Andrzej Stasiewicz

C++

Ćwiczenia praktyczne

helion.pl

Helion

Wszystkim, którzy programują

Spis treści

Wprowadzenie

Dlaczego język C++ jest tak ważny?

Język C++ jest obecnie najważniejszym, najczęściej stosowanym narzędziem do programowania komputerów. Zastanówmy się przez chwilę, dlaczego tak jest. Dlaczego — rozpoczynając naukę programowania — warto stawiać na język C++?

Po pierwsze, jest to język wręcz ubogi, ascetycznie zdefiniowany, o niewielkiej liczbie stosunkowo łatwych do opanowania słów kluczowych, niemal w całości przejętych z kultowego języka C. Owa ubogość jest jednym z głównych źródeł siły języka. Za pomocą kilkunastu doskonale przemyślanych poleceń — oczywiście przy wykorzystaniu odpowiednich bibliotek gotowych algorytmów — z komputerem można zrobić dosłownie wszystko.

Ubogość leksykalna języka powoduje, że programy źródłowe są stosunkowo łatwo przenoszone między różnymi platformami.

Ubogi język C++ dysponuje obszernymi, doskonałymi bibliotekami na wszelkie okazje — internet, grafika, nauka, inżynieria... Jest zatem narzędziem niezwykle uniwersalnym, znajdującym zastosowanie w każdej dziedzinie, w której wykorzystujemy komputery.

Istnieją języki specjalistyczne, najlepiej pasujące do konkretnych potrzeb. Jeśli jednak szukają Państwo jednego języka, który w przyszłości umożliwi programowanie bardzo szerokiego spektrum zagadnień, nauka języka C++ jest właściwym krokiem. Wiele języków (Java, JavaScript, PHP) wyrosło z języka C++. Nawet elementarna znajomość składni języka C++ da Państwu bardzo dobrą pozycję wyjściową do poznawania tych języków.

Mam nadzieję, że część właśnie dokonujących wyboru języka Czytelników przekona odmienny argument — intensywne studiowanie języka C++ jest wielką przygodą intelektualną. Jest to naprawdę piękny język. Ktoś, kto zna język C++, należy do najbardziej elitarnego grona programistów.

Być może spotkali się Państwo ze stwierdzeniami, że język C++ jest nadmiernie rozbudowany, zawiera mechanizmy, bez których można się z powodzeniem obejść. Za takie często uważa się np. modyfikowanie działania operatorów — oto powszechnie znany

operator dodawania nagle może zacząć robić coś zupełnie innego — np. dopisywać daną do pliku dyskowego. Otóż stwierdzenia te są całkowicie nieuprawnione. Wiele mechanizmów, które jeszcze kilka lat temu wydawały się nadmiarowe, znalazło zastosowanie w tak zwanej **bibliotece standardowej** języka C++ — rozszerzeniu, które zmieniło obraz współczesnego programowania.

Współczesny język C++ jest doskonale spójnym i niebywale silnym narzędziem programowania.

Co da Czytelnikowi ta książka?

Książka niniejsza jest elementarzem języka C++. Dedykujemy ją początkującemu programiście.

Skoncentrujemy się wyłącznie na najprostszych i zarazem najważniejszych zagadnieniach. Najpierw nauczymy się operować danymi i algorytmami, potem wprowadzimy do gry klasy i ich konkretne egzemplarze, zwane obiektami. Wszystkie zagadnienia zostaną skomentowane skrajnie uproszczonymi przykładami, bezpośrednio ilustrującymi akurat opisywane problemy.

Niektóre zagadnienia współczesnego języka C++ pozostawimy jednak nietknięte, dbając przede wszystkim o to, by Czytelnik wyniósł z lektury spójny i dobrze ugruntowany obraz, stanowiący pewną podstawę dalszych studiów.

Po przeczytaniu książki i przede wszystkim przerobieniu owej setki mikroprzykładów Czytelnik będzie gotów zmierzyć się z literaturą dotyczącą realnego programowania, nastawionego na uzyskanie nie ćwiczeniowego skrawka algorytmu, a prawdziwej, współczesnej aplikacji. Podręczniki do takich narzędzi, jak C++ Builder czy Visual C++, zakładają pewną znajomość języka C++. Nie wyjaśnia się tam zasad deklarowania zmiennych czy konstrukcji elementarnego algorytmu. Niezbędna jest wiedza zawarta w tej książce.

Co będzie potrzebne do korzystania z książki?

Książkę należy czytać, jednocześnie wprowadzając do komputera liczne przytaczane tutaj przykłady. Jest do tego niezbędny kompilator współczesnego języka C++.

Zdajemy sobie sprawę, że praca nad naszymi przykładami jest pewnym etapem wstępnym w Państwa karierze programistycznej. Ci, którzy przejdą przez ten wstępny etap, szybko rozejrzą się za kompletnym środowiskiem programistycznym w rodzaju C++ Builder czy Visual C++. Z tego powodu dobrze byłoby od samego początku pracować w tych narzędziach, choćby tylko miało to oznaczać oswajanie się z ich wyglądem.

Z drugiej jednak strony wystarczy nam najprostszy kompilator, zupełnie niezorientowany na tworzenie skomplikowanych, okienkowych interfejsów. Nasze mikroprogramy w żadnym wypadku nie będą doskonalone pod względem funkcjonalności, wyglądu, walorów użytkowych czy komercyjnych. Będą to programy czysto szkoleniowe, do pisania których nie musimy oddawać gigabajta pamięci dyskowej pod współczesne środowisko programistyczne.

Kompilator musi pracować w tak zwanym trybie konsoli — dawno temu powiedzielibyśmy, że byłby to kompilator *DOS-owy*. Jest to tryb archaiczny, a programista konsolowy raczej nie znajdzie pracy na współczesnym rynku. Tryb konsoli ma jednak ciągle pewną bezwzględną zaletę — zupełnie oddala wszelkie problemy związane z konstrukcją interfejsu. Jest idealny do ćwiczeń poszczególnych elementów języka.

Przestarzałość konsoli wcale nie oznacza, że zadowolimy się przestarzałym kompilatorem w rodzaju słynnego Borland C++ wersja 3.1. Wręcz przeciwnie — kompilator musi być ekstremalnie nowoczesny, bo od razu sięgniemy na górną półkę współczesnego języka C++ — do **biblioteki standardowej**. Okazuje się bowiem, że w tym przypadku nowoczesność oznacza zarazem łatwość i prostotę.

Wszystkie przykłady przygotowywałem, uruchomiłem i sprawdziłem w bezpłatnym narzędziu o nazwie DEV-C++. Kompilator ten — a raczej całe środowisko programistyczne z kompilatorem, edytorem i wieloma innymi narzędziami — z łatwością ściągniemy z internetu. Wystarczy w wyszukiwarce wpisać jego nazwę. Nie bez znaczenia jest też fakt, że środowisko DEV-C++ jest spolszczone oraz że zajmuje zadziwiająco mało miejsca na dysku.

Podkreślę raz jeszcze, że równie dobrym rozwiązaniem będzie testowanie naszych mikroprogramów w Builderze czy Visualu, tyle że od razu przestawionych z trybu okienkowego na tryb konsoli.

Jak uczyć się języka z tej książki?

Książkę — która jest serią praktycznych ćwiczeń — należy czytać przy włączonym komputerze. Każdy kolejny przykład, szczególnie, gdy nie jest absolutnie zrozumiały, powinien być natychmiast wprowadzony do komputera, skompilowany, uruchomiony i dokładnie przeanalizowany. W każdym miejscu każdego przerabianego programu należy dokonywać wszelkich możliwych zmian i modyfikacji. Nic złego się nie stanie, jeśli wskutek tych zmian ćwiczebny programik nagle przestanie działać.

Zadbam o to, by każdy przykład był tak prosty, jak to tylko możliwe, ale przy tym dokładnie ilustrował akurat omawiany element języka. Będą to programy wręcz banalne, ale o określonej wartości merytorycznej. Postaram się nie popełnić często powtarzanego w takich elementarzach błędu, gdy sam przykład jest na tyle złożony, że już nie bardzo wiadomo, o co autorowi w nim chodziło.

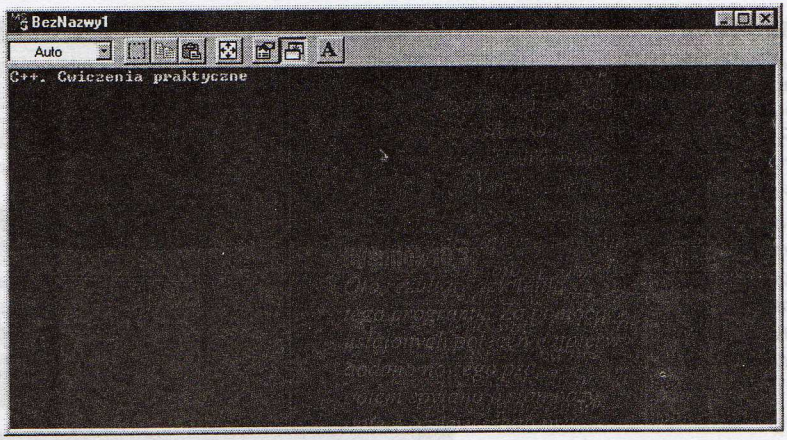
Cieszyłbym się, gdyby konieczność uruchomienia kolejnego przykładu była wyczekiwaną chwilą odprężenia, zabawą w programowanie, eksperymentowaniem z językiem i z komputerem.

Rozdział 1.

Nasz programistyczny warsztat

Naukę języka C++ zaczniemy od zainstalowania odpowiednio nowoczesnego środowiska programistycznego, dysponującego trybem pracy konsolowej. Mój wybór padł na bezpłatne środowisko o nazwie DEV-C++, które z łatwością znajdziemy w internecie lub na krążkach dołączanych do miesięczników informatycznych.

Podkreślę, że DEV-C++ nie jest bezwzględną koniecznością — może nawet lepiej pracować w doskonałych środowiskach typu Builder czy Visual C++. Środowiska te — teraz ustawione na ubogi tryb pracy konsolowej — w niedalekiej przyszłości posłużą Państwu do opracowywania pięknych aplikacji.

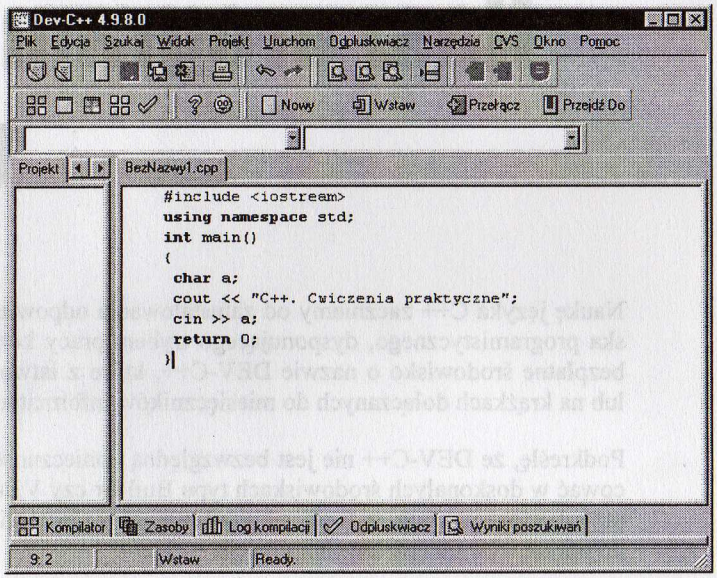


Rysunek 1.1. *Oto tak zwana konsola — charakterystyczny czarny ekran, przystosowany do prezentowania informacji znakowej. Ubogość możliwych do zaprezentowania za pomocą konsoli środków wyrazu skazała ją na śmierć. Jednak konsola idealnie nadaje się do nauki programowania*

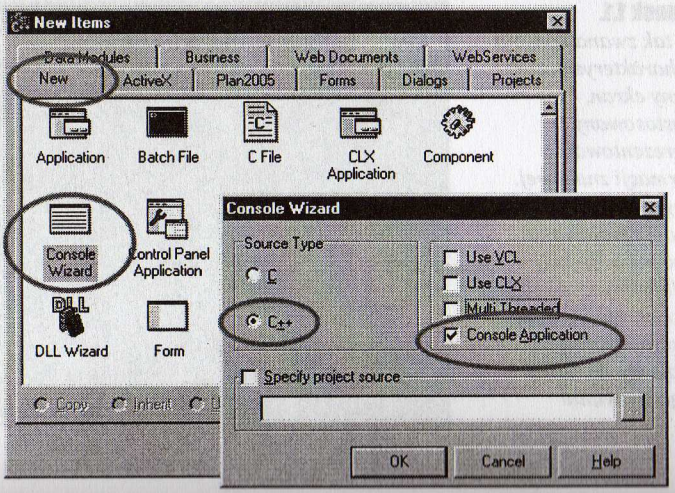
Do naszej pracy nie nadają się kilkuletnie środowiska — takie jak np. wciąż niezwykle popularny Borland C++ wersja 3.x. W ciągu ostatnich kilku lat język C++ bardzo się zmienił. Jeszcze bardziej zmieniły się biblioteki, uzupełniające każde wydanie kompilatora C++. Wiele kompilatorów należy po prostu wyrzucić, co znajduje swoje odbicie np. na oficjalnych stronach Borlanda, który ogłasza, że przestaje się interesować swoimi niedawno sztandarowymi produktami i kto chce, może je użytkować bez żadnych zobowiązań.

Zanim przejdziemy do następnego rozdziału, muszą Państwo zdobyć swój kompilator i zainstalować go. Czynności instalacyjne są zdecydowanie najłatwiejsze do wykonania — choćby z racji objętości plików — w środowisku programistycznym DEV, pokazanym na rysunku 1.2.

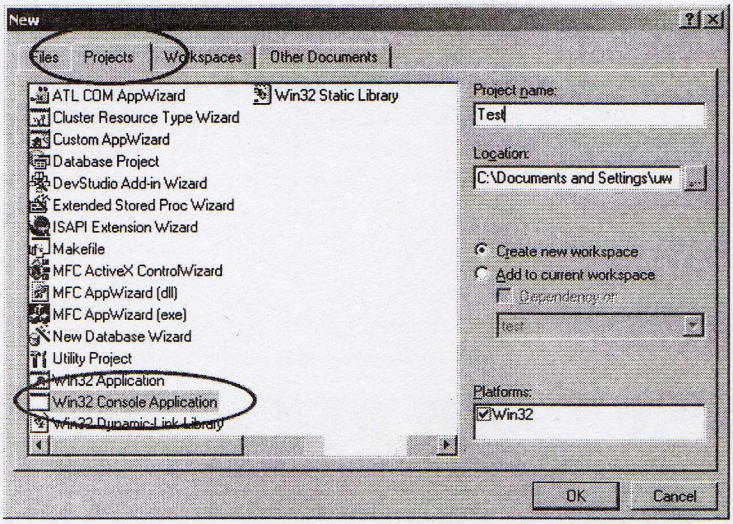
Rysunek 1.2. *Wszystkie przykłady i ćwiczenia przygotowałem w DEV-C++. DEV jest dostępne bezpłatnie na bardzo wielu serwerach — wystarczy w internetowej wyszukiwarce wpisać nazwę tego niezmiernie ciekawego oprogramowania*



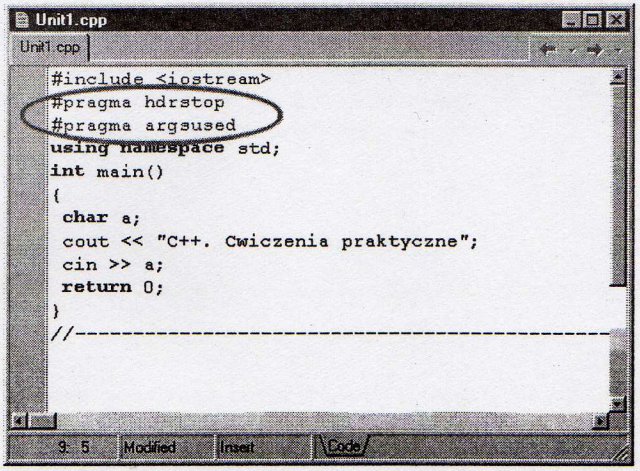
Rysunek 1.3. *Aplikacje konsolowe możemy też pisać w bardzo rozbudowanych środowiskach, takich jak C++ Builder i Visual C++. Widoczne tutaj środowisko Borlanda najpierw trzeba przestawić na tryb pracy konsolowej, bowiem zaraz po włączeniu jest ono przygotowane do pracy nad aplikacją okienkową*



Rysunek 1.4.
Podobne czynności przestawienia środowiska na tryb pracy konsolowej trzeba też wykonać w Visual C++



Rysunek 1.5.
W środowisku DEV, pokazanym na rysunku 1.2, można dostrzec czysty kod bardzo prostego programu, co za chwilę będziemy obszernie omawiać. W potężnych środowiskach BCB i Visual — zorientowanych na tworzenie złożonych aplikacji okienkowych — zauważymy kilka dodatkowych linii technicznego kodu. Nie wnikając w to, co te linie oznaczają, musimy uważać, by ich nie uszkodzić



Rozdział 2.

Nasz pierwszy program

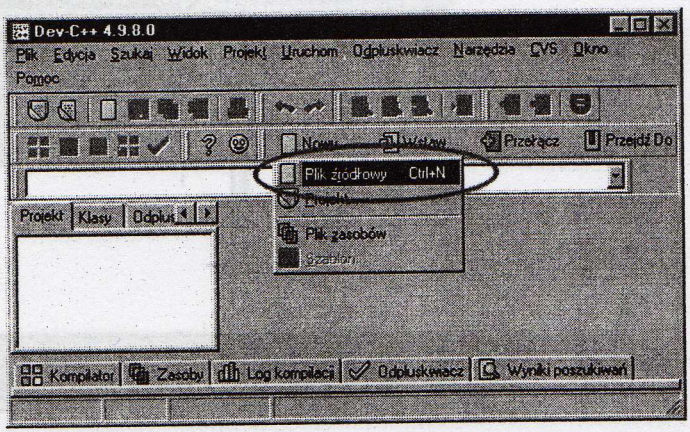
Właśnie ukończyliśmy instalację środowiska C++, prawdopodobnie instalator zalecił standardowe w takich sytuacjach wykonanie restartu maszyny, wreszcie zorientowaliśmy kompilator na pracę w trybie konsoli (porównaj rysunek 1.1 i następne). Pisząc program, najlepiej sprawdzimy, czy środowisko jest właściwie przygotowane do pracy.

Czy to działa?

Ćwiczenie 2.1.

Wykonaj odpowiednie czynności techniczne, prowadzące do uzyskania nowego programu i napisz program wyprowadzający na ekran Twoje imię i nazwisko.

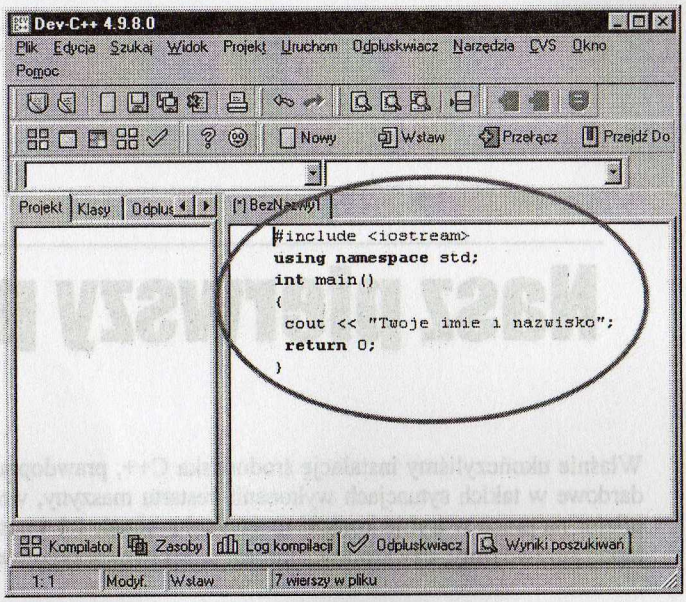
1. Z menu *Nowy* wybierz polecenie *Nowy plik źródłowy* (*Ctrl+N*) (rysunek 2.1). Otrzymasz czyste okienko edycyjne, gotowe do przyjęcia tekstu programu.



Rysunek 2.1. *Pracę nad nowym programem rozpoczynamy od wydania polecenia Nowy plik źródłowy*

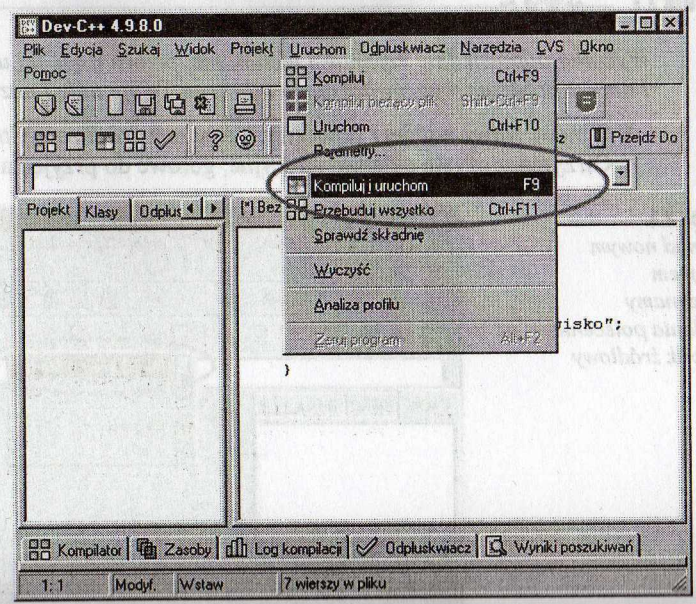
2. W okienku edycyjnym wpisz treść mikroprogramu według rysunku 2.2. Wkrótce lepiej poznamy jego strukturę, na razie postarajmy się przytoczony tutaj listing bezbłędnie przenieść do edytora.

Rysunek 2.2. *Treść programu, wprowadzona do okienka edycyjnego*

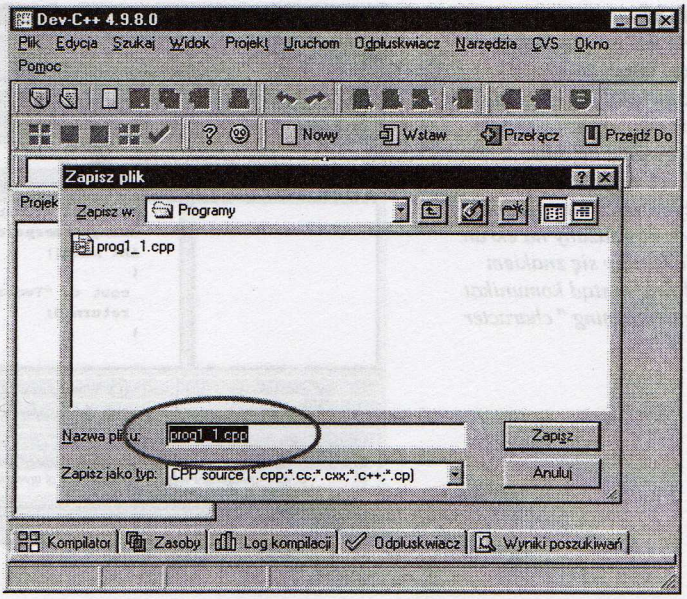


3. Po wpisaniu treści programu skompiluj go, czyli przekształć z języka C++ na język procesora, i uruchom. Służą do tego operacje umieszczone w menu *Uruchom* (rysunek 2.3). Zwróć uwagę, że środowisko DEV zapisuje program na dysku (rysunek 2.4) przed kompilacją (rysunek 2.5).

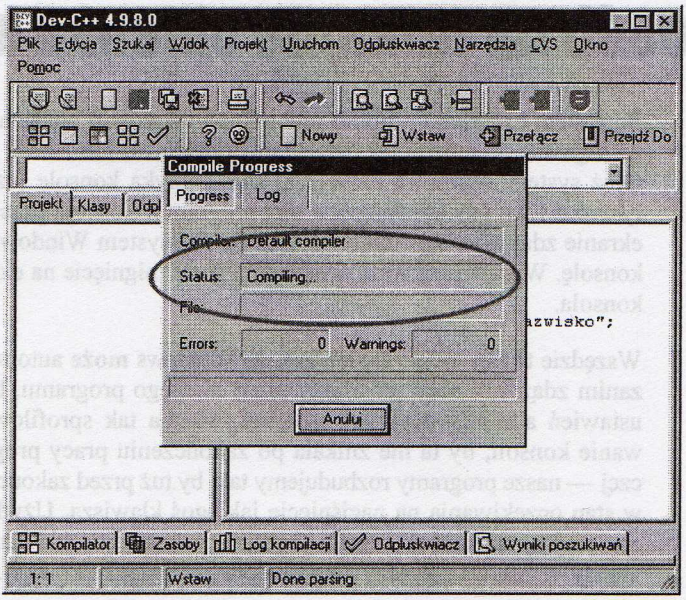
Rysunek 2.3. *Kompilacja i uruchomienie gotowego programu*



Rysunek 2.4. *Środowisko DEV (ale nie Builder czy Visual) przed kompilacją zapisuje program na dysku. Przy pierwszym zapisie padnie standardowe pytanie o nazwę pliku i jego dyskową lokalizację*

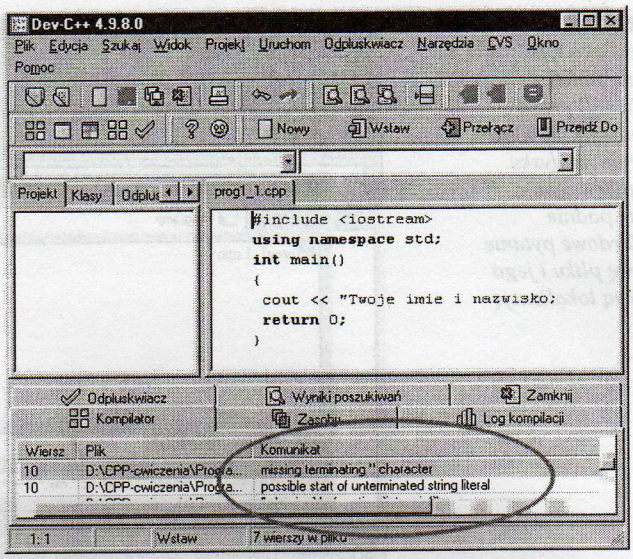


Rysunek 2.5. *Okienko kompilatora w środowisku DEV*



4. Jeśli program się nie uruchamia, sprawdź, jakie popełniłeś błędy, analizując informacje w okienku komunikatów (rysunek 2.6), popraw program i ponownie go skompiluj oraz uruchom.

Rysunek 2.6.
Mylić się jest rzeczą ludzką — niemal każdy program będzie zawierał jakieś błędy. Diagnostyka, czyli sygnalizowanie błędów jest jedną z najważniejszych cech użytkowych kompilatora. Tutaj wyprowadzany na ekran tekst nie kończy się znakiem cudzysłowu — stąd komunikat missing terminating " character



Sposób na znikanie okienka konsoli

Poprzedni program — mimo braku komunikatów o błędach, na które koniecznie trzeba zareagować — może i tak działać nieprawidłowo i to z dość nieoczekiwanych powodów. Otóż system Windows zazwyczaj sam zamyka konsolę (czarne okienko, pokazane na rysunku 1.1), gdy działający w niej program zakończył pracę. Nasz program wypisał na ekranie zdanie tekstu i zakończył pracę, zaś system Windows mógł ostatecznie zamknąć konsolę. Wprawne oko może nawet dostrzec mignięcie na ekranie czegoś czarnego — to konsola.

Wszędzie tu używałem stwierdzeń, że Windows **może** automatycznie zniszczyć konsolę, zanim zdążymy obejrzeć efekty pracy swojego programu, bowiem wszystko zależy od ustawień administracyjnych systemu. Można tak sprofilować systemowe oprogramowanie konsoli, by ta nie znikała po zakończeniu pracy programu. Zrobimy jednak inaczej — *nasze programy rozbudujemy tak, by tuż przed zakończeniem swojej pracy weszły* w stan oczekiwania na naciśnięcie jakiegoś klawisza. Użytkownik programu uruchomi go, *obejrzy w konsoli wyniki jego pracy i naciśnie klawisz, kończąc życie programu.* Wtedy Windows zniszczy niepotrzebne już okienko konsoli.

Ćwiczenie 2.2.

Uzupełnij program o algorytm przejścia w stan oczekiwania na naciśnięcie klawisza. Jest to prosty sposób na powstrzymanie Windows przed zbyt szybkim zamykaniem konsoli:

```
#include <iostream>
using namespace std;
int main()
{
 cout << "Twoje imie i nazwisko";
```

```
  char c;
  cin >> c;
  return 0;
}
```

W programie przybyła instrukcja wczytywania znaku do zmiennej o nazwie `c`. Znak ten nas nie interesuje — jak widać, nic z nim w programie nie robimy, nie przetwarzamy go w żaden sposób. Potrzebne jest tylko samo oczekiwanie programu na wprowadzenie znaku. Ten program zakończy swoją pracę dopiero wtedy, gdy użytkownik naciśnie klawisz i dodatkowo zakończy wprowadzanie danych naciśnięciem klawisza *Enter*.

Podsumowanie

W rozdziale tym napisaliśmy dwa programy, ale niekoniecznie musimy je rozumieć. Programy te były przede wszystkim testem na poprawność instalacji środowiska programistycznego i naszą umiejętność przeprowadzenia procesu kompilacji i uruchamiania procesów konsolowych. Mogli Państwo nawet bez większego zrozumienia, znak po znaku przepisać podawaną tutaj treść programów. W dalszej części książki — już pewni prawidłowej pracy używanych narzędzi — będziemy zwracać coraz baczniejszą uwagę na elementy języka C++.

Rozdział 3.

Pliki źródłowe w języku C++

Pliki jako nośniki programów

Treść programu komputerowego zazwyczaj umieszczamy w plikach dyskowych. Zawartość tych plików będzie odczytywana przez kompilator języka C++ i tłumaczona na ciąg binarnych poleceń dla procesora komputera.

Programowanie nie zawsze jest równoznaczne z zapisywaniem czegoś w plikach — np. w przemyśle spotykamy się z sytuacjami wprowadzania programu do komputera za pomocą odpowiedniego ustawiania mikroprzełączników. Kiedyś powszechne było umieszczanie programu na odpowiedniej ilości dziurkowanych kart.

Przygotowywanie programu w formie zapisów umieszczanych w plikach jest bardzo wygodne, tanie i uniwersalne. Zawsze można taki program odtworzyć, poprawić, zlecić jego wykonanie, zarchiwizować na całe lata.

Zapis programu dla komputera zazwyczaj ma strukturę zwykłego tekstu — mamy zatem do czynienia z plikami tekstowymi. Rzeczywiście, program napisany w zdecydowanej większości znanych języków daje się otworzyć i przeczytać za pomocą zwykłego Notatnika. Jest to dodatkowe uproszczenie sposobu kodowania i przechowywania współczesnych programów.

Skoro pliki źródłowe są zwyczajnymi plikami tekstowymi, do programowania wystarczy najzwyklejszy edytor tekstowy — np. popularny Notatnik. Jednak większość współczesnych środowisk programistycznych udostępnia programiście własne, wbudowane edytory. Są to edytory tekstowe, ale „znające" składnię języka i na przykład odpowiednio kolorujące niektóre frazy języka. Praca nad programem w takim edytorze jest prawdziwą przyjemnością! Pamiętajmy jednak, że poradzilibyśmy sobie także dysponując zwykłym Notatnikiem.

Nośniki programów w C++

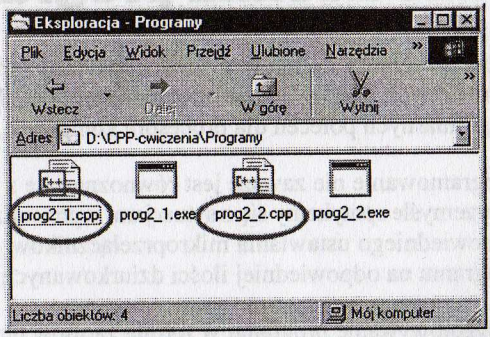
W języku C++ przyjęto powszechnie konwencję, że głównym nośnikiem algorytmów jest plik o rozszerzeniu *cpp*, czyli np. plik o nazwie *test.cpp*. Spotkamy się także z plikami o rozszerzeniu *h*, czyli np. o nazwie *test.h*, które są nośnikami nie tyle algorytmów, ile ich zapowiedzi lub ściślej — deklaracji. Wiadomo, skąd pochodzi nazwa *cpp*, natomiast literka *h* w nazwie pliku z deklaracjami wzięła się od słowa *header* — nagłówek.

Swoje programy będziemy spisywać w pliku o nazwie np. *test.cpp* lub *przyklad.cpp* lub *cokolwiek.cpp*. Plik ten powinien mieć strukturę zwykłego pliku tekstowego i mógłby być przygotowany w dowolnym edytorze, potem odczytany przez kompilator języka C++, skompilowany i uruchomiony.

Ćwiczenie 3.1.

Pliki źródłowe naszych programów:

1. Po wykonaniu ćwiczeń z poprzedniego rozdziału na dysku Twojego komputera powinny pojawić się ich pliki źródłowe. Odszukaj katalog, w którym środowisko DEV zapisało te pliki (rysunek 3.1).



Rysunek 3.1. *Oto rzut oka na katalog roboczy — widzimy tutaj dwa pliki źródłowe (są to programy napisane w poprzednim rozdziale) i utworzone w wyniku ich kompilacji dwa finalne pliki* exe, *nadające się do uruchamiania w systemie Windows*

2. Spróbuj otworzyć swoje pliki źródłowe za pomocą zwykłego Notatnika.

Najprostsze programy w całości spisuje się w pliku *cpp*. Jeśli zachodzi konieczność zadeklarowania czegokolwiek, odpowiednie frazy umieszcza się raczej w górnej części tego pliku (gdzieś przed zasadniczą funkcją `main()`), niż w oddzielnym pliku *h*. Umieszczanie deklaracji w pliku nagłówkowym jest wyrazem profesjonalizmu programisty, jego wysokiej kultury, dobrego smaku i zamiłowania do porządku. Jednak drobniutkie algorytmy z całym spokojem możemy umieszczać wyłącznie w pliku *cpp*.

Postarajmy się zapamiętać, że język C++ w najlepszym, profesjonalnym wydaniu operuje parą plików *cpp* i *h* oraz że para ta nazywa się **modułem**.

Dyrektywa #include i scalanie plików cpp i h

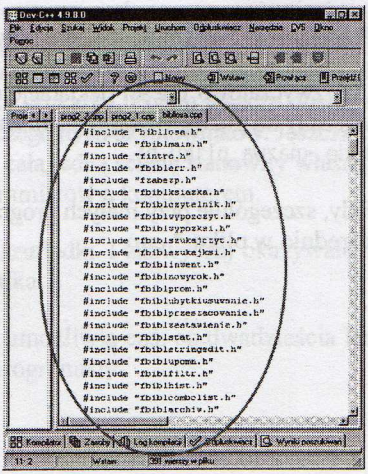
A oto następny szczegół, na który powinniśmy zwrócić uwagę. Skoro język C++ wprowadza do gry dwa pliki źródłowe, tym samym rozdzielając tak zwane *deklaracje* (zapowiedzi algorytmów) od tak zwanych *implementacji* (algorytmów), to w strukturze języka powinna znaleźć się dyrektywa łączenia pary takich plików w całość. Dla porównania — w Pascalu, Fortranie czy Basicu problem ten nie występuje, bo zarówno deklaracje, jak i implementacje umieszczamy w jednym i tym samym pliku.

Zazwyczaj na samej górze pliku *cpp* — czyli na samym początku spisywanych algorytmów — pojawia się dyrektywa, nakazująca kompilatorowi spojrzenie w jakiś plik nagłówkowy. Przypomnijmy sobie którykolwiek z programów z poprzedniego rozdziału:

```
#include <iostream>
using namespace std;
int main()
{
 ...
```

Widoczna tutaj w pierwszej linii dyrektywa nakazuje kompilatorowi przeczytanie pewnego innego pliku, tutaj o nazwie *iostream*. Co prawda plik ten nie ma oczekiwanej nazwy *iostream.h*, a tylko *iostream*, i jest to swego rodzaju naruszenie zasad uświęconych tradycją, ale dyrektywie wklejania w niczym to nie przeszkadza. Rzeczywiście — wklejaniu za pomocą dyrektywy `#include` podlegają wszelkie pliki, nawet te o najdziwniejszych nazwach, czego raczej nie należy nadużywać.

Wklejać pliki nagłówkowe trzeba praktycznie zawsze. Oddzielną sprawą jest orientowanie się, jaki plik akurat należy wkleić. Jeśli potrzebne jest wyprowadzanie informacji na ekran — chętnie wklejamy plik nagłówkowy *iostream*. Jeśli potrzebujemy jakiejś funkcji matematycznej, wkleimy prawdopodobnie plik *math.h* zawierający zapowiedzi instrumentów matematycznych — stałej *pi*, sinusa, pierwiastka czy logarytmu (porównaj rysunek 3.2).



Rysunek 3.2. *Poważniejsze programy intensywnie wykorzystują dyrektywę wklejania. Niech nas nie zaniepokoją cudzysłowy zamiast ostrych nawiasów, jak w naszych mikroprogramach — jest to pozostałość po dawnych czasach, gdy maszyny nie były tak szybkie jak dzisiaj. Ostre nawiasy nakazują rozpoczęcie wyszukiwania od katalogów bibliotecznych kompilatora. Kultura nakazuje, by swoje deklaracje wklejać średnikami, biblioteczne — nawiasami ostrymi*

Ćwiczenie 3.2.

Teraz zrobimy coś złego. Z któregoś z poprzednich programów usuńmy dyrektywę #include i poddajmy program kompilacji oraz uruchomieniu (porównaj rysunek 2.3):

```
//#include <iostream>
using namespace std;
int main()
{
 cout << "Twoje imie i nazwisko";
 return 0;
}
```

Pierwsza linia została poprzedzona podwójnym ukośnikiem, zatem jest zamieniona na komentarz i nie podlega kompilacji. Czy ten program się uruchamia? Nie. Nawet się nie kompiluje — kompilacja kończy się komunikatem „niezadeklarowane cout", „nie rozumiem cout", „nie wiem, co znaczy cout"! Widocznie w pliku *iostream* znajdował się opis algorytmu `cout`.

Podsumowanie

Nośnikami współczesnych programów komputerowych są zazwyczaj zwykłe pliki tekstowe.

Języki rezerwują sobie rozszerzenia nazw plików — i tak pliki w języku C++ mają nazwy **.cpp* i **.h*, pliki pascalowe nazywają się **.pas*, pliki z Fortranem **.for* itd.

Zapisuj swoje algorytmy w swoich plikach i strzeż ich jak oka w głowie!

Staraj się wyrobić w sobie nawyk, by pliki każdego programu umieszczać w oddzielnym katalogu. Jest to ważne dlatego, że współczesne, duże programy zazwyczaj składają się z wielu plików źródłowych i trudno jest lokalizować je, gdy mieszają się z plikami innych programów.

Plik **.cpp* jest głównym nośnikiem algorytmów spisanych w języku C++.

Plik **.h* zwyczajowo mieści deklaracje (zapowiedzi) algorytmów w języku C++. Plik **.h* jest włączany do pliku głównego **.cpp* za pomocą dyrektywy `#include "nazwa_pliku.h"` lub `#include <nazwa_pliku.h>`.

Niekiedy, szczególnie przy małych programach, pomija się plik **.h* i deklaracje umieszcza bezpośrednio w pliku **.cpp*.

Rozdział 4.

Więcej o strumieniach cin i cout

Celem każdego ćwiczenia jest doskonalenie umiejętności programowania. Trudno byłoby osiągnąć ten cel, gdybyśmy nie mieli natychmiastowej możliwości oceny osiągniętego rezultatu, czyli sprawdzenia, jak nasz mikroprogram działa. To z kolei będzie możliwe, jeśli wbudujemy w program elementarne instrukcje wypisywania treści na ekranie. W ten sposób najszybciej przekonamy się, czy nasz algorytm dodawania liczby 2 do siebie rzeczywiście daje wartość 4, zatem działa poprawnie.

Niech mi będzie wolno zrobić małą dygresję, że to oczywiste testowanie poprawności działania programów nie zawsze było takie proste. Jeszcze 20 lat temu programy dostarczało się do ośrodka obliczeniowego w formie czytelnie wypełnionych, specjalnych formularzy. Tam programy trafiały do hali dziurkarek, gdzie kilkadziesiąt pań zajmowało się przenoszeniem treści z formularzy dostarczonych przez programistów na perforowane karty. Karty te następnie były czytane przez specjalne urządzenie wejściowe komputera i program zostawał wykonany. Urządzeniem wyjściowym komputera była ogromna jak szafa, bardzo głośna i niebywale szybka drukarka. Potem jeszcze obsługa drukarki cięła wydruki i wkładała je do szufladek poszczególnych użytkowników. Jeszcze tylko wspomnę, że w ośrodku była specjalna hala, której całą jedną ścianę stanowiły właśnie owe szufladki — jedyny interfejs między rzeszą programistów a komputerem.

Po tygodniu należało zajrzeć do swojej szufladki. I zazwyczaj okazywało się, że program „nie poszedł", bo gdzieś zabrakło średnika…

Doceńmy dzisiejszą technologię, która umożliwia coś, co dwadzieścia lat temu było nie do pomyślenia — interaktywną pracę programisty.

Standardowe strumienie wejścia i wyjścia

W tym rozdziale określimy instrukcje służące do wprowadzania informacji z klawiatury i wypisywania jej na ekranie. Nie zdefiniujemy tych operacji ściśle. Po prostu nauczymy się nimi posługiwać, co okaże się zaskakująco łatwe.

Chodzi mi o nowoczesne operacje wejścia-wyjścia, zwane **standardowymi strumieniami**. Mówiąc nieco ściślej, standardowe strumienie są to **obiekty** (za jakiś czas dowiemy się, co oznacza słowo obiekt) zajmujące się pobieraniem informacji o naciskanych klawiszach i wyprowadzaniem informacji na ekran.

Podam Państwu receptę na posługiwanie się strumieniami, ale niestety nie wyjaśnię licznych zawiłości ich natury. Na szczęście nie jest to potrzebne — prawdę mówiąc, nawet zaawansowani programiści posługują się strumieniami, ale rzadko kiedy znają ich budowę.

Obiekt o nazwie `cin` (ang. *console input* — wejście z klawiatury) obserwuje klawiaturę i rejestruje wciskane przez użytkownika klawisze. Jeśli pod symbolem `a` kryje się jakaś zmienna, to fraza:

```
cin >> a;
```

oznacza, że program oczekuje na wczytanie do zmiennej `a` wartości ze standardowego strumienia wejściowego, czyli z klawiatury.

Strumienia `cin` użyjemy też do zatrzymania konsoli na ekranie aż do momentu naciśnięcia jakiegoś klawisza (porównaj ćwiczenie 2.2).

Obiekt o nazwie `cout` (ang. *console output* — wyjście na ekran) wypisuje określone dane w czarnym okienku konsoli, pokazanej na rysunku 1.1. Fraza:

```
cout << a;
```

oznacza, że program wypisze na ekranie wartość kryjącą się pod zmienną o nazwie `a`.

Ćwiczenie 4.1.

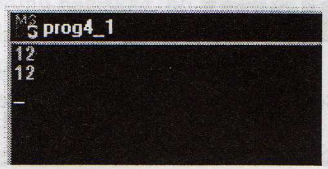
Niech program zaczeka na wprowadzenie jakiejś liczby i potem niech wypisze ją na ekranie.

Wykonajmy sekwencję czynności opisanych i zilustrowanych w ćwiczeniu 2.1. Czynności te prowadziły do uzyskania czystego okienka edytora, czyli do rozpoczęcia prac nad nowym programem.

Po wykonaniu czynności wstępnych wprowadźmy do edytora następujący tekst programu, realizującego postawione tutaj zadanie.

```
#include <iostream>
using namespace std;
int main()
{
 int a;
 cin >> a;     //pobranie liczby i wpisanie jej do zmiennej a
 cout << a;    //wyprowadzenie pobranej liczby
 char c;       //na koniec oczekiwanie na naciśnięcie czegokolwiek
 cin >> c;
 return 0;
}
```

Rysunek 4.1.
Efekt działania programu (przy założeniu, że wprowadzono z klawiatury liczbę 12)



W pierwszej linii zapoznajemy kompilator ze strumieniami standardowego wejścia `cin` i wyjścia `cout`. W drugiej linii udostępniamy programowi algorytmy *biblioteki standardowej* współczesnego języka C++. Dalej widoczna funkcja `main()` jest koniecznym elementem konsolowego programu, napisanego w języku C++. W funkcji tej deklarujemy zmienną roboczą o nazwie `a`, wczytujemy do niej numeryczną wartość z klawiatury i wypisujemy ją na ekranie.

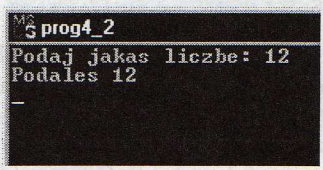
Końcowe linie mają znaczenie techniczne — realizują oczekiwanie na wciśnięcie klawisza, zapobiegając tym samym natychmiastowemu zamknięciu okienka konsoli przez system operacyjny, gdy tylko program dobiegnie końca (porównaj ćwiczenie 2.2).

Ćwiczenie 4.2.

Niech program uprzejmiej zapyta o jakąś liczbę i potem uprzejmiej wypisze ją na ekranie.

```
#include <iostream>
using namespace std;
int main()
{
 int a;
 cout << "Podaj jakas liczbe: ";
 cin >> a;
 cout << "Podales " << a;
 char c;
 cin >> c;
 return 0;
}
```

Rysunek 4.2.
Efekt działania programu (przy założeniu, że wprowadzono z klawiatury liczbę 12)



Przed zapytaniem o liczbę wyprowadzamy na ekran tekst zachęty, tak by użytkownik wiedział, czego program od niego oczekuje. Potem następuje znana już fraza pobierania liczby z klawiatury. A potem wyprowadzanie na ekran nie tylko liczby, ale i stosownego komentarza. Zauważmy tam bardzo przydatny szczegół — tak zwaną *kaskadowość* strumienia wyjściowego, polegającą na kolejnym kierowaniu do strumienia cout kilku wartości.

Kaskadowe posługiwanie się strumieniami

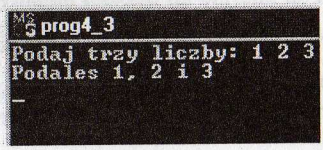
Niech *kaskadowość* strumieni będzie przedmiotem następnego ćwiczenia:

Ćwiczenie 4.3.

Niech program uprzejmie zapyta o trzy liczby i uprzejmie wypisze je na ekranie.

```
#include <iostream>
using namespace std;
int main()
{
 int a1, a2, a3;
 cout << "Podaj trzy liczby: ";
 cin >> a1 >> a2 >> a3;
 cout << "Podales " << a1 << ", " << a2 << " i " << a3;
 char c;
 cin >> c;
 return 0;
}
```

Rysunek 4.3. *Efekt działania programu (przy założeniu, że wprowadzono z klawiatury liczby 1, 2 i 3)*



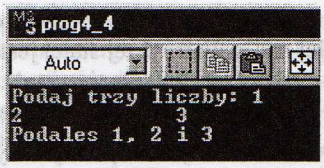
Przykład powyższy ilustruje ważną i przydatną cechę strumieni — możliwość kaskadowego pobierania danych z klawiatury i równie kaskadowego wyprowadzania informacji na ekran. Zatrzymajmy się na chwilę przy kaskadowym pobieraniu danych.

Ćwiczenie 4.4.

Uruchom ponownie ostatni program i jako jego użytkownik sprawdź, że obowiązują następujące, raczej oczywiste zasady:

1. Dane w sekwencji (kaskadzie) muszą być oddzielone tak zwanymi białymi znakami — *spacją*, *tabulatorem*, *enterem*. Ilość takich rozdzielników, kończących wprowadzanie jednej zmiennej i zaczynających wczytywanie następnej, jest nieistotna (rysunek 4.4). Zasada ta oznacza, że po wprowadzeniu pierwszej jedynki w poprzednim przykładzie należy nacisnąć dowolną ilość razy jakiś klawisz białego znaku, po czym wprowadzić kolejną wartość.

Rysunek 4.4.
Oto obraz konsoli



2. Całą sekwencję wprowadzanych danych kończymy klawiszem *Enter*. Program opuszcza instrukcję wczytywania i przechodzi do następnej instrukcji.

3. Typy zmiennych, oczekujących na napływające ze strumienia `cin` wartości, określają, jakich klawiszy nie wolno naciskać, bo wprowadzanie danych się nie powiedzie. Jeśli w poprzednim przykładzie zmienne `a1`, `a2`, `a3` były typu `int` (czyli całkowitego), należało wciskać klawisze numeryczne.

Przyznajmy, że zasad takich można było oczekiwać.

Odrobina formatowania

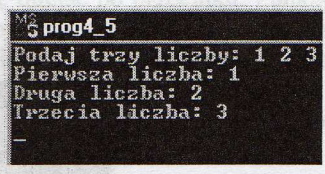
Strumienie `cin` i `cout` są dość bogato oprogramowane. Nie chcemy jednak tracić z oczu użytkowego charakteru naszej wiedzy na temat wejścia i wyjścia i dlatego ograniczymy się do kilku prostych funkcji formatujących.

Ćwiczenie 4.5.

Tak zmodyfikuj ostatni program, by trzy pobrane z klawiatury wartości zostały wyprowadzone w trzech oddzielnych liniach:

```
#include <iostream>
using namespace std;
int main()
{
 int a1, a2, a3;
 cout << "Podaj trzy liczby: ";
 cin >> a1 >> a2 >> a3;
 cout << "Pierwsza liczba: " << a1 << endl << "Druga liczba: " << a2 << endl <<
"Trzecia liczba: " << a3;
 char c;
 cin >> c;
 return 0;
}
```

Rysunek 4.5.
Efekt działania programu (przy założeniu, że wprowadzono z klawiatury liczby 1, 2 i 3)



W powyższym programie należy zauważyć w linii kaskadowego wypisywania danych wtrącenia stałej `endl`. Wartość tej stałej jest uzgodniona ze strumieniem i oznacza *zmień linię!* (`endl` — *end of line*). Zapamiętaj to!

Program z ćwiczenia 4.5 jest napisany brzydko. Byłoby lepiej już na poziomie listingu zaznaczyć miejsca, w których linia wypisywanego tekstu jest łamana:

```
...
cout << "Pierwsza liczba: " << a1 << endl;
cout << "Druga liczba: " << a2 << endl;
cout << "Trzecia liczba: " << a3;
...
```

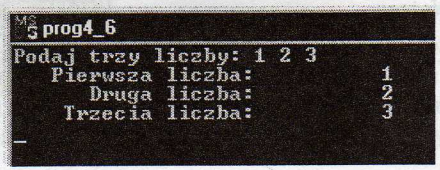
Czyli nie przesadzajmy z kaskadowością! Czytelność programu też jest ważna.

Może też przydać się taki element formatowania, który zadecyduje o szerokości pola, w jakim ma być wyprowadzona zmienna. Załóżmy, że piszemy program księgowy — wiadomo, że kolejne kwoty muszą być starannie wyrównane w polach o ustalonej szerokości. Jak ustalić szerokość pola przeznaczonego na zmienną?

Ćwiczenie 4.6.

Tak zmodyfikuj ostatni program, by wyprowadzane teksty i następujące po nich wartości układały się w czytelne kolumny:

```
#include <iostream>
using namespace std;
int main()
{
 int a1, a2, a3;
 cout << "Podaj trzy liczby: ";
 cin >> a1 >> a2 >> a3;
 cout.width( 20);
 cout << "Pierwsza liczba: ";
 cout.width( 10);
 cout << a1 << endl;
 cout.width( 20);
 cout << "Druga liczba: ";
 cout.width( 10);
 cout << a2 << endl;
 cout.width( 20);
 cout << "Trzecia liczba: ";
 cout.width( 10);
 cout << a3;
 char c;
 cin >> c;
 return 0;
}
```



Rysunek 4.6. *Efekt działania programu (przy założeniu, że wprowadzono z klawiatury liczby 1, 2 i 3)*

Ustalanie szerokości pola przeznaczonego na zmienną wymaga wywołania należącej do strumienia funkcji `width()` — szerokość. Argumentem tej funkcji jest właśnie szerokość pola. Funkcja należy do obiektu strumienia, dlatego sięgamy po nią za pomocą charakterystycznej dla języków obiektowych frazy `cout.width()` dającej się wypowiedzieć jako „uruchom algorytm `width()`, należący do obiektu `cout`!".

Zauważmy, że funkcję `width()` wywołujemy przed każdym wyprowadzaniem z ustalaną szerokością pola.

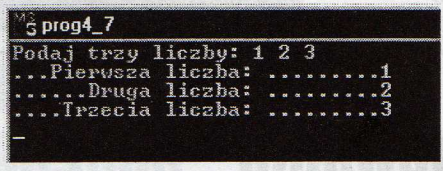
Nadmiar pola o szerokości zdefiniowanej funkcją `width()` nie musi być wypełniany spacjami. Możemy zdecydować, że będzie to jakiś inny znak.

Ćwiczenie 4.7.

Tak zmodyfikuj ostatni program, by nadmiar szerokości pól był wypełniany kropkami:

```
#include <iostream>
using namespace std;
int main()
{
 int a1, a2, a3;
 cout << "Podaj trzy liczby: ";
 cin >> a1 >> a2 >> a3;
 cout.fill( '.');
 cout.width( 20);
 cout << "Pierwsza liczba: ";
 ...              //dalej bez zmian
```

Rysunek 4.7. *Efekt działania programu*



Funkcja `fill()` — wypełnij, także należąca do obiektu strumienia wyjściowego, jest wywoływana z argumentem znakowym — jest to ten znak, którym zostanie wypełniony nadmiar pola określonego funkcją `width()`. W przeciwieństwie do funkcji `width()`, którą należało wykonywać przed każdym wyprowadzaniem zmiennej, strumień zapamiętuje ustalenie dokonane za pomocą funkcji `fill()`. Wypełnianie nadmiarowej szerokości określonym znakiem obowiązuje aż do ponownego uruchomienia funkcji `fill()` z jakimś innym znakiem.

Trzecim po `width()` i `fill()` elementem formatowania wyprowadzanej informacji jest funkcja `precision()` — ustal liczbę miejsc po przecinku. Funkcja ta — podobnie jak `fill()`, ale nie jak `width()` — wprowadza regułę obowiązującą do końca życia programu albo do kolejnego jej wywołania, zmieniającego precyzję. Funkcja ta także należy do obiektu reprezentującego wyjście na ekran konsoli `cout`.

Ćwiczenie 4.8.

Niech kolejny program pobierze ze standardowego wejścia liczbę rzeczywistą i wypisze jej odwrotność, ograniczając liczbę miejsc po przecinku do dwóch:

```
#include <iostream>
using namespace std;
int main()
{
 double a;
 cout << "Podaj liczbe rzeczywista (z kropka) ";
 cin >> a;      //pobranie liczby i wpisanie jej do zmiennej a
 cout.precision( 2);
 cout << "1 / " << a << " = " << 1/a << endl;
 char c;        //na koniec oczekiwanie na naciśnięcie czegokolwiek
 cin >> c;
 return 0;
}
```

Rysunek 4.8. *Efekt działania programu (przy założeniu, że podano liczbę 3)*

Rysunek 4.9. *Gdyby nie było ustalenia precyzji na dwa znaki po przecinku, program zakończyłby się nieco innym efektem*

Podsumowanie

Po doklejeniu pliku nagłówkowego `iostream` oraz udostępnieniu algorytmów biblioteki standardowej:

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

możemy posługiwać się strumieniowym wejściem i wyjściem na konsolę i z konsoli operatorskiej. Choć natura strumieni jest bardzo złożona, to posługiwanie się nimi jest przyjemnością.

Oprócz samej umiejętności pobierania danych ze strumienia albo wprowadzania do niego, zapamiętajmy użyteczną właściwość *kaskadowości*:

```
cin >> a1 >> a2 >> a3;
cout << "Podales: " << a1 << ", " << a2 << " i " << a3;
```

Mogą też przydać się najprostsze operacje formatowania informacji wypisywanej na ekranie. Zapamiętaj, co powoduje wyprowadzenie stałej `endl` oraz co robią należące do strumienia `cout` funkcje `width()`, `fill()` i `precision()`.

Rozdział 5.

Przestrzeń na Twoje algorytmy

Po napisaniu pierwszych mikroprogramów po raz kolejny uporządkujmy swoją wiedzę i tym razem dokładniej przeanalizujmy strukturę programu konsolowego w języku C++.

Początek — najlepsze miejsce na dyrektywy #include

W pierwszych liniach programu zwyczajowo (bo nie ma wyraźnego nakazu) wklejamy pliki nagłówkowe, opisujące deklaracje różnych zmiennych, stałych (np. `endl`), funkcji czy obiektów (np. `cin`, `cout`). Dlaczego w pierwszych liniach? Bo z pewnością program musi najpierw poznać deklarację (czyli zapowiedź) jakiegoś elementu, a dopiero potem wolno nam go użyć w algorytmie. Skoro tak, najbezpieczniej jest przestrzegać zasady, by linie wklejeń plików nagłówkowych otwierały tekst źródłowy programu. Będzie to też zgodne z niepisanymi zasadami estetycznego programowania.

Przypomnijmy sobie, że wszystkie dotychczasowe programy rozpoczynały się od linii wklejenia nagłówka z deklaracjami strumieni informacji wchodzących do programu i wychodzących z niego na ekran konsoli:

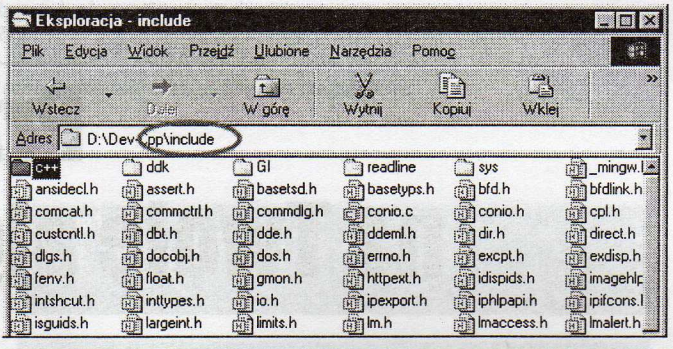
```
#include <iostream>
```

Zależnie od potrzeb będziemy w kolejnych liniach dodawać różne pliki z innymi deklaracjami.

Ćwiczenie 5.1.

Zapoznaj się z bogactwem bibliotek udostępnionych programiście:

1. Odszukaj na dysku swojego komputera katalog, w którym zainstalowaliśmy środowisko programistyczne.
2. W katalogu tym — o ile podczas instalacji nie dokonano jakichś innych ustaleń — powinien znajdować się podkatalog o nazwie *include*, a w nim kilkaset plików nagłówkowych, oddanych do dyspozycji programisty (rysunek 5.1).



Rysunek 5.1. *W katalogu include znajdujemy kilkaset plików nagłówkowych, a także dalsze podkatalogi — np. gl zawiera pliki umożliwiające wejście w świat biblioteki OpenGL czy c++ — zawierający deklaracje algorytmów biblioteki standardowej*

Ćwiczenie 5.2.

Napisz program, który spyta użytkownika o wartość kąta i po jego określeniu wyliczy oraz wyprowadzi na ekran wartości jego sinusa i cosinusa.

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
 double a;       //deklaracja liczby rzeczywistej
 cout << "Podaj kat w radianach: ";
 cin >> a;       //pobranie liczby i wpisanie jej do zmiennej a
 cout << "sin( " << a << ") = " << sin( a) << endl;
 cout << "cos( " << a << ") = " << cos( a);
 char c;         //na koniec oczekiwanie na naciśnięcie czegokolwiek
 cin >> c;
 return 0;
}
```

Rysunek 5.2. *Efekt działania programu*

W tym przykładzie wywołujemy dwie biblioteczne funkcje *sinus* i *cosinus*, zadeklarowane w pliku *math.h* — zatem plik ten trzeba dokleić w sekcji doklejania nagłówków.

Po nagłówkach — dostęp do biblioteki standardowej

Umówmy się też, że po sekcji wklejeń plików nagłówkowych udostępnimy programowi algorytmy biblioteki standardowej. Znów nie ma wyraźnego, ostrego nakazu, by robić to dokładnie w tym miejscu programu, ale jakiś porządek być musi:

```
//blok wklejeń nagłówków
using namespace std;
...
```

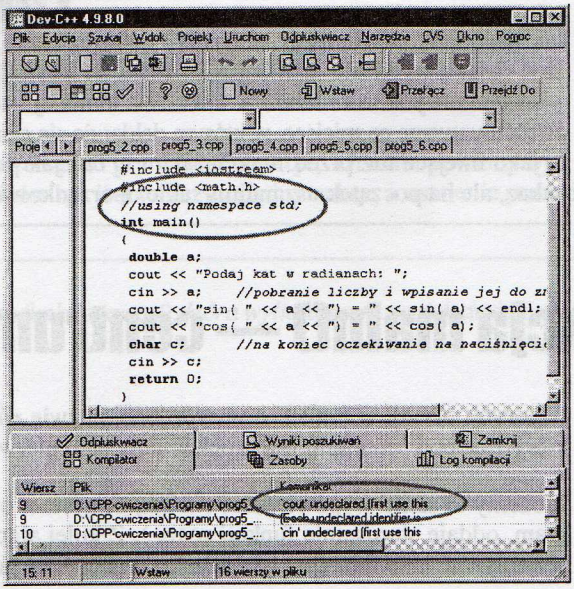
Po wklejeniu odpowiednich nagłówków (co dyskutowaliśmy w poprzednim podrozdziale) i udostępnieniu **przestrzeni nazw** (jest to wolne tłumaczenie frazy `namespace`) algorytmów biblioteki standardowej dysponujemy ogromnymi możliwościami programistycznymi, z czego zapewne jeszcze nie zdajemy sobie sprawy.

Ćwiczenie 5.3.

W poprzednim programie usuń frazę udostępniającą algorytmy biblioteki standardowej i spróbuj uruchomić program:

```
#include <iostream>
#include <math.h>
//using namespace std;
int main()
{
 double a;      //deklaracja liczby rzeczywistej
 cout << "Podaj kat w radianach: ";
 ...
```

Rysunek 5.3. *Program nie kompiluje się — kompilacja w środowisku DEV kończy się komunikatem cout undeclared — niezadeklarowany napis cout*



Po bibliotece standardowej — nasze własne deklaracje

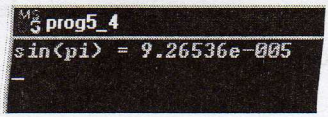
Po *wklejeniu nagłówków* i *udostępnieniu przestrzeni nazw biblioteki standardowej* w typowym programie znajduje się sekcja deklaracji (zapowiedzi) globalnych danych i algorytmów. Globalnych — to znaczy widocznych w całym programie, a nie w jakiejś wybranej funkcji.

Ćwiczenie 5.4.

Napisz program, w którym zadeklarujesz liczbę pi jako globalną stałą, a w funkcji main() wyliczysz i wyprowadzisz na ekran wartość jej sinusa.

```
#include <iostream>
#include <math.h>
using namespace std;
const double pi = 3.1415; //deklaracja stałej pi
int main()
{
 cout << "sin(pi) = " << sin( pi) << endl;
 char c;                //na koniec oczekiwanie na naciśnięcie czegokolwiek
 cin >> c;
 return 0;
}
```

Rysunek 5.4. *Efekt działania programu*



Jak wiadomo, powinniśmy w wyniku otrzymać zero, ale zapewne nasza wartość *pi* jest zbyt daleka od ideału. Zmienna o nazwie *pi* jest tutaj opatrzona modyfikatorem `const` — zatem nie jest to *zmienna*, a *stała*. Jest to mniej istotny szczegół — przede wszystkim zwróćmy uwagę na miejsce, w którym deklaruje się elementy globalne programu. Określmy je jako miejsce tuż przed funkcją `main()`. Tak jak poprzednio nie jest to rygorystyczny nakaz, ale na początek uszanujmy takie uporządkowanie.

Funkcja main() — centrum programu

Algorytm ukryty w funkcji o wyróżnionej nazwie *main* (główna) jest tym algorytmem, który z całego programu jest uruchamiany jako pierwszy.

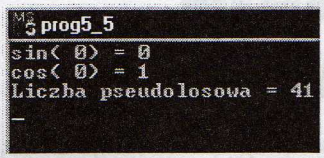
Możemy to samo wypowiedzieć też tak, że system operacyjny, uruchamiając nasz program, oddaje sterowanie funkcji `main()`. Z kolei algorytm ujęty w funkcji `main()` może wywoływać inne funkcje i tym samym uruchamiać jakieś inne algorytmy.

Ćwiczenie 5.5.

Napisz program, który z funkcji main() wywołuje jakieś inne, biblioteczne funkcje:

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
 cout << "sin( 0) = " << sin( 0.0) << endl;
 cout << "cos( 0) = " << cos( 0.0) << endl;
 cout << "Liczba pseudolosowa = " << rand();
 char c;
 cin >> c;
 return 0;
}
```

Rysunek 5.5.
Oto wyjście programu



Z wnętrza funkcji `main()` nastąpiło wywołanie trzech innych funkcji, dostarczonych wraz z bibliotekami języka C++. Skazanie języka na operowanie jedyną w swoim rodzaju funkcją `main()` w istocie nie jest żadnym ograniczeniem, bowiem z wnętrza tej funkcji natychmiast i bez ograniczeń możemy wywołać inne funkcje. Funkcja `main()` wyznacza tylko miejsce startu programu.

Po funkcji main() — definicje innych funkcji

Jeśli nasz program wprowadza do gry własne funkcje, ich ciała powinny być spisane za ciałem funkcji `main()`. Znów nie jest to bezwzględny nakaz, a ustalenie porządkowe.

Ćwiczenie 5.6.

Napisz program operujący funkcją suma(), która wylicza sumę swoich dwóch argumentów:

```
#include <iostream>
using namespace std;
int suma( int a, int b);
int main()
{
 cout << "2 + 3 = " << suma( 2, 3);
 char c;
 cin >> c;
 return 0;
}
```

```
//-----------------------------------------------
int suma( int a, int b)
{
 return a + b;
}
```

Rysunek 5.6.
Wynik działania tego programu

Poszczególne elementy tego programu omówimy w rozdziale 7. Teraz skoncentrujmy się przede wszystkim na jego strukturze — zauważmy linię deklaracji funkcji `suma()`, jej wywołanie w funkcji `main()` i jej definicję za ciałem funkcji `main()`.

Podsumowanie

Program w języku C++ ma określoną strukturę, podyktowaną zarówno koniecznością, jak i powszechnie przyjętą estetyką programowania. Pisząc nawet najprostszy program, starajmy się przestrzegać następującego układu kodu źródłowego:

1. Najpierw dyrektywy `#include`, włączające akurat potrzebne pliki nagłówkowe (pliki z deklaracjami).
2. Potem udostępnienie przestrzeni nazw biblioteki standardowej.
3. Deklaracje globalnych elementów programu — funkcji i danych.
4. Funkcja `main()` wraz z jej algorytmami.
5. Definicje funkcji, napisane przez programistę i zadeklarowane w punkcie 3.

Rozdział 6.

Algorytmy

Programy komputerowe składają się z dwóch wielkich elementów: porozmieszczanych w pamięci maszyny **danych** i operujących na nich **algorytmów**. To stwierdzenie nadaje rytm najbliższym rozdziałom książki — teraz zajmiemy się algorytmami, a potem zobaczymy, jak wprowadza się do gry różne zestawy danych.

Algorytmy konstruujemy za pomocą tak zwanych **instrukcji sterujących**. Każdy język programowania ma zestaw specyficznych instrukcji sterujących. W języku C++ znajdujemy siedem instrukcji: dwie zwrotnice, trzy pętle, polecenie przerwania i kontynuowania. Są jeszcze pewne drobiazgi, ale skoncentrujemy się na tej siódemce.

Zwrotnica if() ... else ...

Bardzo często w programie zachodzi potrzeba podjęcia jakiejś decyzji, warunkowego wykonania określonego skrawka algorytmu.

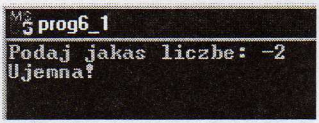
Ćwiczenie 6.1.

Napisz program, który oceni, czy wczytana liczba jest ujemna.

```
#include <iostream>
using namespace std;
int main()
{
 double a;
 cout << "Podaj jakas liczbe: ";
 cin >> a;
 if( a < 0)
   cout << "Ujemna!";
 else
   cout << "Nieujemna!";
```

```
 char c;                    //na koniec oczekiwanie na naciśnięcie czegokolwiek
 cin >> c;
 return 0;
}
```

Rysunek 6.1.
Efekt działania ostatniego programu



Najbardziej klasyczna zwrotnica `if(warunek) ... else ...` daje się bardzo łatwo przetłumaczyć z języka programowania na język potoczny:

Jeśli warunek jest spełniony, rób to, a w przeciwnym wypadku — tamto.

W naszym przykładzie warunkiem było pytanie, czy `a` jest mniejsze od zera. Jeśli tak, to program wypisał na ekranie konsoli komentarz `ujemna!`, a w przeciwnym wypadku `nieujemna!`.

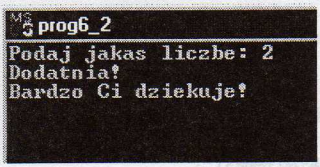
To było proste, ale w następnym przykładzie musimy zwrócić uwagę na bardzo ważny drobiazg, o którym programiści często zapominają.

Ćwiczenie 6.2.

Napisz program, który oceni, czy wczytana liczba jest dodatnia i jeśli tak, niech dodatkowo pochwali on użytkownika.

```
#include <iostream>
using namespace std;
int main()
{
 double a;
 cout << "Podaj jakas liczbe: ";
 cin >> a;
 if( a > 0)
 {
   cout << "Dodatnia!" << endl;
   cout << "Bardzo Ci dziekuje!";
 }
 else
 {
   cout << "Nie jest dodatnia?";
 }
 char c;                    //na koniec oczekiwanie na naciśnięcie czegokolwiek
 cin >> c;
 return 0;
}
```

Rysunek 6.2.
Efekt działania ostatniego programu



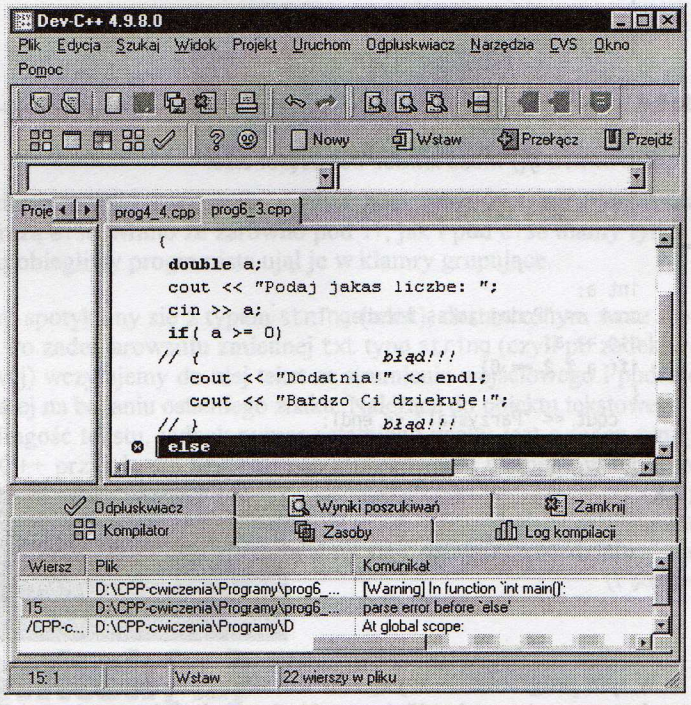
Ten program niewiele różni się od poprzedniego. Różnica ta polega na tym, że teraz pod instrukcją sterującą `if()` mamy nie jedną, a więcej linijek programu — konkretnie dwie. To bardzo częsta sytuacja, bo niby dlaczego warunkowo wykonujący się skrawek algorytmu miałby składać się z tylko jednej instrukcji? W takich sytuacjach musimy zgrupować instrukcje za pomocą pary giętych nawiasów, zwanych obrazowo **instrukcją grupującą**.

Ćwiczenie 6.3.

W poprzednim programie usuń instrukcję grupującą pod warunkiem if().

```
...
if( a > 0)
   cout << "Dodatnia!" << endl;
   cout << "Bardzo Ci dziekuje!";
else
{
   cout << "Nie jest dodatnia?";
}
...
```

Program nie działa, bo podczas kompilacji pojawia się błąd, polegający na niemożności zinterpretowania części zwrotnicy `else` (rysunek 6.3).



Rysunek 6.3. *Błąd w strukturze zwrotnicy else ... if*

Analizując przyczynę błędu i nie ingerując w merytoryczną stronę programu, z powodów estetycznych zmieńmy wcięcia przed frazami:

```
...
if( a > 0)
   cout << "Dodatnia!" << endl;
cout << "Bardzo Ci dziekuje!";
else
{
   cout << "Nie jest dodatnia?";
}
...
```

Fraza wypisywania tekstu `Bardzo Ci dziekuje!` nie jest uwarunkowana (jak mawiamy — nie jest umieszczona pod *ifem*...). Jest to zwyczajna linijka programu, która wykonuje się zawsze, niezależnie od wyniku warunku logicznego `if()`. Ale po tej linijce w programie pojawia się fraza `else`, która nie może samodzielnie istnieć. Fraza `else` jest za daleko od frazy `if()` i kompilator zgłasza błąd.

Jest na to wszystko prosta rada — w zwrotnicach (a także w pętlach) zawsze stosuj *instrukcje grupujące*, czyli nawiasowanie sześcienne, nadając algorytmowi taką oto postać:

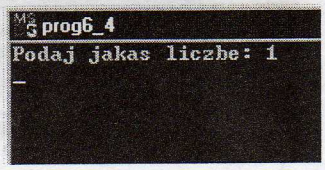
```
if( warunek)
{
 ...
}
else
{
 ...
}
```

Ćwiczenie 6.4.

Zwrotnica if() może istnieć bez części else:

Uwaga — zaczynamy pomijać oczywiste części programu!

```
int a;
cout << "Podaj jakas liczbe: ";
cin >> a;
if( a % 2 == 0)
{
  cout << "Parzysta!" << endl;
}
```



Rysunek 6.4. *Efekt działania programu (podano liczbę 1)*

W tym programie — wyłącznie dla skrócenia zapisu w książce — pominięto oczywiste elementy włączania plików nagłówkowych, udostępniania biblioteki standardowej oraz nagłówka i zakończenia funkcji `main()`. Przytoczyliśmy tylko kilka kluczowych linii algorytmu.

Program ten wypisuje komentarz `Parzysta!`, ale tylko wtedy, gdy reszta z dzielenia podanej liczby przez 2 jest równa 0. W warunku logicznym `if()` widzimy ciekawy operator `%` uzyskiwania reszty z dzielenia oraz operator == (powtórzony znak równa się) — operator pytania, czy ta reszta jest równa 0. Brakuje natomiast części `else`, która nie jest konieczna.

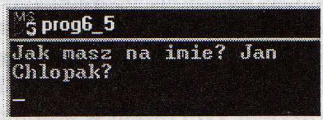
Ćwiczenie 6.5.

Napisz program, który spyta użytkownika o imię i na podstawie analizy brzmienia rozpozna jego płeć:

Uwaga — zaczynamy pomijać oczywiste części programu!

```
string txt;
cout << "Jak masz na imie? ";
cin >> txt;
if( txt[ txt.size() - 1] == 'a')
{
  cout << "Dzieczyna!" << endl;
}
else
{
  cout << "Chlopak?" << endl;
}
```

Rysunek 6.5. *Efekt działania programu (podano imię Jan)*



W tym programie fraza `if()` występuje — choć jak mówiliśmy, nie musi — wraz ze stowarzyszoną frazą `else`. Mimo że zarówno pod `if`, jak i pod `else` mamy tylko po jednej instrukcji, zapobiegliwy programista ujął je w klamry grupujące.

W programie tym spotykamy się z typem `string` (tekst), dostarczonym wraz z biblioteką standardową. Po zadeklarowaniu zmiennej `txt` typu `string` (czyli po zadeklarowaniu zmiennej tekstowej) wczytujemy do niej tekst ze strumienia wejściowego i poddajemy go analizie, polegającej na badaniu ostatniego znaku. Należąca do obiektu tekstowego funkcja `size()` zwraca długość tekstu, jednak numer ostatniego znaku jest o jeden mniejszy od tej długości (w C++ przyjęło się rozpoczynać numerowanie elementów tablic od zera, nie od jedności).

O bibliotece standardowej trochę więcej dowiemy się z ostatniego rozdziału.

Zwrotnica switch{ ...}

Przed chwilą omówiona zwrotnica `if() ... else ...` decydowała, czy wykonywać ten skrawek algorytmu, czy tamten. Zubożona zwrotnica `if()` umożliwiała warunkowe wykonanie części programu (albo — jak kto woli — pomijanie fragmentów programu).

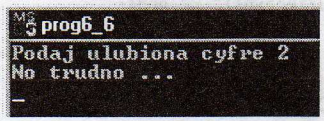
Teraz opiszemy zwrotnicę wielokierunkową, będącą w stanie wykonać to lub tamto albo tamto i tak dalej.

Ćwiczenie 6.6.

Napisz program, który spyta o ulubioną cyfrę i odpowiednio skomentuje wybór:

```
int a;
cout << "Podaj ulubiona cyfre ";
cin >> a;
switch( a)
{
 case 0: cout << "Nie wstyd Ci?";
         break;
 case 1: cout << "Taka mala?";
         break;
 case 2: cout << "No trudno ...";
         break;
//itd., itd.
}
```

Rysunek 6.6. *Efekt działania programu (podano cyfrę 2)*



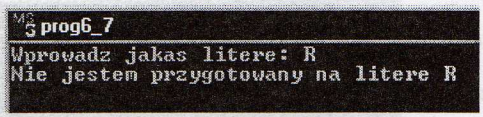
Nie chciało nam się obstawiać wszystkich typów — stąd komentarz pod koniec instrukcji `switch`, ale zapewne wyczuwają Państwo rytm zwrotnicy wielokierunkowej. W nagłówku tej konstrukcji zawsze jest wymieniona **zmienna kluczująca** (musi to być tak zwana zmienna wyliczeniowa, czyli znak lub liczba całkowita, nie tekst albo liczba z przecinkiem). A tuż za nagłówkiem, tym razem w absolutnie obowiązkowych klamrach sześciennych, następuje opisanie reakcji programu na różne warianty stanu zmiennej kluczującej.

Ćwiczenie 6.7.

Napisz program, który poprosi o jakąś literę i poda odpowiednie imię:

```
char znak;
cout << "Wprowadz jakas litere: ";
cin >> znak;
switch( znak)
{
 case 'a':
 case 'A':
         cout << "A jak Ania";
         break;
 case 'b':
 case 'B':
         cout << "B jak Bogdan";
         break;
//itd., itd.
 default:
         cout << "Nie jestem przygotowany na litere " << znak;
         break;
}
```

Rysunek 6.7. *Efekt działania programu (podano literę R)*



Z przykładu tego wynika, że warianty opisane frazą `case` (przypadek, wariant) można w zwrotnicy wielokierunkowej łączyć, wymieniając je bezpośrednio jeden pod drugim. Takie łączenie oznacza, że warianty te kojarzymy z tym samym kodem. W naszym przypadku są to np. warianty, gdy zmienna kluczująca (tym razem typu znakowego `char`) jest równa *a* albo *A* — te dwie różne wartości są przełączane na ten sam tor.

Zauważmy też nowy i **nieobowiązkowy** element w zwrotnicy `switch` — frazę `default`. Wariant `default` (domyślny, pozostały) zostanie wykonany dla wszystkich wartości klucza, nie wymienionych bezpośrednio w wariantach `case`.

Powtórzmy — wariantu `default` może nie być i wtedy zwrotnica, nie znalazłszy odpowiedniego klucza `case` dla swojej zmiennej kluczującej, nie zrobi nic. Jeśli wariant `default` istnieje, oznacza tor przeznaczony dla tych wszystkich stanów zmiennej kluczującej, które nie są bezpośrednio wymienione wraz z frazą `case`. Ta sytuacja trochę przypomina podobną dowolność z występowaniem frazy `else` w prostej zwrotnicy `if()`.

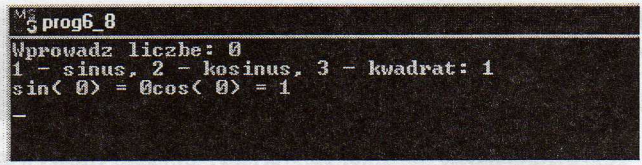
Kolejna sprawa — zauważmy słowo `break` (przerwij) po algorytmie przypisanym każdemu wariantowi. Słowo to nie jest integralną częścią zwrotnicy, ale praktycznie zawsze musi w niej się znaleźć i to wiele razy. Moim zdaniem konieczność kończenia każdego skrawka algorytmu poleceniem `break` jest błędem w konstrukcji języka.

Ćwiczenie 6.8.

Napisz program, który poprosi o liczbę i zależnie od dalszej komendy wyliczy jej sinus albo cosinus, albo kwadrat. Zapomnij o jednym poleceniu break w zwrotnicy switch:

```
#include <math.h>
...
double a;
char znak;
cout << "Wprowadz liczbe: ";
cin >> a;
cout << "1 - sinus, 2 - cosinus, 3 - kwadrat: ";
cin >> znak;
cout.precision( 2);    //2 znaki po przecinku
switch( znak)
{
 case '1':     //sinus
        cout << "sin( " << a << ") = " << sin( a);
 case '2':     //cosinus
        cout << "cos( " << a << ") = " << cos( a);
        break;
 case '3':     //kwadrat
        cout << a << " * " << a << " = " << a * a;
        break;
 default:
        cout << "Niezdefiniowana operacja" << znak;
        break;
}
...
```

Rysunek 6.8.
Oto wynik działania programu przy wybraniu wariantu 1



Zauważmy brak instrukcji `break` w końcówce pierwszego wariantu. Nie jest to błąd językowy, bo program się kompiluje, ale jednak zachowuje się źle.

Wariant *1* nie jest zakończony instrukcją `break`, zatem program po zrealizowaniu wariantu *1* wchodzi na tor przeznaczony dla wariantu *2*. Po wyprowadzeniu wartości sinusa nieproszony przystępuje do wyprowadzania cosinusa. Głupie to, bo czyż instrukcja `break` nie powinna być jakoś wbudowana w strukturę zwrotnicy, np. tak, by kolejny wariant automatycznie kończył poprzedni? Możemy jednak sobie narzekać, ale jakże często programiści zapominają o instrukcji `break` i poprawnie kompilujący się program działa źle. Jest to poważna sprawa, potrafiąca niewprawnemu programiście sprawić dużo kłopotu. Muszą Państwo wypracować w sobie odruch natychmiastowego dopisywania fraz `break` na wszystkich torach zwrotnicy `switch`. Także na ostatnim, bo być może za kilka lat dopiszemy do zwrotnicy jeszcze jeden wariant i zapewne nie sprawdzimy, czy poprzedni wariant kończy się *breakiem*... Zło czai się w chaosie i bałaganie.

Zwrotnica wielotorowa `switch` na pierwszy rzut oka jest dużo doskonalsza od skromnej, co najwyżej dwutorowej frazy `if() ... else`. Ale jest to mylące stwierdzenie, bowiem dotychczas nie rozważaliśmy pewnego istotnego i trudnego do zauważenia szczegółu. Otóż oznaczenia torów w zwrotnicy `switch` muszą być wykonane literałami wprowadzonymi na stałe. Mówiąc to samo inaczej, kompilator musi znać oznaczenia jej torów, zanim program zostanie uruchomiony. Zwrotnica `if()` nie ma takich ograniczeń.

Ćwiczenie 6.9.

Napisz program, który poprosi o dwie liczby i wyprowadzi trzy stosowne teksty na okoliczność, gdy pierwsza jest mniejsza, równa albo większa od drugiej:

```
double a, b;
cout << "Wprowadz liczbe A: ";
cin >> a;
cout << "Wprowadz liczbe B: ";
cin >> b;
if( a < b)
{
   cout << "A < B";
}
else
if( a == b)
{
   cout << "A = B";
}
else
{
cout << "A > B";
}
...
```

Rysunek 6.9. *Efekt działania programu*

W tym mikroprogramie interesuje nas jazda nie po torach ściśle oznaczonych za pomocą etykiet w rodzaju *1* czy *'A'*. Tutaj w momencie pisania programu nie wiemy, o jakie wartości może chodzić, bo użytkownik określi je dopiero po uruchomieniu programu, a ponadto poszukujemy realizacji toru dla wszystkich wartości *a* i *b* takich, że *a* jest mniejsze od *b* (albo równe, albo większe). Są w tych drobiazgach ukryte subtelne i ogromne przewagi zwrotnicy `if()`. Może pracować ze zmiennymi i może operować przedziałami, nie konkretnymi wartościami zmiennych kluczujących. W istocie zwrotnica `if()` jest o wiele bardziej dynamiczna, podczas gdy zwrotnica `switch` musi być obmyślona i zaprojektowana już w momencie pisania programu.

W tym przykładzie widzimy też charakterystyczny ciąg zwrotnic `if() ... else`, w pewnym sensie zastępujący zwrotnicę wielokierunkową. Sekwencję taką można ciągnąć dowolnie długo, dzieląc na przykład jakiś przedział liczb na wiele podprzedziałów.

Ćwiczenie 6.10.

Napisz program, który poprosi o wysokość Twojego miesięcznego dochodu i odpowiednio go skomentuje:

```
double a;
cout << "Powiedz mi, ile zarabiasz... ";
cin >> a;
if( a < 100)
{
   cout << "Do widzenia ...";
}
else
if( a < 500)
{
   cout << "Nie wstyd Ci?";
}
else
if( a < 1000)
{
   cout << "Ferrari za to sobie nie kupisz ...";
}
else
{
   cout << "Sam bym chcial ...";
}
...
```

Rysunek 6.10. *Efekt działania programu*

Znów mamy ów charakterystyczny ciąg zwrotnic dwukierunkowych, dzielący zarobki na jakieś przedziały i odpowiednio je komentujący. Tę konstrukcję warto zapamiętać, bo w żaden sposób takiego wielotorowego algorytmu nie zrealizujemy za pomocą wielotorowej zwrotnicy `switch`.

Pętla for(...; ...; ...)

Co zrobić, gdy pewien fragment algorytmu trzeba wykonać większą ilość razy? Wielokrotnie go powtórzyć w tekście programu? Dobra odpowiedź, ale niezbyt praktyczna, bo zdarzają się algorytmy, które należy powtarzać miliony razy...

Ćwiczenie 6.11.

Napisz program, który zsumuje liczby naturalne od 1 do 1000 oraz wyprowadzi wynik na ekran:

```
int i, a;
for( i = a = 0; i <= 1000; ++i)
{
   a = a + i;
}
cout << "1 + 2 + ... + 1000 = " << a;
...
```

Rysunek 6.11. *Efekt działania programu*

Pętla `for()` **zawsze** w swym ustroju ma dwa średniki, które tworzą trzy charakterystyczne pola.

Pierwsze pole jest *polem inicjowania*. Umieszczony tam algorytm wykonuje się tylko raz, tuż przed pierwszym obrotem pętli.

Drugie (środkowe) pole zawiera warunek, który ma udzielić odpowiedzi na pytanie: *czy wykonać kolejny obrót?*. Pole warunku jest wykonywane wielokrotnie, tuż przed każdym obrotem pętli.

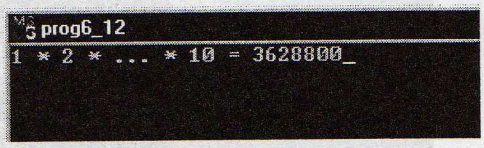
Trzecie (ostatnie) pole zawiera algorytm, który wykonuje się *na koniec każdego obrotu pętli*. Zazwyczaj jest to pole, w którym zwiększamy licznik obrotów pętli, czyli w naszym przykładzie zmienną `i`.

Ćwiczenie 6.12.

Napisz program, który pomnoży liczby naturalne od 1 do 10 za pomocą pętli for(), ale nie wykorzysta jej pierwszego i trzeciego pola:

```
int i = 1, a = 1;
for( ; i <= 10; )
{
   a = a * i;
   ++ i;
}
cout << "1 * 2 * ... * 10 = " << a;
...
```

Rysunek 6.12. *Efekt działania programu*



Pierwsze pole (pole inicjowania, wykonywane tylko przed pierwszym obrotem) jest niewykorzystane. Nic w tym złego, bo programista *zainicjował* zmienne w momencie ich *deklarowania*. Ale jednak przyznajmy, że nie wygląda to ładnie. Pamiętajmy też, że niewykorzystanie możliwości jakiegoś pola nie zwalnia nas od obowiązku wstawienia średnika. Dwa i tylko dwa średniki w pętli `for()` muszą być zawsze.

Drugie pole (zawierające pytanie „czy powtarzać?") jest obecne. Pętla będzie działać, dopóki testowana w tym polu zmienna `i` będzie mniejsza lub równa *10*.

Trzecie pole (wykonywane na koniec każdego obrotu pętli) jest puste, ale znów nic złego się nie dzieje, bo w ostatniej linii ciała pętli programista ręcznie zwiększa licznik obrotów. Powinien to jednak zrobić właśnie w tym polu — choćby tylko z powodów estetycznych.

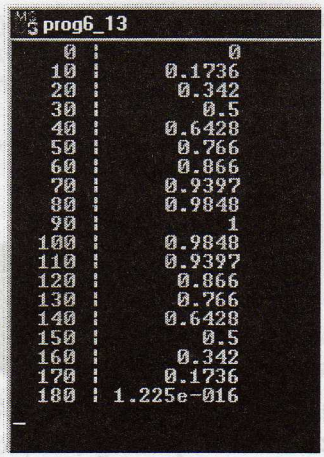
Niewykorzystywanie pól pętli `for()` jest nietaktem. W dodatku każdy programista ma tak wyćwiczone oko, że analizując pola pętli natychmiast spostrzega, skąd ona startuje i ile razy się obróci. Jeśli mamy zamiar nie wykorzystywać jakichś pól pętli `for()`, to za chwilę omówimy jej uproszczoną siostrę, która operuje tylko jednym polem pytania „czy powtarzać?".

Ćwiczenie 6.13.

Napisz program, który stablicuje wartości funkcji sinus dla kątów od 0 do 180 stopni co 10 stopni:

```
#include <math.h>
...
double i, a;
cout.precision( 4);
for( i = 0; i <= 180; i = i + 10)
{
   a = sin( M_PI / 180.0 * i);
   cout.width( 5);
   cout << i << " | ";
   cout.width( 10);
   cout << a << endl;
}
...
```

Rysunek 6.13. *Efekt działania programu*



Analizując strukturę pętli, od razu odczytujemy, że na początku zmienna robocza `i` jest równa *0*, że po każdym obrocie zwiększa się o *10* i że pętla będzie działać tak długo, dopóki `i` będzie mniejsze lub równe *180*. Wewnątrz pętli wyliczamy wartość sinusa (pamiętamy, że stopnie trzeba przeliczyć na radiany). Widoczna tam stała o nazwie `M_PI` reprezentuje liczbę *pi*, zdefiniowaną w doklejonym do tego programu pliku nagłówkowym *math.h*. Nie musimy się więcej martwić o *pi*.

W następnych liniach troszeczkę dbamy o formatowanie kolumn tabelki. Pamiętamy przy tym, że funkcja `width()`, określająca szerokość pola przeznaczonego na wypisanie zmiennej, musi być bezustannie odświeżana, bo jej ustalenie obowiązuje tylko podczas najbliższego użycia strumienia `cout`.

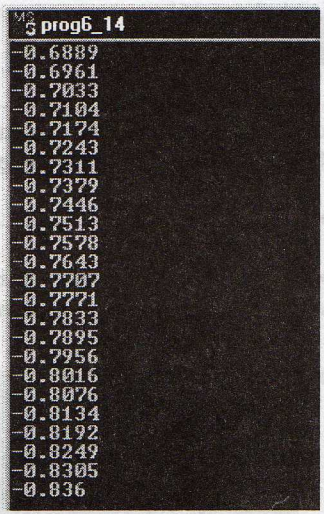
Ćwiczenie 6.14.

Licznik pętli wcale nie musi być dany liczbą całkowitą. Bezładnie wypisz wartości funkcji cosinus w zakresie od 0 do –1, sukcesywnie zmniejszając jej argument o jedną setną:

```
#include <math.h>
...
double x;
cout.precision( 4);
for( x = 0; x >= -1; x = x - 0.01)
{
   cout.width( 5);
   cout << sin( x) << endl;
}
...
```

W tym programie pętla kręci się do tyłu — to znaczy startuje od wartości *0* i dochodzi do wartości *–1*. Jest to możliwe, bo w każdym obrocie zmniejszamy zmienną roboczą o $^1/_{100}$. Ponadto wart uwagi jest drobiazg, że zmienna robocza pętli jest typu rzeczywistego `double`. Widzimy zatem, że w tym przypadku tę zmienną trudno nazywać licznikiem obrotów pętli — wszak liczba obrotów, podobnie jak liczba ptaków w sadzie czy ryb w akwarium, powinna być całkowita. Zmienna `x` określonym krokiem przemierza określony przedział. Do tego też świetnie nadaje się pętla `for()`.

Rysunek 6.14.
Efekt działania programu nie jest zachwycający, bo źle sformatowane dane uciekają z ekranu

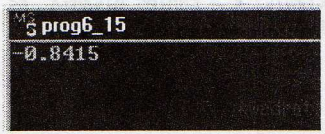


Ćwiczenie 6.15.

Okropny i niestety bardzo częsty błąd z pętlą for() — postaw bezpośrednio za nią średnik:

```
#include <math.h>
...
double x;
cout.precision( 4);
for( x = 0; x >= -1; x = x - 0.01);
{
   cout.width( 5);
   cout << sin( x) << endl;
}
...
```

Rysunek 6.15.
Wynik działania programu z niepotrzebnym średnikiem bezpośrednio za pętlą for()



Czym ten program różni się od poprzedniego? Średnikiem za pętlą `for()`. Ten jeden znak dramatycznie zmienia sens programu, bo pętla wykonuje zadaną liczbę razy obszar programu między sobą a średnikiem, czyli nic. Pętla ileś razy między powietrze, a potem w zwykłym (niepętlowym) reżimie jeden raz wykonuje się to, co uważaliśmy za ciało pętli. Średnik za pętlą jest składniowo poprawny, ale z całą pewnością niepotrzebny.

Pętlę `for()` bardzo lubimy za jej uniwersalność i czytelność. Ktoś oszacował, że 95% wszystkich użyć pętli na warsztacie typowego programisty przypada na pętlę `for()`. Dwóm pozostałym pętlom razem pozostaje zaledwie pięcioprocentowy udział.

Pętla while(...)

Tę instrukcję sterującą można nazwać zubożoną pętlą `for()`. Wszędzie tam, gdzie pracuje pętla `for()`, można zastosować pętlę `while()` i odwrotnie.

Ćwiczenie 6.16.

Napisz program, który będzie pobierał znaki z klawiatury tak długo, dopóki nie wprowadzisz znaku k (ale uwaga — w ćwiczeniu 6.20 niniejszą realizację poddamy krytyce):

```
char c = ' ';
cout << "Klawisz k - koniec pisania" << endl;
while( c != 'k')
{
   cin >> c;
}
...
```

Rysunek 6.16. *Efekt działania programu*

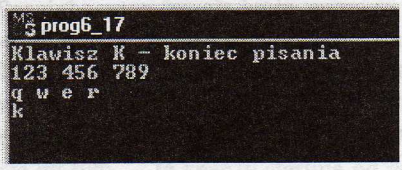
Skoro pętle `while()` i `for()` są kompatybilne, napiszmy to samo inaczej.

Ćwiczenie 6.17.

Napisz ten sam program, ale z wykorzystaniem pętli for() zamiast while():

```
char c = ' ';
cout << "Klawisz k - koniec pisania" << endl;
for( ; c != 'k'; )
{
   cin >> c;
}
...
```

Rysunek 6.17. *Efekt działania programu.*



Te dwa programy działają identycznie, jednak pierwszy jest czytelniejszy, a jego treść daje się jasno wypowiedzieć: wczytuj znaki, dopóki nie uzyskasz znaku *k*. Natomiast pętla `for()` wydaje się konstrukcją nazbyt bogatą do realizacji takiego najprostszego z możliwych powtórzeń. Jej znaczony średnikami, niewykorzystany ustrój bardzo psuje czytelność programu.

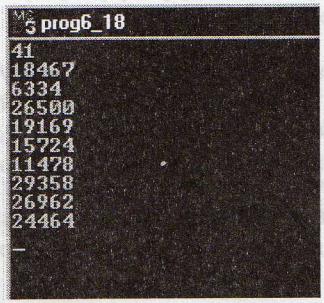
To teraz — dla symetrii — napiszmy program, w którym ładniej (i tylko ładniej) wygląda pętla `for()` niż `while()`:

Ćwiczenie 6.18.

Napisz program, który wylosuje i wypisze na ekranie 10 liczb pseudolosowych:

```
int i;
for( i = 0; i < 10; i ++)
{
   cout << rand() << endl;
}
...
```

Rysunek 6.18. *Efekt działania programu*



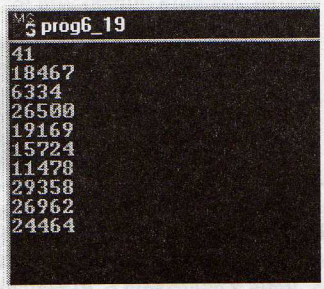
Nie jest to teraz istotne, ale widoczna tutaj funkcja `rand()` dostarcza naturalnych liczb pseudolosowych — to znaczy nie przypadkowych w pełnym tego słowa znaczeniu, ale jednak posianych równomiernie.

Ćwiczenie 6.19.

Napisz ten sam program, ale zastąp pętlę for() pętlą while():

```
int i;
i = 0;
while( i < 10)
{
   cout << rand() << endl;
   i ++;
}
...
```

Rysunek 6.19. *Efekt działania programu*



Program działa identycznie — nawet dostarcza tych samych liczb pseudolosowych! Ale jest mniej czytelny i ma więcej linii, bo zarówno inicjowanie licznika obrotów `i`, jak i jego zwiększanie po każdym obrocie musi być wykonane ręcznie. Pętla `for()` miała w swoim ustroju na taką okazję dwa specjalne pola, pętla `while()` tych pól nie ma.

Na podstawie analizy ćwiczeń 6.16 i 6.17, gdzie pętla `while()` zdawała się mieć przewagę nad pętlą `for()`, i 6.18 oraz 6.19, gdzie wynik rywalizacji był przeciwny, wypowiemy następujące zalecenie:

Pętla `for()` lepiej wygląda w sytuacjach, gdzie należy wykonać *zadaną* liczbę obrotów, przebiec z jakimś krokiem przez określony zbiór wartości (porównaj np. ćwiczenie 6.13). Wtedy bardzo przydają się jej trzy pola, wyraźnie pokazujące, skąd pętla startuje i dokąd oraz w jaki sposób zmierza.

Prostsza pętla `while()` lepiej wygląda w sytuacjach, gdzie *z góry nie wiadomo*, ile obrotów należy wykonać albo przez jaki zbiór należy przebiec. Klasyczną sytuację tego typu zrealizowaliśmy w ćwiczeniu 6.16, gdzie pętla oczekiwała na konkretny klawisz i w żaden sposób nie można było z góry zaplanować, ile obrotów zostanie wykonanych.

Ważniejsza ideowo jest pętla `while()` — pętla `for()` jest jej wyspecjalizowaną realizacją. Pętla `for()` jest za to ładniejsza... Ale głębokich różnic między nimi nie ma.

Pętla do {...} while(...)

A to jest zupełnie inna historia. Oto pętla prawdziwie niezbędna, wnosząca istotnie nową jakość do konstrukcji algorytmów, choć dziwnie rzadko używana. Uwaga na tę pętlę!

Ćwiczenie 6.20.

Powtórz ćwiczenie 6.16, ale zrób to lepiej...

```
char c;
cout << "Klawisz k - koniec pisania" << endl;
do
{
   cin >> c;
}while( c != 'k');
...
```

Rysunek 6.20. *Efekt działania programu*

Dwie poprzednio omówione pętle — funkcjonalnie tożsame, ale oferujące różną estetykę programowania — realizowały polecenie dające się wypowiedzieć tak: *dopóki coś jest prawdą, rób to*. Pętla z ćwiczenia 6.20 realizuje subtelnie inny algorytm postępowania: *rób, dopóki coś jest prawdą*. Na czym polega różnica?

Pętle `while()` i `for()` najpierw testowały warunek logiczny, a potem — gdy warunek był prawdziwy — wykonywały kolejny obrót. Pętla `do{...} while()` najpierw wykonuje obrót, a dopiero potem testuje swój warunek.

Wynika z tego uderzająca właściwość — pętla `do{...} while()`, niezależnie od sytuacji, przynajmniej jeden raz wykona się zawsze.

Strategiczna pomyłka w doborze pętli prowadzi do wielkich komplikacji w algorytmie. A najlepszym kluczem jest odpowiedzenie sobie na pytanie — *czy moja pętla może nie wykonać żadnego obrotu*? Jeśli odpowiedź jest twierdząca, zastosujmy pętlę `while()` lub `for()`, bo te sprawdzają warunek przed obrotem i mogą do niego nie dopuścić. Pętla `do{...} while()` nie nadaje się do takiej sytuacji, bo przynajmniej jeden obrót wykona zawsze.

Dlaczego realizacja 6.20 jest lepsza niż 6.16? Żeby ocenić klawisz, najpierw należy klawisz ten odczytać. Zatem zdecydowanie lepiej pasuje tutaj pętla `do{...} while()`, która najpierw coś robi, a dopiero potem się zastanawia nad kolejnym obrotem.

Konsekwencje pomyłki, o której tutaj wspominałem, widać w programie 6.16:

```
char c = ' ';
while( c != 'k')
{
   cin >> c;
}
```

Ponieważ w tej konstrukcji ocena klawisza następuje *przed* jego wczytaniem ze strumienia `cin`, zmienną `c` należało wstępnie jakoś przygotować, co widać w pierwszej linii, gdzie wpisujemy do niej spację. Lepiej byłoby to napisać tak, by jednak przeczytać znak przed pętlą:

```
char c;
cin >> c;
while( c != 'k')
{
   cin >> c;
}
```

ale przyznajmy, że daleko temu algorytmowi do programu 6.20, który wszystkie te problemy rozwiązuje naturalnymi siłami odpowiedniej instrukcji sterującej.

A wystarczyło zdać sobie sprawę, że aby ocenić klawisz, należy najpierw go poznać.

Instrukcje break i continue

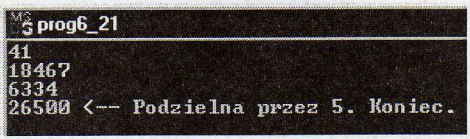
Po niełatwym rozważaniu na temat trzech pętli języka C++ teraz omówimy dwie łatwe instrukcje o charakterze ratunkowym. Z jedną z nich już się spotkaliśmy przy omawianiu zwrotnicy wielotorowej `switch{...}`.

Ćwiczenie 6.21.

Napisz program, który losuje dużo liczb, ale po napotkaniu liczby podzielnej przez 5 kończy pracę:

```
int i, a;
for( i = 0; i < 100; i++)
{
   a = rand();
   cout << a;
   if( a % 5 == 0)
   {
      cout << " <-- Podzielna przez 5. Koniec.";
      break;          //porzucenie pętli
   }
   cout << endl;      //nowa linia
}
...
```

Rysunek 6.21. *Efekt działania programu*



Program operuje pętlą `for()`, jawnie nastawioną na *100* obrotów (pamiętamy, że do takich zastosowań pętla `for()` nadaje się wręcz idealnie). Wewnątrz pętli losują się liczby, a każda z nich jest badana za pomocą warunku logicznego `if()`. Jeśli reszta z dzielenia liczby przez *5* wynosi zero (czyli jeśli liczba dzieli się przez *5*), obroty pętli są nagle i brutalnie przerywane. Program kończy działanie.

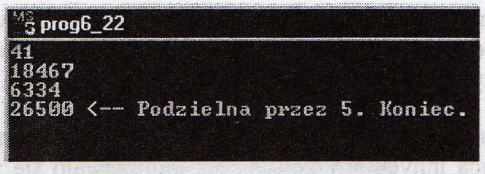
Instrukcja `break` — jak wskazuje jej nazwa — przerywa naturalny bieg programu. Stosuje się ją w pętlach i w zwrotnicy `switch{...}`. O ile jednak w zwrotnicy musimy z niej korzystać (zobacz ćwiczenie 6.6 i następne), w pętlach zawsze możemy sobie poradzić bez niej. Niektórzy nauczyciele programowania rygorystycznie domagają się tego. Powiem tak — jeśli wyeliminowanie instrukcji `break` nie psuje czytelności programu, starajmy się ją eliminować.

Ćwiczenie 6.22.

Przebuduj poprzedni program tak, by wyeliminować instrukcję break:

```
int i = 0, a;
do
{
   a = rand();
   cout << a;
   if( a % 5 == 0)
   {
      cout << " <-- Podzielna przez 5. Koniec.";
   }
   cout << endl;      //nowa linia
   i ++;
} while( i < 100 && a%5 != 0);
...
```

Rysunek 6.22.
Oto ekran wyjściowy



Program ma losować *100* liczb, ale losowanie musi zostać przerwane, gdy pojawi się liczba podzielna przez *5*. Wszystko to oznacza, że powinna zostać zastosowana pętla, która najpierw coś robi (tutaj losuje liczbę), a dopiero potem ocenia i decyduje o powtarzaniu cyklu lub zaniechaniu tego powtarzania.

Tę realizację możemy też wypowiedzieć następująco: tak długo powtarzaj losowanie liczb i ich wypisywanie, dopóki jest ich mniej niż *100* lub równo *100* i żadna nie jest podzielna przez *5*.

Dzięki zastosowaniu odpowiedniej pętli udało się uprościć logikę zagadnienia i wyeliminować wtrącenie `break`. Przyznam jednak, że większość programistów wybrałaby „gorszą" realizację 6.21, dlatego, że jest nieco czytelniejsza.

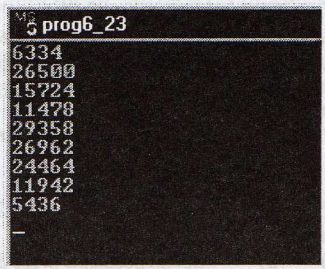
Instrukcja `continue` też występuje w konstrukcjach z pętlą, ale nie przerywa działania pętli, a je przyspiesza.

Ćwiczenie 6.23.

Napisz program, który losuje 20 liczb, ale wyprowadza na ekran tylko parzyste:

```
int i, a;
for( i = 0; i < 20; i++)
{
   a = rand();
   if( a % 2 != 0)    //jeśli liczba nieparzysta
      continue;       //pomiń dalszą część pętli
   cout << a << endl;
}
...
```

Rysunek 6.23.
Efekt działania programu



Program ma losować *20* liczb, zatem do jego napisania wykorzystaliśmy pętlę `for()`, wręcz idealną do takich zastosowań. Wewnątrz pętli losujemy liczbę, ale zanim ją wyprowadzimy na ekran, badamy, czy dzieli się przez *2*. Jeśli nie, czyli jeśli jest nieparzysta, przyspieszamy obrót pętli przez pominięcie tego, co zostało do wykonania, czyli wypisania liczby na ekranie.

Instrukcja `continue` nakazuje porzucenie wykonywania bieżącego obrotu pętli i rozpoczęcie następnego — oczywiście o ile sama pętla ma jeszcze zamiar wykonywać następny obrót.

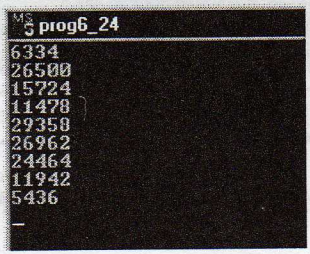
Instrukcję `continue` można wyeliminować z programu równie łatwo, jak `break`, choć za pomocą innych środków. `Break` eliminowało się za pomocą odpowiedniej konstrukcji samej pętli, co nie zawsze było pożądane, bo niekiedy psuło czytelność.

Ćwiczenie 6.24.

Napisz poprzedni program, ale bez instrukcji continue:

```
int i, a;
for( i = 0; i < 20; i++)
{
   a = rand();
   if( a % 2 == 0)          //jeśli liczba parzysta
      cout << a << endl;    //to ją wyprowadź
}
```

Rysunek 6.24. *Efekt działania programu*



Skoro instrukcja `continue` pomija fragment pętli, zazwyczaj wystarcza ten fragment zawrzeć we wnętrzu instrukcji warunkowej — zwrotnicy, która skieruje wykonywanie programu na ten tor tylko wtedy, gdy zaistnieją określone warunki. Nie należy jednak o tej instrukcji zapominać, gdyż znów najważniejszym kryterium jej stosowania jest czytelność programu.

Ćwiczenie 6.25.

Napisz program, który losuje liczby tak długo, dopóki na ekranie nie pojawi się ich 5, ale z wylosowanych wartości wyklucza te, które są podzielne przez 2 lub przez 3:

```
int i = 0, a;
while( true)              //wieczna pętla, bo warunek jest zawsze prawdziwy
{
   a = rand();
   if( a % 2 == 0)        //pomiń podzielne przez 2
      continue;
   if( a % 3 == 0)        //pomiń podzielne przez 3
      continue;
   cout << a << endl;
   i ++;                  //zwiększ licznik poprawnych liczb
   if( i == 5)            //jeśli liczb jest dostatecznie dużo...
      break;              //... to przerwij pętlę
}
...
```

Rysunek 6.25. *Efekt działania programu*

Program stawia nam dość zagmatwane warunki. Po długim zastanowieniu prawdopodobnie udałoby się skonstruować odpowiednią pętlę, ale dzięki instrukcjom `break` i `continue` kolejno i czytelnie wypowiadamy jeden po drugim warunek pomijania dalszej części pętli i ponownego losowania, a także przerwania pętli, gdy liczb uzbierało się wystarczająco dużo.

W tym programie w ogóle zrezygnowaliśmy z możliwości kończenia obrotów pętli za pomocą wbudowanego w nią warunku — wykorzystujemy tutaj tzw. *wieczną pętlę*, która zostawiona sama sobie, działałaby w nieskończoność. Za jej przerwanie odpowiada instrukcja `break`. Oczywiście sprawą zasadniczą jest dokładne sprawdzenie, czy taka wieczna pętla zostanie przerwana, czyli czy przerywnik `break` jest odpowiednio uwarunkowany i w odpowiednim momencie wejdzie do gry.

Czy to jest dobra szkoła programowania? Jak wspominałem, są nauczyciele, którzy tego nie lubią. Jeśli jednak programiście wygodniej jest zorganizować algorytm w taki sposób, w jaki tutaj go opisujemy, należy przystać na to drobne poniechanie purytańskich zasad programowania.

Podsumowanie

Instrukcje sterujące służą do konstruowania algorytmów.

Język C++ oferuje programiście dwie zwrotnice, umożliwiające warunkowe wykonywanie różnych części programu, trzy pętle, realizujące warunkowe powtarzanie określonych partii, i dwie instrukcje pomocnicze, służące do modyfikowania działania pętli.

Wśród instrukcji języka C++ wyróżnia się też niekiedy instrukcję grupującą — parę sześciennych nawiasów, zapewniającą, że dana zwrotnica czy pętla obejmuje nie jedną instrukcję, a całą ich grupę.

Rozdział 7.

Funkcje

Wydzielone i odpowiednio zatytułowane skrawki algorytmów nazwiemy funkcjami. Przykładem takich wydzielonych algorytmów są powszechnie znane i już wielokrotnie tutaj przywoływane funkcje `sin()`, `cos()` czy `rand()`.

Z jedną funkcją spotykamy się od samego początku — to funkcja `main()`, wbudowana na stałe w język C++. Czy możemy jednak wprowadzać do gry swoje własne funkcje, czy też jesteśmy ograniczeni do użytkowania funkcji bibliotecznych, dostarczonych wraz ze środowiskiem? Oczywiście, że możemy deklarować i definiować własne funkcje. Jest to jeden z najważniejszych elementów umiejętności programowania.

Deklarowanie funkcji

Każda funkcja musi być zadeklarowana (zapowiedziana) przed jej pojawieniem się w programie. W rozdziale 5. (porównaj ćwiczenie 5.6) zwracaliśmy uwagę, że w prostych (jednoplikowych) programach funkcje należy deklarować w przestrzeni między frazą dołączania algorytmów biblioteki standardowej a nagłówkiem funkcji `main()`.

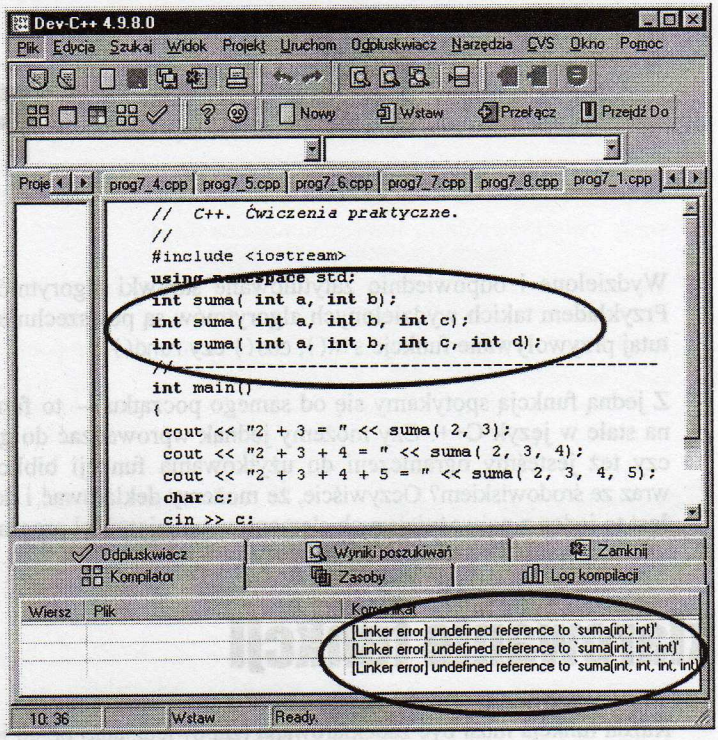
Ćwiczenie 7.1.

Zadeklaruj rodzinę funkcji suma(), obliczającą sumę dwóch, trzech i czterech argumentów typu całkowitego:

Uwaga — ten program jeszcze nie działa.

```
#include <iostream>
using namespace std;
int suma( int a, int b);
int suma( int a, int b, int c);
```

```
int suma( int a, int b, int c, int d);
//-----------------------------------------------
int main()
{
 cout << "2 + 3 = " << suma( 2, 3) << endl;
 cout << "2 + 3 + 4 = " << suma( 2, 3, 4) << endl;
 cout << "2 + 3 + 4 + 5 = " << suma( 2, 3, 4, 5);
 char c;
 cin >> c;
 return 0;
}
```



Rysunek 7.1. *Program nie kompiluje się, bo brakuje w nim definicji (czyli tak zwanych ciał) funkcji*

W przykładzie tym wyraźnie widzimy trzy linie, deklarujące nowe funkcje. W funkcji `main()` z kolei widzimy wywołania tych funkcji. Wywołania te są zgodne z ich zapowiedziami — to znaczy zgadzają się nazwy funkcji i liczby ich argumentów.

Niestety, ten program nie działa. Kompilacja kończy się komunikatami: `Brak ciała funkcji suma( int , int)`, `Brak ciała funkcji suma( int , int, int)`, `Brak ciała funkcji suma( int , int, int, int)`. Środowisko programistyczne nie znalazło algorytmów, które poprawnie zadeklarowaliśmy i poprawnie wywołaliśmy w funkcji `main()`. Wydaje się to oczywiste, bo przecież nigdzie nie zdefiniowaliśmy tych algorytmów. Nie napisaliśmy ustrojów funkcji.

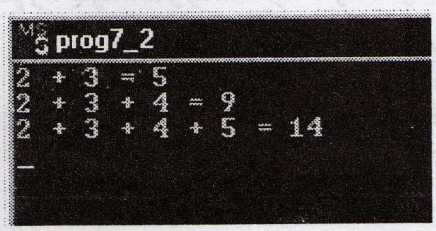
Definiowanie funkcji

Ćwiczenie 7.2.

Uzupełnij poprzedni przykład, definiując ciała funkcji, które zostały tam zadeklarowane:

```
//Tutaj umieść w całości treść poprzedniego przykładu,
//czyli wklejenie nagłówków, udostępnienie biblioteki standardowej,
//deklaracje nowych funkcji i definicję funkcji main()
int suma( int a, int b)
{
 return a + b;
}
//----------------------------------------
int suma( int a, int b, int c)
{
 return a + b + c;
}
//----------------------------------------
int suma( int a, int b, int c, int d)
{
 return a + b + c + d;
}
```

Rysunek 7.2. *Oto wyniki ostatniego programu*



Przykład ten tym różni się od poprzedniego (niedziałającego), że zawiera definicje (czyli ustroje) wszystkich zadeklarowanych tam i użytkowanych funkcji. Jest to kompletny program z funkcjami użytkownika.

Ćwiczenie 7.3.

Zadeklaruj, zdefiniuj i wywołaj funkcję parabola(), oddającą wartość trójmianu kwadratowego dla konkretnych jego współczynników a, b i c:

```
#include <iostream>
using namespace std;
double parabola( double x, double a, double b, double c);
//----------------------------------------
int main()
{
 double a, b, c, x, y;
 cout << "Podaj 3 wspolczynniki paraboli a, b, c: ";
 cin >> a >> b >> c;
 cout << "Podaj x, dla ktorego wyliczyc wartosc paraboli: ";
 cin >> x;
 y = parabola( x, a, b, c);
 cout << a << " * x * x + " << b << " * x + " << c << " = " << y;
 char cc;
```

```
 cin >> cc;
 return 0;
}
//---------------------------------------------
double parabola( double x, double a, double b, double c)
{
 return a * x * x + b * x + c;
}
```

Rysunek 7.3. *Oto wyniki tego programu*

```
prog7_3
Podaj 3 wspolczynniki paraboli a, b, c: 1 2 3
Podaj x, dla ktorego wyliczyc wartosc paraboli: 1
1 * x * x + 2 * x + 3 = 6
_
```

Funkcja `parabola()` przyjmuje 4 argumenty, wylicza matematyczną wartość i oddaje ją programowi głównemu za pomocą frazy `return`. Jednak nie wszystkie funkcje muszą mieć argumenty — wcześniej spotkaliśmy się z bezargumentową funkcją `rand()` — porównaj ćwiczenie 6.18. Nie wszystkie funkcje muszą też wyliczać i oddawać programowi jakiekolwiek wartości.

Ćwiczenie 7.4.

Zadeklaruj, zdefiniuj i wywołaj funkcję wyprowadzającą Twoje dane osobowe:

```
#include <iostream>
using namespace std;
void twoje_dane( void);
//---------------------------------------------
int main()
{
 twoje_dane();
 cout << endl << endl;
 twoje_dane();
 char c;
 cin >> c;
 return 0;
}
//---------------------------------------------
void twoje_dane( void)
{
 cout << "Imie i nazwisko: Jan Kowalski" << endl;
 cout << "Wiek           : 23" << endl;
 cout << "Zawod          : Programista AI" << endl;
}
```

Rysunek 7.4. *Ten program generuje takie oto informacje*

```
prog7_4
Imie i nazwisko: Jan Kowalski
Wiek           : 23
Zawod          : Programista AI

Imie i nazwisko: Jan Kowalski
Wiek           : 23
Zawod          : Programista AI
_
```

Funkcje szczególnie warto wyodrębniać wtedy, gdy będziemy je wielokrotnie wywoływać — tak jak w tym programie dwukrotnie wywołaliśmy funkcję podającą dane osobowe, zamiast każdorazowo przytaczać odpowiedni fragment algorytmu.

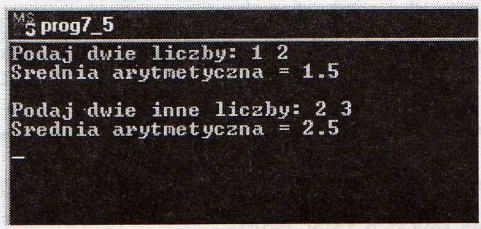
Funkcje wyodrębniamy też wtedy, gdy za pomocą tego samego algorytmu wyliczamy wiele różnych wartości.

Ćwiczenie 7.5.

Zadeklaruj, zdefiniuj i wywołaj funkcję wyliczającą średnią arytmetyczną dwóch podanych liczb:

```
#include <iostream>
using namespace std;
double srednia( double a, double b);
//-----------------------------------------------
int main()
{
 double q, w;
 cout << "Podaj dwie liczby: ";
 cin >> q >> w;
 cout << "Srednia arytmetyczna = " << srednia( q, w) << endl << endl;
 cout << "Podaj dwie inne liczby: ";
 cin >> q >> w;
 cout << "Srednia arytmetyczna = " << srednia( q, w);
 char c;
 cin >> c;
 return 0;
}
//-----------------------------------------------
double srednia( double a, double b)
{
 return (a + b) / 2;
}
```

Rysunek 7.5.
Oto ekran wyjściowy



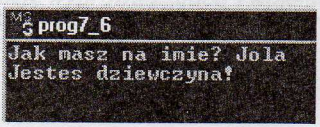
Ćwiczenie 7.6.

Zadeklaruj, zdefiniuj i wywołaj funkcję oceniającą płeć osoby o podanym imieniu (patrz też ćwiczenie 6.5):

```
#include <iostream>
#include <string>
using namespace std;
bool czy_dziewczyna( string imie);
//-----------------------------------------------
```

```
{
 string txt;
 cout << "Jak masz na imie? ";
 cin >> txt;
 if( czy_dziewczyna( txt))
 {
    cout << "Jestes dziewczyna!";
 }
 else
 {
    cout << "Jestes chlopakiem?";
 }
 char c;
 cin >> c;
 return 0;
}
//-----------------------------------------------
bool czy_dziewczyna( string imie)
{
 char ostatni_znak = imie[ imie.size() - 1];
 if( ostatni_znak == 'a')
    return true;       //dziewczyna
 else
    return false;      //chlopak
}
```

Rysunek 7.6.
Oto ekran wyjściowy



Ta funkcja jako argument otrzymuje od programu nadrzędnego tekst, bada ostatni jego znak i na tej podstawie zgaduje, czy ma do czynienia z imieniem żeńskim. Jeśli tak, zwraca wartość typu logiczną (czyli typu `bool`) równą `true`.

Podstawową zaletą wyodrębniania algorytmów w oddzielne funkcje jest możliwość wielokrotnego ich wywoływania z różnymi wartościami argumentów. Ale zdarza się też wprowadzać do gry funkcję tylko dlatego, że podnosi to czytelność programu.

Ćwiczenie 7.7.

Usprawnij pytania o imię i nazwisko, wprowadzając do programu odpowiednie funkcje:

```
string spytaj_o_imie( void);
string spytaj_o_nazwisko( void);
//-----------------------------------------------
int main()
{
 string si, sn;
 si = spytaj_o_imie();
 sn = spytaj_o_nazwisko();
 cout << "Nazywasz sie " << si << " " << sn;
 char c;
 cin >> c;
 return 0;
}
```

```
//-----------------------------------------
string spytaj_o_imie( void)
{
 string imie;
 cout << "Jak masz na imie? ";
 cin >> imie;
 return imie;
}
//-----------------------------------------
string spytaj_o_nazwisko( void)
{
 string nazwisko;
 cout << "Jak masz na nazwisko? ";
 cin >> nazwisko;
 return nazwisko;
}
```

Rysunek 7.7.
Oto ekran wyjściowy

Dwie widoczne w powyższym fragmencie funkcje prawdopodobnie wyodrębniono tylko po to, by podnieść czytelność zapisu we wnętrzu funkcji `main()`. Zamiast definiować funkcje, można by wprost przytoczyć ich ciała, bezpośrednio w funkcji `main()`.

Ćwiczenie 7.8.

Zadeklaruj, zdefiniuj i wywołaj funkcję o nazwie stop(), która poradzi sobie z problemem znikającej konsoli, opisanym w ćwiczeniu 3.1:

```
void stop( void);
//-----------------------------------------
int main()
{
 cout << "Funkcja stop() zatrzyma konsole na ekranie...";
 stop();
 return 0;
}
//-----------------------------------------
void stop( void)
{
 char c;
 cin >> c;
}
```

Rysunek 7.8.
Oto ekran wyjściowy

Przykład ten dobitnie ilustruje, że warto też wprowadzać do gry funkcje wywoływane tylko jeden raz, ale za to podnoszące czytelność programu. Możemy je nazwać ładnymi opakowaniami brzydkiego kodu.

Argumenty funkcji i referencja

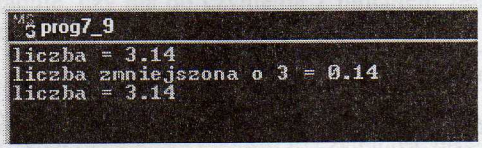
Teraz dokładniej zbadajmy zagadnienie argumentów funkcji, czyli danych przekazywanych do jej wnętrza.

Ćwiczenie 7.9.

Napisz prosty przykład z jednoargumentową funkcją, która wypisuje wartość swojego rzeczywistego argumentu na ekranie. Przykład ten za chwilę posłuży do dalszych testów (uwaga — przytaczamy tylko kluczowe fragmenty algorytmów):

```
void zmniejsz_o_3_i_wypisz( double a);
//-----------------------------------------------
int main()
{
 double r = 3.14;
 cout << "liczba = " << r << endl;
 zmniejsz_o_3_i_wypisz( r);
 cout << "liczba = " << r;
 char c;
 cin >> c;
 return 0;
}
//-----------------------------------------------
void zmniejsz_o_3_i_wypisz( double a)
{
 a = a - 3;
 cout << "liczba zmniejszona o 3 = " << a << endl;
}
```

Rysunek 7.9.
Oto ekran wyjściowy



Dokładnie przeanalizujmy ten i następny przykład, bo za chwilę dowiemy się czegoś bardzo ważnego o argumentach funkcji. W funkcji `main()` najpierw wypisujemy wartość zmiennej `r` i na ekranie otrzymujemy oczekiwane *3.14*. Potem wywołujemy funkcję, która jako argument otrzymuje tę zmienną, ma ją zmniejszyć o *3* i znów wypisać na ekranie. Na ekranie otrzymujemy oczekiwane *0.14*. Potem — po zakończeniu pracy funkcji — ponownie wypisujemy na ekranie zmienną `r` i dostajemy nie *0.14*, a *3.14*.

Co w tym zagadkowego? Zagadkowy jest tu *mechanizm ochrony danych*, przekazywanych do wnętrza funkcji. Nasza funkcja przerobiła daną z wartości *3.14* na *0.14*, a mimo tego stan zmiennej `r` przed i po wywołaniu funkcji nie zmienił się.

Funkcje *kopiują* sobie argumenty, a nie pracują na oryginałach. Wartości argumentów wewnątrz funkcji mogą być modyfikowane, ale to nie wpłynie na ich wartości poza funkcją.

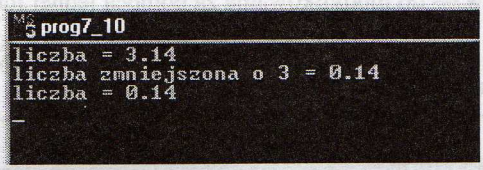
Proces ten może jednak przebiegać zupełnie inaczej...

Ćwiczenie 7.10.

Zmodyfikuj poprzedni przykład tak, by funkcja otrzymała argument w procesie referencji:

```
void zmniejsz_o_3_i_wypisz( double &a);
//-----------------------------------------------
int main()
{
 double r = 3.14;
 cout << "liczba = " << r << endl;
 zmniejsz_o_3_i_wypisz( r);
 cout << "liczba = " << r;
 char c;
 cin >> c;
 return 0;
}
//-----------------------------------------------
void zmniejsz_o_3_i_wypisz( double &a)
{
 a = a - 3;
 cout << "liczba zmniejszona o 3 = " << a << endl;
}
```

Rysunek 7.10.
Oto ekran wyjściowy



Trudno dostrzec, na czym polega różnica, ale ta funkcja nie pracowała *na kopii* danej r, a na jej *oryginale*. Świadczy o tym fakt, że zmniejszenie wartości danej o *3* obowiązuje nie tylko we wnętrzu funkcji, a w całym programie poniżej tego wywołania.

Funkcja pracuje na oryginałach swoich argumentów (a nie na ich kopiach), gdy argumenty są przekazywane referencyjnie, czyli gdy przy ich nazwach w deklaracji i nagłówku definicji znajduje się symbol &.

Ćwiczenie 7.11.

Napisz funkcję, która otrzyma poprzez referencję kilka argumentów i je wyzeruje:

```
void wyzeruj( int &a, int &b, int &c);
//-----------------------------------------------
int main()
{
 int i, j, k;
 cout << "Przed wywolaniem funkcji: i = " << i << ", j = " << j << ", k = " << k <<
endl;
 wyzeruj( i, j, k);
 cout << "Po wywolaniu funkcji: i = " << i << ", j = " << j << ", k = " << k;
 char c;
 cin >> c;
 return 0;
}
```

```
//-----------------------------------------
void wyzeruj( int &a, int &b, int &c)
{
 cout << "Zerowanie argumentow wewnatrz funkcji ..." << endl;
 a = b = c = 0;
}
```

Rysunek 7.11. *Oto ekran wyjściowy*

```
prog7_11
Przed wywolaniem funkcji: i = 4370436, j = 4370432, k = 4198592
Zerowanie argumentow wewnatrz funkcji ...
Po wywolaniu funkcji: i = 0, j = 0, k = 0
```

W pierwszej linii otrzymujemy wartości zmiennych bezpośrednio po ich zadeklarowaniu. Ponieważ język C++ nie inicjuje zmiennych (powiemy o tym więcej w następnym rozdziale), w zadeklarowanych zmiennych znajdują się po prostu śmieci. W następnej linii zmienne te są *przekazywane przez referencję* do wnętrza funkcji. Oznacza to, że funkcja pracuje na oryginałach, nie kopiach zmiennych i wyzerowanie wartości ma szerszy zasięg niż tylko ciało funkcji.

Przykład ten ilustruje to, do czego zazwyczaj używa się referencyjnego przekazu argumentów — do modyfikowania za pomocą jednej funkcji większej ich liczby.

Łatwo też domyślić się, że referencja jest także szybsza niż kopiowanie, ale zazwyczaj różnica jest na tyle mała, że dotyczy ekstremalnych zastosowań programowania.

Ćwiczenie 7.12.

Postąpimy źle. Napisz funkcję kwadrat(), która otrzyma przez referencję argument rzeczywisty i kanałem referencyjnym zwróci wartość drugiej potęgi swojego argumentu:

```
void kwadrat( double &a);
//-----------------------------------------
int main()
{
 double a;
 cout << "Podaj liczbe: ";
 cin >> a;
 kwadrat( a);
 cout << "Kwadrat Twojej liczby wynosi " << a;
 char c;
 cin >> c;
 return 0;
}
//-----------------------------------------
void kwadrat( double &a)
{
 a = a * a;
}
```

Rysunek 7.12. *Oto ekran wyjściowy*

```
prog7_12
Podaj liczbe: 4
Kwadrat Twojej liczby wynosi 16
```

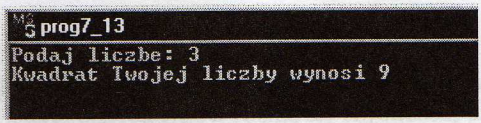
Ten program — choć działa dobrze — prawdopodobnie nadużywa referencji. Nie izoluje wnętrza funkcji, nie zmusza jej do wykonania kopii argumentu, nie wykorzystuje też naturalnej możliwości zwracania wartości przez funkcję. Jednak formalnie nie można mu niczego zarzucić.

Ćwiczenie 7.13.

Napisz funkcję kwadrat(), która otrzyma argument rzeczywisty i w naturalny sposób zwróci wartość drugiej potęgi swojego argumentu:

```
double kwadrat( double a);
//-----------------------------------------------
int main()
{
 double a;
 cout << "Podaj liczbe: ";
 cin >> a;
 cout << "Kwadrat Twojej liczby wynosi " << kwadrat( a);
 char c;
 cin >> c;
 return 0;
}
//-----------------------------------------------
double kwadrat( double a)
{
 return a * a;
}
```

Rysunek 7.13.
Oto ekran wyjściowy



Naturalnym kanałem, którym funkcja zwraca pojedynczą wartość, jest zakończenie jej ciała frazą `return` — oddaj. Tak skonstruowane funkcje mogą być wbudowywane w różne wyrażenia. Natomiast kanały referencyjne są usprawiedliwione wtedy, gdy danych do przekazania jest dużo — jak np. w ćwiczeniu 7.10 — albo gdy przekaz argumentów do ciała funkcji musi być ekstremalnie szybki.

Podsumowanie

Język C++ umożliwia wyodrębnianie fragmentów algorytmów, zwanych funkcjami.

Każda funkcja ma określoną *nazwę*, *typ zwracanego rezultatu* (`void`, gdy nie zwraca rezultatu — porównaj ćwiczenie 7.4) oraz *typy argumentów*, czyli danych wejściowych dla swojego algorytmu (`void`, gdy nie ma argumentów).

Argumenty mogą być przekazywane do funkcji na dwa sposoby — jako *kopie* i jako *oryginały* (zwane *referencjami*). Kopie są bezpieczniejsze, referencje często wygodniejsze, szczególnie, gdy funkcja musi zmodyfikować wiele różnych zmiennych albo gdy musi być ekstremalnie szybka i nie powinna tracić czasu na kopiowanie swoich argumentów.

Rozdział 8.
Dane

Omówione w dwóch poprzednich rozdziałach algorytmy zawsze operują na zbiorze jakichś wartości — liczbach, tekstach, znakach... Program komputerowy składa się z *algorytmów* operujących na *danych*.

Typy danych

Typem danej określamy jej rodzaj — na przykład, że jest to dana całkowitoliczbowa lub liczba z kropką dziesiętną. Dane różnych typów różnią się szybkością działania, rozmiarem zajmowanej pamięci i oczywiście funkcjonalnością, np. są dane, które wolno mnożyć przez siebie, ale i takie, dla których operacja mnożenia nie istnieje — choćby teksty.

Istotą języka C++ jest możliwość definiowania własnych typów danych i to o nieograniczonej różnorodności. Typy te to tak zwane *klasy* i bez trudu możemy sobie wyobrazić typ mapy bitowej, modelu rekina czy próbki dźwiękowej.

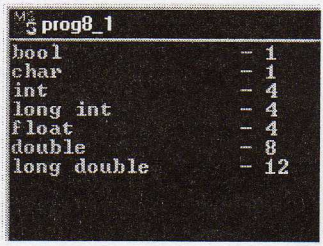
Język C++ wiele typów ma wbudowanych w swoją strukturę — są to tak zwane *typy proste*. Zbadajmy ich podstawowy zestaw.

Ćwiczenie 8.1.

Napisz program, który wyprowadzi informację o tym, ile pamięci zajmuje zmienna konkretnego typu wbudowanego:

```
cout << "char               - " << sizeof( char) << endl;
cout << "int                - " << sizeof( int) << endl;
cout << "long int           - " << sizeof( long int) << endl;
cout << "float              - " << sizeof( float) << endl;
cout << "double             - " << sizeof( double) << endl;
cout << "long double        - " << sizeof( long double) << endl;
...
```

Rysunek 8.1.
Efekt działania programu zależy od kompilatora, a oto wyniki dla środowiska DEV



W programie pojawił się operator `sizeof()` — ile bajtów zajmuje zmienna danego typu. Widzimy z powyższego ćwiczenia, że są zmienne małe w sensie zajętości pamięci, zatem zapewne szybkie w działaniu, są też duże i prawdopodobnie znacznie cięższe w operowaniu. Nie możemy jednak patrzeć tylko na wielkość zmiennej i szybkość jej działania, bo równie ważna jest jej specyfika, funkcjonalność. Zmienna typu `long double` umożliwia przeprowadzanie bardzo dokładnych rachunków w dziedzinie rzeczywistej, gdy zmienna `unsigned char` zliczy wartości całkowite zaledwie od *0* do *255*.

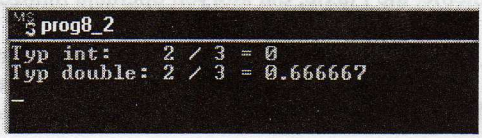
Poważnym zadaniem jest przemyślenie, jakiego typu zmiennej akurat użyć, by pogodzić dwa przeciwieństwa — wymaganą funkcjonalność i dokładność z szybkością działania i optymalizacją wykorzystania pamięci.

Ćwiczenie 8.2.

Napisz program, który wyliczy wartość ułamka $^2/_3$, korzystając z typu całkowitoliczbowego i rzeczywistego:

```
int a = 2, b = 3;
cout << "Typ int:    2 / 3 = " << a / b << endl;
double u = 2, v = 3;
cout << "Typ double: 2 / 3 = " << u / v << endl;
...
```

Rysunek 8.2.
Efekt działania programu



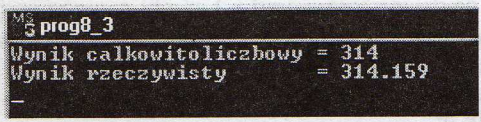
Widzimy z powyższego, że typ całkowitoliczbowy zawiódł — nie mając możliwości operowania ułamkiem, wynik został przybliżony w dół. Nieuwzględnienie takich efektów prowadzi do trudnych do wykrycia błędów rachunkowych.

Ćwiczenie 8.3.

Napisz program, który wyliczy pole koła o promieniu 10 metrów, operując danymi typu int i typu double:

```
#include <math.h>      //zawiera deklaracje stalej M_PI
...
int ir = 10, ipole;
ipole = M_PI * ir * ir;
cout << "Wynik calkowitoliczbowy = " << ipole << endl;
```

```
double dr = 10, dpole;
dpole = M_PI * dr * dr;
cout << "Wynik rzeczywisty       = " << dpole << endl;
...
```



Rysunek 8.3. *Efekt działania programu*

Obliczenia całkowitoliczbowe — choć szybsze — z oczywistych powodów bardzo często zawodzą.

Deklarowanie i inicjowanie prostych danych

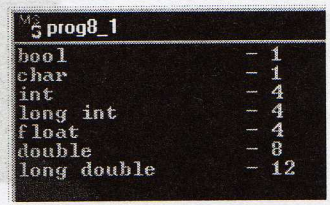
Typy to wzorce danych. Typy są po to, by deklarować dane o określonym wzorcu. Kompilator języka C++ musi być poinformowany, co kryje się pod każdym napisem nie będącym słowem kluczowym języka. Taki proces zgłaszania nazw nazywa się deklarowaniem. W języku C++ wszystkie dane muszą być wstępnie zadeklarowane.

Ćwiczenie 8.4.

Wróć do ćwiczenia 8.1 i zadeklaruj po jednej danej różnego typu oraz wyprowadź ich rozmiar:

```
bool b;
char c;
int i;
long int li;
float f;
double d;
long double ld;

cout << "bool             - " << sizeof( b) << endl;
cout << "char             - " << sizeof( c) << endl;
cout << "int              - " << sizeof( i) << endl;
cout << "long int         - " << sizeof( li) << endl;
cout << "float            - " << sizeof( f) << endl;
cout << "double           - " << sizeof( d) << endl;
cout << "long double      - " << sizeof( ld) << endl;
...
```



Rysunek 8.4. *Efekt działania programu jest taki sam, jak programu 8.1. Zauważmy jednak, że tam nie deklarowaliśmy zmiennych i nie pytaliśmy o ich rozmiary, a o rozmiary samych typów*

Ćwiczenie 8.5.

Wróć do poprzedniego ćwiczenia i wyprowadź nie rozmiary zmiennych, a ich wartości:

```
bool b;
char c;
int i;
long int li;
float f;
double d;
long double ld;

cout << "c   = " << c << endl;
cout << "i   = " << i << endl;
cout << "li  = " << li << endl;
cout << "f   = " << f << endl;
cout << "d   = " << d << endl;
cout << "ld  = " << ld << endl;
...
```

Rysunek 8.5. *Efekt działania programu*

```
prog8_5
b   = 0
c   = @
i   = 7798296
li  = 2013306216
f   = 1.09277e-038
d   = 4.24399e-314
ld  = -1.#IND
```

Przykra wiadomość — język C++ nie inicjuje danych o typach wbudowanych. Deklarując dane o typach stworzonych przez nas (czyli tak zwane obiekty), znajdziemy na ten mankament sposób. Jednak dane o typach wbudowanych przed użyciem muszą być zainicjowane, bo po procesie deklaracji znajdują się w nich śmieci.

Ćwiczenie 8.6.

Wróć do poprzedniego ćwiczenia i zadeklaruj oraz zainicjuj dane:

```
bool b = true;
char c = 'a';
int i = 123;
long int li = -123456L;
float f = -123.456;
double d = -12.3e-11;
//dalsza część programu bez zmian
```

Rysunek 8.6. *Efekt działania programu po poprawkach*

```
prog8_6
b   = 1
c   = a
i   = 123
li  = -123456
f   = -123.456
d   = -1.23e-010
```

Język C++ umożliwia inicjowanie wartości danych już w momencie ich deklaracji. Należy z tej możliwości korzystać tak często, jak tylko się da.

Zwróćmy też uwagę na specyficzne szczegóły — wprowadzanie wartości znakowej w apostrofach, długiej całkowitej za pomocą literki *L* i typów rzeczywistych za pomocą klasycznej notacji przecinkowej i notacji wykładniczej (zwanej też inżynierską). Mówimy, że każdy typ ma określone literały inicjujące, czyli sposób zadawania wartości danych.

Deklarowanie i inicjowanie danych tablicowych

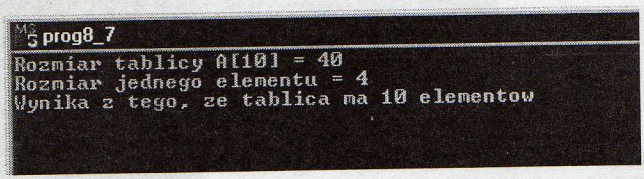
Tablicą nazwiemy zespół danych określonego typu, dostępnych za pośrednictwem indeksu (numeru). Tablicę zmiennych określonego typu — np. typu `int` — deklarujemy w charakterystyczny sposób, podając w kwadratowych nawiasach jej rozmiar.

Ćwiczenie 8.7.

Zadeklaruj tablicę 10 liczb całkowitych i wyprowadź na ekran rozmiary tablicy i jej pierwszego elementu:

```
int i, A[ 10];
cout << "Rozmiar tablicy A[10] = " << sizeof( A) << endl;
cout << "Rozmiar jednego elementu = " << sizeof( A[ 0]) << endl;
cout << "Wynika z tego, ze tablica ma " << sizeof( A) / sizeof( A[0]) << " elementow";
...
```

Rysunek 8.7. *Wynik działania tego programu ilustruje, że tablica jest ciągiem danych określonego typu, ułożonych w pamięci jedna za drugą. Rozmiar tablicy jest sumą rozmiarów wszystkich jej elementów*



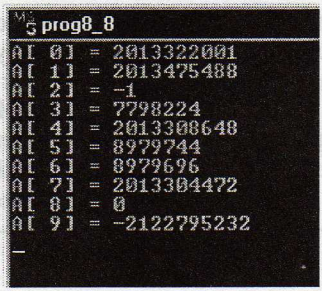
Zapamiętajmy też koniecznie, że pierwszy element tablicy ma indeks *0*, nie jeden. To samo inaczej — ostatni element tej tablicy ma indeks *9*, nie *10*.

Ćwiczenie 8.8.

Zadeklaruj tablicę 10 liczb całkowitych i wyprowadź jej wartości na ekran:

```
int i, A[ 10];
for( i = 0; i < 10; i ++)
{
   cout << "A[ " << i << "] = " << A[ i] << endl;
}
...
```

Rysunek 8.8. *Efekt działania tego programu*



Zapamiętajmy koniecznie, że skoro pierwszy element tablicy ma indeks *0*, zatem w tym przypadku ostatni ma indeks *9*, nie *10*.

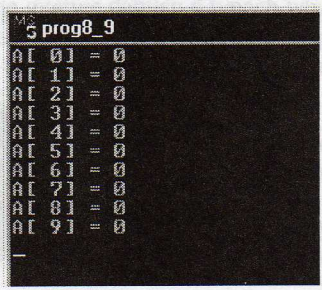
Dlaczego ta tablica zawiera śmieci? Powód jest ten sam, co w ćwiczeniu 8.5 — mianowicie język C++ *nie inicjuje* wartości swoich zmiennych podczas ich deklaracji.

Ćwiczenie 8.9.

Zadeklaruj tablicę 10 liczb całkowitych, wyzeruj ją i wyprowadź jej wartości na ekran:

```
int i, A[ 10] = {0,0,0,0,0,0,0,0,0,0};
for( i = 0; i < 10; i ++)
{
   cout << "A[ " << i << "] = " << A[ i] << endl;
}
...
```

Rysunek 8.9. *Efekt działania tego programu*



W programie tym widzimy tzw. *inicjator tablicowy* — sekwencję oddzielonych przecinkami wartości, ujętych w nawiasy sześcienne. Wartości w nawiasach nie może być więcej niż zadeklarowany rozmiar tablicy. Może natomiast być ich mniej — końcowe elementy nie będą zainicjowane.

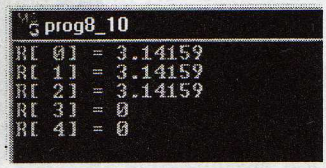
Ćwiczenie 8.10.

Zadeklaruj tablicę 5 liczb rzeczywistych i trzy pierwsze pola zainicjuj wartościami pi. Co zawierają pozostałe pola?

```
int i;
double R[ 5] = {M_PI, M_PI, M_PI};
for( i = 0; i < 5; i ++)
```

```
{
   cout << "R[ " << i << "] = " << R[ i] << endl;
}
...
```

Rysunek 8.10. *Efekt działania tego programu*



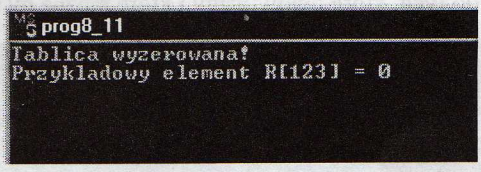
Trzy pierwsze wartości tablicy, zgodnie z warunkami zadania, zostały zainicjowane wartością stałej *pi*. Dwa pozostałe pola zawierają wartości przypadkowe, akurat tutaj traf chciał, że są to zera. Nigdy jednak nie oczekujmy, że niezainicjowane dane będą miały wartość *0*.

Ćwiczenie 8.11.

Zadeklaruj tablicę 1000 liczb rzeczywistych i wstępnie ją wyzeruj:

```
int i;
double R[ 1000];
for( i = 0; i < 1000; i ++)
{
   R[ i] = 0;
}
cout << "Tablica wyzerowana!" << endl;
cout << "Przykladowy element R[123] = " << R[123];
...
```

Rysunek 8.11. *Efekt działania tego programu*



W tym programie demonstrujemy proces *programowego inicjowania tablicy*. Tablica jest na tyle duża, że nie ma sensu bezpośrednie inicjowanie jej w taki sposób, jak w ćwiczeniu 8.9.

Ćwiczenie 8.12.

Zadeklaruj tablicę 5 tekstów, zainicjuj ją imionami swoich przyjaciół i wyprowadź jej wartości na ekran:

```
#include <string>
...
int i;
string txt[ 5] = {"Zosia", "Janek", "Jurek"};
for( i = 0; i < 5; i ++)
{
   cout << "txt[ " << i << "] = " << txt[ i] << endl;
}
...
```

Rysunek 8.12.
Efekt działania tego programu

Zmienna o nazwie `txt` jest tablicą typu `string`, liczącą pięć elementów. Trzy pierwsze elementy zostały zainicjowane za pomocą inicjatora tablicowego. Pozostałe elementy nie są inicjowane. Ale akurat w tym wypadku elementy niezainicjowane nie zawierają śmieci — typ `string` przy braku innego inicjowania jest inicjowany tekstem pustym. Czy to jest jakiś wyłom w strukturze języka C++, który nie inicjuje danych typów prostych? Nie, bo typ `string` nie jest typem prostym. Jest to typ dostarczony do języka wraz z *biblioteką standardową*.

Deklarowanie i inicjowanie danych wskaźnikowych

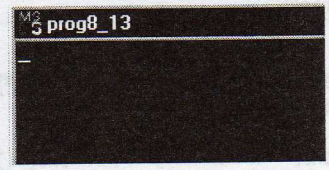
Wskaźniki, w pewnym uproszczeniu zwane adresami, stanowią zespół danych alternatywny do tych, które omówiliśmy i które można by nazwać danymi klasycznymi.

Ćwiczenie 8.13.

Napisz program, w którym zadeklarujesz kilka wskaźników dla zmiennych podstawowych typów:

```
int main()
{
 int *i_adr, *j_adr, i, j;
 double *r_adr;
 string *txt_adr;
 char c;
 cin >> c;
 return 0;
}
```

Rysunek 8.13.
Ten program nie generuje żadnych danych wyjściowych



Wskaźniki deklarujemy prawie tak samo, jak zmienne typów klasycznych. Jedyną różnicą jest gwiazdka postawiona przed nazwą zmiennej wskaźnikowej. W tym programie w pierwszej linii zadeklarowaliśmy dwa wskaźniki dla zmiennych typu całkowitego i dwie zmienne typu całkowitego. Potem deklarujemy jeszcze wskaźnik dla zmiennej typu rzeczywistego i zmienną typu tekstowego.

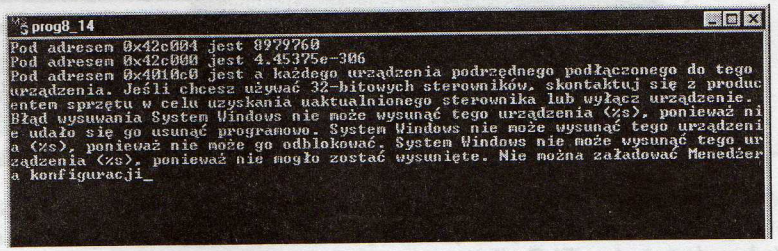
Być może warto w nazwie zmiennej zaznaczać, że jest to wskaźnik — np. tak jak tutaj — do nazw dodając przyrostek *adr*.

Podczas operacji na wskaźnikach kluczową umiejętnością jest odpowiedź na pytanie, jaka wartość znajduje się pod danym wskaźnikiem. Jest to umiejętność przejścia od świata adresów do świata zmiennych.

Ćwiczenie 8.14.

Napisz program, w którym zademonstrujesz, jak wyciąga się wartość zmiennej spod wskaźnika (spod adresu):

```
int main()
{
 int *i_adr;
 double *r_adr;
 string *txt_adr;
 cout << "Pod adresem " << i_adr << " jest " << *i_adr << endl;
 cout << "Pod adresem " << r_adr << " jest " << *r_adr << endl;
 cout << "Pod adresem " << txt_adr << " jest " << *txt_adr << endl;
 char c;
 cin >> c;
 return 0;
}
```



Rysunek 8.14. *Ten program może (ale nie musi) działać źle. Jeśli generuje bardzo dużo danych, usuń trzecią linię cout, wyprowadzającą nieznany (niezainicjowany) tekst spod wskaźnika txt_adr*

Tutaj dowiadujemy się, jak zadaje się kluczowe pytanie: „co jest pod wskaźnikiem adr?".

Wskaźnik opatrzony gwiazdką jest wartością znajdującą się pod tym wskaźnikiem.

To samo inaczej — operator * wyłuskuje wartość spod adresu.

To samo jeszcze inaczej — operator * zapewnia przejście od świata adresów do świata zmiennych.

W powyższym przykładzie opatrujemy gwiazdką wcześniej zadeklarowane wskaźniki, zatem w wyniku operacji *wyłuskania* otrzymujemy *wartości* znajdujące się pod tymi *adresami*. Ponieważ jednak język C++ niczego nie inicjuje, wskaźniki są przypadkowe

(wskazują nieznany punkt w przestrzeni pamięciowej Twojego komputera) i znajdują się pod nimi różne śmieci. W szczególności kłopotliwe może być wyłuskanie tekstu spod wskaźnika tekstowego — może się okazać, że jest tam ciąg znaków o długości milionów liter... Miliony znaków przemkną przez Twoją konsolę!

Ćwiczenie 8.15.

Napisz katastrofalnie ryzykowny program, który zmodyfikuje zmienną, znajdującą się pod niezainicjowanym wskaźnikiem:

```
//Uwaga — ten program prawdopodobnie zawiesi komputer...
int main()
{
 char c;
 cout << "Uwaga - ten program prawdopodobnie zawiesi komputer ..." << endl;
 cout << "Kontynuowac? T - tak ";
 cin >> c;
 if( c != 't' && c != 'T')
    return 0;
 int *i_adr;
 double *r_adr;
 string *txt_adr;
 *i_adr = 0;       //???
 *r_adr = 0;       //???
 *txt_adr = "ABC";   //???
 ...
}
```

Rysunek 8.15. *Ten program najpierw wyświetla ostrzeżenie — użytkownik uruchamia go na własną odpowiedzialność*

Język C++ w momencie deklaracji nie inicjuje wskaźników, zatem widoczne tutaj trzy wskaźniki wskazują cokolwiek. Jednak to cokolwiek może oznaczać, że takie wskaźniki wskazują żywotne dla stabilności systemu parametry — ot choćby liczbę dysków czy ich bieżący spis treści. Wpisanie w to miejsce naszych wartości przynosi dramatyczne rezultaty.

Nigdy nie operuj niezainicjowanym wskaźnikiem!

Ćwiczenie 8.16.

Zadeklaruj i zainicjuj wskaźniki:

```
...
int *i_adr = new int;
double *r_adr = new double[ 10];
string *txt_adr = new string;
*i_adr = 0;
*r_adr = 0;
*txt_adr = "ABC";
cout << "Pod adresem " << i_adr << " jest " << *i_adr << endl;
cout << "Pod adresem " << r_adr << " jest " << *r_adr << endl;
cout << "Pod adresem " << txt_adr << " jest " << *txt_adr;
...
```

Rysunek 8.16.
Program generuje takie oto wyniki (adresy mogą być inne)

Wskaźniki inicjuje się za pomocą operatora new. Operator ten *przenosi* wskaźnik spod przypadkowego adresu pod taki, gdzie jest wolna pamięć na zmienną.

Operator new może zainicjować wskaźnik pod cały zestaw danych (czyli pod tablicę) określonego typu. W drugiej linii prosimy operator new o przygotowanie wskaźnika pod tablicę *10* zmiennych typu rzeczywistego.

W powyższym programie jest jednak poważny i wstydliwy błąd, zwany **wyciekiem pamięci**.

Ćwiczenie 8.17.

Z poprzedniego programu usuń niezmiernie wstydliwy błąd wycieku pamięci:

```
...
int *i_adr = new int;
double *r_adr = new double[ 10];
string *txt_adr = new string;
...
delete i_adr;
delete [] r_adr;
delete txt_adr;
...
```

Rysunek 8.17.
Program generuje takie same wyniki, choć adresy mogą już być inne (operatory new znalazły wolną pamięć w innym miejscu mapy pamięci)

Zawsze, gdy używamy operatora przydzielającego pamięć, na koniec programu musimy użyć operatora zwalniającego tę pamięć. Jeśli zapomnimy o operatorze delete w przypadku pojedynczej komórki pamięci lub delete[] w przypadku tablicy komórek, zubażamy zasoby pamięciowe systemu operacyjnego. Sytuacja wróci do normy dopiero po restarcie komputera.

Niektóre języki same zwalniają niepotrzebną programowi pamięć — nie ma w nich operatora delete. Zwalnianiem zajmuje się wydzielony podprogram, stale pracujący w tle. Język C++ nie skorzystał z tego rozwiązania, bo ma ambicje być językiem doskonale szybkim. Język C++ robi tylko to, co zaprojektował programista. Jeśli nie zaprogramował zwalniania pamięci, to pamięć nie zostanie zwolniona.

Potrafimy zadeklarować i zainicjować wskaźnik oraz spytać, jaka wartość znajduje się pod nim lub zmodyfikować tę wartość. A jak zadać pytanie odwrotne — *jaki wskaźnik (jaki adres) ma konkretna zmienna?*

Ćwiczenie 8.18.

Zadeklaruj zmienne klasyczne i wyprowadź na ekran ich adresy:

```
...
int i = 5;
double pi;
string txt;
cout << "Zmienna i znajduje sie pod adresem: " << &i << endl;
cout << "Zmienna pi znajduje sie pod adresem: " << &pi << endl;
cout << "Zmienna txt znajduje sie pod adresem: " << &txt;
...
```

Rysunek 8.18. *Program wypisuje niniejsze informacje*

Na pytanie „pod jakim adresem znajduje się zmienna?" odpowiada operator &, zwany **operatorem adresacji**.

Przypomnijmy, że na odwrotne pytanie „jaka zmienna znajduje się pod danym adresem?" odpowiada **operator wyłuskania** *.

Para operatorów & i * zapewnia dwukierunkową łączność między światem zmiennych i światem ich adresów.

Operacje na danych

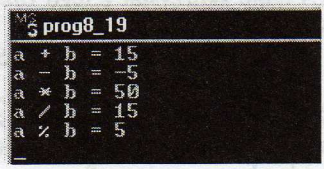
Każdy program w jakiś sposób przetwarza dane. Wartości danych są sumowane, mnożone, porównywane... Usystematyzujmy operacje arytmetyczne.

Ćwiczenie 8.19.

Napisz program, w którym zademonstrujesz podstawowe operacje arytmetyczne:

```
int a = 5, b = 10;
cout << "a + b = " << a + b << endl;
cout << "a - b = " << a - b << endl;
cout << "a * b = " << a * b << endl;
cout << "a / b = " << a + b << endl;
cout << "a % b = " << a % b << endl;
...
```

Rysunek 8.19. *Efekt działania tego programu*



Ostatnia operacja oznacza wyliczanie reszty z dzielenia pierwszego argumentu przez drugi.

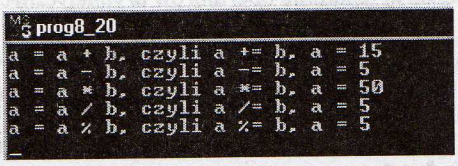
Mniej znanym zestawem operacji arytmetycznych są zapisy w rodzaju a = a + 5 czy b = b / 2. Operacje te — wyjątkowo często pojawiające się w programowaniu, w języku C++ doczekały się uproszczonego zapisu i specjalnie zoptymalizowanych implementacji w języku procesora.

Ćwiczenie 8.20.

Napisz program, w którym zademonstrujesz podstawowe operacje dwuargumentowe, modyfikujące lewy argument:

```
int a = 5, b = 10;
a += b;
cout << "a = a + b, czyli a += b, a = " << a << endl;
a -= b;
cout << "a = a - b, czyli a -= b, a = " << a << endl;
a *= b;
cout << "a = a * b, czyli a *= b, a = " << a << endl;
a /= b;
cout << "a = a / b, czyli a /= b, a = " << a << endl;
a %= b;
cout << "a = a % b, czyli a %= b, a = " << a << endl;
...
```

Rysunek 8.20. *Efekt działania tego programu*



Takich operacji nie ma w matematyce (jako zdefiniowanych wprost), ale ich ślady znajdujemy w języku potocznym — mówimy, że *zwiększamy* lub *zmniejszamy* wartość `a` o wartość `b` czy *domnażamy* do wartości `a` wartość `b`.

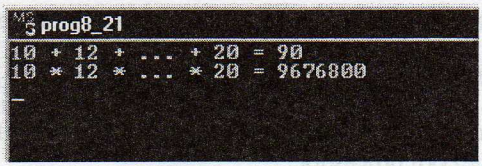
Operacje modyfikujące lewy argument sumy, różnicy, mnożenia, dzielenia i brania reszty, zapisane za pomocą powyższych dwuznakowych operatorów, są bardzo czytelne. Starajmy się o nich nie zapominać. Gdy chcemy napisać frazę w rodzaju `a = a + b`, napiszmy raczej `a += b`. Zapis jest czytelny, a ponadto kompilator taki kod lepiej przetłumaczy na wewnętrzny język komputera.

Ćwiczenie 8.21.

Znajdź sumę i iloczyn parzystych liczb z zakresu od 10 do 20:

```
int i, suma = 0;
long int iloczyn = 1L;
for( i = 10; i <= 20; i += 2)
{
   suma += i;
   iloczyn *= i;
}
cout << "10 + 12 + ... + 20 = " << suma << endl;
cout << "10 * 12 * ... * 20 = " << iloczyn << endl;
...
```

Rysunek 8.21. *Efekt działania tego programu*



W powyższym algorytmie zwróćmy uwagę na zwiększanie licznika pętli o wartość *2*, co dało się zapisać jako wyrażenie `i += 2`. Ponadto do zmiennej `suma` dodajemy, a do zmiennej `iloczyn` domnażamy bieżącą wartość podawaną przez pętlę. Oczywiście wykorzystujemy do tego operatory arytmetyczne `+=` i `*=`.

Choć nie jest to teraz przedmiotem naszego zainteresowania, zwróćmy też uwagę na deklarację zmiennej `iloczyn`. Istnieje poważne niebezpieczeństwo, że podczas kolejnych domnożeń ta zmienna przyjmie ogromną wartość, dlatego zadeklarowaliśmy ją jako zmienną najdłuższego dostępnego typu całkowitego `long int`. Porównaj też ćwiczenie 8.6.

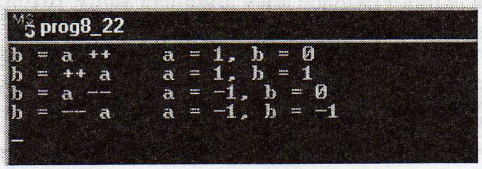
Mamy jeszcze dwa bardzo ciekawe i niezwykle popularne operatory arytmetyczne, dostarczające aż czterech operacji matematycznych.

Ćwiczenie 8.22.

Zademonstruj działanie operatorów arytmetycznych inkrementacji ++ i dekrementacji --:

```
int a, b;
a = 0; b = a ++;
cout << "b = a ++    a = " << a << ", b = " << b << endl;
a = 0; b = ++ a;
cout << "b = ++ a    a = " << a << ", b = " << b << endl;
a = 0; b = a --;
cout << "b = a --    a = " << a << ", b = " << b << endl;
a = 0; b = -- a;
cout << "b = -- a    a = " << a << ", b = " << b << endl;
...
```

Rysunek 8.22. *Efekt działania tego programu*



Jednoargumentowy operator ++ jest nazywany operatorem *inkrementacji*, gdyż zwiększa o *1* wartość argumentu. Operator -- jest operatorem *dekrementacji*, gdyż zmniejsza wartość argumentu o *1*. Przeanalizujmy jednak powyższy przykład, bo w powyższych operatorach tkwi jeszcze dodatkowa, istotna tajemnica tych — zdawałoby się — prostych operatorów. Tajemnica ta uwidacznia się, gdy operatory znajdą się w wyrażeniu.

Operator ++ stojący *za* argumentem inkrementuje go dopiero *po* opracowaniu całego wyrażenia. Dlatego w pierwszej linii wyników naszego programu *najpierw* wartości `b` została przypisana wartość `a`, a dopiero *potem* zmienna `a` została zwiększona o *1*. Operator ++ stojący *za* argumentem bywa nazywany operatorem *postinkrementacji*.

Operator ++ stojący *przed* argumentem inkrementuje go *przed* opracowaniem wyrażenia. Dlatego w drugiej linii najpierw została zwiększona o *1* wartość zmiennej a, a *potem* ta wartość została przypisana zmiennej b. Operator taki nazywa się operatorem *preinkrementacji*.

Zupełnie tak samo zdefiniowano operatory *postdekrementacji* i *predekrementacji*, co uwidaczniają linie 3. i 4. wyników programu.

Wynika z tego, że sama inkrementacja (i odpowiednio dekrementacja) argumentu a dokonuje się zarówno w zapisie a++, jak i ++a. Jednak jeśli operacja ta jest częścią wyrażenia arytmetycznego, należy rozróżniać sytuacje, gdy operator inkrementacji stoi przed czy za argumentem.

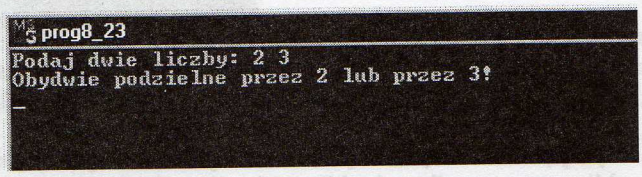
Poświęćmy jeszcze odrobinę uwagi operacjom dającym odpowiedzi typu *prawda-fałsz*, czyli *operacjom logicznym*.

Ćwiczenie 8.23.

Napisz program, który spyta o dwie liczby i sprawdzi, czy każda z nich jest podzielna przez 2 lub przez 3:

```
int a, b;
cout << "Podaj dwie liczby: ";
cin >> a >> b;
if(( a % 2 == 0 || a % 3 == 0) && (b % 2 == 0 || b % 3 == 0))
{
   cout << "Obydwie podzielne przez 2 lub przez 3!";
}
else
{
   cout << "Ktoras nie jest podzielna ani przez 2, ani przez 3...";
}
...
```

Rysunek 8.23. *Efekt działania tego programu*



Program musi podjąć pewną decyzję, wykonując obliczenia logiczne. W języku C++ jest kilka charakterystycznych miejsc, w których oczekuje się podejmowania podobnych decyzji: są to pola warunków w zwrotnicy dwukierunkowej (porównaj ćwiczenie 6.1) oraz pola pytań „czy powtarzać?" w trzech pętlach (ćwiczenia 6.11, 6.16, 6.20).

Konstruując wyrażenia logiczne, mamy do dyspozycji 6 oczywistych operatorów: <, <=, >, >=, !=, == o łatwo dającym się wypowiedzieć działaniu: *czy mniejsze?*, *czy mniejsze lub równe?*, *czy większe?*, *czy większe lub równe?*, *czy różne?*, *czy równe?*.

Jest też jeden *jednoargumentowy* operator zaprzeczenia wartości logicznej, zamieniający prawdę na fałsz i odwrotnie, mający brzmienie ! (wykrzyknik).

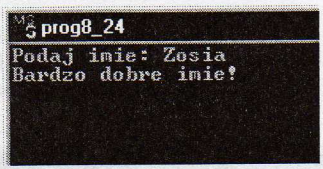
Mamy do dyspozycji też dwa spójniki || i &&, dające się wypowiedzieć jako *lub* oraz *i*. Spójniki pozwalają konstruować bardziej złożone wyrażenia logiczne (czyli dostarczające wartości *prawda-fałsz*), co nie jest zbyt trudne.

Ćwiczenie 8.24.

Napisz program, który spyta o imię i pochwali, jeśli ma ono nie więcej niż 6 liter i zaczyna się albo kończy literą a albo A:

```
#include <string>
...
string imie;
char pierwsza_litera, ostatnia_litera;
bool dobre_imie;
cout << "Podaj imie: ";
cin >> imie;
pierwsza_litera = imie[ 0];
ostatnia_litera = imie[ imie.size() - 1];
dobre_imie = imie.size() <= 6 &&
             (pierwsza_litera == 'a' || pierwsza_litera == 'A' ||
              ostatnia_litera == 'a' || ostatnia_litera == 'A');
if( dobre_imie == true)
{
   cout << "Bardzo dobre imie!";
}
else
{
   cout << "Marne imie ...";
}
...
```

Rysunek 8.24. *Efekt działania tego programu*



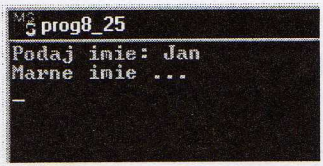
Ten program musi sprawdzić warunek (czyli obliczyć wartość wyrażenia logicznego) dość złożony, a w każdym razie brzydko zapisywalny. Dlatego wprowadzamy do gry pomocnicze zmienne `pierwsza_litera` i `ostatnia_litera`, przechwytujące znaki rozpoczynające i kończące imię. Ponadto wyrażenie logiczne jest najpierw, niejako na boku, obliczone, a wynik tych obliczeń jest przechowywany w zmiennej `dobre_imie` typu logicznego. Dzięki tym wstępnym przygotowaniom zwrotnica `if()` ma przejrzysty zapis.

Ćwiczenie 8.25.

Napisz poprzedni program, nie posługując się pomocniczymi zmiennymi typu char i bool:

```
string imie;
 cout << "Podaj imie: ";
 cin >> imie;
 if( imie.size() <= 6 &&
    (imie[ 0] == 'a' || imie[ 0] == 'A' ||
     imie[ imie.size() - 1] == 'a' || imie[ imie.size() - 1] == 'A'))
```

```
{
   cout << "Bardzo dobre imie!";
}
else
{
   cout << "Marne imie ...";
}
```



Rysunek 8.25. *Efekt działania tego programu jest taki sam, jak poprzedniego*

Unikniemy większości błędów związanych z konstruowaniem wyrażeń logicznych, jeśli zapamiętamy, że:

1. Operator relacji *czy równe?* to dwa znaki równości ==, nie jeden =.
2. Jeśli używamy spójników logicznych || (*lub*) albo && (*i*), w celu uzyskania dobrej czytelności zapisu warunku obficie nawiasujmy wyrażenia logiczne.

Podsumowanie

Dane to — obok algorytmów — drugi filar programowania komputerów. Na program komputerowy składa się odpowiednio zadeklarowany zestaw danych i operujące na nich algorytmy.

Wszystkie dane muszą być zadeklarowane. Powinny też być zainicjowane, bo język C++ nie inicjuje danych w momencie ich deklaracji.

Spośród wielu dostępnych typów danych zawsze musimy dokonywać wyboru najodpowiedniejszego rodzaju, mając na uwadze takie kryteria, jak zajętość pamięci, pojemność numeryczna, szybkość operowania, czytelność programu.

Rozdział 9.

Klasy i obiekty

Za chwilę bardzo istotnie poszerzymy dotychczasowy obraz języka C++. Do tej pory — mówiąc w wielkim skrócie — umiejętność programowania sprowadzała się do deklarowania odpowiednich zmiennych i komponowania algorytmów, operujących na tych zmiennych.

W tym rozdziale nauczymy się tworzyć nieograniczony zbiór nowych typów, zwanych *klasami* i deklarować zmienne tych typów, zwane *obiektami*.

Klasa jako nowy typ danych

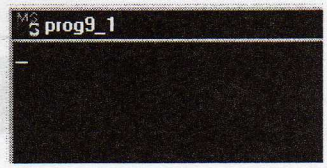
Od tego momentu w języku C++ nie jesteśmy ograniczeni typami danych, wbudowanymi w strukturę języka. Możemy tworzyć własne typy, które będziemy nazywać *klasami*.

Ćwiczenie 9.1.

Utwórz typ (czyli klasę) Kot, umożliwiającą deklarowanie numerycznych modeli kotów:

```
#include <iostream>
using namespace std;
void stop( void);
//---------------------------------------
class Kot
{
 private:
 public:
    string imie;
    int wiek;
};
//---------------------------------------
int main()
```

```
{
 Kot Mruczek;
 stop();
 return 0;
}
//-----------------------------------------
// Zatrzymanie konsoli na ekranie
void stop( void)
{
 char c;
 cin >> c;
}
```



Rysunek 9.1. *Ten program niczego nie wypisuje na ekranie. Niemniej jednak powinien poprawnie się skompilować i uruchomić*

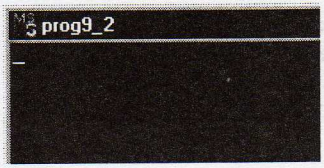
W programie tym wykorzystaliśmy (i w następnych też będziemy wykorzystywać) funkcję `stop()`, napisaną w ćwiczeniu 7.8.

W programie po raz pierwszy pojawia się *deklaracja klasy*. Po takiej deklaracji napis `Kot` — będący nazwą tej klasy — funkcjonuje tak, jak napisy określające inne typy: `char`, `int`, `double`, ... i umożliwia deklarowanie danych typu `Kot`. Zmienna `Mruczek` jest przykładem takiej danej.

Ćwiczenie 9.2.

Uzupełnij poprzedni przykład, deklarując obok zmiennej Mruczek tablicę 1000 innych kotów:

```
...
int main()
{
 Kot Mruczek, ObceKoty[ 1000];
 int A, B[ 1000];          //dla porównania — typy klasyczne
 stop();
 return 0;
}
...
```



Rysunek 9.2. *Ten program też niczego nie wypisuje na ekranie. Jednak również powinien poprawnie się skompilować i uruchomić*

Zwróćmy uwagę, że nazwy klas umożliwiają też standardowe deklarowanie tablic. Typy klasowe doskonale wmontowują się w system deklarowania zmiennych — niewprawne oko nie odróżni typu klasowego od klasycznego typu, wbudowanego w strukturę języka.

Wewnętrzny ustrój klasy — dane

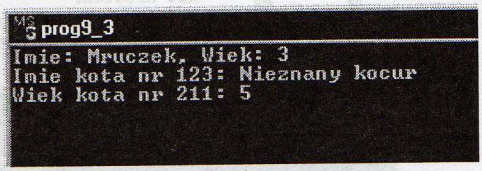
Klasa Kot, przytoczona w dwóch poprzednich ćwiczeniach, miała dwie publiczne, wewnętrzne dane: imie i wiek. Co to oznacza?

Ćwiczenie 9.3.

Uzupełnij poprzedni przykład, inicjując wewnętrzny ustrój zadeklarowanych obiektów (uwaga — w tekście pominiemy deklarację klasy identyczną, jak w poprzednim ćwiczeniu):

```
int main()
{
 Kot Mruczek, ObceKoty[ 1000];
 Mruczek.imie = "Mruczek";
 Mruczek.wiek = 3;
 for( int i = 0; i < 1000; i++)
 {
    ObceKoty[ i].imie = "Nieznany kocur";
    ObceKoty[ i].wiek = 5;
 }
 cout << "Imie: " << Mruczek.imie << ", " << "Wiek: " << Mruczek.wiek << endl;
 cout << "Imie kota nr 123: " << ObceKoty[123].imie << endl;
 cout << "Wiek kota nr 211: " << ObceKoty[211].wiek;
 stop();
 return 0;
}
```

Rysunek 9.3.
Oto wyjście programu



Wewnętrzne składniki zadeklarowanego obiektu, opisane w deklaracji klasy, są dostępne w zademonstrowanej tutaj składni z kropką. Ze składnią tą już się spotkaliśmy np. w ćwiczeniu 4.6, gdzie chodziło o obiekt cout i jego składnik width().

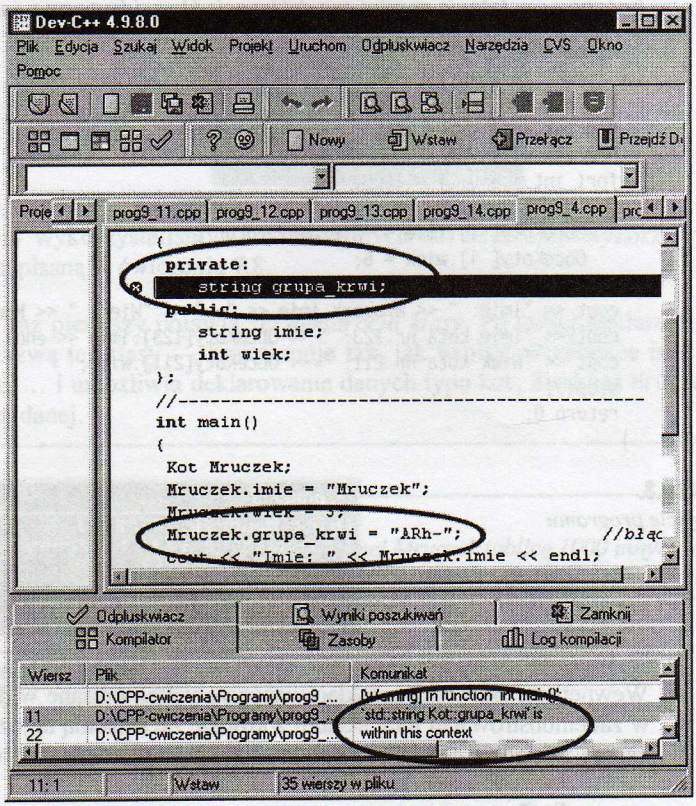
Ćwiczenie 9.4.

Uzupełnij poprzedni przykład w taki sposób, by się przekonać, że dostępne są tylko te składniki obiektu, które w klasie są zadeklarowane jako publiczne, nie prywatne:

```
class Kot
{
 private:
    string grupa_krwi;
 public:
    string imie;
    int wiek;
};
```

```
//-----------------------------------------
int main()
{
 Kot Mruczek;
 Mruczek.imie = "Mruczek";
 Mruczek.wiek = 3;
 Mruczek.grupa_krwi = "ARh-";          //błąd!
 cout << "Krew: " << Mruczek.grupa_krwi;  //błąd!
 stop();
 return 0;
}
```

Rysunek 9.4. *Program nie generuje żadnej informacji, bo zawiera błędy i kompilacja kończy się komunikatem „zmienna grupa_krwi jest niedostępna"*



Zmienne prywatne są *niedostępne* dla użytkownika klasy, czyli kogoś, kto (tak jak w powyższym przykładzie) zadeklarował sobie obiekt i próbuje manipulować zmienną prywatną. Do czego zatem służą elementy prywatne klasy? Stanowią wewnętrzny ustrój klasy, która za chwilę okaże się czymś więcej niż tylko zbiorem zmiennych, takich jak `imie` czy `wiek`.

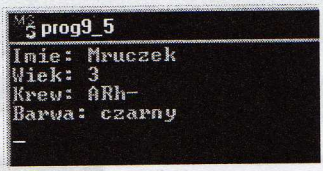
Przykład ten skompiluje się, jeśli w funkcji `main()` zrezygnujemy z sięgania po zmienną prywatną albo jeśli odpowiednio przebudujemy deklarację klasy, przenosząc grupę krwi do sekcji elementów publicznych.

Ćwiczenie 9.5.

Skomplikuj ustrój modelu kota, dbając o rozmieszczenie jego cech we właściwych sekcjach dostępności:

```
class Kot
{
 private:
 public:
    string imie;
    int wiek;
    string grupa_krwi;
    string kolor;
    double waga;
//i tak dalej, ... aż uzyskamy wystarczająco dokładny model kota
};
//---------------------------------------
int main()
{
 Kot Mruczek;
 Mruczek.imie = "Mruczek";
 Mruczek.wiek = 3;
 Mruczek.grupa_krwi = "ARh-";
 Mruczek.kolor = "czarny";
//i tak dalej...
 cout << "Imie: " << Mruczek.imie << endl;
 cout << "Wiek: " << Mruczek.wiek << endl;
 cout << "Krew: " << Mruczek.grupa_krwi << endl;
 cout << "Barwa: " << Mruczek.kolor;
//i tak dalej...
 stop();
 return 0;
}
```

Rysunek 9.5. *Program generuje niniejszy ekran*



Programowanie obiektowe w pewnych zastosowaniach bywa nazywane modelowaniem. Typ `Kot` staje się coraz bardziej złożony i coraz dokładniej opisuje przedmiot modelowania. Jednak brakuje tutaj czegoś niezwykle istotnego.

Wewnętrzny ustrój klasy — algorytmy

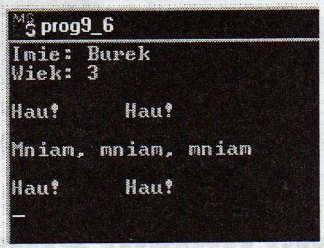
Typ złożony, zwany *klasą*, zawiera *dane* i *algorytmy*. W pewnym (numerycznym) sensie możemy powiedzieć, że model cyfrowego kota zaczyna naprawdę funkcjonować. By jednak nie być jednostronnymi, przerzućmy się na psy.

Ćwiczenie 9.6.

Skonstruuj klasę Pies z algorytmem szczekania i jedzenia:

```
class Pies
{
 private:
 public:
    string imie;
    int wiek;
    double waga;
    void jedz( void);
    void szczekaj( void);
//i tak dalej, ... aż uzyskamy wystarczająco dokładny model psa
};
//---------------------------------------
int main()
{
 Pies p1;
 p1.imie = "Burek";
 p1.wiek = 3;
 cout << "Imie: " << p1.imie << endl;
 cout << "Wiek: " << p1.wiek << endl;
 p1.szczekaj();
 p1.jedz();
 p1.szczekaj();
 stop();
 return 0;
}
//---------------------------------------
void Pies :: jedz( void)
{
 cout << endl << "Mniam, mniam, mniam" << endl;
}
//---------------------------------------
void Pies :: szczekaj( void)
{
 cout << endl << "Hau!     Hau!" << endl;
}
```

Rysunek 9.6. *Oto ekran wyjściowy programu*



W powyższym przykładzie w klasie Pies pojawiły się funkcje. Jest to zasadnicze odkrycie programowania obiektowego.

Kluczową cechą programowania obiektowego jest możliwość tworzenia typów łączących w sobie dane i algorytmy.

Zwróćmy uwagę na szczegóły składniowe: jak deklaruje się algorytmy, jak wywołuje się je w funkcji `main()` i jak definiuje w dalszej części pliku źródłowego. Deklaracje i definicje funkcji nie odbiegają od tego, co powiedzieliśmy w rozdziale o funkcjach (porównaj ćwiczenie 7.1 i następne). Jedynie w definicji musimy pamiętać o dodatkowym wtrąceniu, określającym, do jakiej klasy należy funkcja:

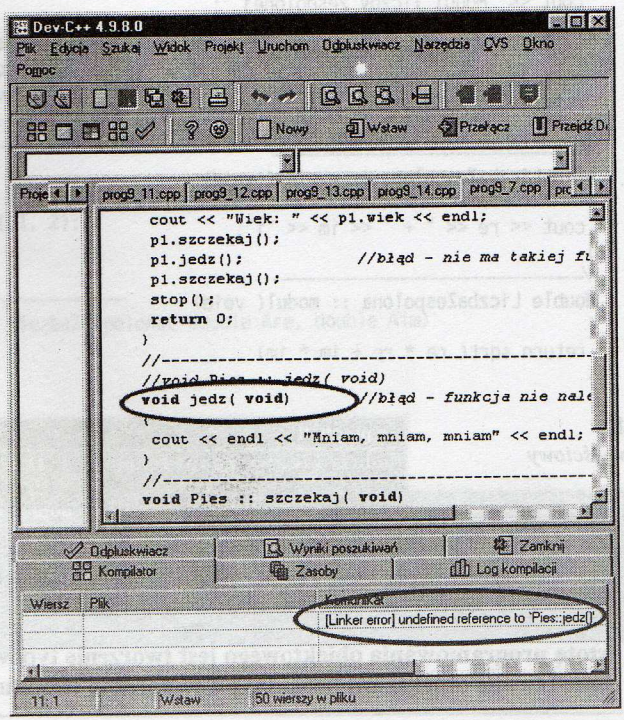
```
void Pies :: szczekaj( void)
{
 ...
```

O wtrąceniu tym niestety często zapominamy.

Ćwiczenie 9.7.

W poprzednim przykładzie zapomnij o wtrąceniu, do jakiej klasy należą funkcje:

```
int main()
{
 ...
 p1.jedz();   //w klasie Pies brakuje funkcji jedz()
 ...
}
//-----------------------------------------
//void Pies :: jedz( void)
void jedz( void)    //błąd!!!
{
 cout << endl << "Mniam, mniam, mniam" << endl;
}
```



Rysunek 9.7. *Kompilator nie może znaleźć funkcji jedz()*

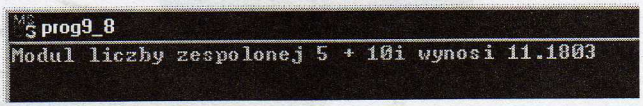
Program nie działa, bo w funkcji `main()` znajduje się wywołanie algorytmu `jedz()`, należącego do obiektu. Algorytm ten jest co prawda poprawnie zadeklarowany w klasie `Pies`, ale w programie brakuje jego definicji. Definicja funkcji `jedz()` nie jest rozumiana jako składnik klasy, bo w jej nagłówku zabrakło charakterystycznej frazy przynależności obiektowej `Pies::`. Jest to błąd dość trudny do zauważenia, dlatego uważnie spisujmy nagłówki funkcji, należących do klas.

Ćwiczenie 9.8.

Napisz program operujący klasą LiczbaZespolona:

```
#include <iostream>
#include <string>
#include <math.h>
...
class LiczbaZespolona
{
 private:
 public:
    double re, im;
    void wypisz( void);
    double modul( void);
};
//-----------------------------------------
int main()
{
 LiczbaZespolona z;
 z.re = 5;
 z.im = 10;
 cout << "Modul liczby zespolonej ";
 z.wypisz();
 cout << " wynosi " << z.modul();
 stop();
 return 0;
}
//-----------------------------------------
void LiczbaZespolona :: wypisz( void)
{
 cout << re << " + " << im << "i";
}
//-----------------------------------------
double LiczbaZespolona :: modul( void)
{
 return sqrt( re * re + im * im);
}
```

Rysunek 9.8. *Oto ekran wyjściowy programu*



Jest to kolejny przykład klasy, czyli typu złożonego, teraz składającego się z dwóch danych rzeczywistych i dwóch algorytmów. Każdy obiekt, zadeklarowany według tego wzorca, potrafi wypisać się na ekranie i obliczyć swój moduł.

Istotą programowania obiektowego jest tworzenie typów (zwanych klasami), których dane (zwane obiektami) posiadają wewnętrzną strukturę i funkcjonalność.

Pewien specjalny algorytm, zwany konstruktorem

Powróćmy na moment do któregokolwiek z ostatnich przykładów i zwróćmy uwagę na inicjowanie wewnętrznego ustroju danych obiektowych:

```
...
LiczbaZespolona z;
z.re = 5;
z.im = 10;
...
```

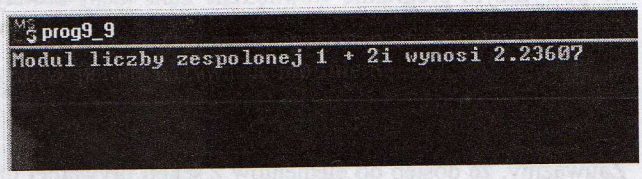
Klasy mogą operować setkami parametrów — zapewne dobry model kota czy psa zbliżyłby się do tego stopnia komplikacji. Czy istnieje zręczniejszy mechanizm inicjowania obiektów?

Ćwiczenie 9.9.

Uzupełnij klasę LiczbaZespolona (porównaj poprzednie ćwiczenie) o deklarację, definicję i wywołanie konstruktora:

```
...
class LiczbaZespolona
{
 private:
 public:
    double re, im;
    LiczbaZespolona( double Are, double Aim);
    void wypisz( void);
    double modul( void);
};
//-----------------------------------------
int main()
{
 LiczbaZespolona z( 1, 2);
 ...
}
//-----------------------------------------
LiczbaZespolona :: LiczbaZespolona( double Are, double Aim)
{
 re = Are;
 im = Aim;
}
...
```

Rysunek 9.9. *Wynik działania programu*



W klasie zadeklarowano nową funkcję. Funkcja ta ma następujące, unikalne cechy:

1. **Nazywa się identycznie jak klasa, czyli akurat tutaj** `LiczbaZespolona`.
2. **Nie zwraca żadnego rezultatu, nawet oznaczonego typem** `void`.

Funkcja spełniająca powyższe warunki nazywa się **konstruktorem klasy** i służy do inicjowania deklarowanych obiektów.

Skoro klasa `LiczbaZespolona` ma konstruktor i nie potrzeba już ręcznie sięgać do zmiennych wewnętrznych `re` i `im`, żeby je zainicjować (przypomnij sobie funkcję `main()` w ćwiczeniu 9.8), przesuń ich deklaracje do sekcji prywatnej klasy.

Ćwiczenie 9.10.

Niech klasa LiczbaZespolona ukryje swój wewnętrzny ustrój, udostępniając użytkownikowi tylko wybrane elementy:

```
class LiczbaZespolona
{
 private:
    double re, im;
 public:
    LiczbaZespolona( double Are, double Aim);
    void wypisz( void);
    double modul( void);
};
//-----------------------------------------
int main()
{
 double a, b;
 cout << "Podaj czesc rzeczywista liczby zespolonej: ";
 cin >> a;
 cout << "Podaj czesc urojona liczby zespolonej: ";
 cin >> b;
 LiczbaZespolona z( a, b);
 ...
}
...
```

Rysunek 9.10. *Wynik działania programu*

```
prog9_10
Podaj czesc rzeczywista liczby zespolonej: 1
Podaj czesc urojona liczby zespolonej: 1
Modul liczby zespolonej 1 + 1i wynosi 1.41421
_
```

W powyższym programie wewnętrzne dane klasy zostały przeniesione do sekcji prywatnej, zatem nie są już dostępne w funkcji `main()` (porównaj ćwiczenie 9.4). Są dostępne tylko pewne funkcje i rzeczywiście korzystamy z tej dostępności w funkcji `main()`. Sama funkcja `main()` w porównaniu z poprzednim przykładem została trochę przebudowana.

Zauważmy, że dostęp do elementów z sekcji prywatnej mają teraz tylko algorytmy należące do klasy, np.:

```
double LiczbaZespolona :: modul( void)
{
 return sqrt( re * re + im * im);
}
```

Funkcja `modul()` swobodnie sięga po prywatne `re` i `im`, bo i ona, i `re` oraz `im` należą do tej samej klasy `LiczbaZespolona`.

Funkcje nie należące do klasy — w naszym ćwiczeniu jest to funkcja `main()` — mogą sięgać tylko do składników publicznych.

Ten, kto zaproponował wyróżnikom dostępności nazwy `private` i `public`, wykazał się bardzo dobrym wyczuciem i smakiem.

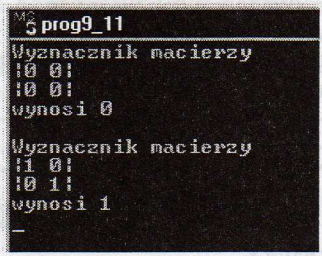
Ćwiczenie 9.11.

Napisz klasę Macierz, realizującą podstawowe operacje na macierzach 2×2. Niech klasa ma dwa konstruktory:

```
class Macierz
{
 private:
    double a11, a12, a21, a22;
 public:
    Macierz( void);
    Macierz( double A11, double A12, double A21, double A22);
    void wypisz( void);
    double wyznacznik( void);
};
//-----------------------------------------
int main()
{
 Macierz M1, M2( 1, 0, 0, 1);
 cout << "Wyznacznik macierzy";
 M1.wypisz();
 cout << "wynosi " << M1.wyznacznik() << endl << endl;
 cout << "Wyznacznik macierzy";
 M2.wypisz();
 cout << "wynosi " << M2.wyznacznik();
 stop();
 return 0;
}
//-----------------------------------------
Macierz :: Macierz( void)
{
 a11 = a12 = a21 = a22 = 0;
}
//-----------------------------------------
Macierz :: Macierz( double A11, double A12, double A21, double A22)
{
 a11 = A11;
 a12 = A12;
 a21 = A21;
 a22 = A22;
}
//-----------------------------------------
void Macierz :: wypisz( void)
```

```
{
 cout << endl << '|' << a11 << ' ' << a12 << '|';
 cout << endl << '|' << a21 << ' ' << a22 << '|' << endl;
}
//----------------------------------------
double Macierz :: wyznacznik( void)
{
 return a11 * a22 - a12 * a21;
}
```

Rysunek 9.11.
Wyniki tego programu



Deklaracja klasy zawiera *dwa* konstruktory. Sformułowane w omówieniu ćwiczenia 9.9 kryteria bycia konstruktorem (nazwa tożsama z nazwą klasy i brak typu zwracanego rezultatu) dopuszczają deklarowanie wielu takich funkcji, z których każda jest pełnoprawnym konstruktorem.

W tym ćwiczeniu wprowadzamy do gry 2 konstruktory — *bezargumentowy* i *4-argumentowy*. W funkcji `main()` każdy z nich jest wykorzystywany do utworzenia i zainicjowania jednego obiektu.

Konstruktor bezargumentowy jest bardzo ważny w strukturze języka C++, bo umożliwia *deklarowanie tablic obiektów*. Gdyby nie było konstruktora bezargumentowego, nie udałoby się zadeklarować tablicy.

Ćwiczenie 9.12.

Wykorzystując poprzedni program, dodaj w funkcji main() deklarację tablicy 1000 macierzy. Sprawdź, że usunięcie konstruktora bezargumentowego uniemożliwia kompilację programu.

```
int main()
{
 Macierz M[ 1000];  //o ile w klasie Macierz jest konstruktor bezargumentowy
 cout << "Wyznacznik macierzy nr 13";
 M[ 13].wypisz();
 cout << "wynosi " << M[13].wyznacznik();
 stop();
 return 0;
}
```

Rysunek 9.12.
Oto wyniki działania programu

Jeśli z powyższej deklaracji i definicji klasy Macierz usuniemy konstruktor bezargumentowy, kompilacja programu kończy się błędem „nie ma konstruktora bezargumentowego".

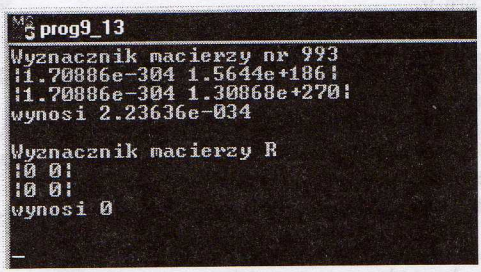
Musimy zatem pamiętać, że jeśli zamierzamy operować tablicami swoich obiektów, w klasie musi znaleźć się konstruktor bezargumentowy. Jest jednak wyjątek od tej zasady: jeśli w klasie nie ma żadnego konstruktora, język C++ uważa, że jest konstruktor bezargumentowy, który nie robi nic. Można jednak deklarować tablice.

Ćwiczenie 9.13.

Niech klasa Macierz nie ma żadnego konstruktora. Sprawdź, że mimo to można deklarować obiekty, a nawet ich tablice:

```
class Macierz
{
 private:
    double a11, a12, a21, a22;
 public:
    void wypisz( void);
    double wyznacznik( void);
};
//---------------------------------------
int main()
{
 Macierz M[ 1000], R;
 cout << "Wyznacznik macierzy nr 993";
 M[ 993].wypisz();
 cout << "wynosi " << M[993].wyznacznik() << endl << endl;
 cout << "Wyznacznik macierzy R";
 R.wypisz();
 cout << "wynosi " << R.wyznacznik() << endl << endl;
 ...
```

Rysunek 9.13. *Program generuje niniejsze wyniki*



Widzimy z tego przykładu, że brak konstruktorów nie jest krytycznym błędem — obiekty według takiej zubożonej klasy można mimo wszystko deklarować. Dzieje się tak dlatego, że **przy braku jakiegokolwiek konstruktora język C++ formalnie dodaje konstruktor bezparametrowy**, który niczego nie robi.

Brak konstruktora jest sytuacją niecodzienną. Niech nasze klasy zawsze mają choćby najprostszy, bezargumentowy konstruktor, który wstępnie zainicjuje (wyzeruje?) wewnętrzne dane. Porządek w programowaniu być musi.

Ćwiczenie 9.14.

Uzupełnij o konstruktory klasę Pies z ćwiczenia 9.6:

```
...
class Pies
{
 private:
    string imie;
    int wiek;
    double waga;
 public:
    Pies( void);
    Pies( string Aimie, int Awiek, double Awaga);
    void jedz( void);
    void szczekaj( void);
//i tak dalej, ... aż uzyskamy wystarczająco dokładny model psa
};
//-----------------------------------------
int main()
{
 string s;
 int a;
 double b;
 cout << "Imie Twojego psa: ";
 cin >> s;
 cout << "Wiek: ";
 cin >> a;
 cout << "Waga: ";
 cin >> b;
 Pies TwojPies( s, a, b), ZwyklyKundel;
 TwojPies.szczekaj();
 TwojPies.jedz();
 ZwyklyKundel.szczekaj();
 ZwyklyKundel.jedz();
 stop();
 return 0;
}
//-----------------------------------------
// Konstruktor bezargumentowy buduje typowego kundelka
Pies :: Pies( void)
{
 imie = "Kundel";
 wiek = 5;
 waga = 5;
}
//-----------------------------------------
// Inny konstruktor tworzy specjalizowanego psa
Pies :: Pies( string Aimie, int Awiek, double Awaga)
{
 imie = Aimie;
 wiek = Awiek;
 waga = Awaga;
}
//-----------------------------------------
void Pies :: jedz( void)
{
 cout << endl << "To ja " << imie << ". Mniam, mniam, mniam" << endl;
}
```

```
//-----------------------------------------
void Pies :: szczekaj( void)
{
 cout << endl << "To ja " << imie << ". Hau! Hau!" << endl;
}
 ...
```

Rysunek 9.14. *Program generuje takie wyniki*

```
prog9_14
Imie Twojego psa: Tina
Wiek: 9
Waga: 35

To ja Tina. Hau! Hau!

To ja Tina. Mniam, mniam, mniam

To ja Kundel. Hau! Hau!

To ja Kundel. Mniam, mniam, mniam
```

Obecność konstruktorów jest chyba najważniejszym wyróżnikiem kompletności klasy. Taką klasą łatwiej się manipuluje — deklaruje dobrze zainicjowane, konkretne obiekty, w tym także tablice obiektów. Tutaj konstruktor bezargumentowy — nie mając dyrektyw, model jakiego psa utworzyć — tworzy pospolitego kundla. Psy konkretne są tworzone za pomocą *konstruktora merytorycznego*, czyli takiego, który jest potrzebny programiście do przeprowadzenia konkretnego inicjowania.

Kompletność klasy jest ważna także dlatego, że jej egzemplarze będziemy zazwyczaj przechowywać w *kontenerach*.

Podsumowanie

Klasy są to nowoczesne, złożone typy, łączące w sobie zarówno dane, jak i algorytmy.

Ujęte w klasę dane i algorytmy są pogrupowane w sekcjach `private` i `public` i tym samym mają określone zasięgi swojej widoczności.

O ile to tylko możliwe, zawsze powinniśmy ukrywać elementy klasy, umieszczając je na liście `private`.

Egzemplarze klas nazywają się obiektami. Obiekty są to zadeklarowane, konkretne dane typów klasowych.

Algorytm, który nazywa się tak jak klasa i nie zwraca żadnego rezultatu, nazywa się konstruktorem i służy do inicjowania egzemplarzy klasy. Starajmy się, by każda nasza klasa miała przynajmniej jeden konstruktor!

Do deklarowania obiektów służą konstruktory.

Rozdział 10.

Kontenery na dane

Manipulowanie zbiorami danych, choćby dodanie kolejnego modelu kota do już istniejącego spisu kotów, usuwanie liczby zespolonej z jakiejś ich tablicy, zrobienie gdzieś w środku tablicy miejsca na nową daną czy posortowanie lub przeszukanie tablicy to elementarz programowania. Do dzisiaj trudny elementarz.

Dokładne omówienie tego, o czym wspomnimy w tym rozdziale, przerasta nasze możliwości techniczne i mogłoby być elementem podobnej książki, ale dla zaawansowanych programistów C++. Jednak użytkowanie algorytmów biblioteki standardowej jest całkiem proste. Wielu programistów w ogóle nie zadaje sobie pytania, w jaki sposób to wszystko działa.

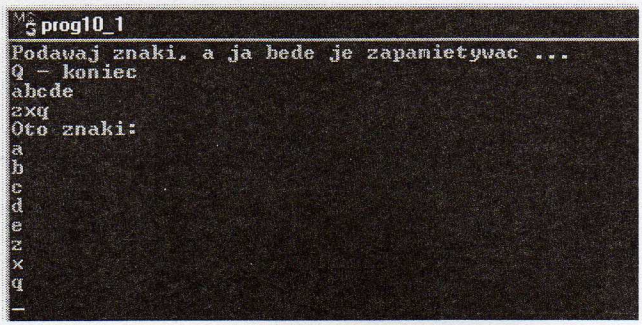
Ćwiczenie 10.1.

Napisz program, który w dynamicznej tablicy będzie gromadził znaki wprowadzane z klawiatury:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
 vector <char> znaki;
 char c;
 int i;
 cout << "Podawaj znaki, a ja bede je zapamietywac ..." << endl;
 cout << "Q - koniec" << endl;
 do
 {
    cin >> c;
    znaki.push_back( c);

 } while( c != 'q');
 cout << "Oto znaki:" << endl;
```

```
    for( i = 0; i < znaki.size(); i ++)
    {
       cout << znaki[ i] << endl;
    }
    stop();
    return 0;
}
```



Rysunek 10.1. *Oto rezultaty działania tego programu*

Program posługuje się pętlą `do{...} while(...)` (porównaj ćwiczenie 6.20). W pętli jest odczytywany znak wprowadzany przez użytkownika, a potem jest dopisywany do tablicy `znaki`:

```
...
znaki.push_back( c);
...
```

Tablica `znaki` nie jest zwykłą tablicą (porównaj ćwiczenie 8.7). Już z powyższej, charakterystycznej składni z kropką możemy się zorientować, że tablica `znaki` jest obiektem. Jest to tzw. *obiekt kontenerowy* (krócej *kontener*), służący do *przechowywania danych.*

Spotykamy się tutaj z jedną z najważniejszych funkcji zdefiniowanych w klasie tego kontenera — `push_back()` — dopisz daną na końcu kontenera. Widzimy też niezmiernie użyteczną funkcję `size()` — ile danych już jest w kontenerze — oraz operację wyciągania danej z kontenera za pomocą klasycznej konstrukcji tablicowej `znaki[ i]`.

Tablica — kontener ma ogromną przewagę nad klasycznymi tablicami języka C++, bo nie musi mieć z góry ustalonego rozmiaru. Jest to — jak mówimy — *tablica dynamiczna*, czyli dopasowująca się do liczebności zbioru, którym manipulujemy.

Ćwiczenie 10.2.

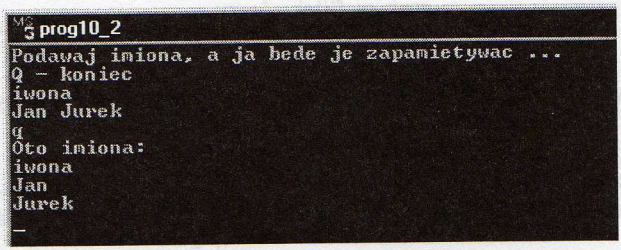
Napisz program, który będzie gromadził w kontenerze wprowadzane z klawiatury imiona:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main()
{
 vector <string> imiona;
 string s;
 int i;
```

```
cout << "Podawaj imiona, a ja bede je zapamietywac ..." << endl;
cout << "Q - koniec" << endl;
while( true)          //wieczna pętla
{
   cin >> s;
   if( s == "q" || s == "Q")
      break;
   imiona.push_back( s);

}
cout << "Oto imiona:" << endl;
for( i = 0; i < imiona.size(); i ++)
{
   cout << imiona[ i] << endl;
}
...
```

Rysunek 10.2. *Oto rezultaty działania tego programu*



Program wykorzystuje konstrukcję *wiecznej pętli*, w odpowiednich warunkach przerywanej instrukcją `break` (porównaj też ćwiczenie 6.25).

Tym razem w kontenerze przechowujemy *stringi*. Widać to z deklaracji zmiennej, reprezentującej obiekt kontenera:

```
vector <string> imiona;
```

Mimo to cały program wygląda uderzająco podobnie. Podobieństwo to wynika z tego, że kontenerowi jest wszystko jedno, jaki typ danych jest w nim przechowywany. Dopisanie danej na końcu kontenera zawsze realizuje funkcja `push_back()`, bieżącą liczbę danych określa funkcja `size()`, a po konkretną daną sięgamy za pomocą klasycznych, tablicowych nawiasów. Musimy tylko zauważyć, że w poprzednim ćwiczeniu za pomocą tych instrumentów operowaliśmy na typie znakowym `char`, a teraz robimy to na typie `string`.

Ćwiczenie 10.3.

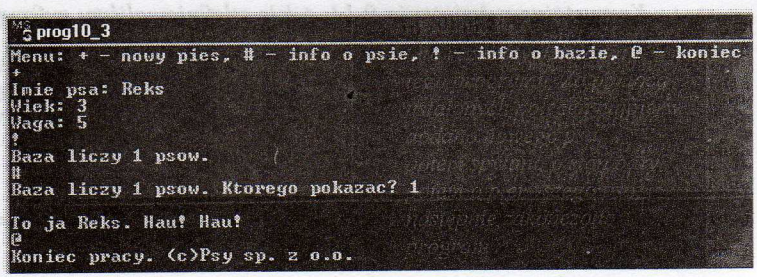
Korzystając z ćwiczenia 9.14, gdzie zdefiniowaliśmy cyfrowy model psa, napisz kontenerową bazę danych o psach:

```
...
#include <vector>
...
class Pies
{
 // porównaj ćwiczenie 9.14
};
//-----------------------------------------
int main()
```

```
{
 vector <Pies> psy;
 string imie;
 int nr, wiek;
 double waga;
 char c;
 bool koniec = false;

 cout << "Menu: + - nowy pies, # - info o psie, ! - info o bazie, @ - koniec" << endl;
 do                    //pętla główna
 {
    cin >> c;
    switch( c)
    {
     case '+':         //dopisanie nowego psa
                cout << "Imie psa: ";
                cin >> imie;
                cout << "Wiek: ";
                cin >> wiek;
                cout << "Waga: ";
                cin >> waga;
                psy.push_back( Pies( imie, wiek, waga));
                break;
     case '#':         //info o psie
                cout << "Baza liczy " << psy.size() << " psow. Ktorego pokazac? ";
                cin >> nr;
                if( nr > 0 && nr <= psy.size())
                {
                   psy[ nr - 1].szczekaj();    //tablice numerujemy od 0, nie 1
                }
                break;
     case '!':         //info o bazie
                cout << "Baza liczy " << psy.size() << " psow." << endl;
                break;
     case '@':         //koniec
                cout << "Koniec pracy. (c)Psy sp. z o.o.";
                koniec = true;
                break;
     default:
                cout << "Nieobslugiwana funkcja" << endl;
                break;
    }
 } while( koniec != true);
 stop();
 return 0;
}
...
```

Rysunek 10.3. *Oto rezultaty działania tego programu. Za pomocą ustalonych poleceń najpierw dodano nowego psa, potem spytano o stan bazy, potem o pierwszego psa, następnie zakończono program*



Program może denerwować nieporadnym interfejsem, ale niewątpliwie realizuje zamierzone cele. Bazę psów zbudowaliśmy w oparciu o kontener `vector`, w którym przechowujemy zmienne typu `Pies`:

```
...
vector <Pies> psy;
...
```

Program ma typową strukturę pętli, w której pobieramy kod naciśniętego klawisza (porównaj ćwiczenie 6.20), a potem za pomocą zwrotnicy wielokrotnej (porównaj ćwiczenie 6.7) podejmujemy decyzję, co będziemy robić.

Jeśli naciśniętym klawiszem jest znak +, do kontenera `psy` zostaje dodany nowy pies. Najpierw w standardowy sposób ustalamy jego `imie`, `wiek` i `waga`, a potem dopisujemy go na końcu bazy danych:

```
...
psy.push_back( Pies( imie, wiek, waga));
...
```

Wykorzystujemy przy tym konstruktor merytoryczny, służący do tworzenia konkretnego modelu psa.

Jeżeli naciśniętym klawiszem jest znak #, pytamy o numer psa, sprawdzamy, czy jest to numer z dobrego zakresu i wydajemy odpowiedniemu psu polecenie, by zaszczekał:

```
...
cin >> nr;
if( nr > 0 && nr <= psy.size())
{
   psy[ nr - 1].szczekaj();    //tablice numerujemy od 0, nie 1
}
...
```

Warto zauważyć, że kontenery, tak jak i zwykłe tablice, numerują swoje elementy od *0*, nie *1*. Jest to uniwersalna zasada — język C++ wszystko numeruje od *0*. **Zapamiętajmy tę zasadę koniecznie!**

Jeśli naciśniętym klawiszem jest znak @, ustawia się taka wartość zmiennej `koniec`, że główna pętla programu kończy swoje obroty:

```
        ...
        koniec = true;
        ...
} while( koniec != true);
...
```

Ćwiczenie 10.4.

Napisz program, który będzie losować i umieszczać w kontenerze 1000 liczb. Po załadowaniu kontenera wyświetl informacyjnie jakiś podzbiór tych liczb:

```
#include <iostream>
#include <vector>
using namespace std;
void stop( void);
```

```
//-----------------------------------------
int main()
{
 vector <int> liczby;
 int i, ile = 1000;

 cout << "Losowanie " << ile << " liczb." << endl;
 for( i = 0; i < ile; i ++)
 {
    liczby.push_back( rand());
 }
 cout << "Zakonczono zapelnianie kontenera." << endl;
 for( i = 100; i < 110; i ++)
 {
    cout << "Liczba nr " << i << " = ";
    cout.width( 10);
    cout << liczby[ i] << endl;
 }
 stop();
 return 0;
}
...
```

Rysunek 10.4.
Rezultat działania programu

```
prog10_4
Losowanie 1000 liczb.
Zakonczono zapelnianie kontenera.
Liczba nr 100 =       4833
Liczba nr 101 =      31115
Liczba nr 102 =       4639
Liczba nr 103 =      29658
Liczba nr 104 =      22704
Liczba nr 105 =       9930
Liczba nr 106 =      13977
Liczba nr 107 =       2306
Liczba nr 108 =      31673
Liczba nr 109 =      22386
```

W programie tym nie ma nic tajemniczego. Po takiej deklaracji kontenera, by był on przystosowany do przechowywania liczb całkowitych, wrzucamy do niego *1000* wartości. Wykorzystujemy przy tym znaną z poprzednich programów funkcję `push_back()` — wsuń nowy obiekt (tutaj liczbę całkowitą) na koniec ich zbioru.

Funkcja `push_back()` doskonale nadaje się do zapełniania kontenera, bo jest bardzo szybka — łatwo bowiem dodać daną na koniec zbioru. Możemy jednak zażądać dodania danej nie na koniec kontenera, a gdzieś w środku.

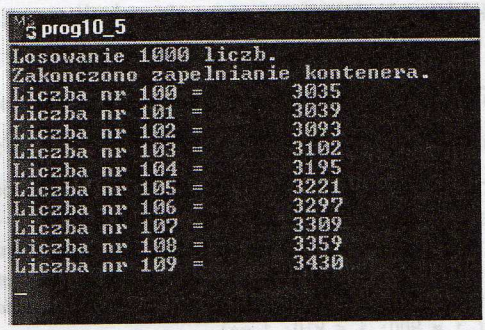
Ćwiczenie 10.5.

Tak zmodyfikuj poprzedni program, by kontener zawierał posortowany ich zbiór. To znaczy kolejne liczby wkładaj nie na koniec kontenera, a od razu na właściwe miejsca, wyznaczone ich wielkością:

```
...
vector <int> liczby;
int a, i, j, pozycja, ile = 1000;

cout << "Losowanie " << ile << " liczb." << endl;
for( i = 0; i < ile; i ++)
```

```
{
    a = rand();          //kolejny obiekt do kontenera...
    pozycja = 0;         //... i wstępnie sugerowana pozycja w kontenerze
    for( j = 0; j < liczby.size(); j ++)
    {
        if( a > liczby[ j])
        {
            pozycja ++;  //szukamy miejsca w szeregu dla obiektu a
        }
    }
    liczby.insert( liczby.begin() + pozycja, a);
}
cout << "Zakonczono zapelnianie kontenera." << endl;
for( i = 100; i < 110; i ++)
{
    cout << "Liczba nr " << i << " = ";
    cout.width( 10);
    cout << liczby[ i] << endl;
}
...
```



Rysunek 10.5. *Próbka posortowanego zbioru liczb pseudolosowych*

Tajemnica tego algorytmu jest prosta — każda liczba musi znaleźć swoje miejsce w ich posortowanym szeregu. Zakładając, że na początku kontenera są najmniejsze liczby, każdą wylosowaną liczbę porównujemy z już obecnymi w kontenerze, zaczynając od początku.

Algorytm ten wymaga umiejętności wstawiania obiektów do środka kontenera, nie — jak poprzednio — na jego koniec. Każda liczba musi być wstawiona w ściśle określone miejsce. Jest to skomplikowana, trudna i czasochłonna operacja. Wyobraźmy sobie, że na wielkie przyjęcie wpuszczamy pojedynczo gości i sadzamy ich przy stole, począwszy od brzegu stołu. Sadzamy ich jednak w porządku alfabetycznym, a to oznacza, że już siedzący bardzo często muszą robić między sobą miejsce, bo został do sali wpuszczony ktoś, którego miejsce nie wypada ani na końcu, ani na początku zbioru już siedzących osób.

Najważniejsza jest w tym wszystkim kontenerowa funkcja, wstawiająca obiekt we wskazane miejsce:

```
...
liczby.insert( liczby.begin() + pozycja, a);
...
```

Jest to funkcja skomplikowana i czasochłonna, ale niekiedy niezbędna i — co najważniejsze — bardzo prosta w użyciu. Widoczna tutaj funkcja pomocnicza `begin()`, jak widzimy, też należąca do obiektu kontenera, wskazuje początek zbioru już zgromadzonych liczb całkowitych.

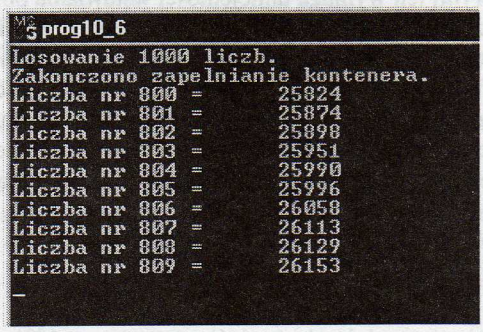
Zauważmy, że zrealizowanie wstawiania liczb w środek (a nie na końcu) ich klasycznie zadeklarowanej tablicy jest zadaniem naprawdę złożonym. Należy bowiem przygotować miejsce na nową daną, rozsuwając dotychczasowe. Przy tym nie może zabraknąć pamięci, zarezerwowanej na całą tablicę. Jeśli pamięci w tablicy brakuje, ambitny, łatwo nie poddający się programista powinien zadeklarować nową, większą tablicę i przepisać do niej zawartość starej.

Ćwiczenie 10.6.

Posortuj bezładnie zapełniony kontener:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
...
vector <int> liczby;
int i, ile = 1000;
cout << "Losowanie " << ile << " liczb." << endl;
for( i = 0; i < ile; i ++)
{
   liczby.push_back( rand());
}
cout << "Zakonczono zapelnianie kontenera." << endl;
sort( liczby.begin(), liczby.end());    //sortowanie kontenera
for( i = 800; i < 810; i ++)
{
   cout << "Liczba nr " << i << " = ";
   cout.width( 10);
   cout << liczby[ i] << endl;
}
...
```

Rysunek 10.6. *Kontener posortowany za pomocą jednej instrukcji*



Biblioteka standardowa zawiera kilka „usługowych" algorytmów, między innymi algorytm sortowania kontenerów:

```
#include <algorithm>
...
sort( liczby.begin(), liczby.end());
```

Funkcja `sort()` otrzymuje informację, gdzie zaczyna się i gdzie kończy zbiór przechowywanych w kontenerze obiektów. I sortuje go za jednym zamachem, bez żadnego dodatkowego wysiłku ze strony programisty.

Podsumowanie

Biblioteka standardowa udostępnia specjalne klasy, które służą do tworzenia i manipulowania zbiorami danych, także obiektów. Klasy te nazywają się kontenerami, a najważniejszym kontenerem jest tablica `vector`.

Uderzającą cechą kontenerów jest to, że „jest im wszystko jedno", jakie zmienne w nich się znajdują. Mogą to być proste liczby, ale także bardzo złożone obiekty, jak choćby cyfrowe modele psów.

Biblioteka standardowa oferuje też kilka algorytmów pomocniczych, operujących na zapełnionych kontenerach (jak np. wspomniany tutaj algorytm sortowania).

Efektywnie pracujący, współczesny programista nie musi znać teorii kontenerów. Wystarczy, że zna ich interfejs — wie, jak dodać zmienną do kontenera, jak ją wyjąć z niego, jak posortować zbiór zmiennych czy odszukać jedną z nich. Zauważmy, że są to bardzo proste, wręcz intuicyjne operacje.

Zakończenie

Język C++ wyraźnie różni się od „klasycznych" języków typu Basic, Fortran czy Pascal, które być może Czytelnik zna. Języka C++ uczymy się po to, by *deklarować obiekty* i *zarządzać nimi* za pomocą *kontenerów biblioteki standardowej*. Te dwie umiejętności dają nam niezwykłą moc, szybkość i pewność programowania. Gdybyśmy w ostatnim przykładzie nie wykorzystali kontenera, a zwykłą tablicę obiektów:

```
Pies psy[100];
```

nasza baza byłaby bardzo sztywna i albo marnowałaby miejsce w pamięci (gdy psów jest 5), albo odmówiła działania po dopisaniu setnego psa. Na programistę czekałoby wiele pułapek, o których aż strach myśleć — choćby obowiązek poszerzenia w odpowiednim momencie tablicy psów.

Cały ten schemat programowania nie byłby potrzebny, gdybyśmy zrezygnowali z obiektów i biblioteki standardowej — w istocie wrócilibyśmy tym samym do starego, trudnego języka C, w którym potrafią efektywnie pracować tylko zawodowi programiści.

Na programowanie obiektowe należy patrzeć jak na *opakowywanie algorytmów* w konstrukcje zwane klasami. Nie jest to trudne — ktoś, kto zadeklaruje i zdefiniuje kilka kompletnych klas, przyswoi sobie składnię opakowywania obiektowego. Twierdzę nawet, że w procesie pisania klas nie ma wiele programowania.

A na klasy, czyli odpowiednio opakowane dane i algorytmy, czekają kontenery, takie jak tablice (`vector`), stosy, listy, kolejki. W obiektach tych typów znajdziemy np. gotowe algorytmy sortowania czy przeszukiwania, które — jak wiemy — uchodzą za złożone i trudne. Te nadzwyczaj skomplikowane struktury danych mamy pod ręką, gotowe do zapełnienia obiektami.

Wynika z tego, że współczesny programista powoli przestaje się uczyć i zajmować budowaniem elementarnych algorytmów, wprowadzając w to miejsce umiejętności korzystania z gotowych rozwiązań.

Ekstremalne programowanie jeszcze nigdy nie było tak łatwe.

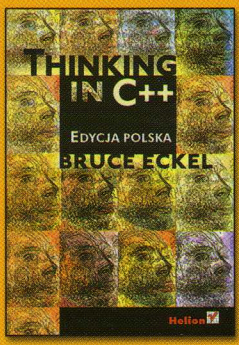
I gdzieś tutaj kończyłby się wstęp do C++. Potrafimy operować pojedynczym obiektem, deklarowanym wprost. Czujemy zapach biblioteki standardowej. Koniec w tym miejscu jest uzasadniony.

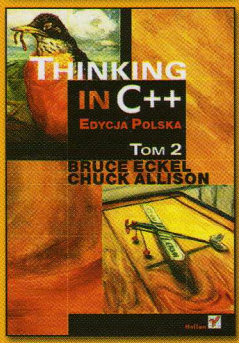
Dalej byłyby techniczne szczegóły budowy klas — *konstruktory kopiujące*, *nowe* znaczenia operatorów, potem mechanizm deklarowania klas w oparciu o tak zwane *dziedziczenie*, no i biblioteka standardowa jako cel najwyższy. Ale to już innym razem.

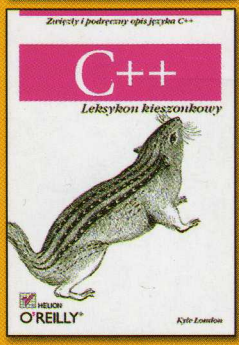
Ćwiczenia praktyczne

Ćwiczenia praktyczne to seria przeznaczona dla czytelników, którzy lubią rozwiązywać problemy i poznawać zagadnienia od podstaw. Każda książka składa się z szeregu ćwiczeń utrwalających zdobywaną wiedzę. Napisane prostym językiem, bogato ilustrowane, opatrzone licznymi przykładami są przyjemną lekturą nawet dla tych, którzy stawiają pierwsze kroki w świecie informatyki.

Książki z tej serii stanowią doskonałe uzupełnienie kursów, dlatego są wykorzystywane przez liczące się ośrodki szkoleniowe w Polsce.







C++

Język C++ jest obecnie najpopularniejszym językiem programowania. Powodów jest kilka: niewielka liczba słów kluczowych, ogromna ilość bibliotek umożliwiających zastosowanie C++ w wielu dziedzinach, a przede wszystkim ogromne możliwości języka, pozwalające na stworzenie praktycznie dowolnej aplikacji. Systemy operacyjne, aplikacje użytkowe, gry – twórcy wszystkich tych programów wykorzystują właśnie język C++.

„C++. Ćwiczenia praktyczne" to książka przeznaczona dla osób, które chcą rozpocząć naukę tego języka. Opisuje najważniejsze zagadnienia niezbędne do pisania programów w C++. Każde z nich zilustrowane jest prostym przykładem. Po lekturze niniejszych ćwiczeń zdobędziesz podstawy niezbędne do dalszej nauki i tworzenia prawdziwych aplikacji.

- Konfiguracja środowiska programistycznego
- Standardowe wejście i wyjście
- Składnia programu
- Sterowanie wykonywaniem programu
- Funkcje
- Typy danych
- Podstawy programowania obiektowego

Szukasz dobrej książki? Zamów najnowszy katalog: http://helion.pl/katalog
Zamów informacje o nowościach: http://helion.pl/nowosci
Zamów cennik: http://helion.pl/cennik

ISBN 83-7361-479-6

Wydawnictwo Helion
ul. Chopina 6, 44-100 Gliwice
skr. poczt. 462, 44-100 Gliwice
(32) 230-98-63, (32) 231-22-19
e-mail: helion@helion.pl http://helion.pl

Informatyka w najlepszym wydaniu

Cena 14,90 zł