3

DZIAŁANIE PROGRAMU

Nic nie dzieje się wbrew naturze,
lecz wbrew temu, co o niej wiemy.
Fox Mulder w serialu „Z archiwum X"

Poznaliśmy już przedsmak uroków programowania w C++ i stworzyliśmy kilka własnych programów. Opanowaliśmy jednocześnie najbardziej podstawowe podstawy podstaw kodowania w tym języku :)

Możemy więc odkryć kolejne, ważne jego elementy, które pozwolą nam tworzyć bardziej interesujące i przydatne programy. Przyszła bowiem pora na spotkanie z instrukcjami sterującymi przebiegiem aplikacji i sposobem jej działania.

Funkcje nieco bliżej

Z funkcjami mieliśmy do czynienia już wcześniej. Powiedzieliśmy sobie wtedy, że są to wydzielone fragmenty kodu realizujące jakąś czynność. Przytoczyłem przy tym dość banalny przykład funkcji `PokazTekst()`, wyświetlającej w konsoli ustalony napis.

Niewątpliwie był on klarowny i ukazywał wspomniane tam przeznaczenie funkcji (czyli podział kodu na fragmenty), jednakże pomijał dwa bardzo ważne aspekty z nimi związane. Chodzi mianowicie o **parametry** oraz **zwracanie wartości**. Rozszerzają one użyteczność i możliwości funkcji tak znacznie, że bez nich w zasadzie trudno wyobrazić sobie skuteczne i efektywne programowanie.

W takiej sytuacji nie możemy wszak przejść obok nich obojętnie - niezwłocznie przystąpimy zatem do poznawania tych arcyważnych zagadnień :)

Parametry funkcji

Nie tylko w programowaniu trudno wskazać operację, którą można wykonać bez posiadania o niej dodatkowych informacji. Przykładowo, nie można wykonać operacji kopiowania czy przesunięcia pliku do innego katalogu, jeśli nie jest znana nazwa tegoż pliku oraz nazwa docelowego folderu.
Gdybyśmy napisali funkcję realizującą taką czynność, to nazwy pliku oraz katalogu finalnego byłyby jej **parametrami**.

Parametry funkcji to dodatkowe dane, przekazywane do funkcji podczas jej wywołania.

Parametry pełnią rolę dodatkowych zmiennych wewnątrz funkcji i można ich używać podobnie jak innych zmiennych, zadeklarowanych w niej bezpośrednio. Różnią się one oczywiście tym, że wartości parametrów pochodzą z „zewnątrz" – są im przypisywane podczas wywołania funkcji.

Po tym krótkim opisie czas na obrazowy przykład. Oto zmodyfikujemy nasz program liczący tak, żeby korzystał z dobrodziejstw parametrów funkcji:

```
// Parameters - wykorzystanie parametrów funkcji

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

void Dodaj(int nWartosc1, int nWartosc2)
{
   int nWynik = nWartosc1 + nWartosc2;
   std::cout << nWartosc1 << " + " << nWartosc2 << " = " << nWynik;
   std::cout << std::endl;
}

void main()
{
   int nLiczba1;
   std::cout << "Podaj pierwsza liczbe: ";
   std::cin >> nLiczba1;

   int nLiczba2;
   std::cout << "Podaj druga liczbe: ";
   std::cin >> nLiczba2;

   Dodaj (nLiczba1, nLiczba2);
   getch();
}
```

Rzut oka na działający program pozwala stwierdzić, iż wykonuje on taką samą pracę, jak jego poprzednia wersja. Z kolei spojrzenie na kod ujawnia w nim widoczne zmiany – przyjrzyjmy się im.

Zasadnicza czynność programu, czyli dodawanie dwóch liczb, została wyodrębniona w postaci osobnej funkcji `Dodaj()`. Posiada ona dwa parametry `nWartosc1` i `nWartosc2`, które są w niej dodawane do siebie i wyświetlane w konsoli.
Wielce interesujący jest w związku z tym nagłówek funkcji `Dodaj()`, zawierający deklarację owych dwóch parametrów:

```
void Dodaj(int nWartosc1, int nWartosc2)
```

Jak sam widzisz, wygląda ona bardzo podobnie do deklaracji zmiennych – najpierw piszemy typ parametru, a następnie jego nazwę. Nazwa ta pozwala odwoływać się do wartości parametru w kodzie funkcji, a więc na przykład użyć jej jako składnika sumy. Określenia kolejnych parametrów oddzielamy od siebie przecinkami, zaś całą deklarację umieszczamy w nawiasie po nazwie funkcji.

Wywołanie takiej funkcji jest raczej oczywiste:

```
Dodaj (nLiczba1, nLiczba2);
```

Podajemy tu w nawiasie kolejne **wartości**, które zostaną przypisane jej parametrom; oddzielamy je tradycyjnie już przecinkami. W niniejszym przypadku parametrowi `nWartosc1` zostanie nadana wartość zmiennej `nLiczba1`, zaś `nWartosc2` – `nLiczba2`. Myślę, że jest to dość intuicyjne i nie wymaga więcej wyczerpującego komentarza :)

***

Reasumując: parametry pozwalają nam przekazywać funkcjom dodatkowe dane, których mogą użyć do wykonania swoich działań. Ilość, typy i nazwy parametrów deklarujemy w nagłówku funkcji, zaś podczas jej wywoływania podajemy ich wartości w analogiczny sposób.

Wartość zwracana przez funkcję

Spora część funkcji pisanych przez programistów ma za zadanie obliczenie jakiegoś wyniku (często na podstawie przekazanych im parametrów). Inne z kolei wykonują operacje, które nie zawsze muszą się udać (choćby usunięcie pliku – dany plik może przecież już nie istnieć).
W takich przypadkach istnieje więc potrzeba, by funkcja **zwróciła** jakąś **wartość**. Niekiedy będzie to rezultat jej intensywnej pracy, a innym razem jedynie informacja, czy zlecona funkcji czynność została wykonana pomyślnie.

Zgodnie ze zwyczajem, popatrzymy teraz na odpowiedni program przykładowy[17] :) Pyta on użytkownika o długości boków prostokąta, wyświetlając w zamian jego pole powierzchni oraz obwód. Czyni to ten kod (jest to fragment funkcji `main()`):

```
// ReturnValue - funkcje zwracające wartość

int nDlugosc1;
std::cout << "Podaj dlugosc pierwszego boku: ";
std::cin >> nDlugosc1;

int nDlugosc2;
std::cout << "Podaj dlugosc drugiego boku: ";
std::cin >> nDlugosc2;

std::cout << "Obwod prostokata: " << Obwod(nDlugosc1, nDlugosc2) <<
std::endl;
std::cout << "Pole prostokata: " << Pole(nDlugosc1, nDlugosc2) <<
std::endl;
getch();
```

Linijki wyświetlające gotowy wynik zawierają wywołania dwóch funkcji – `Obwod()` i `Pole()`. Można je umieścić w tym miejscu, gdyż zwracają one wartości – w dodatku te, które chcemy zaprezentować :) Wyświetlamy je dokładnie w ten sam sposób jak wartości zmiennych i wszystkich innych wyrażeń liczbowych.

Cała istota działania aplikacji zawiera się więc w tych dwóch funkcjach. Prezentują się one następująco:

```
int Obwod(int nBok1, int nBok2)
{
   return 2 * (nBok1 + nBok2);
}

int Pole(int nBok1, int nBok2)
{
   return nBok1 * nBok2;
}
```

Cóż ciekawego możemy o nich rzec? Widoczną różnicą w stosunku do dotychczasowych przykładów funkcji, jakie mieliśmy okazję napotkać, jest chociażby zastąpienie słówka

[17] Całość programu jest dołączona do tutoriala, tutaj zaprezentujemy tylko jego najważniejsze fragmenty

void przez nazwę typu, int. To oczywiście nie przypadek – w taki właśnie sposób informujemy kompilator, iż nasza funkcja ma zwracać wartość oraz wskazujemy jej typ. Kod funkcji to tylko jeden wiersz, zaczynający się od return ('powrót'). Określa on ni mniej, ni więcej, jak tylko ową wartość, która będzie zwrócona i stanie się wynikiem działania funkcji. Rezultat ten zobaczymy w końcu i my w oknie konsoli:

```
   OBLICZANIE OBWODU I POLA PROSTOKATA
------------------------------------------

Podaj dlugosc pierwszego boku: 10
Podaj dlugosc drugiego boku: 14
Obwod prostokata: 48
Pole prostokata: 140
_
```

Screen 15. Miernik prostokątów w akcji

return powoduje jeszcze jeden efekt, który nie jest tu tak wyraźnie widoczny. Użycie tej instrukcji skutkuje mianowicie natychmiastowym **przerwaniem** działania funkcji i powrotem do miejsca jej wywołania. Wprawdzie nasze proste funkcje i tak kończą się niemal od razu, więc nie ma to większego znaczenia, jednak w przypadku poważniejszych podprogramów należy o tym fakcie pamiętać.

Wynika z niego także możliwość użycia return w funkcjach niezwracających żadnej wartości – można je w ten sposób przerwać, zanim wykonają swój kod w całości. Ponieważ nie mamy wtedy żadnej wartości do zwracania, używamy samego słowa return; - bez wskazywania nim jakiegoś wyrażenia.

***

Dowiedzieliśmy się zatem, iż funkcja może zwracać wartość jako wynik swej pracy. Rezultat taki winien być określonego typu – deklarujemy go w nagłówku funkcji jeszcze przed jej nazwą. Natomiast w kodzie funkcji możemy użyć instrukcji return, by wskazać wartość będącą jej wynikiem. Jednocześnie instrukcja ta spowoduje zakończenie działania funkcji.

Składnia funkcji

Gdy już poznaliśmy zawiłości i niuanse związane z używaniem funkcji w swoich aplikacjach, możemy tą wydatną porcję informacji podsumować ogólnymi regułami składniowymi dla podprogramów w C++. Otóż postać funkcji w tym języku jest następująca:

```
typ_zwracanej_wartości/void nazwa_funkcji([typ_parametru nazwa, ...])
{
   instrukcje_1
   return wartość_funkcji_1;
   instrukcje_2
   return wartość_funkcji_2;
   instrukcje_3
   return wartość_funkcji_3;
   ...
   return wartość_funkcji_n;
}
```

Jeżeli dokładnie przestudiowałeś (i zrozumiałeś! :)) wiadomości z tego paragrafu i z poprzedniego rozdziału, nie powinna być ona dla ciebie żadnym zaskoczeniem.

Zauważ jeszcze, ze fraza `return` może występować kilka razy, zwracać w każdym z wariantów **różne** wartości, a całość funkcji mieć logiczny sens i działać poprawnie. Jest to możliwe między innymi dzięki instrukcjom warunkowym, które poznamy już za chwilę.

***

W ten oto sposób uzyskaliśmy bardzo ważną umiejętność programistyczną, jaką jest poprawne używanie funkcji we własnych programach. Upewnij się przeto, iż skrupulatnie przyswoiłeś sobie informacje o tym zagadnieniu, jakie serwowałem w aktualnym i poprzednim rozdziale. W dalszej części kursu będziemy często korzystać z funkcji w przykładowych kodach i omawianych tematach, dlatego ważne jest, byś nie miał kłopotów z nimi.

Jeśli natomiast czujesz się na siłach i chcesz dowiedzieć czegoś więcej o funkcjach, zajrzyj do pomocy MSDN zawartej w Visual Studio.

Sterowanie warunkowe

Dobrze napisany program powinien być przygotowany na każdą ewentualność i nietypową sytuację, jaka może się przydarzyć w czasie jego działania. W niektórych przypadkach nawet proste czynności mogą potencjalnie kończyć się niepowodzeniem, zaś porządna aplikacja musi radzić sobie z takimi drobnymi (lub całkiem sporymi) kryzysami. Oczywiście program nie uczyni nic, czego nie przewidziałby jego twórca. Dlatego też ważnym zadaniem programisty jest opracowanie kodu reagującego odpowiednio na nietypowe sytuacje, w rodzaju błędnych danych wprowadzonych przez użytkownika lub braku pliku potrzebnego aplikacji do działania.

Możliwe są też przypadki, w których dla kilku całkowicie poprawnych sytuacji, danych itp. trzeba wykonać zupełnie inne operacje. Ważne jest wtedy rozróżnienie tych wszystkich wariantów i skierowanie działania programu na właściwe tory, gdy ma miejsce któryś z nich.

Do wszystkich tych zadań stworzono w C++ (i w każdym języku programowania) zestaw odpowiednich narzędzi, zwanych **instrukcjami warunkowymi**. Ich przeznaczeniem jest właśnie dokonywanie różnorakich wyborów, zależnych od ustalonych **warunków**.
Jak widać, przydatność tych konstrukcji jest nadspodziewanie duża, żal byłoby więc ominąć je bez wnikania w ich szczegóły, prawda? :) Niezwłocznie zatem zajmiemy się nimi, poznając ich składnię i sposób funkcjonowania.

Instrukcja warunkowa `if`

Instrukcja `if` ('jeżeli') pozwala wykonać jakiś kod tylko wtedy, gdy spełniony jest określony warunek. Jej działanie sprowadza się więc do sprawdzenia tegoż warunku i, jeśli zostanie stwierdzona jego prawdziwość, wykonania wskazanego bloku kodu.
Tę prostą ideę może ilustrować choćby taki przykład:

```
// SimpleIf – prosty przykład instrukcji if

void main()
{
   int nLiczba;

   std::cout << "Wprowadz liczbe wieksza od 10: ";
   std::cin >> nLiczba;
```

```
    if (nLiczba > 10)
    {
          std::cout << "Dziekuje." << std::endl;
          std::cout << "Wcisnij dowolny klawisz, by zakonczyc.";
          getch();
    }
}
```

Uruchom ten program dwa razy – najpierw podaj liczbę mniejszą od 10, zaś za drugim razem spełnij życzenie aplikacji. Zobaczysz, że w pierwszym przypadku zostaniesz potraktowany raczej mało przyjemnie, gdyż program bez słowa zakończy się. W drugim natomiast otrzymasz stosowne podziękowanie za swoją uprzejmość ;)
„Winna” jest temu, jakżeby inaczej, właśnie instrukcja `if`. W linijce:

```
if (nLiczba > 10)
```

wykonywane jest bowiem sprawdzenie, czy podana przez ciebie liczba jest rzeczywiście większa od 10. Wyrażenie `nLiczba > 10` jest tu więc warunkiem instrukcji `if`.
W przypadku, gdy okaże się on prawdziwy, wykonywane są trzy instrukcje zawarte w nawiasach klamrowych. Jak pamiętamy, sekwencję taką nazywamy **blokiem kodu**.
Jeżeli zaś warunek jest nieprawdziwy (a liczba mniejsza lub równa 10), program **omija** ten blok i wykonuje następną instrukcję występującą za nim. Ponieważ jednak u nas po bloku `if` nie ma żadnych instrukcji, aplikacja zwyczajnie kończy się, gdyż nie ma nic konkretnego do roboty :)

Po takim sugestywnym przykładzie nie od rzeczy będzie przedstawienie składni instrukcji warunkowej `if` w jej prostym wariancie:

```
if (warunek)
{
   instrukcje
}
```

Stopień jej komplikacji z pewnością sytuuje się poniżej przeciętnej ;) Nic w tym dziwnego – to w zasadzie najprostsza, lecz jednocześnie bardzo często używana konstrukcja programistyczna.
Warto jeszcze zapamiętać, że blok *instrukcji* składający się tylko z jednego polecenia możemy zapisać nawet bez nawiasów klamrowych[18]. Wtedy jednak należy postawić na jego końcu średnik:

```
if (warunek) instrukcja;
```

Taka skrócona wersja jest używana często do sprawdzania wartości parametrów funkcji, na przykład:

```
void Funkcja(int nParametr)
{
   // sprawdzenie, czy parametr nie jest mniejszy lub równy zeru -
   // jeżeli tak, to funkcja kończy się
   if (nParametr <= 0) return;

   // ... (reszta funkcji)
}
```

[18] Zasada ta dotyczy prawie każdego bloku kodu w C++ (z wyjątkiem funkcji)

Fraza `else`

Prosta wersja instrukcji `if` nie zawsze jest wystarczająca – nieeleganckie zachowanie naszego przykładowego programu jest dobrym tego uzasadnieniem. Powinien on wszakże pokazać stosowny komunikat również wtedy, gdy użytkownik nie wykaże się chęcią współpracy i nie wprowadzi żądanej liczby. Musi więc uwzględnić przypadek, w którym warunek badany przez instrukcję `if` (u nas `nLiczba > 10`) **nie jest** prawdziwy i zareagować nań w odpowiedni sposób.
Naturalnie, można by umieścić stosowny kod po konstrukcji `if`, ale jednocześnie należałoby zadbać, aby nie był on wykonywany w razie prawdziwości warunku. Sztuczka z dodaniem instrukcji `return;` (przerywającej funkcję `main()`, a więc i cały program) na koniec bloku `if` zdałaby oczywiście egzamin, lecz straty w przejrzystości i prostocie kodu byłyby zdecydowanie niewspółmierne do efektów :))
Dlatego też C++, jako pretendent do miana nowoczesnego języka programowania, posiada bardziej sensowny i logiczny sposób rozwiązania tego problemu. Jest nim mianowicie fraza `else` ('w przeciwnym wypadku') – część instrukcji warunkowej `if`. Korzystająca z niej, ulepszona wersja poprzedniej aplikacji przykładowej może zatem wyglądać chociażby tak:

```
// Else - blok alternatywny w instrukcji if

void main()
{
   int nLiczba;

   std::cout << "Wprowadz liczbe wieksza od 10: ";
   std::cin >> nLiczba;

   if (nLiczba > 10)
   {
         std::cout << "Dziekuje." << std::endl;
         std::cout << "Wcisnij dowolny klawisz, by zakonczyc.";
   }
   else
   {
         std::cout << "Liczba " << nLiczba
                     << " nie jest wieksza od 10." << std::endl;
         std::cout << "Czuj sie upomniany :P";
   }

   getch();
}
```

Gdy uruchomisz powyższy program dwa razy, w podobny sposób jak poprzednio, w każdym wypadku zostaniesz poczęstowany jakimś komunikatem. Zależnie od wpisanej przez ciebie liczby będzie to podziękowanie albo upomnienie :)

Screeny 16 i 17. Dwa warianty działania programu, czyli instrukcje `if` i `else` w całej swej krasie :)

Występujący tu blok `else` jest uzupełnieniem instrukcji `if` – kod w nim zawarty zostanie wykonany tylko wtedy, gdy określony w `if` warunek **nie będzie** spełniony. Dzięki temu możemy odpowiednio zareagować na każdą ewentualność, a zatem nasz program zachowuje się porządnie w obu możliwych przypadkach :)

Funkcja `getch()` jest w tej aplikacji wywoływana poza blokami warunkowymi, gdyż niezależnie od wpisanej liczby i treści wyświetlanego komunikatu istnieje potrzeba poczekania na dowolny klawisz. Zamiast więc umieszczać tę instrukcję zarówno w bloku `if`, jak i `else`, można ją zostawić całkowicie poza nimi.

Czas teraz zaprezentować składnię pełnej wersji instrukcji `if`, uwzględniającej także blok alternatywny `else`:

```
if (warunek)
{
   instrukcje_1
}
else
{
   instrukcje_2
}
```

Kiedy *`warunek`* jest prawdziwy, uruchamiane są *`instrukcje_1`*, zaś w przeciwnym przypadku (`else`) – *`instrukcje_2`*. Czy świat widział kiedyś coś równie elementarnego? ;) Nie daj się jednak zwieść tej prostocie – instrukcja warunkowa `if` jest w istocie potężnym narzędziem, z którego intensywnie korzystają wszystkie programy.

Bardziej złożony przykład

By całkowicie upewnić się, iż znamy i rozumiemy tę szalenie ważną konstrukcję programistyczną, przeanalizujemy ostatnią, bardziej skomplikowaną ilustrację jej użycia. Będzie to aplikacja rozwiązująca równania liniowe – tzn. wyrażenia postaci:

$$ax + b = 0$$

Jak zapewne pamiętamy ze szkoły, mogą mieć one zero, jedno lub nieskończenie wiele rozwiązań, a wszystko zależy od wartości współczynników *a* i *b*. Mamy zatem duże pole do popisu dla instrukcji `if` :D
Program realizujący to zadanie wygląda więc tak:

```
// LinearEq - rozwiązywanie równań liniowych

float fA;
std::cout << "Podaj wspolczynnik a: ";
std::cin >> fA;

float fB;
std::cout << "Podaj wspolczynnik b: ";
std::cin >> fB;

if (fA == 0.0)
{
   if (fB == 0.0)
         std::cout << "Rownanie spelnia kazda liczba rzeczywista."
                   << std::endl;
   else
         std::cout << "Rownanie nie posiada rozwiazan." << std::endl;
}
else
   std::cout << "x = " << -fB / fA << std::endl;

getch();
```

Zagnieżdżona instrukcja `if` wygląda może cokolwiek tajemniczo, ale w gruncie rzeczy istota jej działania jest w miarę prosta. Wyjaśnimy ją za moment.
Najpierw powtórka z matematyki :) Przypomnijmy, iż równanie liniowe *ax + b = 0*:

- posiada nieskończenie wiele rozwiązań, jeżeli współczynniki *a* i *b* są jednocześnie równe zeru
- nie posiada w ogóle rozwiązań, jeżeli *a* jest równe zeru, zaś *b* nie
- ma dokładnie jedno rozwiązanie (*-b/a*), gdy *a* jest różne od zera

Wynika stąd, że istnieją trzy możliwe przypadki i scenariusze działania programu. Zauważmy jednak, że warunek „*a* jest równe zeru" jest konieczny do realizacji dwóch z nich – możemy więc go wyodrębnić i zapisać w postaci pierwszej (bardziej zewnętrznej) instrukcji `if`.
Nadal wszakże pozostaje nam problem współczynnika *b* – sam fakt zerowej wartości *a* nie przecież pozwala na obsłużenie wszystkich możliwości. Rozwiązaniem jest umieszczenie instrukcji sprawdzającej *b* (czyli także `if`) **wewnątrz** bloku `if`, sprawdzającego *a*! Umożliwia to poprawne wykonanie programu dla wszystkich wartości liczb *a* i *b*.

```
   ROWNANIE LINIOWE (ax + b = 0)
-----------------------------------
Podaj wspolczynnik a: 12
Podaj wspolczynnik b: 6
x = -0.5
_
```

Screen 18. Program rozwiązujący równania liniowe

Używamy toteż dwóch instrukcji `if`, które razem odpowiadają za właściwe zachowanie się aplikacji w trzech możliwych przypadkach. Pierwsza z nich:

```
if (fA == 0.0)
```

kontroluje wartość współczynnika *a* i tworzy pierwsze rozgałęzienie na szlaku działania programu. Jedna z wychodzących z niego dróg prowadzi do celu zwanego „dokładnie jedno rozwiązanie równania", druga natomiast do kolejnego rozwidlenia:

```
if (fB == 0.0)
```

Ono też kieruje wykonywanie aplikacji albo do „nieskończenie wielu rozwiązań", albo też do „braku rozwiązań" równania – zależy to oczywiście od ewentualnej równości *b* z zerem.

Operatorem równości w C++ jest `==`, czyli **podwójny** znak „równa się" (`=`). Należy koniecznie odróżniać go od operatora przypisania, czyli pojedynczego znaku `=`. Jeśli omyłkowo użylibyśmy tego drugiego w wyrażeniu będącym warunkiem, to najprawdopodobniej byłby on zawsze albo prawdziwy, albo fałszywy[19] – na pewno jednak nie działałby tak, jak byśmy tego oczekiwali. Co gorsza, można by się o tym przekonać dopiero w czasie działania programu, gdyż jego kompilacja przebiegłaby bez zakłóceń. Pamiętajmy więc, by w wyrażeniach warunkowych do sprawdzania równości używać zawsze operatora `==`, rezerwując znak `=` do przypisywania wartości zmiennym.

***

[19] Zależałoby to od wartości po prawej stronie znaku równości – jeśli byłaby równa zeru, warunek byłby fałszywy, w przeciwnym wypadku - prawdziwy

Ostrzeżeniem tym kończymy nasze nieco przydługie spotkanie z instrukcją warunkową `if`. Można śmiało powiedzieć, że oto poznaliśmy jeden z fundamentów, na których opiera się działanie wszelkich algorytmów w programach komputerowych. Twoje aplikacje nabiorą przez to elastyczności i będą zdolne do wykonywania mniej trywialnych zadań… **jeżeli** sumiennie przestudiowałeś ten podrozdział! :))

Instrukcja wyboru `switch`

Instrukcja `switch` ('przełącz') jest w pewien sposób podobna do `if`: jej przeznaczeniem jest także wybór jednego z wariantów kodu podczas działania programu. Pomiędzy obiema konstrukcjami istnieją jednak dość znaczne różnice.
O ile `if` podejmuje decyzję na podstawie prawdziwości lub fałszywości jakiegoś warunku, o tyle `switch` bierze pod uwagę **wartość** podanego **wyrażenia**. Z tego też powodu może dokonywać wyboru spośród większej liczby możliwości niż li tylko dwóch (prawdy lub fałszu).
Najlepiej widać to na przykładzie:

```
// Switch - instrukcja wyboru

void main()
{
   // (pomijam tu kod odpowiedzialny za pobranie od użytkownika dwóch
   // liczb - robiliśmy to tyle razy, że nie powinieneś mieć z tym
   // kłopotów :)) Liczby są zapisane w zmiennych fLiczba1 i fLiczba2)

   int nOpcja;
   std::cout << "Wybierz dzialanie:" << std::endl;
   std::cout << "1. Dodawanie" << std::endl;
   std::cout << "2. Odejmowanie" << std::endl;
   std::cout << "3. Mnozenie" << std::endl;
   std::cout << "4. Dzielenie" << std::endl;
   std::cout << "0. Wyjscie" << std::endl;
   std::cout << "Twoj wybor: ";
   std::cin >> nOpcja;

   switch (nOpcja)
   {
         case 1: std::cout << fLiczba1 << " + " << fLiczba2 << " = "
                           << fLiczba1 + fLiczba2; break;
         case 2: std::cout << fLiczba1 << " - " << fLiczba2 << " = "
                           << fLiczba1 - fLiczba2; break;
         case 3: std::cout << fLiczba1 << " * " << fLiczba2 << " = "
                           << fLiczba1 * fLiczba2; break;
         case 4:
               if (fLiczba2 == 0.0)
                     std::cout << "Dzielnik nie moze byc zerem!";
               else
                     std::cout << fLiczba1 << " / " << fLiczba2 << " = "
                                 << fLiczba1 / fLiczba2;
               break;
         case 0: std::cout << "Dziekujemy :)"; break;
         default: std::cout << "Nieznana opcja!";
   }

   getch();
}
```

No, to już jest program, co się zowie: posiada szeroką funkcjonalność, prosty interfejs – krótko mówiąc pełen profesjonalizm ;) Tym bardziej więc powinniśmy przejrzeć

dokładniej jego kod źródłowy - zważywszy, iż zawiera interesującą nas w tym momencie instrukcję `switch`.
Zajmuje ona zresztą pokaźną część listingu; na dodatek jest to ten fragment, w którym wykonywane są obliczenia, będące podstawą działania programu. Jaka jest zatem rola tej konstrukcji?

```
   KALKULATOR
----------------

Podaj pierwsza liczbe: 243
Podaj druga liczbe: 3

Wybierz dzialanie:
1. Dodawanie
2. Odejmowanie
3. Mnozenie
4. Dzielenie
0. Wyjscie
Twoj wybor: 4
243 / 3 = 81
```

Screen 19. Kalkulator w działaniu

Cóż, nie jest trudno domyśleć się jej – skoro mamy w naszym programie menu, będziemy też mieli kilka wariantów jego działania. Wybranie przez użytkownika jednego z nich zostaje wcielone w życie właśnie poprzez instrukcję `switch`. Porównuje ona kolejno wartość zmiennej `nOpcja` (do której zapisujemy numer wskazanej pozycji menu) z pięcioma wcześniej ustalonymi przypadkami. Każdemu z nich odpowiada fragment kodu, zaczynający się od słówka `case` ('przypadek') i kończący na `break;` ('przerwij'). Gdy któryś z nich zostanie uznany za właściwy (na podstawie wartości wspomnianej już zmiennej), wykonywane są zawarte w nim instrukcje. Jeżeli zaś żaden nie będzie pasował, program „skoczy" do dodatkowego wariantu `default` ('domyślny') i uruchomi jego kod. Ot, i cała filozofia :)

Po tym pobieżnym wyjaśnieniu działania instrukcji `switch`, poznamy jej pełną postać składniową:

```
switch (wyrażenie)
{
   case wartość_1:
         instrukcje_1
         [break;]
   case wartość_2:
         instrukcje_2
         [break;]
   ...
   case wartość_n;
         instrukcje_n;
         [break;]
   [default:
         instrukcje_domyślne]
}
```

Korzystając z niej, jeszcze prościej zrozumieć przeznaczenie konstrukcji `switch` oraz wykonywane przez nią czynności. Mianowicie, oblicza ona wpierw wynik *`wyrażenia`*, by potem porównywać go kolejno z podanymi (w instrukcjach `case`) *`wartościami`*. Kiedy stwierdzi, że zachodzi równość, skacze na początek pasującego wariantu i wykonuje cały kod aż **do końca bloku `switch`**.
Zaraz – jak to do końca bloku? Przecież w naszym przykładowym programie, gdy wybraliśmy, powiedzmy, operację odejmowania, to otrzymywaliśmy wyłącznie różnicę liczb – bez iloczynu i ilorazu (czyli dalszych opcji). Przyczyna tego tkwi w instrukcji

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
```

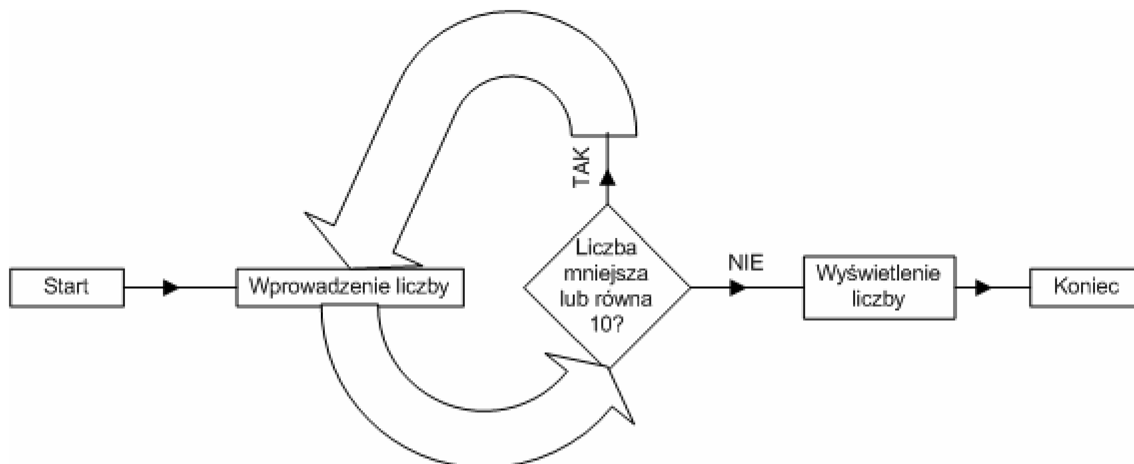
```
#include <iostream>
#include <conio.h>

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 20. 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 21. 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;
```

```
// wypisanie dziesięciu liczb całkowitych w osobnych linijkach
while (nLicznik <= 10)
{
   std::cout << nLicznik << std::endl;
   nLicznik++;
}
```

Powyższe rozwiązanie jest z pewnością poprawne, aczkolwiek istnieje jeszcze lepsze :) W przypadku, gdy znamy z góry liczbę przebiegów pętli, bardziej naturalne staje się użycie instrukcji `for` ('dla'). Została ona bowiem stworzona specjalnie na takie okazje[20] i sprawdza się w nich o wiele lepiej niż uniwersalna `while`. Korzystający z niej ekwiwalent powyższego kodu może wyglądać na przykład tak:

```
for (int i = 1; i <= 10; i++)
{
   std::cout << i << std::endl;
}
```

Jeżeli uważnie przyjrzysz się obu jego wersjom, z pewnością zdołasz domyśleć się ogólnej zasady działania pętli `for`. Zanim dokładnie ją wyjaśnię, posłużę się bardziej wyrafinowanym przykładem do jej ilustracji:

```
// For - pętla krokowa

int Suma(int nLiczba)
{
   int nSuma = 0;

   for (int i = 1; i <= nLiczba; i++)
         nSuma += i;

   return nSuma;
}

void main()
{
   int nLiczba;
   std::cout << "Program oblicza sume od 1 do podanej liczby."
             << std::endl;
   std::cout << "Podaj ja: ";
   std::cin >> nLiczba;

   std::cout << "Suma liczb od 1 do " << nLiczba << " wynosi "
             << Suma(nLiczba) << ".";
   getch();
}
```

Mamy zatem kolejny superużyteczny programik do przeanalizowania ;) Bezzwłocznie więc przystąpmy do wykonania tego pożytecznego zadania.
Rzut oka na kod tudzież kompilacja i uruchomienie aplikacji prowadzi do słusznego wniosku, iż przeznaczeniem programu jest obliczanie sumy kilku początkowych liczb naturalnych. Zakres dodawania ustala przy tym sam użytkownik programu.

Czynnością sumowania zajmuje się tu odrębna funkcja `Suma()`, na której skupimy obecnie całą naszą uwagę.

---

[20] `for` nie jest tylko wymysłem twórców C++. Podobne konstrukcje spotkać można właściwie w każdym języku programowania, istnieją też nawet bardziej wyspecjalizowane ich odmiany. Trudno więc uznać tę poczciwą pętlę za zbędne udziwnienie :)

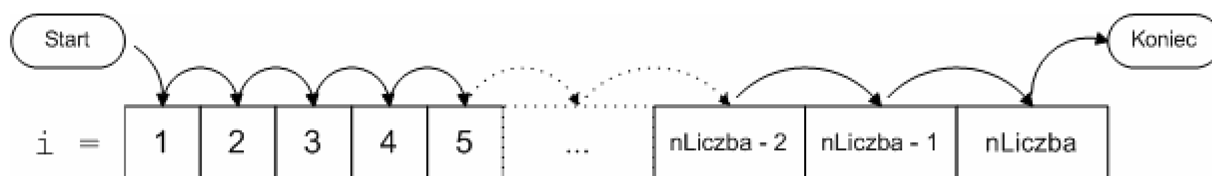
Pierwsza linijka tej funkcji to znana już nam deklaracja zmiennej, połączona z jej inicjalizacją wartością `0`. Owa zmienna, `nSuma`, będzie przechowywać obliczony wynik dodawania, który zostanie zwrócony jako rezultat całej funkcji.
Najbardziej interesującym fragmentem jest występująca dalej pętla `for`:

```
for (int i = 1; i <= nLiczba; i++)
   nSuma += i;
```

Wykonuje ona zasadnicze obliczenia: dodaje do zmiennej `nSuma` kolejne liczby naturalne, zatrzymując się na podanym w funkcji parametrze. Całość odbywa się w następujący, dość prosty sposób:

- Instrukcja `int i = 1` jest wykonywana raz na samym początku. Jak widać, jest to deklaracja i inicjalizacja zmiennej `i`. Nazywamy ją **licznikiem pętli**. W kolejnych cyklach będzie ona przyjmować wartości `1`, `2`, `3`, itd.
- Kod `nSuma += i;` stanowi blok pętli[21] i jest uruchamiany przy każdym jej przebiegu. Skoro zaś licznik `i` jest po kolei ustawiany na następujące po sobie liczby naturalne, pętla `for` staje się odpowiednikiem sekwencji instrukcji `nSuma += 1; nSuma += 2; nSuma += 3; nSuma += 4;` itd.
- Warunek `i <= nLiczba` określa górną granicę sumowania. Jego obecność sprawia, że pętla jest wykonywana tylko wtedy, gdy licznik `i` jest mniejszy lub równy zmiennej `nLiczba`. Zgadza się to oczywiście z naszym zamysłem.
- Wreszcie, na koniec każdego cyklu instrukcja `i++` powoduje zwiększenie wartości licznika o jeden.

Po dłuższym zastanowieniu nad powyższym opisem można niewątpliwie dojść do wniosku, że nie jest on wcale taki skomplikowany, prawda? :) Zrozumienie go nie powinno nastręczać ci zbyt wielu trudności. Gdyby jednak tak było, przypomnij sobie podaną w tytule nazwę pętli `for` – **krokowa**.
To całkiem trafne określenie dla tej konstrukcji. Jej zadaniem jest bowiem przebycie pewnej „drogi" (u nas są to liczby od `1` do wartości zmiennej `nLiczba`) poprzez serię małych kroków i wykonanie po drodze jakichś działań. Klarownie przedstawia to tenże rysunek:



Schemat 5. "Droga" przykładowej pętli for

Mam nadzieję, że teraz nie masz już żadnych kłopotów ze zrozumieniem zasady działania naszego programu.

Przyszedł czas na zaprezentowanie składni omawianej przez nas pętli:

```
for ([początek]; [warunek]; [cykl])
{
   instrukcje
}
```

[21] Jak zapewne pamiętasz, jedną linijkę w bloku kodu możemy zapisać bez nawiasów klamrowych `{}` – dowiedzieliśmy się tego przy okazji instrukcji `if` :)

Na jej podstawie możemy dogłębnie poznać funkcjonowanie tego ważnego tworu programistycznego. Dowiemy się też, dlaczego konstrukcja `for` jest uważana za jedną z mocnych stron języka C++.

Zaczniemy od początku, czyli komendy oznaczonej jako… *początek* :) Wykonuje się ona jeden raz, jeszcze przed wejściem we właściwy krąg pętli. Zazwyczaj umieszczamy tu instrukcję, która ustawia licznik na wartość początkową (może to być połączone z jego deklaracją).
warunek jest sprawdzany przed każdym cyklem *instrukcji*. Jeżeli nie jest on spełniony, pętla natychmiast kończy się. Zwykle więc wpisujemy w jego miejsce kod porównujący licznik z wartością końcową.
W każdym przebiegu, po wykonaniu *instrukcji*, pętla uruchamia jeszcze fragment zaznaczony jako *cykl*. Naturalną jego treścią będzie zatem zwiększenie lub zmniejszenie licznika (w zależności od tego, czy liczymy w górę czy w dół).

Inkrementacja czy dekrementacja nie jest bynajmniej jedyną czynnością, jaką możemy tutaj wykonać na liczniku. Posłużenie się choćby mnożeniem, dzieleniem czy nawet bardziej zaawansowanymi funkcjami jest jak najbardziej dopuszczalne.
Wpisując na przykład `i *= 2` otrzymamy kolejne potęgi dwójki (2, 4, 8, 16 itd.), `i += 10` – wielokrotności dziesięciu, itp. Jest to znaczna przewaga nad wieloma innymi językami programowania, w których liczniki analogicznych pętli mogą się zmieniać jedynie w postępie arytmetycznym (o stałą wartość - niekiedy nawet dopuszczalna jest tu wyłącznie jedynka!).

Elastyczność pętli `for` polega między innymi na fakcie, iż **żaden** z trzech podanych w nawiasie „parametrów" nie jest obowiązkowy! Wprawdzie na pierwszy rzut oka obecność każdego wydaje się tu absolutnie niezbędna, jednakże pominięcie któregoś (czasem nawet wszystkich) może mieć swoje logiczne uzasadnienie.
Brak *początku* lub *cyklu* powoduje dość przewidywalny skutek – w chwili, gdy miałyby zostać wykonane, program nie podejmie po prostu żadnych akcji. O ile nieobecność instrukcji ustawiającej licznik na wartość początkową jest okolicznością rzadko spotykaną, o tyle pominięcie frazy *cykl* jest konieczne, jeżeli nie chcemy zmieniać licznika przy każdym przebiegu pętli. Możemy to osiągnąć, umieszczając odpowiedni kod np. wewnątrz zagnieżdżonego bloku `if`.
Gdy natomiast opuścimy *warunek*, iteracja nie będzie miała czego weryfikować przy każdym swym „obrocie", więc zapętli się w nieskończoność. Przerwanie tego błędnego koła będzie możliwe tylko poprzez instrukcję `break`, którą już za chwilę poznamy bliżej.

***

W ten oto sposób zawarliśmy bliższą znajomość z pętla krokową `for`. Nie jest to może łatwa konstrukcja, ale do wielu zastosowań zdaje się być bardzo wygodna. Z tego względu będziemy jej często używali – tak też robią wszyscy programiści C++.

Instrukcje `break` i `continue`

Z pętlami związane są jeszcze dwie instrukcje pomocnicze. Nierzadko ułatwiają one rozwiązywanie pewnych problemów, a czasem wręcz są do tego niezbędne. Mowa tu o tytułowych `break` i `continue`.

Z instrukcją `break` ('przerwij') spotkaliśmy się już przy okazji konstrukcji `switch`. Korzystaliśmy z niej, aby zagwarantować wykonanie kodu odpowiadającego tylko jednemu wariantowi `case`. `break` powodowała bowiem przerwanie bloku `switch` i przejście do następnej linijki po nim.

Rola tej instrukcji w kontekście pętli nie zmienia się ani na jotę: jej wystąpienie wewnątrz bloku do, while lub for powoduje dokładnie ten sam efekt. Bez względu na prawdziwość lub nieprawdziwość warunku pętli jest ona błyskawicznie przerywana, a punkt wykonania programu przesuwa się do kolejnego wiersza za nią.
Przy pomocy break możemy teraz nieco poprawić nasz program demonstrujący pętlę do:

```
// Break – przerwanie pętli

void main()
{
   int nLiczba;

   do
   {
         std::cout << "Wprowadz liczbe wieksza od 10" << std::endl;
         std::cout << "lub zero, by zakonczyc program: ";
         std::cin >> nLiczba;

         if (nLiczba == 0) break;
   } while (nLiczba <= 10);

   std::cout << "Nacisnij dowolny klawisz.";
   getch();
}
```

Mankament niemożności zakończenia aplikacji bez spełnienia jej prośby został tutaj skutecznie usunięty. Mianowicie, gdy wprowadzimy liczbę zero, instrukcja if skieruje program ku komendzie break, która natychmiast zakończy pętlę i uwolni użytkownika od irytującego żądania :)

Podobny skutek (przerwanie pętli po wpisaniu przez użytkownika zera) osiągnęlibyśmy zmieniając warunek pętli tak, by stawał się prawdziwy również wtedy, gdy zmienna nLiczba miałaby wartość 0. W następnym rozdziale dowiemy się, jak poczynić podobną modyfikację.

Instrukcja continue jest używana nieco rzadziej. Gdy program natrafi na nią wewnątrz bloku pętli, wtedy automatycznie kończy bieżący cykl i rozpoczyna nowy przebieg iteracji. Z instrukcji tej korzystamy najczęściej wtedy, kiedy część (zwykle większość) kodu pętli ma być wykonywana tylko pod określonym, dodatkowym warunkiem.

***

Zakończyliśmy właśnie poznawanie bardzo ważnych elementów języka C++, czyli pętli. Dowiedzieliśmy się o zasadach ich działania, składni oraz przykładowych zastosowaniach. Tych ostatnich będzie nam systematycznie przybywało wraz z postępami w sztuce programowania, gdyż pętle to bardzo intensywnie wykorzystywany mechanizm – nie tylko zresztą w C++.

Podsumowanie

Ten długi i ważny rozdział prezentował możliwości C++ w zakresie sterowania przebiegiem aplikacji oraz sposobem jej działania.

Pierwszym zagadnieniem było bystrzejsze spojrzenie na funkcje, co obejmowało poznanie ich parametrów oraz zwracanych wartości. Dalej zerknęliśmy na instrukcje warunkowe, które wreszcie dopuszczały nam przewidywać różne ewentualności pracy programu. Na

koniec, pętle dały nam okazję stworzyć nieco mniej banalne aplikacje niż zwykle – w tym i jedną grę! :D
Tą drogą nabyliśmy przeto umiejętność tworzenia programów wykonujących niemal dowolne zadania. Pewnie teraz nie jesteś o tym szczególnie przekonany, jednak pamiętaj, że poznanie instrumentów to tylko pierwszy krok do osiągnięcia wirtuozerii. Niezastąpiona jest praktyka w prawdziwym programowaniu, a sposobności do niej będziesz miał z pewnością bez liku - także w niniejszym kursie :)

Pytania i zadania

Tak obszerny i kluczowy rozdział nie może się obejść bez słusznego pakietu zadań domowych ;) Oto i one:

Pytania

1. Jaka jest rola parametrów funkcji?
2. Czy ilość parametrów w deklaracji i wywołaniu funkcji może być różna?
   Wskazówka: Poczytaj w MSDN o domyślnych wartościach parametrów funkcji.
3. Co się stanie, jeżeli nie umieścimy instrukcji `break` po wariancie `case` w bloku `switch`?
4. W jakich sytuacjach, oprócz niepodania warunku, pętla `for` będzie się wykonywała w nieskończoność? A kiedy nie wykona się ani razu?
   Czy podobnie jest z pętlą `while`?

Ćwiczenia

1. Stwórz program, który poprosi użytkownika o liczbę całkowitą i przyporządkuje ją do jednego z czterech przedziałów: liczb ujemnych, jednocyfrowych, dwucyfrowych lub pozostałych.
   Która z instrukcji – `if` czy `switch` – będzie tu odpowiednia?
2. Napisz aplikację wyświetlającą listę liczb od 1 do 100 z podanymi obok wartościami ich drugich potęg (kwadratów).
   Jaką pętlę – `do`, `while` czy `for` – należałoby tu zastosować?
3. Zmodyfikuj program przykładowy prezentujący pętlę `while`. Niech zlicza on próby zgadnięcia liczby podjęte przez gracza i wyświetla na końcu ich ilość.