktor klasy bazowej"
                                << endl; }
10 };

12 class pochodna : public bazowa
   { public:
14    pochodna() { cout << "konstruktor klasy pochodnej"
                        << endl; }
16   ~pochodna () {cout << "destruktor klasy pochodnej"
                        << endl; }
18 };
```

```
20 int main()
   { bazowa * wsk = new pochodna ;
22   cout << "przydzielony adres = " << wsk << endl;
     delete wsk;
24    getche();
      return 0;
26 }
```

W wyniku uruchomienia tego programu mamy komunikat:

```
konstruktor klasy bazowej
konstruktor klasy pochodnej
przydzielony adres 10050348
destruktor klasy pochodnej
destruktor klasy bazowej
```

Analizując program widzimy, że działanie destruktora jest zgodne z naszym oczekiwaniem. Ten wynik otrzymaliśmy przy pomocy wirtualnego destruktora zdefiniowanego w klasie bazowej:

```
virtual ~bazowa () {cout << "destruktor klasy bazowej"
                         << endl; }
```

W klasie bazowej przy pomocy słowa kluczowego virtual zdefiniowany został destruktor. W klasie bazowej nastąpiła redefinicja destruktora. Wirtualny destruktor jest wywołany dla wskaźnika obiektu, mamy do czynienia z wiązaniem dynamicznym. Wskaźnik wsk wskazuje na obiekt klasy pochodna, dlatego zlecenie delete wsk wywoła destruktor tej klasy. W dalszej kolejności zgodnie z ogólnymi zasadami zostanie wywołany destruktor klasy bazowa. Cały program działa poprawnie.

8.5. Funkcje abstrakcyjne

W praktycznych zastosowaniach dziedziczenia, występują często przypadki, gdy w klasie bazowej nie wiemy jak użytecznie zdefiniować funkcję wirtualną, natomiast doskonale wiemy ją zrealizować ją w klasie pochodnej lub też zachodzi przypadek, że pragmatycznie jest nie podawać definicji funkcji wirtualnej w klasie bazowej. W języku C++ możemy w takich przypadkach wykorzystać koncepcję funkcji abstrakcyjnych.

Funkcja abstrakcyjna to taka, której nie zdefiniowano w klasie bazowej oraz została zainicjowana zerem. Formalnie deklaracja funkcji abstrakcyjnej ma postać:

```
virtual typ_zwracany nazwa_funkcji() = 0 ;
```

Czasami taka funkcja jest nazywana funkcją czysto wirtualną. Klasa zawierająca funkcję abstrakcyjną nosi nazwę klasy abstrakcyjnej. Nie można utworzyć obiektu na podstawie klasy abstrakcyjnej, można tworzyć obiekty przy pomocy klas pochodnych. Próba realizacji obiektu klasy abstrakcyjnej prowadzi do błędu składni. Gdy funkcja wirtualna zostanie utworzona jako funkcja abstrakcyjna, w klasach pochodnych koniecznie należy utworzyć jej nowe definicje. Jeżeli pominiemy definicję funkcji abstrakcyjnej w klasach pochodnych kompilator zgłosi błąd.

Zastosowanie funkcji abstrakcyjnych zilustrujemy prostym przykładem.

Listing 8.10. abstrakcyjna klasa bazowa z wirtualną funkcją składową

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;
  class baza
5 { public:
     virtual int wynik() = 0; // czysta funkcja wirtualna
7 };

9 class dana : public baza
  { int x;
11   public:
     dana (int xx) { x = xx ; }
13   int wynik() { return x ; }
  };
15
  class suma : public baza
17 {  baza *a, *b;
     public :
19     suma (baza *aa, baza *bb) { a = aa; b = bb ; }
       int wynik () { return a->wynik() + b->wynik() ; }
21 };

23 int main()
  { dana d1(5);
25  dana d2(10);
    suma s(&d1, &d2);
27  cout << " + " << d2.wynik() << " = "
         << s.wynik() << endl;
29  cout << "suma = " << s.wynik()  << endl;

31  getche();
    return 0;
33 }
```

Po uruchomieniu programu otrzymujemy następujący komunikat:

```
5 + 10 = 15
suma = 15
```

W programie wykorzystano abstrakcyjną klasę bazową o nazwie baza. W klasie bazowej baza mamy abstrakcyjną funkcję wirtualnę o nazwie wynik(). W klasach pochodnych – dana i suma, abstrakcyjna funkcja wynik() jest redefiniowana. Funkcja wynik() wywołana na rzecz dowolnego obiektu klasy pochodnej zwraca wartość wyrażenia zgodnie z definicją w danej klasie potomnej. Bardziej rozbudowany przykład użycia klas abstrakcyjnych pokażemy na kolejnym przykładzie.

Listing 8.11. abstrakcyjna klasa bazowa; wirtualne funkcje składowe

```
#include <iostream>
#include <string>
using namespace std;

class zwierz           // abstrakcyjna klasa bazowa
{ public:
   zwierz() { };
   zwierz (string iimie) { imie = iimie; }
    virtual void typ()  = 0;
    virtual void glos() = 0;
    virtual void opis() = 0;
    string imie;
};

class pies : public zwierz
{ public:
    pies ( string iimie ) : zwierz ( iimie ) {  };
   private:
    void typ() { cout << "pies " ; }
    void glos() { cout << " szczeka " ; }
    void opis() { cout << " nie lubi kota " ; }
};

class kot : public zwierz
{ public:
    kot ( string iimie) : zwierz ( iimie ) {  };
   private:
    void typ() { cout << " kot " ; }
    void glos() { cout << " miauczy " ; }
    void opis() { cout << " nie lubi myszy " ; }
};


```

```
36 void info (zwierz *tab[ ], int ile)
   { for (int i = 0; i < ile; i++)
38     { tab[i]->typ();
         cout << tab[i]->imie;
40       tab[i]->glos();
         tab[i]->opis();
42       cout << endl;
       }
44 };

46 int main()
   { zwierz *tab[ ] =  { new pies ("Burek"),
48                           new pies ("Fafik"),
                             new kot  ("Pucek")
50                          };
     info(tab,3);
52   cin.get();
     return 0;
54 }
```

Wynikiem uruchomienia programu jest komunikat:

```
pies Burek szczeka  nie lubi kota
pies Fafik   szczeka  nie lubi kota
kot Pucek miauczy  nie lubi myszy
```

W naszym programie mamy abstrakcyjną klasę bazową o nazwie zwierz. Zawiera ona trzy czysto wirtualne funkcje:

```
virtual void typ()  = 0;
virtual void glos() = 0;
virtual void opis() = 0;
```

Taki projekt jest rozsądny, ponieważ pozwala nam w klasach pochodnych napisać żądaną wersję. W naszym przypadku metoda glos() ma inną implementację w klasie pochodnej kot, a inną w klasie pochodnej pies. Podobnie jest z innymi funkcjami wirtualnymi. Wszystkie informacje o naszych zwierzętach otrzymujemy przy pomocy zwykłej funkcji info(). Argumentem tej funkcji jest tablica wskaźników do zwierz. Funkcja info() wywołuje metody typ(), glos() i opis(). Dzięki polimorfizmowi, wywoływane są właściwe metody. Należy zwrócić uwagę, że w tym programie możemy dołączyć nową klasę pochodną, zbudowana tak samo jak klasy pies i kot, a żadna inna zmiana programu nie będzie potrzebna – funkcja info() bez kłopotu obsłuży nową klasę pochodną. Klasy abstrakcyjne i funkcje wirtualne mają wiele zastosowań praktycznych – wiele bibliotek jest implementowanych z wykorzystaniem tych technik.

Możemy stworzyć mały program do obsługi funkcji matematycznych.

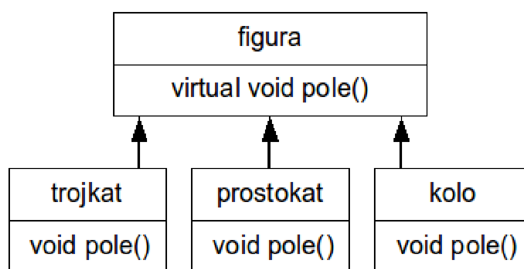
Listing 8.12. abstrakcyjna klasa bazowa; wirtualne funkcje składowe

```
#include <iostream>
#include <string>
#include <math>
using namespace std;

const double PI = 2.0 * asin(1.0) ;

class oblicz
{ protected:
      double x;        //wartosc argumentu
      double y;        //obliczona wartosc funkcji
    public:
      oblicz (double z) { x = z ; }
      double getwynik() { return y ; }
      double getx() { return x ; }
      virtual void obliczfun() = 0;
                   //czysta funkcja wirtualna
};

class fsin : public oblicz
{ public:
    fsin(double xx) : oblicz( xx)  {  }
    void obliczfun() { y = sin(x) ; }
};

class fcos : public oblicz
{ public:
    fcos(double xx) : oblicz( xx)  {  }
    void obliczfun() { y = cos(x) ; }
};

int main()
{ oblicz *wsk;                    //wskaznik klasy bazowej
  fsin  fs(PI/4.0);                   //argument funkcji sin
  wsk = &fs;
  cout << "dla x= " << wsk->getx();
  wsk->obliczfun() ;
  cout << "   sin(x)= " << wsk->getwynik() <<endl;

  fcos  fc(PI/4.0);               //argument funkcji cos
  wsk = &fc;
  cout << "dla x= " << wsk->getx();
  wsk->obliczfun() ;
  cout << "   cos(x)= " << wsk->getwynik() <<endl;
  cin.get();
  return 0;
}
```

Wynikiem tego programu jest komunikat:

```
dla x= 0.785398    sin(x)= 0.707107
dla x= 0.785398    cos(x)= 0.707107
```

W pokazanym przykładzie zrealizowana została metoda „jeden interfejs, wiele metod”. W programie mamy kolekcję funkcji trygonometrycznych, dla danych kątów obliczamy wartości funkcji sinus i cosinus. W klasie bazowej znajduje się funkcja abstrakcyjna obliczfun(), która musi być zdefiniowana w każdej klasie pochodnej. Wynik obliczeń będzie zależał od wybranej wersji klasy pochodnej. W programie mamy dwie klasy pochodne: fsin (do obliczania wartości sinusa) oraz fcos (do obliczania wartości cosinusa). W każdej z tych klas mamy nadpisanie funkcji abstrakcyjnej obliczfun(). Widzimy wyraźnie zaletę stosowania klas abstrakcyjnych. W każdej chwili możemy dopisać nową klasę (np. do obliczania funkcji tangensa), w programie nic się nie zmieni.



Rysunek 8.1. Hierarchia klas z funkcjami wirtualnymi

W kolejnym programie zilustrujemy użycie funkcji wirtualnych w praktycznym przykładzie. Opracujemy program obliczający pola płaskich figur: koło, trójkąt i prostokąt. Zbudujemy odpowiednią hierarchię klas. W klasie bazowej umieścimy funkcję wirtualną realizującą obliczanie powierzchni figur. Odpowiednie funkcje wirtualne, realizujące konkretne obliczenia zdefiniowane będą w klasach pochodnych. Klasa bazowa ma postać:

```
class figura
{   protected:
  double a, b;
  public:
  void ustaw (double aa = 0.0, double bb = 0.0)
    {a = aa;    b = bb;
    }
  virtual void pole() {  }
};
```

W klasie bazowej występuje funkcja pole(). Jest to ogólna funkcja, metoda void pole() nie musi być tu definiowana, ponieważ należałoby podać szczegółowy wzór na obliczanie powierzchni konkretnej figury. W programie tworzymy także trzy dodatkowe klasy pochodne: trojkat, prostokat i kolo. W tych klasach umieszczamy konkretne definicje funkcji pole().

We wzorach na obliczanie pola powierzchni figury musimy podać odpowiednie dane. Dla trójkąta musi podać wartość podstawy i wysokość trójkąta, dla prostokąta musimy podać długości boków a dla koła musimy podać wartość promienia. Należy zauważyć, że tworząc obiekt kolo, przekazujemy jeden argument, w tej sytuacji funkcja ustaw() musi zawierać ustawienia wartości początkowych. Metoda ustaw() ma postać :

```
void ustaw (double aa = 0.0, double bb = 0.0)
{a = aa;    b = bb;
}
```

Gdyby metoda ta była napisana w postaci (brak wartości początkowych):

```
void ustaw (double aa , double bb )
{a = aa;    b = bb;
}
```

w momencie tworzenia obiektu:

```
figura *wsk;
kolo k;
wsk = &k;
wsk-> ustaw(1.0);
wsk-> pole();
```

pojawi się komunikat:

```
[C++ Error] Unit1.cpp(50): E2193 Too few parameters
in call to 'figura::ustaw(double,double)'
```

i program nie skompiluje się. Inna możliwa poprawna definicja dla metody z drugim domyślnym parametrem ma postać:

```
void ustaw (double aa, double bb = 0.0)
{a = aa;    b = bb;
}
```

Poniżej pokazujemy wydruk programu.

Listing 8.13. abstrakcyjna klasa bazowa; wirtualne funkcje składowe

```
1  #include <iostream>
   using namespace std;
```

```
3 class figura
  { protected:
5   double a, b;
    public:
7   void ustaw (double aa = 0.0, double bb = 0.0)
      {a = aa;   b = bb;
9     }
    virtual void pole() {  }
11 };

13 class trojkat : public figura
  { public:
15   void pole()
      {cout << "pole␣trojkata␣=␣" << 0.5*a*b << endl;
17    }
  };
19
  class prostakat : public figura
21 { public:
    void pole()
23    {cout << "pole␣prostokata␣=␣" << a*b << endl;
      }
25 };

27 class kolo : public figura
  { public:
29   void pole()
      {cout << "pole␣kola␣=␣" << 3.1415926*a*a << endl;
31    }
  };
33
  int main()
35 {      figura *wsk;
  trojkat t;
37 prostakat p;
  kolo k;
39
  wsk = &t;
41 wsk-> ustaw(1.0, 2.0);
  wsk-> pole();
43
  wsk = &p;
45 wsk-> ustaw(1.0, 2.0);
  wsk-> pole();
47
  wsk = &k;
49 wsk-> ustaw(1.0);
  wsk-> pole();
51   cin.get();     return 0;
  }
```

Po uruchomieniu tego programu mamy następujący komunikat:

```
pole trojkata = 1
pole prostokata = 2
pole kola = 3.14159
```

Zwracamy uwagę, że w funkcji main() tworzymy wskaźnik do obiektu figura (klasa bazowa) oraz tworzymy zmienne klas pochodnych trojkat, prostokąt i kolo :

```
figura *wsk;
trojkat t;
prostakat p;
kolo k;
```

Wywołania metod dla naszych obiektów (klasy trojkat, prostokąt i kolo) metodą wskaźnika klasy bazowej jest preferowaną techniką (wydajne podejście).

```
wsk = &t;
wsk-> ustaw(1.0, 2.0);
wsk-> pole();
```

Rozdział 9

Klasy wirtualne i zagnieżdżone

9.1. Wstęp **204**
9.2. Klasy wirtualne **204**
9.3. Klasy zagnieżdżone **212**

9.1. Wstęp

Ważną cechą języka C++ jest możliwość stosowania wirtualnych klas bazowych. Konieczność stosowania tego mechanizmu wynika z możliwości pojawienia się niejednoznaczności gdy korzystamy w skomplikowany sposób z wielu powiązanych hierarchicznie klas bazowych. Inną cechą języka jest możliwość definiowania klas zagnieżdżonych, to znaczy klas definiowanych jedna w drugiej. W praktyce technika klas zagnieżdżonych jest stosunkowo rzadko stosowana.

9.2. Klasy wirtualne

W języku C++ mamy możliwość realizacji tzw. dziedziczenia wielobazowego (wielokrotnego), tzn. mamy do czynienia z sytuacją gdy klasa pochodna ma wiele klas bazowych. Wielu zawodowych programistów uważa, że dziedziczenie wielokrotne jest zbyt skomplikowane i najczęściej zbędne w programowaniu obiektowym. Autor cenionych podręczników o języku C++ Nicolai Josuttis wręcz namawia do unikania dziedziczenia (nawet prostego), sugerując rozwiązania typu kompozycji (złożenia) a używania dziedziczenia tylko w sytuacjach, gdy inna implementacja nie jest możliwa. Te wnioski odnoszą się z jeszcze większą siłą do dziedziczenia wielokrotnego, tym bardziej, że ten typ dziedziczenia może prowadzić do wielu konfliktów (np. konflikt nazw). Definiowanie klasy pochodnej w przypadku dziedziczenia wielokrotnego jest proste. W definicji należy te klasy po prostu wyliczyć, podając specyfikator dostępu:

```
class pochodna : public bazowa_1, public bazowa_2
{
    //instrukcje
};
```

W powyższym przykładzie mamy zdefiniowaną klasę pochodną (o nazwie pochodna), która ma dwie klasy bazowe (bazowa_1 oraz bazowa_2). Obiekt klasy pochodna będzie miał następujące właściwości:
— będzie wywoływał wszystkie publiczne funkcje składowe klasy bazowa_1 i bazowa_2
— będzie zawierał wszystkie pola klasy bazowa_1 i bazowa_2 podczas tworzenia nowego obiektu klasy pochodna, automatycznie jest wywołany konstruktor domyślny klas bazowa_1 oraz bazowa_2,
— zniszczenie obiektu klasy pochodna automatycznie wywoła destruktory klas bazowych.

W kolejnym przykładzie zilustrujemy zasady wykorzystania dziedziczenia wielokrotnego. Tworzymy dwie klasy bazowe o nazwach rower i motor

oraz klasę pochodną o nazwie motorower. Użycie klasy pochodnej dziedziczącej wielokrotnie niczym nie różni się od użycia obiektu klasy pochodnej prostej. Praktycznie kod klienta nie musi wiedzieć, że wykorzystuje obiekt klasy dziedziczącej wielokrotnie.

Listing 9.1. Przykład dziedziczenia wielokrotnego

```
   #include <iostream>
 2 using namespace std;

 4 class rower
   { public:
 6     virtual void pokaz_r()
       {cout << "rower ma dwa kola " << endl;
 8     }
   };
10
   class motor
12 { public:
       virtual void pokaz_m()
14     {cout << "motor ma dwa cylindry " << endl;
        }
16 };

18 class motorower : public rower, public motor
   { public:
20     virtual void pokaz_mr()
       {cout << "mamy motorower " << endl;
22     }
   };
24
   int main()
26 { motorower mr;
     mr.pokaz_r();
28   mr.pokaz_m();
     mr.pokaz_mr();
30   cin.get();
     return 0;
32 }
```

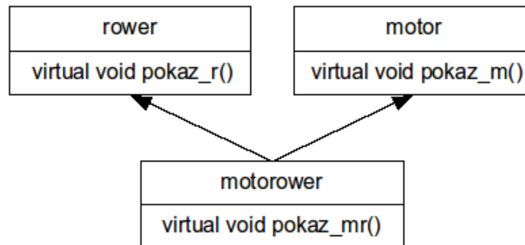
Po uruchomieniu tego programu, mamy następujący wydruk:

```
rower ma dwa kola
motor ma dwa cylindry
mamy motorower
```

Zgodnie z ogólnymi zasadami obiekt klasy motorower obsługuje wszystkie metody publiczne klas motor i rower. Sytuacja komplikuje się, gdy w klasach bazowych wystąpi funkcja składowa o takiej samej nazwie. Założymy,

że w obu klasach bazowych mamy metodę o nazwie waga(). Klasy bazowe nie są ze sobą powiązane, nie dysponują żadną informacją o zdefiniowanych metodach. Dopóki obiekt klasy pochodnej (w naszym przypadku klasy motorower) nie wywoła tej metody , nic specjalnego się nie dzieje. Gdy jednak metoda waga() będzie wywołana, kompilator wygeneruje błąd. Pojawi się informacja, że żądane wywołanie jest niejednoznaczne.

Hierarchia klas wykorzystana w naszym przykładzie pokazana jest na rysunku 9.1.



Rysunek 9.1. Hierarchia klas w dziedziczeniu wielokrotnym

W kolejnym przykładzie zilustrujemy błąd wywołany niejednoznacznością nazw.

Listing 9.2. Dziedziczenie wielokrotne; błąd niejednoznaczności nazw

```
1 #include <iostream>
  using namespace std;
3 class rower
  { public:
5     virtual void pokaz_r()
       {cout << "rower ma dwa kola " << endl;
7      }
      virtual void waga()               //niejednoznaczna nazwa
9      {cout << "waga roweru = 10 kg " << endl;
       }
11 };
  class motor
13 { public:
      virtual void pokaz_m()
15     {cout << "motor ma dwa cylindry " << endl;
       }
17    virtual void waga()               //niejednoznaczna nazwa
       {cout << "waga motoru = 5 kg " << endl;
19     }
  };
21 class motorower : public rower, public motor
  { public:
23    virtual void pokaz_mr()
       {cout << "mamy motorower " << endl;
```

```
25       }
   };
27 int main()
   { motorower mr;
29   mr.pokaz_r();
     mr.pokaz_m();
31   mr.pokaz_mr();
     mr.waga();                  //Blad - niejednoznaczna nazwa !
33   cin.get();
     return 0;
35 }
```

Próba uruchomienia tego programu spowoduje wygenerowanie następującego komunikatu:

```
[C++ Error] Unit1.cpp(36): E2014
Member is ambiguous: 'rower::waga' and 'motor::waga'
```

Problem można rozwiązać na kilka sposobów:
— jawne rzutowanie obiektu
— w klasie pochodnej na nowo definiujemy sporną metodę
— wprowadzenie dwukrotnego dziedziczenia po tej samej klasie
— wprowadzenie klasy wirtualnej

Zmodyfikujemy funkcję main() tak, aby wykorzystać metodę jawnego rzutowania:

```
int main()
{ motorower mr;
  mr.pokaz_r();
  mr.pokaz_m();
  mr.pokaz_mr();
  static_cast<rower>(mr).waga(); // jawne rzutowanie w gore
  mr.motor::waga();          // poprawne wywolanie
  cin.get();
  return 0;
}
```

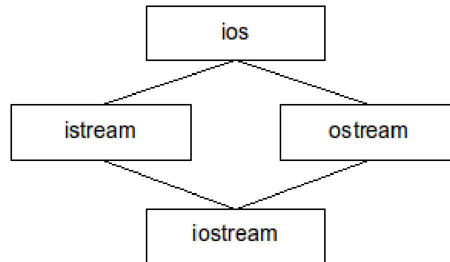
Po uruchomieniu naszego programu otrzymamy całkiem poprawny wynik:

```
rower ma dwa kola
motor ma dwa cylindry
mamy motorowe
waga roweru = 10 kg
waga motoru = 5kg
```

Omówiliśmy zagadnie niejednoznaczności nazw dla przypadku dziedziczenia wielokrotnego na nieco zbyt dydaktycznym przykładzie. W praktyce jednak

dziedziczenie wielokrotne jest często używane. Przykładem może być hierarchia klas wykorzystana w bibliotece standardowej do stworzenia klasy iostream (rys.9.2).

Klasa ios jest klasą bazową dla klas istream i ostream. Klasa iostream dziedziczy po klasach istream i ostream. Aby efektywnie rozwiązać możliwy konflikt nazw stosuje się wirtualne klasy bazowe (podstawowe).



Rysunek 9.2. Diagram klas dla wielokrotnego dziedziczenie z klasy bazowej ios potrzebny do utworzenia klasy pochodnej iostream (tzw. dziedziczenie rombowe lub diamentowe)

Użycie klasy wirtualnej zilustrujemy całkiem praktycznym przykładem. Pokażemy program służący do obliczania powierzchni całkowitej walca. Projekt obliczeniowy składa się z etapów: osobno obliczamy pole podstawy walca (klasa pole_podstawy) oraz pole powierzchni bocznej walca (klasa pole_boczne). Całkowitą powierzchnię walca obliczamy przy pomocy klasy pole_walca. Hierarchia klas pokazana jest na rys. 9.3. W klasach pochodnych mamy metodę o nazwie licz_pole(). Wywołanie tej metody może prowadzić do konfliktu nazw. Problem konfliktu nazw w dziedziczeniu wielokrotnym rozwiązuje się stosując tzw. dziedziczenie wirtualne. Każda klasa podstawowa jest dziedziczona jako wirtualna. Formalnie aby wprowadzić dziedziczenie wirtualne należy nazwę klasy bazowej poprzedzić słowem kluczowym virtual. W naszym przykładzie tego typu deklaracje mają postać:

```
class pole_podstawy : virtual public walec
 { ... };

class pole_boczne : virtual public walec
 { ... };
```

W typ przykładzie klasy pochodne pole_podstawy oraz pole_boczne muszą zadeklarować klasę bazową walec jako wirtualną.

Listing 9.3. Dziedziczenie wielokrotne; klasa wirtualna

```
1 #include <iostream>
  #include <math>
```

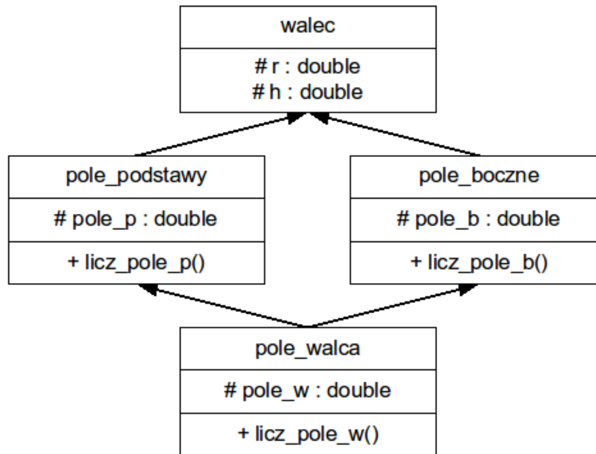
```
3  using namespace std;
   const double PI = 2.0 * asin(1.0) ;
5
   class walec
7  { protected:
       double r, h; //promien podstawy i wysokosc
9    public:
      walec(double rr=1, double hh=1)   { r = rr;   h = hh; }
11    void pokaz() {cout << "r= " << r <<"   h= " << h << endl;}
   };
13
   class pole_podstawy : virtual public walec
15 { protected:
       double pole_p; // pole podstawy walca
17   public:
      pole_podstawy(double rr, double hh) : walec(rr, hh) { }
19    void licz_pole_p()
       { pole_p = 2.0*PI*r*r ; //obliczone pole 2 podstaw
21     }
   };
23
   class pole_boczne : virtual public walec
25 { protected:
       double pole_b; // pole podstawy walca
27   public:
      pole_boczne(double rr, double hh) : walec(rr, hh) { }
29    void licz_pole_b()
       { pole_b = 2.0*PI*r*h ; //obliczone pole sciany bocznej
31     }
   };
33
   class pole_walca : public pole_podstawy, public pole_boczne
35 { protected:
       double pole_w; // pole powierzchni calego walca
37   public:
      pole_walca(double rr, double hh): pole_podstawy(rr,hh),
39 pole_boczne(rr, hh),   walec(rr, hh)   { }
      void licz_pole_w()
41     { licz_pole_p(); //obliczone pole podstawy
         licz_pole_b(); //obliczone pole boczne
43       pole_w = pole_p + pole_b; //obliczone pole walca
         cout << "powierzchnia calkowita walca = "
45             << pole_w << endl;
       }
47 };

49 int main()
   { pole_walca w(1.0, 1.0);
51   w.pokaz();
     w.licz_pole_w();
53   cin.get();
```

```
    return 0;
55 }
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
r = 1      h = 1
powierzchnia calkowita walca = 12.5664
```

Zastosowana w naszym przykładzie hierarchia klas pokazana jest na rys. 9.3.



Rysunek 9.3. Hierarchia klas, dziedziczenie wielokrotne, klasa bazowa walec

Gdyby deklaracje klas pochodnych miały postać:

```
class pole_podstawy : public walec
class pole_boczne :  public walec
```

to pojawia się komunikat kompilatora C++ (Builder 6, Borland):

```
[C++ Error] Unit1.cpp(41): E2312 'walec' is
  not an unambiguous base class of 'pole_walca'
[C++ Error] Unit1.cpp(52): E2014 Member is ambiguous:
  'walec::pokaz' and 'walec::pokaz'
```

gdzie wyspecyfikowane (numeracja w nawiasach okrągłych) linie kodu z błędem to:

```
(40)  pole_walca(double rr, double hh): pole_podstawy(rr,hh),
(41)  pole_boczne(rr, hh),  walec(rr, hh)  { }
(51)  w.pokaz();
```

Metoda pokaz() w tym przypadku nie jest jednoznacznie określona bo może być dziedziczona po klasach pole_podstawy lub pole_boczne.

Korzystanie z wirtualnych klas bazowych jest eleganckim i mało kłopotliwym rozwiązaniem. W pewnych przypadkach można uniknąć niejednoznaczności nazw metody stosując kwantyfikator zakresu (operator czterokropka ::) z nazwą klasy. W kolejnym przykładzie omówimy ten sposób usuwania niejednoznaczności nazw.

Listing 9.4. Dziedziczenie wielokrotne; operator ::

```
#include <iostream>
using namespace std;
// wersja bez metod wirtualnych

class rower
{ public:
    void pokaz_r()
     {cout << "rower ma dwa kola " << endl;
     }
    void waga()                    //niejednoznaczna nazwa
     {cout << "waga roweru = 10 kg " << endl;
     }
};

class motor
{ public:
    void pokaz_m()
     {cout << "motor ma dwa cylindry " << endl;
     }
    void waga()                    //niejednoznaczna nazwa
     {cout << "waga motoru = 5 kg " << endl;
     }
};

class motorower : public rower, public motor
{ public:
    void pokaz_mr()
     {cout << "mamy motorower " << endl;
     }
};

int main()
{ motorower mr;
  mr.pokaz_r();
  mr.pokaz_m();
  mr.pokaz_mr();
  mr.motor::waga();              //usuwamy niejednoznacznosc!
  cin.get();
  return 0;
}
```

Po uruchomieniu tego programu otrzymujemy komunikat:

```
rower ma dwa kola
motor ma dwa cylindry
mamy motorower
waga motoru = 5 kg
```

Zastosowanie operatora zakresu (::) w instrukcji:

```
mr.motor :: waga() ;      //usuwamy niejednoznacznosc!
```

pozwala na uniknięcie wygenerowania błędu związanego z niejednoznacznością nazw. W sposób jawny została wybrana metoda z klasy pochodnej motor. Podobnie możemy wybrać metodę z klasy rower. Należy jednak zdawać sobie sprawę z faktu, że ta metoda nie jest zbyt dobra, ponieważ mamy w klasie motorower więcej niż jedną kopię metody waga(). Rekomenduje się w zasadzie stosowanie metod wirtualnych.

9.3. Klasy zagnieżdżone

W języku C++ istnieje możliwość zdefiniowania klasy wewnątrz funkcji lub innej klasy. Nie są to zbyt popularne techniki programistyczne, gdyż prawdę mówiąc trudno znaleźć realne zastosowania dla tego typu konstrukcji. Na początku omówimy definicje klasy wewnątrz funkcji wykorzystując niewielki program testowy.

Listing 9.5. Klasa lokalna (wewnątrz funkcji)

```
1 #include <iostream>
  using namespace std;
3 void motorower()
  { class cc        // definicja klasy w funkcji
5    { int r, m ;            // waga: motor, rower
       public:
7       void waga_r(int rr) { r = rr ; }
        void waga_m(int mm) { m = mm ; }
9       int get_r() { return r; }
        int get_m() { return m; }
11    } mr;
   mr.waga_r(10);
13 mr.waga_m(5) ;
   cout << "waga motoroweru = "<< mr.get_r()
15       + mr.get_m()<< " kg" << endl;
  }
17
  int main()
19 { motorower();
```

```
      cin.get();
21    return 0;
   }
```

Wynikiem działania tego programu jest następująca informacja:

```
waga motoroweru = 15 kg
```

Ponieważ klasa cc została zadeklarowana wewnątrz funkcji, jest widoczna wyłącznie wewnątrz funkcji motorower(). Podczas tworzenia klasy lokalnej mamy szereg ograniczeń. Wszystkie metody klasy muszą być zdefiniowane wewnątrz deklaracji klasy. Klasy lokalne nie mogą także zawierać zmiennych statycznych. Klasy lokalne nie mogą korzystać ze zmiennych lokalnych funkcji. Język C++ dopuszcza możliwość definiowania klasy wewnątrz innej klasy. Tego typu klasa nosi nazwę klasy zagnieżdżonej (inna nazwa – klasy wewnętrznej, o klasie zewnętrznej mówimy, że jest klasą otaczającą. W kolejnym przykładzie demonstrujemy wykorzystanie klasy zagnieżdżonej, tworząc program do obsługi pacjentów, danymi wejściowymi są imię i nazwisko pacjenta oraz koszt badania i koszt lekarstw. Na końcu drukowany jest raport końcowy podający całkowity koszt wizyty pacjenta. W naszym przykładzie, omawianym poniżej, klasa Koszt jest klasą zagnieżdżoną, a klasą Pacjent jest klasą otaczającą. Klasy zagnieżdżone są dostępne tylko wewnątrz klasy otaczającej.

Listing 9.6. Klasa zagnieżdżona

```
 1 #include <iostream>
   #include <string>
 3 using namespace std;

 5 class Pacjent
   {   static int nr;
 7     string nazwisko;
       string imie;
 9     const int id;
       int badanie, apteka;
11  public:
      class Koszt
13      {  int id;
           int ile;
15       public:
           Koszt (int id, int ile) : id(id), ile(ile) {  }
17         void pokaz();
        };
19  Pacjent (const string &n, const string &i= "" )
      : nazwisko(n), imie(i),
21          id(++nr), badanie(0), apteka(0) { }
```

```
    Pacjent & wplata_b(int w1) {badanie += w1; return *this;}
    Pacjent & wplata_a(int w1) {apteka  += w1; return *this;}
    Koszt * daj_koszt();
    const string &naz() const {return nazwisko;}
    const string &im()   const {return imie ; }
    ~Pacjent() { }
};

int Pacjent :: nr = 0;
Pacjent :: Koszt* Pacjent :: daj_koszt()
 { return new Koszt(id, badanie + apteka);
 }

void Pacjent :: Koszt :: pokaz()
 { cout << "id: " << id << " koszt: " << ile << endl;
 }

int main()
{ Pacjent p1("Kowalski");
  p1.wplata_b(100).wplata_a(50);
  cout << "pacjent " << p1.naz() << "  " << p1.im() <<endl;
  Pacjent :: Koszt * w1 = p1.daj_koszt();
  w1->pokaz();

  Pacjent p2("Kwiatek", "Ewa");
  p2.wplata_b(200).wplata_a(150);
  cout << "pacjent " << p2.naz() <<"  " << p2.im()<< endl;
  Pacjent :: Koszt * w2 = p2.daj_koszt();
  w2->pokaz();

  cin.get();
  return 0;
}
```

Po uruchomieniu tego programu mamy następujący komunikat:

```
pacjent Kowalski
id : 1  koszt :150
pacjent Kwiatek Ewa
id : 2  koszt : 350
```

Klasa Koszt jest zadeklarowana wewnątrz klasy Pacjent. Klasa może być zdefiniowana wewnątrz klasy zewnętrznej lub poza nią. Również metody klasy zagnieżdżonej mogą być definiowane poza klasą otaczającą. W naszym przypadku tak zdefiniowana jest metoda pokaz(). Jeżeli metoda klasy zagnieżdżonej definiowana jest na zewnątrz musimy użyć podwójnej kwalifikacji, w naszym przypadku metoda pokaz() jest w zakresie klasy Koszt, która z kolei jest w zakresie klasy Pacjent:

```
void Pacjent :: Koszt :: pokaz()
 { cout << "id: " << id << " koszt: " << ile << endl;
 }
```

Do obsługi imienia i nazwiska pacjenta wykorzystaliśmy klasę string (plik nagłówkowy <string>) , deklaracja zmiennych ma postać:

```
string nazwisko;
string imie;
```

Wykorzystujemy także konstruktor, którego parametrem jest nazwisko i opcjonalnie imię (domyślne imie to pusty łańcuch), inicjujemy nazwisko i imie za pomocą przekazanych parametrów:

```
Pacjent (const string &n, const string &i= "" )
  : nazwisko(n), imie(i),
  id(++nr), badanie(0), apteka(0) { }
```

Zdefiniowane są także dwie metody zwracające nazwisko i imię:

```
const string &naz() const {return nazwisko;}
const string &im()   const {return imie ; }
```

W funkcji main() tworzenie obiektów jest teraz proste:

```
Pacjent p1("Kowalski");
p1.wplata_b(100).wplata_a(50);
cout << "pacjent " << p1.naz() << "  " << p1.im() <<endl;
Pacjent :: Koszt * w1 = p1.daj_koszt();
w1->pokaz();
```

Klasy zagnieżdżone stosowane są rzadko, najczęściej praktycznie klasy zagnieżdżone stosowane są do klas wyjątków (obsługa błędów).

W przypadku klas zagnieżdżonych obowiązują normalne zasady kontroli dostępu. Na zewnątrz dostęp możliwy jest tylko, gdy klasa zagnieżdżona jest publiczna. Gdy klasę zagnieżdżoną zadeklarujemy jako prywatną lub chronioną, dostępna będzie tylko w klasie zewnętrznej.

Rozdział 10

Wskaźniki do klas

10.1. Wstęp **218**
10.2. Wskaźniki klasowe **218**
10.3. Wskaźniki składowych klas **224**

10.1. Wstęp

Zagadnienie stosowania wskaźników w odniesieniu do obiektów klasy jest dość ważne aczkolwiek nieco skomplikowane (jak to zazwyczaj jest z niebanalnym wykorzystaniem wskaźników w języku C++). Obecnie koncentrować będziemy się na zagadnieniach dostępu do danych klasy i funkcji składowych klasy przy pomocy wskaźników. Oczywiście wskaźnikami do obiektów posługujemy się podobnie jak wskaźnikami do zmiennych innych typów.

10.2. Wskaźniki klasowe

W krótkim przykładzie zademonstrujemy jak przy pomocy wskaźnika możemy uzyskać dostęp do obiektu.

Listing 10.1. Wskaźniki; dostęp do obiektu

```
1 #include<iostream>
  using namespace std;
3
  class punkt
5 {  int x;
   public:
7    punkt ( int x=0 )  : x(x)  {   }
     int get()  { return x ; }
9 };

11 int main()
  { punkt p(10), *wp;
13   wp = &p;
     cout <<"skladowa = " << wp->get() << endl;
15     cin.get();
       return 0;
17 }
```

Wynikiem działania tego programu jest komunikat:

```
skladowa = 10
```

W instrukcjach:

```
punkt p(10), *wp;
wp = &p;
```

tworzymy obiekt p, inicjalizujemy składową obiektu p wartością 10, tworzymy zmienną wskaźnikową wp oraz przypisujemy jej adres obiektu p. W kolejnej instrukcji:

```
cout <<"skladowa = " << wp->get() << endl;
```

dysponując wskaźnikiem uzyskujemy dostęp do zmiennej składowej obiektu p korzystając z operatora strzałki (− >). Zgodnie z arytmetyką wskaźnikową możemy zwiększając wskaźnik odwoływać się do kolejnych elementów tego samego typu.

W kolejnym przykładzie stworzymy tablicę obiektów i dzięki wskaźnikowi uzyskamy dostęp do elementów tablicy obiektów.

Listing 10.2. Wskaźniki; dostęp do elementów tablicy obiektów

```
1 #include<iostream>
  using namespace std;
3
  class punkt
5 { int x;
   public:
7   punkt(int x=0) : x(x)  {  }
    int get() { return x ; }
9 };

11 int main()
  { punkt p[2] = {1, 13};
13   punkt *wp;
    wp = p;
15   cout <<"skladowa 1 = " << wp->get() << endl;
    wp++;
17   cout <<"skladowa 2 = " << wp->get() << endl;
      cin.get();
19     return 0;
  }
```

Wynikiem działania programu jest komunikat:

```
skladowa 1 = 1
skladowa 2 = 13
```

W instrukcjach:

```
punkt p[2] = {1, 13};
punkt *wp;
wp = p;
```

kolejno tworzona jest tablica obiektów, deklarowany jest wskaźnik wp, a następnie do tego wskaźnika przypisany jest adres początkowy tablicy p[ ].

W instrukcjach:

```
wp++;
```

```
cout <<"skladowa 2 = " << wp->get() << endl;
```

inkrementacja wskaźnika wp powoduj, że uzyskujemy adres kolejnego obiektu i można wydrukować wartość zmiennej składowej drugiego obiektu. W pokazanym przykładzie klasa miała tylko jedną zmienną składową. Wobec tego inicjalizacja tablicy obiektów była prosta:

```
punkt p[2] = {1, 13};
```

W bardziej złożonych przypadkach do inicjalizacji elementów tablicy można zastosować konstruktor. W pokazanym programie trzeba go wywołać oddzielnie dla każdego elementu tablicy.

Listing 10.3. Wskaźniki; dostęp do elementów tablicy obiektów

```
1 #include<iostream>
  using namespace std;
3
  class punkt
5 { int x, y;
   public:
7   punkt ( int x=0, int y=0) : x(x), y(y)  {  }
    int get_x() { return x ; }
9   int get_y() { return y ; }
  };
11
  int main()
13 { punkt p[2] = { punkt (1, 11), punkt (3, 33) };
                    //inicjalizacja tablicy
15  punkt *wp;
    wp = p;
17  cout <<"obiekt 1, x = " << wp->get_x() << endl;
    cout <<"obiekt 1, y = " << wp->get_y() << endl;
19  wp++;
    cout <<"obiekt 2, x = " << wp->get_x() << endl;
21  cout <<"obiekt 2, y = " << wp->get_y() << endl;
      cin.get();
23    return 0;
  }
```

W wyniku działanie tego programu mamy informacje:

```
obiekt 1, x = 1
obiekt 1, y = 11
obiekt 2, x = 3
obiekt 2, y = 33
```

Poszczególne elementy tablicy inicjalizowane zostały przy pomocy konstruktora klasy punkt :

```
punkt ( int x=0, int y=0) : x(x), y(y)  {  }
        //konstruktor
punkt p[2] = { punkt (1, 11), punkt (3, 33) };
        //inicjalizacja tablicy obiektow
```

Możemy wykorzystywać wskaźniki do tworzenia tablicy obiektów w pamięci swobodnej (dynamiczny przydział pamięci) kolejny przykład ilustruje to zagadnienie.

Listing 10.4. Wskaźniki klasowe; operator new

```
   #include <iostream>
 2 using namespace std;

 4 class punkt
   {  int x, y ;
 6  public:
     void set ( int xx, int yy ) { x = xx; y = yy ; }
 8   void pokaz( ) { cout << "x= "<< x << "  " <<"y= "<< y
                              << endl; }
10  };

12 int main()
   { punkt *wsk = new punkt [3];
14   for (int i=0; i<3; i++)
       { wsk[i].set(i,i+3);
16       wsk[i].pokaz();
       }
18   cin.get();
     return 0;
20 }
```

W wyniku działania tego programu otrzymujemy następujący komunikat:

```
x= 0   y = 3
x= 1   y = 4
x= 2   y = 5
```

W funkcji main() tworzymy dynamicznie tablice trzech obiektów typu punkt. W pętli for następuje nadawanie wartości danym oraz ich wyświetlanie. Wykorzystano metody set() oraz ustaw() klasy punkt.

W języku C++ ogólna zasada, ściśle przestrzegana mówi, że wskaźnik ustalonego typu nie może wskazywać na obiekt innego typu. Istnieje jednak ważny wyjątek od tej reguły – dotyczy on obsługi obiektów klas pochodnych.

Można utworzyć wskaźnik typu klasy bazowej, który bez problemów obsłuży obiekty klasy pochodnej. Odwrotna sytuacja nie jest możliwa (wskaźnik klasy pochodnej nie obsłuży obiektu klasy bazowej, zasadniczo). Pokazany program ilustruje to zagadnienie. W omawianym przykładzie zostaje utworzony wskaźnik klasy bazowej, z jego pomocą uzyskamy dostęp do pola obiektu klasy pochodnej.

Listing 10.5. Wskaźniki klasowe; dziedziczenie

```
1 #include<iostream>
  using namespace std;
3 class punkt_x
  { int x;
5  public:
    punkt_x ( int x=0) : x(x) {   }
7   set_x(int xx) {x = xx;}
    int get_x() { return x ; }
9 };

11 class punkt_y : public punkt_x
  { int y;
13 public:
    punkt_y ( int y=0) : y(y) {   }
15  int get_y() { return y ; }
  };
17 int main()
  { punkt_x *wsk ; // wskaznik klasy bazowej
19  punkt_y py;     // obiekt klasy pochodnej
    wsk = &py;      // wskaznik klasy bazowej wskazuje
21                  // na obiekt klasy pochodnej
    wsk->set_x(10); //ustawienie danej akcesorem
23  cout <<"obiekt klasy bazowej, x = " << wsk->get_x()
         << endl;
25
    *wsk = punkt_x(100); // ustawienia pola konstruktorem
27  cout <<"obiekt klasy bazowej, x = " << wsk->get_x()
         << endl;
29
      cin.get();
31    return 0;
  }
```

W wyniku działania pokazanego programu otrzymamy następujący komunikat:

```
obiekt klasy bazowej, x =10
obiekt klasy bazowej, x =100
```

Próba uzyskania dostępu do danej y klasy pochodnej przy pomocy wskaźnika do klasy bazowej:

```
wsk->get_y()
```

nie powiedzie się. Nie można uzyskać dostępu do danych klasy pochodnej wykorzystując bezpośrednio wskaźnik klasy bazowej. Rozwiązaniem jest wykonanie odpowiedniego rzutowania wskaźnika klasy. Ilustruje to następny program.

Listing 10.6. Wskaźniki klasowe; rzutowanie wskaźnika

```
1 #include<iostream>
  using namespace std;
3 class punkt_x
  { int x;
5  public:
    void set_x(int xx) {x = xx ;}
7   int get_x() { return x ; }
  };
9 class punkt_y :public punkt_x
  { int y;
11 public:
    void set_y(int yy) {y = yy ;}
13  int get_y() { return y ; }
  };
15 int main()
  { punkt_x *wsk ;          //wskaznik klasy bazowej
17  punkt_y py;              //obiekt klasy pochodnej
    wsk = &py;               //ustawienie wskaznika
19
     wsk->set_x(100);
21   cout <<"obiekt klasy bazowej,    x = "
          << wsk->get_x() << endl;
23   ((punkt_y *)wsk)->set_y(200); //metoda klasy pochodnej!
     cout <<"obiekt klasy pochodnej, y = "
25        << ((punkt_y *)wsk)->get_y() << endl;
      cin.get();
27    return 0;
  }
```

Wynikiem działania tego programu jest komunikat:

```
obiekt klasy bazowej,     x =100
obiekt klasy pochodnej,  y =200
```

Mimo ogólnej reguły, można obejść ograniczenie i uzyskać dostęp do wszystkich składowych klasy pochodnej. W tym celu, jak to pokazaliśmy, należy

dokonać odpowiedniego rzutowania wskaźnika klasy bazowej na wskaźnik klasy pochodnej:

```
((punkt_y *)wsk)->set_y(200);
cout <<"obiekt klasy pochodnej, y = "
     << ((punkt_y *)wsk)->get_y() << endl;
```

10.3. Wskaźniki składowych klas

Czasami zachodzi potrzeba użycia wskaźnika do elementu składowego klasy. W języku C++ mamy specjalny typ wskaźnika noszący nazwę wskaźnika składowych klasy, który wskazuje ogólnie składową klasy a nie składową konkretnego wystąpienia. Należy pamiętać, że obiekty tej samej klasy zawsze mają taki sam rozmiar w pamięci. Z kolei składowe obiektu znajdują się zawsze w takiej samej odległości od początku tego obiektu (tzw. ofset względem początku). Poszczególne składowe są dobrze zlokalizowane w danym obiekcie. Dzięki temu znamy ich pozycje w pamięci. Należy pamiętać, że nie chodzi tu o bezwzględny adres składowej obiektu a o jego przesuniecie względem początku obiektu. Gdy znamy adres obiektu to na podstawie tego przesunięcia wewnątrz obiektu, kompilator jest w stanie wyliczyć adres bezwzględny. Wskaźniki składowych klas nie są zwykłymi wskaźnikami, inne są ich definicje i sposób posługiwania się nimi.

Listing 10.7. Wskaźniki składowych klas

```
1  #include <iostream>
   using namespace std;
3
   class punkt
5  {public :
       punkt (int xx, int yy)   { x = xx; y = yy;}
7      int kwadrat ( ) { return x*x + y*y ; }
       int x, y;
9  };

11 int  punkt ::* wskd;                    // wskaznik danej
   int (punkt ::* wskf) () ;               // wskaznik metody
13
   int main()
15 { punkt p1(3,4);                        // tworzenie obiektu
     wskd = &punkt::x ;                   // pobranie ofsetu x
17     cout << "x= " << p1.*wskd << endl;
     wskd = &punkt::y;                    // pobranie ofsetu y
19     cout << "y= " << p1.*wskd<< endl;
     wskf = &punkt::kwadrat;      // pobranie ofsetu metody
21      cout << "kwadrat= " << (p1.*wskf) () << endl;
```

```
23   cin.get();
     return 0;
25 }
```

Wydruk z programu ma postać:

```
x= 3
y= 4
kwadrat = 25
```

W programie tworzone są dwa wskaźniki składowych:

```
int  punkt ::*wskd;                    // wskaznik danej
int (punkt ::*wskf) () ;               // wskaznik metody
```

Zwracamy uwagę na składnię deklaracji wskaźników składowych klas – są one inne niż zwykłych wskaźników. Mamy specjalna konwencje tworzenia wskaźników składowych klasowych w języku C++. Deklaracja zmiennej składowej klasy ma postać:

```
typ_zm nazwa_klasy ::* nazwa_wskaznika
```

gdzie typ_zm oznacza typ składowej w klasie nazwa_klasy. Wartością zmiennej nazwa_wskaznika będzie przesunięcie (ofset) publicznej składowej w obiekcie klasy nazwa_klasy. W deklaracji obowiązkowo występuje specyfikator zakresu :: (w pokazanej definicji mamy zapis nazwa_klasy ::).

W pokazanym programie mamy w klasie punkt dwie zmienne składowe: int x i int y. Definicja wskaźnika i przypisania (inicjowanie wskaźników adresami elementów składowych) mogą mieć postać:

```
int  punkt ::* wskd;                 // wskaznik danej
wskd = &punkt :: x ;                 // pobranie ofsetu x
wskd = &punkt :: y;                  // pobranie ofsetu y
```

W tych przypisaniach symbol & nie oznacza tak jak w typowym wskaźniku operatora pobranie bezwzględnego adresu. W przypadku wskaźników składowych klas pobierane jest względne przesuniecie składowej x lub y względem początkowego obiektu klasy punkt. Nie można wskaźników składowych klas inkrementować ani wykonywać innych operacji arytmetycznych (operacja wskd++ jest nieprawomocna).

Zadeklarowany wskaźnik nie ma większego znaczenie jeżeli nie zostanie utworzony konkretny obiekt klasy. W naszym przykładzie tworzymy obiekt o nazwie p1, przy pomocy konstruktora polom x i y przypisujemy odpowiednio wartości 3 i 4.

```
punkt p1(3,4);                    // tworzenie obiektu
```

W tym momencie możemy już odwołać się do konkretnej składowej obiektu przy pomocy operatora .* , tak jak to mamy w naszym przykładzie:

```
p1.*wskd
cout << "x= " << p1.*wskd << endl;
```

Trochę bardziej skomplikowana jest deklaracja wskaźników składowych klasy dla metod (funkcji składowych klasy). Przypominamy, że metody nie są zawarte fizycznie w obiektach istnieje tylko jedna wersja metody. Podczas wywoływania metody, jest do niej przekazywany wskaźnik do wywołującego obiektu (tzn. do obiektu, który wywołuje konkretną funkcję), jest to wskaźnik this. Dla funkcji składowej klasy mamy następującą postać deklaracji wskaźnika:

```
typ_m  (nazwa_klasy ::* nazwa_wskaznika ) (argumenty)
```

gdzie typ_m jest typem zwracanym przez metodę klasy nazwa_klasy, nazwa_wskaznika jest wskaźnikiem do funkcji składowej klasy, argumenty oznaczają listę parametrów przekazywanych do funkcji. W deklaracji obowiązkowo występuje specyfikator zakresu :: , nawiasy są konieczne do poprawnego przyporządkowanie operatora ::* . W pokazanym programie mamy w klasie punkt metodę o nazwie kwadrat(). Definicja wskaźnika i przypisania (inicjowanie wskaźnika metody) mogą mieć postać:

```
int (punkt ::* wskf) ( ) ;      // wskaznik metody
punkt p1 (3, 4);                // tworzenie obiektu
wskf = &punkt :: kwadrat;       // pobranie ofsetu metody
```

Odwołanie się do metody, korzystając z operatora .* może mieć postać:

```
cout << "kwadrat= " << (p1 .* wskf) ( ) << endl;
```

Często zamiast nazwy obiektu, możemy utworzyć wskaźnik do obiektu. Aby odwołać się do składowej obiektu lub metody musimy posłużyć się innym operatorem, operatorem $- > *$. W kolejnym przykładzie ilustrujemy to zagadnienie.

Listing 10.8. Wskaźniki składowych klasy; wskaźnik do obiektu

```
1 #include <iostream>
  using namespace std;
3
  class punkt
5 {public :
```

```
      punkt (int x=0, int y=0) :x(x), y(y) { }
7     int kwadrat ( ) { return x*x + y*y ; }
      int x, y;
9 };

11 int main()
  { int  punkt ::* wskx;      // wskaznik danej  x
13   int  punkt ::* wsky;      // wskaznik danej  y
     int  (punkt ::* wskf) () ;// wskaznik metody
15   wskx = &punkt::x;         // pobranie ofsetu x
     wsky = &punkt::y;         // pobranie ofsetu y
17   wskf = &punkt::kwadrat;   // pobranie ofestu kwadrat()
     punkt p1(3,4);            // tworzenie obiektu, inicjacja
19   punkt *w1 = &p1;          // wskaznik do obiektu, inicjacja

21    cout << "obiekt, x=" << p1.*wskx << endl;
      cout << "wskaznik do obiektu, y=" << w1->*wsky << endl;
23    cout << "wskaznik do obiektu, kwadrat = "
           << (w1->*wskf) () << endl;
25
     cin.get();
27   return 0;
  }
```

Wynik działania programu ma postać :

```
obiekt, x = 3
wskaznik do obiektu, y = 4
wskaznik do obiektu, kwadrat = 25
```

W pokazanym programie, w1 jest nazwą wskaźnika do obiektu klasy punkt. Inicjalizacja wskaźnika (przypisanie mu adresu) oraz dostęp do składowej y i metody kwadrat() maja postać:

```
punkt *w1 = &p1; // wskaznik do obiektu, inicjacja
cout << "wskaznik do obiektu, y=" << w1->*wsky << endl;
cout << "wskaznik do obiektu, kwadrat = "
     << (w1->*wskf) () << endl;
```

Pokazano również odwołanie się do składowej klasy przy pomocy nazwy obiektu i wskaźnika do zmiennej składowej klasy:

```
cout << "obiekt, x=" << p1.*wskx << endl;
```

Użyteczności wskaźników do składowych klasy nie jest zbyt duża. Argumentuje się, że czasami warto tej techniki użyć do konstruowania różnego rodzaju opcji wyboru (np. menu). W kolejnym przykładzie demonstrujemy program w którym wykorzystano tablice wskaźników składowych klasy.

Listing 10.9. Wskaźniki składowych klasy; tablice wskaźników

```
1 #include <iostream>
  using namespace std;
3 class punkt
  { public:
5     int x, y, dx, dy;
      punkt (int x=0, int y=0, int dx=0, int dy=0)
7            : x(x), y(y), dx(dx), dy(dy) {  }
      int kwadrat ( )        { return x*x + y*y ; }
9     int przesun_x ( )    { x = x+dx; return x ; }
      int przesun_y ( )    { y = y+dy; return y ; }
11 };
  int main()
13 { int  punkt :: *p[4];   //tablica wskaznikow do zmiennych
    int (punkt :: *m[3]) ( ); //tablica wskaznikow do metod
15  p[0] = &punkt :: x;        //pobranie ofsetow danych
    p[1] = &punkt :: y;
17  p[2] = &punkt :: dx;
    p[3] = &punkt :: dy;
19  m[0] = &punkt :: kwadrat; //pobranie ofsetow metod
    m[1] = &punkt :: przesun_x;
21  m[2] = &punkt :: przesun_y;
    punkt ob( 1,2, 1, 1);      //utworzenie obiektu
23  cout <<"x = " << ob.*p[0] << "  dx= " << ob.*p[2]
         << endl;
25  cout <<"y = " << ob.*p[1] << "  dy= " << ob.*p[3]
         << endl;
27  cout <<"d stare = " << (ob.*m[0]) () << endl;
    cout <<"x + dx =  " << (ob.*m[1]) () << endl;
29  cout <<"y + dy =  " << (ob.*m[2]) () << endl;
    cout <<"d nowe = " << (ob.*m[0]) () << endl;
31 cin.get();
   return 0;
33 }
```

Po uruchomieniu tego programu mamy następujący komunikat:

```
x = 1     dx = 1
y =1      dy = 1
d stare = 5
x + dx = 2
y + dy = 3
d nowe = 13
```

Pokazany wynik działania programu jest poprawny i widać, że wykorzystanie wskaźników do składowych klas, nie powinno sprawiać kłopotów.

Rozdział 11

Techniki obsługi błędów

11.1. Wstęp **230**
11.2. Funkcje walidacyjne **230**
11.3. Graniczne wartości – plik limits.h **234**
11.4. Narzędzie assert() i funkcja abort() **237**
11.5. Przechwytywanie wyjątków **239**

11.1. Wstęp

W zasadzie jest bardzo trudno uniknąć błędów w kodzie programu. Nie istnieją ugruntowane zasady jak pisać bezbłędne programy, mamy w podręcznikach lepsze lub gorsze zalecenia jak unikać złośliwych błędów. Z drugiej strony dążymy do pisania bezbłędnych kodów – język C++ i jego biblioteki wspierają obsługę błędów. Wszystkie błędy możemy podzielić na dwie kategorie:
— błędy czasu kompilacji
— błędy czasu wykonania

Przyjmuje się, że mniej groźne są błędy powstające w czasie kompilacji – większość kompilatorów wyposażona jest w doskonałe (i bardzo rozbudowane) techniki diagnostyczne, tak, że przy pierwszym wykryciu błędu kompilator przerwie proces kompilacji i wyświetli odpowiedni komunikat o błędzie. Oczywiście w wielu przypadkach komunikat taki będzie mało czytelny, czasem bez sensu i nie zawsze wskazany wiersz kodu zawiera błąd. Pocieszającym jest fakt, że błąd kompilacji zawsze się ujawni. Po stosunkowo krótkim czasie, programista nabywa doświadczenia i usuwanie błędów czasu kompilacji nie jest zbyt skomplikowane. Istnieją popularne techniki (np. wyłączanie linijki kodu w którym kompilator sygnalizuje błąd przy pomocy znaków komentarza, czy też umieszczanie w programie testowych wydruków częściowych wyników) pomagający usunąć takie błędy. Wiele platform programistycznych wyposażonych jest zestaw narzędzi diagnostycznych – są to tzw. debugery.

Błędy czasu wykonania są w wielu przypadkach trudne do zidentyfikowania i usunięcia. Przyczyną tego jest fakt, że takie błędy nie generują komunikatów błędu oraz że mogą pojawiać się bardzo rzadko. Klasyczne przyczyny wystąpienia błędów to próba otworzenia nieistniejącego pliku, żądanie większej ilości pamięci niż możemy otrzymać, próba dzielenia przez zero, napotkanie złej wartości danych wejściowych, itp. W programach możemy stosować różne techniki zapobiegający wystąpieniom możliwych błędów, klasyczną techniką jest stosowanie funkcji walidacyjnych, sprawdzanie poprawności danych wejściowych czy wykorzystanie wyspecjalizowanych funkcji takich jak assert(). Odrębną techniką obsługi błędów w języku C++ (podobnie jak w języku Java czy Pyton) jest przechwytywanie wyjątków (ang. exception).

11.2. Funkcje walidacyjne

Jeżeli wiemy jakie warunki muszą być spełnione aby wykonać zaplanowane operacje, możemy uniknąć błędów sprawdzając te warunki. Bar-

dzo często znamy ograniczenia na wartości argumentów przekazywanych do funkcji. Klasycznym przykładem jest obliczanie pierwiastków kwadratowych z liczb. Funkcja sqrt() z biblioteki math (lub math.h) akceptuje tylko argumenty większe lub równe zero. Sprawdzamy ten warunek i gdy argument jest ujemny wywołujemy funkcję biblioteczną exit(1), która formalnie umieszczona jest w pliku nagłówkowym stdlib.h .

```
if (x < 0) {
        cout << " ujemna wartosc x" << endl;
        getche();
        exit(1);
       }
```

Funkcja exit() jest bardzo użyteczna – elegancko kończy działanie programu. Opróżnia ona wszystkie bufory wyjściowe, zamyka wszystkie otwarte strumienie i usuwa pliki tymczasowe. Następnie zwraca ona kontrolę do systemu operacyjnego. Przykład zastosowania tej funkcji pokazany jest poniżej.

Listing 11.1. Sprawdzanie parametrów funkcji

```
1 #include <iostream>
  #include <conio.h>
3 #include <math>

5 int main()
  {
7   using namespace std;
    float x;
9   cout << " Podaj x: ";
    cin >> x;
11     if (x < 0) {
        cout << " ujemna wartosc x" << endl;
13      getche();
        exit(1);
15     }
    cout << " Pierwiastek z " << x << " = " << sqrt(x);
17  getche();
  return 0;
19 }
```

Działanie programu jest następujące:
Podaj x: 9 Pierwiastek z 9 = 3

```
Podaj x: -4
Ujemna wartosc
```

Pokazana technika zapobiegania błędom jest stosunkowo prosta i daje dobre rezultaty. Gdy występują bardziej skomplikowane warunki możemy zastoso-

wać odrębną funkcję, która sprawdzi te warunki. Taka funkcja walidacyjna testuje pewne założenia i w przypadku ich poprawności zwraca kod, który umożliwia kontynuowanie programu. Zastosujemy funkcję walidacyjną do sprawdzenia czy nie żądamy obliczenia pierwiastka kwadratowego z ujemnego argumentu. Funkcja walidacyjna ma postać:

```
int walid (float xx)
{
    if (xx>0) return 1;
    else
                return 0;
 }
```

Wykorzystanie funkcji walidacyjne w programie pokazane jest poniżej.

Listing 11.2. Sprawdzanie parametrów funkcji; funkcja walidacyjna

```
   #include <iostream>
 2 #include <conio.h>
   #include <math>
 4
   int walid(float);
 6
   int main()
 8 { using namespace std;
     float x;
10   cout << " Podaj x: ";
     cin >> x;
12     if (walid(x)) cout << "pierwiastek z "
                 << x << " = " << sqrt(x)<< endl;
14     else
         cout << "argument ujemny " << endl;
16   getche();
   return 0;
18 }

20 int walid (float xx)
   { if (xx>0) return 1;
22   else
                return 0;
24 }
```

Kolejną techniką zapobiegającą błędom jest próba zastąpienia błędnie wprowadzonego argumentu innym poprawnym. Jest to technika dość ryzykowna, ale czasem stosowana. W kolejnym przykładzie obliczania pierwiastka kwadratowego , możemy zapobiec potencjalnym błędom sprawdzając wartość liczby, w przypadku gdy jest ona mniejsza niż zero zastępujemy ją liczbą dodatnią. W tym celu możemy posłużyć się prostą funkcją :

```
float abs_war(float xx)
  { if (xx < 0)    xx = -xx;
    return xx;
  }
```

Jeżeli pomyłkowo wprowadzimy ujemną wartość będzie ona zamieniona na wartość dodatnią, do funkcji wyliczającej pierwiastek kwadratowy przekazana jest zawsze wartość dodatnia.

W programie pokazanym niżej pokazujemy takie rozwiązanie, przy czym stosujemy własną funkcję obliczającą pierwiastek kwadratowy przy pomocy iteracyjnej metody Newtona – Raphsona. Jest to zadziwiająco dokładna i szybka metoda. Algorytm obliczania wartości pierwiastka kwadratowego z liczby x z dokładnością err (w naszym przykładzie err = 0.001) tą metoda ma postać pokazaną na wydruku 11.3.

Algorytm Newtona-Raphsona obliczania pierwiastka kwadratowego:
1. Ustal wartość początkową n_0 (najczęściej wybieramy $n_0 = 1$)
2. Sprawdzamy wyrażenie $|n_i^2 - x| < err$, jeśli prawdziwe idź do 4
3. Ustalamy kolejną wartość $n_i = 0.5 * (xx/n_{i-1} + n_{i-1})$, idź do 2
4. Koniec obliczeń, przybliżona wartość pierwiastka z x jest równa n_i

Listing 11.3. Sprawdzanie parametrów funkcji; automatyczna zmiana

```
   #include <iostream>
 2 #include <conio.h>
   using namespace std;
 4 float abs_war (float);
   float pierwiastek(float);
 6 int main()
   {
 8   float x;
     cout << "Podaj x: ";
10   cin >> x;
     x = abs_war(x);
12   cout << " Pierwiastek z " << x << "  = "
          << pierwiastek(x)<< endl;;
14   getche();
     return 0;
16 }
   float abs_war(float xx)
18   {  if (xx < 0)    xx = -xx;
        return xx;
20   }
   float pierwiastek(float xx)
22   {  float err = 0.001;
        float n = 1.0;
24      int j = 0;
        while (abs_war(n*n - xx) >= err)
26           {    n = 0.5 * (xx/n  + n);
```

```
                j++; }
28      cout << " Liczba iteracji = " << j << endl;
        return n;
30 }
```

Wykonanie pokazanego programu ma postać:

```
Podaj x : 25
Liczba iteracji = 5
Pierwiastek z 25 = 5.00002
```

W przypadku wprowadzenia ujemnej wartości wykonanie programu ma postać:

```
Podaj x : -100
Liczba iteracji = 7
Pierwiastek z 100 = 10
```

Jak widzimy, prosta funkcja testująca skutecznie zapobiega błędom.

11.3. Graniczne wartości – plik limits.h

W języku C i C++ nie są sprawdzane zakresy wartości typów danych (zakresy zależą od ilości bajtów na jakiej kodowany jest konkretny typ w konkretnym kompilatorze). W pliku bibliotecznym limits.h oraz climits pokazane są graniczne wartości (maksymalne i minimalne) rozmaitych typów danych jakie wykorzystywane są przez kompilator w danym systemie operacyjnym. Np. stała INT_MAX reprezentuje maksymalną wartość dla zmiennych całkowitych typu int, a np. UINT_MAX dla typu unsigned int. Te stałe wykorzystujemy w programach do testowania, czy obliczana wartość mieści się w poprawnym zakresie. Zagadnie to ilustrujemy programem który wylicza silnię. Silnia nieujemnej liczby n, oznaczana jako n! jest iloczynem

$$n * (n-1) * (n-2) * \dots\dots\dots * 1 \tag{11.1}$$

Przyjmuje się, że 0! =1 oraz 1! = 1. Silnia jest bardzo szybko rosnącą funkcją, już dla niewielkich wartości n może dawać bardzo dużą wartość. W tej sytuacji obliczenia silni będziemy przeprowadzać na typie unsigned long. Specyfikacja języka C++ wymaga, aby zmienna typu unsigned long była przechowywana przynajmniej na 32 bitach (4 bajty), co daje zakres od 0 do 4294967295. Przekroczenie zakresu nie jest sygnalizowane w żaden sposób, programista musi sam zadbać aby nie wyjść poza zakres w czasie obliczeń. Tworzymy w programie funkcję:

```
bool w_zakresie(int nn)
{
    unsigned long max = ULONG_MAX;
    for (int i = nn; i >+1; --i)
            max /=i;
     if (max < 1)
            return false;
     else
            return true;
}
```

która przyjmie wartość true gdy n jest w zakresie oraz false gdy n jest zbyt dużą liczbą. Stała ULONG_MAX (plik nagłówkowy limits.h) jest równa maksymalnej liczbie całkowitej typu unsigned long jaki może być użyty w danym kompilatorze i systemie operacyjnym. Dostępne stałe to:

Nazwa stałej	Typ zmiennej
UCHAR_MAX	unsigned char
USHRT_MAX	unsigned short
UINT_MAX	unsigned integer
ULONG_MAX	unsigned long

Listing 11.4. Sprawdzanie granicznych wartości; plik limits.h

```
   #include <iostream>
 2 #include <conio.h>
   #include <limits.h>
 4
   using namespace std;
 6
   bool w_zakresie(int);
 8 unsigned long silnia (int);

10 int main()
   {  int i;
12    cout << " Podaj liczbe : ";
      cin >> i;
14    if (w_zakresie(i))
         cout << " Silnia z liczby " << i << " = "
16        << silnia(i) << endl;
      else
18       cout << " Poza zakresem" << endl;
    getche();
20  return 0;
   }
22
   unsigned long silnia ( int n)
24 {  if ( n <= 1)
```

```
      return 1;
26    else
      return n*silnia(n - 1);
28 }

30 bool w_zakresie(int nn)
   {  unsigned long max = ULONG_MAX;
32     for (int i = nn; i >=1; --i)
           max /=i;
34     if (max < 1)
           return false;
36     else
           return true;
38 }
```

Po uruchomieniu programu mamy wynik:

```
Podaj liczbe : 10
Silnia z liczby 10 =  36288800

Podaj liczbe : 12
Silnia z liczby 12 =  479001600

Podaj liczbe : 13
Poza zakresem
```

W następnym przykładzie pokażemy technikę zapobiegającą wprowadzeniu błędnych danych korzystając z komunikatu jaki wysyła obiekt cin. Jeżeli użytkownik nie poda wartości liczbowej lub wartość ujemną wtedy wyrażenie

```
(!(cin >> x ) || x < 0)
```

z deklaracją int x przyjmie wartość true, obiekt strumienia cin zostanie opróżniony i przygotowany do kolejnej próby pobrania poprawnych danych.

Listing 11.5. Sprawdzanie stanu obiektu cin

```
1 #include <iostream>
  #include <math>
3
  int main()
5 {
    using namespace std;
7   int x ;
    cout << " Podaj dodatnia liczbe calkowita : ";
9   while (!(cin >> x ) || x < 0)
    {
```

```
11      cin.clear();
        cin.ignore(100, '\n');
13      cout << " Podaj dodatnia liczbe calkowita : ";
      }
15    cin.ignore(100, '\n');

17    cout << " Pierwiastek z " << x << " = " << sqrt(x);
      cin.get();
19  return 0;
  }
```

Działanie programu może mieć postać:

```
Podaj dodatnia liczba calkowita : -4
Podaj dodatnia liczba calkowita : k
Podaj dodatnia liczba calkowita : $
Podaj dodatnia liczba calkowita : 9
Pierwiastek z 9 = 3
```

Dopóki nie zostanie wprowadzona poprawnie dodatnia liczba całkowita program nie przystąpi do obliczania pierwiastka kwadratowego (wprowadzenie znaku powoduje błąd ponieważ cin oczekuje liczby całkowitej typu int, jeśli cin ma wartość false, oznacza to, że wystąpił błąd)

```
while (!(cin >> x ) || x < 0)
```

Aby przywrócić cin do stanu początkowego, tak aby ten obiekt mógł ponownie przyjąć dane wejściowe możemy użyć funkcji clear(), która wymazuje znacznik błędu (ale nie czyści bufora).

```
cin.clear();
```

W celu odrzucenia wejścia (kasowanie zawartości bufora) możemy użyć funkcji ignore().

```
cin.ignore(100, '\n');
```

W powyższym przykładzie będzie odrzuconych 100 kolejno wczytanych znaków, chyba, że wcześniej będzie napotkany znak nowego wiersza `\n`.

11.4. Narzędzie assert() i funkcja abort()

Zagadnienie zapobiegania błędom jest ważne, dlatego też programiści korzystający z języka C i C++ mają całkiem dobre narzędzie diagnostyczne jakim jest makro o nazwie assert(). Makro assert() pobiera wyrażenie całko-

wite i je testuje. Jeżeli wyrażenie jest fałszywe, wysyłany jest komunikat błędu do standardowego wyjścia dla błędów (stderr) i wywoływana jest funkcja abort(), która kończy program. W języku C assert() znajduje się w pliku nagłówkowym assert.h , natomiast abort() w pliku stdlib.h. Po przerwaniu programu przez assert() wyświetlony jest warunek, który nie został spełniony, nazwa pliku i numer wiersza. W języku C++ aby stosować asercję należy zastosować dyrektywę `#include <cassert>` lub `#include<assert.h>`.

Listing 11.6. Wykorzystanie asercji do obsługi błędów

```
1 #include <iostream>
  #include <math>
3 #include <conio.h>
  #include <assert.h>
5
  int main()
7 {
    using namespace std;
9   int x,y ;
    float m;
11  cout << " Podaj x : " ;
    cin >> x;
13  cout << " Podaj y : " ;
    cin >> y;
15  m = x*x - y*y;
    assert(m >= 0);
17  cout << " Pierwiastek z " << m << " = " << sqrt(m);
    getche();
19 return 0;
  }
```

Jeżeli w programie nie będzie instrukcji:

```
assert(m >= 0);
```

to przykładowe wykonanie programu może mieć postać:

```
Podaj x : 1
Podaj y : 4
sqrt: DOMAIN error
Pierwiastek z -15 = nan
```

Jeżeli w programie umieścimy asercje to program z tymi danym nie uruchomi się, nastąpi przerwanie programu i powrót do środowiska programistycznego.

Obsługę błędu bardzo podobną do wywołania makra assert() jest wywołanie funkcji abort(). Zmodyfikujemy poprzedni program, rezygnując z assert() na rzecz zastosowania funkcji abort().

Listing 11.7. Wykorzystanie funkcji abort() do obsługi błędów

```
#include <iostream>
#include <math>
#include <conio.h>
#include <cstdlib>
int main()
{
  using namespace std;
  int x,y ;
  float m;
  cout << " Podaj x : " ;
  cin >> x;
  cout << " Podaj y : " ;
  cin >> y;
  m = x*x - y*y;
  if ( m <= 0 )
    { cout << " argument w sqrt ujemny";
      abort();
    }
  cout << " Pierwiastek z " << m << " = " << sqrt(m);
  getche();
 return 0;
}
```

Zaleca się stosowanie makra assert(), ponieważ w wyniku jego działania otrzymujemy więcej informacji na temat błędu (np. numer wiersza, w którym błąd wystąpił).

11.5. Przechwytywanie wyjątków

Język C++ posiada bardzo wydajny mechanizm obsługi błędów, nazywany obsługą wyjątków. Obsługa wyjątków pozwala programiście zarządzać błędami wykonania w sposób zorganizowany, dający poczucie bezpieczeństwa. Konstrukcje związane z obsługą wyjątków w języku C++ związana jest z trzema słowami kluczowymi:
— try - próbuj, testuj, sprawdzaj
— throw - wyrzuć, zgłoś, sygnalizuj, prześlij
— catch - złap, przechwyć.

W bloku try umieszczamy instrukcje w których mogą wystąpić wyjątki (błędy). Jeżeli wystąpi błąd jest on zgłaszany przy pomocy instrukcji throw. Wyjątki (błędy) są wychwytywane w bloku catch, który znajduje się za blokiem try. W bloku catch wyjątek jest odpowiednio przetwarzany – nastąpi próba naprawienia błędu, zignorowania lub zakończenia działania programu. Jeśli zostanie zgłoszony wyjątek, który nie jest obsługiwany w bloku catch, wykonywanie programu jest przerwane, w takim przypadku

wywoływana jest funkcja biblioteczna terminate(), która z kolei wywołuje funkcje abort(). Dla celów dydaktycznych pokażemy prosty przykład obsługi wyjątków. Rozważymy program monitorujący bezpieczeństwo uczestników spływu kajakowego – każdy kajakarz powinien posiadać kamizelkę ratunkową, co oznacza , że ilość kamizelek nie może być mniejsza niż ilość kajakarzy. Pokazany poniżej program monitoruje tą sytuację. Jeżeli uruchamiamy nasz program na platformie Borlad Builder 6 należy w zakładce Tools/Debbuger options. . . wyłączyć przycisk „ Integrated debugging”.

Listing 11.8. Proste obsługiwanie wyjątków

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

using namespace std;
int main()
{ int kam, kaj;
  try        //blok try
    {
    cout << "Liczba kamizelek : ";
    cin >> kam;
    cout << "Liczba kajakarzy : ";
    cin >> kaj;
         if (kaj>kam) throw (kaj - kam);//instrukcja throw
    cout << "Zaczynamy splyw kajakowy";
    }
  catch (int t)                //blok catch
     {  cout << "Brakuje " << t
         <<" kamizelek, splyw odwolany";
     }
   getche();
   return 0;
}
```

Należy pamiętać, że demonstrowane tu programy z przykładami obsługi wyjątków są bardzo proste i normalnie nikt nie zastosował by takich mechanizmów obsługi błędów. Przy spełnionych warunkach bezpieczeństwa działanie programu może mieć postać:

```
Liczba kamizelek 5
Liczba kajakarzy  5
Zaczynamy splyw kajakowy
```

W przypadku gdy mamy więcej kajakarzy niż kamizelek, działanie programu może mieć postać:

```
Liczba kamizelek 5
Liczba kajakarzy  7
```

```
Brakuje 2 kamizelek, splyw odwolany
```

Przyjrzyjmy się pokazanemu programowi. Widzimy blok try :

```
try    //blok try
  {
   cout << "Liczba kamizelek : ";
   cin >> kam;
   cout << "Liczba kajakarzy : ";
   cin >> kaj;
       if (kaj>kam) throw (kaj - kam);//instrukcja throw
   cout << "Zaczynamy splyw kajakowy";
  }
```

W tym bloku znajduje się fragment kodu z obsługą wyjątków. Wewnątrz tego bloku jest instrukcja:

```
if (kaj>kam) throw (kaj - kam); //instrukcja throw
```

Widzimy, że gdy liczba kajakarzy jest większa niż liczba kamizelek, dzieje się coś wyjątkowego, wykonana jest zatem instrukcja throw. Gdy liczba kamizelek jest większa niż liczba kajakarzy, nie ma sytuacji wyjątkowej i program wykonywany jest dalej.

Widzimy, że w bloku try umieszczony jest kod programu, w zależności od danych wejściowych możemy otrzymywać różne wyniki, można spodziewać się sytuacji gdy należy podjąć specjalne działania. Jeżeli coś jest wyjątkowe (w naszym przypadku naruszono zasady bezpieczeństwa kajakarzy) to blok wyrzuca (ang. throw) wyjątek, W naszym programie instrukcja throw wyrzuca liczbę całkowitą będąca różnicą między liczbą kajakarzy i kamizelek. Oczywiści wyrzucać można wartości różnych typów (w naszym przypadku jest to wartość typu int).W momencie wyrzucenia wyjątku zatrzymywane jest wykonywanie kodu w bloku try, zaczyna się wykonywanie instrukcji zawartych w bloku catch (ang. przechwycenie). Mówimy, że wyjątek jest obsłużony w bloku catch. W naszym programie blok catch ma postać:

```
catch (int t)              //blok catch
   {  cout << "Brakuje " << t
        <<" kamizelek, splyw odwolany";
   }
```

Parametr t typu int w słowie kluczowym catch nosi nazwę parametru bloku catch. Parametr bloku catch ma określony typ, dzięki czemu wiemy, jaki został wyrzucony wyjątek (można wyrzucać różne wyjątki). Po drugie, ma on nazwę, dzięki czemu w bloku catch możemy na tej wartości wykonać jakieś operacje, co może na przykład prowadzić do usunięcia błędu.

Gdyby liczba kamizelek była większa niż liczba kajakarzy, wyjątek by nie został wyrzucony i wykonane byłyby kolejne instrukcje bloku try. Po wyjściu z bloku try, instrukcje zawarte w bloku catch są ominięte i program wykonuje pozostałe instrukcje. Mamy wniosek - gdy nie zostaje wyrzucony wyjątek, blok catch jest przez program ignorowany. Pamiętajmy, że typ wyjątku musi odpowiadać typowi umieszczonemu w instrukcji catch. Dla instrukcji catch z argumentem float :

```
catch (float t)          //blok catch
  {  cout << "Brakuje " << t
     <<" kamizelek, splyw odwolany";
  }
```

w pokazanym powyżej programie, wyjątek nie będzie obsłużony i program zakończy działania.

Wyjątek może zostać wyrzucony z funkcji, jeżeli wywołanie funkcji jest realizowane wewnątrz bloku try, to taki wyjątek będzie obsłużony. Pokazuje tą sytuację kolejny program.

Listing 11.9. Proste obsługiwanie wyjątków; instrukcja throw w funkcji

```
   #include <iostream>
 2 #include <conio.h>
   using namespace std;
 4 void test(int,int);
   int main()
 6 { int kam, kaj;
     try
 8     { cout << "liczba kamizelek : ";
        cin >> kam;
10      cout << "liczba kajakarzy : ";
        cin >> kaj;
12      test(kam, kaj);
        cout << "zaczynamy splyw kajakowy";
14     }
     catch (int t)
16     {cout << "brakuje " << t
        <<" kamizelek, splyw odwolany";
18     }
       getche();
20     return 0;
   }
22 void test(int x,int y)
   {  int r;
24    r = x - y;
      if ( r < 0) throw -r;
26 }
```

Blok try może być umieszczony wewnątrz funkcji.

Listing 11.10. Proste obsługiwanie wyjątków; blok try i catch w funkcji

```
   #include <iostream>
 2 #include <conio.h>

 4 using namespace std;
   void test(int,int);
 6 int main()
   { int kam, kaj;
 8     cout << "liczba␣kamizelek␣:␣";
       cin >> kam;
10     cout << "liczba␣kajakarzy␣:␣";
       cin >> kaj;
12    test(kam, kaj);
      getche();
14    return 0;
   }
16 void test(int x,int y)
   { try
18     {   if ( x < y) throw (y - x);
           cout << "zaczynamy␣splyw␣kajakowy";
20     }
    catch (int t)
22     { cout << "brakuje␣" << t
          <<"␣kamizelek,␣splyw␣odwolany";
24     }
   }
```

Blok try może wyrzucać więcej niż jeden wyjątków, wobec tego mamy możliwość umieszczenia wielu bloków catch, należy jednak pamiętać, że każdy taki blok catch musi przechwytywać wyjątek innego typu.

Instrukcje catch sprawdzane są w takiej kolejności w jakiej umieszczone są w programie. Gdy wykonywany jest blok try może być wyrzucony tylko jeden wyjątek, ale tych wyjątków może być wiele (za każdym razem może to być wyjątek innego typu). Blok catch wychwytuje tylko jeden wyjątek określonego typu. Bloki catch umieszczone są jeden pod drugim, tak aby była możliwość dopasowania żądanego typu. Jeżeli wyrzucony zostanie wyjątek, którego typ nie jest obsługiwany przez żaden blok catch – mamy problem. Program jest zatrzymywany awaryjnie. Należy unikać takich sytuacji.

Gdy w programie przechwytujemy wiele wyjątków, jest istotna kolejność w jakiej umieszczamy bloki catch. Sprawdzanie są kolejne bloki catch, gdy program natrafi na odpowiadający typ, który pasuje do typu wyrzuconego wyjątku, wykonywany jest ten blok. Widzimy, że działanie programu jest zależne od kolejności wyrzucanych wyjątków i kolejności poszczególnych bloków catch. W naszym przykładzie instrukcje catch przechwytują

typ całkowity (int) oraz tablicę znaków (char *). Pierwszy jest wyrzucany wyjątek typu int, blok catch z argumentem int także jest umieszczony jako pierwszy.

Listing 11.11. Proste obsługiwanie wyjątków; wiele bloków catch

```
1  #include <iostream>
   #include <conio.h>
3
   using namespace std;
5  int main()
   { int i;
7    char zn;
       cout << "Podaj liczbe nieujemna : ";
9      cin >> i;
       cout << "Podaj znak, q konczy program: ";
11     cin >> zn;
     try
13     {   if ( i < 0) throw (i);
           else
15             cout << "Wprowadzona liczba : " << i << endl;
           if ((zn == 'q') || (zn == 'Q')) throw "Koniec";
17         else
              cout << "Wprowadzony znak : " << zn << endl;
19     }
    catch (int t)
21     { cout << "liczba " << t <<" jest ujemna";
       }
23   catch (char *napis)
       { cout << napis << " programu " ;
25     }

27   getche();
     return 0;
29 }
```

W wyniku działania programu mamy następujące wyniki:

```
Podaj liczbe nieujemna : 5
Podaj znak, q konczy program: w
Wprowadzona liczba : 5
Wprowadzony znak : w
```

Zmieniając dane wejsciowe możemy mieć następujący wydruk:

```
Podaj liczbe nieujemna : -3
Podaj znak, q konczy program: t
Liczba -3 jest ujemna
```

Inny zestaw danych wejsciowych da wynik:

```
Podaj liczbe nieujemna : 5
Podaj znak, q konczy program: q
Wprowadzona liczba : 5
Koniec programu
```

Na koniec wprowadzamy jeszcze inny zestaw danych:

```
Podaj liczbe nieujemna : -5
Podaj znak, q konczy program: q
Liczba -5 jest ujemna
```

Analizując wyniki widzimy, że w zależności od wprowadzonych danych mamy różne reakcje programu (otrzymujemy inne wyniki). W języku C++ mamy bardzo wygodną konstrukcję do przechwytywania wyjątków. Jest to blok catch, który przechwytuje wszystkie wyjątki. Dzieje się tak dlatego, że nie ma podanego typu wyjątków. Tego typu blok catch ma postać:

```
catch (...) {
//instrukcje
}
```

Nawias, który posiada wewnątrz trzy kropki, oznacza w takim bloku dowolny typ wyjątku. W następnym programie demonstrujemy działanie uniwersalnego bloku catch.

Listing 11.12. Proste obsługiwanie wyjątków; uniwersalny blok catch

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;
  int main()
5 { int i;
    char zn;
7     cout << "Podaj liczbe : ";
      cin >> i;
9     cout << "Podaj znak : ";
      cin >> zn;
11    try
      {   if ( i < 0) throw i;
13        if ((zn == 'q') || (zn == 'Q')) throw 'q';
      }
15    catch (...)
      { cout << "Mamy wyjatek" << endl;
17    }
      cout << "Koniec" << endl;
19    getche();
      return 0;
21 }
```

Możemy mieć następujący wynik działania tego programu:

```
Podaj liczbe : 5
Podaj znak : w
Koniec
```

Dla innego zestawu danych wejściowych mamy wynik:

```
Podaj liczbe : -3
Podaj znak : t
Mamy wyjatek
Koniec
```

Przechwycenie drugiego wyjątku pokazane jest dla nowego zestawu danych:

```
Podaj liczbe : 7
Podaj znak : q
Mamy wyjatek
Koniec
```

Analizując ten program, widzimy, że pojedynczy blok catch obsługuje wszystkie wyjątki. Gdy to możliwe zaleca się stosowanie tego uniwersalnego bloku catch. Ma to jeszcze inną istotną zaletę – dzięki przechwyceniu wszystkich wyjątków nie mamy awaryjnego zatrzymania programu, gdy wystąpi nieobsługiwany błąd. Jak już pokazaliśmy, funkcja może wyrzucać wyjątki. W wielu przypadkach należy ograniczyć typy wyjątków jakie może wyrzucać funkcja, można nawet zakazać funkcji wyrzucania jakichkolwiek wyjątków. Aby wprowadzić ten mechanizm musimy zmodyfikować prototyp (oraz definicję) naszej funkcji:

zwracany_typ nazwa_funkcji (lista_argumentów) throw (lista_typów)

W takiej sytuacji funkcja może wyrzucić tylko takie wyjątki, których typy znajdują się na liście typów. Jeżeli na liście typów nie ma żadnego typu (lista jest pusta) to funkcja nie może zwrócić żadnego wyjątku. Należy pamiętać, że gdy funkcja będzie próbowała wyrzucić wyjątek nieobsługiwanego typu natychmiast zostanie wywołana funkcja biblioteczne unexpected(), która wywoła funkcje abort(), która kończy wykonywanie programu.

Listing 11.13. Ograniczenia typów wyjątków zgłaszanych przez funkcję

```
  #include <iostream>
2 #include <conio.h>
  void test (int, char) throw (int, char, float);
4 using namespace std;
  int main()
6 { int i;
    char zn;
```

```
8       cout << "Podaj liczbe : ";
        cin >> i;
10      cout << "Podaj znak : ";
        cin >> zn;
12    try
        { test(i,zn);          }
14    catch (int n)
        { cout << "Mamy wyjatek typu int n = " << n <<endl; }
16      catch (char zz)
        { cout << "Mamy wyjatek typu char c = " << zz << endl; }
18      catch (float x)
        { cout << "Mamy wyjatek typu float x = " << x << endl; }
20    getche();
      return 0;
22           }
   void test (int i , char c) throw (int, char, float)
24 {    float x = 0.5*i;
           if  (i == 0) throw i;
26         if ( c == 'q') throw 'Q';
           if (x < 1) throw x;
28 }
```

Po uruchomieniu tego programu mamy wynik:

```
Podaj liczbe : 1
Podaj znak : s
Mamy wyjatek typu float x = 0.5
```

Z innym zestawem danych otrzymamy:

```
Podaj liczbe : 0
Podaj znak : r
Mamy wyjatek typu int n = 0
```

Widzimy, że pokazana funkcja test() poprawnie zgłasza wyjątki obsługiwane przez bloki catch w funkcji main(). Ponieważ wyjątek może być dowolnego typu, powszechną praktyka jest definiowanie klas, których obiekty będą zawierać dokładne informacje, które są przeznaczone do wyrzucenia a następnie do obsłużenia przez instrukcję catch. W takich przypadkach mówimy, że tworzymy klasy wyjątków.

W kolejnym przykładzie pokazujemy proste wykorzystanie klasy do obsługi wyjątków. W programie obliczamy średnią harmoniczną (zmienna sh):

$$sh = 2.0 * a * b/(a + b); \tag{11.2}$$

Gdy liczba $a = -b$, otrzymamy zero w mianowniku i próbę dzielenia przez zero, naszym zadaniem jest obsłużenie tego błędu.

Listing 11.14. Wykorzystanie klasy wyjątków

```
#include <iostream>
#include <conio.h>
using namespace std;

class zly_wynik
{
  private:
  int x, y;
  public:
   zly_wynik(){ };
   zly_wynik(int xx, int yy ) : x(xx), y(yy) {};
   void info();
};

void zly_wynik::info()
  {cout << " x = " << x << "  " << " y = " << y
         << endl;
   cout << " Zle dane x = -y " << endl;
  }

int main()
{ int a,b;
  float sh;
  cout << "Podaj dwie liczby: ";
  while (cin >> a >> b )
  {try
     { if (a == -b)
              throw zly_wynik(a,b);
        sh = 2.0*a*b/(a+b);
        cout << "srednia = " << sh << endl;
        cout << "Podaj kolejne dwie liczby, q konczy, : "
             << endl;
     }
    catch (zly_wynik er)
      { cout <<  "mamy wyjatek" << endl;
        er.info();
        cout << " podaj kolejne liczby.\n";
      }
    }
   getche();
   return 0;
}
```

Po uruchomieniu tego programu mamy testowe wyniki:

```
Podaj dwie liczby: 1
2
Srednia harmoniczna = 1.33333
Podaj kolejne dwie liczby, q konczy, :
```

```
1
-1
Mamy wyjatek
x = 1    y = -1
Zle dane x = -y
Podaj kolejne liczby
q
```

Klasa wyjątku ma postać:

```
class zly_wynik
{ private:
    int x, y;
  public:
   zly_wynik(){ };
   zly_wynik(int xx, int yy ) : x(xx), y(yy) {};
   void info();
};
```

Obiekt zły_wynik jest inicjalizowany w funkcji main() przy pomocy składni:

```
if (a == -b)
throw zly_wynik(a,b);
```

Jest to klasyczne wywołanie konstruktora dwuargumentowego klasy zły_wynik i inicjalizacja obiektu za pomocą przekazanych argumentów.

Wyjątek obsługiwany jest w bloku catch:

```
catch (zly_wynik er)
     { cout <<  "mamy wyjatek" << endl;
       er.info();
       cout << " podaj kolejne liczby.\n";
     }
```

Metoda info() klasy zły_wynik wywołana na rzecz obiektu er dostarczyć może detalicznych informacji o występującym wyjątku.Pokazany poniżej program jest inną modyfikacją programu wyliczającego średnią harmoniczną.

Listing 11.15. Wykorzystanie klasy wyjątków

```
1 #include <iostream>
  #include <string.h>
3 #include <conio.h>
  using namespace std;
5 class Wyjatek
  { public :
7     int x,y;
      char k[80];
9   Wyjatek();
```

```
     Wyjatek (int , int , char *);
11  };
    Wyjatek::Wyjatek()
13               {x=0;   y=0;   *k = 0;    }
    Wyjatek::Wyjatek ( int xx, int yy, char *kk)
15              {strcpy(k,kk);  x = xx;  y = yy;
                }
17 int main()
   { int x,y;
19   float sh;
       cout << "Podaj liczbe x: ";
21     cin >> x;
       cout << "Podaj liczbe y: ";
23     cin >> y;
      try
25      {  if ( x == -y)
            throw Wyjatek(x,y,"Niepoprawne argumenty");
27         sh = 2.0*x*y/(x+y);
           cout << "Srednia harmoniczna z liczb "
29              <<" = " << sh;
        }
31    catch (Wyjatek err)
         { cout << err.k << "  "<< endl;
33         cout << "x = " << err.x << "  y = "
                << err.y << endl;
35         cout << " x = -y" << endl;
         }
37    getche();
      return 0;
39 }
```

Po uruchomieniu programu ze złymi liczbami mamy wynik:

```
Podaj liczbe x: -5
Podaj liczbe y: 5
Niepoprawne argumenty
x = -5    y = 5
x = -y
```

Kolejna modyfikacja pokazuje wykorzystanie klasy wyjątków, które wyrzucane są przez funkcję usługową.

Listing 11.16. Klasy wyjątków; funkcja ze specyfikacją wyjątków

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;
  class zly_wynik
5   {    int x, y;
       public:
```

```
 7          zly_wynik(int xx=0, int yy = 0)
                      : x(xx), y(yy) {}
 9        void info();
  };
11 void zly_wynik:: info()
       {  cout << " x = " << x << "  " << " y = "
13                << y << endl;
          cout << " Zle dane poniewaz x = -y "
15                << endl;
       }
17 double sr_harmon(double, double) throw (zly_wynik);

19 int main()
  { double a,b,sh;
21   cout << "Podaj dwie liczby: ";
     while( cin >> a >>b)
23    {  try
          {
25             sh = sr_harmon(a,b);
               cout << "Srednia = " << sh << endl;
27             cout << " Podaj kolejna pare liczb ,
  w aby skonczyc:";
29           }
        catch (zly_wynik & er)
31           {  er.info();
               cout << " Podaj kolejne liczby.\n";
33           }
        }
35    getche();
      return 0;
37 }
  double sr_harmon(double x, double y) throw (zly_wynik)
39 { if (x == -y)
          throw zly_wynik(x,y);
41    return 2.0*x*y/(x+y);
  }
```

Po uruchomieniu programu możemy mieć komunikat:

```
Podaj dwie liczby: 3
-3
  x = 3       y = -3
  Zle dane poniewaz x = -y
  Podaj kolejne liczby,
q
```

W naszym przykładzie klasa wyjątków ma postać:

```
class zly_wynik
  {     int x, y;
```

```
      public:
         zly_wynik(int xx=0, int yy = 0) : x(xx), y(yy) {}
        void info();
};
```

Obiekt zły_wynik jest inicjalizowany za pomocą argumentów przekazanych do funkcji sr_harmon(). W deklaracji tej funkcji usługowej mamy dodaną specyfikację wyjątków dzięki której określamy jakie wyjątki ta funkcja będzie wyrzucać:

```
double sr_harmon(double, double) throw (zly_wynik);
```

Funkcja sr_harmon() wyrzuca wyjątki przy pomocy instrukcji:

```
if (x == -y)
throw zly_wynik(x,y);
```

W tej instrukcji mamy klasyczne wywołanie konstruktora klasy zły_wynik i inicjalizacje obiektu za pomocą przekazanych argumentów. Blok try ma postać:

```
  try
        {
           sh = sr_harmon(a,b);
           cout << "Srednia = " << sh << endl;
           cout << " Podaj kolejna pare liczb ,
                    w aby skonczyc:";
        }
```

W tym bloku wywoływana jest funkcja usługowa sr_harmon(), która może wyrzucić wyjątek. Wyjątek przechwytuje blok catch:

```
catch (zly_wynik & er)
      {  er.info();
         cout << " Podaj kolejne liczby.\n";
      }
```

W tym bloku wywołana jest funkcja info(), która jest metodą klasy wyjątków zły_wynik. Ponieważ preferowanym sposobem reprezentacji klasy wyjątków jest użycie klasy wyjątków omówimy kolejną prostą aplikację do wykrycia błędu dzielenia przez zero.

Listing 11.17. Klasy wyjątków; prosty przykład

```
  #include <iostream>
2 #include <conio.h>
  using namespace std;
4
```

```
class MianownikZero
{ int x, y;
  public:
   MianownikZero() : x(0),y(0) { }
   MianownikZero(int a, int b) : x(a), y(b) { }
   void komunikat()
    { cout << "proba dzielenia przez zero" << endl;
    }
   void pokaz()
     {cout << "dzielna = " << x << "   dzielnik = "
           << y << endl;
     }
};

double dzielenie( int li, int mi) throw(MianownikZero)
  { if ( mi == 0) throw MianownikZero(li,mi);
    return (double)li/mi;
  }

int main()
{
 int licznik, mianownik;
 double wynik;
 cout << "podaj dwie liczby calkowite : ";
 while (cin >> licznik >> mianownik)
   {try
     { wynik = dzielenie(licznik, mianownik);
       cout << " wynik = " << wynik << endl;
     }
       catch(MianownikZero &er)
          { er.komunikat() ;
            er.pokaz();
          }
    cout << "podaj dwie liczby calkowite, q konczy : ";
   }

  getche();
  return 0;
       }
```

Wynik działania programu jest następujący:

```
podaj dwie liczby calkowite : 3 6
   wynik = 0.5
podaj dwie liczby calkowite, q konczy : 1 3
   wynik = 0.333333
podaj dwie liczby calkowite, q konczy : 1 0
proba dzielenia przez zero
dzielna = 1    dzielnik = 0
podaj dwie liczby calkowite, q konczy : q
```

Klasa o nazwie MianownikZero jest klasą wyjątków, będzie ona wykorzystana do wykrycia błędu dzielenia przez zero:

```
class MianownikZero
{   int x, y;
  public:
   MianownikZero() : x(0), y(0) { }
   MianownikZero(int a, int b) : x(a), y(b) { }
   void komunikat()
      { cout << "proba dzielenia przez zero" << endl;
      }
};
```

W klasie mamy dwie dane prywatne x i y, wiemy także, że gdy chcemy wykonać dzielenie x/y, to y nie może być zerem. W pokazanej klasie mamy dwa konstruktory oraz funkcję komunikat(), która wyświetla ostrzeżenie o wystąpieniu błędu i funkcję pokaz(), która wyświetla wartości dwóch wprowadzonych liczb. W programie występuje funkcja dzielenie():

```
double dzielenie( int li, int mi)  throw(MianownikZero)
  { if ( mi == 0) throw MianownikZero(li,mi);
    return (double)li/mi;
  }
```

Funkcja najpierw sprawdza, czy mianownik jest równy zero. Gdy jest to prawda wyrzuca wyjątek typu MianownikZero, gdy mianownik jest różny od zera wykonuje dzielenie i wynik jest zwracany do programu. W funkcji main() mamy sekcję try – catch:

```
try
       { wynik = dzielenie(licznik, mianownik);
          cout << " wynik = " << wynik << endl;
       }
        catch(MianownikZero &er)
            { er.komunikat() ;
            }
```

W tym bloku następuje wywołanie funkcji dzielenie(). Jak widać z definicji tej funkcji, zwraca ona wyjątek :

```
if ( mi == 0) throw MianownikZero(li,mi);
```

Jest on wykorzystywany w klauzuli catch:

```
catch(MianownikZero &er)
{ er.komunikat() ;
}
```

Argumentem klauzuli catch jest referencja do obiektu wyjątku.

Pokazany program ma dość rozbudowaną klasę wyjątków, możemy ją uprościć, tak jak to pokazano w kolejnym przykładzie.

Listing 11.18. Klasy wyjątków; prosty przykład

```
1  #include <iostream>
   #include <conio.h>
3  using namespace std;

5  class MianownikZero
   {  char *komunikat;
7    public:
       MianownikZero()
9        : komunikat ("proba dzielenia przez zero") { }
       char *kom()  {return komunikat; }
11 };

13 double dzielenie( int li , int mi)
    { if ( mi == 0) throw MianownikZero();
15     return (double)li/mi;
    }
17
   int main()
19 {
    int licznik , mianownik;
21  double wynik;
    cout << "podaj dwie liczby calkowite : ";
23  while (cin >> licznik >> mianownik)
      {try
25      { wynik = dzielenie(licznik , mianownik);
          cout << " wynik = " << wynik << endl;
27      }
          catch(MianownikZero er)
29          { cout << er.kom() << endl;
            }
31     cout << "podaj dwie liczby calkowite , q konczy : ";
      }
33
     getche();
35   return 0;
   }
```

W naszym przykładzie klasa wyjątków o nazwie MianownikZero ma postać:

```
class MianownikZero
{  char *komunikat;
  public:
    MianownikZero()
```

```
        : komunikat ("proba␣dzielenia␣przez␣zero") { }
      char *kom()  {return komunikat; }
};
```

Klasa zawiera prywatną zmienną napisową, publiczną funkcję składową oraz konstruktor, który wskazuje pole komunikatu z napisem ”proba dzielenia przez zero”. Blok try – catch otacza kod który może zwrócić wyjątek:

```
try
    { wynik = dzielenie(licznik, mianownik);
      cout << "␣wynik␣=␣" << wynik << endl;
    }
      catch(MianownikZero er)
         { cout << er.kom() << endl;
         }
```

Zwróćmy uwagę na fakt, że wyjątek może być zgłoszony w wywoływanej funkcji dzielenie(). Gdy wystąpi próba dzielenia przez zero, funkcja dzielenie() wyrzuci wyjątek, który będzie zgłoszony przez blok try. Ten wyjątek zostaje wychwycony przez blok catch, który określa odpowiedni typ dopasowany do zgłoszonego wyjątku. W naszym programie wyjątek powinien być typu MianownikZero. Gdy instrukcja if w funkcji dzielenie() wykryje, że próbujemy podzielić przez zero, wygeneruje instrukcję throw, określającą nazwę konstruktora dla obiektu wyjątku. Zostanie utworzony wyjątek klasy MianownikZero. Instrukcja catch przechwyci ten obiekt. Zgłoszony obiekt odbierany jest w argumencie określonym w procedurze obsługi catch:

```
catch(MianownikZero er)
{ cout << er.kom() << endl;
}
```

co powoduje wyświetlenia napisu z komunikatem o wystąpieniu próby dzielenia przez zero.

Nic nie stoi na przeszkodzie aby wykorzystać dziedziczenie podczas stosowania wyjątków. Różne klasy wyjątków mogą być klasami pochodnymi wspólnej klasy podstawowej, klasy wyjątków mogą tworzyć hierarchie. Oznacza to, że klauzula catch wychwycić może wskaźnik lub referencję do wszystkich obiektów wyjątków klas pochodnych.

Kolejny przykład demonstrujący wykorzystanie dziedziczenia do obsługi wyjątków wzorowany jest na przykładzie pokazanym w monografii V. Sheterna pt. „C++ inżynieria oprogramowania”. Jak to bywa w klasycznych podręcznikach wykorzystano zadanie obliczania wartości ułamka po wprowadzeniu z klawiatury licznika i mianownika.

Listing 11.19. Klasy wyjątków; dziedziczenie

```
#include <iostream>
#include <limits.h>      //LONG_MAX
#include <conio.h>
using namespace std;

class Info
{      static char *tekst[ ];
   public:
       static char * kom(int n)
          { return tekst[n] ;   }
} ;

char *Info :: tekst[ ] = { " mianownik rowny zero\n ",
                " mianownik ujemny ",
                "\n podaj licznik i nieujemny mianownik : ",
                "\n (podaj q aby zakonczyc)",
                " wartosc ulamka : "
                              } ;

class MianownikZero
{ protected :
         char *kom;
  public :
         MianownikZero(char * wiad) : kom(wiad)  {  }
         void pokaz() { cout << kom; }
};

class MianownikUjemny : public MianownikZero
{      long war;
     public :
       MianownikUjemny (char * wiad, long w)
     : MianownikZero(wiad), war(w)   {  }
        void pokaz() { cout << kom << war << endl;}
};

inline void odwrot( long w, double & wynik)
               throw (MianownikZero, MianownikUjemny)
{ wynik = (w) ? 1.0/w : LONG_MAX;
  if (wynik == LONG_MAX)
        throw MianownikZero( Info :: kom(0) );
  if (w < 0)
        throw MianownikUjemny(Info :: kom(1), w);
}

inline void ulamek (long licznik, long mianownik,
                    double & wartosc)
                throw (MianownikZero, MianownikUjemny)
{ odwrot(mianownik, wartosc);
  wartosc = licznik*wartosc;
}
```

```
51 int main()
   { while (true)
53   { long licznik , mianownik;
       double ul;
55     cout << Info::kom(3) << Info::kom(2);
       if ( (cin >> licznik >> mianownik) == 0 )    break;
57     try
         { ulamek( licznik , mianownik, ul);
59         cout << Info::kom(4) << ul << endl;
         }
61      catch( MianownikUjemny & neg)
         { neg.pokaz();
63       }
        catch( MianownikZero & zer)
65       { zer.pokaz();
         }
67   }
    getche();
69  return 0;
   }
```

Przykładowe działanie programu ma postać:

```
(podaj q aby zakonczyc)
podaj licznik i nieujemny mianownik : 1 0
mianownik rowny zero

(podaj q aby zakonczyc)
podaj licznik i nieujemny mianownik : 1 -1
mianownik ujemny -1

(podaj q aby zakonczyc)
podaj licznik i nieujemny mianownik : 1 3
wartosc ulamka : 0.333333

(podaj q aby zakonczyc)
podaj licznik i nieujemny mianownik : -1 9
wartosc ulamka : -0.111111

(podaj q aby zakonczyc)
podaj licznik i nieujemny mianownik : q
```

W programie do obliczenia wartości ułamka wykorzystujemy dwie funkcje:

```
inline void odwrot( long w, double & wynik)
            throw (MianownikZero , MianownikUjemny)
{ wynik = (w) ? 1.0/w : LONG_MAX;
      if (wynik == LONG_MAX)
              throw MianownikZero( Info :: kom(0) );
    if (w < 0)
```

```
                throw MianownikUjemny(Info :: kom(1), w);
    }

    inline void ulamek (long licznik, long mianownik,
                        double & wartosc)
                        throw (MianownikZero, MianownikUjemny)
    {
     odwrot(mianownik, wartosc);
      wartosc = licznik*wartosc;
    }
```

Funkcja odwrot() zwraca odwrotność argumentu. Przy argumencie różnym od zera, funkcja oblicza jego odwrotność (zmienna wynik). Gdy argumentem jest zero, zmienna wynik przyjmuje wartość LONG_MAX (stała zdefiniowana w pliku limits.h). Wtedy wyrzucany jest wyjątek stwierdzający, że mianownika ma wartość zero.

Gdy argument ma wartość ujemną (z powodów technicznych przy obsłudze ułamków lepiej unikać ujemnego mianownika) generowany jest wyjątek sygnalizujący, że mianownik ma wartość ujemną. Funkcja odwrot() jest wywoływana przez funkcję ulamek(). W tej funkcji następuje mnożenie wartości jej pierwszego argumentu przez wartość obliczoną w funkcji odwrot() (czyli odwrotność mianownika). Podczas wykonywania programu zachodzi konieczność wykorzystania kilku komunikatów (napisów), które są wyświetlane na ekranie. Aby usprawnić ich obsługę, zostały one zgrupowane w klasie Info, i zdefiniowane w postaci prywatnej statycznej tablicy łańcuchów znakowych tekst[ ] :

```
    class Info
    {
       static char *tekst[ ];
       public:
          static char * kom(int n)
             { return tekst[n] ;   }
    } ;

    char *Info :: tekst[ ] = { " mianownik rowny zero\n ",
                 " mianownik ujemny ",
                 "\n podaj licznik i nieujemny mianownik : ",
                 "\n (podaj q aby zakonczyc)",
                " wartosc ulamka : "
                                 } ;
```

Do konkretnego komunikatu odwołujemy się przy pomocy funkcji statycznej o nazwie kom(), której argumentem jest indeks odpowiedniego komunikatu tablicy tekst[ ]. Często podczas projektowania obsługi wyjątków należy ustalić jakie dane powinny zostać wysłane do klasy, której obiekty będą

obsługiwały wystąpienie błędu. Metody takiej klasy powinny dopuszczać dostęp do danych obiektu wewnątrz bloku catch. W naszym przykładzie klasa MianownikZero obsługuje informacje o próbie dzielenia przez zero, klasa MianownikUjemny obsługuje wystąpienie ujemnego mianownika. W naszym przypadku można zaprojektować hierarchię klas. Klasa Mianownik-Zero jest klasą bazową, klasa MianowniUjemny dziedziczy po tej klasie (jest klasą pochodną).

Klasa bazowa ma postać:

```
class MianownikZero
{ protected :
        char *kom;
  public :
        MianownikZero(char * wiad) : kom(wiad)  {  }
        void pokaz() { cout << kom; }
};
```

Klasa MianownikUjemny wywodzi się z klasy MianownikZero:

```
class MianownikUjemny : public MianownikZero
{     long war;
    public :
     MianownikUjemny (char * wiad, long w)
         : MianownikZero(wiad), war(w)  {  }
       void pokaz() { cout << kom << war << endl;}
};
```

W funkcji main() wywoływana jest funkcja ulamek() wewnątrz bloku try:

```
try
  { ulamek( licznik, mianownik, ul);
    cout << Info::kom(4) << ul << endl;
   }
 catch( MianownikUjemny & neg)
   { neg.pokaz();
   }
 catch( MianownikZero & zer)
   { zer.pokaz();
   }
```

Gdy podczas wykonywania funkcji ulamek() wystąpi błąd, wyjątek przechwytywany jest przez kolejne bloki catch.

Obecnie, wyjątki są w zasadzie częścią języka. Plik nagłówkowy exception definiuje klasę exception, która jest klasą bazową dla innych klas wyjątków. Klasa ta posiada funkcję składową what(), która wykonuje operacje na obiektach tej klasy. Wirtualna funkcja składowa what() zwraca ciąg znaków, zależny od implementacji, ten ciąg znaków opisuje wyjątek. Poniższa tabela

pokazuje hierarchię standardowych klas wyjątków. Klasą bazową jest klasa exception.

exception $< -$	bad_alloc
	bad_cast
	bad_typeid
	logic_error
	ios_base::failure
	runtaim_error
	bad_exception

Klas pochodna logic_error jest klasą nadrzędną dla kolejnych klas pochodnych. Kolejna tabela pokazuje standardowe klasy wyjątków dziedziczących z tej klasy.

logic_error $< -$	domain_error
	invalid_argument
	length_error
	out_of_range

Podobnie klasa runtime_error jest klasą nadrzędną dla innych klas pochodnych, hierarchia pokazana jest w kolejnej tabeli.

runtime_error $< -$	range_error
	overflow_error
	underflow_error

Klasy wyjątków zdefiniowane są w różnych plikach nagłówkowych. Informacje o najczęściej używanych klasach pokazuje kolejna tabela.

nr	klasa	plik	Opis
1	bad_alloc	<new>	Wyjątek zgłaszany przez operator new gdy nie udał się przydział pamięci
2	bad_cast	<typeinfo>	Wyjątek zgłaszany przez operator dynamic_cast gdy nie udała się konwersja
3	bad_typeid	<typeinfo>	Wyjątek zgłaszany przez operator typeid, gdy wskaźnik będący jego argumentem jest pusty
4	bad_exception	<exception>	Wyjątek zgłaszany gdy pojawi się nieznany wyjątek
5	ios::failure	<ios>	Wyjątek zgłaszany, gdy zmienił się stan strumienia w niepożądany sposób

Dynamiczny przydział pamięci jest ważną operacją, zależy nam aby się powiodła. Mamy dwa sposoby obsługi błędu przydziału pamięci. Możemy sprawdzić wskaźnik, gdy jest pusty, przydział pamięci nie powiódł się. Do kończenia programu często korzystano z funkcji assert. Gdy wartość zwracana przez wywołanie new wynosi 0, makroinstrukcja assert kończy program. Jest to stosunkowo dobre rozwiązanie, ale nie pozwala na powrót do normalnego stanu. Standard języka C++ określa, że gdy alokacja pamięci tworzona operatorem new nie powiedzie się, zgłaszany jest wyjątek bad_alloc. W kolejnym programie pokazano obsługę błędu przydziału pamięci za pomocą zgłoszenia wyjątku.W pętli for, która zamknięta jest wewnątrz bloku try, chcemy wykonać 100 pętli i przydzielić w każdym przejściu 5000000 wartości typu double. Prędzej czy później może zabraknąć pamięci i wtedy zostanie zgłoszony wyjątek bad_alloc, pętla kończy działanie. Sterowanie programem przekazywane jest to klauzuli catch , która wychwytuje wyjątek i go przetwarza:

```
catch (bad_alloc wyjatek)
   {cout << "obsluga wyjatku : " << wyjatek.what()
         << endl;
   }
```

Metoda what() zwraca komunikat, zależny od wyjątku. W naszym przypadku jest to komunikat:

```
bad alloc exception thrown
```

Listing 11.20. Klasy wyjątków; wywołanie new; zgłoszenie bad_alloc

```
#include <iostream>
#include <conio.h>
#include <new>

using namespace std;

int main()
{double *w[100];
 long licznik = 0;
 try
    { for (int i=0; i<100; i++)
        {w[i] = new double[5000000];
         ++licznik;
        }
    }
  catch (bad_alloc wyjatek)
    {cout << "obsluga wyjatku : " << wyjatek.what() << endl;
    }

    cout <<"udalo sie przydzielic " << licznik -1 <<" blokow"
         << endl;
    int r = sizeof(double);
    cout <<"rozmiar double to : " << r << " bajtow" <<endl;
    cout <<"w sumie przydzielono : "<< (licznik -1)* 5000000 *r
         << " bajtow"<<endl;
 getche();
 return 0;
}
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
obsluga wyjatku : bad alloc exception thrown
udalo sie przydzielic 48 blokow
rozmiar double to : 8 bajtow
w sumie przydzielono: 1920000000 bajtow
```

Wynik wykonania pokazanego programu oczywiście będzie różny na różnych systemach. Zależeć będzie od ilości bajtów użytych do kodowania typu double i ilości dostępnej pamięci (fizycznej pamięci i przestrzeni dyskowej dostępnej dla pamięci wirtualnej).

Rozdział 12

Szablony w C++

12.1. Wstęp **266**
12.2. Przeciążanie funkcji **266**
12.3. Szablony funkcji **276**
12.4. Szablony klas **292**

12.1. Wstęp

W języku C++ istnieje ścisła kontrola zgodności typów. Czasami taka ścisła kontrola typów (która generalnie jest zaletą) powoduje znaczne komplikacje przy tworzeniu wydajnego oprogramowania. Gdy na przykład opracujemy funkcję pobierającą argumenty typu int, to taka funkcja nie będzie poprawnie obsługiwała argumentów typu double. Dla obsługi argumentów innego typu musimy opracować inną funkcję. Język C++ oby ominąć to ograniczenie ściśle związane z kontrolą typów wprowadza szablony, czasami zwane w literaturze wzorcami (ang. template). Przy pomocy kodu jednej funkcji lub jednej klasy możemy obsługiwać dane różnych typów. Możemy opracować pojedynczy szablon (wzorzec) funkcji sortującej tablice z danymi typu int, double, char czy nawet napisów. Podobnie możemy pisać uniwersalne szablony klas.

Mechanizmy wzorców zostały szczegółowo opisane i omówione w pracy Bjarne'a Stroustrupa w 1998 roku w pracy pod tytułem „Parametrized Types for C++".

Szablony, prawdopodobnie ze względu na szerokie rozpowszechnienie się zastosowań biblioteki szablonów STL (ang. Standard Template Library) uważane są za jedną z najważniejszych właściwości języka C++. Dzięki szablonom mamy możliwości programowania ogólnego (ang. generic programming). Tworząc szablon funkcji (podobnie jak szablon klasy) tworzymy kod działający na nieopisanych jeszcze typach – funkcja działa na typach ogólnych (które nie istnieją w języku C++), w momencie wywołania funkcji pod te typy ogólne podstawiamy konkretne typy, jak np. int czy char. Przekazujemy szablonowi typy jako parametry, kompilator generuje funkcje konkretnego typu.

12.2. Przeciążanie funkcji

Przeciążanie funkcji (inna nazwa – polimorfizm funkcji) jest bardzo eleganckim rozszerzeniem języka C++ względem C. Dzięki przeciążaniu funkcji możemy zdefiniować całą rodzinę funkcji realizujących takie same zadania ale dla różnych argumentów. Nie należy mylić przeciążania funkcji z szablonami funkcji. Formalnie język C++ pozwala na definiowanie wielu funkcji o tej samej nazwie, pod warunkiem, że będą miały różne zestawy argumentów. Kluczem do przeciążania funkcji jest lista argumentów i zwracany typ. Dla każdej funkcji kompilator tworzy tzw. sygnaturę funkcji, koduje nazwę funkcji oraz liczbę i typy jej parametrów (nazywamy to zniekształceniem nazwy, ozdabianiem nazwy lub dekorowaniem funkcji). Język C++ pozwala nam definiować funkcje o takich samych nazwach ale o różnych sygnaturach.

```
int kwadrat(int x);
int kwadrat (int  &x);
```

argumenty przekazywane do funkcji są różne, ale nie mamy tu do czynienia z przeciążaniem funkcji ponieważ z punktu widzenia kompilatora wywołanie:

```
kwadrat(x);
```

pasuje do prototypu int x oraz do prototypu &x. W takiej sytuacji kompilator raczej nie jest w stanie zadecydować jakiej wersji funkcji kwadrat() ma użyć. Kompilator traktuje referencje do danego typu i sam typ jako równoważne. Podobnie, z punktu widzenia kompilatora następujące deklaracje są równoważne:

```
int suma ( int *x ) ;
int suma ( int x[ ] );
```

Dla kompilatora *x i x[ ] oznacza to samo.

Należy też mieć na uwadze, że o przeciążeniu funkcji decyduje sygnatura a nie typ funkcji. Nie możemy przeciążyć funkcji:

```
int suma ( int a, double b)
double suma ( int a, double b)
```

ponieważ mają takie same sygnatury. Przeciążyć możemy następujący zestaw funkcji:

```
int suma ( int a, double b)
double suma ( double a, double b)
```

Zagadnienie przeciążania funkcji zilustrujemy klasycznym przykładem dydaktycznym jakim jest funkcja obliczająca kwadrat liczby.

Listing 12.1. Przykład przeciążenia funkcji

```
   #include <iostream>
 2 #include <conio.h>

 4 int kwadrat (int);
   double kwadrat (double);
 6 int kwadrat (int, int);

 8 using namespace std;

10 int main()
   { cout << kwadrat (3)   << endl;
12   cout << kwadrat (3.3) << endl;
     cout << kwadrat (3,4) << endl;
```

```
14  getche();
    return 0;
16 }
   int kwadrat(int x)
18  { cout << x <<" do kwadratu = " ;
      return x*x;                    }
20 double kwadrat(double x)
    { cout << x <<" do kwadratu = ";
22    return x*x;                    }
   int kwadrat(int a, int b)
24  { cout << " a*a + b*b = " ;
      return a*a + b*b;
26 }
```

Po uruchomieniu tego programu mamy komunikat:

```
3 do kwadratu = 9
3.3 do kwadratu = 10.89
a*a + b*b = 25
```

Trzy funkcje o nazwie kwadrat() są rozróżnialne bez kłopotu przez kompilator ze względu na typ parametrów (kwadrat(int) oraz kwadrat (double)) oraz ze względu na liczbę argumentów (kwadrat (int) i kwadrat (int, int)). W kolejnym przykładzie pokazano rozbudowany zestaw użytecznych funkcji przeciążonych do wyznaczania najmniejszej liczby ze zbioru. Kompilator bez trudu radzi sobie z identyfikacją funkcji.

Listing 12.2. Przykład przeciążenia funkcji

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;

5 int min (int, int);
  int min (int, int, int);
7 int min (int *, int);
  long min( long, long);
9 double min (double, double);
  int main()
11 { int a = 22,    b = 7,    c = 3;
    long e = 8,    f = 11;
13  double g = 14,     h = 99;
    cout << "min = " << min(a, b) << endl;
15  cout << "min = " << min(a, b, c) << endl;
    cout << "min = " << min(&a, f) << endl;
17  cout << "min = " << min(e, f)  << endl;
    cout << "min = " << min(g, h) << endl;
19  cout << "min = " << min((double)a,h) << endl;
    getche();
```

```
21    return 0;
   }
23 int min (int x, int y)
        { return x<y ? x : y; }
25 int min (int x, int y, int z)
        { if (x<y) return x<z ? x : z;
27         else return y<z ? y : z; }
   int min (int *x, int y)
29      { return  *x<y ? *x : y; }
   long min( long x, long y)
31      { return x<y ? x : y; }
   double min (double x, double y)
33      { return x<y ? x : y; }
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
min = 7
min = 3
min = 11
min = 8
min = 14
min = 22
```

Kompilator nie ma problemu z identyfikacją odpowiedniej realizacji funkcji min().

W wywołaniu:

```
cout << "min = " << min((double)a,h) << endl;
```

musimy dokonać konwersji jawnej, gdyż kompilator nie jest w stanie w stanie rozróżnić typów. Gdyby wywołanie miało postać:

```
cout << "min = " << min(a,h) << endl;
```

pojawi się następujący komunikat błędu:

```
[C++ Error] : E2014 Ambiguity between
"min(double,double)" and "min(int,int)"
```

W definicjach:

```
int min (int, int);
int min (int *, int);
```

liczby parametrów są równe ale typy się różnią, druga funkcja ma wskaźnik do typu int zamiast czystego int. To wystarcza kompilatorowi aby rozróżnić te funkcje.

Dość ciekawą konstrukcja jest przeciążanie funkcji przy pomocy wskaźnika do funkcji. Możemy deklarować wskaźniki i przypisywać im adresy funkcji przeciążonych. Poniżej przypominamy w jaki sposób wykorzystujemy wskaźnik do funkcji.

Listing 12.3. Przykład użycia wskaźnika do funkcji

```
  #include <iostream>
2 #include <conio.h>
  using namespace std;
4
  int kwadrat (int);
6 int main()
  { int (*w1) (int);//deklaracja wskaznika
8                   //do funkcji typu int
  w1 = &kwadrat;    //przypisanie wskaznikowi adresu funkcji
10 cout << (*w1)(5);// wywolanie funkcji z argumentem 5
     getche();    // i wyswietlenie kwadratu argumentu
12  return 0;
  }
14 int kwadrat(int x)
              {return x*x; }
```

W pokazanym poniżej programie pokazano przeciążanie funkcji posługując się wskaźnikami do funkcji.

Listing 12.4. Przykład użycia wskaźnika do przeciążanych funkcji

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;

5 void ramka(int, char);
  void ramka(int, char, char);
7 int main()
  {
9  void (*w1) (int,char);          // wskaznik do funkcji
   void (*w2) (int, char, char);   // wskaznik do funkcji
11 w1 = ramka;            // dopasowanie (int, char)
   w2 = ramka;            // dopasowanie (int, char,char)
13
   w1(10,'-');
15 w2(10,'|','.');
   w1(10, '\xCD');
17 getche();
   return 0;
19 }

21 void ramka (int n, char z)
     {  for (int i=0; i<=n; i++) cout << (char)z ;
```

```
23          cout << endl;
        }
25  void ramka(int n, char z1, char z2)
        {   cout << z1;
27          for (int i=1; i<n; i++) cout << z2 ;
            cout << z1 << endl;;
29      }
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
- - - - - - - - - - -
|. . . . . . . . . .|
= = = = = = = = = = =
```

W pokazanym przykładzie mamy zadeklarowane dwie funkcje o nazwie ramka. Różnią się one liczbą argumentów. Pierwsza umieszcza na ekranie zadana liczbę znaków, druga umieszcza na ekranie znak początkowy i końcowy oraz drugi znak podaną ilość razy.

W instrukcjach zadeklarowano wskaźniki do dwóch funkcji

```
void (*w1) (int,char);          //wskaznik do funkcji
void (*w2) (int, char, char);   //wskaznik do funkcji
```

a w instrukcjach:

```
w1(10,'-');
 w2(10,'|','.');
 w1(10, '\xCD');
```

wywołano te funkcje. Dopasowanie funkcji do konkretnej postaci realizuje się w instrukcjach:

```
w1 = ramka;    // dopasowanie (int, char)
w2 = ramka;    //dopasowanie (int, char,char)
```

Zadeklarowane metody możemy także przeciążać, pod znanym warunkiem, że inna jest liczba argumentów lub inne są ich typy. Na przykład w klasie Osoba mamy dwie przeciążone metody o nazwie set().

Listing 12.5. Przykład przeciążania metod

```
  #include <iostream>
2 #include <conio.h>
  using namespace std;
4
  class Osoba
6 {    string nazwisko;
       int rok;
8   public:
```

```
        void drukuj();
        void set (string nn){ nazwisko = nn;} //przeciazona
                                              //metoda
        void set (int rr) { rok = rr; }    //przeciazona metoda
};
  void Osoba :: drukuj ()
     {cout << nazwisko << "  " << rok ;
     }

int main()
{ Osoba os;
        os.set("Kowalski");
        os.set(1999);
        os.drukuj();
  getche();
  return 0;
}
```

Po uruchomieniu tego programu mamy wydruk:

```
Kowalski 1999
```

Jak pamiętamy, w klasie możemy mieć wiele konstruktorów, wszystkie o takiej samej nazwie. Praktycznie mamy do czynienie z przeciążaniem konstruktorów. Kolejny przykład przypomni przeciążanie konstruktorów.

Listing 12.6. Przykład przeciążania konstruktorów

```
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;

class Data
{       int miesiac, dzien, rok;
   public:
       Data(int = 8, int = 7, int = 2006);   // konstruktor
       Data(long);                           // konstruktor
   void drukuj_Data();
};
  Data :: Data(int mm, int dd, int rr)
      { miesiac = mm;
        dzien = dd;
        rok = rr;
      }
  Data :: Data(long k_data)
     { rok = int(k_data/10000.0);
       miesiac = int ((k_data - rok * 10000.0)/100.0);
       dzien = int(k_data - rok*10000.0 - miesiac * 100.0);
     }
```

```
23 void Data :: drukuj_Data()
     { cout << setfill ('0')
25          << setw(2) << miesiac << '/'
            << setw(2) << dzien << '/'
27          << setw(2) << rok %100 << endl;
     }
29 int main()
   { Data a, b(20081225L), c(5,1,1947);
31     cout << "data a : " ;
            a.drukuj_Data() ;
33     cout << "data b : " ;
            b.drukuj_Data();
35     cout << "data c : " ;
            c.drukuj_Data();
37  getche();
    return 0;
39 }
```

Po uruchomieniu programu, mamy następujący wydruk:

```
data a : 08/07/06
data b : 12/25/08
data c : 05/01/47
```

W pokazanym przykładzie zastosowaliśmy konstruktor konwertujący typy (ang. type conversion constructor):

```
Data(long);   // konstruktor
```

W danych klasy nie mamy typu long. W naszym przykładzie konstruktor konwertujący będzie przekształcał dane typu long na obiekt typu Data. Nasz obiekt typu Data opisuje datę w formie miesiąc/ dzień/ rok. Jest to klasyczna konwencja stosowana w Stanach Zjednoczonych. Zmienna typu long integer przechowuje datę w formie :

```
rok * 10000 + miesiac * 100 + dzien
```

Reprezentowanie daty w tej postaci posiada kilka zalet: po pierwsze umożliwia reprezentowanie daty w postaci liczby całkowitej, dane tego typu mają wzrastający porządek, po drugie sortowanie dat staje się ekstremalnie łatwe i szybkie. Konstruktor konwertujący ma postać:

```
Data :: Data(long k_data)
   { rok = int(k_data/10000.0);
     miesiac = int ((k_data - rok * 10000.0)/100.0);
     dzien = int(k_data - rok*10000.0 - miesiac * 100.0);
   }
```

W pokazanym przykładzie nie musimy stosować przeciążonego konstruktora, możemy utworzyć funkcję operatorową. Funkcja operatorowa przekształca datę w postaci 1,5,1947 (miesiąc, dzień rok) do postaci typu long integer.

Listing 12.7. Funkcja operatorowa do konwersji typu

```
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;

class Data
{     int miesiac, dzien, rok;
   public:
      Data(int = 8, int = 7, int = 2006); // konstruktor
      operator long();      // funkcja operatorowa
      void drukuj_Data();
};
 Data :: Data(int mm, int dd, int rr)
     { miesiac = mm;
       dzien = dd;
       rok = rr;
     }
 Data :: operator long ()
     { long rmd;
       rmd = rok *10000.0 + miesiac*100.0 + dzien;
       return rmd;
     }
 void Data :: drukuj_Data()
     { cout << setfill ('0')
          << setw(2) << miesiac << '/'
          << setw(2) << dzien << '/'
          << setw(2) << rok %100 << endl;
     }
int main()
 {   Data a(1,5,1947); // deklaracja obiektu typu Data
     long b = a;       // deklaracja obiektu typu long
        cout << "data a : "  ;
           a.drukuj_Data()  ;
        cout << "ta data jako long : " << b;

 getche();
 return 0;
}
```

Po uruchomieniu tego programu otrzymujemy następujący komunikat:

```
data a : 01/05/47
ta data jako long : 19470105
```

Deklaracja funkcji operatorowej ma postać:

```
operator long();          // funkcja operatorowa
```

a definicja jest następująca:

```
Data :: operator long ()
{  long rmd;
   rmd = rok *10000.0 + miesiac*100.0 + dzien;
   return rmd;
}
```

W kolejnym przykładzie pokażemy użyteczne przeciążania konstruktora służące także do obsługi daty. Bardzo często zdarza się, że data podawana jest w dwóch formatach – jako dane typu int albo dane są typu napisowego. Przeciążenie konstruktora obsłuży te dwa sposoby pisania dat. W programie wykorzystamy specyficzną funkcje wejścia jaką jest funkcja sscanf().

Funkcja sscanf() (wymagany jest plik <cstdio>), której prototyp ma postać:

```
int sscanf() (const char *buf, const char *format, ...);
```

służy do czytania danych, w przeciwieństwie do funkcji scanf() nie czyta danych z standardowego wejścia stdin ale z tablicy wskazywanej przez buf. W łańcuchu formatującym wykorzystano symbol *. Symbol * umieszczony miedzy znakiem % a kodem formatującym sprawia, że wczytywane są dane konkretnego typu, natomiast przypisania są ignorowane. W instrukcji:

```
scanf("%d*c%d", &a, &b);
```

jeżeli wpiszemy dane wejściowe w postaci 21/10, wtedy 21 zostanie przypisane zmiennej a, / zostanie zignorowane, a 10 będzie przypisane zmiennej b.

Listing 12.8. Przeciążanie konstruktorów (obsługa typu int i napis)

```
1 #include <iostream>
  #include <conio.h>
3 #include <cstdio>
  using namespace std;
5  class Data
    {   int dzien, miesiac, rok;
7      public:
         Data(char *);           //konstruktor: obsluga napisow
9        Data (int, int, int);//konstruktor: obsluga liczb
         void drukuj_Data();
11    };
    Data :: Data(char *buf)
```

```
13     { sscanf(buf,"%i%*c%i%*c%i", &miesiac, &dzien, &rok);
       }
15   Data :: Data( int m, int d, int r)
       { dzien = d;
17       miesiac = m;
         rok = r;
19     }
     void Data :: drukuj_Data()
21     { cout << miesiac << "/" << dzien;
         cout << "/" << rok << endl;
23     }
   int main()
25 { Data d1(11, 21, 2008);
     Data d2("12/27/2008");
27     d1.drukuj_Data();
       d2.drukuj_Data();
29   getche();
     return 0;
31 }
```

Po uruchomieniu programu otrzymujemy komunikat:

```
11/21/2008
12/27/2008
```

Widzimy, że chociaż obiekty typu Data były inicjowane przy użyciu trzech wartości typu int :

```
Data d1(11, 21, 2008);
```

(obiekt d1) oraz za pomocą łańcucha znakowego:

```
Data d2("12/27/2008");
```

(obiekt d2), to wyświetlanie daty zostało wykonane poprawnie. Dzięki przeciążaniu konstruktorów, użytkownik może decydować w jaki sposób będzie wprowadzał daty do programu. Takie podejście zmniejsza ryzyko popełnienia błędu w danych wejściowych.

12.3. Szablony funkcji

W poprzednich rozważaniach wykazaliśmy użyteczność przeciążania funkcji. Jednak jak pokazuje wydruk programu 12.2 aby szeroko obsłużyć typy danych jakie mogą być argumentami w funkcji min() musieliśmy napisać pięć wersji tej funkcji. W naszym przykładzie nie było zbyt dużo instrukcji do zapisu, ale łatwo możemy sobie wyobrazić funkcję, której kod składa się

z np. 60 instrukcji. Napisanie pięciu wersji takiej funkcji jest pracochłonne, ale co najważniejsze wraz z ilością napisanych instrukcji wzrasta prawdopodobieństwo popełnienie błędu. Można zadać pytanie, czy nie można uniknąć żmudnego kodowania wielu funkcji wykonujących takie samo zadanie. Jak się domyślamy, odpowiedź na to pytanie jest pozytywna. Nowoczesne kompilatory języka C++ pozwalają na kodowanie funkcji z typami ogólnymi (nieokreślonymi), w momencie wywołania funkcji typy ogólne zastępowane są typami konkretnymi, mówimy o kodowaniu funkcji wzorcowych albo inaczej szablonów funkcji. Szablony umożliwiają programowanie ogólne (ang. generic programming). Bardzo często mamy do czynienia np. z sortowaniem tablic. W takim przypadku musimy napisać funkcję do sortowania tablic przechowujących np. elementy typu int. Gdybyśmy chcieli posortować tablicę z elementami typu double musielibyśmy na nowo napisać inną funkcję sortującą. Szablon funkcji umożliwia napisanie jednej funkcji sortującej elementy tablicy wykorzystując typ ogólny, a kompilator w zależności od sposobu wywołania wygeneruje automatycznie kod funkcji sortującej konkretny typ. Będą to w naszym przypadku dwie funkcje – jedna sortująca elementy typu int i jedna sortująca elementy typu double. Formalnie szablon funkcji nie jest niczym nowym. Programiści języka C (w którym nie można pisać szablonów funkcji) mają do dyspozycji makrodefinicje preprocesora – dzięki nim można otrzymać różne realizacje kodu dla każdego typu wywołania. Wadą makrodefinicji jest duża podatność na błędy i powstawanie efektów ubocznych. Aby wyeliminować kłopoty związane z makrodefinicjami, twórcy języka C++ wprowadzili wzorce (szablony) funkcji.

Listing 12.9. Użycie wzorca funkcji

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;

5 template <class T >
  T kwadrat (T x)
7  {   return x*x;
   }
9
  int main()
11 { cout << "wynik int:   " << kwadrat (2)   << endl;
    cout << "wynik float: " << kwadrat (2.2) << endl;
13  getche();
    return 0;
15 }
```

Po uruchomieniu tego programu mamy następujący wynik:

```
wynik int:    4
wynik float: 4.84
```

Szablon funkcji obliczającej kwadrat argumentu może mieć postać:

```
template <class T >
T kwadrat (T x)
{   return x*x;
}
```

Definicja wzorca funkcji zaczyna się od słowa kluczowego template, za nim w nawiasach ostrych umieszczona jest lista formalnych typów ogólnych. Każdy parametr formalny poprzedzony jest słowem kluczowym class:

```
template < class T >
```

W naszym przykładzie mamy jeden parametr ogólny o nazwie T. Słowa kluczowe template i class są obowiązkowe. W najnowszych kompilatorach nieco mylące słowo kluczowe class można zastąpić innym słowem kluczowym typename:

```
template < typenameT >
```

Obowiązkowo musimy stosować nawiasy ostre. Po deklaracji tworzenia szablonu następuje zwykła definicja funkcji:

```
T kwadrat (T x)
{   return x*x;
}
```

Widzimy zwracany typ (w naszym przypadku jest to typ ogólny T), nazwę funkcji (w naszym przypadku kwadrat) oraz listę argumentów. W nawiasach klamrowych umieszczone jest ciało funkcji.

Bardziej rozbudowany przykład użycia szablonu funkcji pokazany jest w kolejnym przykładzie. W programie użyjemy wzorca funkcji max(), która pobiera trzy argumenty i zwraca wartość największego argumentu oraz funkcji foo() która wczytuje dane z klawiatury i wyświetla wyniki. Działanie szablonów funkcji testujemy na trzech typach danych : int, double i char.

Listing 12.10. Użycie wzorców funkcji

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;
```

```
5  template <class T>
   T max(T w1, T w2, T w3)
7  { T max = w1;
     if (w2 > max) max = w2;
9    if (w3 > max) max = w3;
     return max;
11 }
   template <class T>
13 void foo(T x1, T x2, T x3)
   { cout << "\nargument 1: " ;  cin >> x1;
15   cout << "\nargument 2: " ;  cin >> x2;
     cout << "\nargument 3: " ;  cin >> x3;
17   cout << "\nnajwieksza wartosc to: " << max(x1,x2,x3);
   }
19 int main()
   {  int n1,n2,n3;              //wersja int
21       foo(n1,n2,n3);
      double d1,d2,d3;        //wersja double
23       foo(d1,d2,d3);
      char c1,c2,c3;            //wersja char
25       foo(c1,c2,c3);
     getche();
27   return 0;
   }
```

Po uruchomieniu programu mamy następujący wydruk:

```
argument 1: 10
argument 2: 22
argument 3: 11
najwieksza wartosc to: 22
argument 1: 1.1
argument 2: 3.3
argument 3: 1.7
najwieksza wartosc to: 3.3
argument 1: x
argument 2: y
argument 3: z
najwieksza wartosc to: z
```

Program działa zgodnie z oczekiwaniem. Gdy do szablonu funkcji przekazujemy np. argumenty typu int, kompilator tworzy kompletna funkcje do obliczenia największej wartości spośród trzech wprowadzonych danych typu int. Gdy wprowadzimy dane typu char tworzona jest funkcja do obsługi argumentów typu char. W programie zdefiniowaliśmy dwa szablony, szablon funkcji max():

```
template <class T>
T max(T w1, T w2, T w3)
```

```
{ T max = w1;
  if (w2 > max) max = w2;
  if (w3 > max) max = w3;
  return max;
}
```

oraz foo():

```
template <class T>
void foo(T x1, T x2, T x3)
{ cout << "\nargument 1: " ;  cin >> x1;
  cout << "\nargument 2: " ;  cin >> x2;
  cout << "\nargument 3: " ;  cin >> x3;
  cout << "\nnajwieksza wartosc to: " << max(x1,x2,x3);
}
```

Realizacja szablonu funkcji max() dla obsługi argumentów typu int ma postać:

```
int max(int w1, int w2, int w3)
{ int max = w1;
  if (w2 > max) max = w2;
  if (w3 > max) max = w3;
  return max;
}
```

Szablony funkcji mogą wykorzystywać jako argumenty dane nie będące prostymi typami. W kolejnym przykładzie zmienne są zdefiniowane strukturami. W programie porównujemy dwa wektory i wybieramy wektor dłuższy. Pamiętamy, że dla wektora dwuwymiarowego, który ma składowe (x, y) długość wektora określona jest wzorem:

$$||d|| = \sqrt{x^2 + y^2} \tag{12.1}$$

Aby określić, który z wektorów jest dłuższy wystarczy obliczyć sumę kwadratów składowych wektorów. Wektor zdefiniowany jest następującą strukturą:

```
struct wektor
{ int x, y;
  wektor (int xx=0, int yy = 0) : x(xx), y(yy) { }
  bool operator < (wektor & w)  //przeciazony operator
          { return (x*x + y*y) < (w.x*w.x + w.y*w.y) ;
          }
};
```

Listing 12.11. Użycie wzorców funkcji; wykorzystanie struktury

```
#include <iostream>
#include <conio.h>
using namespace std;

struct wektor
{ int x, y;
  wektor (int xx=0, int yy = 0) : x(xx), y(yy) { }
  bool operator < (wektor & w) //przeciazony operator
           { return (x*x + y*y) < (w.x*w.x + w.y*w.y) ;
           }
};

template < class T>
T max(T a, T b)
  { return a < b ? b : a;
  }

int main()
{  wektor w1(1,1), w2(2,2), ww;
   ww = max(w1,w2);
   cout << "dluzszy wektor ma skladowe: "<< endl;
   cout << ww.x << "   " << ww.y << endl;

   getche();
   return 0;
}
```

Po uruchomieniu programu mamy wydruk:

```
dluzszy wektor ma skladowe:
2   2
```

Wzorzec funkcji doskonale radzi sobie ze zmienną zdefiniowaną strukturą. Szablon funkcji określającej który z wektorów jest dłuższy ma postać:

```
template < class T>
T max(T a, T b)
  { return a < b ? b : a;
  }
```

Przypominamy, że w deklaracji szablonu możemy użyć słowa kluczowego typename:

```
template < typename T>
T max(T a, T b)
  { return a < b ? b : a;
  }
```

Tworząc szablony funkcji możemy korzystać z więcej niż jednego typu ogólnego.

Listing 12.12. Użycie wzorców funkcji – dwa typy uogólnione

```
#include <iostream>
#include <conio.h>
using namespace std;
template <class T, class R>
void pokaz (T x1, R x2)
{ cout << " argument 1: " << x1;
  cout << " , argument 2: "<< x2 << endl;
}
int main()
{ int n1 = 1947;
  double d1 = 99.99;
  char *xx = "Wacek";
      pokaz(n1, xx);
      pokaz(n1, d1);
 getche();
 return 0;
}
```

Po uruchomieniu programu mamy komunikat:

```
argument 1: 1947 , argument 2: Wacek
argument 1: 1947 , argument 2: 99.99
```

Aby wprowadzić żądaną ilość argumentów uogólnionych w nagłówku szablonu funkcji musimy to zadeklarować przy pomocy słowa kluczowego class: class T1, class T2, class T3...... W naszym przykładzie żądaliśmy dwóch typów uogólnionych, wobec tego szablon funkcji miał postać:

```
template <class T, class R>
void pokaz ( T x1, R x2 )
{  cout << " argument 1: " << x1;
   cout << " , argument 2: "<< x2 << endl;
}
```

Argument formalny x1 jest typu uogólnionego T, a argument formalny x2 jest typu ogólnego R. Szablony funkcji możemy przeciążać. Możemy np. szukać większego z dwóch wprowadzonych argumentów lub też szukać największego elementu tablicy. W tym celu możemy przeciążyć szablon funkcji, podobnie jak to robimy ze zwykłymi funkcjami.

Listing 12.13. Użycie wzorców funkcji – przeciążanie szablonów funkcji

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

```
3 using namespace std;

5 template <class T>
  T max (T & x1, T & x2)
7   { if (x1 > x2) return x1;
      else         return x2;
9   }

11 template <class R>
   R max (R *m, int n)
13  { R mm = m[0];
        for (int i = 1; i < n; i++ )
15           if (m[i] > mm) mm = m[i];
      return mm;
17  }

19 int main()
   { int n1 = 10, n2 = 33;
21      cout << "wieksza z dwoch = " << max(n1,n2);
     double dd[10] = {2, 3, 4, 11, 5, 77, 33, 1, 9, 10};
23      cout << " \nnajwieksza w tablicy = " << max(dd, 10);
    getche();
25  return 0;
   }
```

Po uruchomieniu tego programu mamy następujący wynik:

```
wieksza z dwoch = 33
najwieksza w tablicy = 77
```

Gdy w pokazanym programie kompilator dojdzie do wywołania:

```
cout << "wieksza z dwoch = " << max(n1,n2);
```

stwierdzi, że użyto dwóch argumentów typu int wobec czego wywoła pasujący szablon:

```
template <class T>
T max (T & x1, T & x2)
  { if (x1 > x2) return x1;
    else         return x2;
  }
```

W kolejnym wywołaniu funkcji max():

```
cout << " \nnajwieksza w tablicy = " << max(dd, 10);
```

argumentem funkcji jest tablica typu double i całkowita liczba 10, wobec czego wywołany zostanie szablon:

```
template <class R>
R max (R *m, int n)
  { R mm = m[0];
      for (int i = 1; i < n; i++ )
          if (m[i] > mm) mm = m[i];
    return mm;
  }
```

Zwróćmy uwagę, że w szablonie funkcji max():

```
template <class R>
R max (R *m, int n)
```

mamy połączenie parametrów uogólnionych (R *m) oraz standardowych (int n). Łączenie parametrów uogólnionych i standardowych jest często wykorzystywane w praktycznych zastosowaniach i nie sprawia kompilatorowi żadnych kłopotów.

Posługując się szablonami możemy je przeciążać w sposób jawny, to znaczy obok szablonu funkcji możemy jednocześnie zdefiniować przeciążona zwykłą funkcję.

W kolejnym przykładzie zilustrujemy to zagadnienie.

Listing 12.14. Użycie wzorców funkcji – jawne przeciążanie szablonów

```
   #include <iostream>
 2 #include <conio.h>
   using namespace std;
 4
   template <class T>
 6 T max (T & x1, T & x2)
     { if (x1 > x2) return x1;
 8     else         return x2;
     }
10 double max (double y1, double y2)
     {cout << "jestem w double" << endl;
12     if (y1 > y2) return y1;
       else         return y2;
14   }
   int main()
16 { int n1 = 10, n2 = 33;
       cout << "wieksza z dwoch int = " << max(n1,n2);
18   char c1 = 'A', c2 = 'a';
       cout << "wieksza z dwoch char = " << max(c1,c2);
20   double d1 = 13.13, d2 = 99.99;
       cout << "wieksza z dwoch double = " << max(d1,d2);
22  getche();
    return 0;
24 }
```

Po uruchomieniu tego programu mamy wynik:

```
wieksza z dwoch int = 33
wieksza z dwoch char = a
jestem w double
wieksza z dwoch double = 99.99
```

Pamiętamy, że w kodzie ASCII znak A ma kod dziesiętny 65, a znak a ma kod 97. Aby przekonać się, że rzeczywiście kompilator użył jawnie przeciążonej wersji funkcji max() w kodzie tej funkcji umieściliśmy odpowiedni komunikat:

```
double max (double y1, double y2)
  {cout << "jestem w double" << endl;
    if (y1 > y2) return y1;
    else         return y2;
  }
```

W programach, w których mamy szablony funkcji i zwykle funkcje, które przeciążają je, możemy w sposób jawny wymusić wykonanie żądanych operacji przez funkcje szablonowe. Kolejny przykład ilustruje to zagadnienie.

Listing 12.15. Użycie wzorców funkcji – jawne przeciążanie szablonów

```
1 #include <iostream>
  #include <conio.h>
3 using namespace std;

5 template <typename T>
  T max( T n1, T n2)
7   { return cout  << "szablon funkcji: ",
         n1 < n2 ? n2 : n1;
9   }
   char *max( char *m1, char *m2)
11  { return cout  << "zwykla funkcja : ",
         strcmp(m1,m2) > 0 ? m2 : m1;
13  }
  int main()
15 { cout << max("Ewa", "Anna") << endl;
    cout << max("Anna", "Ewa") << endl;
17  cout << max<>("Ewa", "Anna") << endl;
    cout << max<>("Anna", "Ewa") << endl;
19  getche();
    return 0;
21 }
```

Po uruchomieniu tego programu mamy następujący wynik:

```
zwykla funkcja : Anna
```

```
zwykla funkcja : Anna
szablon funkcji: Anna
szablon funkcji: Ewa
```

Wiemy, że funkcja ogólna sama dokonuje przeciążenia. W konkretnych przypadkach chcemy mieć kontrolę nad działaniem programu i chcemy wykonać przeciążenie w sposób jawny. Takie działanie nosi nazwę „jawne tworzenie funkcji specjalizowanej”. Przeciążona funkcja ogólna przesłania funkcję ogólną utworzoną specyficzną wersją. Dla jawnego wykonania funkcji szablonowej wykorzystano oznaczenie <> oraz postać alternatywną <char*>. Widzimy, że wersja szablonowa funkcji nie działa poprawnie. Kod jest źle zaprojektowany. Dobrze jest stworzyć specjalizacje szablonu funkcji dla wybranego typu.

Dla przypomnienia, nasz program porównuje dwa napisy. Zwykła funkcja działa poprawnie, wykorzystuje funkcje biblioteczna strcmp().

Prototyp tej funkcji ma postać:

```
int strcmp(const char *s1, const char *s2);
```

Funkcja strcmp() zdefiniowana jest w pliku <string.h>. Funkcja zwraca zero jeżeli napisy są identyczne, zwraca liczbę ujemna gdy $s1 < s2$, zwraca liczbę dodatnią w przeciwnym przypadku.

Widzimy, że szablon nie działa poprawnie dla wszystkich typów danych, chcemy aby program wymuszał wersję zwykłą, gdy musimy obsłużyć napisy. W tym celu musimy posłużyć się oznaczeniem template <>. W takim przypadku zawsze wywołana będzie funkcja zwykła. Specjalizację szablonu funkcji dla wybranego typu ilustruje kolejny program.

Listing 12.16. Użycie wzorców funkcji – jawne przeciążanie szablonów

```
1  #include <iostream>
   #include <conio.h>
3  using namespace std;

5  template <typename T>
   T max( T n1, T n2)
7    { return cout << "szablon funkcji: ",
                      n1 < n2 ? n2 : n1;
9    }
   template<>
11  char *max( char *m1, char *m2)
     { return cout << "zwykla funkcja : ",
13                     strcmp(m1,m2) > 0 ? m2 : m1;
     }
15
   int main()
17 { cout << max("Ewa", "Anna") << endl;
```

```
    cout << max("Anna", "Ewa") << endl;
19  cout << max<>("Ewa", "Anna") << endl;
    cout << max<>("Anna", "Ewa") << endl;
21
   getche();
23 return 0;
 }
```

Po uruchomieniu tej wersji programu mamy następujący wynik:

```
zwykla funkcja : Anna
zwykla funkcja : Anna
zwykla funkcja : Anna
zwykla funkcja : Anna
```

Widzimy, że we wszystkich przypadkach wywoływana jest zwykła funkcja. Należy pamiętać, że jawna specjalizacja jest stosunkowo nową składnią, i nie wszystkie kompilatory radzą sobie z konstrukcją template <>. Należy podchodzić z duża ostrożnością do tego zagadnienia i zawsze testować działanie jawnej specjalizacji dla wybranego typu.

Szablony funkcji są bardzo często wykorzystywane w praktyce, Klasycznym przykładem jest szablon funkcji do sortowania dowolnej tablicy. Ze względu na prosty kod wybieramy metodę sortowania bąbelkowego (ang. bublbe sort)

Listing 12.17. Użycie wzorców funkcji – sortowanie bąbelkowe

```
  #include <iostream>
2 #include <conio.h>
  using namespace std;
4 template <class T>
  void bubble(T *mm, int n)
6   { T x;
      for (int i = 1; i < n; i++)
8       for(int j = n-1; j >=i; j--)
          if (mm[j-1] > mm[j])
10           { x = mm[j-1];
               mm[j-1] = mm[j];
12             mm[j] = x;
             }
14  }
   template <class R>
16 void pokaz(R *mm, int p)
     { for (int i =0; i<p; i++)
18         cout << "   " << mm[i] << "   ";
       cout << endl;
20   }

22 int main()
```

```
   { int m1 [5] = { 11, 33, 23, 15, 77};
24   char m2[5] = {'d', 'r', 'a', 'e', 'g'};
     bubble(m1, 5);
26   bubble(m2, 5);
     cout << "posortowane liczby calkowite: "<< endl;
28   pokaz(m1, 5);
     cout << "posortowane znaki : "<< endl;
30   pokaz(m2, 5);
    getche();
32  return 0;
   }
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
posortowane liczby calkowite:
11  15  23  33  77
posortowane znaki:
a   d   e   g   r
```

Dla testów utworzyliśmy dwie tablice pięcioelementowe : tablicę typu int oraz tablicę typu char. Obie tablice zostały poprawnie posortowane.

Pokazany szablon funkcji sortującej jest automatycznie przeciążany do żądanego typu – mamy bardzo sprawny szablon funkcji do sortowania tablic dowolnych elementów.

Użyteczny może być także szablon funkcji do przeszukiwania tablicy dowolnego typu. Pokażemy szablon funkcji realizujący przeszukiwanie metodą połowienia (ang. binary search).

Definicja szablonu funkcji realizującej przeszukiwanie może mieć postać:

```
template <class T>
int BinSearch(T x, T *tablica, int n)
{int p = 0;
 int k = n -1;
 int s;
    while (p <= k)
      {s = (p + k) / 2;
       if (x == tablica[s])
                 return s;
       else if (x < tablica[s])
            k = s - 1;
       else
            p = s +1;
      }
      return -1;
}
```

W pokazanym szablonie, T jest typem uogólnionym. Funkcja szuka elementu

o nazwie x w posortowanej tablicy nazywanej tablica, która przechowuje n elementów.

Zdefiniowana została także szablonowa funkcja pomocnicza drukTab().

```
template <class T>
void drukTab( T * tablica , int n)
{ for (int i = 0; i < n; i++)
      cout  << tablica[i] << "  ";
      cout << endl;
}
```

W pokazanym poniżej programie najpierw definiujemy tablicę (z ustalonym typem, np. int) a potem wywołujemy funkcję szablonową BinSearch().

Listing 12.18. Użycie wzorców funkcji – przeszukiwanie binarne

```
#include <iostream>
2 #include <conio.h>
using namespace std;
4
template <class T>
6 int BinSearch(T x, T *tablica , int n)
{ int p = 0;
8   int k = n -1;
    int s;
10    while (p <= k)
        {s = (p + k) / 2;
12        if (x == tablica[s])
                  return s;
14        else if (x < tablica[s])
               k = s - 1;
16        else
               p = s +1;
18        }
        return -1;
20 }

22 template <class T>
void drukTab( T * tablica , int n)
24 { for (int i = 0; i < n; i++)
      cout  << tablica[i] << "  ";
26    cout << endl;
}
28
int main()
30 { int tab1[] = { 1, 11, 21, 31, 41, 51, 61};
    int t1 = 31;
32  drukTab(tab1,7);
    cout << "element " << t1 << " ma indeks "
34        <<BinSearch (t1, tab1, 7) << endl;
    char tab2[] = { 'a', 'e', 'g', 'h', 'i', 'k', 'x', 'y'};
```

```
36    char t2 = 'k';
      drukTab(tab2,8);
38    cout << "element␣" << t2 << "␣ma␣indeks␣"
           <<BinSearch (t2, tab2, 8) << endl;
40    getche();
      return 0;
42 }
```

Po uruchomieniu tego programu mamy wydruk:

```
1    11    21    31    41    51    61
element 31 ma indeks 3
a    e    g    h    i    k    x    y
element k ma indeks 5
```

Program działa poprawnie, dopasowanie typu przebiegało bez problemu.

Konkretyzacja funkcji dla typu int ma postać:

```
int tab1[ ] = { 1, 11, 21, 31, 41, 51, 61};
int t1 = 31;
drukTab(tab1,7);
cout << "element␣" << t1 << "␣ma␣indeks␣"
     << BinSearch (t1, tab1, 7) << endl;
```

W pokazanym przykładzie tworzymy tablicę liczb całkowitych tab1[ ]. Jeżeli chcemy ustalić położenie elementu o wartości zapisanej w zmiennej t1 to wywołujemy funkcję BinSearch() w następujący sposób:

```
BinSearch (t1, tab1, 7)
```

Skonkretyzowana dla typu int funkcja szablonowa wymaga trzech argumentów : poszukiwanego elementu, tablicy i rozmiaru tablicy. Podobnie postępujemy dla poszukiwania elementu w tablicy znakowej. Na liście parametrów szablonu funkcji możemy umieszczać typy klas. W kolejnym przykładzie zademonstrujemy skomplikowaną obsługę zarówno zmiennych typów wbudowanych jak i obiektów klasy. Kompilatory zachowują się bardzo inteligentnie. W naszym przykładowym programie chcemy porównać długości wektorów i wyznaczyć mniejszy wektor, oczywiście chcemy mieć możliwość wyznaczenia mniejszej z dwóch liczb typu int.

Klasa wek do obsługi wektora 2D ma postać:

```
class wek
{   int x,y;
  public:
    wek( int xx, int yy ) { x = xx ;  y = yy ; }
    int getX() { return x ; }
    int getY() { return y ; }
```

```
    int operator < (wek & );   //przeciazony operator
} ;
```

Szablon funkcji min() do wyznaczenia mniejszej zmiennej ma postać:

```
template <class T>
T& min( T& wr1, T& wr2)
{ if ( wr1 < wr2 )
        return wr1;
  return wr2;
}
```

Funkcja min() może być wykorzystana do porównania dowolnych zmiennych. W naszym przypadku będzie skonkretyzowana dla typu wek, będzie wyznaczała mniejszy z dwóch obiektów typu wek. Do wykonania poprawnie operacji porównania w klasie wek umieszczony został przeciążony operator <.

Dla testów utworzyliśmy dwa obiekty klasy wek : w1 i w2.

Listing 12.19. Użycie wzorców funkcji – typ klasy argumentem szblonu

```
   #include <iostream>
 2 #include <conio.h>
   using namespace std;
 4 class wek
   {   int x,y;
 6   public:
       wek( int xx, int yy ) { x = xx ;  y = yy ; }
 8     int getX() { return x ; }
       int getY() { return y ; }
10     int operator < (wek & );   //przeciazony operator
   } ;
12 template <class T>
   T& min(T& wr1, T& wr2)
14 { if (wr1 < wr2)
          return wr1;
16   return wr2;
   }
18
   int main()
20 { wek w1( 1,1);
     wek w2( 2,2);
22   wek w = min(w1, w2);
     cout << " mniejszy wektor : " << w.getX()
24        << "  " << w.getY() << endl;
     int x1 = 3;
26   int x2 = 4;
     cout << " mniejsza liczba : " << min (x1, x2) << endl;
28
     getche();
```

```
30   return 0;
          }
32
   int wek :: operator < (wek & ww)
34  { if ( x*x + y*y < ww.x * ww.x + ww.y * ww.y)
         return 1;
36    return 0;
    }
```

Po uruchomieniu programu mamy następujący wynik:

```
mniejszy wektor : 1  1
mniejsza liczba  :  3
```

Podczas wywołania postaci :

```
wek w = min(w1, w2);
```

następuje konkretyzacja funkcji szablonu min() dla typu wek. Skonkretyzowana funkcja wyznacza mniejszy obiekt za pomocą przeciążonego operatora mniejszości:

```
int wek :: operator < (wek & ww)
 { if ( x*x + y*y < ww.x * ww.x + ww.y * ww.y)
        return 1;
   return 0;
 }
```

W przypadku porównywania dwóch liczb całkowitych:

```
  int x1 = 3;
  int x2 = 4;
cout << " mniejsza liczba : " << min (x1, x2) << endl;
```

tzn. funkcja min() wywoływana jest z argumentami typu int, do porównania dwóch liczb całkowitych używany jest elementarny operator mniejszości ¡ a nie jego przeciążona wersja.

12.4. Szablony klas

Na podobnej zasadzie co szablony funkcji można także definiować szablony klas (inne określenia: wzorce klas, klasy ogólne). Należy zwrócić uwagę na fakt, że szablony klas (ang. class template) są doskonałym mechanizmem języka C++, Wiemy z praktyki, że niewiele problemów powstaje gdy projektujemy szablony funkcji, to w odniesieniu do klas, szablony są pojęciowo i składniowo skomplikowane. Z tego powodu szablony klas nie są stosowa-

ne tak często jak by można było przypuszczać. W dalszym ciągu obsługa szablonów w różnych kompilatorach może różnie być realizowana. Z drugiej strony coraz większe stosowanie biblioteki STL (Standard Template Library) wymusza zapoznanie się z podstawami i logiką tej biblioteki. Jądrem biblioteki STL są ogólne kontenery i algorytmy. Wszystkie kontenery STL to szablony, można w nich zapisywać dane dowolnego typu. Za pomocą szablonów można pisać kod niezależny nie tylko od przekazanych wartości ale także od typów tych wartości. Szablony klas służą przede wszystkim jako kontenery, są to struktury danych zawierające obiekty.

Deklaracja szablonu klasy (w literaturze przedmiotu spotkamy także nazwy: klasa wzorcowa, wzorzec klasy, klasa ogólna, itp.) ma postać:

```
template  <class T_typ>
class  nazwa_klasy
 {     .
 .
 . //instrukcje
 .

  };
```

Widzimy, że szablon klasy ma postać podobną do deklaracji zwykłej klasy, jedynie na początku musi znajdować się kod postaci:

```
template < class T_typ>
```

Słowo kluczowe template informuje kompilator, że będzie definiowany szablon. T_typ oznacza nazwę typu, który będzie określony podczas tworzenia egzemplarza klasy. Zazwyczaj stosujemy jeden typ ogólny, ale możemy stosować więcej takich typów. W takim przypadku w nawiasach ostrych umieszczamy listę typów ogólnych oddzielonych przecinkami.. Egzemplarz klasy tworzymy przy pomocy konstrukcji:

```
nazwa_klasy < typ> nazwa_obiektu;
```

W tej instrukcji nazwa typ odnosi się do konkretnego typu na którym będzie operować klasa. Nowsze implementacje języka C++ wprowadziły alternatywny zapis:

```
template <typename T_typ>
```

Zastąpienie słowa kluczowego class słowem kluczowym typename wydaje się znacznie logiczniejsze. Podobnie jak w wywołaniu funkcji, podczas tworzenia obiektu klasy, a więc w momencie wywołania szablonu, w miejsce T_typ podstawiany jest konkretny typ taki jak np. int, double, char, itp. Należy

pamiętać, że podczas definiowania metod szablonu klasy, używamy typów ogólnych, a definicja metody musi mieć zapowiedź szablonu:

```
template < class T_typ>
```

Gdy definiujemy metodę w deklaracji szablonu klasy, wtedy opuszczamy zapowiedź szablonu i kwalifikator klasy.

Przy pomocy szablonów klas możemy zbudować kolekcje obiektów dowolnego typu, korzystając z tego samego szablonu klasy, możemy zadeklarować i zdefiniować klasę dla dowolnego typu. O takiej klasie mówimy , że jest sparametryzowana.

Omówimy prosty przykład tworzenia wzorca klasy. Bardzo często operujemy liczbami całkowitymi i rzeczywistymi. Dla każdego takiego zbioru musimy pisać oddzielne klasy : do obsługi liczb całkowitych (np. typ int) oraz do obsługi liczb rzeczywistych (np. typ double). Dzięki wzorcom możemy utworzyć sparametryzowaną klasę, która obsłuży dowolny typ danych.

Listing 12.20. Użycie wzorców klas – prosty przykład

```
 1 #include <iostream>
   #include <conio.h>
 3 using namespace std;

 5 template <class T>
   class war
 7 {    T n;                       //typ ogolny T
     public:
 9       war(): n(0) { }           //konstruktor
         void set_war() { cin >> n ; }
11       void pokaz()     { cout << n << endl; }
   };
13
   int main()
15 { war <char> zn;                //obsluga typu char
     cout << "podaj znak : " ;
17   zn.set_war();
     cout << "wprowadzony znak : " ;
19   zn.pokaz();

21  war <double> x;                //obsluga typu double
     cout << "podaj liczbe rzeczywista : " ;
23   x.set_war();
     cout << "wprowadzona liczba : " ;
25   x.pokaz();
     getche();
27   return 0;
   }
```

Po uruchomieniu tego programu mamy następujący wynik:

```
podaj znak : d
wprowadzony znak : d
podaj liczbe rzeczywista : 13.13
wprowadzona liczba : 13.13
```

W tym przykładzie wprowadzamy z klawiatury wartość żądanego typu, a następnie na ekranie monitora mamy wyświetloną wartość naszej danej. W tym konkretnym przypadku testowano obsługę znaków i liczb zmiennoprzecinkowych. Implementacja ogólnej klasy war ma postać:

```
template <class T>
class war
{   T n;                        //typ ogolny T
  public:
    war(): n(0) { }             //konstruktor
    void set_war() { cin >> n ; }
    void pokaz()    { cout << n << endl; }
};
```

Widzimy deklarację szablonu klasy:

```
template <class T >
  class war
```

Specyfikacja szablonu klasy (nasza klasa nosi nazwę war) zawiera słowo kluczowe template a po nim mamy nawiasy ostre <>. Wewnątrz nawiasów umieszczamy identyfikator, który reprezentuje typ sparametryzowany, w naszym przykładzie jest to identyfikator o nazwie T. Nazwa identyfikatora może być dowolna. Wewnątrz nawiasów można umieścić więcej niż jeden identyfikator, np. :

```
< class T, class W >
```

Parametry muszą być oddzielone przecinkami. Na liście parametrów szablonu mogą być dowolne elementy, możemy mieć parametry ogólne i konkretne:

```
< class T1, int n, class T2 >
```

Jako parametry szablonu mogą być używane typy wbudowane, typy zdefiniowane przez użytkownika oraz wyrażenia stałe. Klasa war posiada jedną daną składową prywatną typu ogólnego:

```
T n;
```

jeden konstruktor bezargumentowy

```
war ( )  :  n ( 0 )  { }          // konstruktor
```

konstruktor możemy zdefiniować bardziej czytelnie:

```
war ( )
{   n = 0;
}
```

oraz dwie publiczne funkcje składowe:

```
void set_war ( ) { cin >> n ; }
void pokaz ( )     { cout << n << endl ; }
```

funkcja set_war () wczytuje z klawiatury wartość skonkretyzowanego typu, a funkcja pokaz() wyświetla tą wartość.

Gdybyśmy nie używali szablonu klasy, to powinniśmy zadeklarować tyle klas ile typów chcielibyśmy obsłużyć. W przypadku szablonu klas mamy jedną deklaracje, a dla danego typu konkretyzujemy naszą klasę. Konkretyzacja następuje gdy użyta będzie nazwa szablonu klasy. Gdy chcemy utworzyć egzemplarz klasy dla liczb zmiennoprzecinkowych, to definicja egzemplarza klasy ma postać:

```
war <double> x;               // obsluga typu double
```

W pokazanym przykładzie identyfikator x jest obiektem klasy typu war <double> , dzięki czemu możemy obsługiwać zmienne typu double. Następuje konkretyzacja typu ogólnego i nasza klasa przyjmuje postać:

```
class war
{    double n;
  public:
    war ( ) : n ( 0 )  { }       // konstruktor
    void set_war ( ) { cin >> n ; }
    void pokaz ( )     { cout << n << endl ; }
};
```

W pokazanym programie wykonaliśmy dwie konkretyzacje klasy szablonu dla typów char i double. Jeżeli chcemy zdefiniować jakąś funkcję poza szablonem klasy (np. w naszym przypadku funkcję pokaz()) musimy użyć następującej deklaracji i definicji tej funkcji (pokazano także szablon funkcji):

```
template <class T>
class war
{    T n;                          // typ ogolny T
  public:
    war ( ) : n ( 0 )  { }          // konstruktor
```

```
      void set_war();
      void pokaz();
   };


   template <class T>
   void  war < T > :: set_war()
     {  cin >> n  ;  }

   template <class T>
   void  war < T > :: pokaz()
     {  cout << n << endl;  }
```

Bardzo często stosujemy szablony klas do stworzenia tzw. klasy kontenerowej (ang. container class). Klasa kontenera to taka klasa, która definiuje kolekcję obiektów. Do obsługi tablic, list powiązanych czy stosów wykorzystujemy klasy kontenerowe, ponieważ możemy napisać uniwersalną aplikacje obsługująca różne typy danych. W kolejnym przykładzie pokażemy klasę kontenera do obsługi tablic.

Listing 12.21. Użycie wzorców klas – klasa kontenera (tablica)

```
1  #include <iostream>
   #include <conio.h>
3  using namespace std;

5  template <class T>
   class tablica
7  {    int rozmiar;
        T *tab;
9    public:
        tablica (int r = 1)
11        { rozmiar = r;
            tab = new T [r];
13        }
        void init_tab();
15      void pokaz_tab();
        ~tablica() { delete [ ] tab ; }
17 };

19 template <class T>
   void tablica< T > :: init_tab()
21   { for ( int i = 0; i < rozmiar; i++)
          { cout << " dane : " ;
23          cin >> tab[i] ;
          }
25   }

27 template <class T>
   void tablica< T > :: pokaz_tab()
```

```
29      { for (int i = 0; i < rozmiar; i++)
                cout << tab[i] << "  " ;
31        cout << endl;
        }
33

35
  int main()
37 { tablica<int> t1(4);
    cout << " podaj liczby calkowite : " << endl;
39  t1.init_tab();
    cout << " elementy tablicy :  ";
41  t1.pokaz_tab();

43  tablica<char> t2(5);
    cout << " podaj znaki : " << endl;
45  t2.init_tab();
    cout << " elementy tablicy :  " ;
47  t2.pokaz_tab();

49  getche();
    return 0;
51 }
```

Po uruchomieniu tego programu mamy wynik:

```
podaj liczby calkowite :
dane : 3
dane : 5
dane : 8
dane : 2
elementy tablicy :  3  5  8  2
podaj znaki :
dane : w
dane : a
dane : c
dane : e
dane : k
elementy tablicy : w  a  c  e  k
```

Szablon klasy dla ogólnej klasy tablica ma postać:

```
template <class T>
class tablica
{   int rozmiar;
    T *tab;
  public:
    tablica (int r = 1)
      { rozmiar = r;
        tab = new T [r];
```

```
        }
        void init_tab();
        void pokaz_tab();
        ~tablica() { delete [ ] tab ; }
};
```

W szablonie klasy mamy zadeklarowany wymiar tablicy - zmienna rozmiar oraz wskaźnik do tablicy tab, w momencie konkretyzacji parametr T będzie zamieniany na typ obsługiwanej klasy (w naszym przypadku int i char). Mamy również konstruktor, który przydziela pamięć dynamiczną dla tablicy dla żądanej ilości elementów także destruktor do zwalniania pamięci. W instrukcji:

```
tablica<int> t1(4);
```

konkretyzujemy tablicę o nazwie t1 aby mogła obsługiwać elementy typu int, rezerwujemy pamięć dla czterech elementów typu int. Podobnie w instrukcji:

```
tablica<char> t2(5);
```

definiowana jest tablica o nazwie t2 i przydzielana jest pamięć dla 5 obiektów typu char.

W klasie tablica zadeklarowane są dwie funkcje składowe: init_tab() oraz pokaz_tab().

Funkcja init_tab() służy do wprowadzania wartości elementów tablicy z klawiatury.

```
template <class T>
void tablica< T > :: init_tab()
  { for ( int i = 0; i < rozmiar; i++)
        { cout << " dane : " ;
          cin >> tab[i] ;
        }
}
```

Jak widać z nagłówka tej funkcji:

```
template <class T>
void tablica< T > :: init_tab()
```

jest ona metodą klasy tablica (wchodzi w skład szablonu klasy), zdefiniowana jest na zewnątrz klasy (klasa ma tylko jeden parametr typu), nie zwraca żadnej wartości. Funkcja ta znajduje się w zasięgu egzemplarza klasy tablica, zdefiniowana jest z typem ogólnym T.

Funkcja pokaz_tab() służy do wyświetlania elementów tablicy:

```
template <class T>
void tablica< T > :: pokaz_tab()
    { for (int i = 0; i < rozmiar; i++)
            cout << tab[i] << "  " ;
      cout << endl;
    }
```

Niejawnie często zakładamy, że przygotowany algorytm będzie działał tak samo dla różnych typów danych, co zresztą legło u podstaw tworzenia szablonów funkcji i klas. Życie programisty jest jednak bardziej skomplikowane niż to się wydaje. Niekiedy okazuje się, że program nie jest w stanie korzystając z jednego szablonu klasy obsłużyć wszystkich danych jakie będzie chciał wprowadzić użytkownik.

Aby powstał bardziej uniwersalny program musimy uwzględnić przypadek szczególny, najczęściej dopisać nową klasę. Język C++ udostępnia technikę specjalizacji (ang. specialization), dzięki której możemy konkretny typ danych obsługiwać w sposób specjalny.

Rozważmy ponownie opisany powyżej program w którym obsługujemy tablice różnych typów danych. Program działa poprawnie na prostych typach (int, double, char, itp.). W programie mamy szablon klasy tablica. Klasa szablonu zawiera konstruktor, destruktor, oraz metody init_tab() oraz pokaz_tab(). Konstruktor przydziela dynamicznie danym wejściowym przestrzeń na stercie. Destruktor zwalnia pamięć przydzieloną na stercie. Przy pomocy metody init_tab() wprowadzamy elementy tablicy żądanego typu z klawiatury, metoda pokaz_tab() powoduje wyświetlenie elementów tablicy na ekranie. Kod klienta tworzy obiekty klasy tablica dla wybranych typów danych, w naszym przykładzie mamy następujące obiekty:

```
tablica<int> t1(4);
```

oraz

```
tablica<char> t2(5);
```

Obsługa danych typu int i typu char jest poprawna. Gdy użytkownik spróbuje obsłużyć napis (tablicę znakową) w postaci:

```
tablica<char*> t3(5);
```

pojawi się problem, program nie uruchomi się gdy napisy wprowadzać będziemy z klawiatury. Musimy na nowo opracować klasę obsługującą tablicę znakową, w praktyce oznacza to dopisanie klasy specjalnej, korzystając z mechanizmu specjalizacji jawnej (ang. explicit specialization). W programie musi być zarówno szablon klasy ogólny oraz jego specjalizacja. Nie można

umieszczać w programie samej specjalizacji szablonu bez szablonu klasy ogólnej. Specjalizacja szablonu jest konkretyzowana przy użyciu tej samej składni, co w przypadku obiektu klasy szablonu. Jawna specjalizacja to definicja konkretnego typu lub typów, które mają zostać użyte zamiast szablonu ogólnego. Załóżmy, że mamy zdefiniowany szablon klasy tablica :

```
template <class T>   // Klasa szablonu
class tablica
{    int rozmiar;
     T *tab;             // tablica danych na stercie
  public:
     tablica (int r = 1) //konstruktor przydziela pamiec
                         //na stercie
       { rozmiar = r;
         tab = new T [r];
       }
     void init_tab();   // metoda do wprowadzania
                        //elementow tablicy
     void pokaz_tab(); // metoda do wyswietlenia
                        //elementow tablicy
     ~tablica() { delete [ ] tab ; }
};
```

Wiemy, że potrzebujemy innej definicji klasy do obsługi tablicy znaków. Specjalizacja klasy tablica może mieć postać:

```
template <>                    // pusta lista szablonu
class tablica <char *>         // typ specjalizaci - napisy
{     int rozmiar;             // rozmiar tablicy
      char * *tab;             // tablica napisow na stercie
  public:
      tablica (int r = 1)
        { rozmiar = r;
          tab = new (char*[r]);
        }
      void init_tab();
      void pokaz_tab();
      ~tablica() {delete [] tab; }
 };
```

Składnia opisu specjalizacji jest połączeniem składni dla samych szablonów (z listą parametrów szablonu) oraz inicjalizacji szablonu w kodzie klienta (z listą typów rzeczywistych). Jeżeli nagłówek szablonu klasy ogólnej ma postać:

```
template <class T>          //Klasa szablonu
class tablica
{  ......
```

to nagłówek specjalizacji szablonu klasy może mieć postać:

```
template <>                    //pusta lista szablonu
class tablica <char *>         //typ specjalizaci - napisy
{
```

W kodzie klienta specjalizowany obiekt szablonu ma postać:

```
tablica<char*> t2(5);      //obiekt tablicy specjalizowany
```

Na kolejnym listingu pokazano program zawierający szablon klasy tablica oraz jego specjalizację dla danych typu tablicy znakowej.

Listing 12.22. Specjalizacja klasy szblonu – klasa kontenera (tablica)

```
1  #include <iostream>
   #include <conio.h>
3  #include <string.h>
   using namespace std;
5
   template <class T>             // Klasa szablonu
7  class tablica
   {   int rozmiar;
9      T *tab;                    // tablica danych na stercie
     public:
11     tablica (int r = 1)        // konstruktor przydziela
                                  //pamiec na stercie
13       { rozmiar = r;
           tab = new T [r];
15       }
       void init_tab();         // metoda do wprowadzania
17                              //elementow tablicy
       void pokaz_tab();        // metoda do wyswietlenia
19                              //elementow tablicy
       ~tablica() { delete [ ] tab ; }
21 };

23 template <class T>
   void tablica< T > :: init_tab()
25   { cout << " dane : " << endl;
       for ( int i = 0; i < rozmiar; i++)
27                cin >> tab[i] ;
     }
29
   template <class T>
31 void tablica< T > :: pokaz_tab()
       { for (int i = 0; i < rozmiar; i++)
33             cout << tab[i] << "  " ;
         cout << endl;
35     }
   //------------------specjalizacja--------------------
```

```
37 template <>                    // pusta lista szablonu
   class tablica <char *>       // typ specjalizacji - napisy
39 {     int rozmiar;             // rozmiar tablicy
         char * *tab;             // tablica napisow na stercie
41   public:
         tablica (int r = 1)
43         { rozmiar = r;
             tab = new (char*[r]);
45         }
         void init_tab();
47       void init_tab1();
         void pokaz_tab();
49       ~tablica() {delete [ ] tab; }
   };
51
   void tablica <char*> :: init_tab()
53   {  cout << " dane : " << endl;
        for ( int i = 0; i < rozmiar; i++)
55         { tab[i]= new (char[20]);
              cin >> tab[i];
57         }
     }
59
   void tablica <char*> :: pokaz_tab()
61   { for ( int i = 0; i < rozmiar; i++)
           cout << tab[i] << endl;
63   }
   //--------------koniec specjalizacji -----------------
65
   int main()
67 { int ile = 1;
     cout << "ile elementow tablicy : ";
69   cin >> ile;
     tablica<int> t1(ile); //obiekt tablicy int
71   cout << "liczby calkowite, ilosc = "<< ile << endl;
     t1.init_tab();
73   cout << " elementy tablicy :  ";
     t1.pokaz_tab();
75   cout << "ile elementow tablicy : ";
     cin >> ile;
77   tablica<char*> t2(ile); //obiekt tablicy specjalizowany
     cout << "napisy, ilosc = " << ile << endl;
79   t2.init_tab();
     cout << "elementy tablicy : " << endl;
81   t2.pokaz_tab();
     cout << "ile elementow tablicy : ";
83   cin >> ile;
     tablica<char> t3(ile); //obiekt tablicy char
85   cout << "znaki, ilosc = " << ile << endl;
     t3.init_tab();
87   cout << " elementy tablicy :  " ;
```

```
   t3.pokaz_tab();
89
   getche();
91 return 0;
 }
```

Po uruchomieniu tego programu możemy mieć następujący wynik:

```
ile elementow tablicy : 3
liczby calkowiet , ilosc = 3
 dane :
1 2 3
elementy tablicy : 1 2 3
ile elementow tablicy : 3
napisy , ilosc = 3
  dane :
Ala Ola Lola
elementy tablicy :
Ala
Ola
Lola
ile elementow tablicy : 5
znaki , ilosc = 5
dane :
W a c e k
 elementy tablicy : W a c e k
```

W naszym programie specjalizacja szablonu klasy została specjalnie opracowana aby poprawnie obsługiwać tablicę znakową. W szablonie klasy ogólnej deklaracja tablicy danych oraz konstruktor mają postać :

```
template <class T>       // Klasa szablonu
class tablica
{   int rozmiar;
    T *tab;              // tablica danych na stercie
  public:
    tablica (int r = 1) // konstruktor przydziela
                         //pamiec na stercie
      { rozmiar = r;
        tab = new T [r];
      }
       .....................
```

Specjalizacja klasy ma postać:

```
template <>                // pusta lista szablonu
class tablica <char *>     // typ specjalizacji - napisy
{     int rozmiar;         // rozmiar tablicy
      char * *tab;         // tablica napisow na stercie
  public:
```

```
     tablica (int r = 1)
       { rozmiar = r;
         tab = new (char*[r]);
       }
    .......................
```

Mamy także różnie zdefiniowane metody init_tab(). W klasie ogólnej mamy:

```
template <class T>
void tablica< T > :: init_tab()
  { cout << " dane : " << endl;
    for ( int i = 0; i < rozmiar; i++)
               cin >> tab[i] ;
  }
```

Metoda init_tab() w specjalizacji klasy ma postać:

```
void tablica <char*> :: init_tab()
  {  cout << " dane : " << endl;
     for ( int i = 0; i < rozmiar; i++)
       { tab[i]= new (char[20]);
          cin >> tab[i];
       }
  }
```

Pokazana metoda obsługuje dynamiczna tablicę wskaźników na napisy stałej długości. Możliwa jest modyfikacja tej metody tak aby obsługiwała dynamiczna tablicę wskaźników na napisy zmiennej długości :

```
void tablica <char*> :: init_tab()
  { char temp[50];
    tab = new char*[rozmiar];
    for ( int i = 0; i < rozmiar; i++)
       {cout << " dane : " ;
         cin >> temp;
         tab[i]= new (char[strlen(temp) + 1]);
         strcpy(tab[i],temp);
       }
}
```

Zwolnienia pamięci zajmowanej przez napisy należy wykonać osobno dla każdego napisu według adresu przechowywanego w elemencie tablicy wskaźników, a następnie zwolnić pamięć przydzieloną na tablice wskaźników:

```
for (i = 0; i < rozmiar; i++)
      delete [ ] tab[i];
delete [ ] Tb;
```

Słownik angielsko-polski

	A
abstrakt data type (ADT)	abstrakcyjny typ danych, zazwyczaj tym terminem określamy typ danych zdefiniowany przez programistę, typowym przykładem jest klasa – jest to formalnie typ danych, proces opisywania funkcji klasy niezależnie od jej implementacji, nazywamy abstrakcją danych
access specyfication	specyfikacja dostępu, etykiety private, public oraz protected regulują dostęp do składowych klasy (danych i metod)
accessor function	funkcje dostępu, są to funkcje (z wyjątkiem konstruktorów), które maja dostęp do prywatnych (private) danych klasy
address	adres, jest to liczba wskazująca miejsce w pamięci,
allocate	alokacja, przydział (pamięci)
application	aplikacja, program komputerowy, traktowany przez użytkownika jako jednostka
argument	argument, wartość przekazywana do funkcji
assertion	asercja, instrukcja mówiąca, że element umieszczony w danym miejscu w programie musi być prawdziwy

B

base class	klasa bazowa, jest to klasy na podstawie której, tworzona jest klasa pochodna, wykorzystująca mechanizm dziedziczenia
binding	wiązanie, interpretacja wywołania funkcji w kodzie źródłowym w celu wykonania kodu właściwej funkcji
bit field	pole bitowe
boolean	wbudowany typ logiczny, wartości true lub false
bug	błąd w kodzie, nieoczekiwana sytuacja, termin wprowadzony przez dr Grace Hopper, która była kontradmirałem w U.S.Navy

C

call	wywołać, wywołanie
call by reference	proces deklarowania parametru funkcji jako zmiennej wskaźnikowej, w konsekwencji przekazanie adresu jako argument
call by value	proces deklarowania parametru funkcji jako zmiennej prostej (nie wskaźnikowej), w konsekwencji przekazanie wartości zmiennej jako argument
calling constructors	wywoływanie konstruktora, konstruktor jest wywoływany zawsze, gdy obiekt jest kreowany, jest wiele sposobów wywoływania konstruktora
calling function	wywołanie funkcji, przekazanie sterowania do funkcji, która wykona żądane operacje
cast	dosłowna, wymuszona, konwersja typów, np. (double)i
cast operator	operator konwersji, służy do wymuszenia konwersji, np. zapis int (x*y) powoduje, że wartość wyrażenia x*y jest konwertowana do wartości typu int.

child class	klasa potomna, pochodna klasa utworzona z klasy bazowej
cin	obiekt cin jest wykorzystywany do wprowadzania danych, definicja obiektu cin jest zawarta w pliku iostream, plik iostream opisuje strumień wejściowy, operator » wykorzystywany jest do pobrania znaków ze strumienia wejściowego
class	klasa, typ zdefiniowany przez programistę, kluczowy element programowania obiektowego
access specifier	specyfikator dostępu (private, public, protected)
access specifier	konstruktor klasy, jest to metoda klasy, służy wyłącznie do tworzenia nowego obiektu klasy i przypisywania wartości jego składowym, nazwa konstruktora jest identyczna z nazwą klasy, konstruktor nie posiada typu zwracanego
declaration	deklaracja klasy, opisuje komponenty klasy: dane i metody,
destructor	destruktor, jest to specjalna funkcja klasy, nazwa destruktora jest taka sama jak nazwa klasy, poprzedzona jest znakiem tyldy, destruktor jest automatycznie wywoływany, gdy obiekt jest niszczony
implementation	implementacja klasy, termin oznacza definicje metod klasy, tzn. kod funkcji składowych klasy
inheritance	dziedziczenie, jest to cecha języka, pozwala na tworzenie nowej klasy z klasy już istniejącej.
instance variables	(data members), dane klasy, pola klasy, zmienne klasy

library	biblioteka klas, jest to biblioteka testowanych i pozbawionych błędów klas, z ich interfejsami i implementacjami, zazwyczaj interfejs jest umieszczany w pliku nagłówkowym, a implementacja jest umieszczana w oddzielnym pliku z implementacjami (implementation file)
members	członkowie klasy, zmienne i funkcje wyszczególnione sekcji deklaracji klasy
methods	metody, funkcje składowe klasy
scope	zasięg klasy, generalnie zasięg (scope) oznacza zakres widoczności nazwy w obrębie pliku, dane składowe i metody klasy cechuje zasięg klasy, te elementy są znane i widoczne w obrębie klasy , poza nią są niewidoczne
collating sequence	porządek sortowania
command line	wiersz poleceń
compiler error	błąd kompilacji
compile-time errors	błędy wykryte w fazie kompilacji, (błędy syntaktyczne)
concatenate	połączenie, sklejenie wielu łańcuchów w jeden łańcuch
console application	proces tworzenia prostych programów na platformach programistycznych z pominięciem okienek
constructor	konstruktor klasy, jest to metoda klasy, służy wyłącznie do tworzenia nowego obiektu klasy i przypisywania wartości jego składowym, nazwa konstruktora jest identyczna z nazwą klasy, konstruktor nie posiada typu zwracanego
base class	konstruktor klasy bazowej

copy	konstruktor kopiujący wykorzystywany jest do kopiowania istniejącego obiektu do nowo utworzonego obiektu, gdy klasa nie zawiera wskaźnikowych danych, zazwyczaj nie ma problemów z konstruktorem kopiującym (nie są generowane błędne wyniki), gdy w klasie występują zmienne wskaźnikowe, należy opracować własny konstruktor kopiujący
conversion	konstruktor przekształcający, jest to konstruktor jednoargumentowy, który umożliwia przekształcenie obiektu jednego typu (włącznie z typami wbudowanymi) na obiekt danej klasy
default	konstruktor domyślny, jeżeli programista nie dostarczy własnego konstruktora klasy, kompilator dostarczy konstruktor domyślny
inline	konstruktor typu inline
overloaded	konstruktor przeciążony
containers	kontenery, zasobniki (podstawowe elementy STL), kontener jest to obiekt, który może przechowywać inne obiekty
conversion automatic	proces ujednolicania typów w wyrażeniach mieszanych
cout	obiekt cout jest wykorzystywany do wyprowadzania danych, definicja obiektu cout jest zawarta w pliku iostream, plik iostream opisuje strumień wyjściowy, operator « wykorzystywany jest do wstawiania znaków do strumienia wyjściowego
	D
debugging	debugowanie, proces usuwania błędów w programach komputerowych
declaration	deklaracja, specyfikowanie typów

definition	definicja, jest to deklaracja alokująca pamięć
dynamic allocation	dynamiczne przydzielanie pamięci jest procesem zachodzącym w czasie wykonywania program w odróżnieniu od statycznego procesu przydzielania pamięci w fazie kompilacji
delete operator	operator zwalniania pamięci, zwalnia zarezerwowaną pamięć
derived class	klasa pochodna, klasa potomna, subklasa, klasa utworzona z innej klasy (klasy bazowej) dzięki mechanizmowi dziedziczenia
destructor	destruktor, jest to specjalna funkcja klasy, nazwa destruktora jest taka sama jak nazwa klasy, poprzedzona jest znakiem tyldy, destruktor jest automatycznie wywoływany, gdy obiekt jest niszczony
dynamic binding	wiązanie dynamiczne, wiązanie późne, wybór odpowiedniej funkcji w fazie wykonania programu
dynamic memory allocation	dynamiczny przydział pamięci, przydział pamięci w fazie wykonania programu, wykorzystywane są operatory new i delete
	E
efficiency	wydajność, wydajność programu oznacza zadawalającą szybkość wykonania, mając do dyspozycji określone zasoby
encapsulation	enkapsulacja, hermetyzacja, ukrywanie danych i metod, jest to podstawowa cecha programowania obiektowego
equality expression	wyrażenie relacyjne równości
errors	błędy
compile-time	błędy wykryte w czasie kompilacji
logic	błędy logiczne
programming	błędy programistyczne

run-time	błędy wykryte w czasie wykonywania programu
syntax	błędy syntaktyczne
typos	błędy typograficzne
evaluate	ocena, wyznaczanie wartości
exit	wyjście, najczęściej jest to zakończenie wykonywania programu lub jakiejś jego części
executable program	program wykonywalny
exponential notation	notacja wykładnicza, jest to sposób zapisywania liczb (np. w notacji wykładniczej zapis 1.625e3 oznacza liczbę dziesiętną 1625, a w notacji naukowej liczbę 1.625x103)
expression	wyrażenie
extern	typ zewnętrznej klasy pamięci, gdy zmienna jest deklarowana poza funkcją (zmienna globalna), przydziela sie jej pamięć klasy extern
external storage class	zewnętrzna klasa pamięci
extraction operator	operator pobierania (symbol »), pobiera znaki ze strumienia wejściowego
	F
field	pole, składnik struktury
file	plik, jest to obszar, w którym przechowywane są dane
FIFO	first in, first out, pierwszy na wejściu, pierwszy na wyjściu
flag	znacznik, flaga, różnorodne znaczniki formatu, określają rodzaj formatowania wykonywanego w czasie operacji strumieniowych wejścia/wyjścia, funkcje składowe setf, unsetf oraz flags kontrolują ustawienia
flow of control	przebieg sterowania
flush	opróżnić (bufor)
formal parameter	parameter formalny

forward declaration	deklaracja wyprzedzająca, deklaracja zapowiadająca, tej konstrukcji używa się w przypadku, gdy w klasie następuje odwołanie do innej, niezadeklarowanej klasy
friend function	funkcja zaprzyjaźniona, funkcja nie będąca metodą klasy, której przyznano prawo dostępu do danych i metod prywatnych i chronionych
function	funkcja, nazwany, wydzielony fragment programu, logiczna jednostka przetwarzania
function	funkcja
argument list	lista parametrów (argumentów)
arguments structure	parametry typu struktury przekazywane do funkcji
array arguments	argument tablicowe
call by reference	wywołanie funkcji z argumentem typu wskaźnikowego
call by value	wywołanie funkcji z argumentem prostym (nie wskaźnikowym)
formal parameter	parametry formalne
friend	funkcja zaprzyjaźniona
header	nagłówek funkcji
inline	funkcja inline, funkcja wbudowana, funkcja wplatana, są to krótkie funkcje, ich wystąpienia w programie są zastępowane ich kodem (mechanizm bardzo podobny do tworzenia makr)
pointer parameter	parameter typu wskaźnikowego przekazywany do funkcji
type specifier	specyfikator typu zwracanej wartości funkcji
virtual	funkcja wirtualna, jest to metoda (funkcja składowa) zadeklarowana w klasie bazowej i zdefiniowana w klasie pochodnej
function binding	wiązanie funkcji, mamy wiązanie statyczne (aktywne w czasie kompilacji) oraz dynamiczne (aktywne w fazie wykonania program)

function template	wzorzec funkcji, szablon funkcji, funkcja ogólna, jest to zdefiniowana kompletna funkcja, która jest wzorcem (modelem) dla rodziny funkcji, w definicji wykorzystuje się symboliczny typ, który podczas wywoływania funkcji przekształcany jest na rzeczywisty typ danych
	H
header file	plik nagłówkowy (dołączany do program)
heap	stos, sterta (typ pamięci)
hexadecimal	szesnastkowy (system liczbowy)
high level languages	język programowania wysokiego poziomu
high-order bit	bit najbardziej znaczący
	I
IDE	Integrated Development Environment, zintegrowane środowisko programistyczne
implementation	implementacja, pisanie i testowanie program, także – kod programu
indirect addressing	adresowanie pośrednie, jest to procedura otrzymywania wartości zmiennej na która wskazuje wskaźnik
indirection operator	operator dereferencji, operator wyłuskania (symbol *), zapis *p oznacza: "zmienna, której adres jest przechowywany w p"
inheritance	dziedziczenie, proces kreowania klasy przez wykorzystanie już istniejącej klasy, istniejąca klasa nazywana jest klasą bazową, nowa klas nosi nazwę klasy pochodnej
initialization	inicjalizacja
array	inicjalizacja tablicy
pointer	inicjalizacja wskaźnika
string	inicjalizacja łańcucha
structure	inicjalizacja struktury

variable	inicjalizacja zmiennej
inline declaration	deklaracja funkcji wbudowanej (inline)
inline function	funkcja wbudowana, funkcja rozwijalna
input/output	wejście/wyjście, dane wejściowe są wprowadzane do programu, dany wyjściowe są wytwarzanie w procesie przetwarzania
input/output stream	strumienie wejścia/wyjścia, strumień to urządzenie logiczne produkujące lub pobierające informacje
insertion operator	operator wstawiania (symbol «) potrzebny jest do wyświetlania danych typów wbudowanych, sam operator wstawia znaki do strumienia
instance variables	dane klasy (data members), członek klasy
integer overflow	przepełnienie, w czasie wykonania program może wystąpić przekroczenie zakresu np. dla liczb całkowitych, można otrzymać niepoprawny wynik
interface	interfejs, zbiór deklaracji, potrzebnych do wywoływania funkcji
invariant	niezmiennik, element, który w danym punkcie programu musi być prawdziwy
item	pozycja, składnik, egzemplarz
iteration	iteracja, wielokrotne wykonywanie tych samych instrukcji

L

languages	języki (programowania)
high-level	języki programowania wysokiego poziomu
low-level	języki programowania niskiego poziomu
middle-level	języki programowania średniego poziomu

object-oriented	języki programowania zorientowane obiektowo
procedure-oriented	proceduralne języki programowania
library	biblioteka, kolekcja plików, pliki zawierają typy, funkcje, struktury, dyrektywy, itp., które są wykorzystywane w wielu programach
LIFO	last in, first out, ostatni na wejściu, pierwszy na wyjściu
linker	konsolidator, program łączący, jest to program który konsoliduje inne pliki i biblioteki z programem obiektowym, w ten sposób otrzymujemy program wykonywalny
logical expression	wyrażenie logiczne
low-order bit	bit najmniej znaczący
	M
machine accuracy	dokładność maszyny liczącej, w metodach numerycznych wygodnie jest zadeklarować eps („epsilon maszynowe”) jako najmniejszą dodatnią wartość zmiennej typu double spełniającą warunek, że wyrażenie $1.0 < 1.0 + eps$ jest prawdziwe
machine language	język maszynowy (wewnętrzny język komputerowy, składający się z serii zer i jedynek)
macro	makrodefinicji (zdefiniowany skrót, konstrukcje preprocesora)
manipulator	manipulator, jest to specjalna funkcja, pozwalająca na kontrolowanie formatu danych podczas operacji wejścia/wyjścia
mask	maska, jest to stała lub zmienna, która jest wykorzystywana do wyodrębnienia żądanego bitu w zmiennej, używana w operacjach bitowych
mathematical library	funkcje matematyczne zebrane w pliku (np. <math.h>
member	element struktury oraz klasy, składnik struktury oraz klasy

member function	funkcja składowa (metoda klasy)
memory dynamic allocation	dynamiczna alokacja pamięci, programista kontroluje przydział pamięci przy pomocy operatorów new i delete
methods	metody klasy, funkcje składowe, metoda operuje na danych składowych klasy
multi-dimensional arrays	wielowymiarowa tablica
multiple inheritance	wielo-bazowe dziedziczenie, tworzenie klasy pochodnej jednocześnie z dwóch lub więcej klas bazowych, należy korzystać z tego mechanizmu z umiarem
	N
namespaces	przestrzeń nazw, jest to koncepcja pozwalająca nadawać nazwy blokom sekcji w programie, dzięki tej koncepcji można stosować te same nazwy dla różnych zmiennych
new operator	operator new, służy do dynamicznego alokowania pamięci
null character	znak zera (symbol `\0`), używany np. do oznaczania końca łańcucha
null pointer value	wskaźnik zerowy, wskaźnik o wartości 0 i NULL nie wskazuje żadnej wartości
	O
object	obiekt, podstawowy element programowania obiektowego, modeluje elementy świata rzeczywistego, obiekty są tworzone przy pomocy odpowiednio zaprojektowanych klas
object program	program pośredni, jest to kod wyprodukowany z kodu źródłowego przez kompilator, potrzebny jest jeszcze konsolidator aby wyprodukować kod wykonywalny
one-dimensional array	tablica jednowymiarowa
operator	operator

addition	operator dodawania
address	aperator adresowy
arithmetic	operator arytmetyczny
assignment	operator przypisania
associativity	łączenie operatora (z lewej do prawej lub z prawej do lewej)
bitwise	operator bitowy
bitwise exclusive or	operator alternatywy bitowej (symbol |)
cast (type)	operator rzutowania jawnego
comma	operator przecinkowy
conditional	operator warunkowy (symbol (?:)
decrement	operator dekrementacji
delete	operator zwalniania pamieci
dereferencing	operator dereferencji (wyłuskania)
division	operator dzielenia
equality	operator relacyjny (porównywania)
equals	operator równości
greater than	operator relacyjny większy niż
greater than or equal to	operator relacyjny większy niż lub równy
increment	operator inkrementacji
left shift	operator przesunięcia bitowego
less than	operator relacyjny mniejszy niż
less than or equal to	operator relacyjny mniejszy niż lub równy
logical	operator logiczny
logical and	operator logiczny koniunkcji (symbol &&)
logical negation	operator logiczny zaprzeczenia
logical or	operator logiczny alternatywy (symbol ||)
modulus	operator modulo (symbol %)
multiplication	operator mnożenia
not equals	operator relacyjny nierówny
one's complement	operator bitowy dopełnienia
precedence	priorytet operatora (określa kolejność wykonywania operacji)
relational	operator relacyjny
right shift	operator bitowy przesunięcia w prawo
sizeof	operator rozmiaru

structure member	operator dostępu do elementu struktury (symbol .)
structure pointer	operator dostępu do element struktury przez wykorzystanie wskaźnika (symbol − >)
subtraction	operator odejmowania
unary minus	jednoargumentowy operator minus
ostream	jest to obsługująca operacje wyjścia klasa pochodna klasy ios
output	wyjście
overflow	przepełnienie (kategoria błędu)
overloaded function	funkcja przeciążona, w języku C++ ta sama nazwa funkcji może być użyta do różnych funkcji, pod warunkiem, że mają one różne listy argumentów
overloaded operators	przeciążanie operatorów, jest to technika pozwalająca na zmianę sposobu działania operatorów języka C++
	P
parameter	parameter, argument
parent class	klasa bazowa, superklasa
parametrized manipulator	sparametryzowane manipulatory, manipulatory strumieniowe są funkcjami formatującymi sposób wyprowadzania danych a także wykonują pewne operacje na danych wejściowych
pass	przekazać (np. argumenty)
pass by reference	przekazanie argument przez referencję
pass by value	przekazanie argumentu przez wartość
passing addresses	przekazanie argumentu w postaci adresu,
passing arrays	przekazanie tablicy
passing file names	przekazanie nazwy pliku
passing structures	przekazanie struktury
pointer	wskaźnik

argument	argument wskaźnikowy (np. przekazywanie wartości do funkcji)
arithmetic	arytmetyka wskaźnikowa, specjalny sposób manipulowania wskaźnikami (np. inkrementacja)
array	tablica wskaźników
assignment	przypisanie wskaźnika (nadawanie wartości zmiennej wskaźnikowej)
class members	wskaźnikowa składowa klasy
constant	stała wskaźnikowa
declaration	deklaracja wskaźnika
indirection	wskazanie pośrednie, dereferencja zmiennej wskaźnikowej
initialization	inicjalizacja wskaźnika
parameter	parametr wskaźnikowy
structure memebers	wskaźnikowa składowa struktury
this	wskaźnik this, każdy obiekt ma dostęp do swego adresu przy pomocy wskaźnika this, wskaźnik this jest przekazywany do obiektu jako niejawny pierwszy argument każdej niestatecznej metody wywoływanej na rzecz obiektu (metody nie są kopiowane, są wspólne dla wszystkich obiektów)
polymorphism	polimorfizm (wielopostaciowość), podstawowy paradygmat programowania obiektowego, dzięki zastosowaniu funkcji wirtualnych i polimorfizmu możliwe jest projektowanie systemu, który jest prosto rozszerzalny, można pisać program komputerowy w sposób uniwersalny,
private access rights	specyfikator dostęp do danych i metod typu private (tylko metody mają dostęp do danych)
program flow	przebieg programu
promote	awansować (typ)

protected access rights	specyfikator dostęp do danych i metod typu protected , do składowych klasy bazowej zadeklarowanych jako protected dostęp mają jedynie jej metody oraz funkcje zaprzyjaźnione, a także metody oraz funkcje zaprzyjaźnione z klasy pochodnej,
public access rights	specyfikator dostęp do danych i metod typu public, dane i metody są dostępne dla wszystkich elementów programu (nie są chronione)
pseudo random number generator	generator liczb pseudolosowych
push	położyć (na stosie)
	R
random file access	bezpośredni dostęp do pliku (każdy znak w pliku może być odczytany bezpośrednio)
random number generator	generator liczb losowych
recursion	rekurencja, proces występuje, gdy funkcja wywołuje sama siebie
register	słowo kluczowe, klasa pamięci register (rejestr)
relational expression	wyrażenie relacyjne
reserved word (keyword)	słowo zastrzeżone (kluczowe)
round off error	błąd obcinania
run-time	czas wykonania (czas wykonywania program)
run-time terror	błąd wykonania programu
	S

STL	Standardowa Biblioteka Wzorców (ang. Standard Template Library), koncepcja STL została opracowana przez Alexandra Stepanowa i Menga Lee z Hewlett Packard, biblioteka wykorzystuje paradygmat programowania ogólnego oparta jest na trzech składnikach: zasobnikach (kontenery), iteratorach i algorytmach, kontenery są uniwersalnymi wzorcowymi strukturami danych,
scope	zakres widoczności (zmiennych)
sequential access	dostęp sekwencyjny
shift	przesunięcie
side effect	efekt uboczny
significant figures	cyfry znaczące (zapis liczb zmiennoprzecinkowych)
simple inheritance	dziedziczenie proste
stack	stos
stack operation	operacja na stercie (typ pamięci)
standard library	biblioteka standardowa
standard template library	Standardowa Biblioteka Wzorców
statement	instrukcja
static	typ klasy pamięci, statyczna
static Winding	wiązanie statyczne (wczesne),
storage allocation	przydział pamięci
storage class	klasa pamięci
auto	klasa pamięci typu auto
extern	klasa pamięci typu ex tern (zewnętrzna)
register	klasa pamięci typy register (rejestrowa)
static	klasa pamięci typu static (statyczna)
static external	klasa pamięci typu static , zewnętrzna
string	łańcuch, napis, w żargonie - string
array of character	tablica znaków, napis, łańcuch
constant	stała łańcuchowa
end-of-string character	znacznik końca łańcucha (symbol `\0`)
initialization	inicjalizacja łańcucha

library function	biblioteka funkcji obsługujących napisy
standard function	standardowa funkcja obsługi łańcucha (biblioteka string.h)
struct	słowo kluczowe typy danych (struktura)
subclass	klasa pochodna
superclass	klasa bazowa (nadklasa)
syntax	syntaks
syntax error	błąd syntaktyczny
	T
template prefix	deklaracja wzorca, napis informuje kompilator, że funkcja występująca za tym napisem (np. template <class T>) jest wzorcem (szablonem), który wykorzystuje uogólniony typ danych o nazwie T,
this pointer	wskaźnik this
top down	metoda "top down" jest jedną z technik programowania
truncate	obciąć (cyfry znaczące)
truth table	tablica, która pokazuje wynik działania funkcji logicznych (bool)
two-dimensional array	tablica dwuwymiarowa
type mismatch	niezgodność typów
type specifier	specyfikator typu
typedef	słowo kluczowe, z jego pomocą definiujemy własny typ
	U
unary operator	operator jednoargumentowy
union	unia, struktura danych
	V
variable	zmienna
virtual functions	funkcje wirtualne, dzięki funkcjom wirtualnym i polimorfizmowi możliwe jest projektowanie systemów, które są w prosty sposób rozszerzane
void	typ danych, słowo kluczowe,

Skorowidz

#define, 65
#include, 6

abort(), 237
abstrakcyjne funkcje, 194
abstrakcyjne klasy, 195
akcesory, 50
argumenty domyślne funkcji, 11
asercja, 238
assert(), 237
auto, 323

błędy, 230
bool, 23
break, 256
bufor, 231
buforowane wejście, 237
buforowane wyjście, 231

catch, 239
cerr, 150
chronione dane, 96
chronione metody, 44
cin, 5
class, 42
clear(), 237
const, 67
cout, 5

dane chronione, 44, 96
dane prywatne, 44, 96
dane publiczne, 44
default, 311
definicja funkcji, 53
deklaracja dostępu, 44
deklaracja funkcji, 33
deklaracja struktury, 31
deklaracja zapowiadająca, 135
delete, 75
destruktor, 58
dynamiczna alokacja pamięci, 318
dyrektywy preprocesora, 39
dziedziczenie, 84

egzemplarz, 58
endl, 44
exit, 231

flaga, 313
friend, 123
funkcje, 29
funkcje abstrakcyjne, 194
funkcje matematyczne, 317
funkcje operatorowe, 146
funkcje prywatne, 44
funkcje przeciążone, 13
funkcje publiczne, 44
funkcje składowe, 47
funkcje statyczne, 179
funkcje wirtualne, 178
funkcje zaprzyjaźnione, 122

getche, 124

ignore, 236, 237
indeks, 72
inicjalizacja tablic, 72
inline, 48

klasa, 32, 33
    abstrakcyjna, 195
    bazowa, 84
    ogólna, 293
    pochodna, 84
    wirtualna, 208
    zagnieżdżona, 215
    zaprzyjaźniona, 123
klasy zaprzyjaźnione, 141

kod, 16
kompilacja, 40
koniec łańcucha, 80
konstruktor, 58
konwersja, 59, 269
kwalifikator zakresu, 212

manipulatory, 167

namespace, 24
new, 72

obiekt, 42
operator alokacji dynamicznej, 75
operator przeciążony, 144

późne wiązanie, 191
pamięć, 20
polimorfizm, 178
private, 44
protected, 44
prywatne dane, 44
prywatne funkcje, 44
prywatne składowe, 44
prywatne zmienne, 44
przeciążanie konstruktorów, 272
przeciązanie funkcji, 13
przeciązanie operatorów, 144
przekazywanie argumentów przez referencję, 7, 129
przestrzeń nazw, 24
public, 44

referencje, 7
rekurencja, 322
rzutowanie, 207, 223

silnia, 235
sizeof, 146
specyfikator dostępu, 34
static, 60, 178
statyczne funkcje, 178
statyczne zmienne, 178
std, 24
stderr, 238
STL, 266
struct, 31
struktury, 31
strumienie, 4, 175

tablice, 71, 75
template, 266
this, 67
throw, 239
try, 239
typedef, 324
typename, 278

using, 24

wejście, 5
what, 260
wiązanie późne, 191
wiązanie wczesne, 191
wskaźnik, 7, 45
wskaźnik do funkcji, 191, 270
wskaźnik do tablic, 299
wyjątek, 239

zaprzyjaźnione funkcje, 122
zaprzyjaźnione klasy, 141
zmienne globalne, 178
zmienne referencyjne, 127
zmienne statyczne, 60
zmienne wskaźnikowe, 9
znak zerowy, 80
zwalnianie pamięci, 181