# Spis treści

# Kompilatory C++

## Czym jest kompilator?

Pisząc program, zarówno w C++, jak i każdym innym języku możemy go w zasadzie pisać w dowolnym edytorze tekstowym, chociażby notatniku. Ten kod programu, który my piszemy jest nazywanym **kodem źródłowym**. Jednak posiadając sam kod źródłowy nie możemy z całą pewnością stwierdzić, że program dobrze działa, albo nawet, że robi to co miał robić - do tego jest właśnie potrzebny kompilator.

Kompilator umożliwia przekształcenie kodu źródłowego do **kodu wynikowego**, który możemy już bezpośrednio uruchomić na komputerze. W momencie kompilacji (czyli przetwarzania kodu źródłowego do kodu wynikowego), następuje prawdziwy test poprawności składni napisanego przez nas kodu. To tutaj, jeśli zrobiliśmy jakiś błąd, np. zapomnieliśmy średnika, zostaniemy poinformowani o błędzie.

Jeśli kompilator wykryje w naszym kodzie chociaż jeden **błąd** - kończy swoje działanie. Kod wynikowy nie zostaje otrzymany, a o błędach jesteśmy zazwyczaj w jakiś sposób poinformowani. Niestety rzadko kiedy łatwo pojąć gdzie popełniliśmy błąd, nawet gdy przeczytamy dokładnie listę błędów. Wszystkie błędy warto zaczynać eliminować od pierwszego błędu, bowiem dość często zdarza się, że my popełniliśmy tylko jeden błąd, natomiast kompilator wypisuje listę np. dwudziestu błędów.

Oprócz błędu, kompilator podczas kompilacji może wyświetlić **ostrzeżenie**. Ostrzeżenie, jest zazwyczaj odróżniane od błędu poprzez dodanie przed właściwym komunikatem słowa **Warning**, co po polsku znaczy właśnie ostrzeżenie. Jeśli podczas kompilacji pojawi się jedno lub więcej ostrzeżeń, a nie pojawi się żaden błąd, to kod wynikowy zostanie otrzymany i możemy już spróbować uruchomić program.

O ile w przypadku błędów sprawa jest oczywista - musisz się ich pozbyć, żeby otrzymać kod wynikowy i móc uruchomić program, to w przypadku ostrzeżeń ktoś może sobie pomyśleć - przecież kod wynikowy już mam, to po co mam się martwić ostrzeżeniami. Sprawa jest jednak nieco bardziej skomplikowana - ostrzeżenia informują nas często o możliwości zaistnienia błędu innego typu - błędu logicznego w danym miejscu. Dlatego też jeśli kompilator ma opcję powiadamiania o ostrzeżeniach, warto ją mieć włączoną i postarać pozbyć się wszystkich warningów. Ja sam w przypadku wszystkich poważniejszych programów właśnie tak robię.

Jak już wspomniałem, podczas kompilacji, kompilator może wypisać listę błędów, jakie popełniliśmy. Błędy wyświetlane przez kompilator to **błędy składniowe** - czyli takie błędy, które można by powiedzieć, że dla nas jako dla człowieka mogą nie mieć większego znaczenia, bowiem patrząc na program z jednym małym błędem, nam udałoby się zrozumieć co program robi. Kompilator natomiast nie jest taki mądry, dlatego potrzebuje mieć napisane wszystko idealnie według zasad, które zna. Oprócz błędów składniowych istnieją już wspomniane wcześniej przeze mnie **błędy logiczne** - to one sprawiają, że program nie działa w sposób przez nas oczekiwany, bo po prostu coś źle wymyśliliśmy. Włączenie ostrzeżeń i ich eliminowanie przez programistę, może zapobiegać niektórym z tego typu błędów.

## Jak wybrać kompilator?

Wybranie kompilatora nie jest naprawdę rzeczą łatwą. Najbardziej godnym do polecenia kompilatorem jest kompilator, który jest maksymalnie zgodny ze standardem języka C++.

Jeśli chodzi o funkcjonalność, istnieją kompilatory w dwóch wersjach - kompilatory połączone z edytorem - to takie, które są w postaci "okienkowej". Kompilacja odbywa się za pomocą jednego kliknięcia w odpowiedni przycisk. Oprócz tego istnieją też kompilatory, które nie są zintegrowane ze środowiskiem graficznym. Sam kod źródłowy piszemy w dowolnym edytorze (może być nim nawet windowsowy Notatnik), a następnie za pomocą konsoli uruchamiamy kompilator wskazując mu, który plik ma skompilować.

Jeśli posiadasz system operacyjny **Linux**, to stoisz na nieco lepszej pozycji. Do kompilowania programów w C++ służy kompilator **g++**, który jest kompilatorem dość dobrym i jednocześnie dość wymagającym - i o to właśnie powinno nam chodzić. Nie jest zintegrowany z żadnym edytorem, co zwłaszcza początkujących użytkowników może nieco odstraszać. Nie znam w zasadzie innych kompilatorów języka C++ dla linuxa, ale tylko dlatego, że wspomniany g++ spełnia w pełni moje wymagania.

Ja obecnie używam kompilatora g++ w wersji **3.3.2**. Tobie również polecam zaopatrzenie się w jedną z nowszych wersji kompilatora. Kompilator g++ jest standardowo dołączany do dystrybucji linuxa, więc jeśli nie masz go nawet zainstalowanego, to powinien się on znajdować na jednej z płyt CD należących do dystrybucji. Najnowsze informacje o kompilatorze g++, możesz znaleźć na stronie http://gcc.gnu.org/. Tutaj możesz też pobrać najnowszą wersję kompilatora.

Sytuacja niestety komplikuje się, jeśli używasz systemu operacyjnego **Windows**. Kompilatorów C++ dla tego systemu jest bardzo dużo, jednak większość z nich tak naprawdę nie jest zgodna ze standardem języka C++. Kompilatory te wprowadzają pewne ograniczenia i rozszerzenia, co niestety powoduje tylko niepotrzebne zamieszanie i brak przenośności kodów źródłowych.

Mimo, że w ostatnim czasie sytuacja wśród kompilatorów dla windows poprawia się na korzyść, to nadal niestety moim faworytem pozostaje linuxowy g++. Na szczęście istnieje kompilator dla windowsa oparty na g++, ale o tym za chwilę.

*Ponieważ wszystkie zamieszczone w tym kursie kody programów będą napisane w języku C++ zgodnym ze standardami, to niestety może się zdarzyć, że część z nich na niektórych kompilatorach nie będzie działać. Jest to dość częste zjawisko i jeśli tylko Ty zaopatrzysz się w któryś z wymienionych kompilatorów, nie powinno Cię zbytnio martwić, że Twój program może się nie chcieć kompilować na innym kompilatorze.*

Przejdźmy zatem do kompilatorów dla systemu operacyjnego windows. Jak już wspomniałem, istnieje kompilator windowsowy oparty na g++. Tym kompilatorem jest kompilator **Dev-C++**. Udostępnia on całe środowisko graficzne i jest polecany dla początkujących użytkowników, gdyż programy kompilujemy jednym kliknięciem. W rzeczywistości Dev-C++ nie jest kompilatorem - jest nakładką na kompilator g++ dla systemu windows, umożliwiającą wygodniejszą pracę z kodem źródłowym programów i łatwiejsze kompilowanie. Dla naszych potrzeb przyjmiemy jednak, że jest to kompilator.

Obecnie Dev-C++ jest jedynym kompilatorem pod systemem windows którego używam. Ma on dużą przewagę nad pozostałymi kompilatorami odnośnie zgodności ze standardem języka, mimo że w ostatnim czasie pojawiają się coraz częściej kompilatory, które są w pełni zgodne z ANSI C++ (nareszcie).

Po instalacji programu często zdarza się, że pojawiła się już nowa wersja kompilatora g++, jednak na szczęście Dev-C++ umożliwia dość łatwą i bezbolesną aktualizację. Po zainstalowaniu Dev-C++ i jego uruchomieniu wybieramy z menu górnego **Narzędzia**, następnie **Sprawdź, czy są nowe uaktualnienia/pakiety** a w końcu **Check for updates**. O ile jesteśmy podłączeni do internetu, po chwili pojawi się lista uaktualnień, które możemy zainstalować. W tym wypadku zainteresuje nas gcc i g++ i jeśli takie pozycje znajdą się na liście, warto zaznaczyć je, a następnie kliknąć **Download selected** i po chwili będziemy się mogli cieszyć nowszą wersją kompilatora.

Jeśli planujesz kompilować programy na systemie windows, zachęcam Cię właśnie do skorzystania z Dev-C++, bo jak widzisz, jest on dość przyjazny dla użytkownika. Najnowsze informacje o kompilatorze oraz najnowszą wersję kompilatora możesz znaleźć na stronie http://www.bloodshed.net/dev/devcpp.html. Na dodatek w dziale dokumentacja na w/w stronie możesz znaleźć artykuły opisujące podstawy programowania w C++.

# Lista kompilatorów

Poniżej znajduje się krótka lista adresów stron znanych kompilatorów:

**Linux** :

- http://gcc.gnu.org/ - g++

**Windows** :

- http://www.bloodshed.net/dev/devcpp.html - **Dev-C++**
- http://www.mingw.org/ - **MinGW**
- http://www.delorie.com/djgpp/ - **DJGPP**
- http://www.rhide.com/ - **RHIDE** (środowisko graficzne m.in. dla DJGPP)

Niewątpliwie nie jest to zbyt bogaty zbiór kompilatorów, ale jak już wspominałem, dość ciężko znaleźć coś właściwego. Możesz się tylko pocieszyć tym, że jeśli udało Ci się przeczytać do tej pory cały ten artykuł, to i tak wiesz, który z nich wybrać.

# g++ - bardzo krótka instrukcja obsługi

Jeśli wiesz jak skompilować program, nie musisz tego czytać, w przeciwnym wypadku - przeczytaj. Nieważne, czy będziesz pracować pod linuxem w terminalu czy w trybie graficznym (X-Window). Upewnij się natomiast, że posiadasz jakikolwiek edytor tekstowy (dla terminala np. vi, vim, joe, a dla X-Window np. nedit, kwrite).

Jeśli pracujesz w trybie okienkowym, uruchom teraz terminal, a jeśli pracujesz w terminalu, nie musisz nic uruchamiać. Następnie wpisz poniższą komendę (po każdej linii naciskaj enter):

g++

Jeśli zostanie wypisany komunikat "no input files" lub podobny, to znaczy, że posiadasz zainstalowany kompilator g++, jeśli natomiast zostanie wypisany komunikat "No such file or directory", to znaczy, że niestety nie posiadasz zainstalowanego kompilatora i musisz go zainstalować. Jeśli nie wiesz w jaki sposób zainstalować kompilator g++, poproś bardziej doświadczoną osobę o pomoc.

Teraz utwórz za pomocą dowolnego edytora tekstowego plik - jako jego treść możesz przepisać któryś z pierwszych przykładów mojego kursu. Zapisz plik i nadaj mu rozszerzenie **cpp**, czyli plik powinien mieć postać nazwa.cc. Upewnij się, że wiesz w jakim katalogu został zapisany plik. Co prawda nie jest konieczne zapisywanie plików akurat z rozszerzeniem cpp, bowiem kompilator g++ pozwala na kompilację plików również z innymi rozszerzeniami, jednak polecam Ci stosowanie właśnie tego rozszerzenia do zapisywania swoich plików, bowiem właśnie to rozszerzenie przyjęło się dla programów pisanych w języku C++.

Jeśli udało Ci się zapisać plik, to czas przejść do katalogu, w którym ten plik został zapisany za pomocą terminala. Przypominam, dwie użyteczne komendy:

**ls** - wyświetla zawartość bieżącego katalogu
**cd xxx** - przechodzi do katalogu o nazwie xxx

Przy okazji przypominam, że w linuxie są rozróżniane małe i wielkie litery.

Zakładając, że plik, który udało Ci się zapisać znajduje się w katalogu **/home/tomek/Documents/przyklady**, wówczas przykładowa sesja z terminalem użytkownika o nazwie **tomek** mogłaby wyglądać tak:

```
cd
ls
cd Documents
cd przyklady
ls
```

Po ostatnim poleceniu **ls**, użytkownik zobaczyłby zapisany przez siebie plik. Postąp według tego schematu i mam nadzieję, że wykonując komendę **ls** widzisz już zapisany przez Ciebie plik.

eraz czas skompilować program. Wpisz w terminalu:

**g++ nazwa.cc -Wall -o nazwa**

Parametr nazwa.cc to nazwa pliku który chcesz skompilować, parametr -Wall włącza wszystkie ostrzeżenia (najlepiej zawsze w ten sposób kompilować program), natomiast parametr -o mówi że, to co się znajduje za nim (czyli w naszym przypadku nazwa), to nazwa pliku, pod którym zostanie zapisany program.

Zwróć uwagę, że po myślnikach nie ma żadnych spacji. Jeśli wystąpiły jakieś błędy podczas kompilacji, sygnalizowane napisem zaczynającym się od słowa **Error** podejrzewam, że niestety nie udało dokładnie Ci się przepisać programu - popraw to za pomocą edytora i skompiluj jeszcze raz.

Gdyby w tym miejscu pojawiły się komunikaty poprzedzone słowem **Warning**, to nie musisz już nic kompilować. Program został skompilowany, jednak zalecane by było w takim przypadku dokonanie pewnych modyfikacji w kodzie, aby nie pojawiały się tu żadne komunikaty.

Jeśli po skompilowaniu nie są wypisane żadne komunikaty, to znaczy, że udało Ci się skompilować program. Możesz go teraz uruchomić, wpisując:

./nazwa

Tutaj również zauważ, że pomiędzy ./ a nazwa nie ma żadnych spacji. Teraz na ekranie powinien pojawić się efekt działania programu (w zależności od tego jaki to był program).

W tym momencie muszę zwrócić Twoją uwagę na następującą sprawę: program wystarczy skompilować jeden raz. Skompilowany program można uruchamiać tyle razy ile się chce. Ponowna kompilacja jest niezbędna tylko wtedy, gdy dokonujemy zmian lub poprawek w kodzie źródłowym.

Zapamiętaj podany tutaj schemat, bowiem jeśli zdecydujesz się używać kompilatora g++, to właśnie w ten sposób będziesz kompilować swoje programy. Dla formalności podam, że przy kompilacji można pominąć 2 ostatnie podane w przykładzie parametry (czyli -o nazwa). Wtedy możemy skompilować program źródłowy wpisując:

g++ nazwa.cc  -Wall

Plik wynikowy zostanie zapisany pod nazwą **a.out** i żeby go uruchomić należy wpisać na tej samej zasadzie co poprzednio:

./a.out

Oczywiście nie zalecam kompilowania tą "krótszą metodą", bowiem jeśli w katalogu znajduje się kilka plików źródłowych, wówczas kompilując dowolny z nich będziemy nadpisywać plik a.out. Poza tym nie jest zbyt mądrze nazywać pliki a.out.

Jeśli udało Ci się tutaj dojść, to znaczy, że znasz już podstawy kompilowania programów za pomocą g++. Oczywiście jest to zagadnienie bardziej złożone, bowiem kompilator g++ posiada znacznie więcej opcji.

Aby zapoznać się z pozostałymi opcjami i możliwościami kompilatora, przeczytaj pomoc dołączoną do kompilatora lub poszukaj takich informacji na stronie http://gcc.gnu.org/.

# Dev-C++ - krótka instrukcja obsługi

Kompilacja za pomocą windowsowego Dev-C++ przebiega znacznie łatwiej niż za pomocą linuxowego g++. Po uruchomieniu kompilatora tworzymy nowy plik za pomocą wyboru z menu rozwijanego **Plik**, najpierw **Nowy**, a następnie **Plik źródłowy**.

Teraz wpisujesz treść pliku - również w tym wypadku jako jego treść możesz przepisać któryś z pierwszych przykładów mojego kursu. Teraz pozostaje zapisać plik, wybierając z menu rozwijanego **Plik** opcję **Zapisz jako**. Teraz wpisujesz swoją nazwę pliku.

Zauważ, że pod miejscem na wpisywanie nazwy, jest wybrana opcja **CPP source(*.cpp;*.cc;*.cxx;*.c++;*.cp)**. Ponieważ chcemy skompilować właśnie plik źródłowy, więc ta opcja jest prawidłowa. Jeśli do nazwy pliku nie dodamy jawnie rozszerzenia, to kompilator zapisze nasz plik z rozszerzeniem cpp. W zasadzie możemy nie podawać nazwy rozszerzenia pliku, bowiem w windowsie właśnie to rozszerzenie jest najczęściej uznawane za rozszerzenie pliku źródłowego w C++.

Po zapisaniu pliku, pozostaje spróbować go skompilować. Kompilację możemy przeprowadzić na różne sposoby: możemy wybrać z górnego menu rozwijanego **Uruchom** opcję **Kompiluj**, możemy też kliknąć odpowiednią ikonę na pasku narzędziowym lub możemy skorzystać ze skrótu klawiaturowego **CTRL + F9**.

Jeśli kompilacja się powiedzie, pojawi się okienko, na którym pisze m.in. **Status: done.** Klikamy zamknij - nasz program jest skompilowany. Jeśli natomiast są jakieś błędy (lub ostrzeżenia), wówczas wspomniane okienko tylko mignie przed naszymi oczami. Wysunie się natomiast dolne menu, w którym przeczytamy o błędach (ostrzeżenia są poprzedzone ciągiem **[Warning]**). W przypadku błędów poprawiamy kod programu i ponownie kompilujemy. W przypadku ostrzeżeń nie musimy kompilować - program został skompilowany.

Aby uruchomić program, możemy wybrać z górnego paska **Uruchom** pozycję **Uruchom**. Możemy też skorzystać z ikony lub ze skrótu **CTRL + F10**. Innym sposobem uruchomienia programu jest znalezienie go na dysku (u mnie pliki są zapisywane w katalogu **Moje dokumenty**) i podwójne kliknięcie w odpowiednią ikonę.

Ostatni sposób jest najmniej wygodny, za to najbardziej zalecany. Na pasku narzędziowym systemu windows klikamy **START**, a następnie **Uruchom**. Wpisujemy **cmd** lub **command** (dla starszych wersji windows). Następnie korzystając z dwóch poleceń **cd** i **dir**, przechodzimy do katalogu, w którym znajduje się skompilowana wersja programu. Program uruchamiamy wpisując jego nazwę (która jest taka sama jak nazwa programu źródłowego, tylko ma rozszerzenie **exe**). Ta metoda jest najbardziej zalecana, bowiem uzyskujemy dokładnie identyczny kod źródłowy dla systemu windows i linux. Jeśli uruchamiamy program pozostałymi metodami w windows, zazwyczaj tylko mignie nam przed oczami czarne okienko z programem - żeby temu zaradzić, musimy zawsze przed kończącym nawiasem klamrowym w kodzie źródłowym dodać raz (lub większą liczbę razy jeśli to nie pomaga) komendę **getchar();** i oczywiście ponownie skompilować program.

Po przeczytaniu tego artykułu, umiesz już posługiwać się kompilatorem. Zachęcam Cię zatem do zapoznania się z kursem C++.

# Lekcja 1: O programowaniu. Pierwszy program w C++

## Wstęp do programowania

Jeśli zamierzasz nauczyć się języka C++, musisz zdawać sobie sprawę z tego, że nikt inny nie nauczy Ciebie ani tego języka programowania ani jakiegokolwiek innego, jeśli Tobie na tym nie będzie naprawdę zależało.

W tym kursie będę starał się zwracać uwagę na najważniejsze moim zdaniem kwestie. Postaram się uwypuklić czyhające w pewnych miejscach pułapki, jednak nie licz, że przedstawię wszystkie możliwości. Programowania nie można nauczyć się na pamięć - albo w pewnym momencie spodoba Ci się to i zaczniesz poświęcać dodatkowy czas, aby kiedyś móc nazwać się programistą, albo w pewnym momencie musisz stwierdzić, że to po prostu nie dla Ciebie.

Programowanie w pewnym sensie jest jak sztuka - ktoś może Ci tłumaczyć jak się programuje w danym języku programowania, może wyjaśnić Ci wszystkie reguły, może Ci tłumaczyć przykładowe programy, jednak w końcu to Ty będziesz pisać programy.

Mogę Cię jednocześnie zapewnić, że nie znajdziesz nigdzie rozwiązań wszystkich swoich problemów programistycznych. W pewnym momencie to Ty będziesz wymyślać różne sposoby rozwiązania problemu i nikt Ci już w tym nie pomoże.

Ten kurs, jak i każdy inny kurs programowania ma Ci przedstawić pewne mechanizmy, sposoby. Wszystkich tych mechanizmów będziesz niewątpliwie używać w swoich przyszłych programach (jednych częściej, a innych rzadziej), jednak nawet jeśli wszystkie te mechanizmy, sposoby i reguły zapamiętasz bardzo dobrze, w żaden sposób nie jest to gwarancją, że uda Ci się napisać jakiś większy, przydatny program.

## O tym kursie

Postaram się w tym kursie przedstawić Ci podstawy programowania w C++. Ponieważ kurs jest cały czas w trakcie tworzenia, więc możliwe, że co jakiś czas niektóre z artykułów będą ulegać małym modyfikacjom.

Jeśli nie posiadasz jeszcze żadnego polecanego przez mnie kompilatora lub nie wiesz czym jest kompilator, przeczytaj artykuł O kompilatorach. Również jeśli nie wiesz jak się posługiwać kompilatorem, przeczytaj artykuł o kompilatorach, w którym dowiesz się czym jest kompilowanie i jak się kompiluje programy.

## Ostrzeżenie

Mimo, że podkreślałem to już w artykule o kompilatorach, muszę to napisać jeszcze raz, żeby nie było żadnych nieporozumień:

Ponieważ wszystkie zamieszczone w tym kursie kody programów będą napisane w języku C++ zgodnym ze standardami, to niestety może się zdarzyć, że część z nich na niektórych kompilatorach nie będzie działać. Jest to dość częste zjawisko i jeśli tylko Ty zaopatrzysz się w któryś z wymienionych kompilatorów, nie powinno Cię zbytnio martwić, że Twój program może się nie chcieć kompilować na innym kompilatorze.

Może się okazać, że niektóre przykłady z zamieszczonych tutaj w ogóle nie będą działać na innych

kompilatorach. Ponieważ chcę się skupić na prawdziwym języku C++, dlatego też nie zamierzam uczyć Cię, co zrobić, żeby przedstawione programy działały w kompilatorach, które są po prostu niezgodne ze standardem języka.

# Uwagi praktyczne

Kurs został podzielony na dość małe lekcje, aby jak najwygodniej było Ci się po nim poruszać. Dzięki temu możesz w wolnym momencie skupić się tylko na jednej lekcji bez obawy, że nie uda Ci się jej skończyć.

Mimo, że zamieszczone tutaj przykłady możesz po prostu skopiować do swojego kompilatora, radzę Ci je wszystkie przepisywać. W ten sposób szybciej nauczysz się składni języka C++ oraz nauczysz się pisać kod w sposób bardziej przejrzysty.

Większość przykładów zamieszczonych na stronie jest bardzo prosta. Dlatego zachęcam Cię po każdej lekcji do wymyślania rozszerzeń programów, zastanowienia się, co by tu jeszcze można było zrobić z posiadaną wiedzą. Starając się wymyślać i pisząc nawet te kilka dodatkowych linijek, możesz się wiele nauczyć.

Mimo, że programy zamieszczone tutaj są bardzo proste, sugeruję Ci utworzyć sobie oddzielny katalog na programy w C++ i gdy będziesz przepisywać przykłady zapisywać każdy pod nową nazwą. Dzięki temu jeśli zapomnisz jak się coś robiło, wystarczy, że zajrzysz do poprzednio napisanych programów.

# Pierwszy program

Poniżej znajduje się pierwszy program w C++ jaki napiszesz:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string imie;
  cout <<"Podaj imie: ";
  cin >>imie;
  cin.ignore();
  cout <<"Witaj "<<imie<<'\n';
  cout <<"Gratulacje. To Twoj pierwszy program!"<<'\n';
  cout <<"Nacisnij ENTER aby zakonczyc"<<'\n';
  getchar();
  return 0;
}
```

Przepisz teraz powyższy kod programu, skompiluj go i uruchom. Jeśli w momencie kompilacji kompilator informuje Cię, że są jakieś błędy, to upewnij się, że wszystkie średniki są na swoim miejscu oraz że używasz właściwego kompilatora, który wcześniej Ci polecałem.

Jeśli juz uruchomisz program, podaj swoje imię i naciśnij ENTER. Program wypisze kilka komunikatów oraz będzie czekał znowu, aż naciśniesz ENTER. Wówczas program zakończy swoje działanie.

W ten oto sposób udało Ci się napisać i uruchomić pierwszy program w C++. Prawda, że to nie takie trudne? Czas przejść do kolejnej lekcji i zrozumieć, co tak naprawdę nasz program zrobił.

# Lekcja 2: Budowa programu w języku C++

## Wprowadzenie

W poprzedniej lekcji udało Ci się napisać oraz przetestować swój pierwszy program w języku C++. Zanim jednak dokładnie wyjaśnię Ci jak ten program działa, postaram Ci się przedstawić ogólną budowę programu w C++. Wszystkie programy, które będziesz pisać, będą miały identyczną lub bardzo zbliżoną budowę.

## Ogólna budowa programu w C++

Mimo, że programy napisane w języku C++ mogą być bardzo różnej długości, wszystkie jednak mają następującą budowę:

- **dyrektywy preprocesora**
  Wszystkie dyrektywy preprocesora poprzedzone są znakiem #. Wszystkie dyrektywy preprocesora są wykonywane przez specjalny program zwany preprocesorem stąd ich nazwa. Na szczęście w większości kompilatorów (o ile nie we wszystkich) nie musimy się martwić tym, że dyrektywy te wykonuje preprocesor. Kompilator automatycznie uruchamia preprocesor gdy wykryje dyrektywę.
  Najczęściej wykorzystywaną dyrektywą preprocesora jest dyrektywa **include** - służy ona do dołączania plików nagłówkowych. Inną popularną dyrektywą jest **define** - służy ona do tworzenia nazw symbolicznych.
  Aby dołączyć jeden ze standardowych plików języka C++ należy napisać:
  #include <iostream>
- **wykorzystywane przestrzenie nazw**
  W nieco bardziej skomplikowanych programach, aby ułatwić sobie życie, grupuje się pewne elementy w przestrzenie nazw. Również wszystkie elementy zdefiniowane standardowo w języku C++ są zdefiniowane w przestrzeni nazw - a konkretnie przestrzeni nazw o nazwie **std**.
  Aby włączyć dowolną przestrzeń nazw wystarczy napisać:
  using namespace nazwa;
  gdzie nazwa jest nazwą wybranej przestrzeni nazw.
- **funkcje zdefiniowane przez użytkownika**
  Ponieważ programy pisane w języku C++ mogą być bardzo duże, dlatego też wymyślono mechanizm, aby grupować pewne fragmenty kodu w funkcje. Dzięki temu znacznie łatwiej poruszać się po takim programie czy też pisać programy w kilka osób. Funkcjami zajmiemy się na nieco dalszym etapie nauki.
- **główna funkcja programu**
  Mimo, że funkcja ta działa jak zwykła funkcja ma ona dwie bardzo charakterystyczne cechy. Po pierwsze funkcja ta musi mieć zawsze nazwę main - tak ktoś sobie wymyślił, że główna funkcja programu tak się nazywa i tak już pozostało do dzisiaj.
  Drugą bardzo charakterystyczną cechą jest to, że od tej funkcji zaczyna się całe wykonanie programu.
  Najprostsza funkcja main (która nic nie robi), wygląda tak:

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

Jeśli chcemy, aby nasz program cokolwiek wykonał, musimy umieścić dodatkowy kod pomiędzy **{** oraz **return 0;**

# Dyrektywa include

Jak już wiesz dyrektywa include jest jedną z najczęściej wykorzystywanych dyrektyw preprocesora. Za jej pomocą dołączamy do programu wybrany plik nagłówkowy.

Zastanawiasz się pewnie, po co włączać do programu jakieś pliki nagłówkowe. Otóż w plikach nagłówkowych zdefiniowane są elementy, z których prawie na pewno będziesz chcieć korzystać. Bez kilku czy kilkunastu pliku nagłówkowych napisanie nawet prostego programu byłoby w zasadzie niemożliwe. Niech przykładem będzie to, że aby nawet coś wypisać na ekran (tak jak to miało miejsce w naszym pierwszym programie), należy dołączyć właściwy plik nagłówkowy.

Niestety w początkowym etapie nauki dużym problemem jest jaki plik należy włączyć, aby można było coś zrobić. Otóż co gdzie się znajduje uda Ci się nauczyć, gdy będziesz często korzystać z pewnych elementów. Przydatne mogą być też pliki pomocy dołączone do Twojego kompilatora.

Dyrektywa include występuje w dwóch postaciach:

#include <nazwa>

oraz

#include "nazwa"

W pierwszym wypadku dołączamy plik nagłówkowy, który znajduje się w miejscu określonym przez kompilator. Z takiego dołączania będziemy najczęściej korzystać. W drugim wypadku dołączamy plik znajdujący się w bieżącym katalogu - ta możliwość jest wykorzystywana na nieco wyższym poziomie nauki.

W obu przypadkach nazwa jest nazwą pliku, który chcemy dołączyć do programu. W tym miejscu następuje jednak małe zamieszanie. Wszystkie pliki nagłówkowe mają standardowo rozszerzenie **h**. Otóż pliki nagłówkowe pochodzące z czystego języka C++ dołączamy podając samą nazwę pliku bez rozszerzenia. Natomiast pliki pochodzące tak naprawdę z poprzednika języka C++, czyli języka C włączamy podając nazwę wraz z rozszerzeniem.
Dla przykładu:

#include <iostream>

ale

#include <stdio.h>

Mimo, że na razie z tego i tak nie będziesz korzystać, to warto dodać, że wykorzystując drugą metodę dołączania plików nagłówkowych, z wykorzystaniem ", musimy dodać rozszerzenie pliku (niekoniecznie h) oraz, że możemy dołączać pliki znajdujące się nie tylko w bieżącym katalogu.

Na przykład:

#include "wazne/moj.hpp"

spowoduje dołączenie pliku o nazwie moj.hpp znajdującego się w katalogu wazne, przy czym katalog wazne znajduje się w bieżącym katalogu.

# Wykorzystywane przestrzenie nazw

Jak już wspomniałem, przestrzenie nazw ułatwiają grupowanie pewnych elementów. Wszystkie standardowe elementy języka są umieszczone w przestrzeni nazw std. Dlatego też warto zadeklarować, że chcemy wykorzystywać tę przestrzeń nazw:

using namespace std;

Mimo, że możemy wykorzystać tą metodę, wcale nie musimy. Jednak deklarując wykorzystywanie jakiejś przestrzeni nazw, ułatwiamy sobie znacznie zapis w dalszej części programu. Dlatego mimo pewnych niekorzystnych skutków takiego działania uważam, że warto zawsze dołączać na początku programu standardową przestrzeń nazw std.

Można też zrezygnować z deklaracji użycia przestrzeni nazw. Wtedy jednak w programie, który przedstawiłem Ci w pierwszej lekcji, należałoby wszędzie zamiast **cout** napisać **std::cout** oraz wszędzie zamiast **cin** napisać **std::cin**. Mam nadzieję, że w ten sposób przekonałem Cię, że jednak warto skorzystać z polecanej przeze mnie pierwszej metody, bowiem znacznie skraca to nam pisanie.

UWAGA: Jeśli zapomnisz włączyć standardową przestrzeń nazw **std**, nie zdziw się, jeśli kompilator będzie wypisywać bardzo dziwne błędy. Czasami zorientowanie się, skąd się wziął błąd może zająć znacznie więcej czasu niż pisanie samego programu. Dlatego radzę Ci dobrze - nie zapominaj o dołączaniu przestrzeni nazw **std**!

# Główna funkcja programu

Mimo, że w programie możemy mieć wiele różnych funkcji, zawsze musimy umieścić główną funkcję o nazwie **main**. Jest to jedyna funkcja, która jest uruchamiana automatycznie. Wszystkie pozostałe funkcje musimy uruchomić sami.

Wiesz już jak wygląda funkcja main i wiesz, że aby program coś wykonywał, to coś musi się znajdować właśnie w funkcji main. Na razie nie będę tłumaczył do czego służą tajemnicze int oraz (). W nawiasach klamrowych umieszczamy właściwą treść funkcji, czyli treść naszego programu.

Warto zwrócić uwagę na tajemniczą linię:

```
  return 0;
```

Otóż ta tajemnicza linia, umieszczona przed nawiasem kończącym treść funkcji, czyli przed **}** zwraca wartość wywołania naszego całego programu. Wartość taka jest wykorzystywana przez system operacyjny do stwierdzenia, czy program zadziałał poprawnie, a jeśli nie, to co spowodowało błąd.

Przyjęło się, że poprawne działanie programu jest sygnalizowane zwróceniem wartości **0**, dlatego też taką wartość będziemy zwracać. Ponieważ w tym kursie będziemy pisać proste programy, dlatego też możesz przyjąć, że w funkcji main należy zawsze zwracać wartość 0. Mimo, że niektóre kompilatory nie sygnalizują błędu, jeśli ta linia zostanie pominięta, jest to linia dosyć ważna i lepiej o niej nie zapominać, zwłaszcza jeśli byśmy chcieli pisać programy, które ktoś w przyszłości będzie wykorzystywać.

# Podsumowanie

Przedstawiłem Ci w tej lekcji prosty schemat budowy programu w C++. Oczywiście schemat to tylko schemat i nie zawsze trzeba wykorzystać wszystkie jego części. Może się też zdarzyć, że w danym programie będzie trzeba wykorzystać coś więcej.

A zatem najważniejsze jest, żeby wszystko robić z głową. To co w jednym programie jest niezbędne, w innym będzie zbędnym dodatkiem, który tylko niekorzystnie wpłynie na szybkość działania programu.

Dla pełnego zobrazowania zagadnienia, poniżej przedstawiam jeszcze 3 przykłady typowych schematów kodu.

# Najprostszy program w C++

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

Powyższy program to najprostszy program. Składa się on tylko z funkcji main, wewnątrz której nic nie robimy, czyli sam program faktycznie też nic nie robi. Ponieważ program nic nie robi, nie musimy dołączać ani żadnego pliku nagłówkowego ani standardowej przestrzeni nazw std.

# Schemat prostego programu

```
#include <iostream>

using namespace std;

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

To jest schemat kodu programu, jaki my zazwyczaj będziemy wykorzystywać. Oczywiście w funkcji main będziemy umieszczać dodatkowe instrukcje.

# Schemat złożonego programu

```
#include <iostream>
#include <string>

using namespace std;

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

Tak naprawdę ten schemat wcale nie jest złożony. Różni się on jedynie od poprzedniego tym, że oprócz pliku nagłówkowego **iostream**, dołączamy dodatkowo plik nagłówkowy **string**. Czasami bowiem będzie zdarzać się, że będziemy musieli dołączyć dwa (tak jak tutaj) lub większą ilość plików nagłówkowych, aby w sposób wygodny i szybki napisać program.

# Lekcja 3: Twój pierwszy program - wyjaśnienie działania

## Wprowadzenie

W tej lekcji dowiesz się w końcu krok po kroku, co robił Twój pierwszy program. Zanim jednak to nastąpi, przedstawię Ci jeszcze inne zagadnieni

## Komentarze w programach

Oprócz przedstawionych w poprzedniej lekcji elementów każdego programu, można wyróżnić jeszcze jeden element - komentarze. W przeciwieństwie jednak do wymienionych wcześniej elementów, komentarze nie mają żadnego znaczenia dla kompilatora.

Jedynym powodem stosowania komentarzy jest wygoda osoby piszącej programy. Co prawda pisząc program, trzeba będzie poświęcić dodatkowy czas na napisanie komentarza, jednak pomyśl jak dużo czasu zyskasz, gdy za kilka dni, tygodni czy miesięcy znów spojrzysz na program i uda Ci się bez problemu zrozumieć zawiłe fragmenty kodu właśnie dzięki komentarzom.

Stosowanie komentarzy ponadto jest niezbędne również, gdy nad programem pracuje kilka osób. Także pisząc programy, których fragmenty będziesz wykorzystywać w przyszłości w innych programach, komentarze są niezbędne.

Komentarze można zapisywać w programach na dwa sposoby:

```
/*To
jest
komentarz
kilkulinijkowy
*/
lub
// To jest komentarz jednolinijkowy.
```

Pierwsza metoda służy do dłuższych komentarzy lub takich komentarzy, które chcemy, aby zajmowały tylko pewną część linii. Cały komentarz mieści się między **/*** oraz ***/**. Znaki rozpoczynające komentarz mogą być rozmieszczone dowolnie - mogą być w tej samej linii co tekst komentarza, mogą być w różnych liniach lub mogą się znajdować w jednej linii.

Druga metoda służy do komentarzy jednolinijkowych. Komentarz rozciąga się od // aż do końca linii. Nie można komentarza zakończyć przed końcem linii. Dlatego jeśli chcemy zakończyć komentarz wcześniej, a za nim umieścić normalny kod programu, musimy wówczas użyć pierwszej metody.

Komentarze mogą być ponadto również bardzo przydatne podczas testowania programu. Jeśli nie wiemy gdzie jest błąd w naszym programie, możemy pewne części programu otoczyć komentarzem - wówczas ta część programu się nie wykona. Otaczając coraz więcej kodu komentarzami, uda nam się w końcu zmniejszyć kod programu tak bardzo, że będziemy mogli stwierdzić, że tam właśnie jest błąd. Po poprawieniu błędu należy wtedy oczywiście usunąć wszystkie znaki komentarzy, tak aby wykonywał się cały kod programu.

# Jak działa Twój pierwszy program

Jeszcze raz przytoczę pierwszy program w C++, jaki Ci przedstawiłem:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string imie;
  cout <<"Podaj imie: ";
  cin >>imie;
  cin.ignore();
  cout <<"Witaj "<<imie<<'\n';
  cout <<"Gratulacje. To Twoj pierwszy program!"<<'\n';
  cout <<"Nacisnij ENTER aby zakonczyc"<<'\n';
  getchar();
  return 0;
}
```

Ponieważ wiesz już co nieco o budowie programu w C++, mam nadzieję, że uda Ci się teraz zrozumieć przynajmniej część programu.

Na początku dołączamy dwa pliki nagłówkowe - już wiesz do czego pliki nagłówkowe służą. Plik **iostream** dołączamy, aby móc skorzystać z wypisywania na ekran (**cout**) oraz pobierania napisów z klawiatury (**cin**). Natomiast plik **string** dołączamy, aby móc skorzystać z typu string, który umożliwia operacje na napisach.

Następnie dołączamy standardową przestrzeń nazw **std**. Mimo, że to już wiesz to powtórzę - dzięki temu skracamy sobie zapis w dalszej części programu.

W kolejnej części programu znajduje się główna funkcja programu **main**, w której pomiędzy { oraz return 0; wstawiliśmy właściwą treść naszego programu.

Wszystkie linie, które zaczynają się od słowa **cout** służą do wypisywania na ekran. Zauważ, że po każdym cout znajduje się operator <<. Zwróć też uwagę, że większość napisów wypisywanych na ekran znajduje się w podwójnych cudzysłowach.

Linia zaczynająca się od **cin** służy do pobrania Twojego imienia z klawiatury. Zwróć uwagę, że po cin znajduje się operator >> a następnie słowo imie ale tym razem bez cudzysłowów.

Linia **string imie;** ma sygnalizować, że w naszym programie będziemy chcieli wykorzystać pewien napis. Później okazuje się, że napis pobieramy z klawiatury, a następnie wypisujemy na ekran.

Linia **getchar();** powoduje, że nasz program czeka aż użytkownik naciśnie ENTER. Tak naprawdę **getchar** to pewna gotowa funkcja, która znajduje się w pliku nagłówkowym **iostream**. Gdybyśmy na początku nie dołączyli tego pliku do naszego programu, nie moglibyśmy wykorzystać tej funkcji.

# Funkcja ignore()

Zauważ, że w kodzie programu znajduje się jeszcze jedna linia, o której w ogóle nie wspomniałem:

```
cin.ignore();
```

Zauważ też, że linię wcześniej znajduje się

```
cin >>imie;
```

Otóż wspomniana przeze mnie linia ze słowem ignore powoduje wyeliminowanie pewnego nieprzyjemnego skutku. Kiedy program poprosił Cię o wpisanie imienia, aby zasygnalizować, że wpisywanie jest już zakończone trzeba było użyć klawisza ENTER. Czy jednak klawisz ENTER wchodzi w skład Twojego imienia? Na pewno nie. Dlatego też klawisz ENTER został niejako zapamiętany, że został wciśnięty, jednak nie został dodany do Twojego imienia.

Co to powoduje? Otóż sprawia to, że w przypadku gdy będziemy chcieli jeszcze raz pobrać jakiś napis z klawiatury mogą pojawić się problemy. Dlatego też użyliśmy funkcji **ignore**. Funkcja ta znajduje się w pliku nagłówkowym **iostream**. Dzięki użyciu tej funkcji sprawiliśmy, że to, że klawisz ENTER został wciśnięty zostanie "zapomniane" i nie będzie żadnych problemów.

Nie pytaj w tym momencie dlaczego jednak zapis linii z funkcją ignore jest taki a nie inny, bowiem na to jest jeszcze dużo za wcześnie. Najlepiej zapamiętaj ten zapis i zapamiętaj, że zawsze kiedy pobieramy coś z klawiatury, warto w następnej linii użyć funkcji **ignore** - dzięki temu unikniesz problemów na przyszłość.

# Jak działa ten program - wyjaśnienie drugie

Jeśli nie do końca udało Ci się zrozumieć jak działa ten program, teraz postaram się jeszcze raz przedstawić, co program robi. Zastrzegam jednak, że będzie to bardzo nieformalne tłumaczenie, więc jeśli jesteś już bardziej zaawansowanym użytkownikiem, lepiej omiń następny akapit.

Ogólnie mówiąc program działa tak: najpierw sygnalizuje, że będzie chciał używać napisu. Następnie wypisuje komunikat na ekran z prośbą o podanie imienia. Teraz pobiera Twoje imię z klawiatury oraz zaraz po tym zapobiega skutkowi naciśnięcia przez Ciebie klawisza ENTER. Teraz wypisuje komunikat na ekran o treści "Witaj " wraz z imieniem podanym przez Ciebie wcześniej z klawiatury. Następnie wypisuje kolejny komunikat, w którym gratuluje Ci napisania pierwszego programu. Teraz wypisuje komunikat informacyjny, że należy nacisnąć ENTER, aby zakończyć program i czeka tak długo gdy naciśniesz ENTER. Na samym końcu informuje system operacyjny, że program wykonał się w sposób prawidłowy.

# Wykorzystanie komentarzy do wyjaśnienia

Poniżej jeszcze jeden przykład jak można wyjaśniać program. Skorzystamy tutaj z komentarzy, bo to one zazwyczaj właśnie służą do wyjaśniania kodu programu.

```
#include <iostream> // dla cin, cout, ignore, getchar
#include <string>  // dla string

using namespace std; // wlaczenie standardowej przestrzeni nazw

/*
Ponizej znajduje sie funkcja main
*/
int main ()
{
  string imie; // bedziemy wykorzystywac napis
  cout <<"Podaj imie: "; // wypisanie na ekran
  cin >>imie; // pobranie imienia z klawiatury
  cin.ignore(); // zignorowanie ENTERa nacisnietego przed momentem
  cout <<"Witaj "<<imie<<'\n'; // wypisanie komunikatu
  cout <<"Gratulacje. To Twoj pierwszy program!"<<'\n'; // wypisanie komunikatu
  cout <<"Nacisnij ENTER aby zakonczyc"<<'\n'; // wypisanie komunikatu
  getchar(); //czekanie na nacisniecie klawisza ENTER
  return 0; // informacja, ze program zadzialal prawidlowo
} // koniec funkcji main
```

# Czym jest ten "Hello world"

Na koniec warto dowiedzieć się czym jest "Hello world", o którym być może kiedyś usłyszysz. "Hello world" jest programem, który jest w zasadzie pierwszym programem, jaki użytkownik pisze poznając nowy język programowania. Można by rzec, że jest to taka swoista programistyczna tradycja.

Jak jednak widzisz, pierwszy program jaki Ci przedstawiłem był zupełnie inny i znacznie trudniejszy od programu "Hello world". Ale, żeby uspokoić swoje sumienie i Twoją ciekawość, oto przedstawiam Ci program "Hello world" w całej okazałości. Mam nadzieję, że komentarz, co robi ten program jest już w tym momencie zbyteczny.

```
#include <iostream>

using namespace std;

int main ()
{
  cout <<"Hello world"<<'\n';
  return 0;
}
```

Jak widzisz program jest bardzo prosty. Jeśli po skompilowaniu i uruchomieniu program tylko miga Ci przed oczami, upewnij się, że znasz różne sposoby uruchamiania programów (pisałem o tym w artykule dotyczącym kompilatorów).

# Wprowadzenie

W tej lekcji przedstawię Ci kilka zagadnień związanych głównie z pisaniem programu tak, jak to piszą zawodowi programiści. Ten sam program można napisać na różne sposoby, a już samo spojrzenie na kod może świadczyć dobrze lub źle o osobie piszącej dany program.

# Średniki w C++

Przyjrzyj się teraz kodom, które przedstawiłem Ci już w tym kursie. Zauważ, że po większości linijek znajduje się średnik. Zwróć jednak uwagę, że średnik nie znajduje się na końcu każdej linii.

Spróbuj teraz w którymś z przykładowych programów skasować jeden ze średników. Spróbuj skompilować program. Zapewne ujrzysz błąd i program się nie skompiluje. Przywróć początkowy stan programu i teraz dla odmiany spróbuj dodać średnik w którejś z linii, w której średnika nie było. O ile tylko średnik nie został dostawiony w linii pustej lub linii zawierającej któryś z nawiasów klamrowych, programu również nie uda się skompilować.

Jak zatem możesz się już teraz domyślać, średniki znajdujące się na końcach linii nie znajdują się tam przypadkowo. Ich wstawienie lub niewstawienie jest za każdym razem bardzo ważne, bowiem mały błąd może spowodować albo to, że program się nie skompiluje, albo co gorsza, że program się skompiluje, ale będzie działał nie tak, jak autor programu sobie życzył.

Tak naprawdę, nie do końca jest prawdą, że średniki musimy stawiać na końcu niektórych linii. Średniki musimy stawiać na końcu każdej **instrukcji**.

# Instrukcja w języku C++

**Instrukcję** w języku C++ można utożsamiać ze zdaniem. Instrukcja jakby mówi, co ma się zdarzyć np. zadeklaruj zmienną, pobierz zmienną x, wypisz komunikat na ekran, zwróć wartość 0.

Oprócz instrukcji, w języku C++ występują również **wyrażenia**. Wyrażenia są jakby częściami instrukcji. Każda instrukcja może składać się z kilku lub kilkunastu wyrażeń. Na przykład, jeśli chcemy wypisać sumę liczb 2 i 3, wówczas suma liczb 2 i 3 stanowi wyrażenie, natomiast cała operacja wypisania stanowi instrukcję.

Zasadniczą różnicą między instrukcjami i wyrażeniami jest to, że instrukcja musi zostać zakończona średnikiem. Natomiast wyrażeń nie kończymy średnikami - jeśli tak zrobimy, wówczas kompilator zasygnalizuje błąd.

# Poprawność kodu, a wygląd kodu

Wszystkie kody źródłowe, które do tej pory Ci przedstawiłem były kodami poprawnymi z punktu widzenia składni języka. Każdy z kodów poprawnie się kompilował i przy kompilacji nie pojawiał się żaden błąd.

Jeśli kody przedstawionych programów były przez Ciebie pisane własnoręcznie, porównaj wygląd kodu mojego i Twojego. Czy w Twoim kodzie źródłowym również niektóre linie są puste oraz są stosowane wcięcia w niektórych liniach?

Język C++ jest językiem, w którym tak naprawdę możemy pisać jak nam się to podoba. Możemy wstawiać puste linie, możemy kilka linii zapisać w jednej linii, a niektóre linie zapisać w kilku liniach. Możemy też stosować komentarze w zasadzie w dowolnym miejscu.

Aby uświadomić Ci wagę zagadnienia, spójrz na poniższe kody źródłowe programów i oceń, który z tych kodów najbardziej Ci się "podoba".

```cpp
#include<iostream>
using namespace std;
int main(){
cout <<"Hello world"<<'\n';
getchar();
return 0;
}
```

```cpp
#include<iostream>
using namespace std;
int main()
{
cout <<"Hello world"<<'\n';
getchar(); return 0;
}
```

```cpp
#include<iostream>
using namespace std;
int main(){
cout <<"Hello world"<<'\n';
getchar();
return 0;}
```

```cpp
#include <iostream>

using namespace std;

int main()
{
  cout <<"Hello world"<<'\n';
  getchar();
  return 0;
}
```

# Lekcja 4: Jak pisać poprawny i przejrzysty kod

A teraz mała niespodzianka - wszystkie przedstawione przed momentem kody źródłowe robią dokładnie to samo! Uruchom kompilator i skompiluj każdy z przedstawionych programów. Zobaczysz, że każdy z nich bez problemu się skompiluje i na dodatek działa dokładnie tak samo.

## Jak kod powinien wyglądać

Tak naprawdę, wygląd kodu dla działania programu nie ma żadnego znaczenia. Ważne jest jedynie, aby kod programu był poprawny. Wszystkie białe znaki (czyli spacje, dodatkowe linie, wcięcia za pomocą tabulatorów) nieznajdujące się pośród " i ', są ignorowane.

Wygląd kodu ma znaczenie tylko dla człowieka. W kodzie napisanym według pewnych niepisanych zasad, zdecydowanie łatwiej znaleźć interesujące nas fragmenty i zdecydowanie łatwiej zrozumieć, co taki kod robi.

Mam nadzieję, że jest oczywiste, że najbardziej czytelnym kodem z pośród grupy wcześniej przedstawionych kodów, jest kod napisany jako ostatni z przedstawionej grupy. W kodzie tym zostało zastosowanych kilka reguł. Przede wszystkim każda instrukcja znajduje się w nowej linii. Oprócz tego, dwie linie zostały pozostawione puste - aby oddzielić pewne fragmenty kodu od siebie. Na dodatek wszystko co znajduje się w funkcji main zostało "wcięte".

Zastosowanie tych trzech prostych zabiegów - pustych linii, wcięć oraz pisania każdej instrukcji w oddzielnej linii, niewątpliwie zwiększyło czytelność kodu.

Musisz jednak wiedzieć, że nie we wszystkich językach można robić "co się podoba". W niektórych starszych językach były na sztywno ustalone pewne zasady i ich nieprzestrzeganie powodowało, że kod się nie kompilował. Skoro język C++ pozwala na dowolne zabiegi poprawiające czytelność, to warto z nich skorzystać.

Przy okazji pragnę zwrócić Twoją uwagę na tzw. wcięcia. Jeśli o mnie chodzi, stosuję wcięcia składające się z 3 znaków spacji. Moim zdaniem jest to najrozsądniejsza wielkość wcięć - są one zarówno widoczne, jak i przy bardziej skomplikowanych programach nie utrudniają pisania kodu. Ty oczywiście zastosujesz wcięcia jakie będziesz chcieć - to była tylko moja sugestia.

Na koniec, pragnę zwrócić Twoją uwagę na to, co wspomniałem już wcześniej - po tym, jak wygląda kod programu, można zazwyczaj dość łatwo ocenić, czy ktoś już napisał kilkanaście - kilkadziesiąt kodów źródłowych, czy dopiero zaczyna naukę języka. Dlatego jeśli nie chcesz wszystkim ogłaszać, że jesteś początkującym programistą, postaraj się, aby Twój kod programu wyglądał jak należy.

## Grupowanie instrukcji

Do grupowania instrukcji służą nawiasy klamrowe. Już w przykładowych programach udało Ci się zetknąć z tymi nawiasami - znajdowały się one wewnątrz funkcji **main** i służyły do zgrupowania instrukcji w funkcji main.

Przyjęte są dwie konwencje dotyczące zapisu nawiasów klamrowych. Obie konwencje przedstawiają poniższe przykłady:

```
#include <iostream>

using namespace std;

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

```
#include <iostream>

using namespace std;

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

## Podsumowanie

W tym artykule przedstawiłem Ci sposoby, jak sprawić, aby kod Twoich programów był jak najbardziej czytelny. Co prawda wszystkie te reguły nie są krytyczne - nie musisz ich koniecznie stosować, jednak ich stosowanie sprawi, że Twój kod, również dla innych osób będzie przejrzysty i profesjonalny.

Jeśli jednak chcesz się w przyszłości zająć programowaniem na poważnie, od razu zacznij pisać zgodnie z niepisanymi zasadami. Znacznie trudniej jest bowiem zerwać ze złymi przyzwyczajeniami niż od razu nauczyć się pisać poprawnie.

# Lekcja 5: Zmienne i podstawowe typy danych

## Czym jest zmienna

Aby móc pisać jakiekolwiek przydatne programy, musimy operować na danych. źródła danych mogą być różne - mogą pochodzić z klawiatury, z pliku lub być przesyłane z zewnętrznego urządzenia.

Z pobieranymi danymi prawie na pewno będziemy chcieli coś zrobić - będziemy chcieli je przesłać gdzieś dalej lub w jakiś sposób przetworzyć. Co jednak, jeśli chcemy pobrać kilka danych np. dwie liczby za pomocą klawiatury a później wyświetlić ich sumę? Musimy przecież jakoś odróżnić pierwszą i drugą pobraną liczbę, żeby później móc je dodać. Do takich celów służą właśnie zmienne.

**Zmienne** są czymś w rodzaju pudełek, w których można przechowywać dane. W zmiennych będą przechowywane bardzo różne dane. Może to być zarówno pojedyncza liczba, pojedyncza litera, jak i kilka liczb, wyraz lub nawet całe zdanie.

To co będziemy chcieli przechowywać w zmiennej zależy tylko i wyłącznie od nas. Raz będzie to liczba, raz litera lub wyraz. Komputer oczywiście nie wie, jakie są nasze zamiary. Musimy mu to w jakiś sposób powiedzieć. Do tego celu służy **typ zmiennej**.

# Typ zmiennej - czy to konieczne

Jak już wspomniałem, to naszym zadaniem jest określenie typu zmiennej. Komputer sam się nie domyśli, czy chcemy, aby to co zostało pobrane np. z klawiatury było liczbą czy zwykłym napisem. Możesz pomyśleć, że to bez sensu, a jednak jest w tym głęboki sens.

Przede wszystkim, każda zmienna, której będziemy chcieli użyć w programie musi zostać umieszczona gdzieś w pamięci komputera. Jeśli podamy z klawiatury napis "ala", to to słowo zostanie gdzieś zapisane w pamięci komputera. Tak samo będzie, gdy podamy liczbę 4 - ona również musi zostać gdzieś zapisana w pamięci komputera.

Chyba nie muszę Cię przekonywać, że dane, które chcemy przekazać do komputera mogą być różnej długości. Może to być jedna litera, jak i 100 stron tekstu. Musimy jasno powiedzieć, jakiego typu ma być zmienna, aby można było zarezerwować w pamięci odpowiednią ilość miejsca.

Można by oczywiście rezerwować jak najwięcej pamięci, jednak nie jest zbyt rozsądnie zarezerwować na jedną literę całej pamięci operacyjnej czy nawet jej połowy. Dzięki temu, że podajemy typ zmiennej, w pamięci operacyjnej zostanie zarezerwowany dla zmiennej tylko fragment pamięci - dokładnie taki, jak wynika to z typu zmiennej.

Na dodatek o tym, że typy danych są konieczne, niech świadczy taka sytuacja. Chcesz dodać do siebie to, co podasz z klawiatury. Podajesz najpierw 2, a później 3. Jaki powinien być wynik? Pomyślisz sobie, że 5. No dobrze, a co jeśli najpierw podasz literę c, a za chwilę literę b. Pomyślisz sobie, że komputer powinien zauważyć, że to nie są liczby i rezultatem powinno być np. cb.

No dobrze. A co jeśli najpierw podasz literę c, a później 2? Pomyślisz sobie tak - jeden z podanych znaków jest literą, więc wynik powinien być c2 - to przecież oczywiste.

W takim razie czas na ostatnią zagadkę. Tym razem chcemy dodać do siebie aż trzy dane podane za pomocą klawiatury. Najpierw podajesz c, później 2 a na końcu 3. Jaki tym razem powinien być rezultat: c23 a może c5? Podejrzewam, że nie wiesz już jak byłoby rozsądniej.

Dlatego też stosowanie typów danych jest konieczne - dzięki temu będzie oczywiste jaki jest wynik np. dodawania danych do siebie. To, jaki rezultat otrzymamy, zależy tylko i wyłącznie od nas. Taki właśnie model przyjęto w C++.

Musisz jednak wiedzieć, że mimo że w języku C++, trzeba jasno i wyraźnie określić typ zmiennej, to istnieją języki, w których typów podawać nie trzeba. Zazwyczaj jednak języki takie są nieco mniej wydajne, bowiem na bieżąco musi być sprawdzany rodzaj operacji na zmiennej, to co zmienna zawiera w danym momencie i wtedy podejmowana jest dopiero odpowiednia akcja.

# Jak określić typ zmiennej

Wiesz już, że w języku C++ musisz jawnie określić typ zmiennej. Określenie typu zmiennej jest bardzo proste. Symbolicznie, robi się to następująco:

nazwa_typu nazwa_zmiennej;

Określenie **nazwa_typu** to typ, jaki będziemy chcieli, aby miała nasza zmienna. Tutaj nie możemy wpisać dowolnego słowa (chyba, że stworzymy własny typ danych). Typy w języku C++ są określone i musimy użyć jednego ze specjalnych słów (o tym za chwilę).

Z kolei **nazwa_zmiennej** to nazwa, jaką wymyślimy dla zmiennej. Nazwa zmiennej to tzw. **identyfikator**.

# Identyfikator a nazwa zmiennej

**Identyfikator** to dowolnie długi napis składający się z małych i dużych liter, cyfr oraz znaku _ (znaku podkreślenia). Identyfikator może się zaczynać od litery lub znaku podkreślenia (nie może się zaczynać od cyfry). Przy okazji dodam, że mówiąc o literach mam tu na myśli wyłącznie podstawowe litery, bez polskich ogonków.

Jak już wspomniałem, nazwa zmiennej jest właśnie identyfikatorem. Przy nadawaniu nazw, podstawową zasadą jest to, że nazwy zmiennych muszą się między sobą różnić. Nie możemy dwóm zmiennym nadać tej samej nazwy, bowiem wtedy ich odróżnienie nie będzie możliwe.

Mimo, że tak naprawdę jedyną zasadą dotyczącą nazw zmiennych jest to, że nazwa ta musi być identyfikatorem oraz, że nazwy zmiennych muszą być unikalne, nie ma innych zasad. Mimo to warto nałożyć kilka dodatkowych ograniczeń, które ułatwią pisanie programów. Zaznaczam jednak, że z ograniczeń tych można korzystać lub nie - nie są one wymagane.

Pierwszą główną zasadą jest to, że nazwa powinna mieć nazwę, która odpowiada jej przeznaczeniu. Jeśli nazwa będzie służyła do przechowywania imienia, warto nazwać ją po prostu **imie**. Reguły tej nie warto stosować dla zmiennych, które służą np. w pętlach jako zmienne sterujące (dowiesz się wtedy przyszłości). Wówczas zwyczajowo nazywa się zmienne: i, j, k itd.

Mimo, że jak już wspomniałem, identyfikatory mogą być dowolnie długie, to jednak zazwyczaj w większości kompilatorów jest jakieś ograniczenie. Ograniczenie to sięga zazwyczaj nawet ponad 100 znaków. Mimo, że nazwy zmiennych powinny się kojarzyć z przeznaczeniem zmiennej, to nie powinny być za długie. Najlepsza długość nazwy zmiennej to **od 6 do 10 znaków**. Stosując długie nazwy, np. po kilkadziesiąt znaków oprócz tego, że musisz się więcej napisać, to jeszcze sprawiasz, że kod staje się mniej czytelny.

Kolejną sprawą jest to, że mimo że duże i małe litery są rozróżniane, raczej nie należy tworzyć zmiennych, które różnią się wielkością liter. Należy unikać jednoczesnego stosowania takich identyfikatorów, bowiem łatwo się pomylić. Poza tym, stosując taką metodę, utrudniamy innym osobom zrozumienie kodu. Niewłaściwe z tego punktu widzenia będą zatem jednocześnie występujące nazwy typu: imie, Imie, IMIE, imiE.

Ostatnią sprawą jest wielkość liter i oddzielanie wyrazów. Wszystkie nazwy zmiennych przyjęło się pisać małymi literami. Co prawda występują od tego pewne odstępstwa, jednak ogólnie nazwy zmiennych najlepiej pisać małymi literami.

W przypadku, gdy chcemy tworzyć nazwy zmiennych składające się z dwóch słów, powszechnie stosowane są dwie notacje. Pierwsza, tzw. **stara notacja** polega na stosowaniu w nazwie zmiennej wyłącznie małych liter i oddzielaniu poszczególnych słów za pomocą podkreślnika, np.

**mala_liczba_tomka**.

Z kolei w tzw. **nowej notacji** znak podkreślenia nie jest stosowany. Oddzielenie słów otrzymuje się poprzez napisanie pierwszej litery nowego słowa za pomocą dużej litery, przy czym pierwsze słowo zawsze zaczyna się z małej litery. Wówczas przykładowa nazwa przedstawiona w starej notacji, w nowej notacji wyglądałaby tak: **malaLiczbaTomka**.

Jeśli chodzi o mnie prywatnie, to do tej pory stosuję tzw. starą notację. Moim zdaniem jest ona po prostu bardziej przejrzysta.

Przedstawione powyżej reguły możesz stosować lub nie. Powszechnie wszystkie te reguły są stosowane, więc mam nadzieję, że szybko uda Ci się ich nauczyć, jeśli tylko będziesz tego chcieć.

# Podstawowe typy liczbowe

Wiesz już jakie, jak można nazywać zmienne w programach. Teraz czas dowiedzieć się, jakie są typy danych. Poniżej znajdują się podstawowe typy liczbowe występujące w języku C++:

**int** - typ całkowity
**float** - typ zmiennopozycyjny (ułamkowy)
**double** - typ zmiennopozycyjny podwójnej precyzji (dokładniejszy niż typ float)

Przedstawione typy liczbowe różnią się przede wszystkim zakresem. Zanim zdecydujemy się deklarować jakąś zmienną, musimy najpierw zdecydować, jak duże liczby będziemy chcieli czytać i czy będą to liczby całkowite czy liczby z częścią ułamkową (zmniennopozycyjne).

Drugą również ważną cechą typów danych jest to, iż zajmują one różną ilość w pamięci operacyjnej. Dlatego, jeśli chcemy czytać tylko małe liczby całkowite, raczej nie powinniśmy deklarować zmiennych typu float czy double, bo w przypadku gdy będziemy czytać setki czy tysiące liczb, może się okazać, że właśnie przez to zabraknie nam pamięci.

Każdy typ danych może zajmować różną ilość pamięci. Aby sprawdzić, ile bajtów zajmuje np. typ int w Twoim komputerze należy napisać **sizeof(int)**.

Ogólnie, aby sprawdzić, ile pamięci zajmuje dowolny typ danych (nie tylko liczbowy) w Twoim komputerze należy wpisać **sizeof(typ)** lub **sizeof(zmienna)**. Ponieważ w języku C++ zmienna jest zawsze określonego typu, który programista sam najpierw podał, więc w tym drugim przypadku rezultat będzie identyczny jak w pierwszym wypadku.

# Przykładowe programy

Poniżej przedstawiam dwa przykładowe programy, w których deklarujemy zmienne. Ponieważ w programach nie ma żadnych nowych elementów w stosunku do poprzednich programów, dlatego też powstrzymam się od dodatkowych komentarzy. Wszystkie niezbędne komentarze są umieszczone w kodzie programów. **using namespace** std;

```
int main()
{
  float mojaLiczba; // liczba zmiennopozycyjna, nazwa zmiennej wg "nowej
notacji"

  cout <<"Podaj wartosc liczby: ";
  cin >>mojaLiczba;
  cin.ignore();

  cout <<'\n'<<"Wartosc liczby podanej przez Ciebie to "<<mojaLiczba<<'\n';
  cout <<"Typ float zajmuje "<<sizeof(float)<<" bajt(y/ow)."<<'\n';
  cout <<"Twoja zmienna zajmuje "<<sizeof(mojaLiczba)<<" bajt(y/ow).\n\n";

  cout <<"Nacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
 double moj_wzrost; // liczba zmiennopozycyjna, "stara notacja"
 int wiek; //liczba calkowita

 cout <<"Podaj swoj wzrost (w metrach): ";
 cin >>moj_wzrost;
 cin.ignore();

 cout <<"Podaj swoj wiek: ";
 cin >>wiek;
 cin.ignore();

 cout <<'\n'<<"Masz "<<moj_wzrost<<" metrow wzrostu.\n";
 cout <<"Typ double zajmuje "<<sizeof(double)<<" bajt(y/ow)."<<'\n';
 cout <<"Zmienna zajmuje "<<sizeof(moj_wzrost)<<" bajt(y/ow).\n\n";

 cout <<'\n'<<"Masz "<<wiek<<" lat.\n";
 cout <<"Typ int zajmuje "<<sizeof(int)<<" bajt(y/ow)."<<'\n';
 cout <<"Zmienna zajmuje "<<sizeof(wiek)<<" bajt(y/ow).\n\n";

 cout <<"Nacisnij ENTER aby zakonczyc\n";
 getchar();
 return 0;
}
```

# Podsumowanie

Dzięki temu artykułowi, umiesz już deklarować zmienne w programie. Wiesz jak prawidłowo nazywać zmienne i czym są identyfikatory. Ponadto znasz już podstawowe typy liczbowe i wiesz jak można sprawdzić rozmiar dowolnego typu danych.

# Lekcja 6: Zmienne - Typ znakowy i typ napisowy

## Wprowadzenie

Przedstawiłem Ci już podstawowe typy liczbowe. Jak już się domyślasz i podejrzewasz, operując tylko na liczbach nie byłoby nam zbyt wygodnie komunikować się z użytkownikiem. Tak naprawdę te dwa typy które Ci teraz przedstawię, służą głównie do komunikacji z użytkownikiem, ale nie tylko.

## Typ znakowy

Typ znakowy umożliwia przechowywanie jednego znaku. Do czego taki typ może nam się przydać? Przede wszystkim umożliwia on potwierdzanie pewnych akcji. Nie raz pewnie udało Ci się natknąć na program, który żądał potwierdzenia jakiejś akcji. Tutaj właśnie poprzez podanie odpowiedniej litery, cyfry lub innego znaku, można właśnie zrealizować.

Typ znakowy w języku C++ to:

**char** - typ znakowy (zajmuje 1 bajt)

Jeśli dokładnie czytasz, powinno zwrócić Twoją uwagę stwierdzenie, że możemy podać jak znak, zarówno literę, cyfrą jak i inny znak. Może Cię to zastanawiać, ale przecież czy cyfry nie można traktować jako znaku? Oprócz tego możemy wykorzystywać w programach również takie, znaki których nie można wpisać za pomocą klawiatury.

Poniżej przedstawiam przykładowy program, w którym pobieramy znak z klawiatury, wypisujemy później pobrany znak. Dodatkowy wypiszemy rozmiar typu **char**. Podany znak musi być znakiem niesterującym, czyli nie może to być znak typu Ctrl, Shift, Alt, F1 itd. Możesz wpisać tylko jeden znak - inaczej program nie zadziała "poprawnie".

```
#include <iostream>

using namespace std;

int main()
{
 char znak; // typ znakowy

 cout <<"Podaj dowolny NIESTERUJACY znak z klawiatury: ";
 cin >>znak;
 cin.ignore();

 cout <<'\n'<<"Znak podany przez Ciebie to: "<<znak<<".\n";
 cout <<"Typ char zajmuje "<<sizeof(char)<<" bajt."<<'\n';
 cout <<"Zmienna zajmuje "<<sizeof(znak)<<" bajt.\n\n";

 cout <<"Nacisnij ENTER aby zakonczyc\n";
 getchar();
 return 0;
}
```

Znaki zapisujemy w pojedynczych cudzysłowach '. Np. poniższa instrukcja spowoduje wypisanie litery a, spacji, a następnie litery b.

```
cout <<'a'<<' '<<'b';
```

Bardzo ważne jest, aby w takim wypadku pomiędzy cudzysłowami znajdował się tylko jeden znak. W innym wypadku kompilator zazwyczaj wypisze błąd. Może się jednak zdarzyć, że kompilator nie zasygnalizuje błędu, program się skompiluje a wskutek błędnego zapisu, program nie będzie działał poprawnie.

Oprócz standardowego zapisu możemy również wypisywać znaki za pomocą sekwencji '**\ooo**', przy czym ciąg **ooo** to liczba w zapisie **ósemkowym**. Poniższa linijka przedstawia sposób wypisania litery S tą metodą. Jak widzisz nie jest to zbyt wygodna i tak naprawdę zbyt przydatna metoda.

```
cout <<'\123';
```

Znacznie bardziej przydatna jest możliwość wypisywania znaków specjalnych. Wszystkie znaki specjalne są poprzedzone znakiem \. Jeśli przyjrzysz się poniższej liście i przypomnisz sobie wcześniejsze programy, zauważysz, że stosowałem w nich dość często pewien znak specjalny, a mianowicie '**\n**', czyli znak nowej linii.

Oto pełna lista znaków specjalnych w języku C++:

```
'\n'  - nowa linia
'\t'   - tabulacja pozioma (czyli "normalny" tabulator)
'\v'  - tabulacja pionowa
'\b'  - cofnięcie (skasowanie ostatniego znaku)
'\r'   - powrót karetki (przesunięcie do początku wiersza)
'\f'   - nowa strona
'\a'  - sygnał dźwiękowy
'\\'  - ukośnik (backslash), czyli znak \
'\?'  - znak zapytania
'\''   - pojedynczy cudzysłów
'\"'  - podwójny cudzysłów
```

Zapamiętanie tej listy znaków nie jest takie trudne. Na dodatek, większości z tych znaków nie będziesz używać zbyt często. Tak naprawdę najczęściej będziesz używać znaku nowej linii.

# Typ napisowy

Mimo, że sam typ znakowy jest przydatny, to ma on jedno ograniczenie: za jednym razem można pobrać/wypisać tylko jeden znak. W przypadku gdy chcemy wypisać kilka liter, posługiwanie się typem znakowym nie jest zbyt wygodne.

Typem napisowym w języku C++ jest:

**string** - typ napisowy

Bardzo ważne jest, że aby korzystać z typu napisowego, musimy dołączyć plik nagłówkowy **string**. Przy okazji warto zaznaczyć, że w języku C, czyli poprzedniku języka C++, nie stosowało się tego typu.

Poniżej przedstawiam przykładowy program, w którym pobieramy napis z klawiatury, a następnie wypisujemy pobrany napis. Napis musi się składać ze znaków niesterujących, czyli nie może to być znaków typu Ctrl, Shift, Alt, F1 itd. Dodatkowo, proszę nie stosować białych znaków (tzn. spacji czy znaków tabulacji), bowiem wtedy program nie zadziała "poprawnie".

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

using namespace std;

int main()
{
string napis; // typ napisowy
cout <<"Podaj napis (bez bialych znakow) z klawiatury: ";
cin >>napis;
cin.ignore();

cout <<'\n'<<"Napis podany przez Ciebie to: "<<napis<<".\n\n";

cout <<"Nacisnij ENTER aby zakonczyc\n";
getchar();
return 0;
}
```

Jeśli przyjrzysz się powyższemu programowi, zauważysz coś jeszcze. Zauważ, że wszystkie napisy, które wypisujemy na ekran są wypisywane w podwójnych cudzysłowach ".

Dzięki stosowaniu napisów, możemy znacznie przyspieszyć wypisywanie komunikatów na ekran. Na przykład dwa poniższe fragmenty kodu w C++ są sobie równoważne:

cout <<'a'<<' '<<'b';

cout <<"a b";

Ponadto w napisach możemy również stosować bez problemu wszystkie znaki specjalne (przedstawione przy omawianiu typu znakowego). Dzięki temu również zaoszczędzimy pisania.

```
Na przykład zamiast napisać:
cout <<"Nacisnij ENTER"<<'\n';
Możemy napisać tak:
cout <<"Nacisnij ENTER\n";
```

# Typ napisowy a pobieranie danych

Mam nadzieję, że Twoją uwagę zwróciło to, że w przykładowym programie nie wolno Ci było podawać w napisie żadnych białych znaków, np. spacji. Z kolei jak widzisz napisy wypisywane na ekran zawierają spacje.

Prawda jest taka, że napisy i zmienne typu **string** mogą bez problemu zawierać różne znaki, w tym również spacje, tabulatory itd. Problemem jest sposób pobierania napisu z klawiatury. Standardowo dane z klawiatury są pobierane do pierwszego białego znaku (czyli znaków takich jak enter, spacja, znak tabulacji) i bez użycia dodatkowych zabiegów nie da się tego zmienić.

My na razie nie będziemy się tym problemem zajmować, bowiem pobieranie danych to znacznie bardziej złożone zagadnienie i na razie myślę, że warto skupić się na innych kwestiach.

# Podsumowanie

Dzięki temu artykułowi wiesz już jak możesz wypisywać wszystkie napisy w dowolnych kombinacjach. Jest to bardzo ważna część, bowiem w ten sposób umożliwiasz sobie komunikację z użytkownikiem.

# Lekcja 7: Operatory część pierwsza - operator przypisania i operatory arytmetyczne

## Czym są operatory

Operatory w C++ pełnią funkcję podobną jak operatory w matematyce. Umożliwiają one po prostu przeprowadzanie różnych działań na zmiennych. Samych grup operatorów jest co najmniej kilka, dlatego czas je poznać, bowiem bez nich dalsza nauka języka nie będzie możliwa.

## Najważniejszy operator - operator przypisania

Jednym z najważniejszych operatorów w języku C++ jest operator przypisania. Operator ten umożliwia przypisywanie wartości danej zmiennej.

= operator przypisania (**to nie jest operator równości!**)

Zwróć uwagę na zapis - operator ten przypomina matematyczny operator równości, ale to nie jest operator równości, tylko operator przypisania. Warto też zastanowić się jak wygląda sama operacja przypisania.

Otóż, aby dokonać przypisania, po lewej stronie operatora musimy mieć zmienną. Po prawej stronie możemy natomiast mieć zmienną, jakąś liczbę, znak lub napis (czyli tzw. literał) albo wyrażenie (na przykład sumę dwóch liczb, różnicę zmiennej i liczby itp.).

Oto przykładowy program demonstrujący użycie operatora przypisania:

```
int main()
{
  int liczba;
  string napis;
  char znak;

  liczba=45; // operator przypisania dla liczby
  napis="Ala ma kota"; // operator przypisania dla napisu
  znak='c'; //operator przypisania dla znaku

  cout <<"Oto wartosci zmiennych: "<<liczba<<' '<<napis<<' '<<znak<<"\n\n";
  cout <<"Podaj wartosc liczby: ";
  cin >>liczba;
  cin.ignore();
  cout <<"Wpisz jakis napis (bez bialych znakow): ";
  cin >>napis;
  cin.ignore();

  cout <<"Podaj znak: ";
  cin >>znak;
  cin.ignore();

  cout <<"Oto wartosci zmiennych: "<<liczba<<' '<<napis<<' '<<znak<<"\n\n";

  cout <<"Nacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Jak widzisz operator przypisania możemy zastosować do różnych typów danych. W przykładowym programie, zastosowaliśmy go zarówno dla liczby (tutaj dla typu int, ale mógł to być dowolny typ liczbowy), dla znaku oraz napisu.

# Operatory arytmetyczne

Kolejną grupę stanowią operatory arytmetyczne. Operatory te działają jak znane z matematyki operatory. Oto lista dostępnych operatorów arytmetycznych:

+ operator dodawania
- operator odejmowania
* operator mnożenia
/ operator dzielenia
% operator reszty z dzielenia (zwany operatorem **modulo**)
+ operator znaku liczby (np. +3). Liczba domyślnie jest zawsze dodatnia.
- operator znaku liczby (np. -45, -5.678)

Większość z wyżej wymienionych operatorów arytmetycznych działa tylko i wyłącznie dla typów liczbowych. Niektóre z operatorów działają również na typie znakowym czy napisowym, ale na razie załóżmy, że tych operatorów będziemy używać wyłącznie dla typów liczbowych.

Warto zwrócić na uwagę, na operator **reszty z dzielenia %**. Operator ten zwraca resztę z dzielenia, czyli np. 10%3 da nam wynik 1, 5%5 da nam wynik 0, 3%5 da nam wynik 3. Aby stosować ten operator, po jego obu stronach musimy mieć **liczby całkowite dodatnie**. Na przykład dla operacji 5%-5 wynik operacji jest nieokreślony.

Zastosowanie operatorów arytmetycznych jest bardzo duże. Z punktu widzenia składni języka, poprawne będą następujące wyrażenia:

```
2 + 3
2 * 3 - 5
2 + 6 + 8 + 10
2 / 3 * 2
```

Jak więc widzisz, można wykonywać bez problemu operacje matematyczne nawet dla większej grupy liczb. To, w jakiej kolejności będą wykonywane działania, zależy od **priorytetów operatorów**. Akurat w przypadku operatorów arytmetycznych, priorytety operatorów są takie same, jak to wiesz z lekcji matematyki.

Oto przykładowy program pokazujący zastosowanie operatorów arytmetycznych. Od teraz w wielu przykładowych programach, danych nie będziemy już pobierać z klawiatury. Będziemy stosować przypisanie za pomocą operatora przypisania. Jedynym celem takiego zabiegu jest skrócenie kodów programów - wynik działania pozostanie ten sam.

```
#include <iostream>
#include <string>

using namespace std;

include <iostream>

using namespace std;

int main()
{
  int liczba1;
  int liczba2;
  double liczba3;
  double liczba4;

  liczba1=14;
  liczba2=6;
  liczba3=56.78;
  liczba4=22.11;

  cout <<liczba1<<' '<<liczba2<<'
'<<liczba3<<' '<<liczba4<<'\n';
  cout <<"Suma dwoch liczb:
"<<liczba1+liczba2<<'\n';
  cout <<"Roznica dwoch liczb:
"<<liczba2-liczba3<<'\n';
  cout <<"Reszta z dzielenia:
"<<liczba1%liczba2<<'\n';

  liczba1=liczba1%liczba2; // zmiana
wartosci zmiennej liczba1
  cout <<liczba1<<' '<<liczba2<<'
'<<liczba3<<' '<<liczba4<<'\n';

  liczba3=liczba4-liczba1+liczba2;
//zmiana wartosci zmiennej liczba3
  cout <<liczba1<<' '<<liczba2<<'
'<<liczba3<<' '<<liczba4<<'\n';

  cout <<"Suma wszystkich liczb:
"<<liczba1+liczba2+liczba3+liczba4<<'\n
';
  cout <<"Inne:
"<<liczba1*liczba2+liczba3/liczba4-
liczba2*liczba4<<'\n';
  cout <<liczba1<<' '<<liczba2<<'
'<<liczba3<<' '<<liczba4<<'\n';

  cout <<"\nNacisnij ENTER aby
zakonczyc\n";
  getchar();
  return 0;
}
```

Jak więc widzisz w powyższym programie zastosowaliśmy poznane operatory. Można było zaobserwować, że rzeczywiście można uzyskać bardzo różne wyniki.

Jednak najciekawszą sprawą jest tutaj możliwość zmiany wartości zmiennej. W linijkach opatrzonych komentarzami, dzięki zastosowaniu operatorów arytmetycznych oraz co ważniejsze operatora przypisania, zmieniamy wartość zmiennej.

# Zmiana wartości zmiennej w oparciu o nią samą

Dość często w programach zdarza się, że chcemy np. dodać do obecnej zmiennej typu liczbowego pewną wartość lub wyrażenie. Np. dla zmiennej o nazwie liczba, zwiększenie jej wartości o 2 uzyskujemy pisząc:

liczba=liczba+2;

Co prawda sam zapis może na początku trochę dziwny, jednak można go dosłownie przetłumaczyć sobie tak: **nowej wartości zmiennej liczba przypisz sumę obecnej wartości tej zmiennej i liczby 2**.

Tego typu operacje wykonywane są bardzo często. Oprócz chęci dodania do zmiennej danej wartości lub wyrażenia, możemy chcieć zastosować również odejmowanie, mnożenie, dzielenie czy przypisanie reszty z dzielenia.

W związku z tym, że tego typu operacje wykonywane są w języku C++ naprawdę bardzo często, została stworzona specjalna grupa operatorów, która umożliwia szybsze zapisanie wykonywanej operacji w oparciu o już istniejącą wartość zmiennej. Grupą tą są **złożone operatory przypisania**.

Oto najważniejsze złożone operatory przypisania:

+= zmienna += 2 to to samo co zmienna = zmienna + 2
-= zmienna -= 2 to to samo co zmienna = zmienna - 2

*=  zmienna *= 2 to to samo co zmienna = zmienna * 2
/=  zmienna /= 2 to to samo co zmienna = zmienna / 2
%=  zmienna %= 2 to to samo co zmienna = zmienna % 2

Dzięki stosowaniu powyższych operatorów, możesz w znacznie prostszy sposób zapisywać pewne operacje, co w rezultacie doprowadzi do zaoszczędzenia Twojego cennego czasu.

## Podsumowanie

W tej lekcji przedstawiłem Ci podstawowe operatory stosowane w języku C++. Operatorów takich jest znacznie więcej. Kontynuacja tematyki dotyczącej operatorów znajduje się w następnej lekcji.

# Lekcja 8: Operatory część druga - operatory inkrementacji i dekrementacji, relacji oraz operatory logiczne

## Wprowadzenie

W poprzedniej lekcji przedstawiłem Ci podstawowe operatory: operator przypisania, operatory arytmetyczne oraz złożone operatory przypisania. Dzięki tej lekcji poznasz kolejne rodzaje operatorów. Wszystkie przedstawione tutaj operatory są nie mniej przydatne i nie rzadziej używane od tych przedstawionych w poprzedniej lekcji.

## Operatory inkrementacji i dekrementacji

Operatory te można nazwać operatorami spokrewnionymi z operatorem przypisania, dodawania i odejmowania, bowiem powodują one zmniejszenie lub zwiększenie wartości zmiennej o 1.

++ zwiększenie wartości o 1, np. i++ to jest to samo co i=i+1
 --  zmniejszenie wartości o 1, np. i-- to jest to samo co i=i-1

Należy jednak zwrócić uwagę, że operatory te można stosować przed i po zmiennej, tzn. **liczba++** lub **++liczba**. Podobnie jest z operatorem dekrementacji --. Mimo, że działanie operatora w obu wypadkach jest podobne, to nie jest jednak identyczne!

Aby to wyjaśnić, muszę Ci zdradzić, że w języku C/C++ prawie każde wyrażenie zwraca jakąś wartość. Np. operacja 5+6 zwraca liczbę 11, i dzięki temu int liczba=5+6; powoduje przypisanie zmiennej liczba wartości 11.

Operatory ++ i -- również zwracają wartość, z tym tylko, że jeśli operator znajduje się przed zmienną, to zwraca wartość zmiennej po zwiększeniu(zmniejszeniu), a jeśli znajduje się po zmiennej, to zwraca wartość zmiennej przed wykonaniem operacji zwiększenia(zmniejszenia). Jednak **wartość zmiennej w obu przypadkach będzie taka sama**.

Jeśli nie do końca udało Ci się zrozumieć jaka jest różnica, oto prosty program, który powinien wyjaśnić wszystkie wątpliwości:

```
#include <iostream>

using namespace std;

int main()
{
  int liczba1;
  int liczba2;
  int liczba3;
  int liczba4;

  liczba1=5;
  liczba2=5;
  liczba3=5;
  liczba4=5;

  cout <<"Wartosc liczba1++ to
"<<liczba1++
       <<" a wartosc zmiennej to
"<<liczba1<<'\n';
  cout <<"Wartosc ++liczba2 to "<<+
+liczba2
       <<" a wartosc zmiennej to
"<<liczba2<<'\n';
  cout <<"Wartosc liczba3-- to
"<<liczba3--
       <<" a wartosc zmiennej to
"<<liczba3<<'\n';
  cout <<"Wartosc --liczba4 to "<<--
liczba4
       <<" a wartosc zmiennej to
"<<liczba4<<'\n';

  cout <<"\nNacisnij ENTER aby
zakonczyc\n";
  getchar();
  return 0;
}
```

Przy okazji chcę zaznaczyć, że operatorów inkrementacji i dekrementacji nie należy stosować w bardziej skomplikowanych wyrażeniach takich jak na przykład **a++ + ++b + 4 + ++a**. Wynik takiej operacji będzie zależeć bowiem od kompilatora, czyli tak naprawdę nigdy nie można być pewnym jaka będzie końcowa wartość wyrażenia.

# Operatory relacji

Operatory operacji umożliwiają porównaniu różnych wyrażeń. Za ich pomocą możemy sprawdzić czy dwie liczby lub wyrażenia są sobie równe. Podobnie jak wszystkie wymienione do tej pory operatory, są bardzo często stosowane.

Oto operatory relacji:

== równa się (np. 2==3 jest nieprawdziwe, a 5==5 jest prawdziwe)
!= jest różne (np. 2!=3 jest prawdziwe, a 5!=5 jest nieprawdziwe)
> jest większe
>= jest większe lub równe
< jest mniejsze
<= jest mniejsze lub równe

O ile zapamiętanie większości z tych operatorów nie sprawi Ci trudności, to bardzo trudno zapamiętać w jaki sposób używamy operatora równości. Aby sprawdzić czy dwie liczby są jednakowe piszemy **liczba1==liczba2**.

Łatwo można się pomylić i napisać liczba1=liczba2, co spowoduje przypisanie wartości zmiennej o nazwie liczba1 wartości zmiennej o nazwie liczba2, gdyż operator = jest (jak już wiesz) operatorem przypisania. Przy okazji całe wyrażenie zwróci wartość zmiennej liczba2, co może spowodować, że znalezienie błędu będzie bardzo trudne.

Dlatego też, warto nawet powiesić sobie kartkę z tymi dwoma operatorami w okolicach komputera. Dzięki temu unikniesz częstych błędów i zaoszczędzisz swój czas.

# Operatory logiczne

Operatory logiczne umożliwiają łączenie ze sobą kilku wyrażeń. Są bardzo przydatne i bardzo często wykorzystywane, gdy w naszym programie podejmujemy pewne decyzje. W języku C++ występują trzy operatory logiczne.

|| suma logiczna - prawdziwe jeśli ktorekolwiek z wyrażeń jest prawdziwe
&& iloczyn logiczny - prawdziwe jeśli oba wyrażenia są prawdziwe
! negacja logiczna - powoduje zaprzeczenie wyrażenia

Dla przykładu wyrażenie **(3==5) || (2==2)** jest prawdziwe, bowiem pierwsza część jest fałszem, a druga prawdą, czyli co najmniej jedno wyrażenie jest prawdziwe. Z kolei **(4>=2) && ((8%2)==0)** jest również prawdziwe, bowiem zarówno pierwsza, jak i druga część są prawdziwe. Z kolei **! (3==5)** jest prawdziwe, bo wyrażenie 3==5 jest nieprawdziwe. Podobnie **!((3==5) || (2==2))** jest nieprawdziwe, bo samo wyrażenie (3==5) || (2==2) jest prawdziwe.

Warto przy okazji wspomnieć, że przypisanie = zwraca wartość przypisania czyli np. a=3 zwraca wartość 3, a każda liczba różna od zera jest w języku C++ traktowana jako prawda. Tylko wartość 0 jest traktowana jako fałsz.

Ma to bardzo duże znaczenie, bowiem jeśli napiszemy (najprawdopodobniej przez pomyłkę) **(a=2) || (3==5)**, to się okaże, że jest prawdziwe, gdyż a=2 zwraca wartość 2, która zostaje potraktowana jako prawda. Druga część jest oczywiście fałszywa, ale ponieważ pierwsza część była prawdziwa, to całość jest również potraktowana jako prawda.

W poniższym programie wypiszemy wartości kilku przykładowych wyrażeń. Wypisana wartość 0 oznacza, że wyrażenie jest fałszywe, a wartość 1, że wyrażenie jest prawdziwe.

```
#include <iostream>

using namespace std;

int main()
{
  int a; // zmienna uzyta w 3. przykladzie

  cout <<"(3==5) || (2==2) to " <<((3==5) || (2==2))<<'\n';
  cout <<"(4>=2) && ((8%2)==0)  to "<< ((4>=2) && ((8%2)==0))<<'\n';
  cout <<"(a=2) || (3==5) to "<<((a=2) || (3==5))<<'\n';
  cout <<"1+1>3 to " << (1+1>3)<<'\n';
  cout <<"2+2>2+2 to " << (2+2>2+2)<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

W tym momencie zachęcam Cię do wymyślenia kilku przykładowych wyrażeń i przed uruchomieniem programu wypisującego ich wartość zachęcam Cię do przemyślenia, jaką wartość program wypisze. Bardzo ważne jest zrozumieć, jak działają takie wyrażenia.

Zwróć również uwagę, że wszystkie wyrażenia wypisywane są otoczone nawiasami okrągłymi. Kiedy będziesz wymyślać własne przykłady, nie zapomnij ich dodać, bowiem w przeciwnym wypadku otrzymasz nieoczekiwane rezultaty.

# Priorytety i łączność operatorów

Jak już wspomniałem w poprzedniej lekcji, o tym które wyrażenia są wykonywane najpierw, decyduje priorytet operatorów.

Żeby udowodnić Ci, jak zgubny skutek ma nieznajomość (lub raczej nieumiejętne posługiwanie się operatorami), zmień w poprzednim przedstawionym programie pierwszą linijkę, w której wypisujemy wynik operacji na taką linijkę:

```
cout <<"(3==5) || (2==2) to " <<(3==5) || (2==2)<<'\n';
```

Skompiluj program i uruchom. Zauważysz, że stało się coś dziwnego - została wypisana wartość 0 (wcześniej była wartość 1), a co najdziwniejsze, nie nastąpiło przejście do nowej linii. Tak właśnie może się skończyć nieumiejętne korzystanie z operatorów. W tym wypadku chodzi o to, że << to również operator (o czym do tej pory nie wspominałem).

Musisz wiedzieć, że wszystkie operatory mają swoje priorytety i łączność. **Priorytet** oznacza, że jeśli użyjemy dwóch operatorów o różnych priorytetach, to któraś z operacji wykona się najpierw (np. mnożenie wykona się przed dodawaniem w wyrażeniu a=b+c*d).

Łączność określa sposób ustawienia "niewidocznych dla użytkownika nawiasów". Np. operator + jest łączny lewostronnie, zatem a=b+c+d oznacza tak naprawdę a=(b+c)+d.

Mimo że istnieją bardzo długie i trudne do zapamiętania tabelki priorytetów operatorów, ja jednak zrezygnuję na razie z przedstawiania ich tutaj. Myślę, że to w tym momencie i tak niepotrzebne, a poza tym nie chcę Cię zbytnio przerazić.

Sam muszę się przyznać, że do tej pory nie znam priorytetów większości operatorów, co wcale nie przeszkadza mi programować. Również Ty możesz sobie poradzić z tym problemem moją metodą - jeśli nie wiesz, w jakiej kolejności zostanie przeprowadzone pewne działanie na wyrażeniu **użyj nawiasów okrągłych**, na przykład **a=(b+c)*d** lub **a=b+(c*d)**. O ile w pierwszym wypadku użycie nawiasów jest uzasadnione, to w drugim - nawiasy są zbędne. Ale zapamiętaj - jeśli nie wiesz w jakiej kolejności działają operatory, lepiej użyć nawet kilku par nawiasów niż martwić się, że później program nie działa poprawnie.

# Podsumowanie

W tym momencie znasz już podstawowe operatory występujące w języku C++. Tak naprawdę operatorów jest co najmniej kilka razy więcej niż znasz ich do tej pory, jednak nie musisz znać wszystkich od razu, a poza tym z niektórych naprawdę rzadko się korzysta.

# Strumienie - czym są i jakie są ich rodzaje

Strumienie można sobie wyobrazić jako dane płynące od źródła do celu. Zarówno cele, jak i źródła strumienia mogą być różne - może to być ekran, klawiatura, plik lub jakieś inne zewnętrzne urządzenie.

W języku C++ wyróżniamy trzy typy strumieni: strumienie **wejściowe** (wczytują dane), strumienie **wyjściowe** (wypisują dane) oraz strumienie **uniwersalne**, umożliwiające zarówno wczytywanie, jak i wypisywanie danych.

Strumieniami posługujemy się wówczas, gdy chcemy przeprowadzić jakiekolwiek operacje wejścia/wyjścia. W przedstawionych programach takie operacje przeprowadzaliśmy wielokrotnie - wiele razy pobieraliśmy dane z klawiatury i wiele razy wypisywaliśmy dane na ekran. W tym celu posługiwaliśmy się właśnie strumieniami.

Musisz sobie zdawać sprawę, że strumienie w języku C++ są ogromnym ułatwieniem. Nie raz wypisywaliśmy już zmienne na ekran i zmienna została wypisywana zawsze poprawnie. Tak

naprawdę wszystkie dane, które strumień wypisuje na ekran muszą zostać **sformatowane** . Jeśli wypisujemy liczbę float, to liczba zostaje wypisana jako liczba float. Taka sama sytuacja jest przy pobieraniu danych.

Jeśli wydaje Ci się to czymś oczywistym, to musisz wiedzieć, że wypisywanie i pobieranie danych w języku C, czyli poprzedniku języka C++ nie było wcale takie łatwe. Tam musieliśmy za każdym razem określić, w jaki sposób ma zostać wypisana dana zmienna - czy mamy wypisać zmienną jako liczbę całkowitą, znak czy może napis. Sprawiało to, że to programista musiał dbać o to, czy dane zostaną wypisane poprawnie - w języku C++ na szczęście przy zwykłym wypisywaniu nie musimy się tym martwić.

Dla przykładu, aby wypisać liczbę typu float, w C++ piszemy:

cout << zmienna;

Z kolei w języku C musieliśmy dołączyć bibliotekę **stdio.h** i zrobić to w następujący sposób:

printf("%f",zmienna);

# Strumienie - czym są i jakie są ich rodzaje

Strumienie można sobie wyobrazić jako dane płynące od źródła do celu. Zarówno cele, jak i źródła strumienia mogą być różne - może to być ekran, klawiatura, plik lub jakieś inne zewnętrzne urządzenie.

W języku C++ wyróżniamy trzy typy strumieni: strumienie **wejściowe** (wczytują dane), strumienie **wyjściowe** (wypisują dane) oraz strumienie **uniwersalne**, umożliwiające zarówno wczytywanie, jak i wypisywanie danych.

Strumieniami posługujemy się wówczas, gdy chcemy przeprowadzić jakiekolwiek operacje wejścia/wyjścia. W przedstawionych programach takie operacje przeprowadzaliśmy wielokrotnie - wiele razy pobieraliśmy dane z klawiatury i wiele razy wypisywaliśmy dane na ekran. W tym celu posługiwaliśmy się właśnie strumieniami.

Musisz sobie zdawać sprawę, że strumienie w języku C++ są ogromnym ułatwieniem. Nie raz wypisywaliśmy już zmienne na ekran i zmienna została wypisywana zawsze poprawnie. Tak naprawdę wszystkie dane, które strumień wypisuje na ekran muszą zostać **sformatowane** . Jeśli wypisujemy liczbę float, to liczba zostaje wypisana jako liczba float. Taka sama sytuacja jest przy pobieraniu danych.

Jeśli wydaje Ci się to czymś oczywistym, to musisz wiedzieć, że wypisywanie i pobieranie danych w języku C, czyli poprzedniku języka C++ nie było wcale takie łatwe. Tam musieliśmy za każdym razem określić, w jaki sposób ma zostać wypisana dana zmienna - czy mamy wypisać zmienną jako liczbę całkowitą, znak czy może napis. Sprawiało to, że to programista musiał dbać o to, czy dane zostaną wypisane poprawnie - w języku C++ na szczęście przy zwykłym wypisywaniu nie musimy się tym martwić.

Dla przykładu, aby wypisać liczbę typu float, w C++ piszemy:

cout << zmienna;

Z kolei w języku C musieliśmy dołączyć bibliotekę **stdio.h** i zrobić to w następujący sposób:

printf("%f",zmienna);

# Dodatkowe informacje dotyczące strumieni

Jak już udało Ci się zaobserwować w programach, za jednym razem możemy wypisać tylko jedną daną lub ich kilka. Na przykład fragment:

```
cout <<zmienna1;
cout <<' ';
cout <<zmienna2;
cout <<'\n';
```

możemy bez problemu zastąpić takim krótszym zapisem:

```
cout <<zmienna1<<' '<<zmienna2<<'\n';
```

Na tej samej zasadzie możemy łączyć kilka danych podczas pracy z każdym innym strumieniem, zarówno wejściowym, jak i wyjściowym.

Oprócz tego warto zwrócić uwagę na pewne domyślne ustawienia strumieni. Jest ich bardzo wiele i na tym etapie kursu chcę Ci jedynie zasygnalizować istnienie takich ustawień.

Otóż strumienie mają zdefiniowanych bardzo wiele standardowych parametrów. Na przykład liczby zmiennopozycyjne zostają wypisane z dokładnością 6 miejsc po przecinku. Jeśli chcemy wypisać liczbę z większą liczbą miejsc po przecinku, będziemy musieli to strumieniowi zasygnalizować.

Podobnie rzecz się ma w przypadku pobierania danych z urządzenia. Na przykład wszystkie białe znaki znajdujące się przed pobieranymi danymi są ignorowane. Ponadto za koniec danych uznawany jest pierwszy biały znak. Dlatego też nie uda nam się tak łatwo wczytać zdania: "Ala ma kota.". Zostanie wczytane tylko pierwsze słowo. Natomiast drugie słowo zostanie uznane za zupełnie nową daną, co może mieć katastrofalne skutki.

Takich domniemań w pracy strumieni jest naprawdę bardzo wiele. Na razie założymy, że jesteśmy zadowoleni ze standardowych ustawień i przejdziemy do ważniejszych kwestii w języku C++.

## Podsumowanie

Dzięki tej lekcji udało Ci się zrozumieć, w jaki sposób można pobierać dane i je wypisywać. Teraz wiesz już czym są strumienie i zdajesz sobie sprawę, że istnieje bardzo wiele opcji umożliwiających zmianę domyślnych ustawień w pracy strumieni, które postaram Ci się przedstawić w odpowiednim momencie.

# Lekcja 9: Strumienie i operacje wejścia/wyjścia

## Strumienie - czym są i jakie są ich rodzaje

Strumienie można sobie wyobrazić jako dane płynące od źródła do celu. Zarówno cele, jak i źródła strumienia mogą być różne - może to być ekran, klawiatura, plik lub jakieś inne zewnętrzne urządzenie.

W języku C++ wyróżniamy trzy typy strumieni: strumienie **wejściowe** (wczytują dane), strumienie **wyjściowe** (wypisują dane) oraz strumienie **uniwersalne**, umożliwiające zarówno wczytywanie, jak i wypisywanie danych.

Strumieniami posługujemy się wówczas, gdy chcemy przeprowadzić jakiekolwiek operacje wejścia/wyjścia. W przedstawionych programach takie operacje przeprowadzaliśmy wielokrotnie - wiele razy pobieraliśmy dane z klawiatury i wiele razy wypisywaliśmy dane na ekran. W tym celu posługiwaliśmy się właśnie strumieniami.

Musisz sobie zdawać sprawę, że strumienie w języku C++ są ogromnym ułatwieniem. Nie raz wypisywaliśmy już zmienne na ekran i zmienna została wypisywana zawsze poprawnie. Tak naprawdę wszystkie dane, które strumień wypisuje na ekran muszą zostać **sformatowane** . Jeśli wypisujemy liczbę float, to liczba zostaje wypisana jako liczba float. Taka sama sytuacja jest przy pobieraniu danych.

Jeśli wydaje Ci się to czymś oczywistym, to musisz wiedzieć, że wypisywanie i pobieranie danych w języku C, czyli poprzedniku języka C++ nie było wcale takie łatwe. Tam musieliśmy za każdym razem określić, w jaki sposób ma zostać wypisana dana zmienna - czy mamy wypisać zmienną jako liczbę całkowitą, znak czy może napis. Sprawiało to, że to programista musiał dbać o to, czy dane zostaną wypisane poprawnie - w języku C++ na szczęście przy zwykłym wypisywaniu nie musimy się tym martwić.

Dla przykładu, aby wypisać liczbę typu float, w C++ piszemy:

```
cout << zmienna;
```

Z kolei w języku C musieliśmy dołączyć bibliotekę **stdio.h** i zrobić to w następujący sposób:

```
printf("%f",zmienna);
```

# Strumienie predefiniowane

Strumienie predefiniowane to strumienie stworzone dla Ciebie i gotowe do korzystania. Gdyby nie istniały, trzeba by je było stworzyć samemu, co niewątpliwie nie było by zbyt łatwym zadaniem. Aby skorzystać ze strumieni predefiniowanych należy dołączyć bibliotekę **iostream** - jak widzisz w każdym programie do tej pory właśnie dołączaliśmy tę bibliotekę.

W języku C++ istnieją cztery strumienie predefiniowane:

**cout** - powiązany ze standardowym urządzeniem wyjścia
**cin** - powiązany ze standardowym urządzeniem wejścia
**cerr** - strumień błędów - połączony ze standardowym urządzeniem wyjścia
**clog** - podobnie jak **cerr**, ale wydajniejszy przy wielu danych

Jak więc widzisz każdy ze strumieni ma jakieś przeznaczenie. To, że strumień jest powiązany ze standardowym urządzeniem wyjścia, oznacza w większości wypadków, że jest połączony z ekranem. Z kolei to, że strumień jest powiązany ze standardowym urządzeniem wejścia, oznacza, że jest on powiązany z klawiaturą.

Na szczęście język C++ jest na tyle elastyczny, że możemy zmieniać powiązanie każdego z predefiniowanych strumieni. Zazwyczaj wiąże się strumień cerr lub clog z plikiem na dysku. Pozostałych powiązań raczej się nie zmienia, chociaż jest to możliwe i niekiedy wygodne.

Nas oczywiście najbardziej interesują dwa pierwsze strumienie: strumień wyjściowy **cout** oraz strumień wejściowy **cin**. Z tych strumieni korzystaliśmy już nie raz.

# Operatory << i >>

Wiesz już czym są strumienie. Same strumienie jednak nam nie wystarczą. Aby móc przeprowadzić jakąkolwiek operację na strumieniach korzystamy z następujących operatorów:

<< - operator wysyłania do strumienia
>> - operator pobierania ze strumienia

W praktyce będzie to oznaczało, że jeśli chcemy coś wypisać, używamy operatora <<. Natomiast gdy chcemy pobrać dane, użyjemy operatora >>. Można zatem sobie zapamiętać, że piszemy **cout** << oraz **cin** >>.

## Dodatkowe informacje dotyczące strumieni

Jak już udało Ci się zaobserwować w programach, za jednym razem możemy wypisać tylko jedną daną lub ich kilka. Na przykład fragment:

```
cout <<zmienna1;
cout <<' ';
cout <<zmienna2;
cout <<'\n';
```

możemy bez problemu zastąpić takim krótszym zapisem:

```
cout <<zmienna1<<' '<<zmienna2<<'\n';
```

Na tej samej zasadzie możemy łączyć kilka danych podczas pracy z każdym innym strumieniem, zarówno wejściowym, jak i wyjściowym.

Oprócz tego warto zwrócić uwagę na pewne domyślne ustawienia strumieni. Jest ich bardzo wiele i na tym etapie kursu chcę Ci jedynie zasygnalizować istnienie takich ustawień.

Otóż strumienie mają zdefiniowanych bardzo wiele standardowych parametrów. Na przykład liczby zmiennopozycyjne zostają wypisane z dokładnością 6 miejsc po przecinku. Jeśli chcemy wypisać liczbę z większą liczbą miejsc po przecinku, będziemy musieli to strumieniowi zasygnalizować.

Podobnie rzecz się ma w przypadku pobierania danych z urządzenia. Na przykład wszystkie białe znaki znajdujące się przed pobieranymi danymi są ignorowane. Ponadto za koniec danych uznawany jest pierwszy biały znak. Dlatego też nie uda nam się tak łatwo wczytać zdania: "Ala ma kota.". Zostanie wczytane tylko pierwsze słowo. Natomiast drugie słowo zostanie uznane za zupełnie nową daną, co może mieć katastrofalne skutki.

Takich domniemań w pracy strumieni jest naprawdę bardzo wiele. Na razie założymy, że jesteśmy zadowoleni ze standardowych ustawień i przejdziemy do ważniejszych kwestii w języku C++.

## Podsumowanie

Dzięki tej lekcji udało Ci się zrozumieć, w jaki sposób można pobierać dane i je wypisywać. Teraz wiesz już czym są strumienie i zdajesz sobie sprawę, że istnieje bardzo wiele opcji umożliwiających zmianę domyślnych ustawień w pracy strumieni, które postaram Ci się przedstawić w odpowiednim momencie.

# Lekcja 10: Instrukcja warunkowa if - podejmowanie decyzji w języku C++

## Instrukcja warunkowa - to daje możliwości

Jak na razie przedstawiłem Ci czym są zmienne, jakie są podstawowe operatory oraz w jaki sposób możemy wypisywać i pobierać dane. Mówiąc szczerze znając te elementy nie możemy napisać żadnego sensownego programu.

Dopiero poznanie instrukcji **if** umożliwi Ci pisanie programów, które będą już naprawdę mogły coś robić. Użytkownik będzie mógł na przykład zdecydować którą opcję chce wybrać, a program w zależności od wybranej opcji będzie mógł wykonać określoną czynność.

# Postać instrukcji warunkowej

Poniżej przedstawiam postać instrukcji warunkowej **if**:

```
if (warunek)
{
  instrukcja_1;
  instrukcja_2;
  ...
  instrukcja_n;
}
else
{
  instrukcja_1;
  instrukcja_2;
  ...
  instrukcja_n;
}
```

Pamiętaj jednak, że jest to tylko schemat instrukcji, więc jeśli wpiszesz w kompilatorze dokładnie coś takiego, to program się oczywiście nie skompiluje.

Instrukcja if działa w **następujący sposób**: jeśli warunek jest spełniony wykonuje się pierwsza lista instrukcji, a jeśli warunek nie jest spełniony wykonuje się druga lista instrukcji.

Wspomniałem kiedyś, że nawiasów klamrowych używa się do grupowania instrukcji - i właśnie jak widzisz, używamy ich tutaj. Możemy ich nie użyć tylko wtedy gdy będziemy chcieli wykonać **jedną** instrukcję. Zatem jeśli chcielibyśmy wykonać tylko jedną instrukcję w każdym przypadku, wystarczyłby taki **schemat**:

```
if (warunek)
  instrukcja_1;
else
  instrukcja_2;
```

Do tej pory przedstawiałem tylko schemat wyglądu instrukcji warunkowej **if**. Tym razem przedstawiam przykładowy program:

```
#include <iostream>

using namespace std;

int main()
{
  char znak;
  int liczba1;
  int liczba2;


  cout <<"Podaj pierwsza liczbe
calkowita: ";
  cin >>liczba1;
  cin.ignore();
  cout <<"Podaj druga liczbe calkowita:
";
  cin >>liczba2;
  cin.ignore();
  cout <<"Teraz wpisz + lub - a
nastepnie nacisnij ENTER: ";
  cin >>znak;
  cin.ignore();
  if (znak=='+') // jesli znak to +
     cout <<"Suma liczb wynosi
"<<liczba1+liczba2<<'\n';
  else //znak nie jest +
  {
     cout <<"Wybrano
(najprawdopodobniej) odejmowanie
liczb\n";
     cout <<"Roznica liczb wynosi
"<<liczba1-liczba2<<'\n';
  }

  cout <<"\nNacisnij ENTER aby
zakonczyc\n";
  getchar();
  return 0;
}
```

Przede wszystkim zwróć uwagę na **instrukcję if**, bowiem to ona jest najważniejsza. Zauważ, że w jednym z warunków nie użyłem nawiasów klamrowych, bowiem jest wykonywana tylko jedna instrukcja, natomiast w drugim wypadku nawiasy klamrowe są użyte, bowiem wypisujemy dwa razy dane na ekran (chociaż oczywiście można by to zrobić za jednym razem).

Program pobiera najpierw od użytkownika dwie liczby całkowite. Następnie oczekuje na podanie znaku + lub -. Jeśli jest to znak +, wówczas wypisuje sumę liczb, a jeśli - to wówczas wypisuje różnicę liczb. Co prawda program oczekuje tylko na znak + lub -, jednak użytkownik może podać złośliwie lub przez przypadek inny znak. W takim wypadku również zostanie wykonane odejmowanie liczb.

# Uproszczona instrukcja warunkowa

Mimo, że przedstawiona w poprzednim paragrafie instrukcja warunkowa nie jest bardzo trudna, to istnieje jej uproszczenie. Otóż w instrukcji if **nie wymaga się części else**. Oznacza to, że jeśli zajdzie pewien warunek, to wykonamy dane działanie, a w przeciwnym wypadku nie wykonamy nic.

Spójrz na poniższy fragment kodu:

```
if (znak=='+') // jesli znak to +
  cout <<"Suma liczb wynosi "<<liczba1+liczba2<<'\n';
cout <<"Ta czesc zostanie wykonana zawsze.";
```

W tym przykładowym fragmencie programu, jeśli znak jest znakiem +, to wówczas zostanie wypisana suma. Jeśli natomiast znak będzie jakimkolwiek innym znakiem, nie zostanie wykonane nic. Zawsze natomiast (czyli zarówno gdy znakiem był + jak i dowolny inny znak) zostanie wypisane stwierdzenie "Ta czesc zostanie wykonana zawsze.".

# Używanie wyrażeń logicznych

Ponieważ znasz już operatory relacyjne, wiedz, że możesz ich również używać w instrukcji warunkowej if. W ten sposób stworzysz bardziej skomplikowane warunki logiczne. Na przykład w poniższym programie został użyty operator sumy logicznej:

```
#include <iostream>

using namespace std;

int main()
{
  int liczba;
  liczba=102; // sprawdz dzialanie dla wartosci np. 20 i 3

  if (liczba>100 || liczba <5)
     cout <<"Liczba jest wieksza od 100 lub mniejsza od 5"<<'\n';
  else
     cout <<"Liczba jest z przedzialu pomiedzy 5 i 100"<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Na podobnej zasadzie możesz tu użyć dowolnych innych, mniej lub bardziej skomplikowanych wyrażeń. Od teraz nie będę już o tym wspominał. W większości bowiem wypadków tam, gdzie można użyć zwykłego wyrażenia, można użyć też wyrażeń z operatorami relacyjnymi i pozostałymi operatorami, które Ci przedstawiłem.

# Wielokrotna instrukcja if

Oprócz przedstawionej na samej początku wersji instrukcji if, istnieje jeszcze bardziej ogólna wersja. W takiej wersji można sprawdzić wiele warunków - w pierwotnej wersji mieliśmy tylko jeden warunek. Oto schemat bardziej zaawansowanej instrukcji if:

```
if (warunek1)
{
  instrukcja1;
  instrukcja2;
  ...
}
else if (warunek2)
{
  instrukcja1;
  instrukcja2;
  ...
}
else if (...)
  ...
else
{
  instrukcja1;
  instrukcja2;
  ...
}
```

Zatem w rzeczywistości możemy użyć tak wielu warunków jak sobie tego życzymy. Możemy zatem zmodyfikować nieco nasz program z początku tej lekcji, aby wykonywał jedno z czterech działań matematycznych. Jeśli natomiast zostanie podany inny znak, niż znak matematyczny, wówczas program wypisze stosowany komunikat. Oto zmodyfikowana wersja programu:

```
#include <iostream>

using namespace std;

int main()
{
 char znak;
 double liczba1;
 double liczba2;


 cout <<"Podaj pierwsza liczbe
calkowita: ";
 cin >>liczba1;
 cin.ignore();
 cout <<"Podaj druga liczbe calkowita:
";
 cin >>liczba2;
 cin.ignore();
 cout <<"Podaj znak operacji (+,-,*,/),
a nastepnie nacisnij ENTER: ";
 cin >>znak;
 cin.ignore();
 if (znak=='+') // jesli znak to +
    cout <<"Suma liczb wynosi
"<<liczba1+liczba2<<'\n';
 else if (znak=='-') // jesli znak to -
    cout <<"Roznica liczb wynosi
"<<liczba1-liczba2<<'\n';
 else if (znak=='*') // jesli znak to *
     cout <<"Iloczyn liczb wynosi
"<<liczba1*liczba2<<'\n';
 else if (znak=='/') // jesli znak to /
     cout <<"Iloraz liczb wynosi
"<<liczba1/liczba2<<'\n';
 else //inny znak
     cout <<"To nie jest prawidlowy
znak dzialania!!!\n";

 cout <<"\nNacisnij ENTER aby
zakonczyc\n";
 getchar();
 return 0;
}
```

Zwróć uwagę na pewien szczegół. Typ zmiennych w stosunku do poprzedniej wersji programu został zmieniony z int na **double**. Zastosowałem tę zmianę, aby wynik dzielenia był taki, jak byśmy sobie tego życzyli. W przeciwnym wypadku na przykład z dzielenia liczby 3 przez 4 otrzymywalibyśmy 0 zamiast 0.75. Tę kwestię omówię kiedyś w przyszłości, na razie tylko chciałem Ci zasygnalizować, że zmiana typów zmiennych jest celowa.

# Zagnieżdżanie instrukcji if

Instrukcja if podobnie jak prawie każda inna instrukcja w języku C++ może być zagnieżdżana. Znaczy to, że w naszych schematach zamiast jednej z instrukcji możemy użyć kolejnej instrukcji if.

```
#include <iostream>

using namespace std;

int main()
{
  int a, b;
  cout <<"Podaj dwie liczby calkowite: ";
  cin >>a>>b;
  cin.ignore();
  if (a>5 && b>6)
     if (b>10)
        cout <<"b wieksze od 10, a wieksze od 5";
     else
        cout <<"b pomiedzy 6 i 10 lub rowne 10, a wieksze od 5";
  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Gdybym w powyższym kodzie nie użył wcięć podejrzewam, że ciężko by Ci się było zorientować jak program działa. Tak naprawdę w programie mamy jedną instrukcję warunkową else (bez części else). Jeśli warunek w tej instrukcji jest prawdziwy wówczas wykonuje się instrukcja - a tutaj zamieszczoną instrukcją jest o dziwo również kolejna instrukcja warunkowa.

Gdybym nie zastosował wcięć, na pewno by Cię zastanowiło, do którego warunku odnosi się część else - do pierwszego, czy drugiego? Otóż w języku C++ przyjęto, że **część else odnosi się zawsze do ostatniej instrukcji warunkowej if**. Czyli u nas odnosi się do warunku b>10.

Przy okazji zwróć uwagę na zapis:

int a, b;

Taki zapis jest równoważny zapisowi:

int a;
int b;

Od teraz będę korzystał w przykładowych programach z tej krótszej notacji.

# Podsumowanie

W tej lekcji zaprezentowałem Ci instrukcję warunkową. Jest to jedna z najczęściej używanych instrukcji w języku C++, dlatego też warto się z nią oswoić oraz dobrze zrozumieć jej działanie.

Instrukcja ta pozwoli Ci w przyszłości na pisanie użytecznych programów dlatego też radzę Ci potrenować i przemyśleć jej działanie na własną rękę, konstruując przykładowe, na razie proste programy.

# Lekcja 11: Modyfikatory typów w języku C++. Używanie stałych w programach

## Do czego służą modyfikatory

W jednej z poprzednich lekcji przedstawiłem Ci podstawowe typy danych występujące w języku C ++. Głównym celem modyfikatorów jest w pewnym sensie zmodyfikowanie istniejących typów danych. Oprócz tej funkcji istnieją też modyfikatory mające inne cele.

Może Cię zastanawiać, po co wprowadzać modyfikatory skoro można korzystać z podstawowych typów danych. W pewnym sensie masz nawet rację. Podstawowe, poznane dotychczas typy danych w podstawowych operacjach powinny Ci wystarczyć.

Sytuacja wygląda jednak trochę inaczej, kiedy na przykład potrzebujesz przeprowadzać operacje na bardzo dużych liczbach. Może się okazać, że wbudowane typy danych nie zawsze wystarczają. Wówczas trzeba stworzyć własny typ danych, który umożliwi przeprowadzanie operacji. Nie muszę Cię chyba przekonywać, że nie jest to zbyt łatwe rozwiązanie.

Często natomiast zdarza się jednak, że tak naprawdę do takich operacji wystarczą nam podstawowe typy danych, ale odpowiednio zmodyfikowane. Wtedy właśnie użyjemy modyfikatorów.

Modyfikatory mogą okazać się również bardzo przydatne, gdy pracujemy na bardzo dużej liczbie danych, ale dane te są tak małe, że wbudowane typy danych są dla nas "za duże". W ten sposób niepotrzebnie jest zużywana tak duża ilość pamięci dla jednej danej, podczas gdy tą tak naprawdę niewykorzystaną pamięć można by wykorzystać dla kolejnych danych. Do rozwiązania tego problemu również można wykorzystać modyfikatory.

## Modyfikatory short i long

Za pomocą tych dwóch modyfikatorów możemy zmniejszyć lub powiększyć rozmiar typu danych. W ten sposób możemy albo zaoszczędzić pamięć (typ short) lub umożliwić zapisanie większej liczby (typ long).

Modyfikatorów tych nie możemy jednak zastosować do dowolnego typu danych. Tak naprawdę możliwe są jedynie następujące połączenia:

**short int** - "pomniejszony" typ int
**long int** - "powiększony" typ int
**long double** - "powiększony" typ double

Używając modyfikatorów, musimy zdawać sobie sprawę z konsekwencji. Stosując modyfikator **short** sprawiamy, że typ będzie zajmował najprawdopodobniej mniej miejsca w pamięci, jednak będziemy mogli zapisywać poprawnie mniejsze liczby niż standardowo.

Z kolei stosując modyfikator **long** musimy wiedzieć, że typ będzie teraz zajmował najprawdopodobniej więcej miejsca w pamięci, ale w nagrodę będziemy mogli zapisywać większe liczby.

Przy okazji warto wspomnieć o wprowadzonej skróconej notacji. Deklarując w programie zmienną typu **short int** wystarczy, że napiszemy tylko **short**. Analogicznie, zamiast pisać **long int** wystarczy, że napiszemy tylko **long**. Dla typu long double nie istnieje skrócona notacja.

## Modyfikatory signed i unsigned

Za pomocą modyfikatorów signed i unsigned możemy określić, czy będziemy używać liczb

ujemnych, czy z liczb ujemnych nie będziemy korzystali. **Modyfikatorów signed i unsigned używamy jedynie do typów całkowitych (int, short int, long int) oraz typu znakowego char**.

W efekcie możemy uzyskać następujące kombinacje:

signed int - unsigned int
signed short int - unsigned short int
signed long int - unsigned long int
signed char - unsigned char

Musisz wiedzieć, że modyfikator signed lub unsigned zostaje zawsze ustawiony, nawet jeśli go nie jawnie nie napiszemy. W przypadku typów całkowitych automatycznie jest ustawiany modyfikator **signed**.

Z kolei dla typu znakowego to, jaki modyfikotor zostanie ustawiony, zależy wyłącznie od komputera i od kompilatora. Przyznam się szczerze, że ja do tej pory nie spotkałem się nigdy, aby został ustawiony modifikator **unsigned**, jednak może się tak zdarzyć, dlatego niekiedy dla pewności dla typu znakowego lepiej jawnie napisać modyfikator.

Jeśli jeszcze nie wiesz, kiedy modyfikatory signed i unsigned mogą się przydać, oto przykład. Załóżmy, że chcemy zapisać jak największą liczbę całkowitą. Jakiego typu użyjemy? Użyjemy typu **signed long int** (czyli w uproszczeniu long int), bowiem może nam chodzić zarówno o liczby dodatnie, jak i ujemne.

A co jeśli wiemy, że będziemy operować na dużych liczbach całkowitych dodatnich? Wtedy z kolei użyjemy typu **unsigned long int**. Podobnie, gdy będziemy chcieli korzystać jedynie z małych liczb, wówczas odpowiednie okażą się typy **signed short int** lub **unsigned short int**.

# Modyfikatory volatile, register, const

Oto grupa trzech modyfikatorów, z których tak naprawdę tylko jeden jest często wykorzystywany.

**volatile** - zmienna może się zmieniać bez wiedzy kompilatora
**register** - zależy nam, żeby dostęp do zmiennej był jak najszybszy
**const** - zmienna ma być stała

Podejrzewam, że z krótkich wyjaśnień obok modyfikatorów niewiele rozumiesz. Ja też niewiele bym zrozumiał.

Modyfikator **volatile** mówi kompilatorowi, żeby nie dokonywał żadnych przyspieszeń w dostępie do zmiennej. Tak naprawdę może to być przydatne tylko wtedy, gdy wartość zmiennej jest pobierana z jakiegoś zewnętrznego urządzenia (np. czujnika). Podejrzewam, że długo z tego modyfikatora nie skorzystasz. Ja nie użyłem go do tej pory ani razu.

Z kolei modyfikator **register** sugeruje, aby kompilator umieścił zmienną w rejestrze. Dzięki temu dostęp do takiej zmiennej będzie nieco szybszy niż w normalnej sytuacji. Kompilator nie zawsze jednak weźmie pod uwagę użyty przez nas modyfikator. Z tego modyfikatora tak naprawdę raczej też nie będziesz korzystać, przynajmniej na początku.

Modyfikator **const** opiszę w oddzielnym paragrafie. Na zakończenie dodam, że modyfikatorów volatile, register i const można stosować do wszystkich poznanych do tej pory typów danych (czyli m.in. int, short, unsigned short, char, double, long double, float itd.).

# Modyfikator const

W stosunku do grupy trzech modyfikatorów: volatile, register i const, modyfikator **const** będzie wykorzystywany przez Ciebie najczęściej. Dlatego też postanowiłem omówić ten modyfikator oddzielnie.

Przede wszystkim, czas zdradzić tajemnicę do czego służy modyfikator **const**. Otóż modyfikator ten służy do "powiedzenia" kompilatorowi, że zmienną z tym modyfikatorem traktujemy jako stałą i nie będziemy dokonywać na niej żadnej modyfikacji.

Czy takie podejście tak naprawdę jest konieczne? Przecież kompilator za nas zmiennych nie zmieni, a to my zdecydujemy, czy chcemy zmienić daną zmienną czy nie. Rzeczywiście, jest w tym dużo prawdy. To od nas zależy, czy daną zmienną będziemy zmieniać w trakcie działania programu czy nie.

Dlaczego zatem stosować modyfikator const? Odpowiedź jest prosta - dla własnej wygody. Załóżmy, że w programie będziesz mieć 100 zmiennych. Czy po kilku tysiącach linii programu będziesz nadal pamiętać, której zmiennej nie chcesz nigdy zmieniać? Podejrzewam, że łatwo by Ci było o tym zapomnieć i w ten sposób zmienić przypadkowo wartość jakiejś zmiennej.

Dzięki zastosowaniu modyfikatora **const** wprowadzasz pewne ograniczenie na siebie - mówisz, że nigdy zmiennej zmieniać nie będziesz. Skutkuje to tym, że jeśli dalej w programie usiłujesz zmienić jakąkolwiek zmienną z modyfikatorem const, kompilator zaprotestuje i program się nie skompiluje. W ten sposób udało się sprawić, że nie musisz pamiętać, których zmiennych nie należy zmieniać - jeśli spróbujesz dokonać zmiany na zmiennej z modyfikatorem **const**, kompilator zacznie krzyczeć, że to błąd.

Kluczową sprawą jest to, że jeśli chcesz powiedzieć, że jakiejś zmiennej nie będziesz nigdy zmieniać, musisz od razu określić jaką ma ona wartość. Dlatego też nie jest możliwe pobranie takiej wartości na przykład z klawiatury czy z pliku.

Poza tym, że wartość zmiennej musi zostać określona od razu i że wartości nie będzie można już nigdy zmienić, zmienna z modyfikatorem będzie się zachowywała tak jak każda inna zmienna.

Poniżej przedstawiam kilka fragmentów kodu z zastosowaniem modyfikatora **const**. Dzięki nim wszystko już będzie jasne.

```
const int zmienna; // blad - nie przypisalismy zadnej wartosci
const double pi=3.14; // dobrze - przypisalismy wartosc
const double pi=3.14; // dobrze - przypisalismy wartosc
pi=2; // blad - nie wolno zmieniac wartosci zmiennej z modyfikatorem const
```

# Stałe w programach

Już wiesz, że aby tworzyć stałe w programach, możesz użyć modyfikatora **const**. Mimo to istnieje, jeszcze inna metoda, jednak nie polecam jej stosowania.

Otóż ta metoda polega na wykorzystaniu dyrektywy preprocesora: **define** O ile dobrze pamiętasz, takie dyrektywy zawsze poprzedzamy znakiem **#**. Zawsze musimy taką dyrektywę umieścić tam, gdzie umieszczamy dyrektywę include, czyli na początku programu (to jest wada).

Przykładowa definicja stałej o nazwie **PI** wygląda tak:

#define PI 3.14

Nie będę się rozwodzić nad tą metodą, bowiem jak już napisałem, odradzam Ci jej stosowania. Zauważ tylko, że nie używamy tutaj nigdzie typu zmiennej (inaczej niż w przypadku modyfikatora const). Nazwa zmiennej jest ponadto napisana dużymi literami - ale tylko dlatego, że tak się przyjęło, że stałe definiowane za pomocą dyrektywy define, pisze się dużymi literami (nie jest to obowiązkiem).

## Podsumowanie

W tej lekcji przedstawiłem Ci przede wszystkim modyfkatory, dzięki którym możemy rozbudowywać podstawowe typy danych lub nadawać im charakterystyczne właściwości. Teraz już wiesz w jaki sposób tworzyć zmienne nie przechowujące znaku (unsigned) lub tworzyć stałe zmienne (const).

Dodatkowo, znasz już metodę definiowania stałych w programie za pomocą dyrektywy preprocesora **#define** i wiesz, że znacznie lepiej tworzyć stałe zmienne przy użyciu modyfikatora **const**.

# Lekcja 12: Tablice w języku C++. Podstawowy sposób organizacji danych.

## Tablice - sens wprowadzania kolejnego "typu"

W tym momencie znasz już podstawowe typy danych. Wiesz już jak posługiwać się zmiennymi typów liczbowych, znakowych oraz napisowych. Znasz również modyfikatory, które umożliwiają Ci mieć większy wpływ na podstawowe typy danych.

Aby wyjaśnić Ci do czego służą tablice, musisz sobie najpierw przypomnieć wcześniejsze programy. W niektórych z nich chcieliśmy, aby użytkownik podał jedną liczbę za pomocą klawiatury, w innych chcieliśmy, żeby podał 2 liczby za pomocą klawiatury. Czy sprawiało nam to jakąś trudność? Nie, było to bardzo proste i łatwe.

Wyobraź sobie jednak, co by było, gdybyśmy chcieli, aby użytkownik z jakiegoś powodu musiał podać 100 lub nawet 1000 liczb. Co prawda nikt normalny aż tylu danych za pomocą klawiatury by podać nie chciał, ale my przyjmiemy, że tak właśnie jest, bowiem nie znasz na razie innych sposobów pobierania danych (z pliku).

Czy w przypadku tak dużej liczby danych byłoby Ci trudniej napisać program? Podejrzewam, że nie. Wszystko działoby się dokładnie w ten sam sposób. Problemem byłoby jednak to, jak nazywać poszczególne zmienne. Gdyby zmienne były nazywane a, b, c itd. wkrótce zabrakłoby Ci liter alfabetu. Poza tym czy w dalszej części programu łatwo byłoby Ci odgadnąć która z danych jest np. daną, która została podana jako dwudziesta?

Sprytniejsza osoba by sobie pewnie pomyślała tak - skoro mamy dużą liczbę danych, możemy je sobie nazywać na przykład tak: zmienna1, zmienna2, zmienna3, zmienna4. I rzeczywiście jest to jakieś rozwiązanie. Dzięki temu nawet w znacznie bardziej odległej części programu wiadomo by było, że np. dana pobrana jako dwudziesta z klawiatury to zmienna20.

Mimo, że sam pomysł jest dość dobry, ma on jednak duże wady. Przede wszystkim każda taka dana zostaje traktowana jako wartość pewnej zmiennej. Dane te nie są ze sobą powiązane w żaden sposób. Poza naszą wygodą nie zmienia się tak naprawdę nic - jeśli będziemy musieli pobrać bardzo dużo danych z klawiatury będziemy musieli niestety postąpić tak samo jak wtedy, gdy zmienne nazywały się a, b, c itd.

Tablice są w pewnym sensie rozwinięciem ostatniego pomysłu. Umożliwiają wygodniejsze nazewnictwo danych, a oprócz tego mają dodatkowe cechy, których nie udałoby się nam osiągnąć zwykłymi metodami. Czas zatem dowiedzieć się o tablicach nieco więcej.

## Tablice - czym są i jak się ich używa

Tablice są metodą organizacji danych tego samego typu.

Co dokładnie oznacza to stwierdzenie? Oznacza ono, że w przypadku każdej tablicy musimy się zdecydować, jakiego typu będziemy przechowywać w niej dane. Musimy jasno określić, czy będziemy przechowywać w takiej tablicy liczby np. typu unsigned int lub double czy znaki, czy może napisy.

Jeśli pomyślisz, że to duże ograniczenie, jesteś w błędzie. Przede wszystkim czy rozsądnie jest pośród grupy liczb przechowywać jakąś literę? Raczej nie - nie dość, że należałoby o tym pamiętać, to tak naprawdę takie działanie mija się z celem.

Tablicę możesz sobie wyobrazić jako karton, w którym znajdują się pudełka - wszystkie takie same, wszystkie o takich samych wymiarach, a w każdym z tych pudełek znajduje się konkretna dana - w jednym na przykład znajduje się liczba 23, w drugim liczba 15. Wszystkie pudełka są jednak takie same (czyli wszystkie zmienne w tablicy są jednego typu), bowiem inaczej nie udałoby nam się umieścić pudełek w kartonie.

Teraz najważniejsza część - w jaki sposób deklarujemy tablice. Robimy to tak:

nazwa_typu nazwa_tablicy[rozmiar];

przy czym:

**nazwa_typu** - nazwa przechowywanego typu danych np. double, int, char.
**nazwa_tablicy** - nazwa tablicy - zasady takie same jak przy nazwach zmiennych
**rozmiar** - ilość elementów, które chcemy przechowywać w tablicy

Jak więc widzisz utworzenie tablicy nie jest takie trudne. Bardzo charakterystyczną cechą jest to, że rozmiar tablicy (poza jednym wyjątkiem, o którym wspomnę później) musi zostać określony w momencie utworzenia tablicy.

Oto kilka przykładowych deklaracji tablic:

```
int calkowita[20];   // tablica o nazwie calkowita - przechowuje 20 liczb typu int
char znaki[5];       // tablica o nazwie znaki - przechowuje 5 znaków
double liczby[1000]; // tablica o nazwie liczby - przechowuje 1000 liczb typu double
string napisy[5];    // tablica o nazwie napisy - przechowuje 5 napisów
```

Zauważ, że dla wszystkich tablic, rozmiar mamy określony za pomocą liczby. Rozmiar tablicy można również określić za pomocą zmiennej z modyfikatorem **const**. Nie jest jednak możliwe pobranie rozmiaru tablicy od użytkownika (np. za pomocą klawiatury), a dopiero później utworzenie tablicy o rozmiarze, który przed momentem został pobrany, bowiem rozmiar tablicy musi być znany w momencie kompilacji programu.

# Jak używać tablic

Czas teraz dowiedzieć się, w jaki sposób odwołać się do poszczególnych danych. Załóżmy, że mamy taką taką tablicę:

int calkowita[7];

Mamy zatem tablicę siedmiu liczb całkowitych typu int (jak już wiemy int to to samo co signed int). Tablicę nazwaliśmy **calkowita**.

W jaki sposób odwołać się do poszczególnych zmiennych? Wbrew pozorom nie jest to takie trudne. Do pierwszej zmiennej odwołujemy się pisząc **calkowita[0]**, do drugiej **calkowita[1]**, a do ostatniej (czyli siódmej) odwołamy się pisząc **calkowita[6]**.

Schematycznie, aby odwołać się do danego elementu tablicy piszemy:

nazwa_tablicy[numer_elementu]

Poniższe stwierdzenie jest jednym z tych, które najlepiej jeśli przepiszesz na dużą kartkę dużymi

czerwonymi literami i powiesisz obok swojego stanowiska pracy:

**W języku C++ tablice indeksujemy od 0**

Zauważ - pierwszy element w naszym przykładzie to nie jest calkowita[1], tylko całkowita[0]. Ostatni element tablicy to całkowita[6] a nie całkowita[7]. Zatem ostatni element ma indeks o jeden mniejszy niż liczba elementów w tablicy. U nas tablica miała 7 elementów, a ostatni, czyli siódmy element miał indeks 6.

To, że tablice są indeksowane od zera jest bardzo ważną sprawą. Wszyscy początkujący programiści popełniają w tym miejscu błąd, więc Tobie radzę sobie to dobrze zapamiętać - już tutaj uzyskasz przewagę nad innymi.

Poza tym, o czym już wspomniałem, na poszczególnych elementach tablic operujemy jak na zwykłych zmiennych. Nie ma tutaj żadnych nowości. Nowością pozostaje jedynie nieco inny zapis odwołania do zmiennej. Poza tym wszystko przebiega według dotychczas poznanych zasad.

# Lekcja 12: Tablice w języku C++. Podstawowy sposób organizacji danych.

Oto przykładowy program z wykorzystaniem tablicy:

```
#include <iostream>

using namespace std;

int main()
{
  double calkowita[6]; // tablica 6 liczb typu double

  cout <<"Podaj pierwsza liczbe: ";
  cin >>calkowita[0]; // pobieramy pierwszy element do tablicy
  cin.ignore();
  cout <<"Podaj druga liczbe (rozna od zera): ";
  cin >>calkowita[1]; // pobieramy drugi element do tablicy
  cin.ignore();

  //do pozostalych elementow tablicy przypiszemy wyniki dzialania na elementach


calkowita[2]=calkowita[0]+calkowita[1];
  calkowita[3]=calkowita[0]-calkowita[1];
calkowita[4]=calkowita[0]*calkowita[1];

calkowita[5]=calkowita[0]/calkowita[1];
  //calkowita[6]=calkowita[0]+calkowita[1];  blad - calkowita[6] nie istnieje!!!

  cout <<"Podane liczby to: "<<calkowita[0]<<' '<<calkowita[1]<<'\n'
       <<"Ich suma wynosi: "<<calkowita[2]<<'\n'
       <<"Ich roznica wynosi: "<<calkowita[3]<<'\n'
       <<"Ich iloczyn wynosi: "<<calkowita[4]<<'\n'
       <<"Ich iloraz wynosi: "<<calkowita[5]<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Jak więc widzisz, rzeczywiście na poszczególnych elementach tablicy możemy wykonywać dowolne operacje i posługujemy się nimi jak zwykłymi zmiennymi.

Przy okazji zwróć uwagę, że żądamy, aby użytkownik podał drugą liczbę taką, żeby nie była ona zerem. Czy wiesz czemu? Tak naprawdę nie ma to żadnego związku z tablicami. Chodzi jedynie o to, że później w programie dokonujemy dzielenia. Gdyby użytkownik podał zero, operacja byłaby niezdefiniowana, bo jak zapewne pamiętasz z lekcji matematyki, przez zero się nie dzieli.

W prawdziwym programie, raczej nie należy zakładać, że użytkownik nas posłucha. Do sprawdzenia, czy użytkownik nie podał czasem wartości zero, należałoby wykorzystać instrukcję warunkową if - jestem pewien, że wiesz jak to zrobić, jeśli udało Ci się przeczytać lekcję zatytułowaną Instrukcja warunkowa if.

# Dodatkowe informacje o tablicach

Wartości poszczególnych elementów tablicy możemy albo pobierać tak jak do tej pory, albo możemy przypisać wartość tych elementów w momencie powstania tablicy (jest to tak zwana inicjalizacja - dokładniej omówię to zagadnienie w jednej z następnych lekcji).

Aby przypisać wartości poszczególnym elementom tablicy należy umieścić te wartości w nawiasie klamrowym.

Jeśli wartości takich będzie dokładnie tyle ile elementów, wówczas każdy element będzie miał określoną wartość. Taki przypadek ilustruje poniższy przykład:

```
#include <iostream>

using namespace std;

int main()
{
  int calkowite[4]={5,67,2, -3}; // kazdy element ma okreslona wartosc
  cout <<calkowite[0]<<' '<<calkowite[1]<<' '<<calkowite[2]<<' '
       <<calkowite[3]<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Jeśli wartości będzie mniej niż rozmiar tablicy, wówczas elementy, dla których "zabraknie" wartości, będą miały przypisaną wartość zerową (w przypadku typów liczbowych). Weź jednak pod uwagę, że w każdym innym wypadku (tzn. w przypadku braku inicjalizacji) wartości elementów tablicy będą przypadkowe. Te sytuacje ilustruje przykład:

```
#include <iostream>

using namespace std;

int main()
{
  int mniej[4]={5,67,2}; // ostatni element nie ma okreslonej wartosci
  cout <<mniej[0]<<' '<<mniej[1]<<' '<<mniej[2]<<' '
       <<mniej[3]<<'\n';

  int przypadkowa[4];
  /* Nie przypisalismy wartosci. Do momentu pobrania wartosci z klawiatury lub
```

```
pliku lub innych operacji na tablicy, wartosci elementow sa PRZYPADKOWE!!!*/
  cout <<'\n'<<przypadkowa[0]<<' '<<przypadkowa[1]
       <<' '<<przypadkowa[2]<<' '<<przypadkowa[3]<<'\n';

  int zerowa[4]={};
  /*Nie przypisalismy zadnej wartosci, ale uzylismy nawiasow tak jakbysmy
chcieli tego dokonac. Wszystkie elementy beda mialy wartosc zerowa*/
  cout <<'\n'<<zerowa[0]<<' '<<zerowa[1]
       <<' '<<zerowa[2]<<' '<<zerowa[3]<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

# Lekcja 12: Tablice w języku C++. Podstawowy sposób organizacji danych.

Jeśli natomiast w nawiasie klamrowym umieścimy więcej wartości niż to wynika z rozmiaru tablicy, kompilator zasygnalizuje błąd. Taka sygnalizacja błędu w przypadku tablic występuje tylko w tym wypadku. W każdym innym przypadku, kiedy utworzymy tablicę o określonym rozmiarze i będziemy się odwoływać do nieistniejącego elementu, żaden błąd nie zostanie zasygnalizowany, jednak takie działania mogą być katastrofalne w skutkach. Oto przykład:

```
#include <iostream>

using namespace std;

int main()
{
//   int calkowite[4]={5,67,2,-3,7};
/*
Powyzsza linia spowodowalaby blad - mamy o jedna wartosc za duzo w nawiasie.
Kompilator ZAPROTESTUJE
*/

  int tablica[4];
//   cout <<tablica[4];
/*
To jest tez BLAD - element tablica[4] nie istnieje. Kompilator jednak NIE
zaprotestuje. TRZEBA SAMEMU UWAZAC!!!
*/

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Ponadto warto dodać, że mimo że rozmiar tablicy musi zostać jawnie podany, to w przypadku gdy podajemy wartości elementów tablicy, możemy pominąć rozmiar tablicy. W takim wypadku kompilator jest w stanie sam wyliczyć rozmiar tablicy na podstawie listy wartości elementów. Ważne jednak, aby nie zapomnieć o nawiasach kwadratowych. Inaczej program się nie skompiluje. Oto przykładowy program:

```
#include <iostream>

using namespace std;

int main()
{
  int calkowite[]={5,67,2, -3}; // 4 wartosci - czyli chcemy tablice o 4
elementach
//   int tablica={5,67,2, -3}; // BLAD - brak nawiasow sugerujacych ze to
tablica

  cout <<calkowite[0]<<' '<<calkowite[1]<<' '<<calkowite[2]<<' '
       <<calkowite[3]<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

## Ostrożnie z tablicami

Mimo, że tablice są bardzo przydatną metodą organizacji danych, to jednak trzeba być w posługiwaniu się nimi bardzo ostrożnym.

Wiąże się to z tym, że w C++ (i również w języku C) nie ma kontroli poprawności odwołania do tablicy. Możemy utworzyć tablicę 2 elementową i odwoływać się do jej 100-ego elementu, mimo że taki element nie istnieje.

Tego typu błędy mogą być trudne do wykrycia, tym bardziej, że gdy spróbujemy zapisać coś do tego 100-ego elementu tablicy dwuelementowej, możemy nadpisać wartość jakiejś innej zmiennej i będziemy szukać błędu nie w tym miejscu co należy.

Jeśli uważasz, że to wada języka, musisz wiedzieć, że tak naprawdę jest to zaleta. To programista musi się zatroszczyć o odpowiednie odwołania. W zamian uzyskujemy bardzo szybkie odwołania do tablicy, co jest niewątpliwie dużą zaletą.

## Cechy tablic w C++

Oto najważniejsze cechy tablicy w języku C++:

- może przechowywać elementy tylko jednego typu
- musi mieć określony stały rozmiar (lub rozmiar ten zostaje wyliczony przez kompilator, gdy określimy wartości poszczególnych elementów tablicy podczas inicjalizacji)
- jest indeksowana od zera (pierwszy element ma indeks zerowy)
- umożliwia szybki dostęp do dowolnego elementu tablicy
- nie jest sprawdzana poprawność odwołań do elementów - to na programiście spoczywa odpowiedzialność za nieprzekroczenie dopuszczalnego zakresu tablicy
- umożliwia łatwiejsze operowanie na danych, łatwiejsze ich wypisywanie i pobieranie

O ile wszystkie z tych przedstawionych punktów powinny być oczywiste, ostatni taki nie jest. O tym, jak łatwo można przeprowadzać operacje na tablicach, przekonasz się gdy poznasz pętle.

## Podsumowanie

W tej lekcji przedstawiłem Ci tablice - najprostszy sposób organizacji danych w języku C++. Jest to bardzo ważna i przydatna lekcja, więc jeśli czegoś nie udało Ci się zrozumieć, zachęcam do ponownego przeczytania.

O tym, jak duże możliwości dają tablice, przekonasz się dopiero, kiedy poznasz pętle i dowiesz się w jaki sposób połączyć znajomość pętli z tablicami.

# Lekcja 13: Pętle w języku C++ - pętla for

## Pętle - sens istnienia

Pętle umożliwiają powtórzenie pewnych instrukcji dopóki nie zostanie spełniony warunek kończący pętlę. Dzięki temu możemy w bardzo łatwy i krótki sposób wypisać na przykład ten sam (lub podobny) komunikat kilka lub kilkaset razy lub pobrać od użytkownika na przykład 100 zmiennych. Nie będzie to stanowiło już większego problemu i dzięki temu, korzystanie z języka C++ stanie się wygodniejsze i przyniesie nowe możliwości.

# Pętla for - podstawy

Pętla for podobnie jak wszystkie pozostałe pętle umożliwi nam powtórzenie określonych operacji tak długo jak warunek końcowy jest spełniony. **Schematycznie** pętlę for można zapisać:

for (stanyPoczatkowe; warunekKoncowy; zmiany)
  lista_instrukcji

Schematyczna forma pętli for, jak się zapewne domyślasz, sugeruje, że zarówno stanów początkowych jak i zmian może być kilka lub nawet kilkanaście. Poszczególne elementy oddzielimy wtedy przecinkami.

Oto prosty program przedstawiający wykorzystanie pętli for:

```
#include <iostream>

using namespace std;

int main()
{
  int licznik;
  for (licznik=1;licznik<10;++licznik)
     cout <<"Wykonuje petle po raz "<<licznik<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

W przykładzie tym mamy tylko jeden stan początkowy, jeden warunek końcowy i tylko jedną zmianę. Lista instrukcji (czyli w tym wypadku wypisanie tekstu na ekran) wykonuje się tak długo jak warunek (czyli stan końcowy) jest prawdziwy. Kiedy warunek staje się fałszywy, wówczas pętla zostaje zakończona. Przy każdym wykonaniu pętli zmienna licznik (jest to tak zwana zmienna sterująca pętlą) zostaje zmieniona tak jak to zostało zapisane w liście zmiana - czyli w naszym przypadku o 1.

Przede wszystkim zauważ, że w tym programie wykonujemy tylko jedną instrukcję w pętli. Gdybyśmy chcieli wykonać więcej instrukcji, musimy je otoczyć nawiasami klamrowymi.

Przy okazji zwróć uwagę, że w linii, w której została zapisana lista stanów początkowych, warunków końcowych i zmian, nie ma średnika. Jeśli postawisz tam średnik, kompilator nie poinformuje Cię o błędzie, ale program nie wykona się tak, jak tego oczekujesz.

Zwróć również uwagę na zapisaną zmianę: **++licznik**. Wiesz oczywiście, że znaczy to to samo, co **licznik+=1**. W większości książek dotyczących programowania spotkasz jednak w tym miejscu zapis **licznik++**. Jak zapewne pamiętasz operator ++ można stosować zarówno przed jak i po zmiennej. Jak powinno być w tym wypadku? Otóż w tym wypadku nie ma to najmniejszego znaczenia z punktu widzenia logiki działania. Jednak operator ++ użyty przed zmienną działa szybciej niż użyty po zmiennej, dlatego też radzę Ci używać w tym wypadku zapisu przed zmienną.

# Uproszczona postać pętli for

Musisz wiedzieć, że w języku C++ możemy zrezygnować z dowolnej części pętli for, a nawet z ich wszystkich. Możemy pominąć zarówno stan początkowy, warunek końcowy, jak i regułę zmian podczas działania pętli. Pomijając dowolną regułę, średniki muszą jednak pozostać, aby można było odróżnić, która z części została pominięta - zatem zawsze pozostaną dwa średniki.

Pomijając jednak niektóre z reguł sprawiasz, że może się zdarzyć, że program się "zapętli" - to znaczy, że nasza pętla będzie zawsze prawdziwa. W normalnym programie oczywiście nie możesz

sobie na to pozwolić, jednak na razie warto o tym wiedzieć i zdawać sobie sprawę.

Poniższe programy przedstawiają jak można pomijać poszczególne części pętli for. Jeśli program nic nie robi, albo wypisuje to samo (czyli się zapętlił), przerwiesz jego działanie naciskając **ctrl + Break** lub **ctrl + C** w systemie MS Windows.

W poniższym programie pominęliśmy każdą z części - pętla wykonuje się w nieskończoność. Użytkownik musi nacisnąć kombinację klawiszy **ctrl + Break** lub **ctrl + C** aby przerwać działanie programu:

```
#include <iostream>

using namespace std;

int main()
{

  for (;;)
  {
     cout <<"Petla nieskonczona\n";
     cout <<"Nacisnij ctrl + c lub ctrl + pause\n";
  }

  return 0;
}
```

Z kolei ten program przedstawia sytuację bardzo często spotykaną - pomijana jest lista stanów początkowych, bowiem takie stany ustaliliśmy już przed pętlą. Zauważ ponadto, że pętla wykonuje się 11 razy, a w środku pętli wypisujemy zmienną pętli zwiększoną o 1. Dlaczego? Z prostego powodu - pętlę zaczęliśmy od 0 - jednak trochę dziwnie byłoby wypisać, że pętla wykonuje się "zerowy raz", prawda? Zwróć jednak uwagę, że wypisanie **i+1** nie zmienia wartości i (**i+1** to nie to samo co **i=i+1**).

```
#include <iostream>

using namespace std;

int main()
{

  unsigned int i=0;
  for (;i<=10;++i)
  {
     cout <<"Wykonuje sie ";
     cout <<i+1<<" raz\n";
  }

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Oczywiście może zostać pominięta również część dotycząca zmian podczas wykonywania pętli, jednak ten przypadek omówię oddzielnie. Jak widzisz istnieje dość sporo sposobów wykorzystania części definiujących zachowanie pętli for - z jednych sposobów korzystamy częściej, a z innych rzadziej, jednak uwierz mi, że każdy z nich jest przydatny.

Przy okazji zwróć uwagę na to, że trzeba być bardzo ostrożnym podczas pisania pętli - bardzo mały

błąd i już zapętlimy program - będzie się on wykonywał w nieskończoność, a zazwyczaj nie jest to naszym celem.

# Zmienna sterująca pętlą

Musisz wiedzieć, że zmienną, która występuje w stanach początkowych lub w warunku końcowym nazywamy zmienną sterującą pętlą - bowiem od stanu tej zmiennej zależy, jak długo pętla będzie się wykonywać. Jeśli mamy kilka stanów początkowych, to tak naprawdę zmiennych sterujących pętlą może być kilka, jednak do tej pory w przykładach stosowaliśmy zawsze jedną zmienną sterującą.

Jedną z zalet języka C++ jest to, że zmienną sterującą pętli nie musimy koniecznie zmieniać w liście zmian, możemy tak samo to wykonać jako zwykłą instrukcję. Przykładowo, pierwszy program znajdujący się w tej lekcji możemy zapisać następująco:

```
#include <iostream>

using namespace std;

int main()
{
  int licznik;
  for (licznik=1;licznik<10;)
  {
     cout <<"Wykonuje petle po raz "<<licznik<<'\n';
     ++licznik;
  }
  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Dodatkowo, możemy połączyć dwa sposoby - zmienną sterującą pętlą możemy zwiększać zarówno na liście zmian, jak i wewnątrz samej pętli. W ten sposób możemy uzyskać bardzo ciekawe efekty. W zależności od pewnych zdarzeń, możemy pomijać pewne zakresy zmiennej sterującej. W nieco bardziej skomplikowanych programach, taka cecha jest bardzo przydatna. Oto 2 przykładowe pętle:

```
int licznik;
  for (licznik=1;licznik<10;++licznik)
  {
     cout <<"Wykonuje petle po raz "<<licznik<<'\n';
     ++licznik;
  }
```

Za pomocą powyższej pętli zwiększamy tak naprawdę zmienną sterującą pętli o 2 w każdym kroku. Raz wykonujemy to na liście zmian, a raz jako zwykłą instrukcję.

```
  int licznik;
  for (licznik=1;licznik<10;++licznik)
  {
     cout <<"Wartosc zmiennej licznik wynosi "<<licznik<<'\n';
     if (licznik==2 || licznik==5)
        licznik+=2;
  }
```

Z kolei w powyższej pętli zmienna sterująca jest zawsze zwiększana o 1. Jeśli jednak ta zmienna

wynosi 2 lub 5, jest wówczas zwiększana dodatkowo o 2.

# Lekcja 13: Pętle w języku C++ - pętla for

Przy okazji warto zwrócić uwagę na to, gdzie deklarujemy zmienną sterującą pętlą. Zmienną sterującą pętlą możemy deklarować tak jak każdą inną zwykłą zmienną w programie, tzn. na początku funkcji main lub przed samą pętlą.

Możemy to jednak również zrobić bezpośrednio w pętli. Przy okazji, chcę zwrócić uwagę, że nie chodzi mi o to, gdzie zmiennej przypiszemy początkową wartość (zainicjalizujemy), tylko o to, gdzie napiszemy, że zmienna jest danego typu.

Oba podejścia mają swoje zalety i wady. Zacznijmy od przyjrzenia się obu metodom:

Metoda 1:

```
  int licznik;
  for (licznik=1;licznik<10;++licznik)
  {
     cout <<"Wykonuje petle po raz "<<licznik<<'\n';
  }
  cout <<"Wartosc licznika po wykonaniu petli to: "<<licznik;
Metoda 2:
  for (int licznik=1;licznik<10;++licznik)
  {
     cout <<"Wykonuje petle po raz "<<licznik<<'\n';
  }
  /*
     Tutaj zmienna licznik juz nie istnieje. Gdyby instrukcja wypisania
     nie byla w komentarzu, kompilator zasygnalizuje blad
  */
//   cout <<"Wartosc licznika po wykonaniu petli to: "<<licznik;
```

**Druga metoda** jest tak naprawdę znacznie częściej stosowana przez początkujących programistów. Zmienna zostaje utworzona tylko na czas działania pętli i wartość zmiennej znana jest tylko wewnątrz pętli. Natomiast już po zakończeniu pętli, wartości uzyskać nie można, bowiem po zakończeniu pętli, taka zmienna już nie istnieje.

Mimo tak oczywistych wad, metoda ma jedną zaletę - stosując tą metodę, tuż po zakończeniu pętli wartość zmiennej i sama zmienna nie są znane - dzięki temu zapobiegamy zaśmiecaniu programu zmiennymi, które są używane tylko w pętlach.

**Pierwsza metoda** natomiast zostanie wykorzystana przez bardziej świadomego i wymagającego programistę. Przede wszystkim, w wielu programach, aby sprytnie wykonać pewną czynność, często przydaje się znać wartość zmiennej tuż po zakończeniu pętli.

Na dodatek warto zdawać sobie sprawę, że jeśli w programie umieścimy 100 pętli, możemy za każdym razem skorzystać z tej samej zmiennej. Dzięki temu zmienna jest tworzona tylko raz, natomiast w metodzie drugiej zmienna będzie tworzona za każdym razem. Mimo, że dla Ciebie nie jest to widoczne, z pewnością zadziała to szybciej. Z drugiej jednak strony, w programie możemy uzyskać niepotrzebne zmienne, które będą nam tylko "przeszkadzały".

To której metody będziesz używać, zależy tylko i wyłącznie od Ciebie. Ja w przypadkach większych programów, staram się stosować metodę pierwszą, bowiem zadziała ona szybciej. Niekiedy jednak, zwłaszcza, gdy w programie nie mamy aż tak wielu pętli i potrzeby znania wartości zmiennej po zakończeniu pętli, można skorzystać z metody drugiej.

# Dwie zmienne sterujące pętlą

Jak już wspomniałem, pętlą mogą sterować dwie (lub więcej) zmiennych. Wówczas w części stanów początkowych oraz zmian, zapisujemy instrukcje dotyczące poszczególnych zmiennych rozdzielając je przecinkiem.

Warto jednak zwrócić uwagę, że warunek końcowy pozostaje tylko jeden. Tylko w taki sposób pętla zadziała. Jeśli pomyślisz, że w warunku końcowym można rozdzielać warunki dotyczące poszczególnych zmiennych za pomocą przecinka, zobaczysz, że program nie działa zgodnie z oczekiwaniami (chyba, że wybierzesz szczególny przypadek).

Dlatego zapamiętaj sobie dobrze - **w pętli for mamy tylko jeden warunek końcowy**. Warunek końcowy może być jednak rozbudowany za pomocą operatorów logicznych, na przykład tak jak w poniższym programie:

```
#include <iostream>

using namespace std;

int main()
{
  int i,j;
  for (i=1,j=5;i<6 && j<=10;++i,j+=2)
     cout <<"i wynosi "<<i<<" a j wynosi "<<j<<'\n';

  cout <<"Po zakonczeniu petli i wynosi "<<i<<" a j wynosi "<<j<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

W powyższym przykładzie, przypisujemy początkowe wartości zmiennym sterującym. W każdym kroku zmieniamy też nasze zmienne sterujące: zmienną i zwiększamy o 1, a zmienną j zwiększamy o 2. Warunek jest tylko jeden, ale jest to warunek złożony - uwzględnia wartości obu zmiennych sterujących.

# Typ bez znaku - mała pułapka

Jeśli dokładnie śledzisz wszystkie przykłady, które pojawiły się do tej pory, to na pewno udało Ci się zauważyć pewną charakterystyczną cechę. Mianowicie, po zakończeniu pętli, zmienna sterująca ma wartość większą niż to zostało określone w warunku końcowym. Wynika to z tego, że aby pętla została przerwana, warunek końcowy musi być niespełniony. Zatem wykonywana jest dodatkowo jeszcze jedna sekcja **zmiana**. Wówczas okazuje się, że warunek już nie jest prawdziwy i pętla zostaje zakończona.

Co prawda na razie nie będziemy korzystać z wartości zmiennej po zakończeniu pętli, jednak mimo to, uda się w ten sposób wyjaśnić pewien problem (tajemniczy dla niektórych początkujących).

# Lekcja 13: Pętle w języku C++ - pętla for

Załóżmy, że chcemy wypisać liczby od 0 do 10. Możemy to zrobić tak:

```
int i;
  for (i=0;i<=10;++i)
     cout <<i<<' ';
```

Dla formalności dodam, że ktoś oczywiście mógł wpaść na pomysł, żeby zrobić to w następujący sposób:

```
  int i;
  for (i=1;i<=11;++i)
     cout <<i-1<<' ';
```

Nam jednak będzie chodziło o pierwszy sposób - wypisywana liczba ma być równa wartości zmiennej sterującej pętlą.
Teraz dla odmiany załóżmy, że chcemy wypisać te same liczby, ale tym razem od 10 **do** 0. Oczywiście nie powinno to być żadną trudnością:

```
  int i;
  for (i=10;i>=0;--i)
     cout <<i<<' ';
```

Jeśli przypomnisz sobie lekcję o modyfikatorach, to zauważysz, że w obu występujących tutaj przykładach, można by było zastosować modyfikator **unsigned**. W obu bowiem przypadkach interesują nas liczby powyżej 0 i równe liczbie 0, więc nie warto umożliwić przechowywania liczb ujemnych, bowiem możemy chcieć tą przestrzeń wykorzystać do przechowywania większych liczb całkowitych dodatnich.

Bardzo często właśnie przy pętlach wykorzystuje się modyfikator **unsigned**. Prawie każdy tak robi, więc dlaczego Ty nie? Otóż zastosowanie modyfikatora w jednym z poniższych przykładów sprawi problem.

Tutaj zastosowaliśmy modyfikator **unsigned**, gdy wypisujemy liczby w sposób rosnący:

```
unsigned int i;
for (i=0;i<=10;++i)
   cout <<i<<' ';
```

Ten przypadek nie sprawia żadnych problemów. Modyfikator **unsigned** możemy stosować bez obaw, kiedy zmienna sterująca jest zwiększana.

Poniżej natomiast stosujemy modyfikator **unsigned**, gdy chcemy wypisać liczby w sposób malejący:

```
unsigned int i;
for (i=10;i>=0;--i)
   cout <<i<<' ';
```

Jeśli uruchomisz powyższy kod, pamiętaj, że aby przerwać działanie pętli należy użyć **ctrl + Break** lub **ctrl + C** w systemie MS Windows.

Jeśli zastanawiasz się dlaczego tak jest, że ten program się zapętla, weź pod uwagę 2 rzeczy. Przede wszystkim modyfikator **unsigned** sprawia, że można w danej zmiennej przechowywać liczby nieujemne, czyli najmniejszą liczbą, którą można przechowywać jest 0.

Z drugiej strony, przypomnij sobie stwierdzenie, że zmienna po zakończeniu pętli ma wartość mniejszą niż to wynika z warunku. W przykładzie mamy warunek: **i>=0** zatem za ostatnią poprawną wartość uważamy 0. Jednak ponieważ zmienna przyjmuje zawsze wartość mniejszą niż to wynika z warunku, będzie się starała przyjąć wartość mniejszą od zera, a konkretnie -1 (bo mamy

**--i** w sekcji zmiany).

Tutaj pojawia się właśnie problem - do liczby z modyfikatorem **unsigned** usiłujemy przypisać liczbę -1, czyli liczbę ujemną. W efekcie liczbie zostaje przypisana największa z możliwych liczb całkowitych (czyli bardzo duża liczba) i znowu wartość zmiennej i będzie się zmniejszać aż do 0. W rezultacie, pętla nigdy nie zostanie zakończona.

W tym wypadku wychwycenie błędu było bardzo łatwe, jednak rzadko kiedy wypisujemy w pętli wartość zmiennej. Zazwyczaj wykonujemy inne operacje i wykrycie takiego błędu, zwłaszcza przez początkującego programistę, może nie być takie łatwe.

# Pętle zagnieżdżone

Jak zapewne udało Ci się zauważyć, we wszystkich przykładach dotyczących pętli for, dokonywaliśmy na razie jedynie prostych operacji. Operacje te mogą być jednak znacznie bardziej skomplikowane. W szczególności, instrukcją wykonywaną w pętli for, może być kolejna pętla for.

Jednak jeśli stosujemy pętle zagnieżdżone, w każdej pętli zmienna sterująca powinna być inna. W przeciwnym wypadku uzyskamy wyniki, które mogą być nieoczekiwane. Użycie takiej samej nazwy zmiennej sterującej w obu pętlach jest dość częstym błędem popełnianym przez początkujących programistów.

Przykładowe zagnieżdżone pętle for znajdują się w poniższym programie - od razu zauważ, że w pierwszej pętli zmienna sterująca to **i**, natomiast w drugiej pętli zmienną sterującą jest zmienna **j**.

Postaraj się przeanalizować i zrozumieć poniższy program na własną rękę - w przyszłości będziesz rozwiązywać problemy samemu, więc na razie postaraj się chociaż zrozumieć jak dokładnie działa program.

```
#include <iostream>

using namespace std;

int main()
{
  int ktory=1, i, j;

  for (i=1;i<=10; i+=2)
  {
     cout <<"Tak bedzie dla i="<<i<<"\n";
     for (j=5;j>=2;--j)
     {
        cout <<ktory<<". raz: i*j="<<i*j<<'\n';
        ++ktory;
     }
     cout <<"Tak bylo dla i="<<i<<'\n';
  }

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Jeśli nie udało Ci się zrozumieć jak działa powyższy program lub nie wiesz na pewno, że dobrze myślisz, możesz kontynuować czytanie tej strony.

Przede wszystkim zmienna **ktory** służy nam do określenia, ile razy dokonaliśmy wypisania iloczynu zmiennych i oraz j. Dlatego też odpowiednio ją zwiększamy w wewnętrznej pętli (zauważ wewnętrznej a nie zewnętrznej). Zmiennej tej mogłoby nie być, jednak używamy jej jako zmiennej niosącej dodatkowe informacje.

Pierwsza pętla przebiega wartości 1,3,5,7,9, druga natomiast 5,4,3,2. Zauważ też jeszcze raz to, o czym wspominałem już wcześniej: zmienne zastosowane w obu pętlach są inne: zmienne **i** oraz **j**.

Działanie odbywa się w następujący sposób: Zmiennej i przypisujemy 1 (inicjalizacja w pętli for) i sprawdzamy warunek pętli (i<=10). Ponieważ warunek jest prawdziwy, wykonujemy listę instrukcji. Poza zwykłymi wypisaniami na ekran, wykonujemy drugą pętlę for (tę ze zmienną j).

W tej pętli j przebiega wartości od 5 do 2. Za ostatnim razem, gdy zmienna **j** osiągnie wartość 1, pętla ta nie zostanie już powtórzona.

Teraz czas zmienić zmienną i (tę od zewnętrznej pętli), czyli i+=2. Zmienna **i** ma teraz wartość 3. Dalsze działanie powinno już być dla Ciebie oczywiste, bowiem dzieje się w analogiczny sposób.

# Lekcja 13: Pętle w języku C++ - pętla for

Ostatni przykład mógł wydać Ci się dosyć trudny, jednak tego typu zagnieżdżenia pętli są codziennością w pisaniu jakichkolwiek programów. Takich pętli zagnieżdżonych może być dużo więcej - poprzednio były tylko dwie, jednak zazwyczaj w programach używa się właśnie 2 lub 3 zagnieżdżonych pętli.

Mimo że jak już wspomniałem, w zagnieżdżonych pętlach powinny być używane różne zmienne, nie jest zabronione używanie zmiennej z pętli zewnętrznej. Czasami nawet jej wartość może się przydać z takich czy innych powodów.

W poniższym przykładzie wypisujemy iloczyny liczb od 1 aż do liczby, którą się obecnie "zajmujemy" (symbolizuje ją zmienna **i**). Zauważ, że zastosowaliśmy modyfikator **unsigned** (bowiem zwiększamy zmienną w kierunku rosnącym i nie musimy obawiać się wartości -1) oraz, że obie zmienne sterujące zostały utworzone bezpośrednio w pętlach i nie są znane po zakończeniu pętli (bo nie zależy nam na tym, żeby były znane).

```
#include <iostream>

using namespace std;

int main()
{
  for (unsigned int i=1;i<=10;i+=1)
  {
     for (unsigned int j=1;j<=i;j++)
        cout <<i*j<<' ';
     cout <<'\n';
  }
  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Jak widać w powyższym programie, w pętli wewnętrznej korzystamy ze zmiennej z pętli zewnętrznej (warunek j<=i). Tak więc to, ile razy powtórzymy pętlę wewnętrzną, zależy w ogromnej mierze od zmiennej i z pętli zewnętrznej. Z resztą wystarczy spojrzeć na wynik działania programu i przez chwilę przeanalizować kod źródłowy.

Muszę Ci również powiedzieć, że takie zastosowanie, mimo że nie jest bardzo często wykorzystywane, nie jest dla osoby programującej czymś trudnym do zrozumienia. Jeśli zatem nie rozumiesz tego przykładu, weź kartkę papieru i przeanalizuj jak się zmieniają poszczególne zmienne i co się dzieje w programie (ja właśnie bardzo często korzystam z tej metody).

## Podsumowanie

Dzięki tej lekcji udało Ci się poznać wspaniały mechanizm pętli, a konkretnie pętlę for. Zagadnienia tutaj przedstawione są bardzo ważne i niezbędne w Twojej dalszej przygodzie z programowaniem.

Należy sobie zdawać sprawę, że pętle są bardzo użyteczne. Z drugiej jednak strony, nieumiejętne ich użycie, może spowodować zapętlenie programu, dlatego trzeba być bardzo ostrożnym.

W tej lekcji zwróciłem Twoją uwagę na rzeczy, których w większości książek nie przeczytasz. Mam zatem nadzieję, że wykorzystasz zdobytą tutaj dodatkową wiedzę w przyszłości.

# Lekcja 14: Pętle w języku C++ - pętla do while

## Sens poznania innego rodzaju pętli

Jeśli zastanawiasz się, czy warto poznawać kolejny rodzaj pętli, to rzeczywiście Twoje wątpliwości nie są bezpodstawne. Musisz wiedzieć, że za pomocą wszystkich pętli można wykonać w zasadzie te same działania.

Po co zatem uczyć się kilku typów, skoro można by było tylko jednego? Odpowiedź jest prosta - dla wygody. To co za pomocą jednego typu pętli można wykonać bardzo prosto, niekiedy wymaga przekształceń, aby zadziałało dokładnie tak samo przy użyciu innej pętli.

Jeśli nawet to Cię nie przekonuje, to niech przekona Cię fakt, że wszyscy programiści używają wszystkich rodzajów pętli dostępnych w języku. Zatem jeśli kiedykolwiek będziesz miał za zadanie przeanalizować kod napisany przez kogoś innego, chyba nie warto najeść się wstydu, nie rozumiejąc tak prostej rzeczy, jaką jest jedna z pętli w języku C++.

## Podstawy pętli do while

Pętla do while podobnie jak pętla for i podobnie jak wszystkie pozostałe pętle, umożliwi nam powtórzenie określonych operacji tak długo jak warunek końcowy jest spełniony. **Schematyczna** postać pętli wygląda następująco:

```
do
{
  lista_instrukcji
}
while (warunekKoncowy);
```

Lista instrukcji może stanowić jedną instrukcję lub ich grupę. Warto jednak zaznaczyć, że **nawiasy klamrowe są konieczne** nawet jeśli chcemy wykonać tylko jedną instrukcję.

Warunek końcowy jest tylko jeden, ale może występować jako złożony warunek (stworzony za pomocą operatorów logicznych).

Warto teraz zastanowić się jak działa pętla do while. W pierwszym kroku jest wykonywana lista instrukcji zawarta między nawiasami klamrowymi. Następnie jest sprawdzany warunek - jeśli jest on prawdziwy, wówczas pętla wykonuje się ponownie. Pętla wykonuje się do momentu, gdy warunek końcowy będzie fałszywy.

Co to oznacza w praktyce? Oznacza to, że **lista instrukcji pętli do while wykona się co najmniej jeden raz**. Łatwo to zapamiętać, gdy przyjrzymy się postaci pętli - najpierw jest lista instrukcji (czyli najpierw zostaje wykonana lista instrukcji) a dopiero później znajduje się warunek (czyli dopiero teraz warunek zostaje sprawdzony).

Jest to bardzo ważna kwestia i odróżnia ona pętlę **do while** od pętli **for**. W pętli for warunek był sprawdzony przed wykonaniem jakiejkolwiek instrukcji, czyli w rezultacie mogło się zdarzyć tak, że lista instrukcji nie została nigdy wykonana. Tutaj natomiast lista instrukcji zostanie wykonana co najmniej jeden raz.

# Przykłady prostego użycia pętli do while

```
#include <iostream>

using namespace std;

int main()
{
  int ile=6;

  do
  {
     cout <<"Jestem w srodku petli do while\n";
  }
  while (ile<5);

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Co się stało w powyższym przykładzie? Zauważ, że zmienna **ile** ma początkową wartość równą 6. Natomiast warunek jest sformułowany tak, że jest fałszywy (bo 6<5 jest fałszem). Mimo to lista instrukcji (czyli w tym wypadku jedna instrukcja) zostanie wykonana jeden raz, co wynika właśnie z własności pętli do while.

Spójrz teraz na poniższy przykład, a następnie go uruchom. Przy okazji przypomnij sobie skróty klawiszowe przedstawione w poprzedniej lekcji, umożliwiające przerywanie działania, kiedy program się zapętli.

```
#include <iostream>

using namespace std;

int main()
{
  int ile=3;
  do
  {
     cout <<"Pierwsza instrukcja\n";
     cout <<"Druga instrukcja\n";
  }
  while (ile<10);

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Zastanawiasz się pewnie co takiego się stało, że program uległ zapętleniu. Zastanówmy się wspólnie. Zmienna **ile** miała początkową wartość 3. Program wykonał instrukcje znajdujące się pomiędzy nawiasami klamrowymi i przeszedł do sprawdzenia warunku. Warunek jest prawdziwy, bo 3<10 jest prawdą. Zatem program ponownie wykonał instrukcje znajdujące się w nawiasach klamrowych i po raz kolejny sprawdził warunek. I jak się okazało, warunek znów jest prawdziwy, bo zmienna **ile** ma znów wartość 3.

Mam nadzieję, że teraz już wiesz w czym tkwi problem. Zmienna sterująca pętlą (czyli w tym wypadku zmienna **ile**) nie ulega nigdzie zmianie, zatem warunek będzie zawsze prawdziwy. Gdzie jednak możemy zmienić wartość zmiennej sterującej? W pętli for mieliśmy do tego specjalną konstrukcję. Tutaj natomiast zmianę wartości zmiennej należy wykonać wewnątrz listy instrukcji (przy okazji przypominam, że w pętli for było to też możliwe).

Poniżej zatem znajduje się pętla do while użyta zgodnie z naszymi oczekiwaniami:

```
#include <iostream>

using namespace std;

int main()
{
  int ile=3;
  do
  {
     cout <<"Pierwsza instrukcja\n";
     cout <<"Druga instrukcja\n";
     ++ile;
  }
  while (ile<10);

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Nie ma tutaj co prawda nic nadzwyczajnego, zwróć jednak uwagę, że wartość zmiennej zostaje zmieniona na samym końcu listy instrukcji. Takie posunięcie jest najlepsze, bowiem gdybyśmy chcieli wypisywać wartość zmiennej sterującej i instrukcję zmiany wartości zmiennej sterującej umieścilibyśmy na przykład pomiędzy innymi instrukcjami, wówczas część obliczeń lub wypisań mogłaby być wykonywana dla "starej" wartości, a część dla "nowej".

Takie właśnie zjawisko przedstawia poniższy program. W programie będziemy chcieli wypisywać wartość liczby i wartość tej liczby do kwadratu. Poprzez zapisanie instrukcji zmiany wartości zmiennej sterującej pomiędzy innymi instrukcjami, widoczne wyniki będą niezrozumiałe:

```
#include <iostream>

using namespace std;

int main()
{
  int ile=3;
  do
  {
     cout <<"Liczba to "<<ile;
     ++ile;
     cout <<" a kwadrat liczby to "<<ile*ile<<'\n';
  }
  while (ile<10);

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Mam nadzieję, że tym właśnie przykładem zachęciłem Cię do stosowania zasady, aby nie dokonywać zmiany wartości zmiennej sterującej pośród zwykłych instrukcji. Zmianę zmiennej sterującej możemy wykonać na końcu wszystkich instrukcji lub na początku instrukcji ( zależy od

potrzeby), ale lepiej nie robić tego nigdy pomiędzy instrukcjami (chyba, że jest to rzeczywiście uzasadnione).

Oprócz tego pragnę dodać, że oczywiście **zmienną sterującą** możemy zmieniać w dowolny sposób (nie tylko zwiększać o 1), podobnie jak to miało miejsce w pętli for.

# Zasady dotyczące pętli do while

Tak naprawdę chcę Ci teraz uświadomić, że musisz nauczyć się samodzielności. Czy w pętli do while możemy używać dwóch zmiennych sterujących? Czy również może pojawić się problem z liczbami bez znaku? Czy w pętli do while możemy zagnieżdżać inne pętle do while? A może możemy w niej zagnieżdżać również inne rodzaje pętli?

Na każde z postawionych wyżej pytań odpowiedź brzmi **tak**. Jeśli to nie było dla Ciebie takie oczywiste, nie musisz się tego wstydzić, jednak o tak oczywistych sprawach nie będę już wspominał w kolejnych lekcjach.

Od teraz musisz zacząć **naukę przez analogię**. Jeśli coś było dozwolone w innym miejscu, to najprawdopodobniej taka konstrukcja jest również możliwa w zagadnieniu, które właśnie omawiam. Na wszystkie pytania, które przed momentem postawiłem w Twoim imieniu, możesz znaleźć odpowiedzi w lekcji dotyczącej pętli for.

Należy kojarzyć - pętla for to była pętla i pętla do while to też pętla. Zatem najprawdopodobniej większość reguł jest identyczna. Analogicznie, jeśli kiedyś wspomniałem, że nawiasy klamrowe służą do grupowania instrukcji, to taką regułę można zastosować w prawie każdym miejscu.

Tak samo kiedyś wspominałem, że warunki logiczne mogą być albo proste albo złożone wówczas używamy operatorów logicznych. Pamiętaj, że w języku C++ taka reguła obowiązuje zawsze.

Jeśli natomiast w którymkolwiek z przypadków nie będzie to dla Ciebie takie oczywiste, że taka reguła w tym wypadku również obowiązuje, wystarczy, że sprawdzisz. Nie dość, że upewnisz się jak jest naprawdę, to przy okazji nabierzesz trochę doświadczenia programistycznego, które jest niewątpliwie bardzo ważne i cenne.

# Pętla do while - praktyka

Dla ćwiczenia spróbujemy teraz przekształcić fragmenty kodów, w których użyto pętli for na kod wykonujący dokładnie to samo zadanie, ale z użyciem pętli do while.

Kody będę przedstawiał parami - najpierw kod z wykorzystaniem pętli for, a następnie z użyciem pętli do while. Zachęcam Cię jednak do samodzielnego napisania i przekształcenia programów. Dzięki temu na pewno znacznie więcej się nauczysz i zrozumiesz.

**Kod 1** z użyciem pętli **for**:

```
#include <iostream>

using namespace std;

int main()
{
  int i;
  for (i=1;i<=10; ++i)
  {
     cout <<"Wypisuje po raz "<<i;
     cout <<". A to tylko dla testu\n";
  }

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

**Kod 1** z użyciem pętli **do while**:

```
#include <iostream>

using namespace std;

int main()
{
  int i=1;
  do
  {
     cout <<"Wypisuje po raz "<<i;
     cout <<". A to tylko dla testu\n";
     ++i;
  }
  while (i<=10);

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

**Kod 2** z użyciem pętli **for**:

```
#include <iostream>

using namespace std;

int main()
{
  int ktory=1, i, j;

  for (i=1;i<=10; i+=2)
  {
     cout <<"Tak bedzie dla i="<<i<<'\n';
     for (j=5;j>=2;--j)
     {
        cout <<ktory<<". raz. i*j="<<i*j<<'\n';
        ++ktory;
     }
     cout <<"Tak bylo dla i="<<i<<'\n';
     getchar(); // czekanie na nacisniecie ENTER
  }


  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

**Kod 2** z użyciem pętli **do while**:

```
#include <iostream>

using namespace std;

int main()
{
  int ktory=1, i=1, j;

  do
  {
     cout <<"Tak bedzie dla i="<<i<<'\n';
     j=5;
     do
     {
        cout <<ktory<<". raz. i*j="<<i*j<<'\n';
        ++ktory;
        --j;
     }
     while (j>=2);
     cout <<"Tak bylo dla i="<<i<<'\n';
     getchar(); // czekanie na nacisniecie ENTER
     i+=2;
  }
  while (i<=10);

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

**Kod 3** z użyciem pętli **for**:

```
#include <iostream>

using namespace std;

int main()
{
  for (unsigned int i=1;i<=10;i+=1)
  {
     for (unsigned int j=1;j<=i;++j)
        cout <<i*j<<' ';
     cout <<'\n';
  }

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

**Kod 3** z użyciem pętli **do while**:

```
#include <iostream>

using namespace std;

int main()
{
  unsigned int i=1, j;
  do
  {
     j=1;
     do
     {
        cout <<i*j<<' ';
        ++j;
     }
     while (j<=i);
     cout <<'\n';
     ++i;
  }
  while (i<=10);

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

Mam nadzieję, że już dobrze rozumiesz pętlę do while. Jak widzisz zamiany pętli for na pętlę do while nie są takie trudne. Zwróć jednak uwagę na to, że w pętlach do while zazwyczaj **zmienne należy utworzyć przed najbardziej zewnętrzną pętlą** (w pętli for możemy tworzyć zmienne bezpośrednio w części stanów początkowych). Dodatkowo bardzo ważną kwestię stanowi odpowiednie **przypisanie wartości początkowych** zmiennym sterującym.

Szczerze mówiąc, zamiany pętli for na pętlę do while nie byłyby takie proste, gdybyśmy zmienili teraz któryś z warunków w pętli for, tak, że warunek jest od razu fałszywy. Pętla do while jak już wiesz, warunek sprawdza dopiero na końcu, więc mimo, że warunek byłby fałszywy, lista instrukcji wykonałaby się jeden raz (a w pętli for lista instrukcji nie wykonałaby się ani razu, bo tam warunek jest sprawdzany na początku).

## Podsumowanie

Przestawiłem Ci w tym artykule kolejny typ pętli w języku C++. Przeczytaj kolejną lekcję, aby poznać ostatni już typ pętli w języku C++ oraz przeczytać podsumowanie dotyczące pętli.

# Lekcja 15: Pętle w języku C++ - pętla while

## Podobieństwo i różnice z pozostałymi pętlami

Pętla while jest niejako czymś pomiędzy pętlą for i pętlą do while. Cechą, którą ją łączy z pętlą for jest to, że warunek jest sprawdzany na początku pętli, czyli pętla może się nie wykonać ani razu. Jak pamiętasz zapewne, już to jest dużą różnicą w porównaniu z pętlą do while, która zawsze jest wykonywana co najmniej jeden raz.

A co ma wspólnego pętla while z pętlą do while? Poza oczywistym podobieństwem, którym jest słowo "while" w nazwie, podobna jest organizacja pętli oraz podobny sposób zapisu.

Dla mnie podobieństwem jest również to, że obu pętli (while i do while) używam, gdy mam do sprawdzenia złożony warunek logiczny, chociaż, jak już wspominałem, to samo można wykonać za pomocą pętli for.

## Podstawy pętli while

Pętla while podobnie jak pętla for oraz jak pętla do while umożliwia powtarzanie instrukcji tak długo jak warunek końcowy jest spełniony. **Schematycznie** pętlę while możemy zapisać następująco:

```
while (warunekKoncowy)
  lista_instrukcji
```

Przede wszystkim musisz zwrócić uwagę, że w tej pętli nie ma słowa "do" oraz co ważniejsze że warunek jest zapisywany na samym początku pętli - dzięki temu łatwo zapamiętasz, że warunek jest sprawdzany zanim cokolwiek innego zostanie wykonane i w ten sposób, jeśli warunek nie jest spełniony, nic nie zostanie wykonane.

Zwróć dodatkowo uwagę, że nie ma tutaj również nawiasów klamrowych, bowiem nie są one wymagane. Nie są one wymagane, kiedy chcemy wykonać tylko jedną instrukcję (w pętli do while nawiasy klamrowe były wymagane nawet dla jednej instrukcji). Oczywiście tak naprawdę prawie zawsze i tak będziemy musieli użyć nawiasów klamrowych, bowiem raczej nie będziemy nigdy chcieli wykonać tylko jednej instrukcji w treści pętli.

Wartym również zapamiętania jest to, że po nawiasach okrągłych znajdujących się za słowem while, nie ma średnika. Natomiast w przypadku pętli do while, średnik się tam znajdował.

Pozostałe cechy są zbieżne z cechami pętli do while. Warunek może być prosty lub może być znacznie bardziej skomplikowanym wyrażeniem. Podobnie jest z listą instrukcji (co już zasygnalizowałem) - może to być tylko jedna instrukcja, a może być znacznie więcej instrukcji - w tym również zagnieżdżone pętle (zarówno for, while, jak i do while).

# Przykłady prostego użycia pętli while

```
#include <iostream>

using namespace std;

int main()
{
 int ile=6;

 while (ile<5)
    cout <<"Jestem w srodku petli do while\n";

 cout <<"\nNacisnij ENTER aby zakonczyc\n";
 getchar();
 return 0;
}
```

Od razu zaznaczę, że przykład jest bardzo podobny do przykładu pokazanego w lekcji dotyczącej pętli do while. Jest jednak dość spora różnica - gdy uruchomisz powyższy program, to zobaczysz, że tak naprawdę nie została wykonana żadna instrukcja w pętli.

Wyjaśnienie dlaczego tak się stało, jest dość proste. Otóż przed pojawieniem się pętli zmienna sterująca pętli **ile** ma przypisaną wartość 6. Teraz pojawia się pętla. Sprawdzany jest warunek pętli. Okazuje się, że warunek jest fałszywy (bo 6<5 jest fałszem), a zatem instrukcje pętli się nie wykonują i program przechodzi do instrukcji znajdujących się za naszą pętlą.

```
#include <iostream>

using namespace std;

int main()
{
 int ile=3;

 while (ile<10)
 {
    cout <<"Pierwsza instrukcja\n";
    cout <<"Druga instrukcja\n";
 }

 cout <<"\nNacisnij ENTER aby zakonczyc\n";
 getchar();
 return 0;
}
```

Z kolei w powyższym programie zapomnieliśmy o zmianie (zmniejszaniu lub zwiększaniu) wartości zmiennej sterującej pętlą. Warunek końcowy jest zatem zawsze prawdziwy. Aby przerwać działanie programu, wiesz już jak postępować (jeśli nie wiesz, to wróć do dwóch poprzednich lekcji). Przy okazji zwróć uwagę, że tutaj już zastosowaliśmy nawiasy klamrowe (bowiem mamy dwie instrukcje).

Ostatecznie, aby pętla wykonała się zgodnie z naszymi oczekiwaniami, program musi wyglądać następująco:

```
#include <iostream>

using namespace std;

int main()
{
 int ile=3;

 while (ile<10)
 {
    cout <<"Pierwsza instrukcja\n";
    cout <<"Druga instrukcja\n";
    ++ile;
 }

 cout <<"\nNacisnij ENTER aby zakonczyc\n";
 getchar();
 return 0;
}
```

Przy okazji chcę wspomnieć, że tutaj, podobnie jak w wypadku pętli do while, należy zmieniać wartość zmiennej sterującej najlepiej na końcu wszystkich instrukcji, bowiem w przeciwnym wypadku uzyskamy nieoczekiwane wyniki (wspominałem już o tym w lekcji dotyczącej pętli do while)

# Pętla while - praktyka

Dla ćwiczenia spróbujemy teraz przekształcić fragmenty kodów, w których użyto pętli do while na kod wykonujący dokładnie to samo zadanie, ale z użyciem pętli while.

Kody będę ponownie przedstawiał parami - najpierw kod z wykorzystaniem pętli do while, a następnie z użyciem pętli while. Zachęcam Cię jednak do samodzielnego napisania i przekształcenia programów. Dzięki temu na pewno znacznie więcej się nauczysz i zrozumiesz.

Przy okazji zwróć uwagę, że to te same programy, które przedstawiłem w poprzedniej lekcji. Kompletując kody z tej lekcji i z poprzedniej, uzyskasz 3 sposoby wykonania tego samego zadania: z wykorzystaniem pętli for, z wykorzystaniem pętli do while i z wykorzystaniem pętli while - taka "ściągawka", może Ci się przydać w przyszłości.

Ponieważ w wypadku wszystkich poniższych przekształceń, nie dzieje się tak naprawdę nic szczególnego, pozwól, że powstrzymam się od zbędnych uwag, komentarzy i wyjaśnień.

**Kod 1** z użyciem pętli **do while**:

```
#include <iostream>

using namespace std;

int main()
{
  int i=1;
  do
  {
     cout <<"Wypisuje po raz "<<i;
     cout <<". A to tylko dla testu\n";
     ++i;
  }
  while (i<=10);

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

**Kod 1** z użyciem pętli **while**:

```
#include <iostream>

using namespace std;

int main()
{
  int i=1;
  while (i<=10)
  {
     cout <<"Wypisuje po raz "<<i;
     cout <<". A to tylko dla testu\n";
     ++i;
  }

  cout <<"\nNacisnij ENTER aby zakonczyc\n";
  getchar();
  return 0;
}
```

**Kod 2** z użyciem pętli **do while**:

```
#include <iostream>

using namespace std;

int main()
{
  int ktory=1, i=1, j;

  do
  {
     cout <<"Tak bedzie dla i="<<i<<'\n';
     j=5;
     do
     {
        cout <<ktory<<". raz. i*j="<<i*j<<'\n';
```

```
        ++ktory;
        --j;
      }
      while (j>=2);
      cout <<"Tak bylo dla i="<<i<<'\n';
      getchar(); // czekanie na nacisniecie ENTER
      i+=2;
   }
   while (i<=10);

   cout <<"\nNacisnij ENTER aby zakonczyc\n";
   getchar();
   return 0;
}
```

**Kod 2** z użyciem pętli **while**:

```
#include <iostream>

using namespace std;

int main()
{
 int ktory=1, i=1, j;

 while (i<=10)
 {
    cout <<"Tak bedzie dla i="<<i<<'\n';
    j=5;
    while (j>=2)
    {
       cout <<ktory<<". raz. i*j="<<i*j<<'\n';
       ++ktory;
       --j;
    }
    cout <<"Tak bylo dla i="<<i<<'\n';
    getchar(); // czekanie na nacisniecie ENTER
    i+=2;
 }

 cout <<"\nNacisnij ENTER aby zakonczyc\n";
 getchar();
 return 0;
}
```

**Kod 3** z użyciem pętli **do while**:

```
#include <iostream>

using namespace std;

int main()
{
  unsigned int i=1, j;
  do
  {
     j=1;
     do
     {
        cout <<i*j<<' ';
        ++j;
     }
```

```
      while (j<=i);
      cout <<'\n';
      ++i;
   }
   while (i<=10);

   cout <<"\nNacisnij ENTER aby zakonczyc\n";
   getchar();
   return 0;
}
```

**Kod 3** z użyciem pętli **while**:

```
#include <iostream>

using namespace std;

int main()
{
   unsigned int i=1, j;
   while (i<=10)
   {
      j=1;
      while (j<=i)
      {
         cout <<i*j<<' ';
         ++j;
      }
      cout <<'\n';
      ++i;
   }

   cout <<"\nNacisnij ENTER aby zakonczyc\n";
   getchar();
   return 0;
}
```

# Pętla while - podsumowanie

W tej lekcji przedstawiłem Ci ostatnią pętlę języka C++ - pętlę while. Pętla while jest tak samo użyteczna jak dwie pozostałe pętle, dlatego jeśli nadal nie do końca rozumiesz jak ona działa, radzę Ci dokończyć czytanie tej lekcji i przeczytać ją ponownie.

# Pętle w języku C++ - podsumowanie

Można powiedzieć, że wiesz już prawie wszystko o pętlach. W tym momencie zamiana pętli i rozumienie ich działania powinno przychodzić Ci już z łatwością. Jeśli tak nie jest, należy jeszcze poćwiczyć, bowiem w przeciwnym wypadku, później będzie Ci bardzo trudno.

Niedługo poznasz jeszcze kilka instrukcji, które przydadzą Ci się w posługiwaniu pętlami i wówczas Twoja wiedza na ten temat będzie naprawdę pełna. Dowiesz się także, w jaki sposób połączyć znajomość pętli z tablicami, co sprawi, że odwoływanie się do elementów tablic będzie bardzo łatwe.

Ostatnią kwestią pozostaje, kiedy należy używać pętli for, kiedy pętli while, a kiedy pętli do while. Tak naprawdę nie ma żadnej zasady i każda pętla może zostać użyta do wykonania dowolnego zadania.

Mimo, że napisałem już trochę mniej czy bardziej skomplikowanych programów, to ciężko tak naprawdę sprecyzować, kiedy warto używać jednej pętli a kiedy innej. Tak naprawdę jest to chyba w dużej mierze kwestią praktyki i przyzwyczajeń.

Jeśli chodzi o mnie, pętli for używam, gdy posługuję się liczbami i gdy używam tylko jednego warunku dla zakończenia pętli. W innych przypadkach używam pętli while i do while.

Jak jednak wiesz, w pętli for można używać wielu warunków i niekoniecznie zmiennymi sterującymi muszą być liczby. Ja jednak mam już swoje przyzwyczajenia. Ty oczywiście zastosujesz własną technikę, bo to Tobie ma się wygodnie i efektywnie pisać programy.

Na koniec chcę zaznaczyć, że pętle są jedną z podstawowych konstrukcji języka C++ i bez ich znajomości nie możesz liczyć na to, że uda Ci się pisać pożyteczne programy. Dlatego jak zwykle zachęcam Cię do upewnienia się, że naprawdę rozumiesz omówione zagadnienia i do spędzenia dodatkowego czasu z pętlami i z kompilatorem.

# Lekcja 16: Tablice i pętle w języku C++. Efektywne zarządzanie danymi w C++

## Wprowadzenie

Pojawienie się tej lekcji sygnalizowałem już co najmniej dwukrotnie: raz, kiedy wspominałem o tablicach w języku C++ i raz, kiedy wspominałem o pętlach występujących w C++.

Tak naprawdę jednak, przedstawione do tej pory możliwości zarówno pętli, jak i tablic są niepełne. Oczywiście wiesz już jak dużo można osiągnąć za pomocą tablic oraz jak dużo można osiągnąć za pomocą pętli, jednak tak naprawdę prawdziwe możliwości daje połączenie tablic i pętli i ich wspólne stosowanie w programach.

W tej lekcji postaram Ci się właśnie przedstawić jak dużo można dzięki takiemu połączeniu osiągnąć oraz udowodnić, że bez dobrej znajomości pętli i tablic, dalsza nauka C++ będzie prawie niemożliwa.

## Czas zacząć kojarzyć

Jeśli nie wiesz jeszcze, w jaki sposób połączyć wykorzystanie pętli w tablicach, postaraj się skojarzyć dwa proste fakty o których wspominałem już w tym kursie.

Przede wszystkim jak już wiesz pętle umożliwiają powtarzanie pewnej operacji tak długo jak warunek końcowy pętli będzie spełniony. Wszystkie pętle posiadają zmienne (jedną lub kilka) sterujące. W bardzo łatwy sposób można zmieniać wartość zmiennej sterującej w kolejnych krokach pętli i bardzo łatwo stworzyć pętlę, która wykona określoną operację dla zmiennej sterującej z określonego przedziału.

Z drugiej strony, tablice umożliwiają przechowywanie zmiennych danego typu i dla zmiennej o nazwie **auto** typu tablicowego, poszczególne elementy tablicy to auto[0], auto[1], auto[2] itd. Schematycznie zatem można zapisać postać dowolnego elementu tablicy jako **auto[indeks]**, gdzie indeks zmienia się od 0 do pewnej liczby określonej rozmiarem tablicy.

Jak zatem połączyć pętle z tablicami? Nie jest to takie trudne - zmienną indeksującą tablicy (czyli u nas nazwaną **indeks**) należy potraktować jako zmienną sterującą pętlą i zmieniać ją od **0** aż do **rozmiar tablicy -1** i dzięki temu, uda nam się dostać za pomocą jednej pętli do każdego elementu tablicy i dokonać na nim dowolnych operacji.

# Przykłady

Na szczęście użycie tablic i pętli jest tak naprawdę bardzo łatwe. Teraz postaram Ci się przedstawić różne przykłady, ilustrujące jak wiele można osiągnąć dzięki umiejętnemu posługiwaniu się poznanymi mechanizmami.

```
#include <iostream>

using namespace std;

int main()
{
  int calkowite[3]={6,7,10};

  // wersja bez petli
  cout <<calkowite[0]<<' '<<calkowite[1]<<' '<<calkowite[2]<<'\n';

  // wersja z uzyciem petli
  for (unsigned int i=0;i<3;++i)
     cout <<calkowite[i]<<' ';

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jak widzisz w powyższym przykładzie pokazałem Ci, w jaki sposób można wypisać wszystkie elementy tablicy. Co prawda w tym wypadku zysk jest niewielki - bowiem tablica ma tylko 3 elementy i w przybliżeniu napisanie kodu wypisującego elementy tej tablicy zajmuje tyle samo czasu, jednak zastanów się, która metoda byłaby wygodniejsza dla tablicy 100-elementowej.

Zastosowaliśmy tutaj przedstawione wcześniej podejście. Zmienna decydująca o tym, którym elementem tablicy się obecnie "zajmujemy" (tutaj oznaczona jako **i**, bo mamy zapis **calkowite[i]**), została wykorzystana jako zmienna sterująca pętlą. Wartość zmiennej sterującej zmienia się od 0 do 2, bowiem nasza tablica jest 3-elementowa.

Zapis przedstawiony w pętli nie jest tak naprawdę najszczęśliwszy, bo dlaczego w pętli warunkiem końcowym jest **<3**? Ty zapewne odpowiesz, że to przecież liczba elementów tablicy. Co jednak jeśli za moment zmienisz liczbę elementów tablicy? Oczywiście tutaj należałoby również zmienić wartość. Dodatkowo, gdyby taka pętla znajdowała się w znacznie odleglejszej części programu od miejsca, gdzie tablica zostaje utworzona, to wówczas w końcu sami byśmy zapomnieli, dlaczego warunek końcowy jest taki a nie inny.

Dlatego też w ogólnym wypadku lub w bardziej skomplikowanych przykładach, najczęściej wprowadzamy zmienną pomocniczą (najlepiej stałą, czyli z modyfikatorem **const**), którą następnie wykorzystujemy w warunku końcowym pętli. Dzięki temu zrozumienie warunku końcowego będzie łatwiejsze i nawet, gdy okaże się, że potrzebujemy mniejszej lub większej tablicy, wystarczy, że zmienimy tylko wartość zmiennej określającej rozmiar tablicy.

Wspomniany przed momentem sposób ilustruje poniższy przykład:

```
#include <iostream>

using namespace std;

int main()
{
  const int ile=3; // rozmiar tablicy
  int calkowite[ile]={6,7,10};

  for (unsigned int i=0;i<ile;++i)
     cout <<calkowite[i]<<' ';

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Dla formalności dodam, że tutaj zmienną określającą rozmiar tablicy, jest zmienna **ile** - jak widzisz, zastosowana nazwa zmiennej mówi dość dużo o jej przeznaczeniu.

Oczywiście możemy nie tylko wypisywać elementy tablicy. Możemy również je pobierać w ten łatwiejszy sposób oraz dokonywać pewnych modyfikacji na danych. Jak to zrobić dowiesz się przyglądając się poniższemu przykładowi:

```
#include <iostream>

using namespace std;

int main()
{
  const int ile=6; // rozmiar tablicy
  int calkowite[ile];
  unsigned int i; //zmienna sterujaca petlami


  // pobieranie elementow tablicy
  for (i=0;i<ile;++i)
  {
     cout <<"Podaj "<<i+1<<". element tablicy: ";
     cin >>calkowite[i];
     cin.ignore();
  }

  // wypisanie elementow tablicy
  cout <<"\nOto tablica: ";
  for (i=0;i<ile;++i)
     cout <<calkowite[i]<<' ';

  // dodanie liczby 2 do kazdego elementu tablicy
  for (i=0;i<ile;++i)
     calkowite[i]+=2;

  // ponowne wypisanie elementow tablicy
  cout <<"\nOto zmodyfikowana tablica: ";
  for (i=0;i<ile;++i)
     cout <<calkowite[i]<<' ';

  cout <<"\n\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jak widzisz, w powyższym przykładzie realizujemy wszystkie operacje, które bez użycia pętli nie byłyby już takie łatwe i szybkie. Dodatkowo zmienna sterująca pętlami została utworzona przed wszystkimi pętlami, dzięki czemu, program wykona się w rzeczywistości nieco szybciej.

Innym przykładowym programem może być obliczenie sumy wszystkich wartości występujących w tablicy. Oczywiście taką operację, jak i wszystkie inne można przeprowadzić za pomocą dowolnej pętli (tak jak wszystkie inne operacje), jednak teraz, żeby pokazać, że do przeglądania tablic można używać nie tylko pętli for, użyjemy dwóch pozostałych typów pętli.

```
#include <iostream>

using namespace std;

int main()
{
  const int ile=6; // rozmiar tablicy
  float calkowite[ile]={3.45, 5, -10, 2.78, 4, 2.22};
  unsigned int i; //zmienna sterujaca petlami
  float suma_pocz, suma_kon;

  // suma liczona "od przodu" petla while
  i=0;
  suma_pocz=0;
  while (i<ile)
  {
     suma_pocz+=calkowite[i];
     ++i;
  }

  // suma liczona "od konca" petla do while
  i=ile;
  suma_kon=0;
  do
  {
     suma_kon+=calkowite[i];
     --i;
  }
  while (i>0);
  //tutaj i ma wartosc 0 i trzeba dodac do sumy calkowite[i]
  suma_kon+=calkowite[i];

  cout <<"Suma elementow liczona od poczatku petla while wynosi "<<suma_pocz;
  cout <<"\nSuma elementow liczona od konca petla do while wynosi "<<suma_kon;

  cout <<"\n\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jak widzisz, w powyższym programie obliczyliśmy sumę elementów tablicy dwoma metodami. Wszystko powinno być dość łatwe do zrozumienia poza jedną kwestią, a mianowicie, dlaczego po pętli **do while** dodajemy do sumy jeszcze wartość początkowego elementu.

Zrobiliśmy tak dlatego, bowiem w pętli ograniczyliśmy ostatnią poprawną wartość pętli do 1 (warunek >0), co spowoduje, że ostatnią wartością zmiennej sterującej będzie 0. Gdybyśmy natomiast zmienili warunek, pisząc przykładowo **>=0**, wówczas otrzymalibyśmy opisywany w poprzednich lekcjach błąd zapętlenia w przypadku zmiennych z modyfikatorem **unsigned** - bo taki modyfikator ma właśnie zmienna **i**.

Oczywiście wystarczyło zastosować zmienną bez modyfikatora unsigned i byłoby po kłopocie. Stosując jednak taką "sztuczkę", teoretycznie udało nam się zwiększyć zakres dopuszczalnych

dodatnich wartości zmiennej sterującej (chociaż tutaj i tak z tego nie skorzystaliśmy).

Na koniec przedstawię przykład realizacji takiego zadania: zmień obecną tablicę tak, aby każdy nowy element był sumą dwóch dotychczasowych elementów: samego siebie i elementu go poprzedzającego. Inaczej mówiąc chcemy, żeby w elemencie o indeksie jeden, znalazła się suma elementu z indeksem jeden i z indeksem zero, a w ostatnim elemencie tablicy, suma ostatniego i przedostatniego elementu. Pierwszy element tablicy (element o indeksie zero) powinien pozostać taki sam, bowiem nie ma przed nim żadnego innego elementu.

Wykonując modyfikacje na tablicy, trzeba być bardzo ostrożnym. Tutaj właśnie dokonując operacji od początku tablicy do końca, nie udałoby się nam osiągnąć właściwego wyniku (bez użycia dodatkowej zmiennej), bowiem "niechcący" zamazywalibyśmy wcześniejsze wartości. Dlatego też w tym wypadku, aby realizacja postawionego problemu była łatwa, należy dokonywać operacje na tablicy od jej końca. Oto prosty przykład realizacji:

```
#include <iostream>

using namespace std;

int main()
{
  const unsigned int ile=6; // rozmiar tablicy
  float tab[ile]={3.45, 5, -10, 2.78, 4, 2.22};

  cout <<"Poczatkowa tablica: ";
  for (unsigned int i=0;i<ile;++i)
     cout <<tab[i]<<' ';

  for (unsigned int i=ile-1;i>0;--i)
     tab[i]=tab[i]+tab[i-1];

  cout <<"\n\nZmodyfikowana tablica: ";
  for (unsigned int i=0;i<ile;++i)
     cout <<tab[i]<<' ';

  cout <<"\n\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

# Lekcja 17: Instrukcja break w C++. Przerywanie pętli w języku C++

## Wprowadzenie

Jeśli wydawało Ci się, że wiesz już zupełnie wszystko o pętlach, to muszę Cię zmartwić: wszystkiego jeszcze nie wiesz. Znasz natomiast już wszystkie typy pętli w języku C++.

W tej i w przyszłej lekcji przedstawię Ci dodatkowe instrukcje związane z pętlami i ich obsługą. Wtedy będziesz wiedzieć już wszystko na temat pętli i więcej o pętlach wspominać już w kursie nie będę - Ty już po prostu będziesz wiedzieć absolutnie wszystko.

## Instrukcja break - podstawy

Tym razem przejdę od razu do rzeczy i krótko i rzeczowo wyjaśnię do czego służy instrukcja **break**.

Instrukcja break pozwoli nam **przerwać działanie jednej z pętli: for, while, do while oraz instrukcji switch**. O ile o pętlach wiesz już prawie wszystko, o tyle instrukcja switch pozostaje dla Ciebie zagadką - ale już niedługo - wkrótce dowiesz się do czego ona służy.

Najprostszy sposób użycia instrukcji **break** przedstawia poniższy przykład: staramy się obliczyć sumę elementów tablicy do pierwszego napotkanego elementu ujemnego.

```
#include <iostream>

using namespace std;

int main()
{
  const int ile=6; // rozmiar tablicy
  float tab[ile]={3.45, 5, 2.78, -2, 4, 2.22};
  unsigned int i; // zmienna sterujaca
  float suma=0;

  for (i=0;i<ile;++i)
  {
     if (tab[i]<0)
        break;
     suma+=tab[i];
  }

  cout <<"Suma liczb wynosi "<<suma<<'\n';
  cout <<"Zakonczono na indeksie i="<<i;

  cout <<"\n\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jak już wspomniałem, program oblicza sumę liczb tablicy aż do momentu gdy dany element tablicy jest ujemny. Jest to realizowane w pętli for. Przed próbą dodania sprawdzane jest, czy element jest ujemny.

Jeśli element jest ujemny, to zostanie wywołana instrukcja break, która powoduje zakończenie pętli for - w tym wypadku oznacza to przejście do instrukcji wypisania sumy, znajdującej się tuż za pętlą.

Jeśli natomiast warunek nie jest spełniony, do sumy zostanie dodana wartość bieżącego elementu tablicy.

Zauważ, że warunek sprawdzania jest zapisany tak, że nie używamy słowa else, aby wykonać zwiększenie sumy - nie jest ono w tym wypadku w ogóle potrzebne, chociaż jego zapisanie nie będzie żadnym błędem.

Na dodatek zwróć uwagę, że po zakończeniu pętli, możemy w dość łatwy sposób określić czy instrukcja break została wykonana czy nie. Musimy jednak w tym celu zadeklarować zmienną sterującą pętlą przed pętlą, a nie w części warunków początkowych (tutaj tak właśnie postąpiliśmy).

Sprawdzając wartość zmiennej sterującej pętlą można w bardzo sprytny sposób wykorzystać do wielu różnych celów, sprawiając, że dzięki temu program wykona się szybciej.

# Przykład zastosowania

Przykładem niech będzie program sprawdzający, czy liczba jest liczbą pierwszą czy nie. Jeśli liczba nie jest liczbą pierwszą, to znaczy, że dzieli się tylko przez samą siebie i przez jeden i po zakończeniu pętli zmienna sterująca powinna mieć wartość równą wartości liczby, którą sprawdzamy.

Natomiast, jeśli liczba dzieli się także przez inne liczby, to używając instrukcji break sprawimy, że zmienna sterująca po zakończeniu pętli będzie miała wartość mniejszą i stąd będziemy wiedzieć, że sprawdzana jest liczbą złożoną.

Oto prosty program, który korzystając z instrukcji **break** ułatwia stwierdzenie, czy dana liczba jest liczbą pierwszą czy złożoną.

```
#include <iostream>

using namespace std;

int main()
{

  unsigned long int liczba;
  unsigned int i; // zmienna sterujaca

  cout <<"Podaj liczbe calkowita wieksza od jednego: ";
  cin >>liczba;
  cin.ignore();
  for (i=2;i<liczba;++i)
     if ((liczba%i)==0)
        break;

  cout <<"Liczba "<<liczba<<" jest liczba ";
  if (i==liczba)
     cout <<"pierwsza.";
  else
     cout <<"zlozona.";


  cout <<"\n\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

# Zagnieżdżone pętle a instrukcja break

Instrukcja break pozwala jedynie na przerwanie pętli w której się znajduje! - jest to bardzo ważne stwierdzenie i trzeba o nim zawsze pamiętać. Tak naprawdę jest to duża wada instrukcji **break**.

Jeśli stosujemy zagnieżdżone pętle i użyjemy instrukcji break w pętli najbardziej wewnętrznej, to **zostanie przerwana jedynie najbardziej wewnętrzna pętla**, pozostałe pętle będą się wykonywać - co może również oznaczać powtórzenie pętli, którą właśnie przerwaliśmy. Oto bardzo prosty przykład obrazujący taką sytuację:

```
#include <iostream>

using namespace std;

int main()
{
  unsigned int i, j; // zmienne sterujace
  for (i=0;i<3;++i)
  {
     cout <<"\nzewnetrzna dla i="<<i<<'\n';
     for (j=0;j<2;++j)
     {
        if (i==1 && j==1)
           break;
        cout <<"wewnetrzna dla j="<<j<<'\n';
     }
  }

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Postaraj się bardzo dokładnie prześledzić działanie powyższego programu, aby zrozumieć wypisywane komunikaty na ekran. Jeśli sprawdzisz działanie powyższego kodu, zauważysz, że mimo że instrukcja break została wykonana, zewnętrzna pętla kontynuowała swoje działanie.

Czasami może chodzić nam właśnie o takie zachowanie, czasami jednak chcielibyśmy, aby instrukcja break wywołana w zagnieżdżonej pętli powodowała zakończenie nawet tej najbardziej zewnętrznej pętli. Takie zadanie możemy zrealizować następująco:

```
#include <iostream>

using namespace std;

int main()
{
  unsigned int i, j; // zmienne sterujace
  unsigned int end=0;
  for (i=0;i<3;++i)
  {
     cout <<"\nzewnetrzna dla i="<<i<<'\n';
     for (j=0;j<2;++j)
     {
        if (i==1 && j==1)
        {
           end=1;
           break;
        }
        cout <<"wewnetrzna dla j="<<j<<'\n';
     }
     if (end) // - to samo co if (end!=0)
        break;
```

```
  }

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jak widzisz, aby udało się zrealizować wychodzenie za pomocą jednej instrukcji break z zagnieżdżonych pętli, można wprowadzić dodatkową zmienną i tuż przed wywołaniem instrukcji break w pętli wewnętrznej zmienić wartość tej zmiennej. Następnie w "nadrzędnej" pętli musimy sprawdzać wartość tej zmiennej.

Jest to pewien sposób, chociaż nie jest tak naprawdę dobry w przypadku dużej liczby zagnieżdżonych pętli, bowiem wówczas należałoby wprowadzić większą liczbę zmiennych pomocniczych i większą liczbę dodatkowych warunków.

Jednak w takim wypadku jak tutaj - czyli gdy mamy dwie zagnieżdżone pętle, warto wiedzieć, jak sobie poradzić. Czasami nie jest łatwo wpaść na takie rozwiązania, więc warto nawet sobie zapisać, jak wykonać wyjście z zagnieżdżonych pętli z użyciem instrukcji **break**.

# Przykład "sprytnego" wykorzystania instrukcji break

W przypadku tablic często się zdarza, że chcemy sprawdzić, czy dany element znajduje się już w tablicy i o ile się znajduje, chcielibyśmy znać jego pozycję. Nie interesuje nas przy tym ile razy ten element się znajduje w tej tablicy, tylko czy w ogóle tam jest.

```
#include <iostream>

using namespace std;

int main()
{
  const int ile=8;
  int tab[ile]={5,4,8,2,0,5,-3,6};
  unsigned int i; // zmienna sterujaca
  int elem; // szukany element

  cout <<"Podaj wartosc szukanego elementu: ";
  cin >>elem;
  cin.ignore();

  for (i=0; i<ile; ++i)
     if (tab[i]==elem)
        break;

  if (i!=ile)
     cout <<"Element jest w tablicy i znajduje sie na pozycji "<<i<<'\n';
  else
     cout <<"Elementu nie ma w tablicy\n";

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

W powyższym programie została wykorzystana cecha pętli for i instrukcja break (ukazana również w przypadku przykładu ze stwierdzeniem czy liczba jest pierwsza czy złożona).

Jeśli element w tablicy się nie znajduje, to zmienna sterująca pętlą będzie miała wartość **ile** - i to właśnie sprawdzamy po zakończeniu pętli.

Jeśli natomiast element znajduje się w tablicy, pętla jest przerywana i dzięki temu jest zapamiętywany indeks w tablicy, gdzie się ten element znajduje.

Przy takim zastosowaniu zaoszczędziliśmy jedną zmienną, w której powinniśmy pamiętać czy element jest w tablicy czy nie. U nas taką rolę spełnia indeks elementu - zawiera on niejako informację, czy element znajduje się w tablicy oraz dodatkowo ewentualny indeks tego elementu.

Jeśli zależałoby nam na znalezieniu indeksu ostatniego z powtarzających się elementów w tablicy, powinniśmy przeszukiwać tablicę od końca.

## Podsumowanie

W tej lekcji przedstawiłem Ci instrukcję break bardzo powszechnie wykorzystywaną podczas bardziej skomplikowanych pętli czy w momencie, kiedy zależy nam na jak najszybszym działaniu programu.

Instrukcja ta na pewno nie jest wykorzystywana w każdym programie, jednak naprawdę warto ją znać, bowiem jak Ci starałem się udowodnić, dzięki jej znajomości, można uzyskać bardzo ciekawe informacje dotyczące liczb czy tablic.

# Lekcja 18: Instrukcje continue i goto w C++. Obsługa pętli i etykiet w programach C++

## Instrukcja continue - podstawy

Instrukcja continue podobnie jak instrukcja break jest związana z pętlami i podobnie jak instrukcję break można ją stosować zarówno w przypadku pętli for, while jak i do while. Różnicą jest to, że instrukcji tej nie stosujemy gdy używamy instrukcji switch (już wkrótce dowiesz się czym jest instrukcja switch).

Instrukcja continue powoduje **przerwanie wykonania bieżącego kroku pętli i przejście do następnego kroku**. Instrukcja continue działa tylko na pętlę w której się bezpośrednio znajduje - nie da się spowodować przerwania wykonania kroku pętli zewnętrznej.

Bardzo prosty przykład obrazujący zachowanie instrukcji continue, obrazuje poniższy program:

```
#include <iostream>

using namespace std;

int main()
{
  for (unsigned int i=0;i<=12;++i)
  {
     if ((i%3)==0)
        continue;
     cout <<"Liczba "<<i<<" nie jest podzielna przez 3\n";
  }

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jak widzisz, nic szczególnego w naszym przykładzie się nie dzieje. Po prostu jeśli liczba jest podzielna przez 3, zostanie wykonywana instrukcja continue, czyli przechodzimy do następnego

kroku wywołania pętli. Sprawia to, że wszystkie instrukcje znajdujące się w dalszej części pętli nie są wykonywane (tutaj taką instrukcją jest wypisanie informacji na ekran).

Oczywiście w tym prostym przykładzie, można by było obejść się bez instrukcji **continue** poprzez zastosowanie instrukcji **if**, jednak nie zawsze skorzystanie z instrukcji if będzie możliwe lub wygodne - czasami po prostu lepiej wykorzystać nowo poznaną instrukcję.

# Instrukcja continue - informacje dodatkowe

Instrukcja **continue** tak naprawdę nie jest instrukcją często używaną. Powiedziałbym, że jest używana bardzo rzadko i to w bardziej skomplikowanych programach/pętlach. Nie przedstawię Ci jednak takiego przykładu, gdyż nie ma to większego sensu, bowiem musiałbym zamieścić tu kilkaset linii kodu, aby zobrazować Ci sens użycia pętli **continue**, a niestety byłoby w tym programie zbyt wiele elementów, których jeszcze nie znasz.

Przedstawię Ci natomiast dwa proste przykłady, które zobrazują, że instrukcja **continue** po prostu upraszcza zapis i niekiedy właśnie z tego powodu warto z niej skorzystać.

```
#include <iostream>

using namespace std;

int main()
{
  for (unsigned int i=3;i<=17;++i)
  {
     if ((i%4)==0)
     {
        cout <<"Liczba ";
        cout <<i;
        cout <<" jest podzielna przez 4\n";
     }
     else // dobrze ze if "miesci sie na ekranie" i wiem do czego jest ten else
     {
        cout <<"Liczba ";
        cout <<i;
        cout <<" nie jest podzielna przez 4\n";
     } // do czego jest ten nawias?
  } // a do czego ten?
  /*
     chyba bedzie trzeba uzyc komentarzy, bo niedlugo nie bede wiedziec o co
tutaj chodzi
  */

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Oczywiście kod został dobrany celowo, aby przedstawić Ci problem. W tym celu mamy więcej instrukcji wypisania (chociaż można by było je zrealizować za pomocą jednej instrukcji). Jak widać z komentarzy, może być w pewnym momencie trudno zrozumieć, co się dzieje w programie.

Mamy dwa razy nawiasy klamrowe, więc wypadałoby je skomentować. Dodatkowo warto by też było skomentować do czego służy **else** - tutaj to jeszcze widać, bowiem po instrukcji **if** są tylko 3 instrukcje, ale co jeśli byłoby ich 20 albo 50?

Jak widzisz nawet w tak prostym programie, można się prędzej czy później pogubić, a nawet jeśli nie pogubić, to niepotrzebnie tracić czas na zrozumienie tego kodu.

Używając instrukcji **continue**, uda nam się w znacznym stopniu uprościć kod programu i uczynić

go nieco łatwiejszym do zrozumienia. Oto ten sam program z użyciem instrukcji continue:

```
#include <iostream>

using namespace std;

int main()
{
  for (unsigned int i=3;i<=17;++i)
  {
     if ((i%4)==0)
     {
        cout <<"Liczba ";
        cout <<i;
        cout <<" jest podzielna przez 4\n";
        continue;
     }
     // nie ma else - problem komentarza zniknal
     cout <<"Liczba ";
     cout <<i;
     cout <<" nie jest podzielna przez 4\n";
  } // tylko jeden nawias - wiec latwo zauwazyc ze dotyczy petli for

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Mam nadzieję, że również dla Ciebie powyższy program wygląda prościej od tego poprzedniego. Przede wszystkim nie ma części else, więc nie będziemy musieli stosować komentarzy, nawet jeśli w części if będzie wiele instrukcji.

Dodatkowo nie pojawiają się już 2 nawiasy klamrowe obok siebie, więc komentarze nie są już konieczne (chociaż oczywiście i tak komentarz może się przydać).

Podsumowując część tej lekcji dotyczącą instrukcji **continue**, ważne jest, aby udało Ci się zapamiętać, że taka instrukcja w ogóle istnieje i do czego w przybliżeniu służy.

Jak już wspomniałem i tak nie będziesz z tej instrukcji zbyt często korzystać. Kiedyś jednak nadejdzie dzień, że i Tobie instrukcja ta bardzo się przyda i ułatwi napisanie poważnego programu.

# Instrukcja skoku goto - wprowadzenie

Instrukcja skoku goto umożliwia przejście do dowolnego miejsca w programie. Pisząc do dowolnego miejsca w programie mam na myśli takie programy, jakie przedstawiłem Ci do tej pory.

Kiedy poznasz już czym są funkcje, wtedy dowiesz się, że za pomocą instrukcji goto można przejść do dowolnego miejsca, ale w bieżącym zasięgu (na razie tego nie musisz rozumieć).

Jednocześnie chcę bardzo wyraźnie zaznaczyć, że instrukcja **goto** nie jest w żaden sposób powiązana z pętlami! Co prawda najczęściej instrukcję skoku **goto** stosuje się właśnie w przypadku pętli, jednak nie jest to żadnym wymogiem (w przeciwieństwie do instrukcji continue i instrukcji break, z tym, że tą drugą można również zastosować w przypadku instrukcji switch).

# Instrukcja skoku goto - etykiety

Instrukcja skoku goto jest ściśle powiązana z etykietami. Ponieważ za pomocą instrukcji goto chcemy przejść do jakiegoś konkretnego miejsca w naszym programie, to musimy to miejsce jakoś oznaczyć. Do tego właśnie służą etykiety.

Sposób deklaracji etykiety jest następujący:

nazwaEtykiety:

Jak więc widzisz to nic trudnego. Zwróć tylko uwagę, że na końcu znajduje się dwukropek, a nie średnik - łatwo można się pomylić.

Nazwy etykiet muszą być różne między sobą, nie muszą jednak mieć innych nazw niż zmienne. Kompilator potrafi po prostu odróżnić nazwy etykiet i nazwy zmiennych.

Jeśli w programie mamy już etykietę, to żeby do niej przejść piszemy **goto nazwaEtykiety;** czyli na przykład **goto koniec;** Przy okazji warto dodać, że etykieta może znajdować się zarówno przed jak i po użyciu instrukcji goto.

Jeśli na przykład chcemy za pomocą instrukcji goto przejść np. do 22 linijki w naszej programie, to powinniśmy w linijce 21 napisać etykietę np. **linia22:**, a następnie w wybranym miejscu należy napisać **goto linia22;**

Oto prosty przykład programu z etykietami i instrukcją **goto**:

```
#include <iostream>

using namespace std;

int main()
{
poczatek:
  int i=5;
  ++i;
  goto koniec;
  i=0;
  cout <<"To jest jakis tekst. I tak nie zostanie wypisany\n";

koniec:
  cout <<"Zmienna i ma wartosc "<<i<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jeśli dobrze się przyjrzysz to zauważysz, że w programie mamy dwie etykiety: **poczatek** i **koniec**. Tak naprawdę etykieta **poczatek** nie jest w ogóle potrzebna, bowiem nigdzie jej nie używamy. Mimo to została zapisana, aby udowodnić Ci, że sama etykieta nie zmienia nic w wykonaniu programu - zmiany powoduje dopiero instrukcja **goto**.

Gdybyśmy w powyższym programie nie zastosowali instrukcji **goto**, to wówczas końcowa wartość zmiennej **i** wyniosłaby 0. Ponieważ jednak używamy instrukcji skoku, omijając w ten sposób 2 linijki, to okazuje się, że wartość zmiennej i wynosi 6.

# Instrukcja skoku goto - nie używaj!

Już teraz pragnę zwrócić Twoją uwagę na jedną kwestię: instrukcja skoku **goto** jest w zasadzie jedyną znaną mi instrukcją, która co prawda została wprowadzona do języka, jednak **jej używanie jest uważane za wyjątkowo nieeleganckie**.

Tak naprawdę program, w którym instrukcja goto została użyta, jest często uznawany przez bardziej zaawansowanych lub zawodowych programistów za pisany przez zupełnie początkującego. Dlatego jeśli tylko to możliwe, wystrzegaj się używania tej instrukcji.

W rzeczywistości instrukcję skoku **goto** wypada jedynie użyć w przypadku zagnieżdżonych pętli, jeśli chcemy przerwać wszystkie pętle (lub kilka z nich), bowiem jak wiesz za pomocą instrukcji **break** nie da się tego osiągnąć (chyba, że użyjesz sposobu pokazanego przeze mnie w poprzedniej lekcji).

Nie należy natomiast w żadnym wypadku stosować tej instrukcji do "zwykłych" zastosowań, bowiem zrozumienie programów napisanych z użyciem instrukcji skoku **goto** jest bardzo trudne, a czasami wręcz niemożliwe.

Właśnie takim przykładem nadużycia instrukcji skoku jest przedstawiony przed momentem program - w rzeczywistym świecie programistów, taki program nie ma się prawa pojawić, no chyba, że jest to rzeczywiście uzasadnione (a w 99% nie jest).

# Przykłady użycia

Poniżej przedstawiam jeszcze 2 przykłady użycia instrukcji skoku **goto** - mam nadzieję, że dzięki nim uda Ci się przekonać, że naprawdę instrukcji tej należy używać tak rzadko jak tylko to możliwe.

```
#include <iostream>

using namespace std;

int main()
{

  for (unsigned i=0;i<3;++i)
  {
     cout <<"zewnetrzna dla i="<<i<<'\n';
     for (unsigned int j=0;j<2;++j)
     {
        if (i==1 && j==1)
           goto koniec;
        cout <<"wewnetrzna dla j="<<j<<'\n';
     }
  }
koniec:
  cout <<"Tekst kontrolny. Wyszlismy z zagniezdzonej petli\n";
  // tutaj oczywiscie moze byc dalsza czesc programu

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jeśli się zastanowisz, to przypomnisz sobie, że jest to ten sam przykład, który pojawił się w przypadku instrukcji **break**. Tutaj problem jest po prostu zrealizowany w prostszy sposób.

Użycie instrukcji skoku **goto** w takim wypadku na pewno nie będzie dla Ciebie powodem do wstydu. Nawet bardzo duże firmy programistyczne, tworzące kompilatory lub inne

oprogramowanie, właśnie w przypadku zagnieżdżonych pętli używają właśnie instrukcji **goto**.

Teraz natomiast czas na bardzo zły przykład. Przyjrzyj się kodowi programu - nie uruchamiaj go ani też nie czytaj komentarza pod programem i postaraj się zgadnąć, co on robi. Dopiero wtedy uruchom program i czytaj dalej artykuł. Oto wspomniany program:

```
#include <iostream>

using namespace std;

int main()
{
  int i=0;
poczatek:
  cout <<"Zmienna i ma wartosc "<<i<<'\n';
  ++i;
  if (i<10)
     goto poczatek;

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Po uruchomieniu programu na pewno już wiesz, że powyższy przykład spełnia taką samą rolę jak pętla, w której zmienna sterująca przyjmuje wartości 1, 2, ..,8, 9. Doskonale wiesz, że taki przykład można zrealizować za pomocą pętli for, while oraz do while.

Po co więc stosować instrukcję goto? No właśnie. Nie należy tego w takim wypadku robić! Nawet najprostszy program można strasznie skomplikować, stosując etykiety i instrukcję skoku goto.

Dodatkowo sprawdź co by się stało, gdyby dwie linijki: **int i=0;** i **poczatek:** zostały zamienione - program się zapętli, bowiem za każdym razem jest tworzona nowa zmienna o wartości 0. Jak więc widzisz - taki prosty program, a używając instrukcji **goto** można doprowadzić do katastrofy.

Radzę Ci zatem stosować instrukcję skoku **goto** tylko wtedy, gdy jest to naprawdę uzasadnione, czyli w rzeczywistości tylko w przypadku przerywania działania zagnieżdżonych pętli.

# Podsumowanie

W tej lekcji przedstawiłem Ci dwie instrukcje: **continue** oraz **goto** - obie używane w pracy z pętlami i obie ułatwiające posługiwanie się nimi.

Na pewno warto znać obie te instrukcje, jednak jak już podkreślałem ich znajomość nie jest absolutnie konieczna do dalszej nauki języka C++.

# Lekcja 19: Operator warunkowy i instrukcja switch. Podejmowanie decyzji w języku C++

## Wprowadzenie

Podejmowanie decyzji w programach jest bardzo ważną kwestią. Niewątpliwie, gdyby nie możliwość podejmowania decyzji, tak naprawdę żadne programy by nie istniały, bowiem wówczas niemożliwe by było sprawdzenie, co użytkownik nacisnął albo jaką opcję wybrał.

Ty oczywiście znasz już sposób podejmowania decyzji w języku C++ - służy do tego instrukcja warunkowa **if**. Mimo to, warto poznać jeszcze 2 inne mechanizmy, które również pozwalają na dokonywanie decyzji w sposób często identyczny jak instrukcja warunkowa **if**.

## Operator warunkowy

Jak łatwo można się domyśleć z samej nazwy, **operator warunkowy** pełni podobne zadanie jak instrukcja warunkowa if - umożliwia **podejmowanie pewnych decyzji w zależności od pewnego warunku**.

Chcę jednak zwrócić uwagę na poprawne nazewnictwo - to jest **operator warunkowy** a nie instrukcja warunkowa. Jedyną instrukcją warunkową w języku C++ jest instrukcja **if**.

**Schematyczna** postać operatora warunkowego wygląda następująco

wyrazenie1 ? wyrazenie2 : wyrazenie3;

Najpierw jest sprawdzane czy **wyrazenie1** jest prawdziwe czy nie. Jeśli jest ono prawdziwe, to obliczana jest wartość **wyrazenie2** i jest ona zwracana. Jeśli natomiast **wyrazenie1** było fałszywe, to obliczana jest wartość **wyrazenie3** i ta wartość jest zwracana.

To, że wartość jest zwracana oznacza, że wynik operacji może zostać przypisany do jakiejś zmiennej, czyli przykładowo możemy napisać tak:

zmienna = wyrazenie1 ? wyrazenie2 : wyrazenie3;

W takiej sytuacji, jeśli **wyrazenie1** będzie prawdą to zmienna będzie miała wartość **wyrazenie2**, natomiast jeśli **wyrazenie1** będzie fałszywe, to zmienna będzie miała wartość **wyrazenie3**.

W rzeczywistości najczęściej w ten właśnie sposób jest wykorzystywany operator warunkowy - wynik działania jest przypisywany do jakiejś konkretnej zmiennej.

## Operator warunkowy - przykłady

Poniżej przedstawię programy z wykorzystaniem operatora warunkowego. Jednocześnie przedstawię te same programy w użyciem instrukcji warunkowej **if**, aby udowodnić Ci, że za pomocą instrukcji if można uzyskać to samo.

**Kod 1 z użyciem operatora warunkowego:**

```
#include <iostream>

using namespace std;

int main()
{
  int a=3, b=2;
  (a<b) ? cout <<"a jest mniejsze" : cout <<"b jest mniejsze";

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

```
Kod 1 z użyciem instrukcji warunkowej if:
#include <iostream>

using namespace std;

int main()
{
  int a=3, b=2;
  if (a<b)
     cout <<"a jest mniejsze";
  else
     cout <<"b jest mniejsze";

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Zwróć uwagę, że oba programy działają oczywiście tak samo. Ponadto zauważ, że w pierwszym programie po pierwszej instrukcji wypisania komunikatu na ekran nie ma średnika (nie może go tam być).

**Kod 2 z użyciem operatora warunkowego:**

```
#include <iostream>

using namespace std;

int main()
{
  int a=3, b=2, mniejsza;
  mniejsza = (a<b) ? a : b;
  cout <<"Wartosc mniejszej liczby wynosi "<<mniejsza<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

```
Kod 2 z użyciem instrukcji warunkowej if:
#include <iostream>

using namespace std;

int main()
{
  int a=3, b=2, mniejsza;
  if (a<b)
     mniejsza=a;
```

```
  else
     mniejsza=b;
  cout <<"Wartosc mniejszej liczby wynosi "<<mniejsza<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Również tym razem w obu wypadkach otrzymujemy dokładnie ten sam rezultat. Oczywiście za pomocą instrukcji warunkowej można tworzyć zagnieżdżone warunki, myślę jednak, że taki zapis nie będzie ani zbyt łatwy do zrozumienia ani zbyt przydatny.

Przyznam się szczerze, że prywatnie nie stosuję prawie w ogóle operatora warunkowego - zamiast niego stosuję zawsze instrukcję warunkową, bowiem jest ona znacznie bardziej uniwersalna (da się więcej osiągnąć), a ponadto zapis jest znacznie łatwiejszy, również do rozbudowy.

# Operator warunkowy - podsumowanie

Jak udało Ci się zauważyć, za pomocą operatora warunkowego można często osiągnąć te same rezultaty, co za pomocą instrukcji warunkowej if. To, czy zdecydujesz się używać operatora warunkowego, zależy tylko od Ciebie. Nawet jeśli się na to zdecydujesz, to i tak używanie instrukcji warunkowej if będzie dla Ciebie koniecznością.

# Instrukcja switch

Instrukcja switch służy do **podejmowania decyzji warunkowych**. Tak naprawdę bez tej instrukcji można się obejść, bowiem instrukcja ta niejako ułatwia nam tylko zapis w przypadku wielu warunków (pisania jest tak samo dużo jak w wypadku instrukcji if, jednak w wielu wypadkach kod jest znacznie czytelniejszy).

Nie ma sztywnej zasady kiedy należy stosować instrukcję switch, a kiedy kilka instrukcji if - else. Wszystko zależy od indywidualnych upodobań, jednak intuicyjnie warto zastosować instrukcję **switch** kiedy mamy więcej niż 3 warunki.

# Instrukcja switch - schematy

**Schematyczna** postać instrukcji switch wygląda następująco:

```
switch (wyrazenie)
{
  case wartosc1:
        listaInstrukcji1;
  case wartosc2:
        listaInstrukcji2;

  ...

  case wartoscN:
        listaInstrukcjiN;
  default:
        listaInstrukcjiDomyslna;
}
```

Przede wszystkim każda wartość znajdująca się po słowie **case** musi być wyrażeniem typu całkowitego (czyli int lub char) i musi to być stała liczba, tzn. nie można tutaj zapisać zmiennej.

Dodatkowo, sekcji **case** może być tyle ile chcemy, ale żadne z wartości nie mogą się powtarzać.

Jeśli żadna z wybranych przez nas wartości się nie pojawi, wówczas zostanie wykonana lista instrukcji po słowie **default**.

Co prawda sekcja **default** nie jest w ogóle obowiązkowa, to jednak zawsze warto ją dodać, aby mieć kontrolę nad programem i przewidzieć nawet najdziwniejsze błędy programu.

Zasada działania jest taka, że jest wykonywana lista instrukcji począwszy od listy instrukcji znajdującej się w sekcji case, dla której wyrażenie ma daną wartość. Przykładowo, jeśli w powyższym schemacie **wyrazenie** ma wartość **wartosc2**, to wówczas wykona się lista instrukcji **począwszy od listaInstrukcji2**. Wykona się zatem nie tylko **listaInstrukcji2**, ale również **listaInstrukcji3**, ..., **listaInstrukcjiN** i **listaInstrukcjiDomyslna**.

Oczywiście o takie działanie rzadko kiedy nam chodzi (naprawdę bardzo rzadko) i taka postać instrukcji **switch** by nam się prawie w ogóle nie przydała.

Dlatego też każdą listę instrukcji należy zakończyć poznaną już przez Ciebie instrukcją **break** - wówczas instrukcja **switch** będzie się zachowywała tak jak grupa instrukcji warunkowych if. Instrukcja break w takim wypadku nie jest jedynie konieczna w sekcji **default**, bowiem nic się już za nią nie znajduje.

Zatem **schemat**, jakiego oczekujemy i jakiego będziesz używać prawie zawsze w przypadku instrukcji **switch** wygląda następująco:

```
switch (wyrazenie)
{
  case wartosc1:
        listaInstrukcji1;
        break;
  case wartosc2:
        listaInstrukcji1;
        break;
  ...

  case wartoscN:
        listaInstrukcjiN;
        break;
  default:
        listaInstrukcjiDomyslna;
}
```

Teraz dodam, że listy instrukcji nie muszą w tym wypadku być brane w nawiasy klamrowe, nawet gdy instrukcji jest kilka, bowiem kompilator łatwo się domyśla gdzie jest koniec listy instrukcji - jest on tam, gdzie pojawia się kolejne słowo **case**.

W powyższym schemacie, wszystko wykona się dokładnie tak, jak tego byśmy oczekiwali. Jeśli **wyrazenie** ma wartość **wartosc2**, to zostanie wykonana jedynie **listaInstrukcji2** i nic więcej.

Żeby nie było żadnych niedomówień, oba przedstawione schematy instrukcji switch są zupełnie poprawne z punktu widzenie składni języka C++ - pierwszy schemat jest tylko po prostu zazwyczaj mniej przydatny i w 99% stosuje się schemat z wykorzystaniem instrukcji **break**.

# Instrukcja switch - przykład użycia

Poniżej przedstawię jeden przykład użycia instrukcji **switch**. Program będzie bardzo prosty i zostanie wykorzystany oczywiście drugi, ten bardziej przydatny schemat postaci instrukcji switch. Oto program ilustrujący działanie instrukcji switch:

```cpp
#include <iostream>

using namespace std;

int main()
{
  int liczba;
  cout <<"Podaj liczbe calkowita: ";
  cin >>liczba;
  cin.ignore();

  switch (liczba)
  {
     case 48: // jak widzisz jest to stala calkowita a nie zadna zmienna
        cout <<"Liczba wynosi 48";
        break;
     case 60:
        cout <<"Liczba wynosi 60\n";
        cout <<"Liczba nie wynosi 48";
        break;
     case 78:
        cout <<"Liczba wynosi 78";
        break;
     default:
        cout <<"Podana liczba jest rozna od 48, 60 i 78\n";
        cout <<"Liczba wynosi "<<liczba<<'\n';
  }

  cout <<"\n\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jak widzisz program działa zgodnie z oczekiwaniami. Spróbuj zmienić typ zmiennej **liczba** np. na float i program się nie skompiluje, bo w instrukcji switch musimy mieć liczby całkowite (wcześniej juz o tym wspomniałem). Podobnie, jeśli utworzysz dodatkową zmienną i napiszesz **case nowaZmienna**, to program również się nie skompiluje, bowiem to nie jest stała.

Mam nadzieję, że ten jeden przykład wystarczył Ci do zrozumienia instrukcji switch, bo tak naprawdę nie ma tu nic trudnego. Oczywiście instrukcje switch można w sobie zagnieżdżać, jednak niezbyt często stosuje się taki zapis, a jeśli Ci naprawdę na tym zależy, to spróbuj na własną rękę napisać taki program.

# Podsumowanie

W tej lekcji udało Ci się poznać operator warunkowy i instrukcję switch. Po raz kolejny, bez tych konstrukcji (zwłaszcza tej pierwszej) można się obyć, warto jednak znać tak podstawowe mechanizmy języka C++.

# Lekcja 20: Typ logiczny i typ wyliczeniowy. Kolejne typy danych w języku C++

## Wprowadzenie

Znasz już podstawowe typy danych i operatory. W ostatnim czasie przedstawiłem Ci tablice, pętle i wiele różnych przydatnych instrukcji. W tej lekcji poznasz 2 kolejne typy danych, z czego jeden jest bardzo ważny i używasz go nieświadomie niemal przez cały czas trwania tego kursu.

## Typ logiczny

Typ logiczny jest tak naprawdę jednym z najprostszych typów języka C++. Zmienna typu logicznego może bowiem przyjmować tylko dwie wartości: prawdę albo fałsz.

Typ logiczny w C++ to:

**bool** - typ logiczny (przyjmuje tylko 2 wartości: **true** lub **false**)

Typ logiczny jest w rzeczywistości bardzo często wykorzystywany, nawet jeśli nie używaliśmy do tej pory zmiennej typu **bool**. Tak naprawdę w każdym wyrażeniu logicznym, kiedy mamy operatory **&&** lub || oraz kiedy używamy instrukcji warunkowej lub operatora warunkowego, tam tak naprawdę jest wykorzystywany typ logiczny.

Zacznijmy jednak od początku. Oto prosty przykład, w którym tworzymy zmienną typu **bool** i wypisujemy jej wartość na ekran.

```
#include <iostream>

using namespace std;

int main()
{
  bool logiczny;

  logiczny=true;
  cout <<"Zmienna logiczny ma wartosc "<<logiczny<<'\n';

  logiczny=false;
  cout <<"Zmienna logiczny ma teraz wartosc "<<logiczny<<'\n';

  logiczny=2;
  cout <<"Zmienna logiczny ma teraz wartosc "<<logiczny<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

W przykładowym programie można tak naprawdę dostrzec wiele wydawałoby się dziwnych rzeczy. Przede wszystkim, kiedy zmienna ma wartość true, na ekran zostaje wypisane 1, a kiedy zmienna ma wartość false na ekran zostaje wypisane 0.

Faktycznie - tak zostało przyjęte w języku C++, jednak w rzeczywistości to nie będzie miało dla Ciebie żadnego znaczenia. Tak naprawdę poza celami poznawczymi, tak jak tutaj, nikt nie wypisuje wartości zmiennej typu logicznego na ekran, bo nie jest to tak naprawdę w żaden sposób uzasadnione.

Dziwniejszą sprawą jest to, że zmiennej zostaje przypisana wartość liczbowa **2**, mimo że wspominałem, że zmienna typu logicznego może przyjmować tylko dwie wartości: false albo true. Rzeczywiście tak właśnie jest, jednak dokonując przypisania jakiejkolwiek wartości niezerowej do zmiennej logicznej, zostaje ona potraktowana jako prawda, czyli **true** (Tylko 0 jest traktowane jako fałsz, czyli **false**).

Potwierdza to wynik po wypisaniu na ekran - zmienna nie ma bowiem wartości 2, tylko 1, a 1 jak już wspomniałem przy wypisaniu wartości zmiennej typu logicznego, oznacza po prostu **true**. Jak więc widzisz wszystko się zgadza.

# Typ logiczny a instrukcje warunkowe

W rzeczywistości wszędzie tam, gdzie są stosowane instrukcje warunkowe if, operatory warunkowe, czy instrukcja switch, tam niejawnie jest stosowany typ logiczny.

Przykładowo, wyrażenie **2<3** zwraca wartość logiczną **true**, a wyrażenie **5!=5** zwraca wartość logiczną **false**. Właśnie tego typu wyrażenia są stosowane we wspomnianych przeze mnie konstrukcjach. Na dodatek, również we wszystkich typach pętli jako warunki końcowe pojawiają się wyrażenia, które w rzeczywistości zwracają **true** lub **false**.

Mam nadzieję, że tym sposobem przekonałem Cię, że typ logiczny jest bardzo powszechnie stosowany w języku C++. Jest on zarazem bardzo prostym typem, więc nie ma co dłużej zastanawiać się nad jego znaczeniem, tylko po prostu świadomie go używać w swoich programach.

Jeśli spojrzysz na program znajdujący się w lekcji Instrukcja break, to zwróć uwagę na zmienną **end**. To właśnie typowa zmienna typu logicznego, chociaż wtedy użyliśmy typu **unsigned int** z prostego powodu, że typ logiczny nie był Ci wtedy jeszcze znany.

Postaraj się teraz zmienić program tak, aby został użyty typ logiczny a działanie się nie zmieniło. Zmień również przypisania wartości na wartości typowe dla typu logicznego (true i false zamiast 1 i 0).

# Typ wyliczeniowy - wprowadzenie

Wszystkie typy, które do tej pory udało Ci się poznać charakteryzowała jedna wspólna cecha - do zmiennej tego typu mogliśmy przypisać **wszystkie dopuszczalne wartości dla danego typu**. Dla typu całkowitego mogliśmy przypisać dowolną liczbę całkowitą z pewnego zakresu, a do zmiennej typu znakowego mogliśmy przypisać dowolny znak.

Oczywiście takie podejście może wydawać Ci się całkiem poprawne. Rzeczywiście, w wielu wypadkach jest to pożądane, a wręcz idealne rozwiązanie. Co jednak, gdybyśmy chcieli ograniczyć ten zbiór wartości? Co gdybyśmy chcieli na przykład stworzyć **nowy typ**, dla którego **są dopuszczalne tylko 3 wartości**?

Po pierwsze, takie podejście może być uzasadnione. Powiedzmy, że chcemy na przykład napisać program o różnych pojazdach i uznajemy, że w programie chcemy mieć pojazdy należące do jednego z trzech typów (np. bezsilnikowe, jednosilnikowe i wielosilnikowe). Nie chcemy mieć możliwości przypisywania do typu pojazdu żadnej innej wartości.

Możemy podejść w różny sposób do problemu. Na przykład możemy założyć, że będziemy zawsze pamiętać, że nie wolno nam przypisać do pewnej zmiennej typu np. całkowitego innych wartości poza 0 (bezsilnikowe), 1 (jednosilnikowe) i 2(wielosilnikowe). Nie muszę Cię chyba zapewniać, że jest to podejście zupełnie nie na miejscu, bowiem nawet Ty w końcu zapomnisz, co znaczy 0, co 1 a co 2 i w końcu przypiszesz do zmiennej inną wartość np. 3. Trochę innym podejściem byłoby sprawdzanie za pomocą instrukcji warunkowej, czy przypisujemy do zmiennej dopuszczalną wartość, a jeśli nie, to podjęcie pewnych środków zaradczych. To rozwiązanie oczywiście jest nieco

lepsze, jednak nadal nie byłoby tym czego oczekujemy.

Przede wszystkim musielibyśmy wszędzie za pomocą instrukcji warunkowej sprawdzać, czy wartość jest dopuszczalna dla naszego "nowego typu" - nie muszę Cię przekonywać, że zaciemniłoby to w znacznym stopniu kod Twojego programu.

Dodatkowo, nadal zmiennej przypisywalibyśmy zwykłe liczby typu 0, 1, 2, co przy pierwszym spojrzeniu na kod Twojego programu, nie znaczy oczywiście zupełnie nic. Należałoby stworzyć gdzieś na początku programu komentarz, że 0 to pojazdy bezsilnikowe, 1 to pojazdy jednosilnikowe, a 2 to pojazdy wielosilnikowe.

Tak naprawdę za każdym razem, kiedy Ty lub ktoś inny chciałby dodać coś do programu, musiałby sobie ponownie przypominać co znaczy 0, co znaczy 1, a co znaczy 2.

Typ wyliczeniowy pozwoli Ci natomiast **tworzyć nowe typy danych** i to bez większego wysiłku. Dodatkowo, całą **odpowiedzialność za poprawność typu danych przerzucisz na kompilator**!

# Typ wyliczeniowy - podstawy

Wiesz już w przybliżeniu do czego służy typ wyliczeniowy - za jego pomocą tworzysz typy danych, dla których zmienne mogą przyjmować tylko pewną określoną liczbę wartości. To kompilator a nie Ty zatroszczy się, aby nie przypisać do zmiennej nowego typu żadnej innej wartości.

**Schematycznie** definicję typu wyliczeniowego zapisujemy tak:

enum nazwaTypu {pierwszaWartosc, drugaWartosc, ..., ostatniaWartosc};

Od razu zaznaczę, że jest to uproszczona definicja typu, jednak w naszym przykładzie dotyczącym pojazdów, taka definicja mogłaby w zupełności wystarczyć.

**nazwaTypu** to wymyślona przez nas nazwa typu, a wartości to nazwy dopuszczalnych wartości dla naszego typu, ale muszą to być zwykłe nazwy - takie jak nazwy zmiennych. Niestety nie możemy tutaj napisać 0, 1 czy 2, bowiem nie są to poprawne nazwy zmiennych (a w rzeczywistości poprawne nazwy identyfikatorów).

Poniżej przedstawiam przykładowy program z użyciem typu wyliczeniowego:

```
#include <iostream>

using namespace std;

int main()
{
  // utworzenie typu
  enum typPojazdu {bezsilnikowe, jednosilnikowe, wielosilnikowe};

  // utworzenie zmiennej noweg typu
  typPojazdu mojPojazd; // lub enum typPojazdu mojPojazd;

  mojPojazd=bezsilnikowe; // przypisanie wartosci zmiennej

  if (mojPojazd==bezsilnikowe)
     cout <<"Twoj pojazd nie ma silnika.\n";
  else
     cout <<"Twoj pojazd ma silnik.\n";

  mojPojazd=wielosilnikowe; // przypisanie nowej wartosci zmiennej

  if (mojPojazd==jednosilnikowe)
     cout <<"A teraz Twoj pojazd ma tylko jeden silnik.\n";
  else
     cout <<"A teraz Twoj pojazd ma wiecej niz jeden silnik.\n";
```

```
  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jak widzisz, program jest całkiem prosty - na początku tworzymy nasz typ wyliczeniowy o nazwie **typPojazdu**z trzema dopuszczalnymi wartościami. Później tworzymy zmienną naszego nowego typu (zauważ, że w tym miejscu można, choć nie trzeba dodać po raz drugi słowo **enum**).

Następnie, jak widzisz, dokonujemy przypisania wartości do zmiennej. Spróbuj przypisać do zmiennej jakiekolwiek inne wartości (poza tymi trzema wymienionymi w definicji typu) i zobaczysz, że kompilator zaprotestuje. Zauważ też, że wszystkie inne operacje są jak w przypadku zwykłego typu - możemy bez problemu dokonywać chociażby operacji porównania.

# Typ wyliczeniowy a typ całkowity

Zacznijmy od przykładu - zmodyfikuję teraz nieco nasz początkowy program, aby ukazać Ci pewne zjawisko:

```
#include <iostream>

using namespace std;

int main()
{
  enum typPojazdu {bezsilnikowe, jednosilnikowe, wielosilnikowe};
  typPojazdu mojPojazd;

  mojPojazd=bezsilnikowe;
  cout <<"Pojazd teraz to "<<mojPojazd<<'\n';
  mojPojazd=jednosilnikowe;
  cout <<"Pojazd teraz to "<<mojPojazd<<'\n';
  mojPojazd=wielosilnikowe;
  cout <<"Pojazd teraz to "<<mojPojazd<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

Jak widzisz, w programie usiłujemy wypisać wartość naszej zmiennej i za każdym razem otrzymujemy liczby całkowite: 0, 1 i 2.

Od razu chcę zaznaczyć, że to zjawisko jest zupełnie inne niż zjawisko występujące w przypadku typu logicznego. Tam również, gdy chcieliśmy wypisać wartość zmiennej logicznej, otrzymywaliśmy na ekr

O ile jednak w tamtym wypadku tak zostało przyjęte, że prawda zostanie wypisana jako 1, a fałsz jako 0, to tutaj to nie jest żadne przyjęcie podczas wypisania - taka jest rzeczywista wartość zmiennej!

Jeśli zastanawia Cię, dlaczego tak jest, to napiszę krótko - typ wyliczeniowy jest w rzeczywistości typem całkowitym! Zapamiętaj to sobie dobrze, bowiem wynika z tego kilka kwestii.

Teraz czas przedstawić pełny **schematyczny** sposób definicji typu wyliczeniowego. A wygląda on następująco:

enum nazwaTypu {nazwa1=wartosc1, nazwa2=wartosc2, ..., nazwaN=wartoscN};

Jak zatem możesz się już domyślić, w poprzednim wypadku podaliśmy jedynie nazwy

dopuszczalnych wartości typu, a nie podaliśmy ich rzeczywistych wartości. Oznaczone tutaj wartosc1, wartosc2 itd. to **wartości całkowite** - zatem każdej naszej wartości typu w rzeczywistości odpowiada pewna wartość całkowita.

Zanim wyjaśnię Ci do czego może posłużyć rzeczywista wartość całkowita, postaram Ci się przybliżyć kilka reguł dotyczących przypisywania wartości w typie numerycznym.

Po pierwsze, **nie musimy koniecznie określać wartości całkowitych** (tak właśnie zrobiliśmy w pierwszym przykładzie), bowiem może to nam nie być do niczego potrzebne.

Wartości przypisane poszczególnym wartościom naszego typu mogą być **tylko całkowite** - niestety nie można przypisywać tam wartości zmiennoprzecinkowych, a wszystko to dlatego, że typ wyliczeniowy jest typem całkowitym (a nie na przykład typem float).

Jeśli nie określimy początkowej wartości całkowitej, to **domyślnie wyliczanie zacznie się od 0**.

**Każda następna wartość będzie o 1 większa**, o ile nie określimy inaczej. Dlatego też jeśli nie określimy w ogóle wartości to pierwsza wartość 0, a każda następna jest o 1 większa, czyli 1, 2, 3 itd.

Możemy w **dowolnym miejscu określić rzeczywistą wartość całkowitą**, pamiętając, że jeśli dla następnej wartości nie będzie określona rzeczywista wartość całkowita, to wówczas będzie ona o 1 większa od podanej jawnie.

Rzeczywiste **wartości całkowite mogą się powtarzać** - zazwyczaj jednak i tak określisz różne wartości, ale czasami ta cecha może się przydać.

Teraz trochę przykładów, żeby zobrazować Ci przedstawione powyżej reguły i upewnić się, że wszystko rozumiesz. Nie będę już ich komentował:

```
enum mojTyp {a, b, c, d, e, f, g}; // wartosci: 0, 1, 2, 3, 4, 5, 6
enum mojTyp {a=3, b, c, d, e, f, g}; // wartosci: 3, 4, 5, 6, 7, 8, 9
enum mojTyp {a, b, c, d=0, e, f, g}; // wartosci: 0, 1, 2, 0, 1, 2, 3
enum mojTyp {a=3, b, c=3, d, e=2, f, g=7}; // wartosci: 3, 4, 3, 4, 2, 3, 7
enum mojTyp {a=3, b=3, c=3, d=3, e=3, f=3, g=3}; // wartosci: 3, 3, 3, 3, 3, 3,
3
enum mojTyp {a=60, b=0, c, d, e, f, g}; // wartosci: 60, 0, 1, 2, 3, 4, 5
```

Mam nadzieję, że powyższe przykłady rozjaśniły Ci, w jaki sposób są wyznaczane wartości dla poszczególnych elementów.

Mimo, że jak już wiesz, zmienna typu wyliczeniowego tak naprawdę przyjmuje rzeczywistości całkowite, to jednak nie będziemy mogli przypisać do niej wartości całkowitej. Jest to właśnie tą dużą zaletą - osiągnęliśmy to co chcieliśmy.

W powyższych przykładach nawet jeśli a miałoby wartość 0, to do zmiennej możemy przypisać tylko a, natomiast wartości 0 przypisać nie możemy. Dzięki temu uniknęliśmy przypisywania nic nie znaczących wartości i konieczności czytania komentarzy, co oznacza np. wartość 0 dla danego typu wyliczeniowego.

# Wykorzystanie wartości dla typu wyliczeniowego

Chcę Ci w końcu pokazać, do czego mogą się przydać również wartości całkowite zmiennych typu wyliczeniowego. W tym celu rozszerzymy nasz typ pojazdów i założymy, że dla podanej prędkości chcemy określić jaką część prędkości generuje jeden silnik.

W powyższym przykładzie zauważysz, że dokonując zwykłych operacji arytmetycznych typu dodawanie, mnożenie itd. (tutaj akurat dzielenie) zmiennej typu wyliczeniowego ze zwykłymi liczbami całkowitymi, taka zmienna zachowuje się właśnie jak zwykła liczba, czyli jej wartość

całkowita może zostać wykorzystana.

```
#include <iostream>

using namespace std;

int main()
{
  int ktory;
  int predkosc;
  float predkoscSilnika;
  enum typPojazdu { bezsilnikowe=0, jednosilnikowe, dwusilnikowe,
czterosilnikowe=4, wielosilnikowe=10};

  enum typPojazdu mojPojazd;

  cout <<"Nacisnij cyfre odpowiadajaca Twojemu typowi pojazdu i ENTER\n";
  // te wartosci tutaj NIE MUSZA sie zgadzac z wartosciami w typie wyliczeniowym
  cout <<"1 - Pojazd bezsilinikowy\n";
  cout <<"2 - Pojazd jednosilnikowy\n";
  cout <<"3 - Pojazd dwusilnikowy\n";
  cout <<"4 - Pojazd czterosinilnikowy\n";
  cout <<"5 - Pojazd wielosilnikowy\n";
  cin >>ktory;
  cin.ignore();
  switch (ktory)
  {
     case 1: mojPojazd=bezsilnikowe; break;
     case 2: mojPojazd=jednosilnikowe; break;
     case 3: mojPojazd=dwusilnikowe; break;
     case 4: mojPojazd=czterosilnikowe; break;
     case 5: mojPojazd=wielosilnikowe; break;
     default: mojPojazd=bezsilnikowe;
  }
  cout <<"Podaj predkosc Twojego pojazdu (wieksza od zera): ";
  cin >>predkosc;
  cin.ignore();

  if (mojPojazd==bezsilnikowe)
     predkoscSilnika=0;
  else
     predkoscSilnika=predkosc/mojPojazd;

  if (predkoscSilnika==0)
     cout <<"Gratulacje. Cala predkosc "<<predkosc
          <<" tworzysz swoimi nogami\n";
  else
     cout <<"Predkosc jednego silnika to okolo "<<predkoscSilnika
          <<" bo "<<predkoscSilnika<<'*'<<mojPojazd<<'='<<predkosc<<'\n';

  cout <<"\nNacisnij ENTER aby zakonczyc...\n";
  getchar();
  return 0;
}
```

# Podsumowanie

W tej lekcji przedstawiłem Ci dwa nowe typy danych: typ logiczny i typ wyliczeniowy. Oba typy są ważne w języku C++, a bez typu logicznego tak naprawdę nie da się w ogóle obejść. Dlatego jeśli nie wszystko udało Ci się zrozumieć, zachęcam do ponownego przeczytania tej lekcji.