Język C++. Szkoła programowania. Wydanie V

Autor: Stephen Prata
Tłumaczenie: Przemysław Steć (rozdz. 17, dod. A–G), Przemysław Szeremiota (rozdz. 9–12), Tomasz Walczak (rozdz. 13–16), Tomasz Żmijewski (rozdz. wstęp, 1–8)
ISBN: 83-7361-958-5
Tytuł oryginału: C++ Primer Plus, 5th Edition
Format: B5, stron: 1168

Doskonały podręcznik dla początkujących programistów

- Typy danych i konstrukcje sterujące
- Programowanie proceduralne i obiektowe
- Biblioteka STL
- Obsługa plików

C++ to jeden z najpopularniejszych języków programowania, jego możliwości są ogromne. Używa się go do pisania aplikacji narzędziowych, gier, a nawet systemów operacyjnych. Nauka języka C++ jest jednak żmudnym i skomplikowanym procesem – to prawdziwe wyzwanie dla programistów. Opanowanie potęgi C++ wymaga poznania zasad programowania obiektowego, korzystania z bibliotek i szablonów, obsługi błędów i wyjątków i wielu innych zagadnień. Ale odpowiednio napisany podręcznik, zawierający podstawową wiedzę na temat tego języka, zdecydowanie ułatwi przyswojenie sztuki programowania w C++.

Książka „Język C++. Szkoła programowania. Wydanie V" to właśnie taki podręcznik. Jego autor Stephen Prata przedstawia C++ w sposób idealnie nadający się dla początkujących programistów chcących opanować tajniki tego języka. Czytając tę książkę, poznasz historię języka C i jego najważniejsze elementy, dowiesz się, czym różni się programowanie proceduralne od obiektowego i jak stosować te dwie techniki, korzystając z C++. Nauczysz się definiować klasy i obiekty, przydzielać zasoby pamięci dla aplikacji, korzystać ze wskaźników i implementować mechanizmy obsługi plików i strumieni wejścia-wyjścia.

- Kompilatory języka C++
- Struktura programu
- Proste i złożone typy danych
- Pętle i instrukcje warunkowe
- Definiowanie funkcji i korzystanie z nich
- Model pamięci w C++
- Podstawowe elementy programowania obiektowego – obiekty i klasy
- Dynamiczne przydzielanie pamięci
- Mechanizmy dziedziczenia
- Obsługa wyjątków
- Biblioteka STL
- Operacje wejścia-wyjścia

Poznaj najważniejsze zasady programowania w C++

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

SPIS TREŚCI

Rozdział 2

Pierwszy program w C++

W rozdziale zajmiemy się następującymi zagadnieniami:

- Tworzenie programu w C++.
- Ogólna postać programu w C++.
- Dyrektywa `#include`.
- Funkcja `main()`.
- Użycie obiektu `cout` do wyprowadzania danych.
- Komentarze w programach C++.
- Użycie `endl`.
- Deklarowanie i użycie zmiennych.
- Użycie obiektu `cin` do wprowadzania danych.
- Definiowanie i użycie prostych funkcji.

Kiedy budujemy dom, zaczynamy od fundamentów i szkieletu. Jeśli nie mamy na początek solidnych podstaw, będziemy mieli później problemy z dalszymi etapami pracy: oknami, framugami, dachem i parkietami. Tak samo w przypadku nauki programowania musimy zacząć od poznania podstawowych elementów konstrukcyjnych programu, a dopiero potem możemy przejść dalej, na przykład do pętli i obiektów. W niniejszym rozdziale przedstawimy w skrócie strukturę programu w C++ oraz krótko omówimy wybrane zagadnienia — najważniejsze z nich to funkcje i klasy — którymi później zajmiemy się bardziej szczegółowo. Chodzi o to, aby przynajmniej niektóre pojęcia wprowadzać stopniowo, bez zaskakiwania.

C++ — początek

Zacznijmy od prostego programu w C++ pokazującego komunikat. Na listingu 2.1 ukazano użycie obiektu `cout` do prezentacji danych wyjściowych. Kod źródłowy zawiera kilka dodatkowych komentarzy, zaczynających się od pary znaków `//`, całkowicie ignorowanych przez kompilator. W języku C wielkość liter ma znaczenie, wobec czego, jeśli w pokazanym programie zamiast `cout` napiszemy `Cout` lub `COUT`, kompilator zgłosi błąd, informując o nieznanych identyfikatorach. Rozszerzenie pliku *cpp* to typowy sposób wskazania, że mamy do czynienia z programem

w C++. Czasami może być konieczne użycie innego rozszerzenia, jak opisaliśmy to w pierwszym rozdziale.

Listing 2.1. myfirst.cpp

```
// myfirst.cpp—wyświetla komunikat

#include <iostream>                          // dyrektywa PREPROCESORA
int main()                                   // nagłówek funkcji
{                                            // początek treści funkcji
     using namespace std;                    // uwidocznienie definicji
     cout << "Zabaw się językiem C++.";       // komunikat
     cout << endl;                           // zaczynamy nowy wiersz
     cout << "Nie pożałujesz!" << endl;       // kolejny komunikat
     return 0;                               // zakończenie działania funkcji main()
}                                            // koniec treści funkcji
```

Uwaga o zgodności ze standardem

W przypadku użycia starszego kompilatora zamiast `#include <iostream>` należy użyć dyrektywy `#include <iostream.h>` oraz pominąć wiersz **`using namespace`** `std;`. Zatem zamiast:

```
#include <iostream>                  // pieśń przyszłości
```

piszemy:

```
#include <iostream.h>                // stan na dzisiaj
```

oraz całkowicie pomijamy wiersz:

```
using namespace std;                 // to też na przyszłość
```

Niektóre bardzo stare kompilatory zamiast `#include <iostream.h>` używają `#include <stream.h>`; jeśli mamy taki właśnie kompilator, powinniśmy albo postarać się o nowszy, albo skorzystać ze starszej książki. Zamiana `iostream.h` na `iostream` zaszła stosunkowo niedawno i niektóre kompilatory mogą jej jeszcze nie uwzględniać.

Pewne środowiska okienkowe uruchamiają program w osobnym oknie i po jego zakończeniu automatycznie okno to zamykają. Jak mówiliśmy w rozdziale 1., można wymusić zachowanie tego okna do chwili wciśnięcia klawisza; wystarczy w tym celu dodać przed instrukcją **`return`** wiersz:

```
cin.get();
```

W przypadku niektórych programów trzeba dodać dwa takie wiersze; powodują one, że program czeka na wciśnięcie klawisza. Kod ten lepiej zrozumiemy po przeczytaniu rozdziału 4.

Poprawki w programie

Może się okazać, że aby uruchomić niektóre przykłady z tej książki, trzeba nieco je zmienić. Najczęściej zmiana ta dotyczy omówionej przed chwilą kwestii zgodności z najnowszą wersją

standardu. Jeśli kompilator nie jest dostatecznie nowy, zamiast *iostream* trzeba używać *iostream.h* oraz pomijać deklarację przestrzeni nazw, **`namespace`**. Druga kwestia to używane środowisko programowania, które może wymagać użycia jednego czy dwóch wywołań `cin.get()`, gdyż bez tego wyniki działania programu znikną z ekranu. Jako że zmiany te są takie same we wszystkich programach, więc powyższej uwagi o zgodności ze standardem nie będziemy już dalej powtarzać. Następne uwagi tego typu będą dotyczyły innych możliwych niezgodności.

Kiedy już za pomocą edytora tekstu przepiszemy powyższy program (lub pobierzemy go ze strony wydawnictwa), za pomocą kompilatora C++ możemy stworzyć plik wykonywalny zgodnie z wytycznymi z rozdziału 1. Oto wynik działania programu z listingu 2.1:

```
Zabaw się językiem C++.
Nie pożałujesz!
```

Wejście i wyjście w języku C

Osoby programujące w języku C, widząc zamiast funkcji `printf()` jakieś `cout`, mogą przeżyć szok. W C++ można tak naprawdę używać funkcji `printf()`, `scanf()` i im podobnych, wystarczy tylko włączyć standardowy nagłówek języka C — *stdio.h*. Jednak niniejsza książka dotyczy języka C++, więc używamy tutaj narzędzi wejścia i wyjścia C++, które pod wieloma względami są doskonalsze od swoich odpowiedników z języka C.

Program w języku C++ składa się z cegiełek nazywanych *funkcjami*. Zwykle program opisuje główne zadania, którym odpowiadają funkcje. Przykład z listingu 2.1 jest prosty i zawiera tylko jedną funkcję, `main()`. Program *myfirst.cpp* ma następujące elementy:

- komentarze oznaczone przez `//`,
- dyrektywę preprocesora `#include`,
- nagłówek funkcji, **`int`** `main()`,
- dyrektywę **`using namespace`**,
- treść funkcji ograniczoną nawiasami klamrowymi, `{` i `}`,
- instrukcje wykorzystujące `cout` do wyświetlenia komunikatu,
- instrukcję **`return`** kończącą działanie funkcji `main()`.

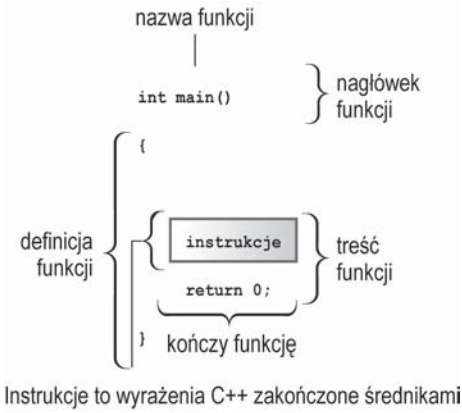
Najpierw przyjrzyjmy się dokładniej poszczególnym elementom. Dobrze będzie zacząć od funkcji `main()`, gdyż elementy ją poprzedzające, jak dyrektywy preprocesora, łatwiej będzie zrozumieć po zrozumieniu `main()`.

Funkcja `main()`

Jeśli odrzucimy wszystkie ozdobniki, okaże się, że program z listingu 2.1 ma bardzo prostą podstawową strukturę:

```
int main()
{
    instrukcje
    return 0;
}
```

Mamy tu funkcję `main()` i opis jej działania. Wszystko to składa się na *definicję funkcji*. Definicja składa się z dwóch części: wiersza **`int`** `main()` będącego *nagłówkiem funkcji* oraz ujętej w nawiasy klamrowe *treści funkcji*. Na rysunku 2.1 pokazano funkcję `main()`. Nagłówek to krótki opis interfejsu funkcji pozwalającego łączyć tę funkcję z resztą programu, treść funkcji zawiera instrukcje wykonywane przez tę funkcję. W języku C++ każda instrukcja powinna kończyć się średnikiem; nie należy zapominać o średnikach także przy przepisywaniu przykładów z książki.



Rysunek 2.1. Funkcja main()

Ostatnia instrukcja funkcji `main()`, **`return`** (nazywana *instrukcją powrotu*), kończy wykonywanie programu. Więcej o instrukcji powrotu dowiemy się dalej w tym rozdziale.

Instrukcje i średniki

Instrukcja to pewne polecenie dla komputera. Aby zrozumieć kod źródłowy, kompilator musi wiedzieć, gdzie jedna instrukcja się kończy, a zaczyna następna. W niektórych językach używa się separatora instrukcji — na przykład w FORTRAN-ie instrukcje są od siebie oddzielane znakami nowego wiersza. Z kolei w Pascalu można separator instrukcji pomijać w pewnych wypadkach — na przykład tuż przed słowem kluczowym `end`, gdzie separator ten nie oddziela od siebie dwóch instrukcji (zresztą tutaj pragmatycy i minimaliści spierają się, czy *można* należy rozumieć jako *trzeba*). Jednak C++, podobnie jak C, używa średnika jako *zakończenia* instrukcji, a nie separatora. Różnica polega na tym, że średnik właściwie jest częścią instrukcji, a nie czymś pomiędzy dwiema instrukcjami. Na praktykę przekłada się to tak, że w C++ nigdy nie wolno pomijać średników.

Nagłówek funkcji jako interfejs

Najważniejsze, co teraz musimy zapamiętać, to to, że składnia C++ wymaga rozpoczęcia definicji funkcji `main()` od następującego nagłówka: **`int`** `main()`. Składnię nagłówka funkcji omówimy dokładniej nieco dalej, ale teraz kilka słów dla tych, którzy nie potrafią powstrzymać swojej ciekawości.

W języku C++ funkcje są uruchamiane, czyli *wywoływane*, przez inne funkcje, a nagłówek funkcji opisuje interfejs funkcji wywoływanej na potrzeby funkcji wywołującej. Część znajdująca się przed nazwą funkcji to *typ wartości zwracanej przez funkcję*; typ ten mówi, jak informacja jest zwracana po wykonaniu funkcji do miejsca wywołania. W nawiasach za nazwą funkcji znajduje się *lista parametrów* lub inaczej *lista argumentów*. Opisuje ona przekazywanie informacji z funkcji wywołującej do funkcji wywoływanej. Trochę niedobrym przykładem jest tu funkcja `main()`, gdyż normalnie nie wywołuje się jej z innych części programu. Zwykle `main()` jest wywoływana przez kod rozruchowy dodawany przez kompilator między program a system operacyjny (Unix, Windows XP i każdy inny). Tak więc nagłówek funkcji w przypadku funkcji `main()` opisuje interfejs między tą funkcją a systemem operacyjnym.

Weźmy pod uwagę interfejs funkcji `main()` zaczynający się od słowa **`int`**. Funkcja C++ wywoływana przez inną funkcję może zwrócić funkcji wywołującej wartość — jest to *wartość zwracana*. W przypadku funkcji `main()` zwracana może być liczba całkowita; świadczy o tym użycie słowa kluczowego **`int`**. Dalej mamy parę pustych nawiasów. Oznaczają one, że funkcja `main()` nie pobiera żadnych informacji, czyli nie ma parametrów.

Tak więc nagłówek:

```
int main()
```

mówi, że funkcja `main()` zwraca funkcji ją wywołującej liczbę całkowitą oraz że nie pobiera z miejsca wywołania żadnych danych.

Obecnie wiele programów korzysta z klasycznego nagłówka języka C:

```
main()        // zgodnie z konwencją języka C
```

W klasycznym C pominięcie typu wartości zwracanej było równoważne stwierdzeniu, że zwracana jest wartość **`int`**. Jednak w C++ już tak nie jest.

Można użyć jeszcze jednej postaci nagłówka:

```
int main(void)        // styl bardzo dobitny
```

Słowo kluczowe **`void`** w nawiasach mówi, że funkcja nie ma żadnych parametrów. W języku C++ (choć już nie w C) oznacza dokładnie to samo co słowo kluczowe **`void`**; w C pusta lista w nawiasach sugeruje, że nie mówimy nic o parametrach.

Niektórzy programiści pomijają instrukcję powrotu i stosują następujący nagłówek:

```
void main()
```

Jest to logiczne, gdyż typ zwracany **`void`** oznacza, że funkcja nie zwraca żadnej wartości. Choć wiele środowisk taką konstrukcję obsługuje, to nie należy ona do standardu C++, więc w niektórych środowiskach może nie działać. Wobec tego należy takiego nagłówka unikać i trzymać się standardu; nie wymaga to zbyt wielkiego nakładu pracy.

W końcu standard ANSI/ISO C++ idzie na rękę tym, którzy narzekają, że muszą na końcu funkcji `main()` zawsze dopisywać instrukcję powrotu. Otóż jeśli kompilator dojdzie do końca funkcji `main()` i nie znajdzie instrukcji **`return`**, zachowa się tak, jakby na końcu było:

```
return 0;
```

Taka niejawna instrukcja powrotu jest dodawana tylko do funkcji `main()` i do żadnej innej.

Dlaczego nazwa `main()` jest jedyna w swoim rodzaju?

Istnieje pewien wyjątkowo przekonujący argument, aby nazwać funkcję z pliku *myfirst.cpp* `main()` — nie mamy innego wyjścia. Po prostu każdy program C++ musi mieć funkcję `main()` (i nie może to być żadne `Main()`, `MAIN()` czy `mane()` — wielkość liter też ma znaczenie). Program *myfirst.cpp* ma tylko jedną funkcję, więc to ona musi nazywać się `main()`. Kiedy uruchamiamy program napisany w języku C++, jego wykonywanie zaczyna się właśnie od funkcji `main()`. Jeśli zatem nie mamy funkcji `main()`, nasz program jest niekompletny i kompilator nie omieszka nam wytknąć tego braku.

Istnieją pewne wyjątki. Jeśli na przykład piszemy w systemie Windows bibliotekę dynamicznie wiązaną (DLL), funkcji `main()` może nie być. Kod takiej biblioteki może być używany przez inne programy Windows; nie jest to samodzielny program, więc nie potrzebuje funkcji `main()`. Także programy przeznaczone do pracy w specjalnych środowiskach, takich jak sterownik robota, mogą nie potrzebować funkcji `main()`. Jednak normalne programy nie mogą się bez tej funkcji obyć, a takimi właśnie programami będziemy się zajmować.

Komentarze w C++

W języku C++ podwójny ukośnik (`//`) oznacza początek komentarza. *Komentarz* to notatka zostawiona przez programistę czytelnikowi, w której zwykle objaśniane są pewne aspekty działania kodu. Kompilator komentarze pomija. W końcu zna przecież C++ przynajmniej równie dobrze jak programista, ale nie potrafi zrozumieć komentarzy. Z punktu widzenia kompilatora listing 2.1 wygląda tak, jakby tam żadnych komentarzy w ogóle nie było:

```
#include <iostream>
int main()
{
        using namespace std;
    cout << "Zabaw się językiem C++.";
    cout << endl;
    cout << "Nie pożałujesz!" << endl;
    return 0;
}
```

Komentarze w C++ zaczynają się od `//` i kończą się wraz z końcem wiersza. Komentarz może sam zajmować cały wiersz lub może być uzupełnieniem zwykłego kodu. Spójrzmy na pierwszy wiersz listingu 2.1:

```
// myfirst.cpp — wyświetla komunikat
```

W niniejszej książce wszystkie programy zaczynają się od komentarza zawierającego nazwę pliku z kodem źródłowym i krótkiego opisu programu. Jak wspomnieliśmy w pierwszym rozdziale, rozszerzenie nazwy pliku z programem zależy od używanego kompilatora. Nazwą może być na przykład też *myfirst.C* czy *myfirst.cxx*.

Wskazówka

Swoje programy trzeba dokumentować za pomocą komentarzy. Im program jest bardziej złożony, tym cenniejsze są komentarze. Nie tylko pomagają innym zrozumieć, o co w programie chodzi, ale pomagają też autorowi — szczególnie jeśli wraca do swojego programu po jakimś czasie.

Komentarze w stylu C

Język C++ pozwala używać także komentarzy znanych z C, ujmowanych w znaki `/*` i `*/`:

```
#include <iostream>  /* komentarz w stylu C */
```

Komentarze w stylu C kończą się znakami `*/`, a nie końcem wiersza, więc mogą mieć wiele wierszy. W programach można stosować jeden z pokazanych rodzajów komentarzy lub oba typy równocześnie. Warto jednak trzymać się komentarzy C++; w ich przypadku nie trzeba pamiętać o parowaniu symboli `/*` i `*/`, co oznacza mniej kłopotów. Zresztą w specyfikacji C99 do języka C dodano też komentarze `//`.

Preprocesor i plik `iostream`

Ten punkt omówimy w dużym skrócie — powiemy tylko o rzeczach niezbędnych. Kiedy w programie chcemy użyć zwykłych narzędzi C++ do obsługi wejścia i wyjścia, konieczne jest dodanie dwóch następujących wierszy:

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

Jeśli kompilatorowi taki zapis się nie spodoba (na przykład będzie narzekał na brak pliku *iostream*), można spróbować użyć jednego tylko wiersza:

```
#include <iostream.h>      // to zrozumieją starsze kompilatory
```

Tyle wiedzy wystarczy do uruchamiania programów, ale wyjaśnijmy jeszcze co nieco.

W języku C++, podobnie jak w C, istnieje *preprocesor*. Jest to program przetwarzający wstępnie pliki źródłowe przed zasadniczą kompilacją (w niektórych implementacjach C++ używa się programu translującego kod C++ na kod C; mówiliśmy o tym w rozdziale 1.; wprawdzie to też jest pewnego rodzaju preprocesor, to nie o taki nam teraz chodzi; preprocesor, który nas interesuje, rozumie dyrektywy zaczynające się od znaku #). Aby wywołać preprocesor, nie trzeba podejmować żadnych specjalnych działań; włącza się on automatycznie podczas kompilacji programu.

Na listingu 2.1 mamy dyrektywę preprocesora `#include`:

```
#include <iostream>      // dyrektywa PREPROCESORA
```

Dyrektywa ta powoduje dodanie do programu zawartości pliku *iostream*. Jest to typowe działanie preprocesora — dodanie lub zamiana tekstu w pliku źródłowym przed jego skompilowaniem.

Pojawia się pytanie, po co trzeba dodawać do naszego programu plik *iostream*. Chodzi o komunikację między programem a światem zewnętrznym. W nazwie *iostream io* oznacza *wejście*, czy-

li informacje przekazywane do programu, oraz *wyjście*, czyli informacje przekazywane z programu. Obsługa wejścia-wyjścia w C++ opiera się na kilku definicjach z pliku *iostream*. Program musi dołączyć odpowiednie definicje przed użyciem `cout` do pokazania komunikatu. Dyrektywa `#include` powoduje, że zawartość pliku *iostream* zostanie wraz z naszym programem przesłana do kompilatora. Zawartość tego pliku zastąpi wiersz `#include <iostream>`. Oryginalny plik nie zostanie zmieniony, natomiast do następnego etapu kompilacji przekazany zostanie plik złożony z naszego pliku źródłowego i z *iostream*.

Zapamiętaj!

Programy wykorzystujące do obsługi wejścia i wyjścia `cin` oraz `cout` muszą mieć dołączony plik `iostream` (lub, w niektórych systemach, *iostream.h*).

Nazwy plików nagłówkowych

Pliki takie jak *iostream* to *pliki włączane* lub inaczej *pliki nagłówkowe*. Kompilatory C++ zawsze są dostarczane wraz z szeregiem plików nagłówkowych, z których każdy obsługuje pewną grupę narzędzi. Zgodnie z tradycją języka C plikom nagłówkowym nadawano rozszerzenie *.h*, co łatwo pozwalało te pliki zidentyfikować. Na przykład plik nagłówkowy C *math.h* zawierał deklaracje funkcji matematycznych. Początkowo w C++ sytuacja wyglądała tak samo, ale ostatnio zmieniono to. Obecnie rozszerzenie *.h* zachowano dla starszych plików nagłówkowych C (które mogą być cały czas używane), natomiast pliki nagłówkowe samego C++ nie mają rozszerzenia w ogóle. Istnieją też pliki nagłówkowe C, które zostały zamienione na pliki nagłówkowe C++. Plikom tym zmieniono nazwę, odrzucając rozszerzenie *.h* oraz poprzedzając ich nazwy literą `c` (od języka C). Na przykład w C++ staremu plikowi nagłówkowemu *math.h* odpowiada plik *cmath*. Czasami wersje plików nagłówkowych C i C++ są takie same, a czasem nowsze wersje mają pewne udoskonalenia. W przypadku plików nagłówkowych samego C++, jak *iostream*, odrzucenie rozszerzenia *.h* to coś więcej niż poprawka kosmetyczna; pliki nagłówkowe bez tego rozszerzenia wykorzystują przestrzenie nazw, które za chwilę omówimy. W tabeli 2.1 zestawiono konwencje nazewnicze dotyczące plików nagłówkowych.

Tabela 2.1. Konwencje nazewnicze plików nagłówkowych

Rodzaj nagłówka	Konwencja	Przykład	Uwagi
Stary styl C++	kończy się **.h**	`iostream`.h	Używane w programach C++.
Stary styl C	kończy się **.h**	`math`.h	Używane w programach C i C++.
Nowy styl C++	brak rozszerzenia	`iostream`	Używane w programach C++ z **`namespace`** `std`.
Konwersja z C	przedrostek *c*, cmath brak rozszerzenia		Używane w programach C++, może zawierać cechy wykraczające poza C, jak choćby **`namespace`** `std`.

Biorąc pod uwagę tradycję C odróżniania różnego rodzaju plików po ich rozszerzeniach, rozsądne wydawało się użycie specjalnego rozszerzenia — na przykład *.hx* lub *.hxx* — do wyróżnienia plików nagłówkowych C++. Tak samo uważał komitet ANSI/ISO. Jednak wobec niemożności porozumienia się, jakie to powinno być rozszerzenie, ostatecznie postanowiono zrezygnować z rozszerzenia w ogóle.

Przestrzenie nazw

Jeśli zamiast pliku *iostream.h* używamy *iostream*, powinniśmy użyć następującej dyrektywy przestrzeni nazw, aby udostępnić *iostream* w programach:

```
using namespace std;
```

Jest to *dyrektywa* **using**. Najłatwiej jest przyjąć, że tak ona wygląda i że być musi, a o resztę się na razie nie martwić — możemy odłożyć dyskusję na przykład do rozdziału 9., gdzie omawiamy modele pamięci i przestrzenie nazw. Aby jednak nie być teraz ciemnym jak tabaka w rogu, powiemy krótko, o co tutaj chodzi.

Przestrzeń nazw to nowa cecha języka C++, która ma pomagać pisać programy składające się z kodu pochodzącego od różnych dostawców. Trzeba się liczyć z tym, że w dwóch nabytych pakietach mogą istnieć tak samo nazywające się funkcje — dajmy na to `wanda()`. Jeśli użyjemy funkcji `wanda()`, kompilator nie będzie wiedział, o którą nam chodzi. Przestrzeń nazw pozwala zgrupować ze sobą wszystkie moduły jednego dostawcy. Załóżmy, że firma MicroFlop Industries umieszcza swoje definicje w przestrzeni nazw `Microflop`. Wtedy ich funkcja `wanda()` będzie się nazywała `Microflop::wanda()`. Analogicznie przestrzeń nazw `Rybia` może zawierać produkty Rybiej Korporacji; w tym wypadku funkcja `wanda()` będzie nazywała się `Rybia::wanda()`. Teraz program będzie mógł używać obu funkcji:

```
Microflop::wanda("idziemy zatańczyć?");  // funkcja z przestrzeni nazw Microflop
Rybia::wanda("rybka zwana Pożądaniem");  // funkcja z przestrzeni nazw Rybia
```

Teraz klasy, funkcje i zmienne będące standardowymi składnikami kompilatora C++ trafiły do przestrzeni nazw `std`; dzieje się tak w wersjach plików nagłówkowych nieposiadających rozszerzeń. Wobec tego, na przykład, zmienna `cout` używana do przekazywania danych na zewnątrz, zdefiniowana w *iostream*, tak naprawdę nazywa się `std::cout`, a `endl` to tak naprawdę `std::endl`. Możemy zatem zrezygnować z dyrektywy **using** i pisać następująco:

```
std::cout << "Zabaw się językiem C++.";
std::cout << std::endl;
```

Jednak mało kto chce przepisywać kod nieuwzględniający przestrzeni nazw, wykorzystujący *iostream.h* i `cout`, na kod z przestrzeniami nazw wykorzystujący *iostream* i `std::cout`; dlatego właśnie stworzono dyrektywę **using** pozwalającą znacząco uprościć sobie to zadanie. Poniższy wiersz powoduje, że możemy używać nazw z przestrzeni `std` bez dopisywania przedrostka `std::`:

```
using namespace std;
```

Dyrektywa **using** powoduje, że wszystkie nazwy z przestrzeni `std` stają się dostępne. Obecnie przyjmuje się, że jest to podejście zbyt wygodnickie; należy raczej udostępnić te nazwy, których będziemy później używać, stosując inną formę dyrektywy **using**:

```
using std::cout;    // udostępnienie cout
using std::endl;    // udostępnienie endl
using std::cin;     // udostępnienie cin
```

Powyższe dyrektywy mogą zostać użyte w naszym programie zamiast poniższej:

```
using namespace std;     // przykład wygodnictwa - udostępnienie wszystkich nazw
```

Można będzie wtedy używać `cin`, `cout` i `endl` bez przedrostka `std::`. Jeśli jednak potrzebna okaże się inna nazwa z *iostream*, trzeba będzie ją dodać osobną dyrektywą **using**. W niniejszej książce będziemy stosowali początkowo podejście wygodnickie — po pierwsze dlatego, że w przypadku małych programów nie ma specjalnego znaczenia to, jak radzimy sobie z przestrzeniami nazw, a po drugie ważniejsze są dla nas na razie inne, bardziej podstawowe elementy języka C++. Dalej pokażemy jeszcze inne techniki obsługi przestrzeni nazw.

Wypisywanie danych — `cout`

Przyjrzyjmy się teraz, jak wyświetla się komunikat. W programie *myfirst.cpp* używamy następującej instrukcji:

```
cout << "Zabaw się językiem C++.";
```

Część w cudzysłowach to pokazywany komunikat. W C++ każdy ciąg znaków ujętych w podwójne cudzysłowy to *łańcuch znakowy*. Zapis << oznacza wysłanie danych do `cout`; symbole te wskazują kierunek przepływu informacji. A czym właściwie jest `cout`? Jest to predefiniowany obiekt, który potrafi pokazywać różne rzeczy: łańcuchy, liczby, pojedyncze znaki (jak zapewne Czytelnicy pamiętają, *obiekt* to konkretny egzemplarz klasy, a *klasa* to definicja sposobu przechowywania danych i zasady ich użycia).

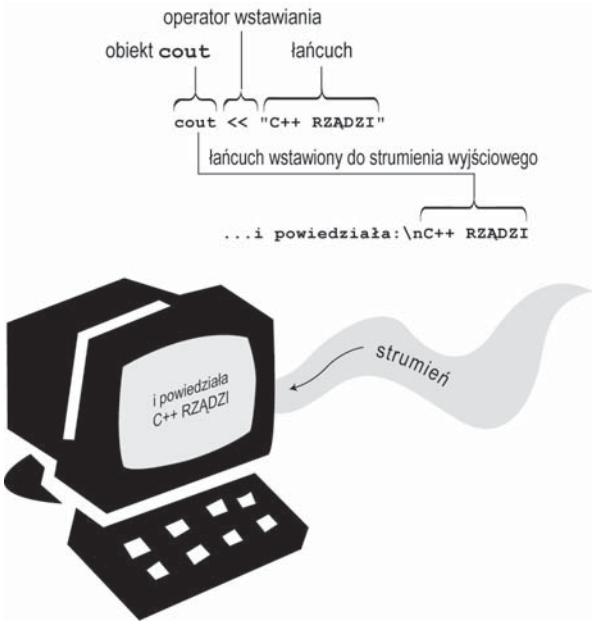
No cóż, omawianie teraz użycia obiektów jest dość trudne, gdyż obiektami nie będziemy zajmować się w kilku następnych rozdziałach. Jest to jedna z mocnych stron obiektowości — aby używać obiektu, nie trzeba zbyt wiele wiedzieć o jego wewnętrznej budowie. Wystarczy znać interfejs obiektu, czyli wiedzieć, jak go używać. Obiekt `cout` ma bardzo prosty interfejs. Jeśli `string` to łańcuch znakowy, aby go wyświetlić, wystarczy zapisać:

```
cout << string;
```

I tyle musimy wiedzieć o wyświetlaniu w C++ napisów, ale teraz zastanówmy się, jak ten proces wygląda w ogólniejszej perspektywie C++. Dane wyjściowe to strumień, czyli ciąg znaków, generowany przez program. Obiekt `cout`, którego właściwości opisano w *iostream*, reprezentuje ten strumień. Dla obiektu `cout` zdefiniowano operator wstawiania (<<) wstawiający dane bezpośrednio do strumienia. Weźmy pod uwagę następujący fragment kodu (pamiętajmy o końcowym średniku):

```
cout << "Zabaw się językiem C++.";
```

Do strumienia wyjściowego wstawiany jest łańcuch „Zabaw się językiem C++.”. Zamiast zatem mówić, że nasz program wyświetla komunikat, możemy powiedzieć, że wstawia on komunikat do strumienia wyjściowego. Brzmi to nieco poważniej (rysunek 2.2).



Rysunek 2.2. Wyświetlanie łańcucha za pomocą cout

Przeciążanie po raz pierwszy

Osoby, które mają już doświadczenie w języku C zauważyły zapewne, że operator wstawienia do strumienia, `<<`, wygląda zupełnie jak operator bitowego przesunięcia w lewo. Jest to przykład *przeciążania operatorów*, dzięki któremu ten sam symbol operatora może zmieniać swoje znaczenie. Kompilator na podstawie kontekstu określa, o które znaczenie chodzi. Zresztą sam język C ma pewne elementy przeciążania operatorów; symbol `&` oznacza operator adresu oraz operator bitowego AND. Symbol `*` oznacza mnożenie oraz dereferencję (wyłuskanie). Dla nas w tej chwili ważne jest nie tyle znaczenie tych operatorów, co fakt, że ten sam symbol może mieć wiele znaczeń i kompilator wybiera jedno z nich na podstawie kontekstu (tak samo jak my, słysząc słowo „piłka”, o czym innym myślimy w kontekście „piłki do drewna”, a o czym innym w przypadku „piłki na aucie”). Język C++ rozszerza pojęcie przeciążania operatorów, umożliwiając zmianę znaczenia operatora w odniesieniu do typów użytkownika; typy takie to klasy.

Manipulator `endl`

Teraz zajmijmy się nieco dziwnie wyglądającym zapisem drugiej instrukcji zapisu danych do strumienia z listingu 2.1:

```
cout << endl;
```

`endl` to specjalny zapis w C++ oznaczający ważne pojęcie, jakim jest początek nowego wiersza. Wstawienie `endl` do strumienia wyjściowego powoduje, że kursor na ekranie przechodzi na początek następnego wiersza. Specjalne symbole, takie jak `endl`, mające dla `cout` specyficzne znaczenie, nazywamy *manipulatorami*. Tak jak `cout`, tak i `endl` zdefiniowano w pliku nagłówkowym `iostream` w przestrzeni nazw `std`.

Zwróćmy uwagę na to, że `cout` po wypisaniu łańcucha znakowego nie przechodzi automatycznie do następnego wiersza, więc pierwsza instrukcja z `cout` z listingu 2.1 zostawia kursor po kropce na końcu łańcucha. Działanie każdego następnego `cout` zaczyna się od miejsca, w którym skończyło się działanie poprzedniego, więc pominięcie `endl` dałoby wynik następujący:

```
Zabaw się językiem C++.Nie pożałujesz!
```

Zwróćmy uwagę na to, że „N" jest tuż za kropką. Oto kolejny przykład. Niech dany będzie następujący kod:

```
cout << "Dobry,";
cout << "zły, ";
cout << "i ukulele";
cout << endl;
```

Kod ten da w wyniku:

```
Dobry, zły, i ukulele
```

Tutaj też początek jednego łańcucha znajduje się zaraz za końcem poprzedniego; jeśli chcemy napisy te oddzielić spacją, musimy tę spację włączyć do jednego z napisów (aby powyższe przykłady wypróbować, trzeba umieścić je w kompletnym programie z funkcją `main()`).

Znak nowego wiersza

C++ ma jeszcze inny, bardziej archaiczny sposób wskazywania końca wiersza — znany z C zapis `\n`:

```
cout << "Co dalej?\n";          // \n oznacza początek nowego wiersza
```

Kombinacja `\n` jest traktowana jako pojedynczy znak, *znak nowego wiersza*. Jeśli wyświetlamy łańcuch znakowy, łatwiej jest wstawić do niego znak nowego wiersza, niż dopisywać osobno `endl`:

```
cout << "Jowisz to wielka planeta.\n";     // pokazuje zdanie,przechodzi
                                           // do następnego wiersza
cout << "Jowisz to wielka planeta." << endl;     // pokazuje zdanie,  przechodzi
                                                 //  do  następnego wiersza
```

Z drugiej strony, jeśli chodzi o wypisanie pojedynczego znaku nowego wiersza, obie metody wymagają tyle samo pisania, ale dla większości osób pisanie `endl` jest łatwiejsze:

```
cout << "\n";                             // zaczynamy nowy wiersz
cout << endl;                             // zaczynamy nowy wiersz
```

W książce zwykle będziemy używali znaku nowego wiersza `\n` wewnątrz łańcuchów i manipulatora `endl` w pozostałych sytuacjach.

Znak nowego wiersza to przykład specjalnej kombinacji znaków z odwrotnym ukośnikiem; konstrukcjami takimi zajmiemy się dokładniej w rozdziale 3.

Formatowanie kodu źródłowego C++

Niektóre języki, na przykład FORTRAN, bazują na podziale na wiersze i muszą mieć jedną instrukcję w wierszu. W ich wypadku znak nowego wiersza[1] służy jako separator instrukcji. Jednak w C++ średniki oznaczają koniec każdej instrukcji, zatem znak nowego wiersza może być traktowany tak samo jak spacje i tabulatory. Wobec tego w C++ można używać spacji w miejsce nowych wierszy i odwrotnie. Dzięki temu jedną instrukcję można podzielić na wiele wierszy, można też w jednym wierszu umieścić wiele instrukcji. Moglibyśmy na przykład program *myfirst.cpp* sformatować następująco:

```
#include <iostream>
      int
main
() {   using
    namespace
         std; cout
            <<
"Zabaw się językiem C++."
;    cout <<
endl; cout <<
"Nie pożałujesz!" <<
endl;return 0; }
```

Kod taki wygląda brzydko, ale jest poprawny i działa. Trzeba jednak pamiętać o pewnych ważnych rzeczach. W C ani w C++ nie można wstawiać spacji, tabulatora ani znaku nowego wiersza do wnętrza nazwy, nie można wstawiać bezpośrednio znaku nowego wiersza do wnętrza łańcucha znakowego. Oto przykłady działań niedozwolonych:

```
int ma in()          // niedopuszczalne - spacje w nazwie
re
turn 0;              // niedopuszczalne - znak nowego wiersza w słowie kluczowym
cout << "Litwo ojczyzno
moja!";              // niedopuszczalne - znak nowego wiersza w łańcuchu znakowym
```

Elementy języka a białe znaki

Niepodzielny element w wierszu kodu nazywany bywa *elementem języka* (ang. *token*) — rysunek 2.3. Ogólnie rzecz biorąc, poszczególne elementy trzeba od siebie oddzielać spacją, tabulatorem lub końcem wiersza; wszystkie te znaki zbiorczo określamy jako *białe znaki*. Niektóre pojedyn-

[1] *A ściślej rzecz biorąc, znak powrotu karetki, co w praktyce na jedno wychodzi*

cze znaki, jak nawiasy czy przecinki, są elementami, które nie muszą być oddzielane białymi znakami. Oto kilka przykładów pokazujących, kiedy białe znaki mogą być używane i kiedy mogą być pomijane:

```
return0;          // ŹLE - powinno być return 0;
return(0);        // DOBRZE - pominięto białe znaki
return (0);       // DOBRZE - użyto białego znaku
intmain()         // ŹLE - pominięto białe znaki
int main()        // DOBRZE - pominięto białe znaki w ()
int main ( )      // też DOBRZE - dodano białe znaki w ( )
```

Rysunek 2.3. Elementy języka a białe znaki

Styl pisania kodu źródłowego w C++

Wprawdzie język C++ pozwala praktycznie dowolnie formatować programy, ale będą one łatwiejsze do czytania, jeśli zachowamy rozsądny styl kodowania. Pisanie działającego, ale brzydkiego kodu nie powinno być dla nas źródłem satysfakcji. Większość programistów stosuje style podobny jak na listingu 2.1:

- W jednym wierszu jest jedna instrukcja.
- Otwierający i zamykający nawias funkcji mają swoje własne wiersze.
- Instrukcje objęte funkcją są wcięte względem nawiasów klamrowych.
- Przy nawiasach związanych z nazwą funkcji nie umieszcza się białych znaków.

Pierwsze trzy zasady po prostu mają zapewnić uporządkowanie kodu i poprawić jego czytelność. Czwarta pomaga odróżnić funkcje od pewnych wbudowanych struktur C++, jak pętle, w których białych znaków już używamy. O innych wytycznych będziemy informować w miarę ich używania.

Instrukcje C++

Program w języku C++ to zbiór funkcji, a każda funkcja to zbiór instrukcji. C++ ma kilka rodzajów instrukcji, więc przyjrzyjmy się możliwym ich rodzajom. Na listingu 2.2 pokazano nowe rodzaje instrukcji. Na początku mamy *instrukcję deklaracji* tworzącą zmienną. Dalej *instrukcja przypisania* nadaje zmiennej wartość. Poza tym pokazano nowe możliwości `cout`.

Listing 2.2. carrots.cpp

```
// carrots.cpp — program przetwarzający jedzenie
// pokazuje użycie zmiennych

#include <iostream>

int main()
{
    using namespace std;

    int carrots;                        // deklarujemy zmienną typu int

    carrots = 25;                       // przypisujemy tej zmiennej wartość
    cout << "Mam ";
    cout << carrots;                    // pokazujemy wartość tej zmiennej
    cout << " marchewek.";
    cout << endl;
    carrots = carrots - 1;              // modyfikujemy zmienną
    cout << "Chrum, chrum. Teraz mam " << carrots << " marchewki." << endl;
    return 0;
}
```

Pusty wiersz służy do oddzielenia deklaracji od reszty programu. Praktyka ta jest standardową konwencją C, ale w C++ też jest powszechnie stosowana. Oto wyniki działania programu z listingu 2.2:

```
Mam 25 marchewek.
Chrum, chrum. Teraz mam 24 marchewki.
```

Teraz zajmiemy się analizą tego programu.

Instrukcje deklaracji i zmienne

Komputery to precyzyjne, uporządkowane maszyny. Aby zapisać w komputerze jakąkolwiek informację, trzeba wskazać położenie tej danej oraz określić ilość zajmowanego przez nią miejsca. Względnie prostą metodą robienia tego w C++ jest użycie *instrukcji deklaracji*, która określa

rodzaj zajmowanej pamięci i nadaje temu miejscu nazwę. Na przykład program z listingu 2.2 ma następującą instrukcję deklaracji (uwaga na przecinek!):

```
int carrots;
```

Instrukcja ta mówi, że program potrzebuje tyle pamięci, ile zajmuje liczba całkowita oznaczana w C++ nazwą **int**. Już sam kompilator zajmie się szczegółami alokacji i oznaczenia odpowiedniego miejsca w pamięci. C++ może obsłużyć kilka rodzajów danych, czyli kilka typów danych; jednym z najbardziej elementarnych jest właśnie typ **int**. Oznacza on liczbę całkowitą bez części ułamkowej. Dane typu **int** mogą być dodatnie lub ujemne, natomiast konkretna ich wielkość zależna jest od używanej implementacji. W rozdziale 3. powiemy więcej o typie danych **int** i innych typach podstawowych.

Deklaracja nie tylko określa typ, ale deklaruje nazwę tak, że natykając się dalej na nazwę `carrots`, program będzie odnosił się zawsze do tego samego miejsca w pamięci. `carrots` to *zmienna* — jak sugeruje sama nazwa, można zmieniać jej wartość. W C++ wszystkie zmienne trzeba deklarować. Jeśli jakaś deklaracja zostanie pominięta, kompilator zgłosi błąd mówiący, że program próbuje gdzieś dalej używać niezadeklarowanej zmiennej. (Można zresztą pominąć deklarację, aby zobaczyć, jak zareaguje na to kompilator. Wtedy, jeśli w przyszłości otrzymamy podobny komunikat, będziemy wiedzieli już, jakiego błędu należy szukać.)

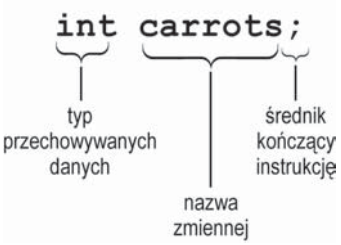
Dlaczego zmienne trzeba deklarować?

W niektórych językach, w tym w BASIC-u, zmiennych się nie deklaruje, a utworzenie nowej zmiennej jest równoważne użyciu nowej nazwy. Może się to wydawać wygodne, ale jest takie tylko na krótką metę. Chodzi o to, że kiedy pomylimy się w nazwie zmiennej, tworzymy nową zmienną, nie zdając nawet sobie z tego sprawy. Weźmy zatem pod uwagę następujący fragment kodu BASIC:

```
StaryZamek = 34
...
StaryZamyk = StaryZamek + JeszczeDuchy
...
PRINT StaryZamek
```

`StaryZamyk` jest napisany niepoprawnie, więc tak naprawdę wartość zmiennej `StaryZamek` nie zostanie zmodyfikowana. Tego typu błędy są trudne do odnalezienia, gdyż program jest w pełni zgodny z zasadami języka BASIC. Jednak w C++ zmienna `StaryZamek` musi być wcześniej zadeklarowana, a błędnie nazwana zmienna `StaryZamyk` nie będzie nigdzie zadeklarowana. Wobec tego analogiczny kod C++ naruszy zasadę obowiązku deklarowania zmiennych przed ich użyciem, dzięki czemu kompilator wychwyci błąd i od razu pozwoli usunąć problem.

Tak więc deklaracja określa, jakiego typu dane będą przechowywane pod określoną nazwą. W naszym konkretnym wypadku program tworzy zmienną `carrots`, która zawiera liczbę całkowitą (rysunek 2.4).



Rysunek 2.4. Deklaracja zmiennej

Powyższa deklaracja to *deklaracja definiująca* lub krócej po prostu *definicja*. Jej obecność powoduje, że kompilator alokuje w pamięci miejsce na taką zmienną. Istnieją bardziej skomplikowane przypadki, kiedy mamy *deklaracje referencji*. Informują one komputer, że ma użyć zmiennej już gdzieś indziej zdefiniowanej. Ogólnie rzecz biorąc, deklaracja nie musi być definicją, choć w pokazanym przypadku akurat jest.

Osoby znające C lub Pascala są z deklaracjami zmiennych za pan brat. Jednak i dla nich znaleźć można niespodziankę — w C i Pascalu wszystkie deklaracje zmiennych pojawiają się na samym początku funkcji lub procedury. Jednak w C++ takie ograniczenie nie istnieje; nawet do stylu pisania w C++ należy deklarowanie zmiennych tuż przed ich pierwszym użyciem. W ten sposób nie trzeba przebijać się przez kod, aby sprawdzić, jakiego typu jest zmienna. Przykłady pokażemy dalej w tym rozdziale. Wadą takiego stylu jest z kolei brak jednego miejsca, w którym można byłoby sprawdzić, jakie zmienne są w danej funkcji użyte. (Standard C99 przewiduje dla języka C prawie takie same zasady deklaracji, jakie obowiązują w C++.)

Wskazówka

W języku C++ zalecany styl deklarowania zmiennych zakłada deklarowanie ich możliwie blisko miejsca pierwszego użycia.

Instrukcja przypisania

Instrukcja przypisania powoduje przypisanie wartości pewnemu miejscu w pamięci. Na przykład instrukcja:

```
carrots = 25;
```

powoduje przypisanie liczby 25 miejscu oznaczonemu jako zmienna `carrots`. Symbol = to *operator przypisania*. Niezwykłą cechą C++ (a także C) jest możność wielokrotnego użycia tego operatora w jednej instrukcji. Na przykład poniższy kod jest poprawny:

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

Przypisanie jest robione od strony prawej do lewej, więc najpierw zmiennej `steinway` przypisujemy 88, potem wartość zmiennej `steinway` (88) przypisujemy zmiennej `baldwin`, w końcu wartość zmiennej `baldwin` (88) przypisujemy zmiennej `yamaha`. Język C++ hołduje obowiązującej w C tendencji do pisania podejrzanie wyglądającego kodu.

Druga instrukcja przypisania z listingu 2.2 pokazuje, że wartość zmiennej można zmieniać:

```
carrots = carrots - 1;   // modyfikacja wartości zmiennej
```

Wyrażenie znajdujące się po prawej stronie operatora przypisania (`carrots - 1`) to przykład działania arytmetycznego. Komputer odejmie od 25, wartości `carrots`, jedynkę, uzyskując w wyniku 24. Operator przypisania wstawi tę nową wartość do zmiennej `carrots`.

Nowa sztuczka z `cout`

Aż do teraz pokazywane w tym rozdziale przykłady użycia `cout` korzystały ze stałych napisów. W kodzie z listingu 2.2 przekazujemy do `cout` także zmienną, której wartość jest liczbą całkowitą:

```
cout << carrots;
```

Program nie pokaże słowa `carrots`, ale pokaże liczbę z tej zmiennej — w tym wypadku 25. Tak naprawdę mamy tutaj dwa ciekawe chwyty. Po pierwsze, `cout` zastępuje `carrots` odpowiednią wartością liczbową, 25. Po drugie, wartość ta jest zamieniana na odpowiednie znaki.

Jak widać, `cout` może działać z łańcuchami znakowymi i z liczbami całkowitymi. Może to wydawać się mało istotne, ale pamiętajmy, że liczba 25 to co innego niż napis „`25`”. Napis zawiera znaki składające się na liczbę (znak 2 i znak 5). Program wewnętrznie przechowuje osobno znak 2, osobno znak 5. Aby pokazać napis, `cout` po prostu pokazuje kolejne jego znaki. Jednak liczba całkowita 25 jest zapisywana wewnętrznie jako liczba właśnie; nie w formie kolejnych cyfr, ale jako liczba binarna (więcej na ten temat można przeczytać w dodatku A). Ważne jest, że `cout` musi przełożyć liczbę całkowitą na znaki, gdyż inaczej nie można jej pokazać. Co więcej, `cout` jest na tyle inteligentne, że potrafi stwierdzić, że `carrots` to liczba całkowita wymagająca konwersji.

Pokazuje to, jak wygodne jest `cout` — o ile wygodniejsze od funkcji `printf()` znanej ze starego C. Aby pokazać w C łańcuch „`25`” i liczbę 25 w języku C, trzeba było użyć następujących wywołań `printf()`:

```
printf("Prezentacja łańcucha: %s\n", "25");
printf("Prezentacja liczby całkowitej: %d\n", 25);
```

Nie wnikając tutaj w zawiłości `printf()`, stwierdźmy tylko, że trzeba użyć specjalnych kodów (`%s` i `%d`), aby wskazać, czy chodzi nam o łańcuch znakowy czy o liczbę całkowitą. Jeśli nakażemy `printf()` pokazać łańcuch, ale pomyłkowo podamy liczbę całkowitą, błąd ten nie zostanie wychwycony — `printf()` pokaże śmieci znajdujące się akurat w pamięci.

Inteligentne zachowanie `cout` wynika z obiektowości C++. Operator wstawiania (`<<`) dostosowuje swoje zachowanie do typu danych, na których operuje. Jest to przykład przeciążania operatorów. W następnych rozdziałach, kiedy będziemy mówić o przeciążaniu funkcji i operatorów, pokażemy, jak samemu oprogramować tak inteligentne zachowania.

cout a printf()

Osoby, które znają C i używały funkcji `printf()`, często uważają, że `cout` wygląda dość dziwnie. Niejednokrotnie nawet wolą korzystać ze zdobytego z takim trudem mistrzostwa operowania tą pierwszą funkcją. Jednak tak naprawdę `cout` wcale nie wygląda dziwniej niż `printf()` wraz ze wszystkimi jej specyfikacjami konwersji. Co ważniejsze, `cout` ma istotne zalety — możliwość „odgadywania" typów oznacza większą odporność na błędy. Poza tym `cout` jest *rozszerzalne* — można przedefiniować operator `<<` tak, aby `cout` rozpoznawało i pokazywało prawidłowo nowe typy danych użytkownika. Nawet osoby korzystające z zaawansowanych możliwości `printf()` mogą uzyskać podobne efekty, korzystając z bardziej zawansowanych form `cout` (więcej na ten temat w rozdziale 17.).

Inne instrukcje C++

Przyjrzyjmy się teraz kolejnym przykładom instrukcji C++. Program z listingu 2.3 stanowi rozszerzenie poprzedniego przykładu, gdyż użytkownik może wprowadzić wartość. Służy do tego obiekt `cin`, wejściowy odpowiednik `cout`. Poza tym w programie tym pokazano jeszcze inny sposób wykorzystania wszechstronności obiektu `cout`.

Listing 2.3. getinfo.cpp

```
// getinfo.cpp — wejście i wyjście
#include <iostream>

int main()
{
    using namespace std;

    int carrots;

    cout << "Ile masz marchewek?" << endl;
    cin >> carrots;     // wczytywanie w C++
    cout << "Proszę, oto jeszcze dwie. ";
    carrots = carrots + 2;
    // w następnym wierszu sklejamy wyniki
     cout << "Teraz masz " << carrots << " marchewek." << endl;
    return 0;
}
```

Oto przykładowe wyniki działania programu z listingu 2.3:

```
Ile masz marchewek?
12
Proszę, oto jeszcze dwie. Teraz masz 14 marchewek.
```

Program pokazuje dwie nowe możliwości: użycie obiektu `cin` do odczytu danych z klawiatury oraz połączenie czterech instrukcji pokazywania danych w jedną.

Użycie obiektu `cin`

Jak widać w wynikach działania programu z listingu 2.3, wprowadzona z klawiatury wartość 12 jest przypisywania zmiennej `carrots`. Cudu tego dokonuje następująca instrukcja:

```
cin >> carrots;
```

Wizualnie wygląda ta instrukcja tak, jakby dane wpływały z obiektu `cin` do zmiennej `carrots`. Oczywiście istnieje też bardziej formalny opis tego procesu. Tak jak C++ traktuje dane wynikowe jako strumień znakowy, tak samo strumień wejściowy jest strumieniem znakowym wpływającym do programu. Plik *iostream* zawiera definicję obiektu `cin` opisującego strumień wejściowy. W przypadku wyjścia operator << wstawia znaki do strumienia wyjściowego; w przypadku wejścia `cin` wykorzystuje operator >> do pobrania znaków ze strumienia wejściowego. Zwykle po prawej stronie tego operatora podaje się zmienną, do której mają być pobrane dane (symbole << i >> zostały wybrane tak, aby wizualnie sugerować kierunek przepływu informacji).

Obiekt `cin`, tak samo jak `cout`, jest inteligentny, gdyż potrafi konwertować dane wejściowe będące łańcuchem znaków z klawiatury na postać odpowiednią dla zmiennej. W tym wypadku w programie zdeklarowano zmienną całkowitoliczbową `carrots`, więc dane wejściowe są konwertowane na taki zapis, w jakim zapisywane są liczby całkowite.

Złączanie za pomocą `cout`

Druga ciekawa technika pokazana w pliku *getinfo.cpp* to połączenie czterech instrukcji wyjścia w jedną. W pliku *iostream* zdefiniowano operator <<, tak że można za jego pomocą łączyć (konkatenować) wyniki następująco:

```
cout << "Teraz masz " << carrots << " marchewek." << endl;
```

Dzięki temu możemy mieć w jednej instrukcji pokazywanie łańcucha wynikowego i liczb wynikowych. Powyższy wiersz kodu jest równoważny następującemu fragmentowi:

```
cout << "Teraz masz ";
cout << carrots;
cout << " marchewek.";
cout << endl;
```

Aby ułatwić sobie zrozumienie tego kodu, możemy zapisać go nieco inaczej — każdy pokazywany element umieścić w osobnym wierszu:

```
cout << "Teraz masz "
     << carrots
     << " marchewek."
     << endl;
```

C++ pozwala dość dowolnie stosować białe znaki i formatować tekst; spacje są równoważne znakom nowego wiersza. Pokazany powyżej kod jest dobrym rozwiązaniem w sytuacjach, kiedy zaczynamy się gubić w zbyt skomplikowanych wyrażeniach.

I jeszcze jedna uwaga. Zdanie:

```
Teraz masz 14 marchewek.
```

pojawia się w tym samym wierszu co:

```
Proszę, oto jeszcze dwie.
```

Dzieje się tak dlatego, że wyniki działania instrukcji z `cout` są umieszczane zaraz za wynikami poprzedniej takiej instrukcji — nawet wtedy, gdy instrukcje te są od siebie oddzielone innymi instrukcjami.

`cin` i `cout` — klasy po raz pierwszy

Wiemy już tyle o `cin` i `cout`, że na miejscu będzie ujawnienie nieco magii obiektowości. Powiemy tutaj głównie o pojęciu klas. W rozdziale 1. krótko powiedzieliśmy, że klasy są jednym z najważniejszych pojęć programowania obiektowego C++.

Klasa to typ danych zdefiniowany przez użytkownika. Aby zdefiniować klasę, opisujemy, jakiego typu informacje mogą być w niej umieszczane i jakiego typu akcje mogą być na tych danych przeprowadzane. Klasa ma się do obiektu tak, jak typ ma się do zmiennej — definicja klasy mówi, jak dane są zbudowane i jak można ich używać, zaś obiekt to byt tworzony zgodnie z tą definicją. Możemy też wyjść poza informatykę — klasa jest analogią do kategorii takiej jak słynni aktorzy, zaś obiekt to konkretny aktor, na przykład żaba Kermit. Możemy tę analogię poprowadzić dalej i powiedzieć, że klasowy opis aktorów zawierać będzie definicje możliwych działań, jak czytanie tekstu roli, wyrażanie smutku, grożenie, przyjmowanie nagrody i tak dalej. Osobom zaznajomionym z inną terminologią obiektowości można powiedzieć, że klasa w C++ odpowiada bytowi określanemu w innych językach jako *typ obiektowy*, a obiekt C++ odpowiada instancji obiektu.

Teraz przejdźmy do konkretów. Przypomnijmy sobie deklarację zmiennej:

```
int carrots;
```

W ten sposób tworzona jest zmienna `carrots` mająca cechy charakterystyczne typu danych **`int`**. Wobec tego zmienna `carrots` może przechowywać liczby całkowite i można używać jej na ściśle określone sposoby, na przykład do dodawania i odejmowania. Teraz przyjrzyjmy się `cout`. Jest to obiekt mający cechy charakterystyczne dla klasy `ostream`. Definicja klasy `ostream` (zdefiniowanej w pliku *iostream*) opisuje, jakiego typu dane przechowywane są w obiektach `ostream` oraz jakie operacje mogą być na takich obiektach wykonywane — na przykład wstawianie do strumienia wyjściowego liczby lub łańcucha znakowego. Analogicznie obiekt `cin` tworzony jest zgodnie z definicją klasy `istream` także zdefiniowanej w pliku *iostream*.

Zapamiętaj!

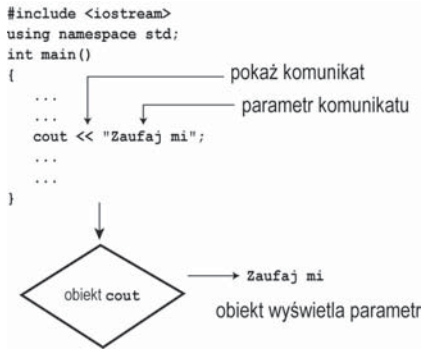
Klasa opisuje wszystkie właściwości typu danych, a obiekt to byt stworzony zgodnie z taką definicją.

Wiemy już, że klasy to typy definiowane przez użytkownika, ale przecież typowy użytkownik nie stworzył klas `ostream` i `istream`. Tak jak funkcje mogą pochodzić z bibliotek funkcji, klasy mogą pochodzić z bibliotek klas. Dotyczy to właśnie klas `ostream` i `istream`. Formalnie nie są one częścią samego języka C++; są przykładami klas, które są udostępniane wraz z tym językiem. Definicje tych klas znajdują się w pliku *iostream*, a nie są wbudowane w kompilator. Można nawet zmodyfikować definicje tych klas, choć nie jest to najlepszy pomysł (tak naprawdę jest to pomysł koszmarny). Rodzina klas `iostream` i związana z nią rodzina `fstream` (opisująca wejście i wyjście plikowe) to jedyne grupy klas, które występują we wszystkich, nawet najwcześniejszych implementacjach C++. Jednak komitet ANSI/ISO C++ dodał do standardu jeszcze kilka innych bibliotek klas. Poza tym większość implementacji zawiera definicje dodatkowych klas. Właśnie te dodatkowe klasy obsługujące systemy Unix, Macintosh i Windows stanowią w dużej mierze o popularności C++.

Opis klasy zawiera wszystkie działania, jakie można na obiektach danej klasy wykonywać. Aby wykonać takie działanie na konkretnym obiekcie, trzeba do tego obiektu przesłać komunikat. Jeśli na przykład chcemy, aby obiekt `cout` wyświetlił napis, wysyłamy mu komunikat typu „Hej, obiekcie, wyświetl to!”. W języku C++ komunikaty można przesyłać na kilka sposobów. Jeden sposób to użycie metod klasy — jest on bardzo podobny do wywoływania zwykłych funkcji. Drugi sposób, którego używamy w odniesieniu do `cin` i `cout`, to przedefiniowanie operatora. Tak więc instrukcja:

```
cout << "Nie jestem żadnym oszustem."
```

wykorzystuje przedefiniowany operator << do wysłania komunikatu do obiektu `cout`. W tym wypadku wiadomość przekazywana jest w parametrze będącym wyświetlanym łańcuchem (rysunek 2.5).



Rysunek 2.5. Wysyłanie komunikatu do obiektu

Funkcje

Funkcje to moduły, z których składają się programy C++, poza tym są one kluczowe dla definicji obiektowych C++, więc trzeba je dobrze poznać. Niektóre kwestie związane z funkcjami są trudne, więc dogłębnie funkcjami zajmiemy się dopiero w rozdziałach 7. i 8. Jeśli jednak chodzi o podstawowe zagadnienia dotyczące funkcji, to ich poznanie już teraz ułatwi zrozumienie dalszego toku wykładu. Reszta tego rozdziału poświęcona jest podstawom funkcji.

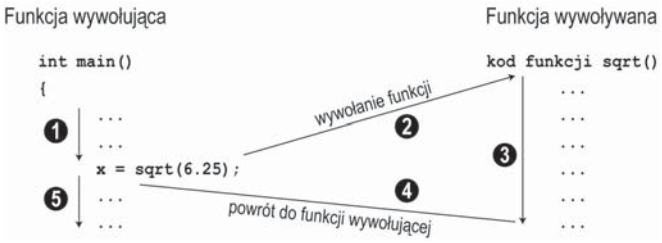
Funkcje w C++ mają dwie odmiany: zwracające wartości i niezwracające wartości. Przykłady obu można znaleźć w standardowej bibliotece funkcji, można też tworzyć nowe funkcje obu tych grup. Przyjrzyjmy się funkcjom bibliotecznym zwracającym wartość i zobaczmy, jak samemu napisać taką funkcję.

Użycie funkcji zwracającej wartość

Jeśli funkcja zwraca wartość, wartość tę można przypisać zmiennej. Na przykład biblioteka standardowa C/C++ zawiera funkcję `sqrt()` zwracającą pierwiastek kwadratowy z przekazanej liczby. Załóżmy, że chcemy wyliczyć pierwiastek kwadratowy z 6,25 i przypisać go zmiennej `x`. Robimy to następującą instrukcją:

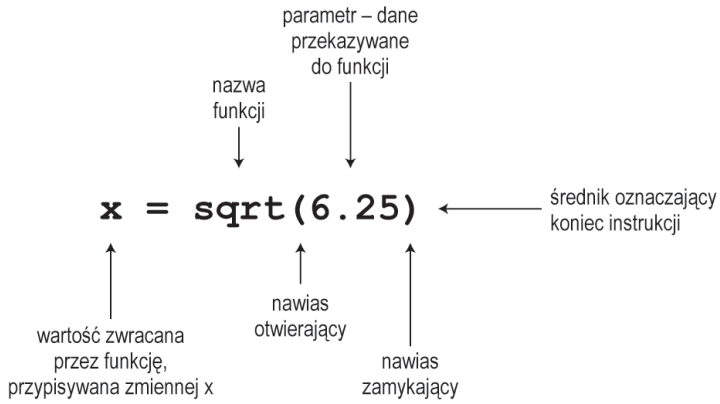
```
x = sqrt(6.25); // zwraca 2.5 i przypisuje tę wartość zmiennej x
```

Wyrażenie `sqrt(6.25)` *wywołuje* funkcję `sqrt()`. Wyrażenie `sqrt(6.25)` to *wywołanie funkcji*, `sqrt()` jest *funkcją wywoływaną*, a funkcja zawierająca ten kod to *funkcja wywołująca* (rysunek 2.6).



Rysunek 2.6. Wywołanie funkcji

Wartość umieszczona w nawiasie (w tym wypadku jest to 6,25) to dane *przekazywane* do funkcji. Wartość tak właśnie przekazywana funkcji nazywana jest *parametrem* lub też *argumentem* (rysunek 2.7). Funkcja `sqrt()` wylicza wynik równy 2,5 i odsyła tę wartość do funkcji wywołującej; odsyłana wartość to *wartość zwracana*. O wartości tej można myśleć jako o wartości podstawianej w miejsce wywołania funkcji. Tak więc pokazany przykład powoduje przypisanie wartości zwracanej zmiennej `x`. Zatem, krótko mówiąc, parametr to wartość przekazywana funkcji, a wartość zwracana to wartość przekazywana z funkcji.



Rysunek 2.7. Składnia wywołania funkcji

W zasadzie to byłoby już wszystko, ale kompilator C++ przed wywołaniem funkcji musi znać jej parametry i typ wartości zwracanej. Czy funkcja zwraca liczbę całkowitą? znak? liczbę z częścią dziesiętną? wyrok? coś jeszcze innego? Jeśli zabraknie tej informacji, kompilator nie będzie potrafił zinterpretować zwracanej wartości. W C++ do określania tego typu informacji używa się prototypu funkcji.

Zapamiętaj

Program w C++ powinien zawierać prototypy wszystkich używanych w nim funkcji.

Prototyp funkcji pełni w stosunku do funkcji taką rolę jak deklaracja do zmiennej — informuje, jakie typy są używane. Na przykład w bibliotece C++ zdefiniowano funkcję `sqrt()` mającą jako parametr liczbę (z opcjonalną częścią ułamkową) oraz wartość zwracaną tego samego typu. W niektórych językach liczby z częścią ułamkową nazywane są *liczbami rzeczywistymi*, ale w języku C++ mówi się o typie **`double`** (szerzej typem tym zajmiemy się w rozdziale 3.). Prototyp funkcji `sqrt()` wygląda następująco:

```
double sqrt(double);      // prototyp funkcji
```

Pierwsze słowo **`double`** oznacza, że funkcja `sqrt()` zwraca wartość typu **`double`**. Słowo **`double`** umieszczone w nawiasach wskazuje, że funkcja `sqrt()` wymaga parametru typu **`double`**. Zatem pokazany prototyp opisuje funkcję, której można użyć następująco:

```
double x;                 // deklaracja x jako zmiennej typu double
x = sqrt(6.25);
```

Końcowy średnik w prototypie wskazuje na koniec instrukcji, tak że wiadomo, że mamy do czynienia z prototypem funkcji, a nie z jej nagłówkiem. Gdyby nie było tego średnika, kompilator zinterpretowałby pokazany wiersz jako nagłówek funkcji i spodziewałby się dalej treści funkcji.

Kiedy używamy funkcji `sqrt()` w programie, także musimy podać jej prototyp. Można to zrobić na jeden z dwóch sposobów:

- Wpisać do kodu źródłowego prototyp funkcji.
- Włączyć plik nagłówkowy *cmath* (lub *math.h* w starszych systemach), gdyż tam zawarty jest odpowiedni prototyp.

Drugi sposób jest lepszy, gdyż trudniej w jego wypadku o pomyłkę. Każda funkcja z biblioteki C++ ma swój prototyp w jednym lub nawet w wielu plikach nagłówkowych; wystarczy sprawdzić w opisie funkcji w podręczniku lub pomocy podręcznej. Na przykład opis funkcji `sqrt()` mówi, że należy użyć pliku nagłówkowego *cmath* (lub w starszych systemach *math.h*; ten ostatni dotyczy zarówno C++, jak i C).

Nie należy mylić prototypu funkcji z jej definicją. Jak widzieliśmy, prototyp opisuje jedynie interfejs funkcji, czyli mówi, jakie informacje są przekazywane do funkcji i jakie informacje ta funkcja zwraca. Definicja z kolei zawiera kod realizujący zadania funkcji, na przykład wyliczający pierwiastek kwadratowy z liczby. W językach C i C++ prototyp i definicja są rozdzielone w przypadku funkcji bibliotecznych — biblioteka zawiera skompilowany kod funkcji, zaś prototypy są w plikach nagłówkowych.

Prototyp funkcji należy umieścić przed pierwszym wywołaniem tej funkcji. Powszechną praktyką jest umieszczanie prototypu przed definicją funkcji `main()`. Na listingu 2.4 pokazano użycie funkcji bibliotecznej `sqrt()`. Prototyp tej funkcji jest włączany wraz z plikiem *cmath*.

Listing 2.4. sqrt.cpp

```
// sqrt.cpp — użycie funkcji sqrt()

#include <iostream>
#include <cmath>      // lub math.h

int main()
{
     using namespace std;

     double area;
     cout << "Podaj powierzchnię swojego mieszkania w metrach kwadratowych: ";
     cin >> area;
     double side;
     side = sqrt(area);
     cout << "Odpowiada to kwadratowi o boku " << side
          << " metrów." << endl;
     cout << "Niesamowite!" << endl;
     return 0;
}
```

Uwaga o zgodności ze standardem

W przypadku korzystania ze starszego kompilatora w listingu 2.4 konieczne może być zastąpienie wiersza `#include <cmath>` wierszem `#include <math.h>`.

Użycie funkcji bibliotecznych

Funkcje biblioteczne C++ są umieszczane w plikach bibliotecznych. Kiedy kompilator kompiluje program, musi odnaleźć w bibliotekach użyte funkcje. Różne kompilatory automatycznie przeszukują różne biblioteki. Jeśli po uruchomieniu programu z listingu 2.4 otrzymamy komunikat mówiący, że `_sqrt` nie zostało zdefiniowane, być może kompilator automatycznie nie sprawdza biblioteki matematycznej (kompilatory często dodają do nazw funkcji podkreślenie — kolejna rzecz, o której warto wiedzieć). Po otrzymaniu takiego komunikatu błędu należy sprawdzić w dokumentacji używanego kompilatora, jak odszukać właściwą bibliotekę. Przykładowo, w systemie Unix używa się opcji `-lm` (co jest skrótem od *library math*):

```
CC sqrt.C -lm
```

Kompilator GNU w systemie Unix działa podobnie:

```
g++ sqrt.C -lm
```

Samo włączenie pliku nagłówkowego *cmath* zapewnia dostępność prototypu, ale nie musi jeszcze spowodować, że kompilator przeszuka prawidłowy plik biblioteczny.

Oto przykładowy wynik wykonania powyższego programu:

```
Podaj powierzchnię swojego mieszkania w metrach kwadratowych:
153
Odpowiada to kwadratowi o boku 12.3693 metrów.
Niesamowite!
```

Funkcja `sqrt()` używa wartości typu **`double`**, więc w przykładzie używamy takiej właśnie zmiennej. Zauważmy, że zmienną typu **`double`** deklarujemy tak samo, jak deklarowaliśmy zmienną typu **`int`**:

```
nazwa-typu  nazwa-zmiennej;
```

Typ **`double`** pozwala zmiennym `area` i `side` zawierać część ułamkową, na przykład 153,0 lub 12,3693. Liczba pozornie całkowita, 153, jest przechowywana wraz z częścią ułamkową równą zeru, czyli ,0. Jak zobaczymy w rozdziale 3., typ **`double`** może pomieścić znacznie większy zakres wartości niż typ **`int`**.

Język C++ pozwala deklarować nowe zmienne w dowolnych miejscach w programie, więc w pliku *sqrt.cpp* zmiennej `side` nie deklarujemy na zapas, ale dopiero wtedy, kiedy będziemy jej potrzebować. C++ pozwala też w chwili deklaracji zmiennej od razu przypisać jej wartość, z czego skorzystaliśmy i tym razem:

```
double side = sqrt(area);
```

Więcej o tym procesie, nazywanym *inicjalizacją*, powiemy w rozdziale 3.

Zauważmy, że obiekt `cin` potrafi zamienić informacje ze strumienia wejściowego na typ **`double`**, a `cout` potrafi wstawić wartość typu **`double`** do strumienia wyjściowego. Jak wspomniano wcześniej, obiekty te są naprawdę inteligentne.

Odmiany funkcji

Niektóre funkcje wymagają więcej niż jednej danej. Funkcje takie mają wiele parametrów rozdzielanych przecinkami. Na przykład funkcja matematyczna `pow()` ma dwa parametry i zwraca wartość pierwszego parametru podniesionego do potęgi wyznaczonej przez drugi parametr. Oto jej prototyp:

```
double pow(double, double); // prototyp funkcji dwuparametrowej
```

Jeśli na przykład chcielibyśmy wyliczyć 5^8 (pięć do potęgi ósmej), moglibyśmy użyć tej funkcji następująco:

```
answer = pow(5.0, 8.0);        // wywołanie funkcji z listą parametrów
```

Są też funkcje bezparametrowe. Na przykład jedna z bibliotek języka C (związana w plikiem nagłówkowym *cstdlib* lub *stdlib.h*) ma funkcję `rand()`, która nie ma żadnych parametrów i która zwraca losową liczbę całkowitą. Jej prototyp wygląda następująco:

```
int rand(void);                // prototyp funkcji bezparametrowej
```

Słowo kluczowe **`void`** jawnie informuje, że funkcja nie ma parametrów. Jeśli pominiemy to słowo i zostawimy puste nawiasy, C++ zinterpretuje to jako niejawną deklarację braku parametrów. Moglibyśmy użyć tej funkcji następująco:

```
myGuess = rand();              // wywołanie funkcji bezparametrowej
```

Zauważmy, że w przeciwieństwie do niektórych języków programowania w C++ do wywołania funkcji trzeba dodawać nawiasy, nawet jeśli jest to funkcja bezparametrowa.

Istnieją też funkcje niemające wartości zwracanej. Załóżmy na przykład, że mamy napisać funkcję wyświetlającą liczbę w formie złotych i groszy. Jeśli funkcja ta otrzyma wartość 23,5, powinna wyświetlić ją jako 23 zł 50 gr. Funkcja ta pokazuje wartość na ekranie i nie zwraca żadnej wartości. Jej prototyp będzie wykorzystywał słowo kluczowe **`void`**:

```
void zlotowki(double);         // prototyp funkcji niezwracającej wartości
```

Funkcja `zlotowki()` nie zwraca żadnej wartości, nie może być częścią instrukcji przypisania ani innego wyrażenia. Występuje jako samodzielna instrukcja wywołania funkcji:

```
zlotowki(1234.56);             // wywołanie funkcji, bez wartości zwracanej
```

W niektórych językach programowania *funkcje* muszą zwracać wartość; jeśli coś nie zwraca wartości, jest *procedurą*. Jednak w językach C++ i C pojęcie funkcji obejmuje jedno i drugie.

Funkcje definiowane przez użytkownika

Standardowa biblioteka C zawiera ponad 140 funkcji predefiniowanych. Jeśli któraś z nich jest nam przydatna, to używajmy jej. Często jednak trzeba pisać własne funkcje, szczególnie podczas tworzenia klas. Poza tym pisanie własnych funkcji to ciekawe zajęcie, więc przyjrzyjmy się temu. Używaliśmy już kilku funkcji zdefiniowanych przez użytkownika; nazywały się `main()`.

Każdy program C++ musi mieć funkcję `main()` i funkcję tę definiuje użytkownik. Załóżmy, że chcemy dodać drugą funkcję użytkownika. Tak jak w przypadku funkcji bibliotecznych możemy wywołać funkcję użytkownika przez jej nazwę i tak jak w przypadku funkcji bibliotecznych przed wywołaniem funkcji musimy podać jej prototyp; zwykle robi się to ponad definicją funkcji `main()`. Tym razem jednak musimy także podać kod źródłowy nowej funkcji. Najprostszym sposobem jest umieszczenie tego kodu w tym samym pliku, po kodzie funkcji `main()`. Pokazano to na listingu 2.5.

Listing 2.5. ourfunc.cpp

```
// ourfunc.cpp — definiujemy własną funkcję
#include <iostream>
void simon(int);      // prototyp funkcji simon()

int main()
{
     using namespace std;
     simon(3);        // wywołanie funkcji simon()
     cout << "Podaj liczbę całkowitą: ";
     int count;
     cin >> count;
     simon(count);    // wywołaj ponownie
     cout << "Gotowe!" << endl;
     return 0;
}

void simon(int n)     // definicja funkcji simon()
{
     using namespace std;
     cout << "Simon prosi, abyś dotknął palców u stóp " << n << " razy." << endl;
}                     // funkcja typu void  nie ma instrukcji return
```

Funkcja `main()` wywołuje funkcję `simon()` dwukrotnie: raz z parametrem 3 i raz ze zmienną `count` jako parametrem. W międzyczasie użytkownik podaje liczbę całkowitą, która jest przypisywana zmiennej `count`. W tym przykładzie nie używamy znaku nowego wiersza po pokazaniu żądania wprowadzenia nowej wartości, więc dane wprowadzone przez użytkownika są w tym samym wierszu co prośba o ich wprowadzenie. Oto przykładowy wynik wykonania programu z listingu 2.5:

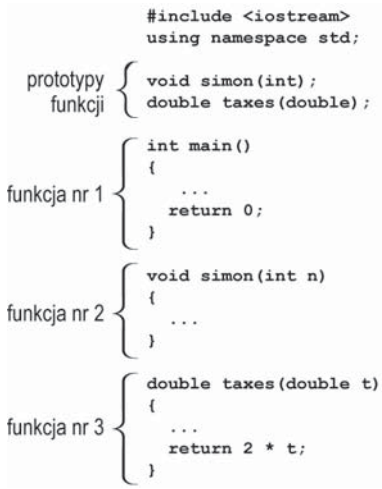
```
Simon prosi, abyś dotknął palców u stóp 3 razy.
Podaj liczbę całkowitą: 512
Simon prosi, abyś dotknął palców u stóp 512 razy.
Gotowe!
```

Postać funkcji

Definicja funkcji `simon()` z listingu 2.5 ma tę samą postać co definicja funkcji `main()`. Najpierw mamy nagłówek funkcji, a potem nawiasy klamrowe, w których zawarta jest treść funkcji. Zatem definicję funkcji możemy uogólnić następująco:

```
typ  nazwafunkcji(listaparametrów)
{
     instrukcje
}
```

Zauważmy, że kod źródłowy z definicją funkcji `simon()` znajduje się za zamykającym nawiasem klamrowym funkcji `main()`. Podobnie jak w C, a w przeciwieństwie do Pascala, w C++ nie można zagnieżdżać definicji jednej funkcji w innej. Każda definicja funkcji jest niezależna od pozostałych i wszystkie funkcje są sobie równe (rysunek 2.8).



Rysunek 2.8. Definicje funkcji znajdujące się w pliku

Nagłówki funkcji

Funkcja `simon()` z listingu 2.5 ma nagłówek:

```
void simon(int n)
```

Początkowe słowo **`void`** oznacza, że `simon()` nie zwraca żadnej wartości. Zatem wywołanie funkcji `simon()` nie wygeneruje żadnej liczby, którą można byłoby przypisać zmiennej w funkcji `main()`. Dlatego właśnie pierwsze wywołanie wygląda tak:

```
simon(3);              // prawidłowo dla funkcji typu void
```

Funkcja `simon()` nie zwraca żadnej wartości, więc nie można jej wywołać tak:

```
simple = simon(3);    // w przypadku funkcji typu void niedopuszczalne
```

Podawana w nawiasach wartość **`int`** `n` oznacza, że oczekujemy, że funkcja `simon()` pobierze jeden parametr typu **`int`**. `n` to nowa zmienna, której jest przypisywana wartość przekazana podczas wywołania funkcji. Zatem wywołanie:

```
simon(3);
```

powoduje przypisanie zmiennej `n` zdefiniowanej w nagłówku funkcji `simon()` wartości 3. Kiedy instrukcja z `cout` znajdująca się w treści funkcji użyje `n`, użyje tak naprawdę wartości przekazanej do funkcji. Dlatego właśnie wywołanie `simon(3)` powoduje wyświetlenie w wyniku trójki. Wywołanie `simon(count)` w pokazanym przykładzie powoduje wyświetlenie 512, gdyż taką wartość nadano zmiennej `count`. Podsumowując, nagłówek funkcji `simon()` informuje nas, że funkcja ta ma jeden parametr typu **`int`** i nie ma wartości zwracanej.

Spójrzmy teraz na nagłówek funkcji `main()`:

```
int main()
```

Początkowe **`int`** oznacza, że `main()` zwraca liczbę całkowitą. Puste nawiasy (lub opcjonalnie zawierające słowo kluczowe **`void`**) wskazują, że funkcja nie ma parametrów. Funkcje mające wartość zwracaną powinny mieć we wnętrzu słowo kluczowe **`return`** ze zwracaną wartością. Dlatego właśnie pod koniec tej funkcji mamy:

```
return 0;
```

Jest to logiczne — funkcja `main()` ma zwrócić wartość typu **`int`**, więc zwraca 0. Ale gdzie właściwie ta wartość jest zwrócona? Przecież w programach wywołanie `main()` nie występuje:

```
sprawdzmy = main();      // nie występuje w programach
```

Naszą funkcję `main()` wywołuje system operacyjny (na przykład Unix czy DOS), więc funkcja `main()` zwraca wartość nie do żadnej innej części programu, ale do systemu operacyjnego. Wiele systemów potrafi taką wartość wykorzystać. Na przykład skrypty powłoki Unix i pliki wsadowe DOS można pisać tak, aby uruchamiały programy i sprawdzały zwracane przez nie wartości, nazywane *kodami wyjścia*. Typowo przyjmuje się, że zerowy kod wyjścia oznacza prawidłowe działanie programu, zaś wartość niezerowa oznacza jakiś błąd. Można zatem zdefiniować program C++ tak, aby zwracał niezerową wartość, jeśli na przykład nie uda mu się otworzyć pliku. Można później zbudować skrypt powłoki lub plik wsadowy, który uruchomi program i podejmie środki zaradcze w przypadku pojawienia się błędu.

Słowa kluczowe

Słowa kluczowe to słownik języka komputerowego. W niniejszym rozdziale użyliśmy czterech słów kluczowych C++: **`int`**, **`void`**, **`return`** oraz **`double`**. Słowa te mają dla kompilatora C++ specjalne znaczenie, więc nie należy używać ich do niczego innego. Nie można zatem użyć **`return`** jako nazwy zmiennej, a **`double`** jako nazwy funkcji. Można jednak oczywiście używać nazw zawierających słowa kluczowe, jak `inteligencja` (gdzie zaszyte jest słowo **`int`**) czy `return_of_the_king`. W dodatku B podano pełną listę

słów kluczowych C++. Słowo `main` nie jest słowem kluczowym, gdyż nie jest to część języka, ale nazwa funkcji obowiązkowej. Można zatem zdefiniować zmienną o nazwie `main` (choć w pewnych, dość wyjątkowych, sytuacjach może to spowodować kłopoty, więc lepiej tak zmiennych nie nazywać). Tak samo inne nazwy funkcji i obiektów nie są słowami kluczowymi, ale użycie na przykład słowa `cout` do nazwania obiektu i zmiennej będzie powodem nieporozumień. Można użyć `cout` jako nazwy zmiennej w funkcji, która nie używa obiektu `cout`, ale w tej samej funkcji nie można będzie użyć już zwykłego `cout`.

Funkcje użytkownika zwracające wartość

Teraz pójdźmy krok dalej i napiszmy funkcję mającą instrukcję **`return`**. Funkcja `main()` pokazała nam już, jak to zrobić: podajemy w nagłówku typ zwracanej wartości, a w treści funkcji używamy instrukcji **`return`**. Tego typu konstrukcji możemy użyć do rozwiązania problemu wielu gości przebywających w Wielkiej Brytanii — wiele wag łazienkowych wyskalowanych jest tam w *kamieniach* zamiast w amerykańskich funtach czy międzynarodowych kilogramach. Jeden kamień to 14 funtów (niecałe 6,5 kg); program z listingu 2.6 zawiera funkcję zamieniającą kamienie na funty.

Listing 2.6. convert.cpp

```
// convert.cpp — zamiana kamieni na funty
#include <iostream>
int stonetolb(int);       // prototyp funkcji
int main()
{
     using namespace std;
     int stone;
     cout << "Podaj wagę w kamieniach: ";
     cin >> stone;
     int pounds = stonetolb(stone);
     cout << stone << " kamieni = ";
     cout << pounds << " funtów." << endl;
     return 0;
}

int stonetolb(int sts)
{
      return 14 * sts;
}
```

Oto przykład uruchomienia programu z listingu 2.6:

```
Podaj wagę w kamieniach: 14
14 kamieni = 196 funtów.
```

W funkcji `main()` program wykorzystuje obiekt `cin` do wczytania wartości zmiennej całkowitoliczbowej `stone`. Wartość ta jest przekazywana do funkcji `stonetolb()` jako parametr,

a w tej funkcji jest przypisywana zmiennej `sts`. Dzięki temu widać, że **`return`** nie musi ograniczać się do przekazywania gotowej liczby. W tym wypadku, kiedy mamy bardziej złożone wyrażenie, unikamy konieczności tworzenia nowej zmiennej na wynik, który zwrócimy. Program wylicza wartość wyrażenia (w tym przykładzie 196) i zwraca tę wartość. Jeśli ktoś nie chce mieć takich skrótów, może zrobić wszystko po kolei:

```
int stonetolb(int sts)
{
    int pounds = 14 * sts;
    return pounds;
}
```

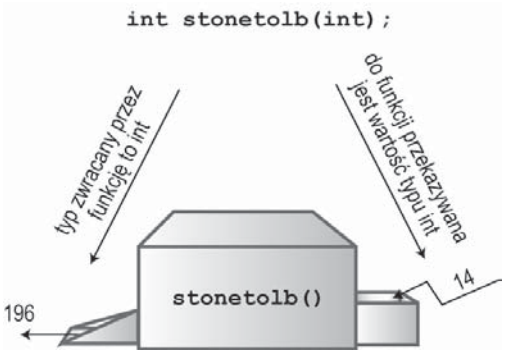
Obie wersje funkcji działają tak samo, tyle że druga jest nieco dłuższa.

Tak w ogóle funkcji zwracającej wartość można użyć wszędzie tam, gdzie można byłoby użyć stałej danego typu. Na przykład funkcja `stonetolb()` zwraca wartość typu **`int`**, więc można byłoby użyć jej następująco:

```
int aunt = stonetolb(20);
int aunts = aunt + stonetolb(10);
cout << "Fryderyka waży " << stonetolb(16) << " funtów." <<
endl;
```

W tym wypadku program wylicza wartość zwracaną, a następnie z wartości tej korzysta w wyrażeniach.

Jak widać w pokazanych przykładach, prototyp funkcji opisuje jej interfejs, czyli sposób interakcji funkcji z resztą programu. Lista parametrów pokazuje, jakie informacje przechodzą do funkcji; typ funkcji mówi, co funkcja zwraca. Czasami programiści opisują funkcje jako *czarne skrzynki* (termin pochodzi z elektroniki) w pełni opisane przez przepływ informacji do nich i z nich. Prototyp funkcji dokładnie opisuje taki właśnie sposób interpretacji (rysunek 2.9).



Rysunek 2.9. Prototyp funkcji i funkcja jako czarna skrzynka

Funkcja `stonetolb()` jest krótka i prosta, ale już zawiera pełną gamę cech funkcji:

- ma nagłówek i treść,
- przyjmuje parametr,
- zwraca wartość,
- wymaga prototypu.

Potraktujmy `stonetolb()` jako standardową postać projektu funkcji. Funkcjami dokładniej zajmiemy się w rozdziałach 7. i 8., ale materiał z niniejszego rozdziału powinien wyrobić Czytelnikowi pogląd, czym są funkcje i jak się ich używa w C++.

Dyrektywa using w programach z wieloma funkcjami

Zauważmy, że w kodzie z listingu 2.5 dyrektywę **`using`** wstawiono w obu funkcjach:

```
using namespace std;
```

Wynika to stąd, że każda z tych funkcji korzysta z obiektu `cout`, więc musi mieć dostęp do jego definicji w przestrzeni nazw `std`.

Przestrzeń nazw `std` można obu funkcjom z listingu 2.5 udostępnić jeszcze inaczej — należy umieścić dyrektywę poza obiema funkcjami, przed nimi:

```
// ourfunc1.cpp — zmiana położenia dyrektywy using
#include <iostream>
using namespace std;
void simon(int);      // prototyp funkcji simon()

int main()
{
     simon(3);        // wywołanie funkcji simon()
     cout << "Podaj liczbę całkowitą: ";
     int count;
     cin >> count;
     simon(count);    // wywołaj ponownie
     cout << "Gotowe!" << endl;
     return 0;
}

void simon(int n)     // definicja funkcji simon()
{
     cout << "Simon prosi, abyś dotknął palców u stóp " << n << " razy." << endl;
}
```

Obecnie przeważa filozofia polegająca na ograniczaniu dostępu do przestrzeni nazw `std` tylko do tych funkcji, które tego dostępu potrzebują. Na przykład w kodzie pokazanym na listingu 2.6 jedynie funkcja `main()` używa obiektu `cout`, więc funkcja `stonetolb()`

nie potrzebuje dostępu do przestrzeni `std`. Tak więc dyrektywa **`using`** została umieszczona tylko w funkcji `main()`.

Podsumowując, elementy przestrzeni nazw `std` możemy udostępniać w programie na kilka sposobów:

- Dyrektywę:
  `using` `std` **`namespace;`**
  można umieścić w pliku przed definicjami funkcji, udostępniając całą zawartość przestrzeni nazw `std` wszystkim funkcjom z pliku.
- Dyrektywę:
  `using` `std` **`namespace;`**
  można umieścić w definicji konkretnej funkcji, udostępniając przestrzeń nazw tej właśnie funkcji.
- Zamiast korzystać z dyrektywy:
  `using` `std` **`namespace;`**
  można zapisać w odpowiedniej definicji funkcji:
  `using` `std::cout;`
  udostępniając tym samym konkretny element przestrzeni nazw `std` — na przykład `cout`.
- Można w końcu całkowicie pominąć dyrektywy **`using`**, a odwołując się do jakichkolwiek elementów przestrzeni `std`, poprzedzać je przedrostkiem `std::`:

```
std::cout << „Używam cout i endl z przestrzeni nazw std” <<
std::endl;
```

Z codziennej praktyki: konwencje nazewnicze

Programiści C++ cieszą się błogosławieństwem (a może ciąży na nich przekleństwo) prawie dowolnego nazywania funkcji, klas i zmiennych. Programiści bardzo się różnią w swoich opiniach co do optymalnego stylu, co często powoduje święte wojny na rozmaitych forach. Wychodząc z tego samego podstawowego pomysłu na nazwę funkcji, programista może wybierać z kilku możliwości:

```
MojaFunkcja()
mojafunkcja()
mojaFunkcja()
moja_funkcja()
moja_funk()
```

Wybór jednej z tych opcji zależy od zespołu programistów, cech charakterystycznych używanych technik i bibliotek, preferencji i przyzwyczajeń poszczególnych programistów. Wystarczy zapewnić, że przyjęta konwencja będzie zgodna z zasadami języka C++ — reszta to kwestia gustu.

Przyzwolenie ze strony języka to jedno, ale trzeba pamiętać, że konsekwentne trzymanie się jednej konwencji nazewniczej pomaga w pracy. Precyzyjna, dająca się rozpoznać konwencja nazewnicza znamionuje dobrego informatyka i pomaga mu w codziennej pracy.

Podsumowanie

Program w języku C++ składa się z jednego lub wielu modułów nazywanych funkcjami. Programy zaczynają swoje działanie od wykonania funkcji `main()` (nazwa zapisywana małymi literami), więc funkcja taka zawsze powinna istnieć. Funkcja składa się z nagłówka i treści. Nagłówek funkcji mówi, czy i ewentualnie jakiego typu wartość funkcja zwraca oraz jakich parametrów oczekuje. Treść funkcji to ciąg instrukcji C++ zamkniętych w nawiasy klamrowe (`{}`).

Instrukcje języka C++ można podzielić na następujące grupy:

- **Deklaracja** — opisuje nazwę i typ zmiennej używanej w funkcji.
- **Przypisanie** — wykorzystuje operator przypisania (=) do przypisania wartości jakiejś zmiennej.
- **Komunikat** — instrukcja ta wysyła komunikat do obiektu, uruchamiając pewną akcję.
- **Wywołanie funkcji** — uruchamia funkcję. Kiedy wywołana funkcja kończy swoje działanie, program wraca do instrukcji znajdującej się zaraz za wywołaniem.
- **Prototyp funkcji** — deklaruje typ zwracany przez funkcję oraz liczbę i typy parametrów funkcji.
- **Instrukcja powrotu** — wysyła wartość z wywołanej funkcji z powrotem do funkcji wywołującej.

Klasa to zdefiniowany przez użytkownika opis typu danych. Specyfikacja taka opisuje sposób przechowywania informacji oraz mówi, jakie działania mogą być na tych danych wykonywane. Obiekt to byt stworzony zgodnie z opisem klasy, tak jak zmienna jest bytem tworzonym zgodnie z opisem typu danych.

Język C++ ma dwa predefiniowane obiekty wejścia i wyjścia, `cin` i `cout`. Obiekty ten należą do klas `istream` i `ostream` zdefiniowanych w pliku *iostream*. Klasy te interpretują strumienie wejściowy i wyjściowy jako ciągi znaków. Operator wstawiania, <<, zdefiniowany w klasie `ostream`, pozwala wstawiać dane do strumienia wynikowego, zaś operator pobrania >> zdefiniowany w klasie `istream` — pozwala pobierać dane ze strumienia wejściowego. Obiekty `cin` i `cout` są inteligentne w tym znaczeniu, że potrafią automatycznie zamieniać informacje z jednego typu na inny, zależnie od kontekstu programu.

Język C++ może intensywnie wykorzystywać funkcje z biblioteki języka C. Aby takiej funkcji użyć, trzeba najpierw dołączyć plik nagłówkowy zawierający prototyp funkcji.

Teraz wiemy już co nieco o całym programie w C++, możemy więc przejść do następnego rozdziału i jedne zagadnienia uszczegółowić, o innych dopiero się nauczyć.

Pytania sprawdzające

Odpowiedzi na te pytania znaleźć można znaleźć w dodatku J.

1. Jak nazywają się moduły, z których zbudowane są programy w C++?
2. Co oznacza poniższa dyrektywa preprocesora?

```
#include <iostream>
```

3. Co robi poniższa instrukcja?

```
using namespace std;
```

4. Jakich instrukcji trzeba użyć, aby napisać „Hello, world" i przejść do nowego wiersza?
5. Za pomocą jakiej instrukcji można stworzyć całkowitoliczbową zmienną `sery`?
6. Jaka instrukcja spowoduje przypisanie zmiennej `sery` wartości 32?
7. Jaka instrukcja spowoduje odczytanie z klawiatury wartości zmiennej `sery`?
8. Jakimi instrukcjami można wypisać zdanie „Mamy X odmian sera", gdzie X zastępowane jest przez aktualną wartość zmiennej `sery`?
9. Co poniższe prototypy mówią o reprezentowanych funkcjach?

```
int froop(double t);
void rattle(int n);
int prune(void);
```

10. Kiedy w definicji funkcji trzeba użyć słowa kluczowego **`return`**?

Ćwiczenia programistyczne

1. Napisz program C++ pokazujący nazwisko i adres autora.
2. Napisz program w C++ proszący o podanie odległości w milach morskich i zamieniający ją na metry (jedna mila morska to 1852 metry).
3. Napisz program w C++ wykorzystujący trzy funkcje (jedną z nich będzie `main()`), dający następujące wyniki:

```
Entliczek pentliczek
Entliczek pentliczek
Czerwony stoliczek
Czerwony stoliczek
```

   Pierwsze dwa wiersze ma wygenerować pierwsza funkcja (wywołana dwukrotnie), następne dwa druga funkcja także wywołana dwukrotnie.

4. Napisz program, w którym w funkcji `main()` zostanie wywołana funkcja użytkownika otrzymująca temperaturę w stopniach Celsjusza i zwracająca odpowiadającą jej temperaturę w skali Fahrenheita. Program ma poprosić użytkownika o podanie wartości z klawiatury i pokazać wynik jak poniżej:

```
Podaj temperaturę w stopniach Celsjusza: 20
20 stopnie Celsjusza to 68 stopnie Farhrenheita.
```

   Zamianę temperatury przeprowadź zgodnie ze wzorem:

   Fahrenheit = 1,8 * stopnie Celsjusza + 32,0

5. Napisz program, w którym funkcja `main()` wywoła funkcję użytkownika pobierającą jako parametr odległość w latach świetlnych i zwracającą odległość w jednostkach astronomicz-

nych. Program powinien prosić o liczbę lat świetlnych, odczytywać ją i przeliczać, a wynik pokazywać. Przykładowe wywołanie pokazano poniżej:

```
Podaj liczbę lat świetlnych: 4.2
4.2 lat świetlnych = 265608 jednostek astronomicznych.
```

Jednostka astronomiczna to średnia odległość od Ziemi do Słońca (około 150 000 000 km), a rok świetlny to odległość, jaką światło pokona w ciągu roku (około 10 bilionów kilometrów). Najbliższa Słońcu gwiazda jest od nas oddalona o 4,2 roku świetlnego. Współczynnik przeliczenia niech będzie liczbą typu **double** (jak w listingu 2.4).

Jeden rok świetlny = 63 240 jednostek astronomicznych

6. Napisz program proszący użytkownika o podanie liczby godzin i minut. Funkcja `main()` ma przekazać obie te wartości do funkcji typu **void**, która je wyświetli w formacie jak poniżej:

```
Podaj liczbę godzin: 9
Podaj liczbę minut: 28
Czas: 9:28
```