Spis treści

Rozdział 1

Programowanie obiektowe w języku C++

Rozdział ten poświęcony jest prezentacji podstawowych koncepcji programowania obiektowego w języku `C++`. Rozpoczyna go krótkie przedstawienie genezy powstania i historii rozwoju tego języka, kolejne podrozdziały obejmują omówienie niezbędnych podstaw teoretycznych ilustrowanych przykładami oraz zestaw ćwiczeń do samodzielnego wykonania.

1.1. Ogólne informacje o języku C++

Język `C++` powstał na bazie języka `C`, co sprawia, że w zakresie podstawowych elementów i konstrukcji językowych istnieje duże podobieństwo. Dlatego w niniejszym opracowaniu pominięte zostaną informacje o typach danych, instrukcjach sterujących, definicjach i deklaracjach funkcji oraz innych podstawowych elementach języka — zakłada się, że na tym etapie poznawania metod i technik programowania, czytelnikowi podstawy te są znane. Podrozdział ten prezentuje krótką historię rozwoju języka `C++` oraz informacje wprowadzające, niezbędne do zrozumienia mechanizmów programowania obiektowego w tym języku.

1.1.1. Krótka historia języka C++

Przełom lat 60–tych i 70–tych ubiegłego wieku to okres początków języka `C`. Powstał on właściwie trochę przez przypadek. Pracownicy laboratorium badawczego firmy telekomunikacyjnej Bell — Kenneth Thompson i Dennis Ritchie — poszukują sobie zajęcia po wycofaniu ich z projektu poświęconego realizacji systemu operacyjnego `MULTICS`. Ich uwaga koncentruje się na realizacji jądra i podstawowych narzędzi systemu, który wkrótce otrzymuje nazwę `UNIX`. Wkrótce również powstaje język `C`, stając się podstawowym językiem realizacji tego systemu [2]. Mimo iż język `C` wyraźnie nie pasuje do nurtu systematycznych i porządnych języków programowania typu `Algol` czy `Pascal`, zdobywa ogromną popularność i wchodzi na rynek języków programowania niejako bocznymi drzwiami, plasując się w grupie liderów.

Na początku lat 80–tych XX w. `C` posiada już ugruntowaną pozycję efektywnego języka programowania. W tym czasie Bjarnie Sroustrup, pracujący nad swoja rozprawą doktorską w Cambridge University, prowadzi badania symulacyjne związane z aplikacjami rozproszonymi. Testowany przez niego obiektowy język programowania `Simula` — idealny do symulacji pod względem funkcjonalnym — jest za mało efektywny. Stroustrup potrzebuje nowego, obiektowego języka programowania łączącego funkcjonalność języka `Simula` z efektywnością języka `C`. Po przeniesieniu się do Centrum Badań Komputerowych Laboratorium fir-

my Bell, Stroustrup projektuje nowy język programowania `C with Classes` [1], łączący w sobie cechy języków `Simula` i `C`. Wersja ta nie zawiera jeszcze znanych aktualnie z `C++` rozbudowanych mechanizmów obiektowych. W 1983 Rick Mascitti proponuje dla nowego języka nazwę `C++`, sugeruje ona ewolucyjna naturę `C++` w stosunku do języka `C`.

W połowie lat 80–tych język `C++` zostaje zaprezentowany publicznie z możliwością wykorzystania poza Laboratorium Bell. Powstaje pierwszy komercyjny kompilator `C++` — `cfront`. Jest to tak na prawdę translator, tłumaczący kod w języku `C++` na kod w języku `C`, przetwarzany następnie zwykłym kompilatorem języka `C`. Dopiero pod koniec lat osiemdziesiątych Michael Tiemann z firmy Cygnus Support, implementuje pierwszy „prawdziwy" kompilator `C++`, zwany `G++`. Wtedy też pojawia się pierwsza publikacja specyfikacji `C++` w postaci książki *The Annoutated Reference Manual* autorstwa Stroustrupa i Ellisa, znanej jako *ARM*. Rozpoczyna się proces standaryzacji języka.

W połowie lat 80–tych Komitet ANSI publikuje pierwszą wersję zarysu standardu `C++`. Rozpoczyna się uciążliwy proces rewizji standardu — wymaga on wielu poprawek. W grudniu 1997 następuje oficjalna premiera pełnej wersji standardu `ANSI C++`. Standard zostaje również zaaprobowany przez ISO. Następuje zamrożenie prac na pięć lat — okres na adaptację producentów kompilatorów i poprawę błędów. Obowiązuje standard ISO/IEC 14882:1998 (*Standard for the C++ Programming Language*) z drobnymi poprawkami zatwierdzonymi w 2003 r. (ISO/IEC 14882:2003). Standardy te są znane pod nazwami `C++98` i `C++03`. Następny standard, zwany roboczo `C++0x` miał wejść w życie w roku 2009, jednak pod koniec 2009 roku dostępna jest jedynie wersja robocza, należy się spodziewać pojawienia oficjalnego standardu w roku 2010, a na standard międzynarodowy być może przyjdzie poczekać do 2011 roku. Zatem znak 'x' w roboczej nazwie standardu `C++0x` zapewne będzie kolejną cyfrą szesnastkową — spodziewajmy się zatem standardu oznaczonego `C++0A` lub `C++0B`.

1.1.2. C++ a programowanie nieobiektowe

W klasyfikacji języków obiektowych `C++` określane jest językiem hybrydowym. Posiada wbudowane mechanizmy programowania obiektowego, lecz można ich wcale nie używać i programować w języku `C++` prawie tak, jakby to był język `C`[1]. Pewne operacje — uciążliwe zwłaszcza na początku nauki programowania — można wykonywać nawet wygodniej. Przykładem są operacje *wejścia–wyjścia*, wykonywane zwyczajowo w języku `C` z wykorzystaniem funkcji z biblioteki identyfikowanej plikiem nagłówkowym *stdio.h*. W bibliotekach języka `C++` zdefiniowano obiekty realizujące operacje wejścia–wyjścia w sposób prostszy od analogicznych operacji w bibliotekach języka `C`. Biblioteka standardowa języka `C++` oferuje predefiniowane obiekty obsługi strumieni programu:

[1] To, że język C++ odziedziczył większość mechanizmów języka C nie oznacza — jak twierdzą niektórzy — że język C jest podzbiorem języka C++. Można wskazać takie konstrukcje programowe, które są legalne w C i kompilują się poprawnie, a jednocześnie są nielegalne w C++. Autor tego rozdziału twierdzi zdecydowanie, że C i C++ to dwa różne języki programowania, posiadające swoja odrębną specyfikę, oczywiście posiadające również wiele oczywistych, wspólnych elementów.

— `cin` — strumień reprezentujący *standardowe wejście* programu, odpowiada strumieniowi `stdin` z `C`. Strumień `cin` odczytuje dane i zapisuje je do odpowiednich zmiennych programu.
— `cout` — strumień reprezentujący *standardowe wyjście* programu. Odpowiada strumieniowi `stdout` z `C`.
— `cerr` — *niebuforowany strumień wyjściowy błędów*. Odpowiada strumieniowi `stderr` z `C`.
— `clog` — *buforowany strumień wyjściowy błędów*. Odpowiada strumieniowi `stderr` z `C`.

W kolejnych podrozdziałach przedstawione zostaną krótkie przykłady realizacji podstawowych operacji na strumieniach wejścia–wyjścia z biblioteki standardowej języka `C++`.

1.1.3. Dostęp do obiektów biblioteki wejścia–wyjścia

Aby uzyskać dostęp do obiektów `cin` i `cout` należy obowiązkowo włączyć do kodu źródłowego odpowiedni plik nagłówkowy. Ten plik nosi nazwę *iostream.h* i umieszczony jest w standardowej, dla danego kompilatora, lokalizacji plików nagłówkowych. Włączenie odpowiedniego pliku nagłówkowego oraz wyprowadzenie przykładowej informacji do `cout` zrealizowane może być następująco:

```
#include <iostream.h>
. . .
cout << "Przykładowy napis";
. . .
```

W powyższym przykładzie wyprowadzanie danych odbywa się z wykorzystaniem operatora «, zwanego *wstawiaczem* (ang. *inserter*). Dane są wyprowadzane zgodnie z ich typem, istnieje możliwość formatowania postaci wyjściowej. Wraz z rozwojem języka `C++` pojawiła się nowa konwencja podawania nazw plików nagłówkowych — bez rozszerzeń:

```
#include <iostream>
```

Jednak przy takiej postaci odwołania do pliku nagłówkowego, próba odwołania się do obiektu `cout` zakończy się błędem kompilacji. Dzieje się tak, gdyż wraz z nowym sposobem podawania nazw plików nagłówkowych wprowadzono tzw. *przestrzenie nazw*[2]. Aby odwołać się do obiektu `cout` należy poprzedzić go nazwą właściwej dla niego przestrzeni `std::cout`, pisząc:

```
#include <iostream>
. . .
std::cout << "Przykładowy napis";
. . .
```

Posługiwanie się pełnym nazwami (nazwa przestrzeni + nazwa elementy z tej przestrzeni) jest niewygodne. Aby korzystać tylko z krótkich nazw, możemy

[2] Intuicyjnie przestrzenie nazw są podobne do rodzin — dopóki w rodzinie *Kowalski* wszyscy mają inne imiona, nie ma problemu z rozróżnianiem jej członków. Jednocześnie nic nie stoi na przeszkodzie, aby w dwóch różnych rodzinach była osoba o tym samym imieniu: *Adam Kowalski* i *Adam Nowak* należą do dwóch rodzin i nie powinno być problemu z ich rozróżnieniem.

na początku programu określić, że domyślnie korzystamy z pewnej przestrzeni nazw:

```
#include <iostream>
using namespace std;
. . .
cout << "Przykładowy napis";
. . .
```

Deklaracja `using namespace std` informuje o korzystaniu z przestrzeni nazw `std` jako domyślnej. Spróbujmy napisać kompletny program wyprowadzający do strumienia wyjściowego programu przykładowy napis, tak aby ten program był zgodny ze standardem ANSI i normą POSIX[3]:

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
  cout << "Przykładowy napis";
  return EXIT_SUCCESS;
}
```

Pierwsza linia programu włącza plik nagłówkowy o nazwie *cstdlib*. Zauważmy, że nie podajemy rozszerzenia nazwy pliku. Jest to plik z biblioteki standardowej języka `C`, w oryginale nosi on nazwę *stdlib.h*. Od momentu rezygnacji z podawania rozszerzeń, pliku nagłówkowe pochodzące z biblioteki języka `C` noszą po prostu prefix 'c'. Po co włączamy ten plik? Zawiera on definicję symbolu `EXIT_SUCCESS` [4], który — wg. normy POSIX — powinien być rezultatem funkcji `main` w przypadku zakończenia programu bez błędów.

1.1.4. Standardowy strumień wyjściowy cout

Element o nazwie `cout` reprezentuje standardowy strumień wyjściowy programu (skojarzony w środowiskach graficznych z konsolą tekstową), odpowiada strumieniowi `stdout` z języka `C`. Wyprowadzanie danych do tego strumienia odbywa się z wykorzystaniem operatora «. Łatwo zauważyć, że operator ten w języku `C` reprezentuje bitowe przesunięcie w lewo, jednak w języku `C++` dodano mu — dla strumienia wyjściowego — dodatkowe funkcje[5]. Operator ten jest *wstawiaczem* (ang. *inserter*), dane umieszczone po prawej stronie tego operatora

[3] Powstanie standardu POSIX (ang. *Portable Operating System Interface for Unix*) wiąże się z próbą standaryzacji różnych odmian systemu Unix. POSIX standaryzuje m.in. interfejs programistyczny, interfejs użytkownika, polecenia systemowe, właściwości powłoki systemu.

[4] Zwykle programiści używają bezwzględnych wartości 0 i 1 jako kod wyjścia programu, norma POSIX zaleca jednak stosowanie symboli EXIT_SUCCESS oraz EXIT_FAILURE, oznaczających odpowiednio zakończenie bezbłędne, oraz zakończenie z informacją o błędzie. Wykorzystanie tych symboli pozwala polepszyć przenośność kodu źródłowego na inne platformy systemowe.

[5] Celowo unikamy precyzyjnego określenia *czym* jest *cout* oraz na czym polega dodanie nowych obowiązków operatorowi « — informacje te są bezpośrednio związane z programowaniem obiektowym w C++ i zostaną przedstawione w dalszej części tego rozdziału

są wyprowadzane do strumienia wyjściowego programu `cout` występującego po lewej stronie — zgodnie z ich typem, przy czym istnieje możliwość formatowania postaci wyjściowej. Przykład wyprowadzający zawartość różnych zmiennych do strumienia wyjściowego programu:

```
int   i = 10;
float f = 5.5;
char  c = 'A';
char  s[] = "Test cout";

cout << "i = ";
cout << i;
cout << '\n';

cout << "f = ";
cout << f;
cout << '\n';

cout << "c = ";
cout << c;
cout << '\n';

cout << "s = ";
cout << s;
cout << '\n';
```

Wyprowadzanie pojedynczych informacji jest uciążliwe, operator « może być użyty wielokrotnie w jednej instrukcji:

```
int   i = 10;
float f = 5.5;
char  c = 'A';
char  s[] = "Test cout";

cout << "i = " << i << endl;
cout << "f = " << f << endl;
cout << "c = " << c << endl;
cout << "s = " << s << endl;
```

W przedstawionym wyżej fragmencie pojawił się również symbol `endl`, wykorzystywany zamiast znaku `\n`. Symbol ten wstawia znak nowej linii — czyli właśnie `\n` — do strumienia wyjściowego programu oraz wymusza opróżnienie bufora tego strumienia, tak aby — być może aktualnie buforowana — informacja jak najszybciej pojawiła się w tym strumieniu. Jeżeli użyjemy symbolu `flush`, nastąpi tylko „wymiecienie" bufora wyjściowego. Symbole `endl` i `flush` są *manipulatorami* — jak sama nazwa wskazuje pozwalają na manipulowanie sposobem działania strumienia `cout`.

Wyprowadzane informacje mogą być formatowane, ten temat jest szeroki, ograniczymy się tutaj do podania jednego przykładu:

```
int i = 12345;
int j = 255;

cout << '|';
cout.width( 10 ); // Ustalenie szerokości pola dla liczby
cout << i << '|' << endl;

cout << '|';
cout.setf(ios::left); // Ustalenie flagi wyrównania do lewej strony pola
cout.width( 10 );     // Ustalenie szerokości pola dla liczby
cout << i << '|' << endl;

cout << "Dziesietnie  : " << dec << j << endl; // Pokaż dziesiętnie
cout << "Osemkowo     : " << oct << j << endl; // Pokaż ósemkowo
cout << "Szesnastkowo : " << hex << j << endl; // Pokaż szesnastkowo
```

Występujące w przykładzie symbole `dec`, `oct`, `hex` są manipulatorami przełączającymi strumień `cout` odpowiednio w tryb wyprowadzania liczb dziesiętnych, ósemkowych, szesnastkowych.

1.1.5. Standardowy strumień wejściowy cin

Element o nazwie `cout` reprezentuje standardowy strumień wejściowy programu, zwykle domyślnie skojarzony z klawiaturą. Umożliwia on wczytywanie danych z wykorzystaniem operatora » zwanego *wydobywaczem* (ang. *extractor*), który pobiera dane ze strumienia wejściowego, przeprowadza konieczne konwersje i wstawia je do zmiennych programu. Przykład wykorzystania `cin` oraz operatora «:

```
int   i;
float f;
char  s[ 80 ];

cout << "Wprowadz lb. calkowita: " << flush;
cin >> i;

cout << "Wprowadz lb. rzeczywista: " << flush;
cin >> f;

cout << "Wprowadz slowo: " << flush;
cin >> s;

cout << "Wprowadono:" << endl;
cout << "i = " << i << endl;
cout << "f = " << f << endl;
cout << "s = " << s << endl;
```

Przykład wczytuje do zmiennych `i`, `f`, `s` dane ze strumienia wejściowego, dokonując odpowiednich konwersji do postaci typu, zgodnego ze typami zmiennych występujących po prawej stronie operatora ». Należy zwrócić uwagę, że

domyślnie operator » pomija lub ignoruje tzw. *białe znaki* — są nimi znaki spacji, tabulacji, znak przejścia do nowego wiersza. Więcej informacji na temat strumieni oraz biblioteki standardowej języka C++ można znaleźć m.in. w [5].

1.1.6. Program przykładowy — wersja nieobiektowa

W następnym podrozdziale przedstawiony zostanie przykład, ilustrujący technikę programowania obiektowego w języku C++. Program dotyczyć będzie problemu wręcz trywialnego, lecz jest to zabieg świadomy — celem jest przystępne przedstawienie obiektowych mechanizmów języka C++. W tym miejscu przedstawimy oczywiste rozwiązanie problemu w postaci programu nieobiektowego.

Przykład 1: Należy napisać program umożliwiający wyliczenie pola kwadratu na podstawie podanej przez użytkownika długości boku. Program kontroluje poprawności liczby będącej dana wejściową — długość boku musi być liczba nieujemną.

Przykładowe rozwiązanie może być następujące:

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
  double bok;
  cout << "\nObliczam pole kwadratu" << endl;

  do
  {
    cout << "\nDlugosc boku: " << flush;
    cin >> bok;
    if( bok < 0 )
      cout << "Dlugosc boku musi byc lb. nieujemna.\n";
  }
  while( bok < 0 );

  cout << "Pole kwadratu : " << bok * bok << endl;

  return EXIT_SUCCESS;
}
```

W następnym rozdziale przedstawione zostanie inne rozwiązanie tego samego problemu — z wykorzystaniem programowania obiektowego.

1.2. Programowanie obiektowe — wprowadzenie

Język C++ oferuje wiele różnorodnych mechanizmów programowania obiektowego. Ich — nawet pobieżna prezentacja — przekracza ramy tego opracowa-

nia, dlatego zdecydowano się na dokładne omówienie zagadnień o charakterze fundamentalnym, wierząc że na tej dobrej podbudowie, indywidualne poznanie dalszych elementów programowania będzie łatwiejsze.

1.2.1. Obiektowość w analizie, projektowaniu i programowaniu

W podrozdziale 1.1.6 na str. 9 przedstawiono zadanie do wykonania, oraz przykładowe jego klasyczne, nieobiektowe rozwiązanie. W rozdziale tym dokonamy syntezy programu, który będzie rozwiązywać dokładnie ten sam problem, lecz rozwiązaniu nadamy charakter obiektowy. Ponownie zwracamy uwagę, że program dotyczy problemu wręcz trywialnego, i że jest to zabieg świadomy — celem jest przystępne przedstawienie obiektowych mechanizmów języka `C++`.

Obliczanie pola kwadratu — obiektowa analiza problemu

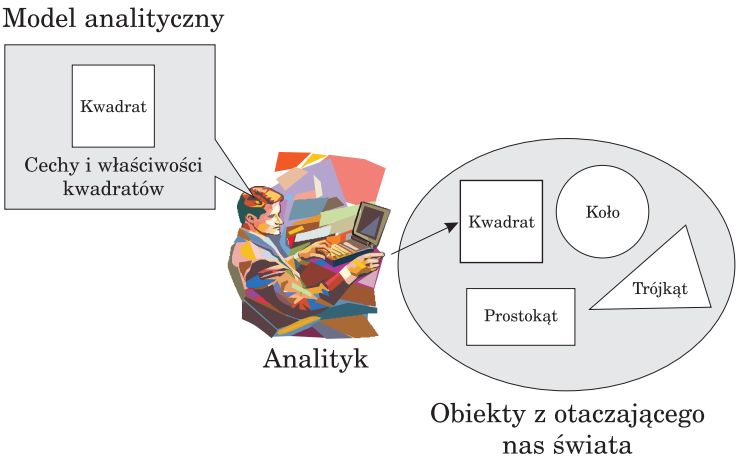
Wyznaczenie pola kwadratu dla znanej długości boku — zawartej przykładowo w zmiennej `dlBoku` — jest oczywiste i natychmiastowe: podnosimy wartość zmiennej `dlBoku` do kwadratu i otrzymujemy rozwiązanie. Zapomnijmy jednak na chwilę, że rozwiązanie jest tak proste, więcej poudawajmy przez chwilę, że nie mamy w tym momencie tzw. „zielonego pojęcia" jak wyznaczyć pole kwadratu.

Taka sytuacja jest dość typowa w momencie rozpoczęcia prac nad nowym programem — należy dokonać analizy problemu, jej celem jest znalezienie metod oraz środków jego rozwiązania. W przypadku obiektowego podejścia do projektowania i programowania, możemy stwierdzić, że:

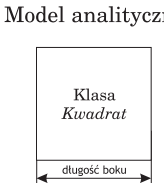
> Celem *analizy obiektowej* jest stworzenie modelu docelowego systemu, w którym każdy element świata rzeczywistego odwzorowany jest odpowiednim obiektem (lub ich grupą), należącym do określonej klasy. Interakcja pomiędzy obiektami jest odbiciem procesów zachodzących w rzeczywistości.

Jakie obiekty występują w problemie, który mamy rozwiązać? Rozważania dotyczą czegoś, co moglibyśmy nazwać roboczo *światem figur* — figur płaskich, dla których nie ważne jest zakotwiczenie w danym miejscu płaszczyzny a jedynie ich geometryczne parametry: pole, obwód. Kwadrat jest rzeczywiście taką figurą i dla naszego problemu istotne jest to, co służy do wyznaczeniu jego pola (no i może też — patrząc przyszłościowo — obwodu). Zatem celem analizy obiektowej w naszym przypadku jest wyodrębnienie najistotniejszych cech kwadratu — ale nie jakiegoś jednego, szczególnego, ale cech wspólnych dla każdego, możliwego kwadratu, na który możemy trafić rozwiązując nasz problem, ilustruje to rysunek 1.1.

W wyniku analizy cech wszystkich potencjalnych kwadratów, pojawia się pewien wspólny opis ich właściwości. Taki nazwany opis wspólnych właściwości obiektów w modelowaniu obiektowym nazywany jest *klasą*. Każda klasa posiada swoją nazwę i określa właściwości, które będzie posiadał *każdy* obiekt należący do danej klasy. Właściwości, które opisuje klasa są dostosowane do specyfiki problemu — zatem nie dotyczą wszystkich możliwych cech obiektów, leczy tylko tych, które w danym problemie są istotne. Każdy potencjalny kwadrat, dla którego chcemy policzyć pole, będzie należał do klasy `Kwadrat`.



Rysunek 1.1. Modelowanie obiektowe na przykładzie świata figur



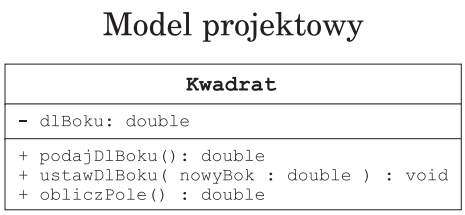
Rysunek 1.2. Model analityczny — klasa *Kwadrat*

Jakie właściwości kwadratów interesują nas w rozważanym problemie? Przypomnijmy, że interesuje nas obliczenie pola kwadratu — dowolnego kwadratu. Co jest do tego potrzebne? Wzór na obliczenie pola P kwadratu znamy, lub z łatwością odnajdziemy w podręcznikach: $P = a^2$ lub $P = a*a$, gdzie a to długość boku kwadratu. Zatem najistotniejszą cechą każdego kwadratu będzie długość jego boku. To w tym momencie niezbędne minimum dla realizacji naszego celu. Ale jednocześnie wystarczające do — potencjalnie możliwego w przyszłości — obliczania obwodu. W podobny sposób moglibyśmy uogólnić właściwości innych figur przedstawionych na rysunku 1.1. Nie interesuje nas w tym momencie nic innego — np. nieważny jest kolor kwadratu, jego położenie na płaszczyźnie — to nie są właściwości istotne dla rozważanego problemu. Rysunek 1.2 ilustruje właściwości klasy `Kwadrat` na etapie analizy — możemy szumnie powiedzieć, że zawiera on *model analityczny* naszej klasy opisu kwadratu.

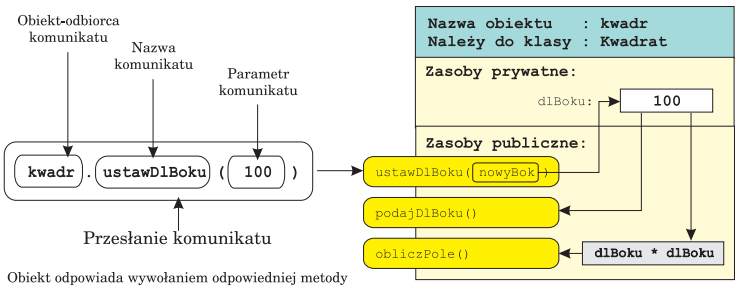
Obliczanie pola kwadratu — projektowanie obiektowe

Od fazy analizy problemu czas przejść do fazy projektowania systemu. Na tym etapie zidentyfikowane obiekty i ich klasy powinny zostać opisane zgodnie z odpowiednimi metodami projektowania obiektowego.

Celem *projektowania obiektowego* jest opisanie struktury projektowanego systemu w kategoriach obiektów i klas do których należą (zidentyfi-



Rysunek 1.3. Model projektowy — klasa *Kwadrat*



Rysunek 1.4. Obiekt klasy *Kwadrat*

kowanych w fazie analizy), stosując odpowiednie notacje, odwzorowujące logiczne i fizyczne właściwości obiektów, związki jakie zachodzą pomiędzy klasami, obiektami oraz opisujące ich właściwości statyczne i dynamiczne.

Aktualnie standardem modelowania obiektowego stał się język `UML`, kolejny rysunek 1.3 przedstawia naszą klasę właśnie jako element diagramu klas `UML`. W projektowanej klasie wyodrębniliśmy pole `dlBoku` (skrót *długość boku*), służące oczywiście do przechowywania informacji o długości boku kwadratu, jest to pole typu liczba rzeczywista podwójnej precyzji (typ `double`). Pole otrzymało na diagramie prefiksowy symbol '−', co oznacza, że to pole jest *prywatne* i odwoływać się do niego będzie mógł tylko sam obiekt klasy `Kwadrat`.

Aby ktokolwiek z otoczenia jakiegoś obiektu klasy `Kwadrat` mógł ustawiać lub pobierać wartość tego pola, należy wbudować w klasę *metody dostępowe*. Jeżeli będziemy dowiedzieć się jaka jest aktualna długość boku pewnego obiektu, musimy ten obiekt o to „poprosić" wywołaniem metody `podajDlBoku`. Jeżeli będziemy chcieli zmienić długość boku jakiegoś obiektu, będziemy musieli „poprosić" o to obiekt wywołaniem metody `ustawDlBoku`, która w postaci parametru otrzyma liczbę, określającą nową długość boku dla kwadratu. Metody dostępowe `podajDlBoku` oraz `ustawDlBoku` mają charakter administracyjny, nie realizują właściwego przetwarzania danych właściwych dla obiektu, do tego służyć będzie metoda `obliczPole`, będąca funkcją, zatem jej rezultat będzie określał pole kwadratu o długości zapisanej w prywatnym polu `dlBoku`. Wszystkie te metody otrzymały prefiksowy symbol '+', co oznacza, że są metodami publicznymi — ogólnie dostępnymi.

Publiczne metody klasy określają jej *interfejs* — komunikacja z obiektem takiej klasy odbywa się za pośrednictwem nazw metod, co ilustruje Rysunek 1.4.

Obliczanie pola kwadratu — programowanie obiektowe

Jak wykorzystać zaprojektowaną klasę do realizacji programu w języku `C++`? Aby można było wykorzystać zaprojektowaną klasę, potrzebny jest przynajmniej jeden jej *obiekt* (inaczej mówi się: *egzemplarz klasy*). Załóżmy na chwilę, że mamy jeden obiekt klasy `Kwadrat`, i że ten konkretny obiekt posiada nazwę `kwadr`. Powołanie do życia obiektu w języku `C++` polega na zdefiniowaniu *zmiennej*, będącej *obiektem*, której *typem* jest *nazwa klasy*:

```
Kwadrat kwadr;
```

Z tak zdefiniowanym obiektem możemy się komunikować, przesyłając komunikaty odpowiadające metodom zdefiniowanym w części publicznej klasy. Ustawienie długości boku o wartości 100:

```
kwadr.ustawDlBoku( 100 );
```

Od tego momentu prywatne pole `dlBoku` obiektu `kwadr` będzie miało wartość 100. Aby wyprowadzić do strumienia wyjściowego informacje o aktualnej długości boku kwadratu reprezentowanego przez `kwadr` napiszemy np.:

```
cout << "Dlugość boku kwadratu: " << kwadr.podajDlBoku() << endl;
```

Zwróćmy uwagę, że metoda `podajDlBoku` jest w istocie funkcją, której rezultatem jest wartość zapisana w polu `dlBoku`. Obiekt `kwadr` posiada również metodę wyznaczająca pole, którą przykładowo możemy wykorzystać następująco:

```
cout << "Pole kwadratu o boku: " << kwadr.podajDlBoku();
cout << " wynosi: " << kwadr.obliczPole();
```

To wystarczy aby spróbować napisać program obliczania pola kwadratu z wykorzystaniem obiektu klasy `Kwadrat`:

```
#include <cstdlib>
#include <iostream>
using namespace std;
. . .
int main()
{
  double bok;
  Kwadrat kwad;

  cout << "\nObliczam pole kwadratu" << endl;

  do
  {
    cout << "\nDlugosc boku: " << flush;
    cin >> bok;
    if( bok < 0 )
```

```
      cout << "Dlugosc boku musi byc lb. nieujemna.\n";
  }
  while( bok < 0 );

  kwad.ustawDlBoku( bok );
  cout << "Pole kwadratu : " << kwadr.obliczPole() << endl;

  return EXIT_SUCCESS;
}
```

Zwróćmy uwagę, że obiekt `kwad` klasy `Kwadrat` jest odpowiedzialny za przechowywanie informacji istotnych dla kwadratu i za realizację obliczeń — my nie posługujemy się już wzorem na pole kwadratu, to kwadrat „wie" jak obliczyć własne pole. W przedstawionym wyżej programie iteracja `do-while` kontroluje poprawność wprowadzanej długości boku. Zastanówmy się chwilę nad tym fragmentem — czy to czasem obiekt `kwad` nie powinien sam weryfikować poprawności danych? Rzeczywiście, dobrze byłoby, gdyby obiekt reprezentujący kwadrat informował nas o poprawnej wartości długości boku. Można to zrealizować na wiele sposobów, tutaj zaproponujemy jeden z najprostszych.

Proponujemy zmodyfikować metodę `ustawDlBoku` tak by była funkcją, a jej rezultatem była wartość *true*, gdy wartość przekazana tej funkcji jest poprawna, jeżeli wartość ta jest niepoprawna (ujemna, równa zero) rezultatem jest wartość *false*. Niepoprawna długość jest odrzucana, i przyjmuje się domyślną długość o wartości 0. Można to spróbować wykorzystać w następujący sposób:

```
#include <cstdlib>
#include <iostream>
using namespace std;
. . .
int main()
{
  double bok;
  bool   poprawneDane;
  Kwadrat kwadr;

  cout << "\nObliczam pole kwadratu" << endl;

  do
  {
    cout << "\nDługość boku: " << flush;
    cin >> bok;

    // Próba ustawienia długości boku i jej kontrola
    poprawneDane = kwadr.ustawDlBoku( bok );
    if( !poprawneDane )
      cout << "Długość boku musi być lb. nieujemną.\n";
  }
  while( !poprawneDane );
```

```
  cout << "Pole kwadratu: " << kwadr.obliczPole() << endl;

  return EXIT_SUCCESS;
}
```

W tej wersji programu to obiekt reprezentujący kwadrat sam jest odpowiedzialny za stwierdzenie, czy przekazywana długość jest poprawna. Linia:

```
poprawneDane = kwadr.ustawDlBoku( bok );
```

zawiera przekazanie obiektowi reprezentującemu kwadrat liczby określającej długość boku, obiekt sam zadecyduje czy tę wartość przyjąć i wstawić do prywatnego pola, a rezultat funkcji `ustawDlBoku`, zapamiętany w zmiennej `poprawneDane` zawiera informacje o tym, czy dane były poprawne, czy nie. Zmienna `poprawneDane` steruje wykonaniem iteracji `do-while` — po jej zakończeniu możemy przejść do wyznaczenia pola kwadratu, co realizuje wywołanie metody `obliczPole()` obiektu `kwadr`.

Porównanie przedstawionej wcześniej nieobiektowej wersji obliczania pola, z wersją obiektową, przedstawioną wyżej, skłania do natychmiastowego stwierdzenie, że wersja obiektowa jest dłuższa. Rzeczywiście tak jest. Co uzyskujemy w zamian? Wyraźne *rozdzielenie kompetencji* — program tylko wczytuje dane, przekazuje do obiektu reprezentującego kwadrat, ten weryfikuje je — zgodnie z własnymi kryteriami, oraz realizuje obliczenie pola — również zgodnie z sobie znanym wzorem. Niestety, wersja obiektowa będzie jeszcze dłuższa . . . , bowiem klasy `Kwadrat` przecież jeszcze nie ma! Następny podrozdział poświęcony jest właśnie je budowie.

1.2.2. Klasy i obiekty

Definiowanie klas polega na określaniu wzorca, według którego będą tworzone obiekty. Wzorzec ten określa dwie istotne właściwości obiektu:

— *dane*, które obiekt będzie przechowywał,
— *operacje*, które obiekt będzie wykonywał.

W zakresie przechowywanych *danych*, obiekt jest podobny do struktury danych zwanej *rekordem*[6], a klasa przypomina definicję typu rekordowego. Istotną różnicą jest wbudowanie w obiekt *operacji*, które może on wykonywać. Zgodnie z koncepcją programowania obiektowego, operacje te realizują *metody* wbudowane w obiekt. Jednak w języku `C++` używa się nazwy *funkcja składowa* (ang. *member function*). Możemy stwierdzić, że *metody* w języku `C++` są implementowane poprzez *funkcje składowe*.

Skoro definiowanie klasy w warstwie jej danych przypomina definiowanie typu rekordowego, pierwsza definicja klasy `Kwadrat` może mieć następującą postać:

```
class Kwadrat
{
```

[6] W języku C i C++ zwykle rekordy nazywa się *strukturami*.

```
  float dlBoku;
};
```

Definicja ta informuje, że od tego momentu w programie będzie występowała klasa `Kwadrat`, posiadająca jedno *pole* będące liczba rzeczywistą — typ `float` — o nazwie `dlBoku`. Gdybyśmy chcieli utworzyć obiekt `kwadr` klasy `Kwadrat` to piszemy oczywiście:

```
Kwadrat kwadr;
```

Możemy się spodziewać — poprzez analogię z typem rekordowym — że poprawne będzie następujące odwołanie do pola `dlBoku` obiektu `kwadr`:

```
kwadr.dlBoku = 10
```

oraz

```
cout << kwadr.dlBoku
```

Niestety odwołania te nie są prawidłowe — w przeciwieństwie do typu rekordowego, pole `dlBoku` jest *prywatne* i odwoływać się mogą do niego tylko metody klasy `Kwadrat`. W definicji klasy mogą wystąpić słowa kluczowe określające *zakres widoczności* elementów klasy. Brak jakiegokolwiek określenia zakresu widoczności elementów w klasie oznacza, że elementy te są prywatne. Możemy to zapisać jawnie, używając słowa kluczowego `private`:

```
class Kwadrat
{
  private:
    float dlBoku;
};
```

Umieszczenie pola w sekcji prywatnej klasy realizuje postulat *hermetyzacji*, oznaczający tutaj ukrywanie szczegółów implementacyjnych przez otoczeniem klasy oraz ochronę danych przed niekontrolowanym dostępem. Dostęp do prywatnego pola `dlBoku` realizowany będzie przez *metody dostępowe* `podajDlBoku` oraz `ustawDlBoku`. Pamiętajmy, że metoda `ustawDlBoku` kontroluje poprawność ustawianej długości boku.

Metody `podajDlBoku` i `ustawDlBoku` muszą być dostępne dla otoczenia klasy, zatem zdefiniujemy je w sekcji *publicznej* klasy, identyfikowanej słowem `public`:

```
class Kwadrat
{
  public:
    float podajDlBoku();
    bool  ustawDlBoku( float nowyBok );

  private:
```

```
    float dlBoku;
};
```

W sekcji `public` pojawiły się nazwy metod dostępowych, syntaktycznie są one *prototypami funkcji* — określają nazwy funkcji, typ ich rezultatu oraz typy parametrów[7]. Prototyp to *deklaracja funkcji*, nie zawiera ona *ciała funkcji*, czyli instrukcji, które mają być uruchomione po jej wywołaniu. Gdzie są te instrukcje? O tym za chwilę, teraz zobaczmy, że w przedstawionej powyżej klasie brak metody realizującej obliczanie pola kwadratu `obliczPole`, dopiszmy ją zatem i otrzymamy prawie kompletną definicję klasy `Kwadrat`:

```
class Kwadrat
{
  public:
    float podajDlBoku();
    bool  ustawDlBoku( float nowyBok );
    float obliczPole();

  private:
    float dlBoku;
};
```

Dlaczego prawie kompletną? Brakuje przecież instrukcji stanowiących ciała trzech — zapowiedzianych prototypami — funkcji składowych. Pełna specyfikacja funkcji, składająca się z jej nagłówka i ciała jest definicją funkcji, i może wyglądać następująco:

```
float Kwadrat::podajDlBoku()
{
  return dlBoku;
}

bool Kwadrat::ustawDlBoku( float nowyBok )
{
  return bool( dlBoku = ( nowyBok > 0 ) ? nowyBok : 0 );
}

float Kwadrat::obliczPole()
{
  return dlBoku * dlBoku;
}
```

W definicji funkcji składowych, przed każdą nazwą występuje prefiks w postaci nazwa klasy: `Kwadrat` oraz operatoratora '::'. Prefiks oznacza przynależność tych funkcji do klasy `Kwadrat` właśnie, a operator '::' jest operatorem zakresu. Dzięki nazwom funkcji z określeniem przynależności do klasy, możemy używać takich samych nazw w różnych klasach, a kompilator będzie wiedział, do której klasy dana funkcja należy.

[7] Przypominamy, że w języku C++ brak parametrów funkcji oznacza domyślnie wystąpienie słowa kluczowego *void*, w języku C oznacza operator '...', a więc dowolna liczbę parametrów.

Istnieje jeszcze jedna możliwość zdefiniowania funkcji składowych — jest nią umieszczenie definicji wewnątrz klasy, tam gdzie do tej pory umieszczane były prototypy funkcji:

```
class Kwadrat
{
  public:
    float podajDlBoku()
    {
      return dlBoku;
    }

    bool ustawDlBoku( float nowyBok )
    {
      return bool( dlBoku = ( nowyBok > 0 ) ? nowyBok : 0 );
    }

    float obliczPole()
    {
      return dlBoku * dlBoku;
    }

  private:
    float dlBoku;
};
```

Definicje funkcji wewnątrz deklaracji klasy nie wymagają stosowania prefiksów z nazwą klasy i operatorem zakresu — definiujemy funkcje w obrębie klasy i nie ma wątpliwości o ich przynależności. Obie formy — definicje poza klasą, oraz definicje wewnątrz klasy — mogą być stosowane zamiennie[8], w niniejszym opracowaniu wykorzystywana będzie forma pierwsza.

Stworzyliśmy już prototypową wersję klasy `Kwadrat`, co należy z nią zrobić? Kod klasy oraz funkcji składowych należy umieścić w miejscu trzech kropek w programie przykładowym umieszczonym na stronie 14. Teraz ten program jest kompletny i powinien działać.

Podsumowanie

Klasa definiuje wspólne właściwości obiektów, które będą reprezentantami takiej klasy. Klasa definiuje:

— typy i nazwy *pól*, w których obiekty będą przechowywać istotne dla nich dane.

— rodzaje oraz nazwy *metod*, które będą reprezentowały operacje, jakie będzie mógł wykonywać obiekt.

Właściwości — dane i metody — które opisuje klasa powinny być dostosowane do specyfiki problemu — zatem nie powinny dotyczyć wszystkich możliwych i szczególnych cech obiektów, leczy tylko tych, które w danym problemie są

[8] W istocie istnieje pewna różnica — jest nią domyślne traktowanie funkcji zdefiniowanych w obrębie klasy jako funkcji *wstawianych*, co zostanie omówione później.

istotne. To pierwsza i fundamentalna zasada programowania obiektowego — *abstrakcja*.

W języku C++ zamiast pojęcia *metoda* używa się zwykle sformułowania *funkcja składowa*. Pola jak i funkcje składowe mogą mieć różny zakres widoczności, można je bowiem deklarować w dwóch[9] sekcjach:

— `private`, elementy umieszczone w tej sekcji mogą być wykorzystywane wyłącznie przez metody danej klasy. Elementami tymi mogą być zarówno pola jak i metody. Mówi się o nich, że są prywatne.
— `public`, elementy umieszczone w tej sekcji są dostępne również dla innych elementów programu. Mówi się o nich, że są publiczne, lub stanowią *interfejs klasy*.

Dzięki możliwości podziału elementów klasy na składowe prywatne i publiczne, możemy kierować się drugą, fundamentalną zasadą programowania obiektowego — *hermetyzacją*. Każe ona przed otoczeniem klasy ukrywać konkretny sposób przechowywania danych i szczegóły realizacji poszczególnych operacji. Zwykle zatem uprywatniamy pola klasy, dostęp do nich obudowujemy odpowiednimi metodami dostępowymi. Te ostatnie są definiowanie w sekcji publicznej klasy. Metody publiczne klasy można nieformalnie podzielić na:

— *akcesory*, metody umożliwiające pobieranie wartości uprywatnionych pól,
— *modyfikatory*, metody dokonujące modyfikacji wartości uprywatnionych pól,
— *realizatory* — metody realizujące właściwe dla danej klasy operacje.

Na koniec tego rozdziału dodatkowy komentarz dotyczący *obiektów* `cin` oraz `cin`. Są to odpowiednio obiekty klas `istream` oraz `ostream`. Oba te obiekty zdefiniowano w bibliotece obsługi strumieniowego wejścia–wyjścia, identyfikowanej przez plik nagłówkowy *iostream*. Oprócz specjalnie zdefiniowanych dla tych strumieni operatorów `<<` oraz `>>`, obiekty te są wyposażone w szereg użytecznych metod, dobrze udokumentowanych w książce [5].

1.2.3. Konstruktory i destruktor

Zastanówmy się, jaka będzie wartość pola kwadratu w następującym przykładzie:

```
Kwadrat kwadr;

cout << "Pole kwadratu: " << kwadr.obliczPole();
```

Definiujemy obiekt `kwadr` i bez ustalenia długości boku prosimy go o obliczenie pola. Metod `obliczPole` oczywiście wyznaczy kwadrat wartości pola `dlBoku`, tylko jaka tam jest wartość? Zgodnie z regułami języka C++, jeżeli obiekt `kwadr` będzie *automatycznym*, wartość jego pól będzie przypadkowa. Takie też będzie wyznaczone pole. Gdyby `kwadr` był obiektem *zewnętrznym*, jego pola zostałyby wyzerowane, i zerowa byłaby wartość pola.

Ta niejednoznaczność nie jest dobra. Znacznie lepiej byłoby, gdyby obiekt tuż po utworzeniu inicjował wartości swoich pól, zgodnie z pewna ustaloną kon-

[9] Istnieje jeszcze jeden zakres widoczności pól — *protected*, który zostanie omówiony później

wencją. W przypadku kwadratu, ta inicjalizacji mogłaby po prostu polegać na wyzerowaniu wartości pola `dlBoku`.

Ponieważ w rzeczywistości obiekty są znacznie bardziej skomplikowane niż rozważany przez nas kwadrat, inicjalizacja ich pól może być zagadnieniem złożonym. Do realizacji procesu inicjalizowania wartości pól nowo utworzonego obiektu wprowadzono do języka C++ szczególny rodzaj funkcji składowej, zwanej *konstruktorem*.

Konstruktor domyślny

Konstruktor jest specjalną funkcją, aktywowaną przez kompilator automatycznie, w momencie gdy obiekt został utworzony. Dzieje się tak zanim programista będzie mógł odwołać się do obiektu. Konstruktor ma przygotować obiekt do „życia" — inicjalizacja pól obiektu jest właśnie jedną z takich czynności.

Konstruktor to specyficzna funkcja:

— nie ma określonego typu rezultatu (nie wolno używać nawet słowa kluczowego `void` jako typu rezultatu);
— nosi taką nazwę, jak nazwa klasy;
— jest wywoływany automatycznie, i zazwyczaj programista nie wywołuje jawnie konstruktora.

Uzbrojeni w tę wiedzę, spróbujmy rozszerzyć klasę `Kwadrat` o deklaracje konstruktora:

```
class Kwadrat
{
  public:
    Kwadrat();

    float podajDlBoku();
    bool  ustawDlBoku( float nowyBok );
    float obliczPole();

  private:
    float dlBoku;
};
```

Deklaracja klasy wzbogaciła się o linię:

```
Kwadrat();
```

będącą deklaracją prototypową konstruktora — rzeczywiście nie został określony rezultat funkcji, jej nazwa jest taka jak nazwa klasy. Dodatkowo tak zadeklarowany konstruktor jest funkcją bezparametrową. Definicja konstruktora umieszczona zostanie poza obrębem klasy i ma następującą postać:

```
Kwadrat::Kwadrat()
{
  dlBoku = 0;
}
```

Nieco dziwnie wygląda nagłówek: „`Kwadrat::Kwadrat()`". W istocie nie ma w nim nic dziwnego, fragment „`Kwadrat::`" informuje, że mamy do czynienia z funkcją składową klasy `Kwadrat`, natomiast reszta nagłówka „`Kwadrat()`" to po prostu nazwa funkcji składowej. Nasz konstruktor inicjalizuje długość boku kwadratu wartością 0.

Kiedy konstruktor jest wywoływany i przez kogo? Konstruktor jest wywoływany automatycznie (kompilator umieszcza w programie odpowiedni kod) tuż po utworzeniu nowego obiektu. Nieważne, czy ten obiekt jest zmienną zewnętrzną, automatyczną, czy alokowaną dynamicznie. W podanym niżej przykładzie:

```
Kwadrat a, b;
Kwadrat k[3];
```

wywołanie konstruktora nastąpi automatycznie dla obiektu `a` i `b`, oraz dla każdego z trzech elementów tablicy `k`. Każdorazowo spowoduje ono wyzerowanie pola określającego długość boku.

Konstruktor posiada jeszcze jedna cechę odróżniająca go od zwykłej funkcji — może posiadać *listę inicjalizacyjną*. Podstawowym jej zastosowaniem jest inicjowanie pól obiektu, które odbywa się przed wykonaniem ciała funkcji. Na liście może wystąpić nazwa pola, a w nawiasach okrągłych wartość inicjalizująca to pole. Alternatywna wersja konstruktora klasy `Kwadrat` może mieć następującą postać:

```
Kwadrat::Kwadrat() : dlBoku( 0 )
{
}
```

Ponieważ pole `dlBoku` zostało zainicjowane na liście inicjalizacyjnej, ciało konstruktora jest puste. Która wersja konstruktora jest lepsza, ta wcześniejsza, czy ta z listą inicjalizacyjną? Programista ma tutaj sporą dowolność, jednak pewne operacje można wykonać tylko na liście inicjalizacyjnej, zatem jej stosowanie jest dobrym nawykiem i będzie tu preferowane.

Zauważmy, że wykorzystywany przez nas konstruktor nie posiada parametrów, i jest wywoływany w sytuacji definiowania zmiennych bez jakichkolwiek inicjatorów. Taki rodzaj konstruktora nazywany jest *konstruktorem domyślnym* (ang. *default constructor*) i odpowiedzialny jest właśnie za inicjalizowanie obiektu wartościami domyślnymi.

Konstruktor ogólny

Język C++ pozwala na inicjalizowanie zmiennych na etapie ich definiowania, np. definicja `int i = 10;` inicjalizuje wartość zmiennej `i` wartością 10. Czy podobnie można postąpić ze zmienną, będącą obiektem? Rzeczywiście, na etapie definiowania obiektu, można przekazać porcję danych, które mają zainicjować pola tworzonego obiektu. Następująca definicja powołuje do życia obiekt `kwadr`, którego pole `dlBoku` ma zostać zainicjowana wartością 100:

```
Kwadrat kwadr( 100 );
```

W jaki sposób wartość 100 ma zostać wstawiona do pola `dlBoku`? Przypomnijmy, że to konstruktor ma być odpowiedzialny za inicjowanie pól obiektu.

Omówiony wcześniej konstruktor domyślny nie nadaję się do tego celu. Okazuje się jednak, że w języku C++ można zdefiniować więcej niż jeden konstruktor, również taki, który zainicjuje obiekt wartościami innymi niż domyślne. Taki konstruktor nazywany jest *konstruktorem ogólnym* (ang. *general constructor*). Uzupełnijmy definicję klasy kwadrat właśnie o konstruktor ogólny:

```
class Kwadrat
{
  public:
    Kwadrat();
    Kwadrat( float startowyBok );
  . . .
};
```

Zauważmy, że mamy teraz dwa konstruktory, będące funkcjami o tej samej nazwie, różniące się parametrami. Taka sytuacja jest w języku C++ dozwolona, nazwy funkcji mogą być *przeciążane*. Drugi konstruktor będzie aktywowany automatycznie w opisywanej powyżej sytuacji — a więc gdy tworzymy obiekt z określeniem wartości startowej boku. Zatem definicja `Kwadrat kwadr( 100 );` spowoduje niejawne wywołanie dla obiektu `kwadr` konstruktora `Kwadrat( 100 )`. Jego zadaniem jest wstawienie wartości 100 do pola `dlBoku`, co może być zrealizowane następująco:

```
Kwadrat::Kwadrat( float startowyBok )
{
  dlBoku = startowyBok;
}
```

Wartość 100, określona na etapie definicji obiektu `kwadr`, trafia do parametru `startowyBok`, skąd jest przekazywana do pola `dlBoku` w ciele konstruktora. Można to zrealizować również z wykorzystaniem listy inicjalizacyjnej:

```
Kwadrat::Kwadrat( float startowyBok ) :  dlBoku( startowyBok )
{
}
```

Zauważmy jednak, że w obu tych przypadkach, konstruktory pozwolą na wstawienie do pola `dlBoku` również wartości ujemnych — a te, jak ustaliliśmy wcześniej, nie powinny być przyjmowane. Powinniśmy skontrolować wstawianą wartość:

```
Kwadrat::Kwadrat( float startowyBok )
{
  dlBoku = ( startowyBok > 0 ) ? startowyBok : 0;
}
```

W konstruktorze można wywoływać oczywiście inne funkcje, zwróćmy uwagę, że funkcja `ustawDlBoku` posiada już zabezpieczenie przed wstawieniem wartości ujemnej, zatem nasz konstruktor parametrowy możemy ostatecznie zapisać tak:

```
Kwadrat::Kwadrat( float startowyBok )
{
  ustawDlBoku( startowyBok );
}
```

Konstruktor kopiujący

W języku C++ można inicjować wartość definiowanej zmiennej, wartością innej zmiennej tego typu, np.:

```
int i = 10;
int j = i;
```

Czy podobnie można postąpić ze zmienną, będącą obiektem?

```
Kwadrat a( 100 );
Kwadrat b = a;
```

Czy można zainicjować obiekt `b`, wartością obiektu `a`? Tak, inicjalizacja polega na skopiowaniu zawartości pola `dlBoku` obiektu `a` (wartość 100 ustalona prze konstruktor ogólny), do pola `dlBoku` obiektu `b`. Gdyby pól było więcej, taką procedurę powinniśmy powtórzyć dla każdego z pól.

Wiemy już, że za inicjowanie obiektów na etapie ich tworzenia odpowiadają konstruktory. Za obsługę opisywanej sytuacji odpowiada trzeci rodzaj konstruktora — *konstruktor kopiujący* — (ang. *copy constructor*). Konstruktor kopiujący odpowiedzialny za skopiowanie zawartości obiektów tej samej klasy na etapie inicjalizacji. Nosi on oczywiście taka samą nazwę jak dwa pozostałe, i różni się od nich parametrem — otrzymuje jeden parametr będący obiektem tej samej klasy, do której należy konstruktor. Parametr ten reprezentuje obiekt inicjalizujący, wewnątrz konstruktora można dokonać przepisania wartości z pól tego obiektu. Uzupełnijmy klasę `Kwadrat` o taki konstruktor:

```
class Kwadrat
{
  public:
    Kwadrat();
    Kwadrat( float startowyBok );
    Kwadrat( Kwadrat & innyKwadrat );
  . . .
};
```

Parametr tego konstruktora to obiekt klasy `Kwadrat` o nazwie `innyKwadrat`. Znak '&' oznacza w języku C++ *referencję*, umieszczony w deklaracji parametru oznacza, że parametr ten jest przekazywany przez zmienną. Fakt ten ma istotne znaczenie — zapamiętajmy, że zapomnienie znaku oznaczającego referencję w konstruktorze kopiującym może spowodować spore problemy. Zadanie skopiowania zawartości pól może zostać zrealizowane w następujący sposób:

```
Kwadrat::Kwadrat( Kwadrat & innyKwadrat )
{
  dlBoku = innyKwadrat.dlBoku;
}
```

Alternatywnie, można wykorzystać listę inicjalizacyjną konstruktora kopiującego:

```
Kwadrat::Kwadrat( Kwadrat & innyKwadrat )
: dlBoku( innyKwadrat.dlBoku )
{
}
```

Przedstawiony powyżej przykład konstruktora kopiującego pozwala zrozumieć, jaka jest jego koncepcja i jak on działa. Jednak w rozważanym przypadku nie trzeba definiować konstruktora kopiującego. Jest tak dlatego, że w przypadku jego braku, kompilator zrealizuje inicjalizację kopiując pole po polu, zawartość obiektu inicjującego do obiektu inicjalizowanego – a w naszym przypadku w zupełności to wystarczy. Zdefiniowanie konstruktora kopiującego jest konieczne w przypadku sytuacji, gdy takie kopiowanie nie wystarcza, a tak jest w przypadku pól wskaźnikowych, odwołujących się do obszarów pamięci operacyjnej przydzielanych dynamicznie.

Konstruktor rzutujący

W języku C++ można inicjować wartość definiowanej zmiennej, wartością innej zmiennej, innego typu, jeżeli istnieje droga konwersji wartości, np.:

```
int i = 10;
float j = i;
```

Załóżmy, że istnieje klasa `Prostokąt`, analogiczna do klasy `Kwadrat`. Czy można zainicjować obiekt klasy `Prostokąt` obiektem klasy `Kwadrat`? Kwadrat jest przecież prostokątem, zatem inicjowany prostokąt będzie po prostu prostokątem o równych bokach.

```
Kwadrat a( 100 );
Prostokąt b = a;
```

Domyślamy się już, że można to zrobić, i że będzie za to odpowiedzialny odpowiedni konstruktor. Taki konstruktor nazywa się *konstruktorem rzutujacym* (ang. *cast constructor*). Ale uwaga — ten konstruktor będzie elementem klasy `Prostokąt`, bo to konwersję na obiekt tej klasy przewidujemy, dla obiektu klasy `Kwadrat`. Konwersja odwrotna nie jest rozsądna.

Zatem by przedstawić koncepcję konstruktora rzutującego, musimy najpierw stworzyć klasę `Prostokąt`.

Klasa Prostokąt

Dla klasy prostokąt najistotniejsze będą długości dwóch boków — stanowią one szerokość i wysokość prostokąta. Będą to podstawowe informacje istotne dla prostokąta, prezentuje to rysunek 1.5.

Model projektowy prezentuje rysunek 1.6, zawiera on podstawowe elementy — pola, akcesory, modyfikatory, realizator. Diagram ten nie zawiera jeszcze konstruktorów.

Pełny kod źródłowy klasy `Prostokat` zawarty jest w dalszej części tego podrozdziału.

Model analityczny

Rysunek 1.5. Model analityczny — klasa *Prostokąt*

Model projektowy

Prostokat
- szer: double - wys : double
+ podajWysokosc(): double + ustawWysokosc(nowaWys : double) : bool + podajSzerokosc(): double + ustawSzerokosc(nowaSzer : double) : bool + obliczPole() : double

Rysunek 1.6. Model projektowy — klasa *Prostokąt*

```
class Prostokat
{
  public:
    Prostokat();
    Prostokat( double startWys, double startSzer );
    Prostokat( Kwadrat & jakisKwadrat );

    double podajWysokosc();
    bool   ustawWysokosc( double nowaWys );
    double podajSzerokosc();
    bool   ustawSzerokosc( double nowaSzer );
    double obliczPole();

  private:
    double szer;
    double wys;
};

Prostokat::Prostokat() : wys( 0 ), szer( 0 )
{
}

Prostokat::Prostokat( double startWys, double startSzer )
{
```

```
  ustawWysokosc( startWys );
  ustawSzerokosc( startSzer );
}

Prostokat::Prostokat( Kwadrat & jakisKwadrat )
: szer( jakisKwadrat.podajDlBoku() ), wys( jakisKwadrat.podajDlBoku() )
{
}

double Prostokat::podajWysokosc()
{
  return wys;
}

bool Prostokat::ustawWysokosc( double nowaWys )
{
  return bool( wys = ( nowaWys > 0 ) ? nowaWys : 0 );
}

double Prostokat::podajSzerokosc()
{
  return szer;
}

bool Prostokat::ustawSzerokosc( double nowaSzer )
{
  return bool( szer = ( nowaSzer > 0 ) ? nowaSzer : 0 );
}

double Prostokat::obliczPole()
{
  return wys * szer;
}
```

Konstruktor rzutujący raz jeszcze

Klasa `Prostokat` posiada zdefiniowany konstruktor rzutujący pozwalający na zainicjowanie danymi kwadratu, obiektu reprezentującego prostokąt. Konstruktor rzutujący klasy `Prostokat` otrzymuje jeden parametr referencyjny innego typu — w naszym przypadku nazywa się on `jakisKwadrat`. Przetransponowanie danych kwadratu do prostokąta polega na zainicjowaniu jego wysokości i szerokości długością boku kwadratu, można to zrobić następująco:

```
Prostokat::Prostokat( Kwadrat & jakisKwadrat )
: szer( jakisKwadrat.podajDlBoku() ), wys( jakisKwadrat.podajDlBoku() )
{
}
```

W przypadku braku zaufania do danych zawartych w obiekcie klasy `Kwadrat`,

można użyć w konstruktorze rzutującym funkcji ustawiających szerokość i wysokość, te realizują odpowiednie czynności kontrolne:

```
Prostokat::Prostokat( Kwadrat & jakisKwadrat )
{
  ustawSzerokosc( jakisKwadrat.podajDlBoku() );
  ustawWysokosc( jakisKwadrat.podajDlBoku() );
}
```

Destruktor

Konstruktory zapewniają automatyczne wykonywanie pewnych czynności dla obiektu, który właśnie powstał. Język C++ oferuje mechanizm komplementarny — automatyczne wykonywanie czynności dla obiektu, który zostanie za chwilę usunięty. Dla tej sytuacji przewidziano również specjalną funkcję, nazywaną *destruktorem*. W przeciwieństwie do konstruktorów w danej klasie przewiduje się istnienie tylko jednego destruktora.

Typowa rola destruktora to wykonanie czynności porządkowych, przed usunięciem obiektu z pamięci operacyjnej. W przypadku klas `Kwadrat` czy `Prostokat` trudno wskazać sensowne wykorzystanie destruktora. Przeanalizujmy jego działanie na innym przykładzie.

Załóżmy, że potrzebujemy obiektu, który będzie rejestrował czas rozpoczęcia programu — rozpoczęcia wykonania funkcji `main`, a tuż przed jej zakończeniem wyliczy ile od tego czasu upłynęło sekund, i wyprowadzi tę informację do strumienia wyjściowego programu. Niech klasa tego obiektu nazywa się `RejestratorCzasu`, a jego wykorzystanie następujące:

```
int main()
{
  RejestratorCzasu r;

  int i;
  cout << "Wpisz liczbe: ";
  cin >> i;

  return EXIT_SUCCESS;
}
```

Program prosi użytkownika o wpisanie liczby, przed zakończeniem programu ma on ma wyświetlić, ile trwała ta operacja. Osiągniemy to bez jakiejkolwiek modyfikacji przedstawionej funkcji `main`. Jak to możliwe?

Pierwsza linia tej funkcji zawiera definicję obiektu `r` klasy `RejestratorCzasu`. Zatem tuż po rozpoczęciu funkcji `main` zmienna ta jest tworzona. Przypomnijmy, że tuż po utworzeniu obiektu `r`, wywoływany jest *automatycznie jego konstruktor*, w naszym przypadku to konstruktor domyślny. Zadaniem konstruktora będzie zarejestrowanie aktualnego stanu zegara systemowego. Obiekt `r` jest automatyczny, zostanie zatem usunięty z pamięci operacyjnej tuż przed zakończeniem funkcji `main`. Wtedy też, zostanie dla tego obiektu *automatycznie wywołany destruktor*. Jego zadaniem będzie zarejestrowanie stanu zegara systemowego w tym momencie, oraz wyznaczenie różnicy pomiędzy tym stanem

zegara, a stanem zapamiętanym w konstruktorze. Różnica da przybliżony czas wykonania funkcji `main`.

Klasa `RejestratorCzasu` będzie posiadał tylko jedno pole prywatne, przechowujące stan zegara systemowego zarejestrowany w konstruktorze. Niech pole nazywa się `start` i jest typu `clock_t`, zdefiniowanego w pliku nagłówkowym *ctime*. Klasa będzie zawierała tylko dwie funkcje składowe — konstruktor domyślny i destruktor.

```
class RejestratorCzasu
{
  public:
    RejestratorCzasu()
    {
      start = clock();
    }

    ~RejestratorCzasu()
    {
       clock_t end = clock();
       cout << "Czas: ";
       cout << int( ( end - start )/CLK_TCK );
       cout << " sek.";
    }

  private:
    clock_t start;
};
```

Konstruktor domyślny, o oczywistej nazwie `RejestratorCzasu`, wywołuje funkcję biblioteczną `clock()`, której rezultatem jest aktualny stan zegara systemowego. I to wszystko. Resztę robi destruktor, zauważmy, że jego nazwa to `~RejestratorCzasu`, nosi więc nazwę klasy poprzedzoną znakiem tyldy, nie ma określonego rezultatu i nie otrzymuje parametrów. Destruktor zapamiętuje w lokalnej zmiennej `end` stan zegara w momencie rozpoczęcia działania destruktora. Różnica `end - start` określa liczbę impulsów zegarowych, po podzieleniu ich przez stałą `CLK_TCK` otrzymujemy przybliżoną liczbę sekund, jaką trwało wykonanie bloku.

Parametry domyślne

Parametr domyślny to wartość określona na etapie deklaracji funkcji, która zostanie automatycznie wstawiona do *parametru formalnego*, jeżeli dana funkcja zostanie wywołana bez odpowiedniego *parametru aktualnego*. Parametry domyślne dotyczą funkcji składowych klas jak i funkcji niezwiązanych z klasami. Prześledźmy to na przykładzie, załóżmy, że w prototypie funkcji określono dwa ostatnie parametry, jako domyślne:

```
void fun( int i, float f = 0, char c = 'A' );
```

Taką funkcję można wywoływać z pominięciem końcowych parametrów, ich wartość zostanie ustalona automatycznie zgodnie z wartościami domyślnymi:

```
fun( 10 );             // i == 10, f == 0,    c == 'A'
fun( 20, 3.15 );       // i == 20, f == 3.15, c == 'A'
fun( 30, 22.1, 'Z' );  // i == 30, f == 22.1, c == 'Z'
```

Jeżeli stosujemy prototypy funkcji, wartości parametrów domyślnych określa się właśnie w prototypie, w definicji funkcji już nie występują.

```
void fun( int i, float f = 0, char c = 'A' );
. . .
void fun( int i, float f, char c )
{
}
```

Parametry domyślne można wykorzystać w celu zmniejszenia liczby konstruktorów. Zauważmy, że rozważana wcześniej klasa `Kwadrat` posiadała konstruktor *domyślny* i *ogólny*:

```
class Kwadrat
{
  public:
    Kwadrat();
    Kwadrat( float startowyBok );
  . . .
};
. . .
Kwadrat::Kwadrat() : dlBoku( 0 )
{
}

Kwadrat::Kwadrat( float startowyBok )
{
  dlBoku = ( startowyBok > 0 ) ? startowyBok : 0;
}
```

Zauważmy, że po wprowadzeniu parametru domyślnego do konstruktora ogólnego, będzie on mógł wypełniać również rolę konstruktora domyślnego, który stanie się zbędny:

```
class Kwadrat
{
  public:
    Kwadrat( float startowyBok = 0 );
  . . .
};
. . .
Kwadrat a;       // Wywołanie Kwadrat() z param. domyśl. 0
Kwadrat b( 10 ); // Wywołanie Kwadrat( 10 )
```

Podsumowanie

Czas na krótkie podsumowanie informacji o konstruktorach — niżej zakładamy, że rozważania dotyczą pewnej klasy o nazwie `A`.

Konstruktor domyślny (ang. *default constructor*):
— Jest bezparametrowy `A()`, lub posiada wszystkie parametry będące parametrami domyślnymi: `A( arg1 = wart1, arg2 = wart2, ... )`.
— Jednoczesne wystąpienie obu powyższych form spowoduje błąd kompilacji.
— Inicjuje obiekty, deklarowane lub tworzone bez parametrów.
— Dotyczy to również obiektów będących elementami tablicy.

```
A a, b, c;    // Aktywacja: a.A(), b.A(), c.A()
A tab[ 10 ];  // Aktywacja: A() dla każdego z 10-ciu elementów tab:
              // tab[ 0 ].A(), tab[ 1 ].A(), itd... .
```

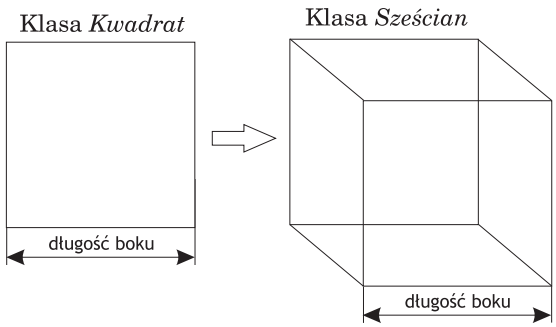
Konstruktor ogólny (ang. *general constructor*):
— Jest to podstawowy konstruktor przeznaczony do inicjowania obiektów na etapie ich deklaracji czy też tworzenia: `A( arg1, arg2, ... )`.
— Argumenty określają zwykle wartości jakie mają być przypisane określonym polom obiektu.
— Konstruktorów głównych może być więcej, mogą one zawierać również parametry domyślne.
— Szczególnym przypadkiem jest konstruktor posiadający tylko parametry domyślne, staje się on wtedy konstruktorem domyślnym.

```
A( int a, float b, char * c  = NULL );

A obj1( 2, 3.4, "Ala");
A obj2( 1, 0.0 );
A obj3( 5, 5.5, "Pięć");
```

Konstruktor kopiujący (ang. *copy constructor*):
— Jest potrzebny jedynie wtedy, gdy przewidziana jest inicjalizacja obiektu danej klasy innym obiektem tejże klasy, przyjmuje różne postacie:
```
A( A & obj )
A( A & obj, arg1 = wart1, ... )
A( const A & obj )
A( const A & obj, arg1 = wart1, ... )
```
— W przypadku braku konstruktora kopiującego, kompilator zrealizuje inicjowanie nowotworzonego obiektu, kopiując pole po polu, wartości z obiektu inicjalizującego.
— Stosowanie konstruktora kopiującego jest dobrą, programistyczną praktyką. Dzięki jawnie zdefiniowanym konstruktorom programista ma kontrolę nad kopiowaniem wartości, występujących w wielu, czasem zaskakujących sytuacjach.

```
A obj1;
A obj2 = obj1;  // Uaktywnienie obj2.A( obj1 )
A obj3( obj2 ); // Uaktywnienie obj3.A( obj2 )
```

Konstruktor rzutujący (ang. *cast constructor*):
— Posiada jeden parametr będący referencją obiektu innej klasy. Innych argumentów może nie być lub powinny być one argumentami domyślnymi: `A( B & obj )` lub `A( const B & obj )`.



Rysunek 1.7. Od Kwadratu do Sześcianu — koncepcja dziedziczenia

— Jest stosowany wszędzie tam, gdzie należy zainicjować obiekt pewnej klasy wartością obiektu innej klasy.
— Programista może dzięki konstruktorowi rzutującemu określić w jaki sposób informacje zapisane w obiekcie klasy B mają zostać odwzorowane (przepisane) w obiekcie klasy A.

```
A obj1;
B obj2( obj1 ); // Uaktywnienie obj2.B( obj1 )
```

1.2.4. Dziedziczenie w programowaniu obiektowym

Koncepcja *dziedziczenia* (ang. *inheritance*) pozwala na budowanie nowych klas na podstawie klas już istniejących. Te nowe klasy, nazywane są *klasami pochodnymi*, zaś klasy stanowiące podstawę dziedziczenia, nazywamy *klasami bazowymi*. Każda klasa pochodna dziedziczy wszystkie właściwości klasy bazowej, rozszerzając ją o nowe pola i/lub metody.

Dziedziczenie jest zatem procesem tworzenia klas potomnych (ang. *derivation*), pozwala ono urzeczywistnić pomysł *powtórnego wykorzystania kodu*. Koncepcja ta w oryginale nosi angielską nazwę *code reusability*. Zakłada ono, że pewnych klas nie trzeba tworzyć od nowa, o ile istnieją takie, które można rozszerzyć lub zaadaptować do stojących przed programistą nowych zadań.

Od kwadratu do sześcianu — przykład dziedziczenia

Załóżmy, że naszym zadaniem jest napisanie programu obliczającego pole i objętość *sześcianu*. Sześcian jest oparty na kwadracie, o jego polu i objętości decyduje długość boku jednej ze ścian. A każda z nich jest kwadratem. Można zatem założyć, że sześcian to specyficzny kwadrat — wyciągnięty w przestrzeni, obdarzony trzecim wymiarem. Ilustruje to rysunek 1.7.

W poprzednim rozdziale zdefiniowaliśmy klasę `Kwadrat`, definiowała ona kwadrat, jako figurę geometryczną określoną długością boku, pamiętaną w polu o nazwie `dlBoku`. Klasa ta definiowała również funkcję składową `obliczPole`, obliczającą pole kwadratu. Nasuwa się pytanie — czy można wykorzystać istniejący już kod klasy `Kwadrat` do utworzenia klasy reprezentującej sześcian? Niech ta klasa nazywa się właśnie `Szescian` i niech zawiera funkcje wyznaczania łącz-

nego pola powierzchni wszystkich ścian: `obliczPole`, oraz funkcję wyznaczania objętości: `obliczObjetosc`.

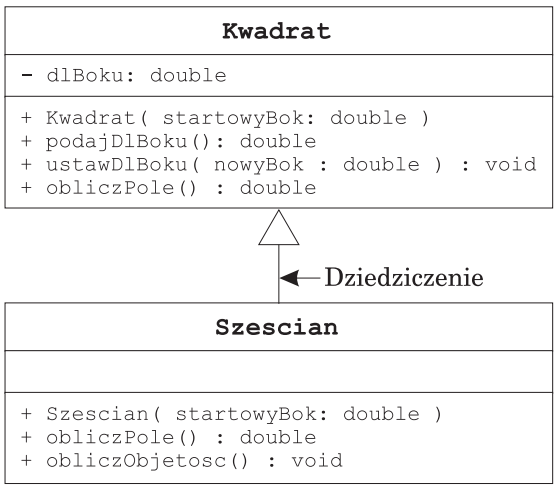
Rzeczywiście, klasa `Kwadrat` może posłużyć jako klasa bazowa do opracowania klasy `Szescian`. Koncepcyjnie wydaje się to poprawne — *bryła* sześcian powstaje na bazie *figury* będącej kwadratem a do opisu parametrów sześcianu wystarczy długość boku kwadratu. Niestety, funkcja `obliczPole` klasy `Kwadrat` oblicza pole kwadratu a nie sześcianu — trzeba będzie coś z tym zrobić. Klasa `Kwadrat` nie posiada również funkcji obliczającej objętość, trzeba ją będzie zdefiniować. Rozpocznijmy jednak od przypomnienia ostatecznej wersji klasy `Kwadrat`:

```
class Kwadrat
{
  public:
    Kwadrat( double startowyBok = 0 );

    double podajDlBoku();
    bool  ustawDlBoku( double nowyBok );
    double obliczPole();

  private:
    double dlBoku;
};
```

Funkcje składowe tej klasy mogą mieć następującą postać:

```
Kwadrat::Kwadrat( double startowyBok )
{
  ustawDlBoku( startowyBok );
}

double Kwadrat::podajDlBoku()
{
  return dlBoku;
}

bool Kwadrat::ustawDlBoku( double nowyBok )
{
  return bool( dlBoku = ( nowyBok > 0 ) ? nowyBok : 0 );
}

double Kwadrat::obliczPole()
{
  return dlBoku * dlBoku;
}
```

Koncepcję dziedziczenia ilustruje rysunek 1.8. Każdy obiekt klasy `Szescian` będzie zawierał wszystkie elementy klasy `Kwadrat` oraz dodatkowo funkcję obliczania objętości. Przypadek funkcji `obliczPole` wymaga dodatkowego komentarza.



Rysunek 1.8. Od Kwadratu do Sześcianu — diagram hierarchii klas

Funkcja obliczania pola zdefiniowana w klasie opisu kwadratu wyznacza jego pole we właściwy dla tej figury sposób. Niestety, po zastosowaniu dziedziczenia, funkcja ta w klasie reprezentującej sześcian działałaby wadliwie — wyznaczałaby dla sześcianu pole wg. wzoru dla kwadratu. Dlatego dokonujemy w klasie `Szescian` *redefinicji* funkcji obliczania pola — definiujemy `obliczPole` ponownie, tak by działała według wzoru właściwego dla sześcianu.

Zobaczmy, jak buduje się klasę pochodną w języku C++. Rozpocznijmy od zdefiniowania pustej klasy `Szescian`:

```
class Szescian : public Kwadrat
{
};
```

W definicji klasy pojawił się nowy zapis — fraza „`: public Kwadrat`" oznacza, że klasa `Szescian` powstaje, dziedzicząc wszystkie pola i metody klasy `Kwadrat`. Słowo kluczowe `public` oznacza, że te składowe klasy, które były publiczne w klasie `Kwadrat`, są również publiczne w klasie `Szescian`, a składowe prywatne pozostają prywatnymi[10].

Rozbudowa klasy opisu sześcianu

Spróbujmy rozszerzyć definicję klasy `Szescian` o nowe elementy:

```
class Szescian : public Kwadrat
{
  public:
    double obliczPole();
```

[10] Inaczej mówiąc — w klasie pochodnej zostaje zachowana taka widoczność składowych klasy, jak obowiązywała w klasie bazowej. Tak być nie musi, lecz na tym etapie pomińmy inne możliwości, aby nie wprowadzać niepotrzebnego zamieszania.

```
    double obliczObjetosc();
};
```

Zadeklarowaliśmy, że klasa `Szescian` będzie posiadała funkcję `obliczObjetosc` oraz własną wersję funkcji obliczania pola `obliczPole`. Spróbujmy zdefiniować pierwszą wersję tych funkcji:

```
double Szescian::obliczPole()
{
  return 6 * podajDlBoku() * podajDlBoku();
}

double Szescian::obliczObjetosc()
{
  return podajDlBoku() * podajDlBoku() * podajDlBoku();
}
```

Zauważmy, że `dlBoku` jest polem prywatnym, zatem funkcje klasy pochodnej nie mają do niego dostępu. Dlatego w obliczeniach pola i objętości wywoływana jest funkcja `podajDlBoku`. Pole sześcianu to rzeczywiście sześciokrotność pola jednego boku, będącego kwadratem. A objętość to iloczyn pola podstawy i długości boku. Zwróćmy uwagę na użyte sformułowanie *sześciokrotność pola jednego boku* i *iloczyn pola podstawy i długości boku*. Czy można by w naszych obliczeniach użyć funkcji obliczania pola, zdefiniowanej w klasie `Kwadrat`? Przecież to ona właśnie służy do wyznaczania pola jednej ściany sześcianu, ta bowiem jest kwadratem....

Okazuje się, że w funkcjach klasy pochodnej możemy używać funkcji klasy bazowej, również tych *przedefiniowanych*. W przypadku tych funkcji pojawia się problem — ich nazwy są jednakowe. Spowodować to może problem przy rozstrzyganiu, która z wersji nas interesuje. Przypomnijmy jednak, że możemy korzystać z operatora zakresu i nazwy klasy — i tak użyjemy prefiksu `Kwadrat::` aby stwierdzić, że interesuje nas wersja funkcji z klasy `Kwadrat`, natomiast prefiksu `Szescian::` dla funkcji z tej klasy właśnie. Wykorzystamy tę właściwość do napisania drugiej — zdaniem autora lepszej — wersji funkcji obliczania pola i objętości sześcianu:

```
double Szescian::obliczPole()
{
  return 6 * Kwadrat::obliczPole();
}

double Szescian::obliczObjetosc()
{
  return Kwadrat::obliczPole() * podajDlBoku();
}
```

Dlaczego te wersje są lepsze? Po pierwsze — oddają one wspomniane wcześniej, intuicyjnie zrozumiałe sposoby wyznaczania pola i objętości sześcianu, wykorzystujące fakt iż bok (podstawa) jest kwadratem. Po drugie — korzystamy ze wzorów zdefiniowanych w klasie `Kwadrat`, nie przepisując ich ponownie.

Ostatecznie — klasa `Szescian` zajmuje się realizacją własnych obliczeń, wykorzystując odziedziczone właściwości klasy `Kwadrat`.

Dziedziczenie a konstruktory

Klasa pochodna dziedziczy wszystkie składowe każdej klasy podstawowej, z wyjątkiem konstruktorów, destruktorów[11]. Przykładowo, nie powiedzie się próba skompilowania przedstawionego niżej kodu:

```
Szescian kostka( 10 );

cout << "Szescian o boku: " << kostka.podajDlBoku() << endl;
cout << "       Objetosc: " << kostka.obliczObjetosc() << endl;
cout << "   Powierzchnia: " << kostka.obliczPole() << endl;
```

W klasie `Kwadrat` zdefiniowano konstruktor ogólny, nie zostanie on jednak aktywowany automatycznie dla obiektu klasy `Szescian`. W klasie pochodnej programista powinien zdefiniować konstruktory na nowo. Istnieje pewne rozluźnienie tej zasady, dotyczące konstruktorów domyślnych (bezparametrowych). Rozluźnienie to jest jednak mocno dyskusyjne, sam twórca języka — Brajne Stroustrup — namawia do definiowania również konstruktorów domyślnych klas pochodnych. Zapamiętajmy zatem: przy tworzeniu klasy pochodnej, programista zdefiniować powinien wszystkie niezbędne dla niej konstruktory.

Przy budowaniu klas pochodnych kierujemy się następującą zasadą: klasie pochodnej definiujemy metody do obsługi nowych pól, obsługę pól odziedziczonych realizujemy z wykorzystaniem metod odziedziczonych. Mimo, że konstruktory klasy pochodnej nie są jawnie dziedziczone, programista ma do nich dostęp. Może zatem aktywować je, i przy ich użyciu inicjować odziedziczone pola klasy bazowej. Deklaracja klasy `Szescian` z konstruktorem ogólnym (realizującym również funkcje konstruktora domyślnego) ma następującą postać:

```
class Szescian : public Kwadrat
{
  public:
    Szescian( double startowyBok = 0 );

    double obliczPole();
    double obliczObjetosc();
};

Szescian::Szescian( double startowyBok ) : Kwadrat( startowyBok )
{
}
```

Jakie zadanie realizuje ten konstruktor? Otrzymuje w postaci parametru startową długość boku, która powinna być wstawiona do pola `dlBoku`, zdefiniowanego w klasie `Kwadrat`. Przypomnijmy, że zadanie to realizuje konstruktor tej klasy, dbając o poprawność wstawianej wartości. W języku C++ zwykle nie

[11] Klasa pochodna nie dziedziczy również przeciążonych operatorów przypisania — tych jednak jeszcze w tym opracowaniu nie omawialiśmy.

wywołuje się jawnie konstruktorów, stąd mówimy raczej o *aktywowaniu* konstruktora na liście inicjalizacyjnej, a nie jego wywołaniu. Lista inicjalizacyjna jest legalnym miejscem aktywowania konstruktora klasy bazowej. Wykorzystanie konstruktora klasy bazowej polega zatem na umieszczeniu jego nazwy na liście inicjalizacyjnej konstruktora klasy pochodnej.

Po aktywowania konstruktora klasy `Szescian`, pierwszą czynnością będzie przetworzenie jego listy inicjalizacyjnej, a to spowoduje aktywację konstruktora klasy `Kwadrat`, któremu przekazujemy wartość parametru `startowyBok`. Dopiero po przetworzeniu listy inicjalizacyjnej, wykonane zostanie ciało konstruktora. W naszym przypadku jest ono puste, bowiem nie przewidujemy żadnych dodatkowych czynności w konstruktorze klasy `Szescian`[12]. W ten sposób, najpierw aktywowany zostanie konstruktor klasy bazowej, a potem dopiero ciało konstruktora klasy pochodnej.

I tak w podanym wcześniej przykładzie dla zdefiniowanego obiektu `kostka` aktywowany zostanie konstruktor parametrowy `Szescian` z parametrem 10. Konstruktor ten uaktywni — umieszczony na liście inicjalizacyjnej — konstruktor klasy `Kwadrat`, który ustawi startową wartość długości boku. Następnie zostaną wywołane funkcje oblicznia pola i objętości — oczywiście wywołana zostanie funkcja `obliczPole` zdefiniowana w klasie `Szescian`.

Redefinicja metod

Redefiniowanie funkcji składowych w klasach pochodnych pozwala na modyfikowanie ich działania tak, by było ono zgodne z wymaganiami stawianymi nowej klasie. W tak przedefiniowanej funkcji, można używać funkcji oryginalnej — odziedziczonej z klasy bazowej. Jednak takie działanie niesie za sobą pewne ryzyko. Załóżmy, że programista piszący funkcję `obliczPole` pomylił się i napisał ją następująco:

```
double Szescian::obliczPole()
{
  return 6 * obliczPole();
}
```

Cóż takiego napisał? Ano napisał, że powierzchnia sześcianu to sześciokrotność powierzchni... sześcianu! Brak nazwy kwalifikowanej `Kwadrat::` — a właśnie na jej pominięciu polegał błąd programisty — spowoduje, że funkcja `obliczPole` będzie wywoływała samą siebie! Mamy tutaj swoistą, niezamierzoną rekurencję, bez warunku jej zakończenia. Jak się zachowa program w takiej wersji? Proponujemy czytelnikom, aby sprawdzili to samodzielnie.

Zakres widoczności protected

W klasie pochodnej nie ma bezpośredniego dostępu do pól prywatnych klasy bazowej. Przypomnijmy, że aby np. obliczyć objętość sześcianu, musieliśmy odwoływać się do pola `dlBoku` klasy `Kwadrat` za pośrednictwem akcesora — funkcji `podajDlBoku`:

[12] Zwykle jednak konstruktor klasy pochodnej, po aktywowaniu konstruktora klasy bazowej, zajmuje się inicjalizacją dodatkowych pól klasy pochodnej. W przypadku klasy *Sześcian* takie pola nie występują.

```
double Szescian::obliczObjetosc()
{
  return Kwadrat::obliczPole() * podajDlBoku();
}
```

W niektórych sytuacjach jest to niewygodne i nienaturalne. W rozważanym przypadku tak jest rzeczywiście — funkcje składowe klasy `Szescian` nie mogą się odwoływać bezpośrednio do podstawowej informacji opisującej tę figurę! Można jednak tę sytuację zmienić i doprowadzić do tego, że klasa pochodna `Szescian` będzie miała dostęp do pola `dlBoku`. Wymaga to zmiany w deklaracji klasy `Kwadrat` — wystarczy zamienić nazwę sekcji, w której zadeklarowane jest pole `dlBoku` z `private` na `protected`:

— Składowe zadeklarowane jako `protected` są dostępne dla obiektów wszystkich klas pochodnych (tak jak składowe `public`).
— Składowe zadeklarowane jako `protected` są niedostępne dla obiektów innych, niezaprzyjaźnionych klas (tak jak składowe `private`).
— Specyfikator `protected` działa jak `private`, z tym wyjątkiem, że obiekty klas pochodnych otrzymują dostęp do składowych `protected` klasy bazowej.
— Pola i funkcje zadeklarowane w sekcji `protected` nazywane są *chronionymi*.

Deklaracja pola `dlBoku` jako chronionego wymaga zatem wykorzystania słowa kluczowego `protected`:

```
class Kwadrat
{
  . . .
  protected:
    double dlBoku;
};
```

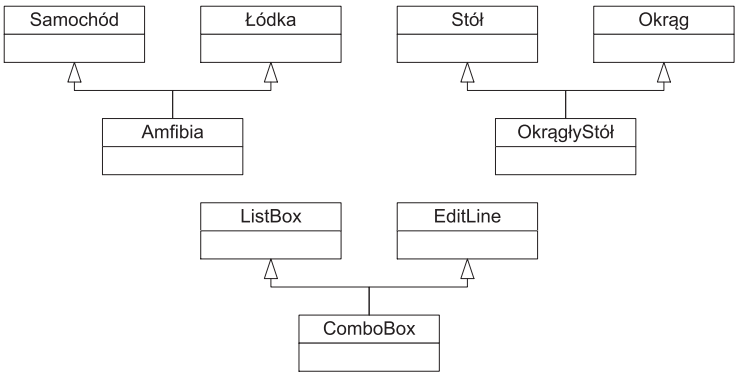
Do tak zadeklarowanego pola można się bezpośrednio odwoływać w funkcjach klasy pochodnej:

```
double Szescian::obliczObjetosc()
{
  return Kwadrat::obliczPole() * dlBoku;
}
```

W rozważanym przypadku takie bezpośrednie odwołanie jest wygodniejsze i naturalniejsze. Stosowanie pól chronionych wymaga przewidywania — w klasie bazowej należy użyć słowa kluczowego `protected` zamiast `private`.

1.2.5. Dziedziczenie jedno i wielobazowe

Przedstawiony w poprzednim podrozdziale przykład prezentował *dziedziczenie jednobazowe* — klasa pochodna posiada *jedną klasę bazową*. Często jednak zdarza się, że klasa, która chcemy utworzyć w naturalny sposób dziedziczy właściwości z więcej niż jednej klasy — przykłady prezentuje rysunek 1.9. Takie dziedziczenie nazywamy *wielobazowym*, pozwala ono na tworzenie klas potomnych, dziedziczących z więcej niż jednej klasy bazowej. Mając np. dwie, za-



Rysunek 1.9. Przykłady dziedziczenia wielobazowego

implementowane klasy, można zbudować nową klasę, dziedziczącą jednocześnie właściwości obu tych klas.

Dziedziczenie wielobazowe przedstawimy na przykładzie klasy reprezentującej okrągły stół, dziedziczące właściwości po klasach reprezentującej okrąg i stół. Załóżmy, że dane są dwie klasy[13]:

```
class Okrag
{
  public:
    Okrag( double r = 0 ) : promien( r ) {}

    double obliczPole()
    {
       return M_PI * promien * promien;
    }

  protected:
    double promien;
};

class Stol
{
  public:
    Stol( int ln = 0 ) : liczbaNog( ln ) {}

    int podajLiczbeNog()
    {
      return liczbaNog;
    }
```

[13] Dla poprawy czytelności przedstawiono tylko niezbędne elementy klas, brakuje w nich np. akcesorów i modyfikatorów.

```
  protected:
    int liczbaNog;
};
```

Wydaje się, że znaczenie pól i funkcji składowych obu klas jest intuicyjnie zrozumiałe. Na bazie tych dwóch klas tworzymy klasę `OkraglyStol`:

```
class OkraglyStol : public Okrag, public Stol
{
  public:
    OkraglyStol( double r = 0, int ln = 0 )
    : Okrag( r ), Stol( ln )
    {
    }
};
```

Klasa ta dziedziczy wszystkie pola i funkcje składowe obu klas bazowych. Możemy zatem dla obiektu klasy `OkraglyStol` zarówno wykorzystywać funkcję `obliczPole` odziedziczoną z klasy `Okrag` jak i funkcję `podajLiczbeNog` odziedziczoną z klasy `Stol`:

```
OkraglyStol stolik( 1, 3 ); // 1 - promien, 3 - lb. nog

cout << "\nLiczba nog : " << stolik.podajLiczbeNog();
cout << "\nPowierzchnia stolu : " << stolik.obliczPole();
```

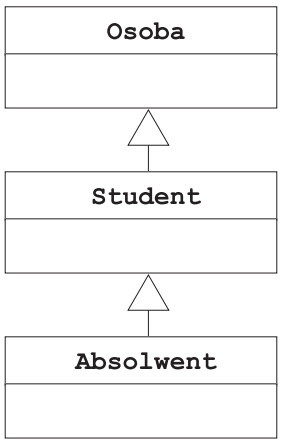
Zwróćmy uwagę na definicję obiektu `stolik` — aktywowany jest dla niego konstruktor klasy `OkraglyStol`. Ponieważ klasa ta dziedziczy po dwóch klasach, na liście inicjalizacyjnej tego konstruktora umieszczamy odwołania do dwóch konstruktorów. W trakcie przetwarzania listy inicjalizacyjnej następuje aktywowanie obu tych konstruktorów.

Kolejność aktywowania konstruktorów dla obiektu klasy pochodnej wynika z kolejności występowania nazw klas bazowych w deklaracji tej klasy. Nie jest istotna kolejność ich umieszczenia na liście inicjalizacyjnej konstruktora klasy pochodnej.

Przedstawiony przykład prezentuje elastyczność dziedziczenia wielobazowego — klasa pochodna nabyła w naturalny sposób cechy klas bazowych, osiągnęliśmy to bez pisania żadnego kodu (z wyjątkiem konstruktora oczywiście). Niestety, z dziedziczeniem wielobazowym wiąże się wiele problemów — np. w przypadku gdy klasy bazowe mają pola o jednakowych nazwach, gdy klasy bazowe same dziedziczą właściwości po wspólnej klasie bazowej. Wszystko to mocno komplikuje praktykę wykorzystania dziedziczenia wielobazowego. Omówienie tych problemów przekracza ramy tego opracowania, osobom zainteresowanym zgłębianiem tajników dziedziczenia wielobazowego polecamy książki [3, 7].

1.2.6. Konstruktory i destruktory a hierarchia klas

Dziedziczenie jest jednym z podstawowych i najczęściej stosowanych mechanizmów programowania obiektowego. Jego wykorzystanie prowadzi często do powstania rozbudowanej hierarchii klas — na jej szczycie jest klasa, która nie dziedziczy właściwości z żadnej innej klasy.



Rysunek 1.10. Przykładowa hierarchia klas

Budując nowe klasy z wykorzystaniem dziedziczenia, programista powinien zadbać o zdefiniowanie konstruktorów dla każdej nowej klasy. Jak wspomniano wcześniej, zwyczajowo konstruktor klasy pochodnej aktywuje konstruktor swej klasy bazowej poprzez umieszczenie jego nazwy na liście inicjalizacyjnej. Konstruktor klasy bazowej uruchamia się zatem *przed* ciałem konstruktora klasy bazowej. Można zatem powiedzieć, że konstruktory są aktywowane w kolejności dziedziczenia — od klasy bazowej ze szczytu hierarchii, poprzez kolejne klasy w hierarchii aż do ostatniej klasy pochodnej.

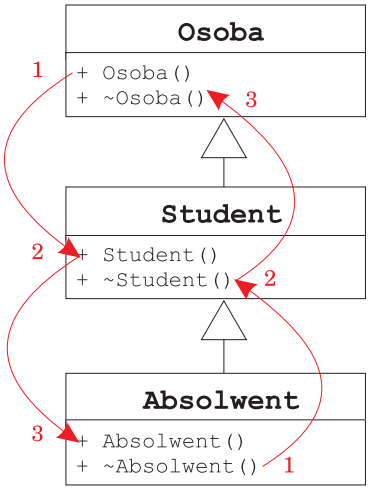
Klasy tworzące hierarchię mogą mieć zdefiniowane destruktory. W momencie usuwania obiektu klasy pochodnej zostanie automatycznie aktywowany jego destruktor, na następnie destruktor jego klasy bazowej. Jeżeli ta jest również klasą pochodną, następuje aktywowanie destruktora jej klasy bazowej, proces ten trwa tak długo, aż zostanie aktywowany destruktor klasy ze szczytu hierarchii klas.

Z aktywowaniem konstruktorów w hierarchii klas wiążą się pewne przypadki szczególne. Omówimy je na przykładzie. Załóżmy, że mamy trzy klasy o nazwach `Absolwent`, `Student`, `Osoba`. Klasa `Absolwent` jest klasą pochodną klasy `Student`, ta zaś jest pochodną klasy `Osoba`. Klasy tworzą zatem hierarchię przedstawioną na rysunku 1.10.

Zdefiniowane w tych klasach konstruktory będą uaktywniane zgodnie z porządkiem dziedziczenia — najpierw konstruktor klasy `Osoba`, potem klasy `Absolwent` a na końcu klasy `Student`, kolejność aktywowania destruktorów będzie odwrotna (kolejność i kierunek wywołań ilustruje rysunek 1.11).

Załóżmy, że klasa bazowa `Osoba` posiada jedno pole, będące jej unikatowym numerem. Załóżmy również, że każda z klas posiada własny konstruktor domyślny i ogólny, a także destruktor. Niech dla potrzeb tego przykładu, każda z tych funkcji wyprowadza do strumienia wyjściowego informację, że to ona właśnie działa:

```
class Osoba
{
```



Rysunek 1.11. Aktywowanie konstruktorów i destruktorów

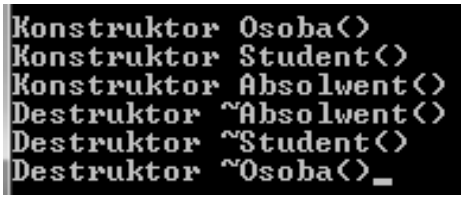
```
  public:
    Osoba() : idOsoby( 0 )
    {
      cout << "\nKonstruktor Osoba()";
    }

    Osoba( int id ) : idOsoby( id )
    {
      cout << "\nKonstruktor Osoba( " << id << " )";
    }

    ~Osoba()
    {
      cout << "\nDestruktor ~Osoba()";
    }

    int idOsoby;
};

class Student : public Osoba
{
  public:
    Student() : Osoba()
    {
      cout << "\nKonstruktor Student()";
    }

    Student( int id ) : Osoba( id )
```

```
    {
      cout << "\nKonstruktor Student( " << id << " )";
    }

    ~Student()
    {
      cout << "\nDestruktor ~Student()";
    }
};

class Absolwent : public Student
{
  public:
    Absolwent() : Student()
    {
      cout << "\nKonstruktor Absolwent()";
    }

    Absolwent( int id ) : Student( id )
    {
      cout << "\nKonstruktor Absolwent( " << id << " )";
    }

    ~Absolwent()
    {
      cout << "\nDestruktor ~Absolwent()";
    }
```

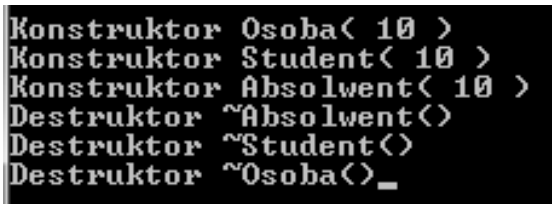
Załóżmy, że w obrębie pewnego bloku zdefiniowano obiekt `a` klasy `Absolwent`:

```
. . .
{
  Absolwent a;
  . . .
}
. . .
```

Na etapie definiowania obiektu `a` zostanie uaktywniony jego destruktor. Ponieważ na jego liście inicjalizacyjnej występuje uaktywnienie konstruktora jego klasy bazowej `Student`, to on powinien wykonać się wcześniej. Ale na jego liście inicjalizacyjnej występuje nazwa konstruktora klasy `Osoba`, zatem to on wykona się jako pierwszy — zgodnie z rysunkiem 1.11. W momencie zakończenia wykonania instrukcji blokowej, w obrębie której zdefiniowano obiekt `a`, obiekt ten będzie usuwany z pamięci operacyjnej. Spowoduje to aktywowanie jego destruktora, a potem destruktorów kolejnych klas bazowych, zgodnie z kolejnością przedstawioną na rysunku 1.11. Wynik działania opisywanego programu prezentuje rysunek 1.12, i rzeczywiście jest on zgodny z oczekiwaniami.



Rysunek 1.12. Aktywacja konstruktorów domyślnych i destruktorów



Rysunek 1.13. Aktywacja konstruktorów ogólnych i destruktorów

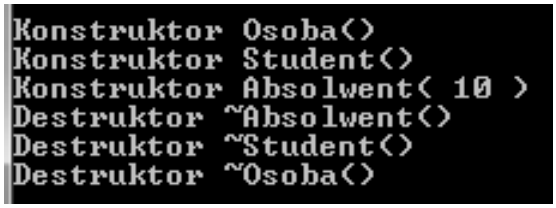
Podobnie będzie w przypadku definicji obiektu `a` z wykorzystaniem parametru inicjalizującego. Również nastąpi aktywowanie konstruktorów — tym razem ogólnych — zgodnie z hierarchią dziedziczenia (co ilustruje rysunek 1.13):

```
. . .
{
  Absolwent a( 10 );
  . . .
}
. . .
```

Na tym etapie rozważań nieodparcie nasuwa się spostrzeżenie, że w aktywowaniu konstruktorów nie ma nic tajemniczego — przecież każdy konstruktor aktywuje najpierw konstruktor klasy bazowej na swojej liście inicjalizacyjnej. Tak dzieje się kolejno, na coraz wyższych poziomach hierarchii klas. Zatem rzeczywiście jako pierwszy powinien wykonać się konstruktor klasy `Osoba`, bo ten nie posiada klasy bazowej.

Zaskoczyć nieco może natomiast automatyzm aktywowania destruktorów — żaden z destruktorów nie wywołuje i nie aktywuje innego destruktora, są one wywoływane automatycznie przez kompilator. Okazuje się jednak, że taki automatyzm działa również dla konstruktorów. Przeprowadźmy następujący eksperyment — usuńmy aktywację konstruktorów klas bazowych z listy inicjalizacyjnej konstruktorów domyślnych klas `Absolwent` i `Student`. Po rekompilacji i uruchomieniu zmodyfikowanego programu, konstruktory będą aktywowane tak jak poprzednio, a wynik działania programu będzie dokładnie taki jak na rysunku 1.12.

Kompilator zauważa, że konstruktory nie inicjują swojej kasy bazowej, i postanawia zrobić to za nas, wywołując automatycznie konstruktory kolejnych klas bazowych. Dlaczego zatem w prezentowanych przykładach mozolnie aktywuje-



Rysunek 1.14. Automatyczna aktywacja konstruktorów domyślnych

my konstruktory klas bazowych na liście inicjalizacyjnej? Bo tak powinno być — taka jest dobra praktyka programistyczna, nie pozostawiająca wątpliwości jakie są intencje programisty.

Uwaga — automatyzm wykorzystuje konstruktory domyślne. Jeżeli usuniemy aktywację konstruktora ogólnego klasy `Student` z listy inicjalizacyjnej konstruktora klasy `Absolwent`, to kompilator owszem, wywoła automatycznie konstruktory klas bazowych, ale *domyślne*! Wyniki działania tak zmodyfikowanego programu przykładowego prezentuje rysunek 1.14.

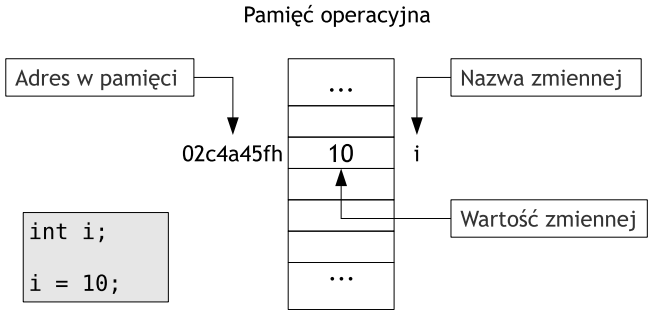
Jakie wnioski wypływają z rozważań przedstawionych w tym podrozdziale? Automatyzm aktywowania destruktorów jest korzystny i przewidywalny. Destruktor w danej klasie jest zawsze jeden, zatem nie ma wątpliwości, który zostanie wywołany. W przypadku konstruktorów jest inaczej — ich nazwy mogą być przeciążane, konstruktorów może być więcej. Dodatkowo w przypadku braku konstruktorów, kompilator syntetyzuje pusty, nie robiący nic pożytecznego konstruktor domyślny. To wszystko powoduje, że automatyczne aktywowanie konstruktorów może czasem być realizowane inaczej niż chcemy. Warto zatem nie pozostawiać kompilatorowi swobody, definiować odpowiednie konstruktory[14] i aktywować na ich liście inicjalizacyjnej konstruktory klas bazowych.

1.2.7. Zmienne wskaźnikowe, referencje i dynamiczny przydział pamięci

Niezależnie od przyjętej metody konstruowania programów, *zmienna* jest elementem programu rezydującym w pamięci operacyjnej, przeznaczonym do przechowywania wartości pewnego typu. Zmienna jest podstawowym elementem przechowującym dane, na których program operuje. Każda zmienna posiada swoją *nazwę* oraz *typ wartości*, jest przechowywana w pamięci operacyjnej, liczba zajętych bajtów zależy właśnie od typu zmiennej. Nazwa zmiennej *identyfikuje* zmienną w programie, zwalniając programistę od zastanawiania się, pod jakim adresem w pamięci zmienna jest zlokalizowana. Ilustruje to rysunek 1.15.

W języku C++ intensywnie wykorzystuje się pewien szczególny rodzaj zmiennych — *zmienne wskaźnikowe* oraz wyrażenia te zmienne wykorzystujące. Dokładne opanowanie zasad posługiwania się wskaźnikami jest niezbędne do efektywnego i sprawnego programowania w języku C++. Tej umiejętności nie można

[14] W omawianym przypadku można nie definiować w każdej z klas osobnego konstruktora domyślnego, a zastosować konstruktory ogólne z parametrem domyślnym o wartości 0 (jak w klasie *Kwadrat* na str. 29).



Rysunek 1.15. Zmienna jako obiekt w pamięci operacyjnej

pominąć, przeskoczyć lub zostawić na później. Koncepcja zmiennych wskaźnikowych oraz metody ich wykorzystania są proste, wymagają one jednak uwagi, zrozumienia i myślenia.

Zmienne wskaźnikowe

Zmienna wskaźnikowa przeznaczona jest do lokalizowania (inaczej wskazywania) obiektów w pamięci operacyjnej. Jedyną rolą zmiennej wskaźnikowej jest umożliwienie odwoływania się do obiektów wskazywanych. Obiektami wskazywanymi mogą być inne zmienne, funkcje oraz nienazwane obszary pamięci operacyjnej — zwykle przydzielane programowi dynamicznie, w trakcie jego działania (co zostanie omówione w dalszej części tego rozdziału). Zatem zmienna wskaźnikowa:

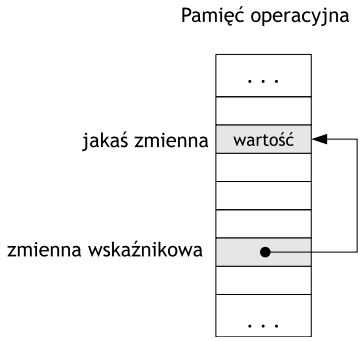
— sama rezyduje w pamięci operacyjnej;
— wobec powyższego, sama może być również „wskazywana" przez inną zmienną wskaźnikową;
— służy do lokalizowania w pamięci operacyjnej innych zmiennych, nienazwanych bloków pamięci oraz bloków zawierające kod programu, np. funkcji.

Koncepcję zmiennej wskaźnikowej ilustruje rysunek 1.16. Zwykle przyjmuje się, że zmienna wskaźnikowa zawiera w sobie adres obiektu wskazywanego, jednak nie musi ona w sobie zawierać adresu bezpośredniego (fizycznego) a inną informację, pozwalającą na precyzyjne i jednoznaczne zidentyfikowanie położenia obiektu w pamięci (np. tzw. *offset* w przypadku procesorów z rodziny Intel8086).

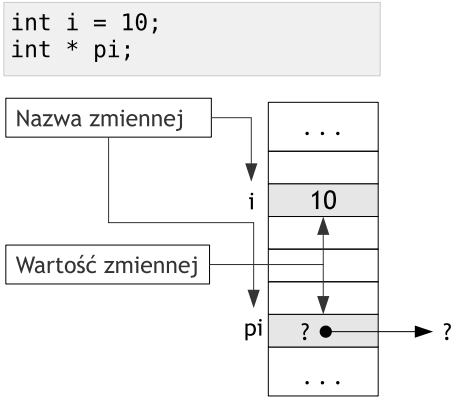
Jak definiuje się zmienne wskaźnikowe i jak się je wykorzystuje? Rozważmy następujący przykład:

```
int i = 10;
int * pi;
```

Deklaracja pierwszej zmiennej nie budzi wątpliwości — `i` to „zwykła" zmienna typu całkowitoliczbowego, zainicjowana wartością 10. W deklaracji zmiennej `pi` występuje nowy symbol '*'. Oznacza on, że zmienna `pi` jest *zmienną wskaźnikową*, przeznaczoną do wskazywania w pamięci operacyjnej obiektów typu `int`. Zawartość pamięci operacyjnej odpowiadającą powyższym deklaracjom prezentuje rysunek 1.17.



Rysunek 1.16. Koncepcja zmiennej wskaźnikowej



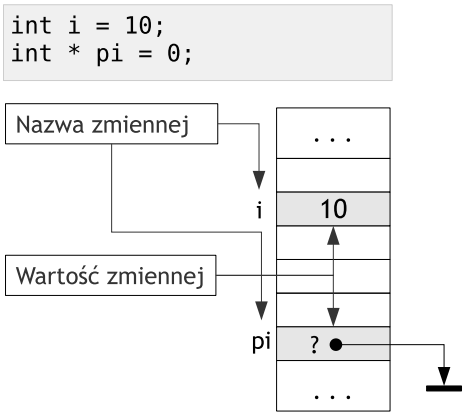
Rysunek 1.17. Deklaracja zmiennej wskaźnikowej

Zwróćmy uwagę, że tuż po deklaracji, zmienna wskaźnikowa ma wartość zależną od zasięgu w jakim została zdefiniowana. Jeżeli zmienna `pi` jest *automatyczną*, jej wartość jest przypadkowa. Dla zmiennej wskaźnikowej oznacza to, że wskazuje ona na przypadkowy obiekt w pamięci operacyjnej. Aby tak nie było, zwykle zmienne wskaźnikowe zeruje[15] się na etapie ich deklaracji. Tak zainicjowana zmienna wskaźnikowa nie wskazuje na żaden obiekt. Ilustruje to „uziemnienie" strzałki symbolizującej wskaźnik na rysunku 1.18.

Potrafimy już zatem zadeklarować zmienną wskaźnikową, zainicjować ją wartością oznaczającą brak powiązania z jakimkolwiek obiektem wskazywanym. W jaki sposób można powiązać taką zmienną z jakimś obiektem w pamięci? Przyjrzyjmy się bliżej następującym dwóm liniom kodu:

```
pi = &i;
*pi = 20;
```

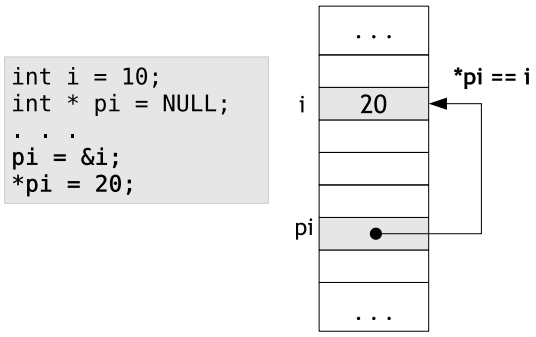
[15] W języku C wykorzystuje się symbol NULL jako wartość wskaźnika pustego — a więc, takiego który nie wskazuje niczego. W języku C++ zrezygnowano z wykorzystania tego symbolu, wykorzystuje się po prostu wartość 0.



Rysunek 1.18. Deklaracja wyzerowanej zmiennej wskaźnikowej

W pierwszej linii, po lewej stronie operatora przypisania występuje nazwa zmiennej wskaźnikowej `pi`, zatem do tej zmiennej zostania przypisana wartość występująca po prawej stronie przypisania. Co oznacza zapis `&i`? Jednoargumentowy operator `&` buduje wyrażenie wskaźnikowe lokalizujące zmienną `i` w pamięci operacyjnej. Zatem do zmiennej wskaźnikowej `pi` wstawiany jest wskaźnik do zmiennej `i`. Od tego momentu do obszaru pamięci zajmowanego przez zmienną `i` możemy się odwoływać wykorzystując jej nazwę lub za pośrednictwem wskaźnika zapisanego w zmiennej `pi`.

Takie właśnie odwołanie zawiera druga linia rozważanego przykładu: `*pi= 20`. Zapis `*pi` oznacza obiekt wskazywany przez `pi`, może on wystąpić wszędzie tam, gdzie może wystąpić `i`. Symbol `*` oznacza jednoargumentowy operator adresowania pośredniego, którego wynikiem jest obiekt wskazywany przez wskaźnik zapisany w zmiennej `pi`. Zatem powyższe przypisanie spowoduje wstawienie wartości 20 do zmiennej wskazywanej przez wskaźnik `pi`, a zatem do zmiennej `i`. Ilustruje to rysunek 1.19.



Rysunek 1.19. Wykorzystanie zmiennej wskaźnikowej

Zmienne wskaźnikowe mają szereg zastosowań, zwykle dzieli się je na następujące kategorie:
— realizacja przekazywania parametrów przez zmienną;
— wykorzystanie pamięci zarządzanej dynamicznie;
— manipulowanie tablicami;
— budowa rekurencyjnych struktur danych.

W tym opracowaniu zostaną przedstawione tylko wybrane zagadnienia związane z zastosowaniem zmiennych wskaźnikowych, dokładny opis zastosowań pozostałych znaleźć można w pozycjach [3, 7].

Zmienne wskaźnikowe a przekazywanie parametrów

W językach C i C++ przekazywanie parametrów do funkcji odbywa się *przez wartość*. Oznacza to, że parametr *aktualny* wywołania kopiowany jest do parametru *formalnego* funkcji. Wnętrze funkcji operuje na kopii wartości orginalnej, nie mogąc zmienić oryginalnej wartości parametru aktualnego wywołania funkcji.

W przedstawionym niżej przykładzie, parametrem aktualnym wywołania jest zmienna `a`. Jej wartość — a zatem liczba 5 — kopiowana jest do parametru formalnego funkcji `inc` o nazwie `i`. Wnętrze funkcji `inc` zwiększa wartość parametru `i` o jeden, operacja ta dotyczy jednak dokładanie tego parametru, wartość zmiennej `a` nie jest modyfikowana. Innymi słowy — operacja zwiększenia o jeden dotyczy, skopiowanej do parametru formalnego `i`, wartości 5. Zatem ten fragment programu wyprowadzi do strumienia wyjściowego niezmodyfikowaną wartość zmiennej `a` równią 5.

```
void inc( int i )
{
  i = i + 1;
}

. . .

int a = 5;
inc( a );

cout << a;
```

Aby funkcja `inc` mogła zwiększyć wartość zmiennej funkcji `a`, należy przekazać jej wskaźnik na tą zmienną. Wtedy wnętrze funkcji może odwołać się do oryginalnej wartości poprzez odwołanie pośrednie zrealizowane z wykorzystaniem symbolu funkcji `*`. Ilustruje to przedstawiony niżej, zmodyfikowany fragment programu. Zauważmy, że parametrem aktualnym wywołania funkcji jest `&a`, zatem funkcji przekazujemy wskaźnik do zmiennej `a`. Wskaźnik ten kopiowany jest do parametru `i`, który jest zmienną wskaźnikową. Zwiększenie o jeden dotyczy obiektu wskazywanego przez wskaźnik `i`, a tym obiektem jest właśnie zmienna `a`.

```
void inc( int * i )
{
```

```
  *i = *i + 1;
}

. . .

int a = 5;
inc( &a );

cout << a;
```

Ten mechanizm przekazywania wartości naśladuje spotykane np. w języku Pascal *przekazywanie parametrów przez zmienną*. Jednak wykorzystanie wskaźników jest w tym przypadku przez wielu uznawane za niewygodne, w języku C++ można skorzystać z parametrów będących *referencjami*.

Zmienne referencyjne

Omawianą powyżej funkcję `inc` można zrealizować bez wykorzystania wskaźników. W języku C++ wprowadzono zmienne referencyjne. Takie zmienne mogą „nakładać się" na inne zmienne, stanowiąc ich alternatywną nazwę. Zmienna referencyjna może być rozumiana jako *alias*, *alternatywna nazwa* jakiegoś obszaru pamięci. Dobrym przykładem jest właśnie referencyjna wersja funkcji `inc`:

```
void inc( int & i )
{
  ++i;
}

. . .

int a = 5;
inc( a );

cout << a;
```

W tym przykładzie zmienną o charakterze referencyjnym jest parametr formalny funkcji `inc`. W jego deklaracji występuję symbol `&`, oznaczający, że parametr `i` będzie nakładał się na obszar pamięci parametru aktualnego, z jakim zostanie wywołana funkcja `inc`. Rzeczywiście, jeżeli wywołamy tę funkcję z parametrem aktualnym `a`, parametr formalny `i` „nałoży" się na zmienną `a`, i każda modyfikacja `i`, będzie automatycznie dotyczyła zmiennej `a`.

Mechanizm referencji w języku C++ pozwala na unikanie stosowania zmiennych wskaźnikowych w niektórych sytuacjach. Referencje są trochę tak jak wskaźniki, których nie trzeba jawnie kotwiczyć o inne obiekty (operator `&`), i jawnie dokonywać ich dereferencji (operator `*`). Zmienne referencyjnie mogą występować nie tylko jako parametry formalne funkcji. Rozważmy następujący przykład:

```
int i = 10;
int & ri = i;
```

```
ri = 20;
```

Zmienna `ri` jest zmienną referencyjną, może być aliasem każdego obiektu typu `int`. Na etapie deklarowania, zmienną referencyjną *należy* zainicjować, jednak przypisanie występujące w jej deklaracji *nie dotyczy* wartości zmiennych — zapis `int & ri = i` oznacza, że zmienna `ri` odnosiła się do tej samej lokalizacji w pamięci, którą zajmuje zmienna `i`. Inaczej mówiąc — zmienna `ri` zostanie „nałożona" na zmienną `i`. Jak należy się spodziewać, każda modygikacja zmiennej `ri` jest jednocześnie modyfikacją zmiennej `i`.

Zmienne wskaźnikowe a obiekty

Czasem programista chce lub musi posługiwać się obiektem pewnej klasy nie za pośrednictwem jego nazwy, a za pośrednictwem zmiennych wskaźnikowych. Załóżmy, że wewnątrz pewnej funkcji `wypiszDane` musimy się odwołać do obiektu klasy `Kwadrat` (zobacz str. 32), przekazanego tej funkcji poprzez wskaźnik:

```
Kwadrat kwadr( 5 );
. . .
wypiszDane( &kwadr );
. . .
. . .
void wypiszDane( Kwadrat * k )
{
  cout << "\nDługość boku : " << (*k).podajDlBoku();
  cout << "\nPole kwadratu: " << (*k).obliczPole();
}
```

Parametr formalny funkcji `wypiszDane` jest wskaźnikiem, aby odwołać się do obiektu wskazywanego, należy dokonać dereferencji wskaźnika: `*k`. Po ujęciu takiego wyrażenia w nawiasy: `(*k)` reprezentuje już ono obiekt `kwadr`, do którego wskazanie było parametrem wywołania funkcji. Zatem po wyrażeniu `(*k)` możemy postawić kropkę i odwołać się do funkcji składowych obiektu klasy `Kwadrat`. Należy zwrócić uwagę na konieczność stosowania nawiasów — priorytety operatorów '.' i '*' są tak ustalone, że zapis:

```
*k.obliczPole()
```

oznaczałby, że `k` jest obiektem, a `obliczPole()` to wskaźnik, którego dereferencji chcemy dokonać za pośrednictwem operatora '*'.

Taki zapis odwołań wskaźnikowych do obiektów jest niewygodny. Ponieważ występuje on jednak w języku C++ bardzo często, wprowadzono specjalną notację dla wskaźników odwołujących się do rekordów (struktur) i obiektów. Zamiast pisać

```
(*k).obliczPole()
```

możemy napisać:

```
k->obliczPole()
```

Operator '`->`' pozwala na odwołania się do składowej rekordu lub obiektu, co pozwala np. zapisać funkcję `wypiszDane` w następujący sposób:

```
void wypiszDane( Kwadrat * k )
{
  cout << "\nDługość boku : " << k->podajDlBoku();
  cout << "\nPole kwadratu: " << k->obliczPole();
}
```

Dynamiczny przydział pamięci

Dynamiczny przydział pamięci polega na polega na rezerwowaniu fragmentu pamięci w obszarze tzw. *pamięci wolnej*. Programista decyduje o tym kiedy zarezerwuje blok pamięci, jaki on będzie miał rozmiar oraz kiedy zrezygnuje z jego wykorzystania, zwracając zarezerwowaną pamięć do puli obszarów wolnych. Pamięć wolna nazywana jest potocznie *stertą* (ang. *heap*). Jest to obszar pamięci:
— przeznaczony do przechowywania danych dynamicznych,
— kontrolowany ręcznie przez programistę,
— ograniczony pod względem rozmiaru,
— przydzielany pasującymi fragmentami.

Typowy scenariusz wykorzystania dynamicznego przydziału pamięci wygląda następująco:
— określenie wielkości potrzebnego obszaru pamięci;
— przydział pamięci i zapamiętanie wskazania tego obszaru w zmiennej wskaźnikowej;
— sprawdzenie czy przydział pamięci się powiódł, jeżeli tak to:
  — wykorzystanie przydzielonego bloku pamięci;
  — zwolnienie przydzielonego bloku pamięci, gdy nie jest już potrzebny.

W języku C++ można używać znanych z C funkcji `malloc`, `calloc`, ... Jednak w C++wprowadzono specjalne operatory zarządzające pamięcią: `new` oraz `delete`. Ich wykorzystanie jest zalecane, a w wielu przypadkach konieczne. Operatory są częścią języka i współpracują z mechanizmami kontroli typów. Zatem w języku C++ zarządzanie pamięcią dynamiczną realizować będzie:
— operator `new` — przydział pamięci (wraz z aktywowaniem odpowiedniego konstruktora),
— operator `delete` — zwolnienie pamięci (wraz z aktywowaniem destruktora, jeżeli go zdefiniowano).

Scenariusz przydziału pamięci dla obiektu klasy `Kwadrat` zawiera przedstawiony niżej przykład:

```
Kwadrat * kwadr;

kwadr = new Kwadrat( 5 );

if( kwadr != 0 )
{
```

```
  cout << "\nDługość boku : " << kwadr->podajDlBoku();
  cout << "\nPole kwadratu: " << kwadr->obliczPole();

  delete kwadr;
}
```

Instrukcja `kwadr = new Kwadrat( 5 )` przydziela pamięć dla obiektu klasy `Kwadrat`, inicjując go przy okazji aktywacją konstruktora ogólnego, z parametrem o wartości 5. Rezultatem działania operatora `new` jest wskaźnik na nowo utworzony obiekt, wskaźnik ten zapamiętywany jest w zmiennej `kwadr`. Instrukcja warunkowa testuje, czy wartość wskaźnika jest niezerowa — a więc, czy przydział pamięci się powiódł. Jeżeli tak jest, wykonujemy operacje na obiekcie wskazywanym przez zmienną `kwadr` a potem zwalniamy pamięć operatorem `delete`.

Ten schemat zakłada, że rezultatem operatora `new` będzie wartość 0 jeżeli przydział pamięci się nie powiedzie. Tak rzeczywiście było w starszych wersjach języka C++. Zgodnie z aktualnie obwiązaującym standardem, aby uzyskać takie zachowanie operatora `new`, należy posłużyć się jego specjalną wersją, i dokonać przydziału pamięci w następujący sposób:

```
. . .
kwadr = new (nothrow) Kwadrat( 5 );
. . .
```

Jeżeli nie posłużymy się specyfikacją `(nothrow)`, kompilator użyje wersji operatora `new`, która wygeneruje *wyjątek* `bad_alloc` oznaczający zazwyczaj brak pamięci dla nowego obiektu. Wyjątek taki należy obsłużyć używając instrukcji `try-catch`, co ilustruje podany niżej fragment kodu. Dokładny opis mechanizmu wyjątków i ich obsługi w języku C++ zawierają książki [7, 3].

```
Kwadrat * kwadr;

try
{
  kwadr = new Kwadrat( 5 );

  cout << "\nDługość boku : " << kwadr->podajDlBoku();
  cout << "\nPole kwadratu: " << kwadr->obliczPole();

  delete kwadr;
}
catch( bad_alloc )
{
   cout << "Brak pamięci dla programu!";
}
```

1.2.8. Polimorfizm i funkcje wirtualne

Dziedziczenie jest jednym z najistotniejszych mechanizmów programowania obiektowego. Dzięki niemu, nie trzeba budować nowych klas od podstaw, moż-

na bowiem wykorzystywać istniejące klasy jako bazę dla nowych klas. Nowe klasy budujemy rozszerzając klasy bazowe — dodając nowe dane i/lub funkcje składowe.

Stosowanie dziedziczenia prowadzi do powstania hierarchii klas — taką hierarchię można przedstawić graficznie, tak jak na rysunkach 1.8, 1.9, 1.10. Przyjrzyjmy się bliżej hierarchii klas przedstawionej na rysunku 1.10. Obiekty klas `Osoba`, `Student`, `Absolwent` są „spokrewnione" — ich klasy powiązane są związkiem dziedziczenia. Owo spokrewnienie oznacza również, że obiekty tych klas są do siebie *podobne*. Rzeczywiście, obiekt klasy `Student` *jest* jednocześnie obiektem klasy `Osoba`, obiekt klasy `Absolwent` *jest* jednocześnie obiektem klasy `Student`, ale również obiektem klasy `Osoba`. Dochodzimy w tym momencie do do bardzo istotnego spostrzeżenia — obiekty klas budujących hierarchię dziedziczenia mogą występować w różnych postaciach. Ta cecha programowania obiektowego nazywana jest *polimorfizmem* i ma wpływ na realizacje ważnych mechanizmów w programowaniu obiektowym. Słowo polimorfizm pochodzi od dwóch greckich słów *poly* czyli *wiele* oraz *morphos* czyli *postać* i oznacza *wielopostaciowość*. Oznacza to, że obiekt może przyjąć jedną z wielu dozwolonych dla niego postaci i wykazywać się innym zachowaniem w zależności od tego, w jakiej postaci występuje. Co to oznacza w praktyce i jak z tego skorzystać? Odpowiedź można znaleźć w dalszej części tego rozdziału.

Obiekty, wskaźniki, referencje a hierarchia klas

Załóżmy, że kod odpowiadający hierarchii klas przedstawionej na rysunku 1.10 ma następującą postać[16]:

```
class Osoba
{
  public:
    Osoba() : nrDowodu( 0 ) {}

    void piszKimJestes()
    {
      cout << "Osoba\n";
    }

   int nrDowodu;
};

class Student : public Osoba
{
  public:
    Student() : Osoba(), nrIndeksu( 0 ) {}

    void piszKimJestes()
    {
      cout << "Student\n";
```

[16] Realizacje klas zostały zmienione w stosunku do podanych wcześniej, istota rozważań bowiem jest inna.

```
    }

   int nrIndeksu;
};

class Absolwent : public Student
{
  public:
    Absolwent() : Student(), nrDyplomu( 0 )  {}

    void piszKimJestes()
    {
      cout << "Absolwent\n";
    }

   int nrDyplomu;
};
```

W klasie `Osoba` zdefiniowano pole `nrDowodu`[17], w klasie `Student` pole `nrIndeksu` a w klasie `Absolwent` pole `nrDyplomu`. Dla potrzeb klarowności tego przykładu pola te pozostawiamy publicznymi[18]. Każda z klas dokonuje redefinicji funkcji składowej `piszKimJestes`, tak aby do strumienia wyjściowego programu została skierowana nazwa odpowiedniej klasy. Rozważmy następujący przykład:

```
Osoba o;
Student s;
Absolwent a;

o.piszKimJestes();
s.piszKimJestes();
a.piszKimJestes();
```

Wynikiem jego działania będą informacje wyprowadzane do strumienia wyjściowego programu przez każdą z funkcji, zgodnie z klasą do której obiekt należy — ilustruje to rysunek 1.20.

Rysunek 1.20. Wyniki działania funkcji *piszKimJestes*

[17] W rzeczywistości pole opisujące numer dowodu osobistego nie powinno być typu całkowitoliczbowego. Numer dowodu zawiera zwykle symbolem literowe. Zastosowano typ całkowity dla uproszczenia rozważań.

[18] Zgodnie z zasadą hermetyzacji powinniśmy te pola uprywatnić i zdefiniować dla nich akcesory

Klasa `Student` jest klasą pochodną klasy `Osoba`, zatem każdy obiekt klasy `Student` jest jednocześnie obiektem klasy `Osoba`. Zatem następujące przypisanie:

```
o = s;
```

jest prawidłowe. Można przypisać do obiektu klasy bazowej obiekt klasy pochodnej. Jednak przy takim przypisaniu dokonywana jest konwersja typów, tracimy dostęp do składowych zdefiniowanych w klasie pochodnej, odwołanie:

```
o.nrIndeksu = 121212;
```

jest nieprawidłowe, bowiem obiekt `o` jest przecież reprezentantem klasy `Osoba`, a ta nie posiada pola `o.nrIndeksu`. Nieprawidłowe jest również przypisanie odwrotne:

```
s = o;
```

Gdyby było ono dozwolone, można by następnie próbować odwoływać się do `nrIndeksu` obiektu `s` — jego wartość byłaby nieokreślona, bowiem takiego pola nie ma w obiekcie `o`.

W podobny sposób możemy realizować odwołania za pośrednictwem wskaźników i referencji. Jeżeli istnieje wskaźnik do klasy bazowej, wolno nam do niego przypisać wskazanie na obiekt tejże klasy, jak również wskazanie na obiekt klasy pochodnej:

```
Osoba * wo;

wo = &o;
. . .
wo = &a;
. . .
wo = &s;
. . .
Osoba & r1 = o;
Osoba & r2 = s;
Osoba & r3 = a;
. . .
```

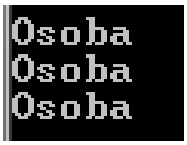
Wielopostaciowość a hierarchia klas

Skoro takie przypisania są możliwe, zastanówmy się, jaki będzie wynik działania następującego fragmentu kodu:

```
Osoba o;
Student s;
Absolwent a;

o.piszKimJestes();
```

```
o = s;
o.piszKimJestes();

o = a;
o.piszKimJestes();
```

Intuicja pozwala się spodziewać, że otrzymamy wyniki podobne do przedstawionych na rysunku 1.20 — po przypisaniu `o = s` obiekt `o` przyjmuje przecież postać obiektu klasy `Student`, a potem w wyniku przypisania `o = a` postać obiektu klasy `Absolwent`. Niestety, wynik działania przedstawionego wyżej prezentuje rysunek 1.21.



Rysunek 1.21. Wywołanie funkcji *piszKimJestes* dla obiektu *o*

Intuicja zawiodła — obiekt `o` zachowuje się jak obiekt klasy `Osoba`, mimo iż przypisaniem zasugerowaliśmy mu, żeby przyjął postać najpierw obiektu `s` klasy pochodnej `Student` a potem obiektu `a` klas pochodnej `Absolwent`. Okazuje się, że wspomniany wcześniej polimorfizm nie działa — powyższe przypisania *przekształcają* obiekty klas pochodnych w obiekt klasy bazowej, a informacja o ich różnorodności ginie.

Sprawdźmy, czy podobnie będzie w przypadku wykorzystania zmiennych wskaźnikowych, rozważmy następujący przykład:

```
Osoba * wo;

wo = &o;
wo->piszKimJestes();

wo = &s;
wo->piszKimJestes();

wo = &a;
wo->piszKimJestes();
```

Okazuje się, że nic się nie zmienia — wyniki działania powyższego kodu są analogiczne do wcześniejszych, przedstawionych przez rysunek 1.21. Mimo zakotwiczenia wskaźnika `wo` do obiektu klasy `Student`, wywołana zostanie funkcja `piszKimJestes` zgodnie z typem występującym w deklaracji tego wskaźnika.

Dlaczego tak jest? Zastanówmy się co robi kompilator w momencie natrafienia na linię w postaci:

```
wo->piszKimJestes();
```

Kompilator zauważa, że następuje wywołanie funkcji składowej `piszKimJestes`,

odnajduje ją sprawdzając typ wskaźnika, za pośrednictwem którego tę funkcję wywołujemy. W efekcie kompilator wstawia w tym miejscu kod wywołujący funkcję `Osoba::piszKimJestes`. Tak wywoływane są wszystkie funkcje, i to niezależnie, czy programujemy obiektowo czy nie. Zauważmy, że to, jak funkcja zostanie wywołana rozstrzyga się już na etapie kompilacji, taki rodzaj wywołania nazywa się *wywołaniem statycznym*. Inaczej mówimy, że następuje *wczesne wiązanie* wywołania z konkretną wersją wywoływanej funkcji.

Późne wiązanie i funkcje wirtualne

Kompilator musiałby działać zupełnie inaczej, gdyby miał wywołać funkcję zgodne *z klasą obiektu na który wskazuje* `wo` — a więc funkcję z klasy `Student` lub klasy `Absolwent`. Zadanie nie jest proste — kompilator musiałby śledzić na jaki obiekt wskazuje `wo`. Proces ten mógłby być złożony, a w niektórych przypadkach zupełnie nieskuteczny — często na etapie kompilacji nie wiemy jeszcze na z jakim obiektem powiązana będzie rzeczywiście zmienna wskaźnikowa. Trudno jest zatem związać wywołania funkcji z jej konkretną wersją *statycznie*. Czy w ogóle można zmusić kompilator, aby wywoływał funkcję zgodnie z klasą obiektu wskazywanego aktualnie przez wskaźnik? Okazuje się, że tak.

Wraz z programowaniem obiektowym wprowadzono do języków programowania tzw. *wiązanie dynamiczne*[19]. Polega ono na tym, że dopiero na etapie wykonania programu rozstrzyga się, która wersja funkcji składowej ma zostać wywołana. Taki rodzaj wiązania wywołania z określeniem funkcji wywoływanej nazywa się również *wiązaniem opóźnionym*.

Aby kompilator wywoływał funkcję `piszKimJestes` zgodne *z klasą obiektu na który wskazuje* `wo`, musimy poinformować o tym kompilator języka C++. Owo poinformowanie polega na umieszczeniu słowa kluczowego `virtual` przed deklaracją funkcji, która ma być wiązana dynamicznie. W naszym przypadku wystarczy umieścić słowo `virtual` w deklaracji funkcji `piszKimJestes` w klasie `Osoba`:

```
class Osoba
{
  public:
    . . .
    virtual void piszKimJestes()
    {
      cout << "Osoba\n";
    }
    . . .
};
```

Od tego momentu, we *wszystkich klasach pochodnych* funkcja `piszKimJestes` będzie podlegała wiązaniu dynamicznemu. W klasach pochodnych nie trzeba już używać słowa `virtual`. W efekcie wynik działania przykładu ze strony 56 będzie taki jak na rysunku 1.22. Funkcje zadeklarowane z wykorzystaniem słowa kluczowego `virtual` nazywamy *funkcjami wirtualnymi*.

[19] Wiązanie dynamiczne było oczywiście wykorzystywane w językach nieobiektowych, jednak była to raczej technika programowania (ang. *callback functions*) a nie możliwość syntaktycznie wbudowana w język programowania.

Rysunek 1.22. Wywołanie funkcji wirtualnej *piszKimJestes*

O tym, która wersja *funkcji wirtualnej* zostanie wywołana decyduje *typ obiektu* wskazywanego przez wskaźnik, lub *typ obiektu* do którego odnosi się referencja. Ta informacja ustalana jest na etapie wykonania programu, na podstawie dodatkowej informacji zapisanej w każdym obiekcie — *tablicy funkcji wirtualnych*. Szczegółowe informacje na temat koncepcji i realizacji tej tablicy można znaleźć w [7, 1], dla przykładów prezentowanych w tym opracowaniu szczegóły te nie są istotne. Każda funkcja zadeklarowana bez słowa kluczowego `virtual` będzie podlegała *wiązaniu statycznemu*, a więc wersja funkcji do wywołania ustalana będzie na podstawie typu wskaźnika lub zmiennej referencyjnej.

W języku C++ programista może zatem decydować, czy funkcja składowa będzie wiązana statycznie czy dynamicznie. Wywołania funkcji wiązanych statycznie są szybsze — w miejscu wywołania funkcji jej adres jest znany. Wywołania funkcji wiązanych dynamicznie będą wolniejsze — w miejscu wywołania należy najpierw odnaleźć adres właściwej funkcji, posługując się wspomnianą wcześniej tablicą funkcji wirtualnych. Nasuwa się pytanie — czy narzut czasu związany z wywołaniem funkcji wirtualnej jest znaczący? Odpowiedź nie jest jednoznaczna, zależy to bowiem od efektywności sprzętu na jakim działa program. Jednak w przypadku komputera osobistego w typowej, nowoczesnej lecz niewyrafinowanej konfiguracji, różnica czasu wywołania funkcji niewirtualnej i wirtualnej jest pomijalnie mała. Nie ma zatem istotnych powodów aby unikać funkcji wirtualnych z powodów wydajnościowych. Pojawia się jednak kolejne pytanie — jakie są korzyści stosowania metod wirtualnych? Odpowiedzi na to pytanie ma dostarczyć kolejny podrozdział.

Funkcje wirtualne w akcji — hierarchia klas

Aby przedstawić konkretne przykłady wykorzystania funkcji wirtualnych, rozwiniemy nieco wprowadzone wcześniej definicje klas `Osoba`, `Student` oraz `Absolwent`. Klasę `Osoba` uzupełnimy o pola przeznaczone do przechowywania imienia i nazwiska — dla uproszczenia pozostawimy pola publicznymi i wykorzystamy zdefiniowany w bibliotece standardowej typ `string`:

```
class Osoba
{
  public:
    Osoba()
    : imie( "" ), nazwisko( "" ), nrDowodu( 0 ) {}

    Osoba( string i, string n, int nr )
    : imie( i ), nazwisko( n ), nrDowodu( nr ) {}

    virtual void piszKimJestes() { cout << "\nOsoba\n"; }
```

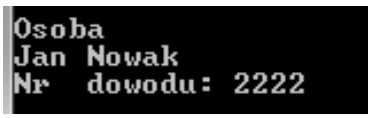
```
    virtual void wypiszDane()
    {
      piszKimJestes();
      cout << imie << " " << nazwisko << endl;
      cout << "Nr  dowodu: " << nrDowodu << endl;
    }

    int    nrDowodu;
    string imie;
    string nazwisko;
};
```

Konstruktor bezparametrowy klasy `Osoba` inicjuje obiekt wartościami domyślnymi — są to odpowiednio napisy puste dla pól przechowujących imię i nazwisko, oraz wartość 0 dla pola przechowującego numer dowodu. Konstruktor ogólny otrzymuje trzy parametry — imię przekazane parametrem `i`, nazwisko przekazane parametrem `n` oraz numer dowodu przekazany parametrem `nr`. Wartości tych parametrów są kopiowane na liście inicjalizacyjnej do odpowiednich pól obiektu klasy `Osoba`. Ciało konstruktora może pozostać puste. Klasa została wyposażona w nową funkcję składową `wypiszDane`, która wyprowadza do strumienia wyjściowego informacje o klasie obiektu (wywołanie funkcji składowej `piszKimJestes`) oraz zawartości pól przechowujących imię, nazwisko i numer dowodu. Zapamiętajmy, że obie funkcje składowe są funkcjami wirtualnymi.

Obiekty klasy `Osoba` możemy tworzyć dynamicznie w następujący sposób:

```
Osoba * wo = new Osoba( "Jan", "Nowak", 2222 );
. . .
wo->wypiszDane();
. . .
delete wo;
```

Obiekt tworzony operatorem `new` otrzymuje komplet danych za pośrednictwem parametrów konstruktora ogólnego. Funkcja składowa `wypiszDane` wyprowadza je do strumienia wyjściowego w postaci przedstawionej na rys. 1.23.



Rysunek 1.23. Wynik działania funkcji *wypiszDane* klasy *Osoba*

Na podstawie klasy `Osoba` budujemy klasę pochodną `Student`, posiadającą dodatkowe pole przechowujące informację o numerze indeksu. W klasie tej zdefiniowano konstruktor bezparametrowy, inicjujący obiekt wartościami domyślnymi oraz konstruktor posiadający cztery parametry — trzy pierwsze są analogiczne do otrzymywanych przez konstruktor parametrowy klasy `Osoba`, czwarty parametr inicjalizuje pole `nrIndeksu`. Zwróćmy uwagę, że do zainicjowania pól

odziedziczonych po klasie `Osoba` wykorzystywany jest konstruktor tejże klasy, aktywowany na liście inicjalizacyjnej konstruktora klasy `Student`. Również na liście inicjalizacyjnej otrzymuje swoją wartość pole `nrIndeksu`. Definicja klasy `Student` ma następująca postać:

```
class Student : public Osoba
{
  public:
    Student() : Osoba(), nrIndeksu( 0 ) {}

    Student( string i, string n, int nr, int nrI )
    : Osoba( i, n, nr ), nrIndeksu( nrI ) {}

    void piszKimJestes() { cout << "\nStudent\n"; }

    void wypiszDane()
    {
      Osoba::wypiszDane();
      cout << "Nr indeksu: " << nrIndeksu << endl;
    }

    int nrIndeksu;
};
```

W klasie `Student` dokonano redefinicji funkcji składowych `piszKimJesteś` oraz `wypiszDane`. Pierwszy przypadek został omówiony wcześniej. Drugą funkcję rozpoczyna wywołanie zdefiniowanej w klasie `Osoba` funkcji `wypiszDane`, której zadaniem jest wyprowadzenie danych właściwych dla tej klasy — imienia, nazwiska, numeru dowodu, po czym następuje wyprowadzenia do strumienia wyjściowego numeru indeksu, a więc informacji właściwej dla klasy `Student`.

W podobny sposób zbudujemy klasę `Absolwent`. Klasa ta powstanie na bazie klasy `Student`, zestaw jej pól zostanie rozszerzony o pole przeznaczone do przechowywania numeru dyplomu. Również w tej klasie dokonana zostanie redefinicja funkcji składowych `piszKimJesteś` oraz `wypiszDane`. Ta ostatnia rozpocznie swoje działanie od wywołania wersji zdefiniowanej w klasie `Student`, po czym wyprowadzi do strumienia wyjściowego informację o numerze dyplomu. Deklaracja klasy `Absolwent` ma następująca postać:

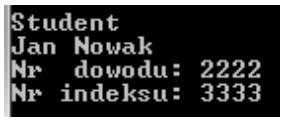
```
class Absolwent : public Student
{
  public:
    Absolwent() : Student(), nrDyplomu( 0 ) {}

    Absolwent( string i, string n, int nr, int nrI, int nrD )
    : Student( i, n, nr, nrI ), nrDyplomu( nrD ){}

    void piszKimJestes() { cout << "\nAbsolwent\n"; }

    void wypiszDane()
```

```
    {
      Student::wypiszDane();
      cout << "Nr dyplomu: " << nrDyplomu << endl;
    }

    int nrDyplomu;
};
```

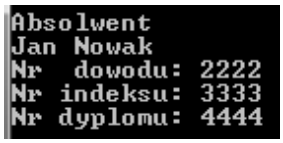
Klasy `Student` i `Absolwent` mogą być wykorzystane następująco:

```
Osoba * wo = new Student( "Jan", "Nowak", 2222, 3333 );
. . .
wo->wypiszDane();
. . .
delete wo;
. . .
Osoba * wo = new Absolwent( "Jan", "Nowak", 2222, 3333, 4444 );
. . .
wo->wypiszDane();
. . .
delete wo;
. . .
```

Zwróćmy uwagę, że mimo wykorzystania wskaźnika do klasy bazowej `Osoba`, każdorazowo wywoła się właściwa wersja funkcji `wypiszDane`, funkcja ta jest bowiem funkcją wirtualną. Wyniki działania wywołań tej funkcji prezentują rys. 1.24 oraz 1.25.



Rysunek 1.24. Wynik działania funkcji *wypiszDane* klasy *Student*



Rysunek 1.25. Wynik działania funkcji *wypiszDane* klasy *Absolwent*

Funkcje wirtualne w akcji — ewidencja osób

Załóżmy, że chcemy zbudować ewidencję osób, w której będą mogły występować obiekty każdej ze zdefiniowanych wyżej klas. Ewidencję tę realizować będzie klasa `EwidencjaOsob`. Zanim przedstawimy szczegóły jej realizacji, zobaczmy

jaka będzie koncepcja jej wykorzystania. Rozpocząć musimy od zdefiniowania obiektu klasy `EwidencjaOsob`:

```
EwidencjaOsob ewidencja;
```

Obiekt `ewidencja` będzie zarządzał przechowywanymi obiektami. Funkcja składowa:

```
void dodaj( Osoba * wo );
```

będzie dopisywała obiekt do ewidencji. Jej parametrem jest wskaźnik na obiekt klasy `Osoba`. Wiemy już, że taki wskaźnik może przyjmować również wskazania na obiekty dowolnej klasy pochodnej od `Osoba`. Zatem funkcję tę będzie można wywołać w następujący sposób:

```
ewidencja.dodaj( new Osoba( "Andrzej", "Kowalski", 2222 ) );
ewidencja.dodaj( new Student( "Jan", "Nowak", 3333, 4444 ) );
ewidencja.dodaj( new Absolwent( "Anna","Nowak",5555,6666,7777 ) );
```

Spowoduje to dopisanie do ewidencji trzech obiektów, dla każdego z nich pamięć jest przydzielana dynamicznie operatorem `new`[20]. Klasa `EwidencjaOsob` zapamiętuje wskaźnik na przydzielony dynamicznie obiekt. W każdym momencie możemy sprawdzić ile obiektów jest w ewidencji, służy do tego funkcja składowa `podajLbOsob`:

```
cout << "\nLiczba osob: " << ewidencja.podajLbOsob() << endl;
```

Do pobierania informacji o osobach zapisanych w ewidencji służy funkcja składowa:

```
Osoba * podajOsobe( int nrOsoby )
```

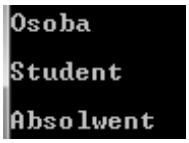
której rezultatem jest wskaźnik na obiekt o numerze przekazanym parametrem `nrOsoby`. W przypadku podanie numeru z poza zakresu, rezultatem tej funkcji będzie wskaźnik pusty. Pierwsza osoba dodana do ewidencji otrzyma numer 0, druga 1, numer ostatniej osoby w ewidencji wyznaczamy z pomocą przedstawionej wyżej funkcji: `ewidencja.podajLbOsob() - 1`. Pozwoli nam to na programowe przemaszerowanie po elementach ewidencji i poproszenie każdego z obiektów o wyprowadzenie do strumienia wyjściowego informacji o klasie, której jest reprezentantem:

```
for( int i = 0; i < ewidencja.podajLbOsob(); i++ )
  ewidencja.podajOsobe( i )->piszKimJestes();
```

Iteracja `for` rozpoczyna swoje działanie z wartością zmiennej `i` równą zero — jest to numer potencjalnego, pierwszego elementu w ewidencji. Potencjalnego, bowiem gdyby ewidencja była pusta, warunek iteracji byłby fałszywy, i iteracja zostałaby zakończona. W naszej ewidencji posiadamy trzy obiekty, zatem iteracja wykona się trzykrotnie.

[20] Dla klarowności przykładu nie kontrolujemy poprawności przydziału pamięci.

W ciele iteracji następuje wywołanie funkcji składowej `podajOsobe`. Zmienna `i` zawiera w każdym przebiegu numer kolejnej osoby zapisanej w ewidencji. Rezultatem wywoływanej funkcji jest wskaźnik na ewidencjonowany obiekt, możemy zatem dla rezultatu funkcji wywołać funkcję składową `piszKimJestes()`. Ponieważ jest to funkcja wirtualna, każdy z obiektów wywoła własną wersję tej funkcji, co ilustruje rys. 1.26.



Rysunek 1.26. Wynik działania wywołań funkcji *piszKimJestes*

Ciało iteracji `for` można zapisać inaczej — z wykorzystaniem roboczego wskaźnika `wo` zapamiętującego rezultat funkcji. Możemy również sprawdzić, czy ten rezultat nie jest wskaźnikiem pustym:

```
for( int i = 0; i < ewidencja.podajLbOsob(); i++ )
{
  Osoba * wo = ewidencja.podajOsobe( i );
  if( wo )
    wo->piszKimJestes();
}
```

W przypadku właściwej realizacji klasy `EwidencjaOsob` oraz ostrożnego manipulowania zakresem zmiennej sterującej `i` kontrolowanie rezultatu funkcji nie jest konieczne, pierwsza wersja iteracji `for` jest wystarczająca.

W analogiczny sposób możemy zapisać iterację wyprowadzającą do strumienia wyjściowego dane każdego obiektu zapisanego w ewidencji:

```
for( int i = 0; i < ewidencja.podajLbOsob(); i++ )
  ewidencja.podajOsobe( i )->wypiszDane();
```

Ponieważ funkcja `wypiszDane` jest wirtualną, w trakcie każdego przebiegu iteracji zostanie wywołana wersja funkcji właściwa dla obiektu zapisanego w ewidencji, co ilustruje rys. 1.27.

Podsumujmy — obiekt klasy `EwidencjaOsob` zarządza zbiorem obiektów. Do ewidencji można dodawać obiekty klasy `Osoba` lub dowolnej klasy pochodnej. Ewidencja ta jest zatem *polimorficzna* — zawierać może obiekty występujące w różnych postaciach, wspólnym mianownikiem jest przynależność do wspólnej hierarchii klas. Jeżeli w każdej z klas potomnych klasy `Osoba` dokonana zostanie redefinicja funkcji wirtualnych, mimo owej polimorficzności, każdy z ewidencjonowanych obiektów zachowa zdolność do własnego zachowania — mimo iż operujemy wskaźnikami do klasy `Osoba`, wywołują się funkcje właściwe dla klas każdego z obiektów.

Przyjrzyjmy się teraz bliżej klasie `EwidencjaOsob`. Do przechowywania informacji o ewidencjonowanych obiektach wykorzystana zostanie tablica wskaźników na obiekty klasy `Osoba`. Tablica ta — o nazwie `tabOsob` jest chronionym

Rysunek 1.27. Dane obiektów zapisanych w ewidencji

polem klasy `EwidencjaOsob`. Jej maksymalny rozmiar określa statyczne pole `maksLbOsob`, będące zmienną typu `const int` o wartości 20. Liczbę obiektów aktualnie przechowywanych w ewidencji zawiera pole `lbOsob`. Pola te zostały zadeklarowane w sekcji `protected` klasy, tak by były dostępne dla potencjalnych klas pochodnych. Sekcja chroniona klasy `EwidencjaOsob` ma zatem następującą postać:

```
class EwidencjaOsob
{
   . . .
  protected:
    static const int maksLbOsob = 20;
    Osoba * tabOsob[ maksLbOsob ];
    int lbOsob;
};
```

Klasa powinna posiadać swój konstruktor. W naszym przypadku będzie on jedynie inicjował wartością 0 pole `lbOsob`. Oznacza to, że w tablicy `tabOsob` o maksymalnej liczebności `maksLbOsob` nie zapisano jeszcze żadnego obiektu — ewidencja jest pusta:

```
EwidencjaOsob::EwidencjaOsob() : lbOsob( 0 )
{
}
```

Dopisywanie obiektów do ewidencji będzie polegało na wstawianiu wskaźnika na dopisywany obiekt w pierwsze puste miejsce tablicy, pod warunkiem że takie istnieje. Realizuje to funkcja `dodaj` o następującej deklaracji:

```
void EwidencjaOsob::dodaj( Osoba * wo )
{
  if( lbOsob < maksLbOsob - 1 )
    tabOsob[ lbOsob++ ] = wo;
}
```

Jeżeli w tablicy `tabOsob` istnieje wolne miejsce, wartość zmiennej `lbOsob` określa indeks tego wolnego elementu. Dzieje się tak, ponieważ obiekty nume-

rowane są od 0, zatem zmienna ta zawiera dwie istotne informacje — liczbę obiektów aktualnie zapisanych w ewidencji, a jednocześnie numer pierwszego wolnego miejsca w tablicy `tabOsob`. Wskaźnik przekazany parametrem `wo` zostaje w to miejsce zapisany, wartość zmiennej `lbOsob` jest inkrementowana.

Funkcja `podajOsobe` posiada następującą implementację:

```
Osoba * EwidencjaOsob::podajOsobe( int nrOsoby )
{
  if( nrOsoby >= 0 && nrOsoby < lbOsob )
    return tabOsob[ nrOsoby ];
  else
    return 0;
}
```

Jeżeli — tak jak w rozważanym wyżej przypadku — obiekty wstawiane do ewidencji są przydzielane dynamicznie, można je usunąć wywołaniem funkcji `usun`:

```
void EwidencjaOsob::usun()
{
  while( --lbOsob >= 0 )
    delete tabOsob[ lbOsob ];
  wyczysc();
}
```

Iteracja `while` fizycznie usuwa obiekty w kolejności od ostatniego do pierwszego. Po usunięciu obiektów zmienna `lbOsob` powinna zostać wyzerowana, ewidencja jest bowiem pusta. Realizuje to funkcja `wyczysc`, która wyłącznie zeruje zmienną przechowującą aktualną liczbę osób:

```
void EwidencjaOsob::wyczysc()
{
  lbOsob = 0;
}
```

Wywołanie funkcji `usun` powoduje zatem fizyczne usunięcie zarządzanych obiektów z pamięci operacyjnej. Funkcja `wyczysc` powoduje wyzerowanie liczby przechowywanych obiektów bez ich usuwania z pamięci.

Klasa `EwidencjaOsob` może wykonywać operacje na przechowywanych obiektach, przykładowo funkcja `wypiszWszystko` wyprowadza do strumienia wyjściowego informację o każdym obiekcie zapisanym w ewidencji:

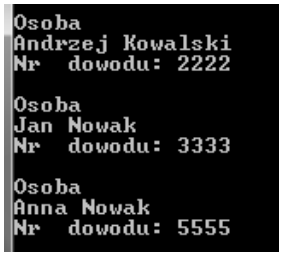
```
void EwidencjaOsob::wypiszWszystko()
{
  for( int i = 0; i < lbOsob ; i++ )
    tabOsob[ i ]->wypiszDane();
}
```

Zwróćmy ponownie uwagę, że informacje wyprowadzane do strumienia wyjściowego programu zależą od klasy obiektu wskazywanego przez element tablicy `tabOsob[ i ]`. Dzieje się tak, ponieważ funkcja `wypiszDane` jest wirtualna. Dzięki temu rezultat działania funkcji `wypiszWszystko` wywołanej dla obiektu

`ewidencja` w rozważanym wcześniej przykładzie będzie taki jak na rys. 1.27. Przypomnijmy:

```
EwidencjaOsob ewidencja;
. . .
ewidencja.dodaj( new Osoba( "Andrzej", "Kowalski", 2222 ) );
ewidencja.dodaj( new Student( "Jan", "Nowak", 3333, 4444 ) );
ewidencja.dodaj( new Absolwent( "Anna","Nowak",5555,6666,7777 ) );
. . .
ewidencja.wypiszWszystko();
```

Zauważmy, że usunięcie słowa `virtual` z definicji obu funkcji składowych klasy `Osoba` zmienia wszystko — funkcje są wiązane statycznie a wynik działania powyższej funkcji byłby taki jak na rys. 1.28.



Rysunek 1.28. Działanie funkcji *wypiszWszystko* dla wiązania statycznego

Bez funkcji wirtualnych znika polimorfizm — wszystkie ewidencjonowane obiekty zachowują się tak, jakby były obiektami klasy `Osoba`. Polimorfizm nie jest zatem cechą klasy `EwidencjaOsob` a właściwością hierarchii klas, zbudowanej na klasie `Osoba`. Stosowanie polimorfizmu wymaga przewidywania — w trakcie budowania klas należy przewidzieć czy będą one wykorzystywane jako klasy bazowe, oraz jakie funkcje składowe powinny być wirtualnymi.

Poniżej przedstawiona została kompletna deklaracja klasy `EwidencjaOsób` oraz definicje dwóch, nieomówionych wcześniej funkcji składowych.

```
class EwidencjaOsob
{
  public:
    EwidencjaOsob();

    void dodaj( Osoba * wo );
    Osoba * podajOsobe( int nrOsoby );
    void wypisz( int nrOsoby );
    void wypiszWszystko();
    int podajLbOsob();
    void usun();
    void wyczysc();

  protected:
```

```
    static const int maksLbOsob = 20;
    Osoba * tabOsob[ maksLbOsob ];
    int lbOsob;
};

void EwidencjaOsob::wypisz( int nrOsoby )
{
  if( nrOsoby >= 0 && nrOsoby < lbOsob )
    tabOsob[ nrOsoby ]->wypiszDane();
}

int EwidencjaOsob::podajLbOsob()
{
  return lbOsob;
}
```

Konwersje wskaźników do obiektów

Obiektami zapisanymi w ewidencji osób można manipulować. Załóżmy, że dla ewidencji osób przedstawionej na str. 66 chcemy zmienić imię studenta Nowaka z Jan na Janusz. Dostęp do drugiego obiektu w ewidencji uzyskamy za pośrednictwem wywołania funkcji `podajOsobe` z numerem 1.

```
Osoba * wo = ewidencja.podajOsobe( 1 );
if( wo )
  wo->imie = "Janusz";
```

W podobny sposób zmienimy nazwisko i numer dowodu każdej z osób zapisanych w ewidencji. Jeżeli spróbujemy jednak zmienić w analogiczny sposób numer indeksu:

```
wo->nrIndeksu = 1234;
```

kompilator zgłosi błąd kompilacji, mówiący o tym, że klasa `Osoba` nie posiada pola `nrIndeksu`. To prawda, posługujemy się bowiem wskaźnikiem do klasy `Osoba` — taki jest rezultat funkcji `podajOsobe`. Spróbujmy zatem użyć wskaźnika do klas `Student`:

```
Student * ws = ewidencja.podajOsobe( 1 );
if( ws )
  ws->nrIndeksu = 1234
```

Niestety, mimo iż rzeczywiście drugi obiekt w ewidencji jest obiektem klasy `Student`, kompilator nie pozwoli na bezpośrednie przypisanie wskaźnika klasy `Osoba` do wskaźnika klasy `Student` — zostanie zgłoszony błąd kompilacji. Aby takie przypisanie mogło być zrealizowane, programista musi dokonać jawnej konwersji wskaźnika będącego rezultatem funkcji `podajOsobe`:

```
Student * ws = ( Student * )ewidencja.podajOsobe( 1 );
if( ws )
  ws->nrIndeksu = 1234
```

W pierwszej linii powyższego przykładu następuje przekształcenie typu wskaźnika będącego rezultatem wywołania funkcji `podajOsobe`. Zapis `( Student * )` oznacza, że rezultat funkcji ma zostać przekształcony do postaci wskaźnika do klasy `Student`. Po tak przeprowadzonej konwersji typu wskaźnika, może on zostać przypisany do zmiennej wskźnikowej `ws`. Tak realizowana konwersja, zwana również *rzutowaniem*, pochodzi wprost z języka C.

Dokonana konwersja wskaźnika pozwala na późniejsze odwoływanie się do składowych obiektu klasy `Student`. Konwersja taka jest jednak bezpieczna tylko wtedy, gdy rzeczywiście wskazywany obiekt należy do klasy `Student`. Załóżmy, że w powyższym przykładzie wywołujemy funkcję `podajOsobe` z parametrem 0. Pierwszy element w ewidencji jest obiektem klasy `Osoba`, po dokonaniu konwersji na wskaźnik do klasy `Student` otrzymamy możliwość odwołania się do pola `nrIndeksu`, którego w obiekcie klasy `Osoba` nie ma! Takie odwołania do nieistniejących elementów klasy są zawsze błędne, ich efekty są trudno przewidywalne i mocno zależne od środowiska systemowego. We większości przypadków spowodują one awaryjne zakończenie programu, chociaż nie jest wykluczone, że program nie zostanie zakończony i będzie działał dalej, zwykle błędnie.

Aby uniknąć tego typu błędów, należy opracować mechanizm identyfikacji klas obiektów przechowywanych w ewidencji, pozwalający na stwierdzenie, czy mamy do czynienia z obiektem klasy `Osoba`, `Student` czy `Absolwent`. Taka identyfikacja może zostać zrobiona na dwa sposoby:

— poprzez wbudowanie przez programistę w poszczególne klasy informacji o ich rodzaju,
— poprzez wykorzystanie mechanizmów języka C++ pozwalających na identyfikowanie typów w czasie wykonania programu.

Każdy z tych sposobów posiada swoje wady i zalety, oba rozwiązania przedstawione są w kolejnych podrozdziałach.

Identyfikacja typów zdefiniowana przez programistę

To rozwiązanie zakłada, że programista wyposaża tworzone przez siebie klasy w elementy pozwalające na identyfikację klasy obiektu w trakcie wykonania programu. Nie istnieje jedynie słuszne rozwiązanie tego problemu, istnieje kilka możliwości. Przedstawione dalej rozwiązanie wykorzystuje funkcje wirtualne. Jego podstawowym elementem będzie typ wyliczeniowy, którego wartości będą odpowiadały nazwom poszczególnych klas wchodzących w skład hierarchii dziedziczenia:

```
enum RodzajeOsob
{
  roOsoba,
  roStudent,
  roAbsolwent
}
```

Prefiks `ro` to akronim sformułowania *rodzaj osoby*. Każdą z klas w hierarchii uzupełnimy o funkcję `rodzaj`, której rezultatem będzie jedna z wartości powyższego typu wyliczeniowego:

```
class Osoba
{
  public:
    . . .
    virtual int rodzaj() { return roOsoba; }
    . . .
};

class Student : public Osoba
{
  public:
    . . .
    int rodzaj() { return roStudent; }
    . . .
};

class Absolwent  : public Student
{
  public:
    . . .
    int rodzaj() { return roAbsolwent; }
    . . .
};
```

Przypomnijmy, że rozważania przedstawione w tym rozdziale dotyczą ewidencji o strukturze utworzonej w przykładzie przedstawionym na str. 66. Zauważmy, że funkcja `rodzaj` jest wirtualna. Tak zdefiniowane funkcje pozwolą na identyfikację klasy obiektu w trakcie wykonania programu:

```
for( int i = 0; i < ewidencja.podajLbOsob(); i++ )
{
  Osoba * wo = ewidencja.podajOsobe( i );
  if( wo->rodzaj() == roOsoba )
    cout << "\nObiekt klasy Osoba";
  if( wo->rodzaj() == roStudent )
    cout << "\nObiekt klasy Student";
  if( wo->rodzaj() == roAbsolwent )
    cout << "\nObiekt klasy Absolwent";
}
```

Identyfikacja klasy obiektu pozwala nam na bezpieczne odwoływanie się do składowych jemu właściwych:

```
Osoba * wo = ewidencja.podajOsobe( 1 );
if( wo->rodzaj() == roStudent )
{
  Student * ws = ( Student * ) wo;
  ws->nrIndeksu = 1234;
}
```

Powyższy fragment można zapisać bez wykorzystania pomocniczego wskaźnika do klasy `Student`, bezpośrednio odwołując się do wskaźnika `wo` po jego konwersji:

```
Osoba * wo = ewidencja.podajOsobe( 1 );
if( wo->rodzaj() == roStudent )
  ( ( Student * ) wo )->nrIndeksu = 1234;
```

Zaprezentowane rozwiązanie identyfikacji klasy obiektu w trakcie wykonania programu bazuje na funkcjach wirtualnych. Jest to rozwiązanie proste, skuteczne i efektywne czasowo[21]. Wymaga ono jednak przewidywania — już na etapie konstruowania klasy bazowej trzeba przewidzieć taką możliwość oraz konsekwentnie redefiniować odpowiednią funkcję wirtualną w klasach pochodnych.

Identyfikacja typów jako element języka

Problem identyfikowania typów obiektów w czasie wykonania programu dostrzeżono w trakcie prac nad rozwojem języka C++, rozszerzając go o taki właśnie mechanizm.

Konwersja typów może być realizowana poprzez wykorzystanie operatorów konwersji. Pierwszy z nich zastępuje klasyczną konwersję wywodzącą się z języka C, stosowaną w poprzednim podrozdziale. Zamiast konwersji w postaci:

```
Student ws = ( Student * )ewidencja.podajOsobe(1);
```

piszemy

```
Student ws = static_cast< Student * >(ewidencja.podajOsobe(1));
```

Operator `static_cast` działa podobnie jak rzutowanie pochodzące z języka C[22]. Operator ten nie realizuje żadnej kontroli typów w trakcie działania programu i jego stosowanie stwarza podobne niebezpieczeństwa jak zwykła konwersja typów.

Z tego powodu wprowadzono w języku C++ operator `dynamic_cast`. Operator ten pozwala na konwersję wskaźników i referencji na obiekty polimorficzne z kontrolą, czy taka konwersja może być przeprowadzona. W przedstawionym niżej przykładzie rezultat funkcji `podajOsobe` — a więc wskaźnik na obiekt klasy `Osoba` — podawany jest na wejście operatora `dynamic_cast`. Jeżeli możliwa jest konwersja do postaci wskaźnika na klasę pochodną `Student`, operator taką konwersję realizuje. Jeżeli konwersja nie może zostać zrealizowana, rezultatem operatora jest wartość zerowa. Konwersja taka jest możliwa, jeżeli wskaźnik aktualnie wskazuje na obiekt klasy `Student` lub obiekt jej klasy pochodnej.

```
Student * ws = dynamic_cast< Student * >(ewidencja.podajOsobe(1));
if( ws != 0 )
  ws->nrIndeksu = 1234;
```

[21] Z uwzględnieniem oczywiście narzutu czasowego związanego z wywołaniem funkcji wirtualnej.

[22] W istocie najlepszym odpowiednikiem klasycznego rzutowania wywodzącego się z języka C jest operator *reinterpret_cast*, który dokonuje określonych przez programistę konwersji, również ryzykownych i potencjalnie niebezpiecznych.

W powyższym przykładzie, drugi element w ewidencji jest rzeczywiście obiektem klasy `Student`, zatem konwersja jest możliwa. Gdyby funkcja `podajOsobe` została wywołana z parametrem 0, rezultatem operatora byłaby wartość zerowa — pierwszy element w ewidencji jest obiektem klasy `Osoba` i konwersja wskaźnika nie jest możliwa. Zwróćmy uwagę, że konwersja powiedzie się również w przypadku wywołania funkcji `podajOsobe` z parametrem 2. Trzeci element ewidencji jest bowiem obiektem klasy `Absolwent`, a ta jest pochodną klasy `Student`. Trzeci obiekt w ewidencji posiada więc wszystkie właściwości klasy `Student`, a zatem i pole `nrIndeksu`.

Operator `dynamic_cast` kontroluje możliwość wykonania konwersji w trakcie działania programu, opierając się na informacjach o typach obiektów dołączonych do skompilowanego programu. Mechanizm ten został wprowadzony do języka C++ w pierwszej połowie lat dziewięćdziesiątych ubiegłego wieku, znany jest on jako Run-Time Type Information, w skrócie RTTI. Mimo iż ten mechanizm nie jest nowy, ciągle jego realizacja może się różnić w zależności od kompilatora. W niektórych kompilatorach mechanizm ten jest domyślnie wyłączony (np. Visual C++), aby skorzystać z jego możliwości należy ustawić odpowiednią opcję kompilacji. Dokładne omówienie RTTI przekracza ramy niniejszego opracowania — zapamiętajmy, że z punktu widzenia sprawnego stosowania metod wirtualnych wystarczy rozważne stosowanie operatora `dynamic_cast`.

Własna czy wbudowana kontrola typów?

Kontrolowanie rzeczywistego typu obiektu polimorficznego może być realizowane poprzez wbudowanie przez programistę dodatkowych informacji do każdej z klas w hierarchii — tak jak w przypadku omówionej wcześniej funkcji `rodzaj`. Rozwiązane alternatywne wykorzystuje RTTI i operator `dynamic_cast`. Które rozwiązanie wybrać? Odpowiedź nie jest jednoznaczna. Mechanizm RTTI jest wygodny wtedy, gdy pracujemy z klasami których nie implementowaliśmy sami. Gdy nie zostały wyposażone w środki identyfikacji konkretnego wcielenia obiektu polimorficznego, RTTI pozwala na taką identyfikację. Przemyślane stosowanie operatora dynamicznej konwersji jest bezpieczne i skuteczne. Rozwiązanie to wymaga jednak działania dynamicznej identyfikacji typów, a to może trwać w przypadku rozbudowanych hierarchii klas.

W przypadku własnej realizacji hierarchii klas warto skorzystać z rozwiązania pierwszego — wprowadzenie własnej identyfikacji, np. opartej na funkcjach wirtualnych, nie jest czasochłonne, nie powiększa znacząco rozmiaru generowanego kodu oraz nie niesie za sobą istotnych narzutów czasowych. Uniezależnienie kodu od RTTI poprawia jego przenośność i pozwala na zastosowanie kompilatorów nie posiadających obsługi RTTI. Dodatkowo własna identyfikacja nie wyklucza wykorzystania RTTI, zwiększając elastyczność tworzonych klas.

Klasy abstrakcyjne

Dziedziczenie oraz polimorfizm to esencjonalne cechy programowania obiektowego. Pozwalają one między innymi na wygodne zarządzanie niejednorodnymi kolekcjami obiektów — co przedstawiono w poprzednim podrozdziale. Zauważmy, że jak istotną rolę pełni klasa `Osoba`. Stanowi ona swoisty szablon, określający metodę komunikacji z obiektami tej klasy, jak i klas pochodnych. Taka

klasa stanowi *interfejs*, określający metodę komunikacji z obiektami tworzącymi hierarchię klas.

Taką klasę bazową można w języku C++ zbudować w sposób specyficzny — może ona być *klasą abstrakcyjną*. Pojęcie wprowadzimy, posługując się następującym przykładem.

Załóżmy, że naszym zadaniem jest napisanie oprogramowania monitorującego stan czujników systemu alarmowego, chroniącego pewien budynek. Czujniki mogą być trojakiego rodzaju:
— czujniki otwarcia okien,
— czujniki otwarcia drzwi,
— czujniki ruchu.

Czujników systemu alarmowego jest zwykle wiele, ich liczba i rodzaj zależy od specyfiki budynku (lb. okien, drzwi itp.). Każdy z czujników jest inaczej zbudowany, w naszym programie dla reprezentowania czujnika każdego rodzaju przewidziana będzie osobna klasa. Jednak niezależnie od różnic, z punktu widzenia systemu alarmowego, wszystkie czujniki posiadają podobne cechy:
— w momencie aktywacji alarmu należy je włączyć, będzie to realizowała funkcja składowa `wlacz` wbudowana w każdą klasę opisu czujnika;
— w momencie dezaktywacji alarmu należy je wyłączyć, będzie to realizowała funkcja składowa `wylacz` wbudowana w każdą klasę opisu czujnika;
— identyfikacja czujnika będzie realizowana poprzez funkcję `rodzaj`, której rezultatem będzie wartość całkowita identyfikująca rodzaj czujnika;
— do monitorowania stanu czujnika służyć będzie funkcja składowa `stanAlarmowy`, której rezultatem jest wartość `true` gdy czujnik zadziałał (wykryto potencjalne włamanie), zaś wartość `false` w przypadku przeciwnym.

Te wspólne właściwości wszystkich czujników staną się podstawą dla zdefiniowania klasy `Czujnik`. Klasa ta będzie reprezentowała pewien abstrakcyjny czujnik. Na tym etapie nie jest istotna zasada działania, ważne są jednak operacje jakie będzie można na takim czujniku wykonać. Klasa opisu abstrakcyjnego czujnika może posiadać następującą postać:

```
class Czujnik
{
  public:
    Czujnik() {}

    virtual void wlacz() {};
    virtual void wylacz() {};
    virtual int  rodzaj() { return 0 };
    virtual bool stanAlarmowy() { return false };
};
```

Zwróćmy uwagę, że w klasie tej nie zdefiniowano żadnego pola. Posiada ona konstruktor, oraz cztery opisane wcześniej funkcje składowe. Funkcje te są oczywiście funkcjami wirtualnymi. Ponieważ klasa `Czujnik` reprezentuje czujnik abstrakcyjny, trudno wskazać, jakie powinny być ciała owych funkcji. Właściwa implementacja funkcji powinna zostać umieszczona w klasach pochodnych. Dokładniej — każda klasa pochodna *powinna* dokonać redefinicji funkcji składowej, wprowadzając jej implementację adekwatną do określonego typu czujnika.

Klasę `Czujnik` można zdefiniować w specyficzny sposób — jako *klasę abstrakcyjną*. Pozwoli to na uzyskanie dwóch efektów — podkreślenie abstrakcyjnego charakteru klasy `Czujnik` oraz wymuszenie konieczności redefinicji funkcji składowych w klasach pochodnych. W języku C++ buduje się klasę abstrakcyjną poprzez umieszczenie w jej definicji przynajmniej jednej *funkcji abstrakcyjnej*. W oryginale funkcja abstrakcyjna nosi nazwę *pure virtual function*, jednak dosłowne tłumaczenie[23] nie wydaje się szczęśliwe, proponuje się zatem nazwę stosowanie nazwy *funkcja abstrakcyjna*. Funkcja abstrakcyjna nie posiada ciała, w jego miejscu występuje symbol „`= 0`”:

```
class Czujnik
{
  public:
    Czujnik() {}

    virtual void wlacz() = 0;
    virtual void wylacz() = 0;
    virtual int  rodzaj() = 0;
    virtual bool stanAlarmowy() = 0;
};
```

Tak zdefiniowana klasa jest abstrakcyjna, posiada cztery funkcje abstrakcyjne, których właściwa definicja musi zostać umieszczona w klasach pochodnych. Nie wszystkie funkcje muszą być abstrakcyjne — wystarczy jedna, by klasa stała się abstrakcyjną.

Rola klasy abstrakcyjnej jest specyficzna — nie deklaruje się jawnie obiektów tej klasy, służy ona jako klasa bazowa, stając się korzeniem, często bardzo rozbudowanej hierarchii klas. W klasach pochodnych należy umieścić konkretne definicje funkcji abstrakcyjnych. W rozważanym przykładzie, klasa `Czujnik` będzie stanowiła klasę bazową dla klas opisujących czujnik otwarcia okien, drzwi i czujnik ruchu. W klasach tych należy umieścić konkretne realizacje funkcji abstrakcyjnych, adekwatne do roli przypisanej danej klasie. W rozważanym przykładzie rozróżniamy trzy czujniki systemu alarmowego, każda z nich identyfikowana będzie stałą wyliczeniową następującego typu:

```
enum RodzajeCzujnikow
{
  rcCzujnikRuchu,
  rcCzujnikOkna,
  rcCzujnikDrzwi
};
```

Klasa opisu czujnika ruchu może mieć poglądową postać umieszczoną niżej. Oczywiście, dla klarowności prezentowanego przykładu, nie jest możliwe przytoczenie w pełni działającego kodu — dlatego np. ciało funkcji `stanAlarmowy` zawiera jedynie słowny komentarz zadania, jakie ta funkcja ma wykonać.

[23] Terminy *czysta funkcja wirtualna* czy *w pełni wirtualna funkcja* nie oddają zdaniem autora istoty sprawy i są niewygodne w stosowaniu.

```
class CzujnikRuchu : public Czujnik
{
  public:
    CzujnikRuchu() : Czujnik() {}

    void wlacz() { cout << "\nCzujnk ruchu wlaczony"; };
    void wylacz() { cout << "\nCzujnk ruchu wylaczony"; };
    int  rodzaj() { return rcCzujnikRuchu; };
    bool stanAlarmowy()
    {
      Tu kod testowania stanu czujnika, rezultat true gdy wykryto
      potencjalne włamanie, false w przeciwnym przypadku.
    }
};
```

W podobny sposób zdefiniowano klasy opisu czujnika otwarcia okien i drzwi:

```
class CzujnikOkna : public Czujnik
{
  public:
    CzujnikOkna() : Czujnik() {}

    void wlacz() { cout << "\nCzujnk okna wlaczony"; };
    void wylacz() { cout << "\nCzujnk okna wylaczony"; };
    int  rodzaj() { return rcCzujnikOkna; };
    bool stanAlarmowy()
    {
      Tu kod testowania stanu czujnika, rezultat true gdy wykryto
      potencjalne włamanie, false w przeciwnym przypadku.
    }
};

class CzujnikDrzwi : public Czujnik
{
  public:
    CzujnikDrzwi() : Czujnik() {}

    void wlacz() { cout << "\nCzujnk drzwi wlaczony"; };
    void wylacz() { cout << "\nCzujnk drzwi wylaczony"; };
    int  rodzaj() { return rcCzujnikDrzwi; };
    bool stanAlarmowy()
    {
      Tu kod testowania stanu czujnika, rezultat true gdy wykryto
      potencjalne włamanie, false w przeciwnym przypadku.
    }
};
```

Załóżmy, że nasze czujniki mają być zainstalowane w budynku z jednymi drzwiami, trzema oknami, wewnątrz mają być zainstalowane trzy czujniki ruchu.

Każdy z fizycznych czujników będzie monitorowany przez obiekt odpowiedniej, wyżej zdefiniowanej klasy. Potrzebujemy łącznie siedem czujników, do zarządzania nimi wykorzystamy tablicę wskaźników na obiekty klasy `Czyjnik` (koncepcja jest analogiczna do tej, przedstawionej przy okazji opisu klasy `EwidencjaOsob`):

```
const int maksLbCzujnikow = 20;
Czujnik * tabCzujnikow[ maksLbCzujnikow ];
int lbCzujnikow = 7;
```

Przedstawiony niżej fragment kodu ilustruje utworzenie siedmiu obiektów odpowiednich klas, wskaźniki do utworzonych obiektów zapisane zostaną w tablicy:

```
tabCzujnikow[ 0 ] = new CzujnikRuchu();
tabCzujnikow[ 1 ] = new CzujnikRuchu();
tabCzujnikow[ 2 ] = new CzujnikRuchu();
tabCzujnikow[ 3 ] = new CzujnikDrzwi();
tabCzujnikow[ 4 ] = new CzujnikOkna();
tabCzujnikow[ 5 ] = new CzujnikOkna();
tabCzujnikow[ 6 ] = new CzujnikOkna();
```

Operacja włączenia alarmu — aktywacji wszystkich czujników — może być zrealizowana następująco:

```
for( int i = 0; i < lbCzujnikow ; i++ )
  tabCzujnikow[ i ]->wlacz();
```

Komplementarna zaś operacja wyłączenia alarmu może mieć następującą postać:

```
for( int i = 0; i < lbCzujnikow ; i++ )
  tabCzujnikow[ i ]->wylacz();
```

Usunięcie obiektów reprezentujących czujniki może być zrealizowane następująco:

```
while( --lbCzujnikow >= 0 )
  delete tabCzujnikow[ lbCzujnikow ];
```

Zwróćmy uwagę, że w powyższych fragmentach kodu wykorzystujemy funkcje zdefiniowane w abstrakcyjnej klasie `Czujnik`. Odwołania wykonywane są za pośrednictwem wskaźników do dynamicznie utworzonych obiektów. Są to funkcje wirtualne, mamy zatem pewność, że wywołane zostaną wersje właściwe dla konkretnego obiektu. Dzięki temu, że są to funkcje abstrakcyjne, mamy również pewność, że rzeczywiście zdefiniowano je w klasach pochodnych.

Operację monitorowania stanu czujników można zrealizować następująco:

```
while( systemAktywny() )
{
  for( int i = 0; i < lbCzujnikow; i++ )
    if( tabCzujnikow[ i ]->stanAlarmowy() )
      zglosAlarm( tabCzujnikow[ i ] );
}
```

Iteracja `while` wykonuje się, dopóki nie zostanie wykryta dezaktywacja systemu alarmowego. Gdy system jest aktywny, rezultatem funkcji `systemAktywny` jest wartość `true`, wyłączenie systemu spowoduje, że rezultatem tej funkcji będzie wartość `false`. W trakcie każdego przebiegu iteracji `while` przeglądana jest tablica obiektów reprezentujących czujniki. Jeżeli któryś z obiektów wykrył zadziałanie czujnika, rezultatem jego funkcji składowej `stanAlarmowy` będzie wartość `true`. Spowoduje to wywołanie funkcji `zglosAlarm`. Funkcja ta otrzymuje jako parametr wskaźnik na obiekt, reprezentujący czujnik, który zgłosił sytuację alarmową. Funkcja ta na podstawie rezultatu wywołania funkcji `rodzaj` określa z jakim typem czujnika mamy do czynienia i wyprowadza do strumienia wyjściowego odpowiedni komunikat:

```
void zglosAlarm( Czujnik * wc )
{
  cout << "\nWykrytu zgloszenie z czujnika ";
  switch( wc->rodzaj() )
  {
    case rcCzujnikRuchu : cout << "ruchu!" << endl;
                          break;
    case rcCzujnikOkna  : cout << "otwarcia okna!" << endl;
                          break;
    case rcCzujnikDrzwi : cout << "otwarcia drzwi!" << endl;
                          break;
  }
}
```

Zauważmy, że odwołując się do czujników cały czas bazujemy na funkcjach zdefiniowanych w klasie abstrakcyjnej `Czujnik`. Określają one sposób wymiany informacji z wszystkimi specjalizowanymi czujnikami, reprezentowanymi przez obiekty klas pochodnych. Klasa `Czujnik` stanowi zatem *interfejs*, definiujący sposób obsługi i wymiany informacji z czujnikami.

Zaprezentowane wcześniej rozwiązanie zarządzania wieloma czujnikami jest skuteczne lecz nieeleganckie. Do obsługi czujników możemy zaprojektować specjalną klasę — niech nazywa się ona `SystemAlarmowy`. Zdefiniujmy obiekt `system`, tej właśnie klasy.

```
SystemAlarmowy system;
```

Dopisanie do systemu obiektów reprezentujących czujniki realizowane jest za pośrednictwem funkcji `dodajCzujnik`:

```
system.dodajCzujnik( new CzujnikRuchu() );
system.dodajCzujnik( new CzujnikRuchu() );
system.dodajCzujnik( new CzujnikRuchu() );
system.dodajCzujnik( new CzujnikDrzwi() );
system.dodajCzujnik( new CzujnikOkna() );
system.dodajCzujnik( new CzujnikOkna() );
system.dodajCzujnik( new CzujnikOkna() );
```

Włączenie czujników, przejście w stan monitorowania oraz wyłączenie czujników po dezaktywacji systemu może mieć następującą postać:

```
system.wlaczCzujniki();
system.monitoruj();
system.wylaczCzujniki();
```

Usunięcie utworzonych obiektów reprezentujących czujniki realizuje wywołanie:

```
system.usunCzujniki();
```

Klasa `SystemAlarmowy` przechowuje wskaźniki do obiektów reprezentujących czujniki w tablicy wskaźników do klasy `Czujnik`. Wszystkie operacje na zapisanych tam czujnikach są wykonywane za pośrednictwem interfejsu zdefiniowanego poprzez funkcje abstrakcyjne tejże klasy. Definicja klasy ma następującą postać:

```
class SystemAlarmowy
{
  public:
    SystemAlarmowy();

    void dodajCzujnik( Czujnik * wc );
    Czujnik * podajCzujnik( int nr );

    void wlaczCzujniki();
    void monitoruj();
    void wylaczCzujniki();

    void zglosAlarm( Czujnik * wc );
    bool systemAktywny();

    void usunCzujniki();
    void wyczysc();

  protected:
    static const int maksLbCzujnikow = 20;
    Czujnik * tabCzujnikow[ maksLbCzujnikow ];
    int lbCzujnikow;
};
```

Definicja konstruktora, funkcji dodawania wskaźnika oraz pobierania informacji o nim, mają następująca postać:

```
SystemAlarmowy::SystemAlarmowy() : lbCzujnikow( 0 )
{
}

void SystemAlarmowy::dodajCzujnik( Czujnik * wc )
{
  if( lbCzujnikow < maksLbCzujnikow - 1 )
    tabCzujnikow[ lbCzujnikow++ ] = wc;
}
```

```
Czujnik * SystemAlarmowy::podajCzujnik( int nrCzujnika )
{
  if( nrCzujnika >= 0 && nrCzujnika < lbCzujnikow )
    return tabCzujnikow[ nrCzujnika ];
  else
    return 0;
 }
```

Operację włączenia i wyłączenia czujników wykonywane są w sposób opisany wcześniej, wykonują je następujące funkcje:

```
void SystemAlarmowy::wlaczCzujniki()
{
  for( int i = 0; i < lbCzujnikow ; i++ )
    tabCzujnikow[ i ]->wlacz();
}

void SystemAlarmowy::wylaczCzujniki()
{
  for( int i = 0; i < lbCzujnikow; i++ )
    tabCzujnikow[ i ]->wylacz();
}
```

Analogicznie jest również zrealizowana funkcja monitorowania stanu czujników:

```
void SystemAlarmowy::monitoruj()
{
  while( systemAktywny() )
  {
    for( int i = 0; i < lbCzujnikow; i++ )
      if( tabCzujnikow[ i ]->stanAlarmowy() )
        zglosAlarm( tabCzujnikow[ i ] );
  }
}
```

Funkcje `systemAktywny` oraz `zglosAlarm` realizują operacje opisane wcześniej.

Klasa `SystemAlarmowy` jest przygotowana na obsługę dowolnych urządzeń, dziedziczących interfejs określony klasą `Czujnik`. Załóżmy, że chcemy rozbudować system alarmowy o czujniki pożarowe, wykrywające dym. Nic nie stoi na przeszkodzie zdefiniowania kolejnej klasy pochodnej — np. `CzujnikDymu` — która po implementacji funkcji abstrakcyjnych może być wykorzystana dla tworzenia kolejnych obiektów, dodawanych do listy monitorowanych czujników.

1.3. Podsumowanie

Język C++ ciągle rozwija się i ewoluuje. Przedstawione w poprzednich rozdziałach mechanizmy programowania obiektowego dostępne są już od wczesnych

wersji tego języka. Mechanizmy te stanowią podstawę obiektowego podejścia do programowania w języku C++. Większość tych mechanizmów została przeniesiona do innych języków programowania — np. do języka Java, C# czy PHP. Dobre zrozumienie opisanych wcześniej rozwiązań obiektowych ułatwi naukę kolejnych języków programowania obiektowego, oraz wychwycenie — czasem bardzo subtelnych — różnic w pojmowaniu i realizacji obiektowości w każdym z języków.

Przedstawione wcześniej rozwiązania dotyczą podstawowych elementów obiektowości w języku C++. Dalekie są od wyczerpania tematu — język C++ zawiera wiele istotnych mechanizmów, które stanowią podstawę rozwiązań obiektowych lub pełną garścią z nich korzystają. Niestety, ograniczone ramy niniejszego opracowania nie pozwalają na szczegółowe ich omówienie. Podstawowe znaczenie mają szczególnie dwa mechanizmy — przeciążanie operatorów (ang. *operator overloading*) oraz stosowanie szablonów (ang. *templates*). Stanowią one interesującą podstawę dla konstrukcji klas kontenerowych, oferujących najróżniejsze możliwości organizowania struktur danych. Aktualnie standardem stało się wykorzystanie biblioteki STL (ang. *Standard Template Library*), która oferuje szereg możliwości przechowywania oraz manimulowania danymi. Zainteresowanym czytelnikom polecamy pozycje [5, 7, 3, 4, 6], w wyczerpujący sposób prezentujące zagadnienia, które nie zmieściły się w niniejszym opracowaniu.

Należy zwrócić uwagę, że koncepcja obiektowości obecna w języku C++ nie jest jedyną i wzorcową. Istnieje wiele różniących się w szczegółach realizacji koncepcji obiektowości — np. w języku Smalltalk, Objective-C czy Ruby. Jednak podejście oferowane przez język C++ jest sprawdzone praktycznie i potwierdzone praktyką realizacji ogromnej liczby projektów programistycznych — o różnej skali i charakterze. Koncepcja ta wspierana jest przez producentów kompilatorów i środowisk programistycznych, również tych, które umożliwiają szybkie prototypowanie programów — środowisk RAD (ang. *Rapid Application Development*). Środowiska takie, mające wcześniej charakter rozwiązań komercyjnych (np. Microsoft Visual C++, Borland C++ Builder) zaczynają być dostępne w wersjach dostarczanych darmowo, bazujących na rozwiązaniach typu oprogramowania otwartego (np. QtCreator firmowany aktualnie przez firmę Nokia).

Inwestycja w nabycie umiejętności sprawnego programowania obiektowego w języku C++ wydaje się inwestycją trafioną. Jednak należy zwrócić uwagę na to, że język C++ jest rzeczywiście językiem skomplikowanym, a aktualne tendencje w programowaniu w C++ oraz aktualny standard języka zdają się to potwierdzać. Codziennością staje się wykorzystanie szablonów, klas z biblioteki STL — ten kierunek wydaje się znacznie upraszczać kod i nakład pracy programisty, jednak wymaga od niego drobiazgowej znajomości nie tylko samego języka, ale również właśnie biblioteki STL. A ta — mimo pozornej prostoty — jest w istocie skomplikowana, sprawne i bezpieczne jej stosowanie wymaga uwagi sporej wiedzy.

Istnieje drugi czynnik sprawiający, że droga od podstaw C++ do sprawnego programowania aplikacji komercyjnych w tym języku jest długa i żmudna. Sam język jak i jego standardowe biblioteki dalej nie wspierają operacji programowania GUI, dostępu do baz danych, programowania sieciowego. Operacje te realizują specjalizowane biblioteki, dostarczane zwykle z określonym środowiskiem programistycznym (np. VCL i C++ Builder). Rozwiązania te są zwykle

nieprzenośne na poziomie kodu źródłowego. Całe szczęście, że powstają niekomercyjne rozwiązania typu biblioteka WxWidgets czy Qt. Biblioteki te oferują pełny zakres narzędzi (najczęściej w postaci bibliotek klas) brakujących językowi C++ a koniecznych do tworzenia w pełni wartościowego oprogramowania. Rozwiązania te wymagają jednak bardzo dobrej znajomości języka C++ oraz — nie oszukujmy się — wyraźnego nakładu pracy na poznanie samych bibliotek i metod ich wykorzystania.

Wskazane powyżej czynniki sprawiają, iż coraz popularniejsze stają się języki pochodne od C++, takie jak Java i C#, w których zrezygnowano z wielu złożonych konstrukcji i które rdzennie zapewniają wsparcie dla programowania GUI czy dostęp do baz danych. Uzupełnione sprawnymi środowiskami RAD, języki te stają się interesująca alternatywą dla C++[24]. Jednak — zdaniem autora — nie należy się obawiać o przyszłość tego języka i spadek jego popularności. Oferowana przez C++ elastyczność, szybkość wsparta nowoczesnymi bibliotekami typu Qt sprawiają, że język ten jeszcze na długi czas pozostanie podstawowym narzędziem programowania dla profesjonalistów.

1.4. Ćwiczenia i zadania

Rozdział ten zawiera zadania i ćwiczenia do samodzielnego wykonania. Koncentrują się one na omówionych wcześniej zagadnieniach programowania obiektowego — zostały podzielone na tematycznie spójne sekcje.

1.4.1. Definiowanie klas

Zad. 1. Zdefiniować klasę `Data` zawierającą pola prywatne: `int dzien`, `int miesiac`, `int rok`. Klasa powinna zawierać funkcje dostępowe, pobierające zawartość poszczególnych pól (np. `podajDzien` itd.), oraz funkcje pozwalające na ustalanie zawartości pól (np. `ustawDzien`, itd.). Klasa ma posiadać konstruktor domyślny, zerujący wszystkie pola, oraz konstruktor ogólny, otrzymujący trzy parametry, trafiające odpowiednio do pól opisujących dzień, miesiąc, rok.

Zad. 2. Zdefiniować klasę `Czas` zawierającą pola prywatne: `int godzina`, `int minuta`, `int sekunda`. Klasa powinna zawierać funkcje dostępowe, pobierające zawartość poszczególnych pól (np. `podajGodzine` itd.), oraz funkcje pozwalające na ustalanie zawartości pól (np. `ustawGodzine`, itd.). Klasa ma posiadać konstruktor domyślny zerujący wszystkie pola, oraz konstruktor ogólny, otrzymujący trzy parametry, trafiające odpowiednio do pól opisujących godzinę, minutę, sekundę.

Zad. 3. Dane jest równanie kwadratowe:

$$Ax^2 + Bx + C = 0$$

Należy zaprojektować klasę `RownanieKwadratowe`, przechowującą współczynniki takiego równania i pozwalającą na obliczanie wyznacznika (delty) oraz

[24] Rozważamy tutaj aspekt tworzenia oprogramowania klasy desktop — liczba rozwiązań alternatywnych w segmencie narzędzi do tworzenia aplikacji internetowych jest wyraźnie większa.

rozwiązań takiego równania. Klasa powinna posiadać prywatne pola A, B, C, będące liczbami całkowitymi, reprezentujące współczynniki równania. Każde z pól powinno posiadać akcesory w postaci funkcji składowych, np. `ustawA()`, `pobierzA()`. Klasa powinna posiadać funkcje składową `delta()` obliczającą wyróżnik trójmianu oraz trzy funkcje obliczające miejsca zerowe: jedną do obliczania pierwiastka podwójnego (dla delta==0) oraz dwie osobne, do obliczania pierwiastków, gdy delta ma wartość dodatnią. Obiekt ten może być wykorzystany w następujący sposób:

```
RownanieKwadratowe r;
double num;

cout << 'Podaj A:'; cin >> num; r.ustawA( num );
cout << 'Podaj B:'; cin >> num; r.ustawB( num );
cout << 'Podaj C:'; cin >> num; r.ustawC( num );
if( r.delta() > 0 )
  cout << "Pierwiastki rownania x1=" << r.obliczX1() << " x2="
  << r.obliczX2() << endl;
else
  if(r.delta() = 0 )
    cout << "Pierwiastek podwójny x12=" << r.obliczX12() << endl;
  else
    cout << "Brak pierwiastkow rzeczywistych" << endl;
```

Klasa powinna być wyposażona w konstruktor domyślny oraz trzyparametrowy konstruktor ogólny, co pozwoli deklarować obiekty klasy `RownanieKwadratowe` w następujący sposób:

```
RownanieKwadratowe r1;        // Konstruktor inicjuje A=B=C=0;
RownanieKwadratowe r2(3,2,1); // Konstruktor inicjuje A=3,B=2,C=1;
```

Należy się zastanowić nad przypadkiem, gdy równanie staje się liniowym, tzn. $A = 0$ i zaproponować rozwiązanie tego problemu.

Zad. 4. Dany jest układ równań liniowych:

$$A_1x + B_1y = C_1$$
$$A_2x + B_2y = C_2$$

Rozwiązanie układu równań może polegać na wyliczeniu odpowiednich wyznaczników W, W_x, W_y a następnie ich iloczynów — zgodnie z informacjami poznanymi na zajęciach z matematyki. Należy zaprojektować i zaimplementować w języku C++ obiektowy program pozwalający na rozwiązywanie dowolnego układu takich równań. Program powinien umożliwiać wczytanie współczynników A_1, B_1, C_1, A_2, B_2, C_2, następnie powinien wyznaczyć rozwiązania równań metodą wyznacznikową. Należy identyfikować i prawidłowo zareagować na sytuację, gdy układ jest nieokreślony.

Zad. 5. Zdefiniować klasę `Punkt`, przeznaczoną do reprezentowania punków w przestrzeni dwuwymiarowej. Klasa powinna posiadać dwa pola prywatne x oraz y przeznaczone do przechowywania współrzędnych punktu, konstruktor domyślny, zerujący wartości tych pól, konstruktor ogólny umożliwiający ustawienie wartości pól (np. w deklaracji `Punkt p(10, 20);`). Ustawianie i pobieranie

wartości pól powinny realizować funkcje dostępowe `ustawX`, `ustawY`, `pobierzX`, `pobierzY`.

Zad. 6. Zdefiniować klasy reprezentujące płaskie figury geometryczne `Kwadrat`, `Prostokat`, `Kolo`, `Trapez`. Klasy powinny posiadać pola przechowujące informacje niezbędne do obliczenia pól tych figur. Pola powinny być prywatne, dostęp do nich powinien być zapewniany przez odpowiednie metody pobierania i ustawiania wartości. Klasy powinny posiadać konstruktor bezparametrowy oraz ogólny, zapewniające inicjalizację wszystkich pól. Obliczenie pola danej figury powinna realizować odpowiednia funkcja składowa o nazwie `obliczPole` — wyznaczona wartość pola powinna być jej rezultatem.

1.4.2. Dziedziczenie

Zad. 7. Bazując na klasie `Punkt` (zdefiniowanej w trakcie rozwiązywania zadania nr 5) oraz wykorzystując mechanizm dziedziczenia, zdefiniować klasę `Punkt3D`, reprezentującą punkt w przestrzeni trójwymiarowej. Klasa ta powinna posiadać dodatkowe pole z przechowujące współrzędną trzeciego wymiaru, odpowiednie funkcje dostępowe `ustawZ`, `pobierzZ`, własny konstruktor bezparametrowy i parametrowy.

Zad. 8. Bazując na klasie `Data` (zdefiniowanej w trakcie rozwiązywania zadania nr 1) zdefiniować z wykorzystaniem dziedziczenia klasę `DataZKontrola` — w tej klasie należy dokonać redefinicji funkcji ustawiających wartości pól, tak by kontrolowały one poprawność wartości przekazywanych parametrów (roku, miesiąca, dnia).

Zad. 9. Bazując na klasie `Czas` (zdefiniowanej w trakcie rozwiązywania zadania nr 2) zdefiniować z wykorzystaniem dziedziczenia klasę `CzasZKontrola` — w tej klasie należy dokonać redefinicji funkcji ustawiających wartości pól, tak by kontrolowały one poprawność wartości przekazywanych parametrów (godziny, minuty, sekundy).

Zad. 10. Bazując na klasach opisu figur płaskich zdefiniowanych w zadaniu nr 6, zdefiniować ich klasy pochodne, reprezentujące bryły: sześcian (pochodna klasy reprezentującej kwadrat), prostopadłościan (pochodna klasy reprezentującej prostokąt), kula (pochodna klasy reprezentującej koło), graniastosłup o podstawie trapezu (pochodna klasy reprezentującej trapez). W klasach pochodnych należy dodać wszelkie informacje konieczne dla obliczenia pól tych brył, oraz należy przedefiniować funkcje składowe obliczania pola (funkcje `obliczPole` kazdej z klas reprezentujących figurę płaską), tak by wyznaczały właściwe pola brył.

1.4.3. Dziedziczenie i polimorfizm

Zad. 11. Zdefiniować klasę `Figura`, reprezentującą abstrakcyjną figurę geometryczną. Klasa powinna posiadać funkcję składową o nazwie `obliczPole`, której rezultatem ma być wartość 0, oraz funkcję `wypiszNazwe`, która wypisze do strumienia wyjściowego programu napis `Figura`. Następnie należy zmodyfikować definicje klas z zadania nr 6, tak by klasy te były klasami pochodnymi klasy `Figura`. W klasach pochodnych należy dokonać redefinicji funkcji `wypiszNazwe`, tak by wyprowadzała ona do strumienia wyjściowego nazwę każdej z klas. Zakła-

da się, że każda z klas dokonuje również redefinicji funkcji `obliczPole`, zgodnie z treścią zadania nr 6. Dziedziczenie oraz redefinicje funkcji składowych powinny być tak zrealizowane, by poprawnie działały następujące wywołania:

```
Kwadrat   kw( 2 );
Kolo      ko( 1 );
Prostokat pr( 3, 2 )
Figura *  f;

f = &kw;
f->wypiszNazwe();
cout << ' ' << f->obliczPole() << endl;  // Kwadrat 4

f = &ko;
f->wypiszNazwe();
cout << ' ' << f->obliczPole() << endl;  // Kolo 6.28318531

f = &pr;
f->wypiszNazwe();
cout << ' ' << f->obliczPole() << endl;  // Prostokat 6
```

Zad. 12. Zdefiniuj klasę `KolekcjaFigur`, pozwalającą na przechowywanie informacji o figurach zdefiniowanych w zadaniu nr 11. Klasa powinna posiadać prywatną tablicę wskaźników, w której będą zapisywane wskaźniki na figury dopisywane do kolekcji. Można założyć, że tablica będzie miała dostatecznie duży, statycznie określony rozmiar. Dopisanie figury do kolekcji powinna realizować funkcja `dodajFigure`, którą można wywołać następująco:

```
KolekcjaFigur kf;

kf.dodajfigurę( new Kwadrat );
kf.dodajfigurę( new Kolo );
kf.dodajfigurę( new Prostokat );
```

Klasa powinna posiadać funkcję `obliczSumarycznePole`, która powinna wyznaczać sumę pól wszystkich figur zapisanych w kolekcji. Operacja ta powinna być wykonana poprzez wywołanie funkcji składowych `obliczPole` każdej figur z kolekcji, oraz zsumowanie ich rezultatów. Klasa powinna posiadać funkcję `wyczysc`, która sprawia, że kolekcja staję się pusta, jednak bez zwalniania jakiejkolwiek pamięci przydzielonej obiektom reprezentującym figury, oraz funkcję `kasuj`, która sprawia, że kolekcja staję się pusta, oraz zwalnia pamięć przydzieloną obiektom reprezentującym figury.

1.4.4. Propozycje projektów

Projekt 1. Na bazie przykładu przedstawionego w rozdz. *Klasy abstrakcyjne* na str. 71, zbudować program symulujący działanie monitoringu systemu alarmowego. Program powinien umożliwiać definiowanie struktury systemu alarmowego, poprzez określenie liczby i typów czujników. Klasy opisu czujników należy tak zmodyfikować, aby losowo generowały stan alarmowy, symulując co

jakiś czas zadziałanie hipotetycznego czujnika. Użytkownik programu powinien mieć możliwość aktywowania alarmu, oraz jego dezaktywacji. W stanie monitorowania, system powinien wyświetlać na ekranie informację o stanie każdego z czujników, zaznaczając wyraźnie zdarzenie otrzymania sygnału alarmowego z danego czujnika.

Projekt 2. Zaprojektować i zbudować program symulujący działanie automatu vendingowego. Program powinien zawierać klasę `Automat`, która powinna zarządzać zestawem obiektów, reprezentujących sprzedawany towar. Klasa ta powinna wykonywać wszystkie operacje na abstrakcyjnej klasie `Towar`, która ma być bazą dla specjalizowanych klas reprezentujących różnorodne towary sprzedawane przez automat. Każdy towar ma być opisywany własną klasą, np. `Kawa`, `Herbata`, `Batonik`, `Czekolada`. Klasa `Automat` powinna umożliwiać dodawanie dowolnego towaru do automatu, z określeniem jego ilości oraz ceny jednostkowej. Użytkownik programu powinien mieć możliwość zakupu dowolnego towaru, klasa `Automat` powinna rejestrować sprzedaż — zmniejszyć liczbę danego towaru i zwiększyć kwotę zarobionych przez automat pieniędzy. W przypadku wyczerpania zapasów któregoś z towaru, program ma informować o jego niedostępności.

Projekt 3. Zaprojektować i zrealizować w technologii obiektowej prostą grę komputerową, w ramach której użytkownik steruje przy użyciu klawiatury ruchomym obiektem, poruszającym się w labiryncie (gra klasy *Pacman*). Obiekt powinien zbierać rozmieszczone w labiryncie obiekty-nagrody, unikając jednocześnie obiektów-pułapek (np. min). Gra kończy się po zebraniu wszystkich nagród lub po utraceniu przez obiekt ruchomy „życia" w wyniku wejścia w pułapki. Grę można rozbudować o ruchome obiekty-pułapki poruszające się w zadanych z góry miejscach planszy lub realizujące aktywny pościg za sterowanym przez użytkownika obiektem ruchomym. Można również zaprojektować wiele plansz, umożliwiając obiektowi ruchomemu przemieszczanie się pomiędzy nimi. Dla gry należy zaproponować prostą fabułę.

Bibliografia

[1] Informacje dostępne online. Bjarne Stroustrup's homepage. *http://www.research.att.com/∼bs/*, 2009. [4, 58]

[2] Informacje dostępne online. The Development of the C Language by Dennis Ritchie. *http://cm.bell-labs.com/cm/cs/who/dmr/chist.html*, 2009. [3]

[3] Lajoie J. Lippman S.B. *Podstawy języka C++.* Wydawnictwa Naukowo–Techniczne, Warszawa, Polska, 2003. [39, 48, 52, 79]

[4] Josuttis N.M. *C++. Biblioteka standardowa. Podręcznik programisty.* Helion, Gliwice, Polska, 2003. [79]

[5] Plauger P.J. *Biblioteka standardowa C++.* Wydawnictwa Naukowo–Techniczne, Warszawa, Polska, 1997. [9, 19, 79]

[6] Meyers S. *STL w praktyce. 50 sposobów efektywnego wykorzystania.* Helion, Gliwice, Polska, 2004. [79]

[7] Prata S. *Szkoła programowania. Język C++.* Helion, Gliwice, Polska, 2006. [39, 48, 52, 58, 79]

Spis rysunków