break, umieszczonej na końcu każdej pozycji rozpoczętej przez case. Polecenie to powoduje bowiem **przerwanie** działania konstrukcji switch i wyjście z niej; tym sposobem zapobiega ono wykonaniu kodu odpowiadającego następnym wariantom.

W większości przypadków **należy** zatem **kończyć** fragment kodu rozpoczęty przez case instrukcją break - gwarantuje to, iż tylko jedna z możliwości ustalonych w switch zostanie wykonana.

Znaczenie ostatniej, nieobowiązkowej frazy default wyjaśniliśmy sobie już wcześniej. Można jedynie dodać, że pełni ona w switch podobną rolę, co else w if i umożliwia wykonanie jakiegoś kodu także wtedy, gdy **żadna** z przewidzianych *wartości* nie będzie zgadzać się z *wyrażeniem*. Brak tej instrukcji będzie zaś skutkować niepodjęciem żadnych działań w takim przypadku.

***

Omówiliśmy w ten sposób obie konstrukcje, dzięki którym można sterować przebiegiem programu na podstawie ustalonych warunków czy też wartości wyrażeń. Potrafimy więc już sprawić, aby nasze aplikacje zachowywały się prawidłowo niezależnie od okoliczności. Nie zmienia to jednak faktu, że nadal potrafią one co najwyżej tyle, ile mało funkcjonalny kalkulator i nie wykorzystują w pełni w ogromnych możliwości komputera. Zmienić to może kolejny element języka C++, który teraz właśnie poznamy. Przy pomocy pętli, bo o nich mowa, zdołamy zatrudnić leniuchujący dotąd procesor do wytężonej pracy, która wyciśnie z niego siódme poty ;)

Pętle

Pętle (ang. *loops*), zwane też **instrukcjami iteracyjnymi**, stanowią podstawę prawie wszystkich algorytmów. Lwia część zadań wykonywanych przez programy komputerowe opiera się w całości lub częściowo właśnie na pętlach.

Pętla to element języka programowania, pozwalający na wielokrotne, kontrolowane wykonywanie wybranego fragmentu kodu.

Liczba takich powtórzeń (zwanych **cyklami** lub **iteracjami** pętli) jest przy tym ograniczona w zasadzie tylko inwencją i rozsądkiem programisty. Te potężne narzędzia dają więc możliwość zrealizowania niemal każdego algorytmu.
Pętle są też niewątpliwie jednym z atutów C++: ich elastyczność i prostota jest większa niż w wielu innych językach programowania. Jeżeli zatem będziesz kiedyś kodował jakąś złożoną funkcję przy użyciu skomplikowanych pętli, z pewnością przypomnisz sobie i docenisz te zalety :)

Pętle warunkowe do i while

Na początek poznamy dwie konstrukcje, które zwane są **pętlami warunkowymi**. Miano to określa całkiem dobrze ich zastosowanie: ciągłe wykonywanie kodu, dopóki spełniony jest określony **warunek**. Pętla sprawdza go przy każdym swoim cyklu - jeżeli stwierdzi jego fałszywość, natychmiast kończy działanie.

Pętla do

Prosty przykład obrazujący ten mechanizm prezentuje się następująco:

```
// Do - pierwsza pętla warunkowa
```

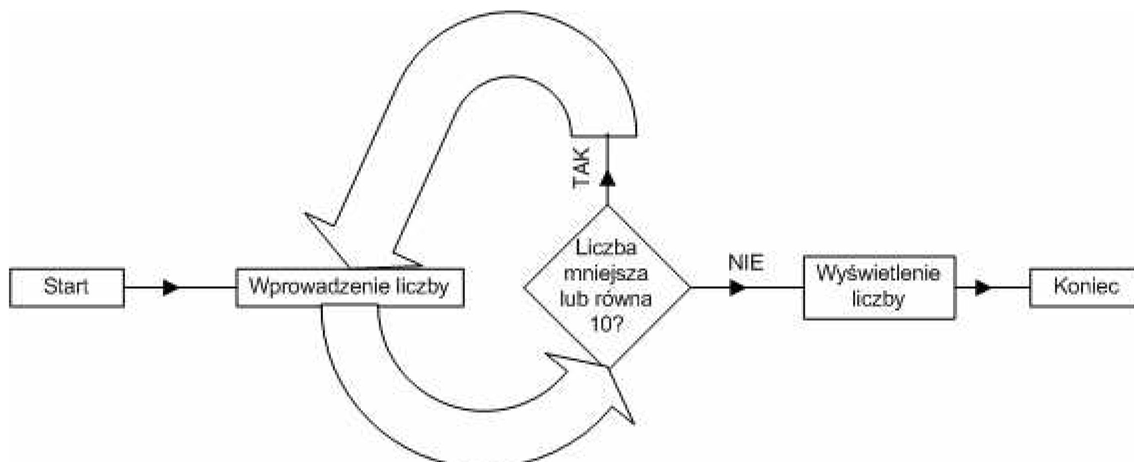
```
#include <iostream>
#include <conio.h>

void main()
{
   int nLiczba;

   do
   {
         std::cout << "Wprowadz liczbe wieksza od 10: ";
         std::cin >> nLiczba;
   } while (nLiczba <= 10);

   std::cout << "Dziekuje za wspolprace :)";
   getch();
}
```

Program ten, podobnie jak jeden z poprzednich, oczekuje od nas o liczby większej niż dziesięć. Tym razem jednak nie daje się zbyć byle czym - jeżeli nie będziemy skłonni od razu przychylić się do jego prośby, będzie ją niezłomnie powtarzał aż do skutku (lub do użycia Ctrl+Alt+Del ;D).

```
Wprowadz liczbe wieksza od 10: 5
Wprowadz liczbe wieksza od 10: 9
Wprowadz liczbe wieksza od 10: -12
Wprowadz liczbe wieksza od 10: 4
Wprowadz liczbe wieksza od 10: 7
Wprowadz liczbe wieksza od 10: 1
Wprowadz liczbe wieksza od 10: 14
Dziekuje za wspolprace :)
```

Screen 15. Nieugięty program przeciwko krnąbrnemu użytkownikowi :)

Upór naszej aplikacji bierze się oczywiście z umieszczonej wewnątrz niej pętli `do` ('czyń') . Wykonuje ona kod odpowiedzialny za prośbę do użytkownika tak długo, jak długo ten jest konsekwentny w ignorowaniu jej :) Przejawia się to rzecz jasna wprowadzaniem liczb, które **nie są** większe od 10, lecz mniejsze lub równe tej wartości – odpowiada to warunkowi pętli `nLiczba <= 10`. Instrukcja niniejsza wykonuje się więc dopóty, dopóki (ang. `while`) zmienna `nLiczba`, która przechowuje liczbę pobraną od użytkownika, nie przekracza granicznej wartości dziesięciu. Przedstawia to poglądowo poniższy diagram:



Schemat 4. Działanie przykładowej pętli `do`

Co się jednak dzieje przy pierwszym „obrocie” pętli, gdy program nie zdążył jeszcze pobrać od użytkownika żadnej liczby? Jak można porównywać wartość zmiennej `nLiczba`, która na samym początku jest przecież nieokreślona?… Tajemnica tkwi w fakcie, iż pętla `do` dokonuje sprawdzenia swojego warunku **na końcu** każdego cyklu – dotyczy to także pierwszego z nich. Wynika z tego dość oczywisty wniosek:

Pętla `do` wykona **zawsze** co najmniej jeden przebieg.

Fakt ten sprawia, że nadaje się ona znakomicie do uzyskiwania jakichś danych od użytkownika przy jednoczesnym sprawdzaniu ich poprawności. Naturalnie, w prawdziwym programie należałoby zapewnić swobodę zakończenia aplikacji bez wpisywania czegokolwiek. Nasz obrazowy przykład jest jednak wolny od takich fanaberii – to wszak tylko kod pomocny w nauce, więc pisząc go nie musimy przejmować się takimi błahostkami ;))

Podsumowaniem naszego spotkania z pętlą `do` będzie jej składnia:

```
do
{
   instrukcje
} while (warunek)
```

Wystarczy przyjrzeć się jej choć przez chwilę, by odkryć cały sens. Samo tłumaczenie wyjaśnia właściwie wszystko: „Wykonuj (ang. `do`) *`instrukcje`*, dopóki (ang. `while`) zachodzi *`warunek`*”. I to jest właśnie *spiritus movens* całej tej konstrukcji.

Pętla `while`

Przyszła pora na poznanie drugiego typu pętli warunkowych, czyli `while`. Słówko będące jej nazwą widziałeś już wcześniej, przy okazji pętli `do` – nie jest to bynajmniej przypadek, gdyż obydwie konstrukcje są do siebie bardzo podobne.
Działanie pętli `while` prześledzimy zatem na poniższym ciekawym przykładzie:

```
// While - druga pętla warunkowa

#include <iostream>
#include <ctime>
#include <conio.h>

void main()
{
   // wylosowanie liczby
   srand ((int) time(NULL));
   int nWylosowana = rand() % 100 + 1;
   std::cout << "Wylosowano liczbe z przedzialu 1-100." << std::endl;

   // pierwsza próba odgadnięcia liczby
   int nWprowadzona;
   std::cout << "Sprobuj ja odgadnac: ";
   std::cin >> nWprowadzona;

   // kolejne próby, aż do skutku - przy użyciu pętli while
   while (nWprowadzona != nWylosowana)
   {
         if (nWprowadzona < nWylosowana)
               std::cout << "Liczba jest zbyt mala.";
         else
               std::cout << "Za duza liczba.";
```

```
            std::cout << " Sprobuj jeszcze raz: ";
            std::cin >> nWprowadzona;
      }

      std::cout << "Celny strzal :) Brawo!" << std::endl;
      getch();
   }
```

Jest to nic innego, jak prosta… gra :) Twoim zadaniem jest w niej odgadnięcie „pomyślanej” przez komputer liczby (z przedziału od jedności do stu). Przy każdej próbie otrzymujesz wskazówkę, mówiącą czy wpisana przez ciebie wartość jest za duża, czy za mała.

```
   ZGADYWANKA
----------------

Wylosowano liczbe z przedzialu 1-100.
Sprobuj ja odgadnac: 56
Liczba jest zbyt mala. Sprobuj jeszcze raz: 78
Za duza liczba. Sprobuj jeszcze raz: 60
Liczba jest zbyt mala. Sprobuj jeszcze raz: 70
Za duza liczba. Sprobuj jeszcze raz: 65
Liczba jest zbyt mala. Sprobuj jeszcze raz: 67
Liczba jest zbyt mala. Sprobuj jeszcze raz: 69
Za duza liczba. Sprobuj jeszcze raz: 68
Celny strzal :) Brawo!
```

Screen 16. Wystarczyło tylko 8 prób :)

Tak przedstawia się to w działaniu. Jako programiści chcemy jednak zajrzeć do kodu źródłowego i przekonać się, w jaki sposób można było taki efekt osiągnąć. Czym prędzej więc ziśćmy te pragnienia :D

Pierwszą czynnością podjętą przez nasz program jest wylosowanie liczby, którą użytkownik będzie odgadywał. Zasadniczo odpowiadają za to dwie początkowe linijki:

```
srand ((int) time(NULL));
int nWylosowana = rand() % 100 + 1;
```

Nie będziemy obecnie zagłębiać się w szczegóły ich funkcjonowania, gdyż te zostaną omówione w następnym rozdziale. Teraz możesz jedynie zapamiętać, iż pierwszy wiersz, zawierający funkcję `srand()` (i jej osobliwy parametr), jest czymś w rodzaju zakręcenia kołem ruletki. Jego obecność sprawia, że aplikacja za każdym razem losuje nam inną liczbę.
Za samo losowanie odpowiada natomiast wyrażenie z funkcją `rand()`. Obliczona wartość tegoż jest od razu przypisywana do zmiennej `nWylosowana` i to o nią toczy bój niestrudzony gracz :)

Kolejny pakiet kodu pozwala na wykonanie pierwszej próby odgadnięcia właściwego wyniku. Nie widać tu żadnych nowości – z podobnymi fragmentami spotykaliśmy się już wielokrotnie i wyjaśniliśmy je dogłębnie. Zauważmy tylko, że liczba wpisana przez użytkownika jest zapamiętywana w zmiennej `nWprowadzona`.

O wiele bardziej interesująca jest dla nas pętla `while`, występująca dalej. To na niej spoczywa zadanie wyświetlania graczowi wskazówek, umożliwiania mu kolejnych prób i sprawdzania wpisanych wartości.
Podobnie jak w przypadku `do`, wykonywanie tej pętli uzależnione jest spełnieniem określonego kryterium. Tutaj jest nim niezgodność między liczbą wylosowaną na początku (zawartą w zmiennej `nWylosowana`), a wprowadzoną przez użytkownika

(zmienna `nWprowadzona`). Zapisujemy to w postaci warunku `nWprowadzona != nWylosowana`. Oczywiście pętla wykonuje się do chwili, w której założenie to przestaje być prawdziwe, a użytkownik poda właściwą liczbę.
Wewnątrz bloku pętli podejmowane zaś są dwie czynności. Najpierw wyświetlana jest podpowiedź dla użytkownika. Mówi mu ona, czy wpisana przed chwilą liczba jest większa czy mniejsza od szukanej. Gracz otrzymuje następnie kolejną szansę na odgadnięcie pożądanej wartości.
Gdy wreszcie uda mu się ta sztuka, raczony jest w nagrodę odpowiednim komunikatem :)

Tak oto przedstawia się funkcjonowanie powyższego programu przykładowego, którego witalną częścią jest pętla `while`. Wcześniej natomiast zdążyliśmy się dowiedzieć i przekonać, iż konstrukcja ta bardzo przypomina poznaną poprzednio pętlę `do`. Na czym więc polega różnica między nimi?…
Jest nią mianowicie **moment sprawdzania warunku** pętli. Jak pamiętamy, `do` czyni to na końcu każdego cyklu. Analogicznie, `while` dokonuje tego zawsze **na początku** swego przebiegu. Determinuje to dość oczywiste następstwo:

Pętla `while` może nie wykonać się **ani razu**, jeżeli jej warunek będzie od początku nieprawdziwy.

W naszym przykładowym programie odpowiada to sytuacji, gdy gracz od razu trafia we właściwą liczbę. Naturalnie, jest to bardzo mało prawdopodobne (rzędu 1%), lecz jednak możliwe. Trzeba zatem przewidzieć i odpowiednio zareagować na taki przypadek, zaś pętla `while` rozwiązuje nam ten problem praktycznie sama :)

Na koniec tradycyjnie już przyjrzymy się składni omawianej konstrukcji:

```
while (warunek)
{
    instrukcje
}
```

Ponownie wynika z niej praktycznie wszystko: „Dopóki (`while`) zachodzi *`warunek`*, wykonuj *`instrukcje`*”. Czyż nie jest to wyjątkowo intuicyjne? ;)

***

Tak oto poznaliśmy dwa typy pętli warunkowych – ich działanie, składnię i sposób używania. Tym samym dostałeś do ręki narzędzia, które pozwolą ci tworzyć lepsze i bardziej skomplikowane programy.
Jakkolwiek oba te mechanizmy mają bardzo duże możliwości, korzystanie z nich może być w niektórych wypadkach nieco niewygodne. Na podobne okazje obmyślono trzeci rodzaj pętli, z którym właśnie teraz się zaznajomimy.

Pętla krokowa `for`

Do tej pory spotykaliśmy się z sytuacjami, w których należało wykonywać określony kod aż do spełnienia pewnego warunku. Równie często jednak znamy wymaganą ilość „obrotów” pętli jeszcze **przed jej rozpoczęciem** – chcemy ją podać w kodzie *explicite* lub obliczyć wcześniej jako wartość zmiennej.
Co wtedy zrobić? Możemy oczywiście użyć odpowiednio spreparowanej pętli `while`, chociażby w takiej postaci:

```
int nLicznik = 1;
```