Michał 'SasQ' Studencki

Praktyczny Kurs C++ dla każdego

z serii „Programuj gry z Game Creatorem”

Spis treści

Wstęp

Bla bla bla...

Bla bla bla...

Tu będzie jakiś wstępniak „ogólny” dla całej książki...

Bla bla bla...

Bla bla bla...

Wstęp do programowania

Komputer jaki jest, każdy widzi ;-)

Kiedyś, gdy byłem jeszcze małym zielonym lamerem :-D myślałem, że aby zmusić komputer do zrobienia czegoś, to trzeba do niego mówić :-P Znaczy nie dosłownie, tylko pisząc na klawiaturze. [Naoglądało się dziecko goopich filmów ;-D] Komputer wypisywał mi na ekran jakieś teksty po angielsku, więc ja także próbowałem wydawać mu polecenia po angielsku, z nadzieją że wreszcie coś sqma i wykona to, o co go tak ładnie proszę :-)

Pół dnia próbowałem, aż wreszcie przyszedł mój qzyn [właściciel komputra] i widząc moje męki postanowił pokazać mi, jak się programuje. Zaczął wpisywać jakieś dziwne "zaklęcia" i nagle ekran zasypały literki jak w Matrixie! :-) Już po chwili pojawiły się kolory, kreski, kółka, a ja patrzyłem na to wszystko z coraz większym podziwem.

Nieco później qzyn podarował mi swój stary komputer, wraz z książką o programowaniu. Książka ta wiele mi wyjaśniła. Zrozumiałem wtedy, że komputer to jest takie jakby szybkie liczydło na sterydach :-) Potrafi operować wyłącznie na liczbach. Przykładowo jeśli chcesz przy jego pomocy retuszować zdjęcie, musisz go najpierw zamienić na postać cyfrową, bo tylko w takiej postaci komputer go "rozumie" i może go przetwarzać. Podobnie jest z programami. Mikroprocesor w twoim komputerze rozumie tylko cyfrowe kody różnych instrukcji, zwane *kodem maszynowym* lub *kodem binarnym*. Tak więc program komputerowy jest po prostu ciągiem cyfrowych rozkazów umieszczonym gdzieś w pamięci komputera.

Języki programowania

Dla człowieka ten gąszcz cyferek kodu binarnego jest mało zrozumiały i chyba tylko Neo mógłby się w tym połapać ;-D Człowiek wolałby wydawać polecenia w swoim własnym języku. Minie jeszcze wiele czasu zanim komputer zrozumie język ludzki, dlatego powstał kompromis - specjalne uproszczone języki zwane *językami programowania*. Zazwyczaj składają się one z małego zestawu słów [tak zwanych *słów kluczowych*] i mają raczej prostą składnię, by łatwo było je przetłumaczyć na cyfrowy kod maszynowy.

Dzięki językom programowania, programiści nie muszą już pamiętać adresów komórek pamięci ani cyfrowych kodów poszczególnych rozkazów procesora. Zamiast tego mogą posługiwać się nazwami, które łatwo jest zapamiętać. Większość tych nazw mogą wymyślać sami. Najprostszym językiem [choć niekoniecznie najłatwiejszym ;-J], który to umożliwia, jest **Assembler**, w którym każda instrukcja procesora ma swoją skrótową

nazwę [mnemonik]. Z czasem powstały języki bardziej rozbudowane [tzw. *wysokiego poziomu*], w których jedna instrukcja może być tłumaczona na wiele instrukcji procesora. Takimi językami są np. **Pascal**, **C** i jego następca **C++**, oraz wiele innych.

Tekst programu, napisany w języku programowania, nazywamy *kodem źródłowym*, lub po prostu *kodem*. Programiści mówią np. "Pokaż mi swój kod", albo "Znalazłem błąd w twoim kodzie". Z tego też względu programistów nazywa się czasem *koderami* ;-) Taki kod można przechowywać w zwykłym pliku tekstowym, więc możesz go wklepywać w dowolnym edytorze czystego tekstu. Od biedy możesz użyć nawet systemowego notatnika ;-P jednak już wkrótce poznasz o wiele lepsze narzędzia :->

No dobra, ale jak te języki programowania są tłumaczone na kod maszynowy? Są dwa sposoby takiego tłumaczenia, a każdy ma swoje zady i walety ;-)

Kompilatory i interpretery

Pierwszy sposób wykorzystuje specjalny program, zwany *interpreterem*, który na bieżąco tłumaczy polecenia użytkownika na kod maszynowy. Z tej metody korzystał język **BASIC**, którego używał mój qzyn, a obecnie kilka języków skryptowych [np. **PHP**]. Języki interpretowane mają tę zaletę, że można nawet wpisywać polecenia prosto z klawiatury i one od razu się wykonują. Jednak program musi być zawsze "w locie" tłumaczony na kod maszynowy, przy każdym uruchomieniu od nowa, co zazwyczaj wprowadza niepotrzebne opóźnienia i program działa wolniej niż by mógł. Poza tym musisz mieć interpreter danego języka na każdym komputerze, na którym chcesz uruchomić swój program. To jak ciągnąć ze sobą tłumacza na wycieczkę zagraniczną ;-P

Ty chcesz programować gry, więc potrzebujesz wydajnego i szybkiego narzędzia. Nie chcesz przy tym rozprowadzać żadnych dodatkowych programów ze swoją grą. Potrzebujesz tego drugiego sposobu, który nazywa się *kompilowaniem* [nie mylić z "komplikowaniem" ;-P choć może się wydawać, że ta druga nazwa jest bardziej odpowiednia ;-J].

Rys. 1: Kompilator tworzy plik z kodem binarnym na podstawie kodu źródłowego

Drugi sposób uwalnia cię od wad, które miał interpreter. Można przecież wczytać kod źródłowy, przetłumaczyć go "raz a porządnie" i tak powstały kod binarny zapisać sobie w pliku. Tak właśnie robi specjalny program, zwany *kompilatorem*.

Takie tłumaczenie może trochę potrwać. Jeśli kodu źródłowego jest bardzo dużo, to nawet kilka godzin :-P

Ale od tego momentu każde uruchomienie programu nie będzie już wymagać ciągłego "tłumaczenia w locie". Po skompilowaniu program jest już w postaci zrozumiałej dla komputera, więc może się wykonywać z prędkością światła :-D Oznacza to także, że do dalszego działania program nie będzie już potrzebował kompilatora.

Kompilator robi przy okazji coś jeszcze - sprawdza czy nie masz przypadkiem jakiegoś błędu w twoim kodzie źródłowym. Jeśli się okaże że masz, od razu przerwie kompilację i powie ci co jest nie tak. Jest to duża zaleta, bo zazwyczaj jeśli twój kod poprawnie się skompiluje, to uda się go też od razu uruchomić.

Niektóre kompilatory tworzą gotowe *pliki wykonywalne* [zwane też żargonowo *binarkami* ;-)]. Są to takie pliki, które system może już załadować do pamięci i uruchomić. W systemie Windows mają one rozszerzenie `.exe` [od ang. "**exe**cutable"], a w systemie Linux i innych unixowych poznasz je po atrybucie "wykonywalny". Na pewno dobrze znasz takie pliki ;-) a programowanie, mówiąc w dużym skrócie, polega na ich tworzeniu ;-) Jednak najczęściej kompilator tworzy jedynie plik binarny, którego nie można jeszcze uruchomić. Nazywa się go zwykle *plikiem obiektowym* lub skompilowanym modułem. Dlaczego tak jest?

Moduły i biblioteki

Otóż z czasem programy zaczęły się stawać coraz większe. Im większy program, tym trudniej się w nim połapać mając wszystko w jednym pliku z kodem źródłowym. Chyba nikt nie lubi przekopywać się przez kilkadziesiąt stron tekstu, żeby zmienić jeden wyraz! Programiści wymyślili więc, że będzie im łatwiej jeśli rozłożą sobie kod źródłowy na mniejsze pliki, czyli właśnie *moduły*. Ułatwia to pracę przy dużych projektach [np. grach], bo nad każdym modułem może pracować inny programista. Każdy taki moduł jest kompilowany osobno. Jeśli więc ktoś zmieni coś tylko w jednym pliku, to wystarczy że skompiluje ponownie tylko ten jeden moduł. Taka sztuczka może znacznie skrócić czas kompilacji.

Jeśli jeden programista oprogramuje np. korzystanie z plików, może umieścić ten kod w tak zwanej *bibliotece*. Biblioteka to moduł [lub cały zestaw modułów] napisany specjalnie po to, by inni programiści mogli z niego skorzystać, oszczędzając czas, robotę i co najważniejsze nerwy :-) Po co ponownie wymyślać koło jeśli ktoś już to zrobił przed nami? ;-J

Biblioteki rozszerzają możliwości języka. Istnieje wiele gotowych bibliotek przeznaczonych do różnych zadań, napisanych przez fachowych programistów :-J Ponadto niektóre biblioteki są dostępne w wersjach dla różnych systemów. Używając takich bibliotek możesz tworzyć programy działające na różnych platformach [np. Windows, Linux, MacOS]. Wystarczy że skompilujesz program dołączając wersje bibliotek dla danej platformy i sprawa załatwiona!

Biblioteki są przechowywane w postaci już skompilowanej [wedle zasady "Kompilujesz raz, używasz cały czas" ;-D], więc wszystko co musisz zrobić, to połączyć je z twoim programem. Oczywiście używasz tylko tego, czego potrzebujesz, więc masz pewność, że w twojej binarce nie znajdzie się np. kod do obsługi plików, jeśli twój program z tego nie korzysta.

Łączenie do kupy

Jak widać, każdy moduł może być pisany przez innego programistę. Może być nawet tak, że każdy programista pisze w innym [swoim ulubionym] języku programowania i kompiluje swoim kompilatorem. Do tego dochodzą jeszcze biblioteki. Jak więc teraz poskładać te wszystkie pliki binarne do kupy?

Służy do tego kolejne narzędzie programistyczne, zwane *linkerem*. Jego zadaniem jest połączenie wszystkich modułów binarnych w jeden plik wykonywalny, który już będzie można uruchomić :-) Moduły są najczęściej jakoś ze sobą powiązane i korzystają z siebie nawzajem. Linker odpowiada więc także za poprawne połączenie modułów ze sobą, jak kawałków puzzli ;-) Kompilator nie mógłby tego zrobić, bo przerabia każdy plik z kodem źródłowym osobno i nie ma pojęcia co znajduje się w pozostałych modułach.

Rys. 2: Linker łączy wszystkie moduły binarne i biblioteki, tworząc gotowy do uruchomienia program

Różne systemy operacyjne używają różnych formatów dla swoich binarek. Dobry linker potrafi tworzyć pliki wykonywalne o różnych formatach, uzyskując binarki dla różnych systemów :-) Możesz więc pisać programy na Linuxie, a binarki tworzyć dla Windowsów. Musisz tylko pamiętać, że współczesne systemy różnią się nie tylko formatem binarek, więc trzeba też użyć bibliotek właściwych dla danego systemu.

Jak uprościć sobie robotę

Jak widzisz, proces budowy gotowego programu składa się z kilku etapów. Najpierw dla każdego pliku źródłowego z osobna wywoływany jest kompilator, który kompiluje kod źródłowy do postaci pliku z kodem binarnym. Gdy już wszystko zostanie skompilowane,

rozpoczyna pracę linker, który łączy wszystkie moduły binarne i wymagane biblioteki w gotowy plik wykonywalny. Wygląda na kupę żmudnej roboty! Zwłaszcza gdy plików z kodem źródłowym jest dużo.

I tutaj z pomocą przychodzą tak zwane *zintegrowane środowiska programistyczne* [ang. "**I**ntegrated **D**evelopment **E**nvironment", w skrócie **IDE**]. Czyli zestaw narzędzi takich jak: kompilator, linker, edytor kodu źródłowego, i wiele innych narzędzi, połączonych w jeden pakiet. Takie środowiska posiadają zwykle *menedżer projektu*, który pamięta jakie pliki składają się na gotowy projekt. Dzięki temu cały proces budowania programu można wykonać jednym kliknięciem.

Również edytory kodu źródłowego, wbudowane w takie środowiska, są dużo bardziej wypasione ;-) Dla poprawy czytelności oznaczają słowa kluczowe i różne elementy języka różnymi kolorami. Niektóre posiadają też funkcję zwijania bloków kodu lub automatyczne uzupełnianie nazw, co bardzo ułatwia pracę.

Od tej chwili znasz już wszystkie etapy powstawania gotowego programu. Czas więc przystąpić do nauki jakiegoś języka programowania ;-) Wybierz sobie język spełniający twoje potrzeby i skołuj sobie jakieś dobre środowisko programistyczne. Każde środowisko programistyczne ma swój własny zestaw bajerów, więc dobrze się im przyjrzyj, zanim wybierzesz najbardziej odpowiednie dla siebie ;-)

Podstawy języka C++

Czyli o języku C++ :-P

Wstęp do „Kursu podstaw języka C++”

Większość kursów C++, jakie widziałem w necie, zaczyna się od omówienia tego, co C++ odziedziczył w spadku po języku C. Uważam, że takie podejście jest szkodliwe. Przede wszystkim dlatego, że nie każdy, kto chce nauczyć się C++, znał wcześniej C lub jakiś inny podobny język programowania. Niektórzy zaczynają od łysego zera i wcale nie jest im łatwo, gdy autor kursu co chwilę fąfra coś o języku C, którego przecież nie znają! :-P

Poza tym kursy zaczynające od C uczą cię tak naprawdę języka C i skazują na jego ograniczenia :-P Ucząc się z takiego kursu nabierzesz złych nawyków: będziesz próbować w C++ pisać programy w taki sam sposób, jak się to robiło w C. A przecież C++ to jednak nie jest C, lecz zupełnie nowy język z nowymi możliwościami, których w C nie było [te dwa plusy nie są tam dla picu ;-P]. Wiele rzeczy w C++ robi się inaczej, niektóre nawet dużo prościej niż w C. Warto więc skorzystać z tych udogodnień - do tego przecież został stworzony! :-)

Jest jeszcze jedna wada zaczynania nauki od C. Język ten był raczej "niskopoziomowy" i nawet najprostsze rzeczy wymagały od programisty sporej wiedzy o szczegółach działania komputera. C++ jest tak pomyślany, że na początku nauki nie musisz wcale wiedzieć zbyt wiele, by móc skorzystać z jego nowych możliwości i wcale nie musisz posiadać dużej wiedzy, by zrobić proste rzeczy. Doświadczenie możesz więc zdobywać stopniowo, w czasie nauki języka, a techniczne szczegóły poznajesz dopiero wtedy, gdy są ci one potrzebne by zrobić coś bardziej nietypowego. Nie trzeba przecież być elektronikiem, by używać komputera ;-)

Dlatego właśnie w moim kursie postanowiłem przyjąć od początku, że C++ jest pierwszym językiem, którego się uczysz. Będę cię zaznajamiał z nowym stylem pisania programów i od początku będziemy korzystać z nowych możliwości, jakie daje biblioteka standardowa C++. Dzięki temu będziesz w stanie od razu pisać użyteczne programy i na bieżąco widzieć efekty ich działania. Nauka będzie przebiegać stopniowo i przyjemnie ;-)

Jeśli należysz do tych, którzy wcześniej znali C, takie podejście może cię trochę niepokoić. Możesz czuć się trochę niepewnie widząc, że odbiegam trochę od tego, co znasz dobrze z C. Możesz też mieć wątpliwości, czy nowe mechanizmy obiektowe dostępne w C++ są tak samo wydajne, jak te używane w C. Proszę jednak, powstrzymaj się narazie z oceną. W trakcie kursu zobaczysz, że jednak ten nowy styl ma swoje zalety i warto programować w taki sposób ;-) Przekonasz się, że dzięki temu programowanie staje się prostsze, mniej podatne na błędy, a wydajność jest przeważnie taka sama [będę niekiedy wyjaśniał, dlaczego tak jest ;-)].

Najbardziej wkurzały mnie zawsze w innych kursach teksty w stylu: "Narazie nie mogę ci tego wyjaśnić, bo jesteś za głupi, dowiesz się jak wyrośniesz z pieluch [czyli za X lekcji] a narazie rób tak, jak ci mówię, i nie zadawaj durnych pytań" :-P Dlatego w moim kursie będę się starał unikać takiego podejścia i wyjaśniać wszystko od razu na tyle, na ile się da, by działanie programu było dla ciebie zrozumiałe. Bardziej szczegółowe wyjaśnienia czasami będę odkładał do czasu, aż okażą się konieczne by coś zrozumieć. Jednak będę tak robił tylko po to, by nie przeciążać ci mózgownicy informacjami, które na początku nauki i tak nie będą ci jeszcze potrzebne ;-)

No, to by było tyle ględzenia na wjazd ;-P Zapraszam do czytania i życzę przyjemnej nauki :-)

Budowa programu

Przed tobą pierwsza lekcja kursu C++, z której dowiesz się wreszcie, jak wygląda pisanie programów w języku C++ :-) Odpal teraz swoje ulubione środowisko i utwórz nowy projekt programu tekstowego. Jeśli w edytorze jest już jakiś kod, skasuj go. Zaczniemy z pustym plikiem, który stopniowo będzie zapełniać się kodem. Będziemy tworzyć nasz program od podstaw, niczym doktor Frankenstein swojego potwora ;-D Początkowo program nie będzie jeszcze robił nic efektownego, jednak dzięki tym przykładom, które dokładnie omówię, będzie ci łatwiej obczaić o co w tym całym progamowaniu chodzi :-)

Na początek wpisz [albo wklej] poniższy kod:

```
int main()
{
  //To jest komentarz
}
```

Komentarze

Na pewno od razu rzuciło ci się w oczy, że pewien fragment wygląda tu zrozumiale. W języku C++ możesz umieszczać w kodzie programu fragmenty tekstu w "ludzkim" języku. Takie fragmenty to *komentarze*. Kompilator do nich nie zagląda, więc możesz w nich pisać co tylko chcesz ;-) Musisz tylko oznaczyć kompilatorowi gdzie komentarz się zaczyna, a gdzie kończy.

Komentarz liniowy, taki jak ten powyżej, zaczyna się znakami `//` i ciągnie się do końca linii. Istnieją także *komentarze blokowe*, które mogą obejmować dowolny fragment tekstu [nawet ciągnący się przez wiele linii]. Zaczynają się od `/*` i ciągną się aż do `*/`. Przykład komentarza blokowego:

```
/* To jest komentarz, który
  ciągnie się przez wiele linii */
```

Komentarzy takich nie można zagnieżdżać, czyli umieszczać jeden w drugim. Dlaczego? Bo gdy kompilator napotka zakończenie tego wewnętrznego komentarza, to uzna, że komentarz właśnie się skończył. Znów zacznie pilnować czy nie robisz błędów i będzie bardzo zdziwiony, że nie piszesz w jego języku ;-P

Możesz się zastanawiać, po co ci komentarze w twoim programie? Przecież jak piszesz program, to chyba wiesz o co w nim chodzi? :-P No tak, w tej chwili wiesz. Jednak wyobraź sobie, że za jakiś czas wracasz do swojego kodu chcąc coś poprawić. Patrzysz na te hieroglify i myślisz sobie: "Ja to napisałem?" :-D

Dobra, może nie każdy koduje po flaszce ;-D ale wystarczy już tydzień czy dwa żeby zapomnieć jak dokładnie działał twój program. Wtedy masz jeszcze komentarz i nie musisz rozgryzać działania własnego kodu ;-) Możesz też zostawiać sobie notatki w kodzie, np. że coś musisz jeszcze poprawić lub dodać. Staraj się tylko nie przesadzać z dokładnością komentarzy. Nie pisz tego, co można wyczytać z samego kodu.

Komentarze przydają się też do innej rzeczy. Wiesz już, że kompilator nie zagląda do komentarzy. Co się stanie jak zamkniesz w znaki komentarzy jakiś fragment twojego kodu? Zgadza się! Kod przestanie być kodem, a stanie się komentarzem. Ukryjesz go w ten sposób przed kompilatorem ;-) Tej sztuczki możesz używać, by wyłączać sobie wybrane fragmenty programu gdy testujesz jego działanie.

Ok, ale ludzki język już znasz, tego nie muszę cię uczyć ;-) Przejdźmy więc do poznawania języka C++. Zobaczmy co nam jeszcze zostało do omówienia w poprzenim przykładzie.

Pora na kod

Oprócz komentarza, w naszym pierwszym programie jest jeszcze coś takiego:

```
int main()
{

}
```

Ten fragment to absolutne minimum. Każdy program w C++ musi zawierać conajmniej to.

Długo myślałem jak wytłumaczyć ci ten pierwszy fragment kodu, gdyż łączy on w sobie wiele składników języka C++, które chciałoby się dokładniej omówić już teraz. W końcu postanowiłem, że zamiast na wjazd zawalać cię informacjami, omówię wszystko po kolei, stopniowo odkrywając przed tobą coraz więcej. Niektóre rzeczy omówię tylko w skrócie, dokładniejsze wyjaśnienia zostawiając na później.

Wszystko ma swój początek. Zacznijmy więc od niego! :-)

Bloki instrukcji

Każdy program to po prostu ciąg instrukcji. Instrukcje są jak zdania, a każde opisuje jedną prostą czynność, którą program ma wykonać. W starożytnych językach programowania pisało się program w postaci jednego długiego ciągu instrukcji, ciągnącego się jak spaghetti ;-P Szybko się okazało, że w takim kodzie łatwo się pogubić. Programiści zaczęli więc dzielić program na części.

W języku C++ taki wydzielony ciąg instrukcji, który stanowi jednolitą całość, nazywany jest *blokiem*. Blok zamyka się w nawiasy klamrowe, by wyraźnie określić jego początek i koniec. Wygląda to tak:

```
{
  //Tu jakieś instrukcje...
}
```

W programie może być oczywiście więcej bloków. Niektóre fragmenty kodu są na tyle przydatne, że programista chciałby ich używać wielokrotnie. Żeby jakoś rozróżniać bloki i ułatwić odwoływanie się do nich, programiści zaczęli nadawać im nazwy. Taki blok instrukcji, który ma swoją nazwę, nazywamy *procedurą* lub *funkcją*.

> W niektórych językach programowania funkcja i procedura to dwie różne rzeczy: procedura to nazwany blok kodu, który można wielokrotnie wywoływać, natomiast funkcja dodatkowo może zwracać pewną wartość, tak jak funkcja matematyczna. W języku C++ nie ma takiego rozróżnienia. Jeśli jakiś blok kodu ma nazwę, to jest funkcją i już, nawet jeśli nic nie zwraca.

Gdy zamienisz jakiś blok kodu w funkcję, nadając mu nazwę, blok przestaje już być "anonimowy" i możesz odwoływać się do niego wielokrotnie, czyli wywoływać funkcję, używając jej nazwy ["Azor! Do nogi!" ;-D].

Funkcja główna

Jeśli jest kilka nazwanych bloków kodu [czyli funkcji], to od którego z nich program ma się zacząć wykonywać? Żeby rozwiązać ten problem, twórcy języka C++ umówili się, że każdy program będzie się zaczynał w funkcji o nazwie `main` [ang. "main" = "główna". Czytaj "mejn"]. Jest to więc pierwsza funkcja, która wykona się w twoim programie, i zarazem ostatnia. Można by rzec, że funkcja `main` jest twoim programem ;-)

A oto jak wygląda owa funkcja main w całości:

```
int main()
{
  //Tu będą instrukcje
}
```

Jak widzisz, jest tutaj blok, a przed nim widać nazwę funkcji, czyli `main`. Po nazwie stoi dodatkowo para nawiasów okrągłych. W przyszłości w tych nawiasach znajdą się parametry dla twojego programu. Narazie twój program nie pobiera żadnych parametrów, bo żadnych nie potrzebuje, więc nawiasy są puste. Mimo to te nawiasy muszą tam być, bo właśnie po nich kompilator rozpoznaje, że tworzysz nazwę dla funkcji.

> W innych językach trzeba to było dodatkowo sygnalizować różnymi słowami kluczowymi, takimi jak **`function`** czy **`procedure`**. W C++ wystarczy para nawiasów po nazwie.

Tajemnicze int

Pewnie zastanawiasz się jeszcze, co znaczy to słówko **`int`** przed nazwą funkcji? ;-J

Otóż, gdy funkcja main się kończy, kończy się też program i oddaje sterowanie temu, kto go uruchomił [tym kimś może być system operacyjny lub inny program]. W tym momencie twój program może dodatkowo zwócić informację o wyniku swojego działania [coś jak "testament" ;-D]. Może to być np. wynik obliczeń, informacja czy program się powiódł czy nie... możliwości jest wiele. Zbyt wiele! Dlatego ustalono, że wynikiem programu ma być zawsze jakaś pojedyncza liczba. Jeśli tą liczbą jest `0`, oznacza to umownie, że program przebiegł prawidłowo. Każdy inny wynik oznacza, że program zakończył się z powodu błędu. Wtedy różne wartości oznaczają różne kody błędów.

Słowo kluczowe **`int`** oznacza właśnie, że wynikiem funkcji `main` [a więc zarazem twojego programu] ma być liczba całkowita. Więcej na temat tego słowa kluczowego powiem ci już wkrótce. A narazie zastanów się, jaką liczbę zwraca ten twój pierwszy program? :->

Pytanie trochę podchwytliwe, bo nie ma tu żadnej liczby :-D Jednak nie ma też narazie żadnych instrukcji, jedynie pusty blok funkcji `main`, więc program po prostu zakończy się od razu po uruchomieniu. Skoro nic nie może pójść źle, to wynikiem programu powinno być `0` i postara się o to sam kompilator.

Sprawdź teraz, czy tak jest naprawdę. Skompiluj ten pierwszy program i uruchom go. Jeśli twoje środowisko programistyczne pozwala ci zobaczyć wynik działania programu, to świetnie :-) [takim środowiskiem jest na przykład Code::Blocks[I]]. Jeśli nie - no cóż, musisz uwierzyć mi na słowo ;-P

No dobrze, a co zrobić, by program zwrócił jakąś inną wartość?

Zwracamy wartość, czyli pierwsza instrukcja

Odpowiedź brzmi: musisz napisać w programie, by to zrobił. On sam się tego nie domyśli :-D Komputer jest jak Dżin z lampy, który potrafi jedynie spełniać życzenia programisty ;-) Jednak nie kiwnie palcem, dopóki mu nie każesz. Usiądź teraz wygodnie, bo właśnie napiszesz swoją pierwszą *instrukcję* :-) Wywal z poprzedniego kodu ten komentarz i w jego miejsce wstaw coś takiego:

```
  return 7;
```

Cały program powinien teraz wyglądać tak:

```
int main()
{
  return 7;
}
```

W języku C++ każda instrukcja kończy się średnikiem. Jest tak dlatego, że C++ nie zmusza cię do umieszczania jednej instrukcji w jednej linijce. Możesz rozpisać jedną instrukcję nawet w kilku linijkach, jeśli tak jest bardziej przejrzyście. Jest tak dzięki temu, że to średnik [a nie koniec linijki] mówi kompilatorowi, w którym dokładnie miejscu instrukcja się kończy. Możesz nawet zapisać cały twój program tak:

```
int main(){return 7;}
```

jednak widzisz chyba, że wygląda to jak jakieś łokoczoko ;-) Kompilator to zrozumie, ale człowiek zbaranieje na widok czegoś takiego :-D Dlatego warto układać program w wielu linijkach, dodając dodatkowe odstępy i wcięcia. Wtedy program staje się czytelniejszy.

Jest też drugi powód, by tak robić - używanie *debuggera*. Debugger to narzędzie, które pozwala ci śledzić wykonanie programu krok po kroku i pokazuje, w której linijce aktualnie się zatrzymał. Jeśli będzie to wciąż jedna i ta sama linijka, to nie wyczujesz którą aktualnie instrukcję twój program wykonuje :-P

Dobra ale wróćmy do tego return. Średnik na końcu mówi nam, że jest to instrukcja. Słowo kluczowe `return` w tej instrukcji oznacza, że chcemy zakończyć funkcję. Tuż po nim stoi liczba `7` i to ona będzie zwrócona systemowi jako wynik twojego programu. Dokładniej o słowie kluczowym `return` opowiem w lekcji poświęconej funkcjom.

Tak więc pierwsze starcie z C++ masz już za sobą :-) Możliwe że budzi się już w tobie ciekawość czego możesz się dowiedzieć z następnej lekcji ;-) Jednak zanim do niej przeskoczysz, spróbuj powstrzymać się jeszcze chwilę i skompiluj ten program by sprawdzić, czy naprawdę jego wynikiem będzie 7 ;-D

Pisanie na ekranie

Witaj znowu :-) W poprzedniej lekcji powiedziałem ci, jak rozpocząć twój program i jak go zakończyć. Jednak czym byłby twój program bez świata zewnętrznego? [Albo może czym byłby świat zewnętrzny bez twojego programu? ;-D] W lekcji, którą właśnie czytasz, dodamy w programie możliwość komunikacji ze światem zewnętrznym. Wreszcie napiszesz program, który będzie robił coś widocznego ;-) Uruchom swoje środowisko programistyczne i otwórz projekt z poprzedniej lekcji, bo będziemy go rozbudowywać.

Standardowe wejście/wyjście

Każdy program [nawet najprostszy] może korzystać z prostego mechanizmu "rozmowy" ze światem zewnętrznym. Ten najprostszy mechanizm to *standardowe wejście/wyjście*. Jest to najbardziej prymitywny mechanizm, jednak bardzo uniwersalny i do większości celów wystarczający. *Standardowe wejście* służy do przyjmowania danych od użytkownika, a *standardowe wyjście* do pokazywania mu wyników [jest jeszcze kilka innych takich "kanałów", jednak narazie nie będą nam potrzebne].

Standardowe wejście jest zwykle powiązane z klawiaturą, a standardowe wyjście z ekranem tekstowym, które razem nazywane są *konsolą*. Dawniej [zanim wymyślono monitory] standardowe wyjście prowadziło do drukarki lub dalekopisu ;-P Obecnie systemy okienkowe "symulują" konsolę w postaci okienka tekstowego [np. słynne "okienko MS-DOS" ;-P], z którym wiążą standardowe wejście/wyjście.

> Tu drobna uwaga. Windows tworzy dla programu okno konsoli, jednak po skończeniu programu od razu zamyka to okienko ;-P więc możesz nie zdążyć zobaczyć wyników programu. Z kolei Linux w trybie graficznym nie tworzy okna tekstowego [zostawia decyzję użytkownikowi], więc to, co wypisuje się na standardowe wyjście, może być niewidoczne. Żeby zaradzić tym problemom, najlepiej jest odpalać programy w już otwartym oknie konsoli.

Standardowe wejście/wyjście można też przekierowywać do pliku na dysku. Ponieważ w systemach unixowych wszystko jest plikiem, można na nich powiązać wejście/wyjście także z drukarką, portem USB, kartą dźwiękową, albo nawet puścić przez sieć do odległego komputera :-) Jak widzisz, w tym prostym mechaniźmie drzemią ogromne możliwości ;-D

Żeby współpraca z tak szeroką gamą różnych urządzeń była możliwa, standardowe wejście/wyjście działa w sposób "strumieniowy". Nazwa wzięła się stąd, że znaki tekstu są wysyłane/odbierane "taśmowo", jeden za drugim, tak jakby płynęły strumieniem ;-) Czasami spotkasz się z nazwą *strumienie wejścia/wyjścia*, więc przygotuj się na to ;-)

Biblioteki i pliki nagłówkowe

Oczywiście nie każdy program potrzebuje korzystać ze strumieni wejścia/wyjścia. Dlatego nie jest to częścią samego języka C++. Zamiast tego jest częścią jego *Biblioteki Standardowej*, czyli takiej, którą posiada każdy kompilator C++. Biblioteka standardowa jest automatycznie linkowana z twoim programem, więc nie musisz tego robić ręcznie. Aby jednak móc skorzystać ze strumieni wejścia/wyjścia, musisz zrobić coś jeszcze. Musisz w swoim programie wstawić tak zwany *plik nagłówkowy*.

Taki pliki zawiera normalny kod źródłowy i zapoznaje kompilator z nazwami użytymi w bibliotece. Dołączając plik nagłówkowy, uczysz twój kompilator nowych sztuczek ;-) Kompilator czyta sobie ten plik i poznaje nowe słowa, nowe polecenia, których od tej pory pozwoli ci już używać w twoim programie, bo będzie je znał i nie będzie panikował, że może robisz coś źle ;-) Będzie też pilnował, czy używasz biblioteki we właściwy sposób. Jedna biblioteka może obsługiwać kilka rzeczy naraz, więc plik nagłówkowy określa także, z których możliwości biblioteki korzystasz.

Wstawiamy nagłówek

Wszystko, co się wiąże z obsługą strumieni wejścia/wyjścia, znajduje się w nagłówku `iostream` [ang. "**i**nput/**o**utput **stream**" = "strumień wejścia/wyjścia"]. Dopisz więc na samym początku twojego kodu:

```
#include <iostream>
```

Jest to tak zwana *dyrektywa preprocesora*. Każda taka dyrektywa zaczyna się znakiem `#` [płotkiem ;-)] i trwa do końca linijki. Dyrektywy są przetwarzane jeszcze zanim kompilator dobierze się do tego pliku! [na etapie tzw. "preprocessingu", czyli wstępnego przygotowania kodu źródłowego do kompilacji.] Dawniej robił to osobny program, zwany *preprocesorem*, teraz jest on zwykle wbudowany w kompilator. W przyszłości poznasz więcej takich dyrektyw dla preprocesora, jednak narazie mamy ciekawsze rzeczy do poznania ;-)

Dyrektywa `#include` nakazuje wstawić w tym miejscu [ang. "include"="wstaw"] zawartość nagłówka `iostream`. Efekt jest taki, jakby kod źródłowy z tego nagłówka został wpisany w miejscu dyrektywy. Kompilator zapozna się z nim i pozwoli ci używać nazw w nim zawartych.

Nazwy plików nagłówkowych dla bibliotek podaje się w nawiasach ostrych ["dzióbkach" ;-J]. Kompilator będzie ich wtedy szukał w swoim katalogu z nagłówkami. Jeśli zamiast tego chcesz dołączyć własny plik nagłówkowy, który masz w katalogu ze swoim projektem, jego nazwę podajesz w cudzysłowach:

```
#include "mojplik.h"
```

Jak widzisz, twoje własne pliki nagłówkowe powinny mieć rozszerzenie `.h` . To samo tyczy się innych bibliotek. Jedynie pliki nagłówkowe Biblioteki Standardowej nie mają rozszerzenia, bo są one szczególną częścią języka C++ i muszą się jakoś wyróżniać ;-)

Dobra, plik nagłówkowy dołączony, więc kompilator już wie, jak pisać po ekranie tekstowym :-) Teraz kolej na ciebie ;-D

"Witaj, świecie!"

Wiesz już, że gdy wrzucisz coś do strumienia płynącego na standardowe wyjście, to pojawi się to na konsoli. Wiesz także, że twój kompilator przeczytał sobie nagłówek iostream i wie, co to są strumienie. Żeby skorzystać ze strumienia wyjściowego musisz jeszcze tylko wiedzieć, jaką nazwę ten strumień posiada ;-) No dobra, nie będę cię dłużej trzymał w napięciu ;-D Rzuć okiem na poniższą instrukcję:

```
std::cout << "Siema, wiara!";
```

Nazwa `cout` oznacza właśnie strumień wyjściowy :-) [od ang. "**c**onsole **out**put", czytaj "si ałt", a nie "śi ałt", bo trza być twardym a nie miętkim ;-D]. To jest właśnie jedna z nazw, która była w nagłówku `iostream`.

Jak widzisz, jest ona poprzedzona przedrostkiem `std::`. Jest tak dlatego, że nazwa `cout` pochodzi z Biblioteki Standardowej. Wszystkie nazwy w tej bibliotece są umieszczone w tak zwanej *przestrzeni nazw* o nazwie `std`. Twórcy Biblioteki Standardowej zrobili tak po to, żeby nazwy w niej zawarte nie kolidowały z nazwami wymyślanymi przez ciebie. Dlatego każda nazwa z Biblioteki Standardowej będzie zaczynać się od `std::`. Gdy dojdziesz już do lekcji o przestrzeniach nazw, wyjaśnię ci to dokładniej i powiem też, co można zrobić, żeby nie pisać wszędzie tego `std::` ;-) A narazie używaj pełnej nazwy strumienia wyjściowego, czyli `std::cout`.

Ale wróćmy narazie do tego, o czym jest ta lekcja. Powyższa instrukcja sprawia, że tekst `"Siema, wiara!"` poleci, zgodnie z kierunkiem "strzałek" `<<`, do strumienia wyjściowego [`cout`] i popłynie sobie tym strumieniem prosto na ekran konsoli ;-) Zobacz jak wygląda nasz drugi program w całości:

```
#include <iostream>
int main()
{
  std::cout << "Siema, wiara!";
}
```

Sprawdź teraz, jak wygląda efekt działania takiego programu. Skompiluj go i uruchom. Jeśli twoje środowisko programistyczne oferuje podgląd konsoli, skorzystaj z niego. Jeśli tworzy systemowe okienko konsoli, to powinieneś zobaczyć go zaraz po uruchomieniu. Niektóre środowiska będą go tam pokazywać, dopóki go nie zamkniesz [np. Code::Blocks poczeka, aż wciśniesz klawisz Enter]. Jeśli jednak okienko zamyka się od razu, to postaraj się uruchamiać programy w już otwartym oknie konsoli systemowej.

Efekt tego programu powinien wyglądać tak:

```
Siema, wiara!
```

Jeśli jest inny, to znaczy że pomyliły ci się programy ;-D Możesz teraz chwilę sobie "podłubać" w tym kodzie. Na przykład zmień wyświetlany tekst na jakiś inny. Jak już się oswoisz z działaniem tego programu, zastanów się, co należałoby zrobić, żeby wyświetlić coś takiego:

```
Siema, wiara!
Jak leci?
```

Hehe ;-) Domyślam się, że pierwszą myślą, jaka ci przyszła do głowy, było dopisanie kolejnej instrukcji takiej jak ta poprzednia, czyli mniej więcej coś takiego:

```
#include <iostream>
int main()
{
  std::cout << "Siema, wiara!";
  std::cout << "Jak leci?";
}
```

Jeśli tak, to na pewno zdziwiło cię, że efekt takiego programu był trochę inny niż zamierzony:

```
Siema, wiara!Jak leci?
```

Dlaczego tak się stało? Czemu teksty pokazały się jeden za drugim?

Obiekty zmienne i stałe

Każdy program korzysta w jakiś sposób z danych. Bez danych, na których program ma pracować, nie ma on przecież żadnego sensu! :-P Programy pobierają dane od użytkownika, coś z nimi robią i zwracają wyniki. Dane muszą być gdzieś przechowywane. Oczywiście do tego celu służy pamięć operacyjna. Jednak żaden programista czy użytkownik nie jest w stanie spamiętać cyfrowych adresów tysięcy komórek pamięci, z których program korzysta! :-P

I tu z pomocą przychodzą języki programowania i kompilatory. Dzięki nim możesz nadawać nazwy [etykietki] dla miejsc w pamięci. Od tej chwili nie musisz już pamiętać cyfrowych adresów, bo kompilator pamięta je za ciebie. W czasie kompilacji, gdy będzie generował kod binarny, pozamienia sobie te twoje nazwy na adresy odpowiednich miejsc w pamięci. Pilnuje on przy tym, by każde miejsce w pamięci miało swoją niepowtarzalną nazwę.

Odkąd posługujemy się nazwą, która jednoznacznie identyfikuje jakieś miejsce w pamięci (wyróżnia go spośród innych), staje się ono już dla nas czymś konkretnym. Pewnym konkretnym obiektem, do którego możemy się odwoływać właśnie za pomocą tej nazwy.

> Pewnie coniektórzy mnie teraz zlinczują że używam określenia "obiekt" już na tym etapie. Jednak zrobiłem to celowo, żeby już od początku uczyć właściwego sposobu myślenia o programowaniu. Może się to wydawać z początku dziwne, jednak szybko okaże się, że takie nazewnictwo jest bardziej logiczne.

Obiekt to jest więc takie cuś, co odróżnia się od pozostałych połaci pamięci ;-) Ma swoją wyjątkową nazwę :-P i w dodatku potrafi przechowywać dane. Ponadto obiekty mogą być zmienne lub stałe. Najpierw zajmiemy się tymi pierwszymi, bo są bardziej przydatne ;-)

I http://codeblocks.org/