4

OPERACJE NA ZMIENNYCH

Są plusy dodatnie i plusy ujemne.
Lech Wałęsa

W tym rozdziale przyjrzymy się dokładnie zmiennym i wyrażeniom w języku C++. Jak wiemy, służą one do przechowywania wszelkich danych i dokonywania nań różnego rodzaju manipulacji. Działania takie są podstawą każdej aplikacji, a w złożonych algorytmach gier komputerowych mają niebagatelne znaczenie.

Poznamy więc szczegółowo większość aspektów programowania związanych ze zmiennymi oraz zobaczymy często używane operacje na danych liczbowych i tekstowych.

Wnikliwy rzut oka na zmienne

Zmienna to coś w rodzaju pojemnika na informacje, mogącego zawierać określone dane. Wcześniej dowiedzieliśmy się, iż dla każdej zmiennej musimy określić **typ danych**, które będziemy w niej przechowywać, oraz **nazwę**, przez którą będziemy ją identyfikować. Określenie takie nazywamy **deklaracją** zmiennej i stosowaliśmy je niemal w każdym programie przykładowym – powinno więc być ci doskonale znane :)

Nasze aktualne wiadomości o zmiennych są mimo tego dość skąpe i dlatego musimy je niezwłocznie poszerzyć. Uczynimy to wszakże w niniejszym podrozdziale.

Zasięg zmiennych

Gdy deklarujemy zmienną, podajemy jej typ i nazwę – to oczywiste. Mniej dostrzegalny jest fakt, iż jednocześnie określamy też obszar obowiązywania takiej deklaracji. Innymi słowy, definiujemy **zasięg** zmiennej.

Zasięg (**zakres**, ang. *scope*) zmiennej to część kodu, w ramach której dana zmienna jest dostępna.

Wyróżniamy kilka rodzajów zasięgów. Do wszystkich jednak stosuje się ogólna, naturalna reguła: niepoprawne jest jakiekolwiek użycie zmiennej **przed** jej deklaracją. Tak więc poniższy kod:

```
std::cin >> nZmienna;
int nZmienna;
```

niechybnie spowoduje błąd kompilacji. Sądzę, że jest to dość proste i logiczne – nie możemy przecież wymagać od kompilatora znajomości czegoś, o czym sami go wcześniej nie poinformowaliśmy.

W niektórych językach programowania (na przykład Visual Basicu czy PHP) możemy jednak używać niezadeklarowanych zmiennych. Większość programistów uważa to za

niedogodność i przyczynę powstawania trudnych do wykrycia błędów (spowodowanych choćby literówkami). Ja osobiście całkowicie podzielam ten pogląd :D

Na razie poznamy dwa rodzaje zasięgów – **lokalny** i **modułowy**.

Zasięg lokalny

Zakres lokalny obejmuje pojedynczy **blok kodu**. Jak pamiętasz, takim blokiem nazywamy fragment listingu zawarty między nawiasami klamrowymi `{ }`. Dobrym przykładem mogą być tu bloki warunkowe instrukcji `if`, bloki pętli, a także całe funkcje. Otóż każda zmienna deklarowana wewnątrz takiego bloku ma właśnie zasięg lokalny.

Zakres lokalny obejmuje kod od miejsca deklaracji zmiennej aż do końca bloku, wraz z ewentualnymi blokami zagnieżdżonymi.

Te dość mgliste stwierdzenia będą pewnie bardziej wymowne, jeżeli zostaną poparte odpowiednimi przykładami. Zerknijmy więc na poniższy kod:

```
void main()
{
    int nX;
    std::cin >> nX;

    if (nX > 0)
    {
          std::cout << nX;
          getch();
    }
}
```

Jego działanie jest, mam nadzieję, zupełnie oczywiste (zresztą nieszczególnie nas teraz interesuje :)). Przyjrzyjmy się raczej zmiennej `nX`. Jako że zadeklarowaliśmy ją wewnątrz bloku kodu – w tym przypadku funkcji `main()` – posiada ona zasięg lokalny. Możemy zatem korzystać z niej do woli w całym tym bloku, a więc także w **zagnieżdżonej** instrukcji `if`.

Dla kontrastu spójrzmy teraz na inny, choć podobny kod:

```
void main()
{
    int nX = 1;

    if (nX > 0)
    {
          int nY = 10;
    }

    std::cout << nY;
    getch();
}
```

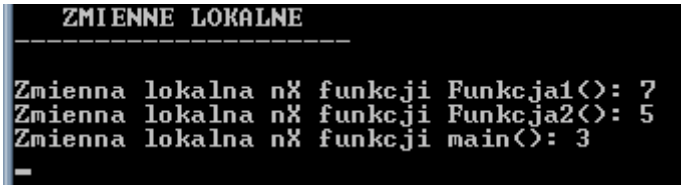
Powinien on wypisać liczbę `10`, prawda? Cóż… niezupełnie :) Sama próba uruchomienia programu skazana jest na niepowodzenie: kompilator „przyczepi” się do przedostatniego wiersza, zawierającego nazwę zmiennej `nY`. Wyda mu się bowiem kompletnie nieznana! Ale dlaczego?! Przecież zadeklarowaliśmy ją ledwie dwie linijki wyżej! Czyż nie możemy więc użyć jej tutaj?…
Jeżeli uważnie przeczytałeś poprzednie akapity, to zapewne znasz już przyczynę niezadowolenia kompilatora. Mianowicie, zmienna `nY` ma zasięg lokalny, obejmujący

wyłącznie blok `if`. Reszta funkcji `main()` nie należy już do tego bloku, a zatem znajduje się **poza** zakresem `nY`. Nic dziwnego, że zmienna jest tam traktowana jako obca – poza swoim zasięgiem ona faktycznie **nie istnieje**, gdyż jest usuwana z pamięci w momencie jego opuszczenia.

Zmiennych o zasięgu lokalnym relatywnie najczęściej używamy jednak bezpośrednio we wnętrzu funkcji. Przyjęło się nawet nazywać je **zmiennymi lokalnymi**[22] lub **automatycznymi**. Ich rolą jest zazwyczaj przechowywanie tymczasowych danych, wykorzystywanych przez podprogramy, lub częściowych wyników obliczeń.
Tak jak poszczególne funkcje w programie, tak i ich zmienne lokalne są od siebie całkowicie niezależne. Istnieją w pamięci komputera jedynie podczas wykonywania funkcji i „znikają" po jej zakończeniu. Niemożliwe jest więc odwołanie do zmiennej lokalnej spoza jej macierzystej funkcji. Poniższy przykład ilustruje ten fakt:

```
// LocalVariables - zmienne lokalne

void Funkcja1()
{
   int nX = 7;
   std::cout << "Zmienna lokalna nX funkcji Funkcja1(): " << nX
             << std::endl;
}

void Funkcja2()
{
   int nX = 5;
   std::cout << "Zmienna lokalna nX funkcji Funkcja2(): " << nX
             << std::endl;
}

void main()
{
   int nX = 3;

   Funkcja1();
   Funkcja2();
   std::cout << "Zmienna lokalna nX funkcji main(): " << nX
             << std::endl;

   getch();
}
```

Mimo że we wszystkich trzech funkcjach (`Funkcja1()`, `Funkcja2()` i `main()`) nazwa zmiennej jest identyczna (`nX`), w każdym z tych przypadków mamy do czynienia z zupełnie **inną** zmienną.



Screen 22. Ta sama nazwa, lecz inne znaczenie. Każda z trzech lokalnych zmiennych `nX` jest całkowicie odrębna i niezależna od pozostałych

[22] Nie tylko zresztą w C++. Wprawdzie sporo języków jest uboższych o możliwość deklarowania zmiennych wewnątrz bloków warunkowych, pętli czy podobnych, ale niemal wszystkie pozwalają na stosowanie zmiennych lokalnych. Nazwa ta jest więc obecnie używana w kontekście dowolnego języka programowania.

Mogą one współistnieć obok siebie pomimo takich samych nazw, gdyż ich zasięgi **nie pokrywają się**. Kompilator słusznie więc traktuje je jako twory absolutnie niepowiązane ze sobą. I tak też jest w istocie – są one „wewnętrznymi sprawami" każdej z funkcji, do których nikt nie ma prawa się mieszać :)

Takie wyodrębnianie niektórych elementów aplikacji nazywamy hermetyzacją (ang. *encapsulation*). Najprostszym jej wariantem są właśnie podprogramy ze zmiennymi lokalnymi, niedostępnymi dla innych. Dalszym krokiem jest tworzenie klas i obiektów, które dokładnie poznamy w dalszej części kursu.
Zaletą takiego dzielenia kodu na mniejsze, zamknięte części jest większa łatwość modyfikacji oraz niezawodność. W dużych projektach, realizowanych przez wiele osób, podział na odrębne fragmenty jest w zasadzie nieodzowny, aby współpraca między programistami przebiegała bez problemów.

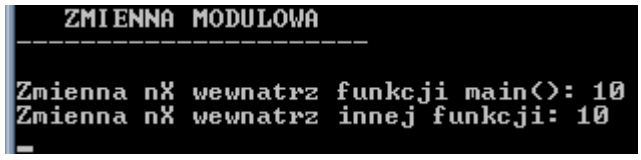
Ze zmiennymi o zasięgu lokalnym spotykaliśmy się dotychczas nieustannie w naszych programach przykładowych. Prawdopodobnie zatem nie będziesz miał większych kłopotów ze zrozumieniem sensu tego pojęcia. Jego precyzyjne wyjaśnienie było jednak nieodzowne, abym z czystym sumieniem mógł kontynuować :D

Zasięg modułowy

Szerszym zasięgiem zmiennych jest zakres modułowy. Posiadające go zmienne są widoczne w całym **module kodu**. Możemy więc korzystać z nich we **wszystkich funkcjach**, które umieścimy w tymże module.
Jeżeli zaś jest to jedyny plik z kodem programu, to oczywiście zmienne te będą dostępne dla całej aplikacji. Nazywamy się je wtedy **globalnymi**.
Aby zobaczyć, jak „działają" zmienne modułowe, przyjrzyj się następującemu przykładowi:

```
// ModularVariables - zmienne modułowe

int nX = 10;

void Funkcja()
{
   std::cout << "Zmienna nX wewnatrz innej funkcji: " << nX
             << std::endl;
}

void main()
{
   std::cout << "Zmienna nX wewnatrz funkcji main(): " << nX
             << std::endl;
   Funkcja();

   getch();
}
```

Zadeklarowana na początku zmienna `nX` ma właśnie zasięg modułowy. Odwołując się do niej, obie funkcje (`main()` i `Funkcja()`) wyświetlają wartość jednej i **tej samej** zmiennej.



Screen 23. Zakres modułowy zmiennej

Jak widać, deklarację zmiennej modułowej umieszczamy bezpośrednio w pliku źródłowym, **poza** kodem wszystkich funkcji. Wyłączenie jej na zewnątrz podprogramów daje zatem łatwy do przewidzenia skutek: zmienna staje się dostępna w całym module i we wszystkich zawartych w nim funkcjach.

Oczywistym zastosowaniem dla takich zmiennych jest przechowywanie danych, z których korzysta wiele procedur. Najczęściej muszą być one zachowane przez większość czasu działania programu i osiągalne z każdego miejsca aplikacji. Typowym przykładem może być chociażby numer aktualnego etapu w grze zręcznościowej czy nazwa pliku otwartego w edytorze tekstu. Dzięki zastosowaniu zmiennych o zasięgu modułowym dostęp do takich kluczowych informacji nie stanowi już problemu.

Zakres modułowy dotyczy tylko jednego pliku z kodem źródłowym. Jeśli nasza aplikacja jest na tyle duża, byśmy musieli podzielić ją na kilka modułów, może on wszakże nie wystarczać. Rozwiązaniem jest wtedy wyodrębnienie globalnych deklaracji we własnym pliku nagłówkowym i użycie dyrektywy `#include`. Będziemy o tym szerzej mówić w niedalekiej przyszłości :)

Przesłanianie nazw

Gdy używamy zarówno zmiennych o zasięgu lokalnym, jak i modułowym (czyli w normalnym programowaniu w zasadzie nieustannie), możliwa jest sytuacja, w której z danego miejsca w kodzie dostępne są dwie zmienne o **tej samej nazwie**, lecz **różnym zakresie**. Wyglądać to może chociażby tak:

```
int nX = 5;

void main()
{
   int nX = 10;
   std::cout << nX;
}
```

Pytanie brzmi: do **której** zmiennej `nX` – lokalnej czy modułowej - odnosi się instrukcja `std::cout`? Inaczej mówiąc, czy program wypisze liczbę `10` czy `5`? A może w ogóle się nie skompiluje?…
Zjawisko to nazywamy **przesłanianiem nazw** (ang. *name shadowing*), a pojawiło się ono wraz ze wprowadzeniem idei zasięgu zmiennych. Tego rodzaju kolizja oznaczeń nie powoduje w C++[23] błędu kompilacji, gdyż jest ona rozwiązywana w nieco inny sposób:

Konflikt nazw zmiennych o różnym zasięgu jest rozstrzygany zawsze na korzyść zmiennej o **węższym** zakresie.

Zazwyczaj oznacza to zmienną lokalną i tak też jest w naszym przypadku. Nie oznacza to jednak, że jej modułowy imiennik jest w funkcji `main()` niedostępny. Sposób odwołania się do niego ilustruje poniższy przykładowy program:

```
// Shadowing - przesłanianie nazw

int nX = 4;

void main()
{
```

[23] A także w większości współczesnych języków programowania

```
    int nX = 7;
    std::cout << "Lokalna zmienna nX: " << nX << std::endl;
    std::cout << "Modulowa zmienna nX: " << ::nX << std::endl;

    getch();
}
```

Pierwsze odniesienie do `nX` w funkcji `main()` odnosi się wprawdzie do zmiennej lokalnej, lecz jednocześnie możemy odwołać się także do tej modułowej. Robimy to bowiem w następnej linijce:

```
std::cout << "Modulowa zmienna nX: " << ::nX << std::endl;
```

Poprzedzamy tu nazwę zmiennej dwoma znakami dwukropka `::`. Jest to tzw. **operator zasięgu**. Wstawienie go mówi kompilatorowi, aby użył zmiennej globalnej zamiast lokalnej - czyli zrobił dokładnie to, o co nam chodzi :)

Operator ten ma też kilka innych zastosowań, o których powiemy niedługo (dokładniej przy okazji klas).

Chociaż C++ udostępnia nam tego rodzaju mechanizm[24], do dobrej praktyki programistycznej należy **niestosowanie** go. Identyczne nazwy wprowadzają bowiem zamęt i pogarszają czytelność kodu.
Dlatego też do nazw zmiennych modułowych dodaje się zazwyczaj przedrostek[25] `g_` (od *global*), co pozwala łatwo odróżnić je od lokalnych. Po zastosowaniu tej reguły nasz przykład wyglądałby mniej więcej tak:

```
int g_nX = 4;

void main()
{
    int nX = 7;
    std::cout << "Lokalna zmienna: " << nX << std::endl;
    std::cout << "Modulowa zmienna: " << g_nX << std::endl;

    getch();
}
```

Nie ma już potrzeby stosowania mało czytelnego operatora `::` i całość wygląda przejrzyście i profesjonalnie ;)

***

Zapoznaliśmy się zatem z niełatwą ideą zasięgu zmiennych. Jest to jednocześnie bardzo ważne pojęcie, które trzeba dobrze znać, by nie popełniać trudnych do wykrycia błędów. Mam nadzieję, że jego opis oraz przykłady były na tyle przejrzyste, że nie miałeś poważniejszych kłopotów ze zrozumieniem tego aspektu programowania.

Modyfikatory zmiennych

W aktualnym podrozdziale szczególnie upodobaliśmy sobie deklaracje zmiennych. Oto bowiem omówimy kolejne zagadnienie z nimi związane – tak zwane modyfikatory

[24] Większość języków go nie posiada!
[25] Jest to element notacji węgierskiej, aczkolwiek szeroko stosowany przez wielu programistów. Więcej informacji w Dodatku A.

(ang. *modifiers*). Są to mianowicie dodatkowe określenia umieszczane w deklaracji zmiennej, nadające jej pewne specjalne własności.

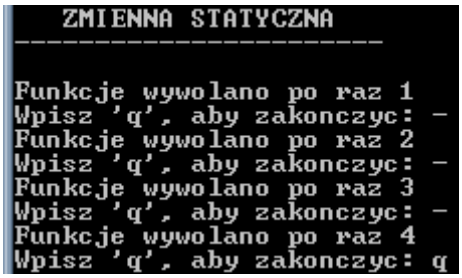
Zajmiemy się dwoma spośród trzech dostępnych w C++ modyfikatorów. Pierwszy – `static` – chroni zmienną przed utratą wartości po opuszczeniu jej zakresu przez program. Drugi zaś – znany nam `const` – oznacza stałą, opisaną już jakiś czas temu.

Zmienne statyczne

Kiedy aplikacja opuszcza zakres zmiennej lokalnej, wtedy ta jest usuwana z pamięci. To całkowicie naturalne – po co zachowywać zmienną, do której i tak nie byłoby dostępu? Logiczniejsze jest zaoszczędzenie pamięci operacyjnej i pozbycie się nieużywanej wartości, co też program skrzętnie czyni. Z tego powodu przy ponownym wejściu w porzucony wcześniej zasięg wszystkie podlegające mu zmienne będą ustawione na swe początkowe wartości.
Niekiedy jest to zachowanie niepożądane – czasem wolelibyśmy, aby zmienne lokalne nie traciły swoich wartości w takich sytuacjach. Najlepszym rozwiązaniem jest wtedy użycie modyfikatora `static`. Rzućmy okiem na poniższy przykład:

```
// Static - zmienne statyczne

void Funkcja()
{
   static int nLicznik = 0;

   ++nLicznik;
   std::cout << "Funkcje wywolano po raz " << nLicznik << std::endl;
}

void main()
{
   std::string strWybor;
   do
   {
         Funkcja();

         std::cout << "Wpisz 'q', aby zakonczyc: ";
         std::cin >> strWybor;
   } while (strWybor != "q");
}
```

Ów program jest raczej trywialny i jego jedynym zadaniem jest kilkukrotne uruchomienie podprogramu `Funkcja()`, dopóki życzliwy użytkownik na to pozwala :) We wnętrzu tejże funkcji mamy zadeklarowaną zmienną statyczną, która służy tam jako licznik uruchomień.



Screen 24. Zliczanie wywołań funkcji przy pomocy zmiennej statycznej

Jego wartość jest **zachowywana** pomiędzy kolejnymi wywołaniami funkcji, gdyż istnieje w pamięci przez cały czas działania aplikacji[26]. Możemy więc każdorazowo inkrementować tą wartość i pokazywać jako ilość uruchomień funkcji. Tak właśnie działają zmienne statyczne :)

Deklaracja takiej zmiennej jest, jak widzieliśmy, nad wyraz prosta:

```
static int nLicznik = 0;
```

Wystarczy poprzedzić oznaczenie jej typu słówkiem `static` i *voila* :) Nadal możemy także stosować inicjalizację do ustawienia początkowej wartości zmiennej.

Jest to wręcz konieczne – gdybyśmy bowiem zastosowali zwykłe przypisanie, odbywałoby się ono przy każdym wejściu w zasięg zmiennej. Wypaczałoby to całkowicie sens stosowania modyfikatora `static`.

Stałe

Stałe omówiliśmy już wcześniej, więc nie są dla ciebie nowością. Obecnie podkreślimy ich związek ze zmiennymi.
Jak (mam nadzieję) pamiętasz, aby zadeklarować stałą należy użyć słowa `const`, na przykład:

```
const float GRAWITACJA = 9.80655;
```

`const`, podobnie jak `static`, jest modyfikatorem zmiennej. Stałe posiadają zatem wszystkie cechy zmiennych, takie jak typ czy zasięg. Jedyną różnicą jest oczywiście niemożność zmiany wartości stałej.

***

Tak oto uzupełniliśmy swe wiadomości na temat zmiennych o ich zasięg oraz modyfikatory. Uzbrojeni w tą nową wiedzę możemy teraz śmiało podążać dalej :D

Typy zmiennych

W C++ typ zmiennej jest sprawą niezwykle ważną. Gdy określamy go przy deklaracji, zostaje on trwale „przywiązany” do zmiennej na cały czas działania programu. Nie może więc zajść sytuacja, w której zmienna zadeklarowana na przykład jako liczba całkowita zawiera informację tekstową czy liczbę rzeczywistą.

Niektórych języki programowania pozwalają jednak na to. Delphi i Visual Basic są wyposażone w specjalny typ `Variant`, który potrafi przechowywać zarówno dane liczbowe, jak i tekstowe. PHP natomiast w ogóle nie wymaga podawania typu zmiennych.

Chociaż wymóg ten wygląda na poważny mankament C++, w rzeczywistości wcale nim nie jest. Bardzo trudno wskazać czynność, która wymagałaby zmiennej „uniwersalnego typu”, mogącej przechowywać każdy rodzaj danych. Jeżeli nawet zaszłaby takowa konieczność, możliwe jest zastosowanie przynajmniej kilku niemal równoważnych

[26] Dokładniej mówiąc: od momentu deklaracji do zakończenia programu

rozwiązań[27].

Generalnie jednak jesteśmy „skazani” na korzystanie z typów zmiennych, co mimo wszystko nie powinno nas smucić :) Na osłodę proponuję bliższe przyjrzenie się im. Będziemy mieli okazję zobaczyć, że ich możliwości, elastyczność i zastosowania są niezwykle szerokie.

Modyfikatory typów liczbowych

Dotychczas w swoich programach mieliśmy okazję używać głównie typu `int`, reprezentującego liczbę całkowitą. Czasem korzystaliśmy także z `float`, będącego typem liczb rzeczywistych.
Dwa sposoby przechowywania wartości liczbowych to, zdawałoby się, bardzo niewiele. Zważywszy, iż spora część języków programowania udostępnia nawet po kilkanaście takich typów, asortyment C++ może wyglądać tutaj wyjątkowo mizernie.

Domyślasz się zapewne, że jest to tylko złudne wrażenie :) Do każdego typu liczbowego w C++ możemy bowiem dołączyć jeden lub kilka **modyfikatorów**, które istotnie zmieniają jego własności. Spróbujmy dokładnie przyjrzeć się temu mechanizmowi.

Typy ze znakiem i bez znaku

Typ liczbowy `int` może nam przechowywać zarówno liczby dodatnie, jak i ujemne. Dosyć często jednak nie potrzebujemy wartości mniejszych od zera. Przykładowo, ilość punktów w większości gier nigdy nie będzie ujemna; to samo dotyczy liczników upływającego czasu, zmiennych przechowujących wielkość plików, długości odcinków, rozmiary obrazków - i tak dalej.

Możemy rzecz jasna zwyczajnie zignorować obecność liczb ujemnych i korzystać jedynie z wartości dodatnich. Wadą tego rozwiązania jest marnotrawstwo: tracimy wtedy połowę miejsca zajmowanego w pamięci przez zmienną. Jeżeli na przykład `int` mógłby zawierać liczby od -10 000 do +10 000 (czyli 20 000 możliwych wartości[28]), to ograniczylibyśmy ten przedział do 0…+10 000 (a więc skromnych 10 000 możliwych wartości).
Nie jest to może karygodna niegospodarność w przypadku jednej zmiennej, ale gdy mówimy o kilku czy kilkunastu tysiącach podobnych zmiennych[29], ilość zmarnowanej pamięci staje się znaczna.

Należałoby zatem powiedzieć kompilatorowi, że nie potrzebujemy liczb ujemnych i w zamian za nie chcemy zwiększenia przedziału liczb dodatnich. Czynimy to poprzez dodanie do typu zmiennej `int` modyfikatora `unsigned` (‘nieoznakowany’, czyli bez znaku; zawsze dodatni). Deklaracja będzie wtedy wyglądać na przykład tak:

```
unsigned int uZmienna;     // przechowuje liczby naturalne
```

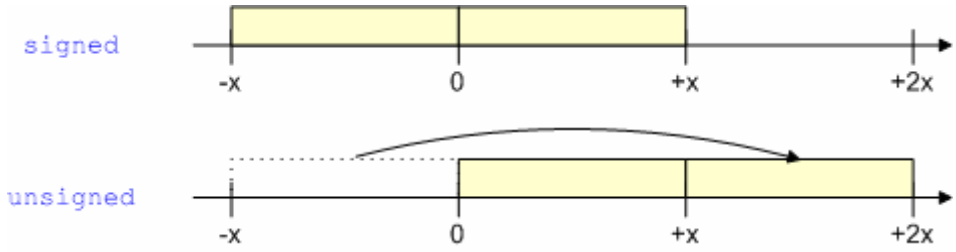
Analogicznie, moglibyśmy dodać przeciwstawny modyfikator `signed` (‘oznakowany’, czyli ze znakiem; dodatni lub ujemny) do typów zmiennych, które mają zawierać zarówno liczby dodatnie, jak i ujemne:

```
signed int nZmienna;       // przechowuje liczby całkowite
```

[27] Można wykorzystać chociażby szablony, unie czy wskaźniki. O każdym z tych elementów C++ powiemy sobie w dalszej części kursu, więc cierpliwości ;)
[28] To oczywiście jedynie przykład. Na żadnym współczesnym systemie typ `int` nie ma tak małego zakresu.
[29] Co nie jest wcale niemożliwe, a przy stosowaniu tablic (opisanych w następnym rozdziale) staje całkiem częste.

Zazwyczaj tego nie robimy, gdyż modyfikator ten jest niejako domyślnie tam umieszczony i nie ma potrzeby jego wyraźnego stosowania.

Jako podsumowanie proponuję diagram obrazujący działanie poznanych modyfikatorów:



Schemat 6. Przedział wartości typów liczbowych ze znakiem (`signed`) i bez znaku (`unsigned`)

Widzimy, że zastosowanie `unsigned` powoduje „przeniesienie" ujemnej połowy przedziału zmiennej bezpośrednio za jej część dodatnią. Nie mamy wówczas możliwości korzystania z liczb ujemnych, ale w zamian otrzymujemy dwukrotnie więcej miejsca na wartości dodatnie. Tak to już jest w programowaniu, że nie ma nic za darmo :D

Rozmiar typu całkowitego

W poprzednim paragrafie wspominaliśmy o przedziale dopuszczalnych wartości zmiennej, ale nie przyglądaliśmy się bliżej temu zagadnieniu. Teraz zatrzymamy się na nim trochę dłużej i zajmiemy rozmiarem zmiennych całkowitych.

Wiadomo nam doskonale, że pamięć komputera jest ograniczona, zatem miejsce zajmowane w tej pamięci przez każdą zmienną jest również limitowane. W przypadku typów liczbowych przejawia się to ograniczonym przedziałem wartości, które mogą przyjmować zmienne należące do takich typów.

Jak duży jest to przedział? Nie ma uniwersalnej odpowiedzi na to pytanie. Okazuje się bowiem, że rozmiar typu `int` jest **zależny od kompilatora**. Wpływ na tę wielkość ma pośrednio system operacyjny oraz procesor komputera.
Nasz kompilator (Visual C++ .NET), podobnie jak wszystkie tego typu narzędzia pracujące w systemie Windows 95 i wersjach późniejszych, jest **32-bitowy**. Oznacza to między innymi, że typ `int` ma u nas wielkość równą 32 bitom właśnie, a więc w przeliczeniu[30] **4 bajtom**.

Cztery bajty to cztery znaki (na przykład cyfry) – czyżby zatem największymi i najmniejszymi możliwymi do zapisania wartościami były +9999 i -9999?…
Oczywiście, że nie! Komputer przechowuje liczby w znacznie efektywniejszej postaci dwójkowej. Wykorzystanie każdego bitu sprawia, że granice przedziału wartości typu `int` to aż $\pm 2^{31}$ – nieco ponad **dwa miliardy**!

Więcej informacji na temat sposobu przechowywania danych w pamięci operacyjnej możesz znaleźć w Dodatku B, *Reprezentacja danych w pamięci*.

Przedział ten sprawdza się dobrze w wielu zastosowaniach. Czasem jednak jest on zbyt mały (tak, to możliwe :D) lub zwyczajnie zbyt duży. Daje się to odczuć na przykład przy odczytywaniu plików, w których każda wartość zajmuje obszar o ściśle określonym rozmiarze, nie zawsze równym `int`'owym 4 bajtom (tzw. plików binarnych).

[30] 1 bajt to 8 bitów.

Dlatego też C++ udostępnia nam poręczny zestaw dwóch modyfikatorów, którymi możemy wpływać na wielkość typu całkowitego. Są to: `short` ('krótki') oraz `long` ('długi'). Używamy ich podobnie jak `signed` i `unsigned` – poprzedzając typ `int` którymś z nich:

```
short int nZmienna;         // "krótka" liczba całkowita
long int nZmienna;          // "długa" liczba całkowita
```

Cóż znaczą jednak te, nieco żartobliwe, określenia „krótkiej" i „długiej" liczby? Chyba najlepszą odpowiedzią będzie tu… stosowna tabelka :)

nazwa	*rozmiar*	*przedział wartości*
`int`	4 bajty	od -2^{31} do $+2^{31} - 1$
`short int`	2 bajty	od -32 768 do +32 767
`long int`	4 bajty	od -2^{31} do $+2^{31} - 1$

Tabela 4. Typy całkowite w 32-bitowym Visual C++ .NET[31]

Niespodzianką może być brak typu o rozmiarze 1 bajta. Jest on jednak obecny w C++ – to typ… `char` :) Owszem, reprezentuje on **znak**. Nie zapominajmy jednak, że komputer operuje na znakach jak na odpowiadającym im **kodom liczbowym**. Dlatego też typ `char` jest w istocie także typem liczb całkowitych!

Visual C++ udostępnia też nieco lepszy sposób na określenie wielkości typu liczbowego. Jest nim użycie frazy `__int`*n*, gdzie *n* oznacza rozmiar zmiennej w bitach. Oto przykłady:

```
__int8 nZmienna;      // 8 bitów == 1 bajt, wartości od -128 do 127
__int16 nZmienna;     // 16 bitów == 2 bajty, wartości od -32768 do 32767
__int32 nZmienna;     // 32 bity == 4 bajty, wartości od -2^31 do 2^31 - 1
__int64 nZmienna;     // 64 bity == 8 bajtów, wartości od -2^63 do 2^63 - 1
```

`__int8` jest więc równy typowi `char`, `__int16` – `short int`, a `__int32` – `int` lub `long int`. Gigantyczny typ `__int64` nie ma natomiast swojego odpowiednika.

Precyzja typu rzeczywistego

Podobnie jak w przypadku typu całkowitego `int`, typ rzeczywisty `float` posiada określoną rozpiętość wartości, które można zapisać w zmiennych o tym typie. Ponieważ jednak jego przeznaczeniem jest przechowywanie wartości ułamkowych, pojawia się kwestia **precyzji** zapisu takich liczb.

Szczegółowe wyjaśnienie sposobu, w jaki zmienne rzeczywiste przechowują wartości, jest dość skomplikowane i dlatego je sobie darujemy[32] :) Najważniejsze są dla nas wynikające z niego konsekwencje. Otóż:

> Precyzja zapisu liczby w zmiennej typu rzeczywistego **maleje** wraz ze **wzrostem wartości** tej liczby

Przykładowo, duża liczba w rodzaju `1000000.0023` zostanie najpewniej zapisana bez części ułamkowej. Natomiast mała wartość, jak `1.43525667` będzie przechowana z dużą

[31] To zastrzeżenie jest konieczne. Wprawdzie `int` zajmuje 4 bajty we wszystkich 32-bitowych kompilatorach, ale w przypadku pozostałych typów może być inaczej! Standard C++ wymaga jedynie, aby `short int` był mniejszy lub równy od `int`'a, a `long int` większy lub równy `int`'owi.
[32] Zainteresowanych odsyłam do Dodatku B.

dokładnością, z kilkoma cyframi po przecinku. Ze względu na tę właściwość (zmienną precyzję) typy rzeczywiste nazywamy często **zmiennoprzecinkowymi**.

Zgadza się – typy. Podobnie jak w przypadku liczb całkowitych możemy dodać do typu `float` odpowiednie modyfikatory. I podobnie jak wówczas, ujrzymy je w należytej tabelce :)

nazwa	*rozmiar*	*precyzja*
`float`	4 bajty	6–7 cyfr
`double float`	8 bajtów	15-16 cyfr

Tabela 5. Typy zmiennoprzecinkowe w C++

`double` ('podwójny'), zgodnie ze swoją nazwą, zwiększa dwukrotnie rozmiar zmiennej oraz poprawia jej dokładność. Tak zmodyfikowana zmienna jest nazywana czasem liczbą **podwójnej precyzji** - w odróżnieniu od `float`, która ma tylko **pojedynczą precyzję**.

Skrócone nazwy

Na koniec warto nadmienić jeszcze o możności skrócenia nazw typów zawierających modyfikatory. W takich sytuacjach możemy bowiem całkowicie **pominąć** słowa `int` i `float`.
Przykładowe deklaracje:

```
unsigned int uZmienna;
short int nZmienna;
unsigned long int nZmienna;
double float fZmienna;
```

mogą zatem wyglądać tak:

```
unsigned uZmienna;
short nZmienna;
unsigned long nZmienna;
double fZmienna;
```

Mała rzecz, a cieszy ;) Mamy też kolejny dowód na dużą kondensację składni C++.

***

Poznane przed chwilą modyfikatory umożliwiają nam większą kontrolę nad zmiennymi w programie. Pozwalają bowiem na dokładne określenie, **jaką** zmienną chcemy w danej chwili zadeklarować i nie dopuszczają, by kompilator myślał za nas ;D

Pomocne konstrukcje

Zapoznamy się teraz z dwoma elementami języka C++, które ułatwiają nieco pracę z różnymi typami zmiennych. Będzie to instrukcja `typedef` oraz operator `sizeof`.

Instrukcja `typedef`

Wprowadzenie modyfikatorów sprawiło, że oto mamy już nie kilka, a przynajmniej kilkanaście typów zmiennych. Nazwy tychże typów są przy tym dosyć długie i wielokrotne ich wpisywanie może nam zabierać dużo czasu. Zbyt dużo.

Dlatego też (i nie tylko dlatego) C++ posiada instrukcję `typedef` (ang. *type definition* – definicja typu). Możemy jej użyć do nadania **nowej nazwy** (**aliasu**) dla już **istniejącego typu**. Zastosowanie tego mechanizmu może wyglądać choćby tak:

```
typedef unsigned int UINT;
```

Powyższa linijka kodu mówi kompilatorowi, że od tego momentu typ `unsigned int` posiada także dodatkową nazwę - `UINT`. Staję się ona dokładnym synonimem pierwotnego określenia. Odtąd bowiem obie deklaracje

```
unsigned int uZmienna;
```

oraz

```
UINT uZmienna;
```

są w pełni **równoważne**.

Użycie `typedef`, podobnie jak jej składnia, jest bardzo proste:

```
typedef typ nazwa;
```

Skutkiem skorzystania z tej instrukcji jest możliwość wstawiania nowej *`nazwy`* tam, gdzie wcześniej musieliśmy zadowolić się jedynie starym *`typem`*. Obejmuje to zarówno deklaracje zmiennych, jak i parametrów funkcji tudzież zwracanych przez nie wartości. Dotyczy więc **wszystkich** sytuacji, w których mogliśmy korzystać ze starego typu – nowa nazwa nie jest pod tym względem w żaden sposób ułomna.

Jaka jest praktyczna korzyść z definiowania własnych określeń dla istniejących typów? Pierwszą z nich jest przytoczone wcześniej skracanie nazw, które z pewnością pozytywnie wpłynie na stan naszych klawiatur ;)) Oszczędnościowe „przydomki" w rodzaju zaprezentowanego wyżej `UINT` są przy tym na tyle wygodne i szeroko wykorzystywane, że niektóre kompilatory (w tym i nasz Visual C++) nie wymagają nawet ich jawnego określenia!
Możliwość dowolnego oznaczania typów pozwala również na nadawanie im znaczących nazw, które obrazują ich zastosowania w aplikacji. Z przykładem podobnego postępowania spotkasz się przy tworzeniu programów okienkowych w Windows. Używa się tam wielu typów o nazwach takich jak `HWND`, `HINSTANCE`, `WPARAM`, `LRESULT` itp., z których każdy jest jedynie aliasem na 32-bitową liczbę całkowitą bez znaku. Stosowanie takiego nazewnictwa poważnie poprawia czytelność kodu – oczywiście pod warunkiem, że znamy znaczenie stosowanych nazw :)

Zauważmy pewien istotny fakt. Mianowicie, `typedef` nie tworzy nam żadnych **nowych typów**, a jedynie duplikuje **już istniejące**. Zmiany, które czyni w sposobie programowania, są więc *stricte* kosmetyczne, choć na pierwszy rzut oka mogą wyglądać na dość znaczne.
Do kreowania zupełnie nowych typów służą inne elementy języka C++, z których część poznamy w następnym rozdziale.

Operator `sizeof`

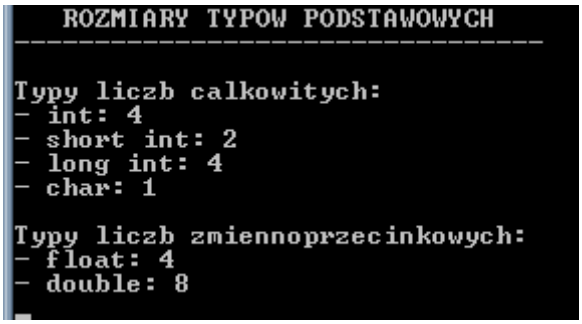
Przy okazji prezentacji różnych typów zmiennych podawałem zawsze ilość bajtów, którą zajmuje w pamięci każdy z nich. Przypominałem też kilka razy, że wielkości te są prawdziwe jedynie w przypadku kompilatorów 32-bitowych, a niektóre nawet tylko w Visual C++.

Z tegoż powodu mogą one szybko stać się po prostu nieaktualne. Przy dzisiejszym tempie postępu technicznego, szczególnie w informatyce, wszelkie zmiany dokonują się w zasadzie nieustannie[33]. W tej gonitwie także programiści nie mogą pozostawać w tyle – w przeciwnym wypadku przystosowanie ich starych aplikacji do nowych warunków technologicznych może kosztować mnóstwo czasu i wysiłku.
Jednocześnie wiele programów opiera swe działanie na rozmiarze typów podstawowych. Wystarczy napomknąć o tak częstej czynności, jak zapisywanie danych do plików albo przesyłanie ich poprzez sieć. Jeśliby każdy program musiał mieć wpisane „na sztywno" rzeczone wielkości, wtedy spora część pracy programistów upływałaby na dostosowywaniu ich do potrzeb nowych platform sprzętowych, na których miałyby działać istniejące aplikacje. A co z tworzeniem całkiem nowych produktów?…

Szczęśliwie twórcy C++ byli na tyle zapobiegliwi, żeby uchronić nas, koderów, od tej koszmarnej perspektywy. Wprowadzili bowiem operator `sizeof` ('rozmiar czegoś'), który pozwala na uzyskanie wielkości zmiennej (lub jej typu) **w trakcie działania** programu. Spojrzenie na poniższy przykład powinno nam przybliżyć funkcjonowanie tego operatora:

```
// Sizeof - pobranie rozmiaru zmiennej lub typu

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

void main()
{
   std::cout << "Typy liczb calkowitych:" << std::endl;
   std::cout << "- int: " << sizeof(int) << std::endl;
   std::cout << "- short int: " << sizeof(short int) << std::endl;
   std::cout << "- long int: " << sizeof(long int) << std::endl;
   std::cout << "- char: " << sizeof(char) << std::endl;
   std::cout << std::endl;

   std::cout << "Typy liczb zmiennoprzecinkowych:" << std::endl;
   std::cout << "- float: " << sizeof(float) << std::endl;
   std::cout << "- double: " << sizeof(double) << std::endl;

   getch();
}
```

Uruchomienie programu z listingu powyżej, jak słusznie można przypuszczać, będzie nam skutkowało krótkim zestawieniem rozmiarów typów podstawowych.



Screen 25. `sizeof` w akcji

[33] W chwili pisania tych słów – pod koniec roku 2003 – mamy już coraz wyraźniejsze widoki na poważne wykorzystanie procesorów 64-bitowych w domowych komputerach. Jednym ze skutków tego „zwiększenia bitowości" będzie zmiana rozmiaru typu liczbowego `int`.

Po uważnym zlustrowaniu kodu źródłowego widać jak na dłoni działanie oraz sposób użycia operatora `sizeof`. Wystarczy podać mu typ lub zmienną jako parametr, by otrzymać w wyniku jego rozmiar w bajtach[34]. Potem możemy zrobić z tym rezultatem dokładnie to samo, co z każdą inną liczbą całkowitą – chociażby wyświetlić ją w konsoli przy użyciu strumienia wyjścia.

Zastosowanie `sizeof` nie ogranicza się li tylko do typów wbudowanych. Gdy w kolejnych rozdziałach nauczymy się tworzyć własne typy zmiennych, będziemy mogli w identyczny sposób ustalać ich rozmiary przy pomocy poznanego przed momentem operatora. Nie da się ukryć, że bardzo lubimy takie uniwersalne rozwiązania :D

Wartość, którą zwraca operator `sizeof`, należy do specjalnego typu `size_t`. Zazwyczaj jest on tożsamy z `unsigned int`, czyli liczbą **bez znaku** (bo przecież rozmiar nie może być ujemny). Należy więc uważać, aby nie przypisywać jej do zmiennej, która jest liczbą ze znakiem.

Rzutowanie

Idea typów zmiennych wprowadza nam pewien sposób klasyfikacji wartości. Niektóre z nich uznajemy bowiem za liczby całkowite (`3`, `-17`, `44`, `67*88` itd.), inne za zmiennoprzecinkowe (`7.189`, `12.56`, `-1.41`, `8.0` itd.), jeszcze inne za tekst (`"ABC"`, `"Hello world!"` itp.) czy pojedyncze znaki[35] (`'F'`, `'@'` itd.).
Każdy z tych rodzajów odpowiada nam któremuś z poznanych typów zmiennych. Najczęściej też nie są one ze sobą kompatybilne – innymi słowy, „nie pasują” do siebie, jak chociażby tutaj:

```
int nX = 14;
int nY = 0.333 * nX;
```

Wynikiem działania w drugiej linijce będzie przecież liczba rzeczywista z częścią ułamkową, którą nijak nie można wpasować w ciasne ramy typu `int`, zezwalającego jedynie na wartości całkowite[36].

Oczywiście, w podanym przykładzie wystarczy zmienić typ drugiej zmiennej na `float`, by rozwiązać nurtujący nas problem. Nie zawsze jednak będziemy mogli pozwolić sobie na podobne kompromisy, gdyż często jedynym wyjściem stanie się „wymuszenie” na kompilatorze zaakceptowania kłopotliwego kodu.
Aby to uczynić, musimy **rzutować** (ang. *cast*) przypisywaną wartość **na** docelowy typ – na przykład `int`. Rzutowanie działa trochę na zasadzie umowy z kompilatorem, która w naszym przypadku mogłaby brzmieć tak: „Wiem, że naprawdę jest to liczba zmiennoprzecinkowa, ale właśnie **tutaj** chcę, aby stała się liczbą całkowitą typu `int`, bo muszę ją przypisać do zmiennej tego typu”. Takie porozumienie wymaga ustępstw od obu stron – kompilator musi „pogodzić się” z chwilowym zaprzestaniem kontroli typów, a programista powinien liczyć się z ewentualną utratą części danych (w naszym przykładzie poświęcimy cyfry po przecinku).

---

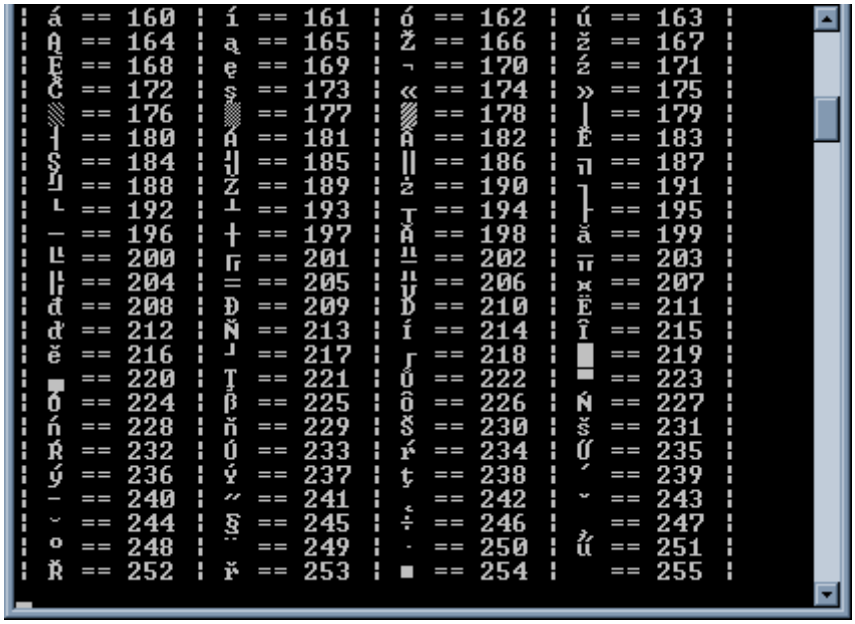
[34] Ściślej mówiąc, `sizeof` podaje nam rozmiar obiektu w stosunku do wielkości typu `char`. Jednakże typ ten ma najczęściej wielkość dokładnie 1 bajta, zatem utarło się stwierdzenie, iż `sizeof` zwraca w wyniku ilość bajtów. Nie ma w zasadzie żadnego powodu, by uznać to za błąd.
[35] Znaki są typu `char`, który jak wiemy jest także typem liczbowym. W C++ kod znaku jest po prostu jednoznaczny z nim samym, dlatego możemy go interpretować zarówno jako symbol, jak i wartość liczbową.
[36] Niektóre kompilatory (w tym i Visual C++) zaakceptują powyższy kod, jednakże nie obejdzie się bez ostrzeżeń o możliwej (i faktycznej!) utracie danych. Wprawdzie niektórzy nie przejmują się w ogóle takimi ostrzeżeniami, my jednak nie będziemy tak krótkowzroczni :D

Proste rzutowanie

Zatem do dzieła! Zobaczmy, jak w praktyce wyglądają takie „negocjacje” :) Zostawimy na razie ten trywialny, dwulinijkowy przykład (wrócimy jeszcze do niego) i zajmiemy się poważniejszym programem. Oto i on:

```
// SimpleCast - proste rzutowanie typów

void main()
{
    for (int i = 32; i < 256; i += 4)
    {
          std::cout << "| " << (char) (i) << " == " << i << " | ";
          std::cout << (char) (i + 1) << " == " << i + 1 << " | ";
          std::cout << (char) (i + 2) << " == " << i + 2 << " | ";
          std::cout << (char) (i + 3) << " == " << i + 3 << " |";
          std::cout << std::endl;
    }

    getch();
}
```

Huh, faktycznie nie jest to banalny kod :) Wykonywana przezeń czynność jest jednak dość prosta. Aplikacja ta pokazuje nam tablicę kolejnych znaków wraz z odpowiadającymi im kodami ANSI.



Screen 26. Fragment tabeli ANSI

Najważniejsza jest tu dla nas sama operacja rzutowania, ale warto przyjrzeć się funkcjonowaniu programu jako całości.
Zawarta w nim pętla `for` wykonuje się dla co czwartej wartości licznika z przedziału od `32` do `255`. Skutkuje to faktem, iż znaki są wyświetlane wierszami, po 4 w każdym.

Pomijamy znaki o kodach mniejszych od 32 (czyli te z zakresu 0…31), ponieważ są to specjalne symbole sterujące, zasadniczo nieprzeznaczone do wyświetlania na ekranie. Znajdziemy wśród nich na przykład tabulator (kod 9), znak „powrotu karetki” (kod 13), końca wiersza (kod 10) czy sygnał błędu (kod 7).

Za prezentację pojedynczego wiersza odpowiadają te wielce interesujące instrukcje:

```
std::cout << "| " << (char) (i)     << " == " << i     << " | ";
std::cout <<         (char) (i + 1) << " == " << i + 1 << " | ";
std::cout <<         (char) (i + 2) << " == " << i + 2 << " | ";
std::cout <<         (char) (i + 3) << " == " << i + 3 << " |";
```

Sądząc po widocznym ich efekcie, każda z nich wyświetla nam jeden znak oraz odpowiadający mu kod ANSI. Przyglądając się bliżej temu listingowi, widzimy, że zarówno pokazanie znaku, jak i przynależnej mu wartości liczbowej odbywa się zawsze przy pomocy **tego samego** wyrażenia. Jest nim odpowiednio `i`, `i + 1`, `i + 2` lub `i + 3`.

Jak to się dzieje, że raz jest ono interpretowane jako **znak**, a innym razem jako **liczba**? Domyślasz się zapewne niebagatelnej roli rzutowania w działaniu tej „magii" :) Istotnie, jest ono konieczne. Jako że licznik `i` jest zmienną typu `int`, zacytowane wyżej cztery wyrażenia także należą do tego typu. Przesłanie ich do strumienia wyjścia w niezmienionej postaci powoduje wyświetlenie ich wartości w formie liczb. W ten sposób pokazujemy kody ANSI kolejnych znaków.
Aby wyświetlić same symbole musimy jednak oszukać nieco nasz strumień `std::cout`, rzutując wspomniane wartości liczbowe na typ `char`. Dzięki temu zostaną one potraktowane jako znaki i takoż wyświetlone w konsoli.

Zobaczmy, w jaki sposób realizujemy tutaj to osławione rzutowanie. Spójrzmy mianowicie na jeden z czterech podobnych kawałków kodu:

```
(char) (i + 1)
```

Ten niepozorny fragment wykonuje całą ważką operację, którą nazywamy rzutowaniem. Zapisanie w nawiasach nazwy typu `char` przed wyrażeniem `i + 1` (dla jasności umieszczonym również w nawiasach) powoduje bowiem, iż wynik tak ujętego działania zostaje uznany jako podpadający pod typ `char`. Tak jest też traktowany przez strumień wyjścia, dzięki czemu możemy go oglądać jako znak, a nie liczbę.

Zatem, aby rzutować jakieś wyrażenie na wybrany typ, musimy użyć niezwykle prostej konstrukcji:

```
(typ) wyrażenie
```

wyrażenie może być przy tym ujęte w nawias lub nie; zazwyczaj jednak stosuje się nawiasy, by uniknąć potencjalnych kłopotów z kolejnością operatorów.

Można także użyć składni *typ*`(`*wyrażenie*`)`. Stosuje się ją rzadziej, gdyż przypomina wywołanie funkcji i może być przez to przyczyną pomyłek.

Wróćmy teraz do naszego pierwotnego przykładu. Rozwiązanie problemu, który wcześniej przedstawiał, powinno być już banalne:

```
int nX = 14;
int nY = (int) (0.333 * nX);
```

Po takich manipulacjach zmienna `nY` będzie przechowywała część całkowitą z wyniku podanego mnożenia. Oczywiście tracimy w ten sposób dokładność obliczeń, co jest jednak nieuniknioną ceną kompromisu towarzyszącego rzutowaniu :)

Operator `static_cast`

Umiemy już dokonywać rzutowania, poprzedzając wyrażenie nazwą typu napisaną w nawiasach. Taki sposób postępowania wywodzi się jeszcze z zamierzchłych czasów języka C[37], poprzednika C++. Czyżby miało to znaczyć, że jest on zły?…

Powiedzmy, że nie jest **wystarczająco dobry** :) Nie przeczę, że na początku może wydawać się świetnym rozwiązaniem – klarownym, prostym, niewymagającym wiele pisania etc. Jednak im dalej w las, tym więcej śmieci: już teraz dokładniejsze spojrzenie ujawnia nam wiele mankamentów, a w miarę zwiększania się twoich umiejętności i wiedzy dostrzeżesz ich jeszcze więcej.

Spójrzmy choćby na samą składnię. Oprócz swojej niewątpliwej prostoty posiada dwie zdecydowanie nieprzyjemne cechy.
Po pierwsze, zwiększa nam ilość nawiasów w wyrażeniach, które zawierają rzutowanie. A przecież nawet i bez niego potrafią one być dostatecznie skomplikowane. Częste przecież użycie kilku operatorów, kilku funkcji (z których każda ma pewnie po kilka parametrów) oraz kilku dodatkowych nawiasów (aby nie kłopotać się kolejnością działań) gmatwa nasze wyrażenia w dostatecznym już stopniu. Jeżeli dodamy do tego jeszcze parę rzutowań, może nam wyjść coś w tym rodzaju:

```
int nX = (int) (((2 * nY) / (float) (nZ + 3)) - (int) Funkcja(nY * 7));
```

Konwersje w formie `(typ) wyrażenie` z pewnością nie poprawiają tu czytelności kodu. Drugim problemem jest znowuż kolejność działań. Pytanie za pięć punktów: jaką wartość ma zmienna `nY` w poniższym fragmencie?

```
float fX = 0.75;
int nY = (int) fX * 3;
```

Zatem?… Jeżeli obecne w drugiej linijce rzutowanie na `int` dotyczy jedynie zmiennej `fX`, to jej wartość (`0.75`) zostanie zaokrąglona do zera, zatem `nY` będzie przypisane również zero. Jeśli jednak konwersji na `int` zostanie poddane całe wyrażenie (`0.75 * 3`, czyli `2.25`), to `nY` przyjmie wartość `2`!
Wybrnięcie z tego dylematu to… kolejna para nawiasów, obejmująca tą część wyrażenia, którą faktycznie chcemy rzutować. Wygląda więc na to, że nie opędzimy się od częstego stosowania znaków `(` i `)`.

Składnia to jednak nie jedyny kłopot. Tak naprawdę o wiele ważniejsze są kwestie związane ze sposobem, w jaki jest realizowane samo rzutowanie. Niestety, na razie jesteś w niezbyt komfortowej sytuacji, gdyż musisz zaakceptować pewien fakt bez uzasadnienia („na wiarę” :D). Brzmi on następująco:

Rzutowanie w formie `(typ) wyrażenie`, zwane też rzutowaniem w stylu C, **nie jest zalecane** do stosowania w C++.

Dokładnie przyczyny takiego stanu rzeczy poznasz przy okazji omawiania klas i programowania obiektowego[38].

[37] Nazywa się go nawet rzutowaniem w stylu C.
[38] Dla szczególnie dociekliwych mam wszakże wyjaśnienie częściowe. Mianowicie, rzutowanie w stylu C nie rozróżnia nam tzw. bezpiecznych i niebezpiecznych konwersji. Za bezpieczną możemy uznać zamianę jednego typu liczbowego na drugi czy wskaźnika szczegółowego na wskaźnik bardziej ogólny (np. `int*` na `void*` - o wskaźnikach powiemy sobie szerzej, gdy już uporamy się z podstawami :)). Niebezpieczne rzutowanie to konwersja między niezwiązanymi ze sobą typami, na przykład liczbą i tekstem; w zasadzie nie powinno się takich rzeczy robić.

→

No dobrze, załóżmy, że uznajemy tą odgórną radę[39] i zobowiązujemy się nie stosować rzutowania „nawiasowego" w swoich programach. Czy to znaczy, że w ogóle tracimy możliwość konwersji zmiennych jednego typu na inne?!
Rzeczywistość na szczęście nie jest aż tak straszna :) C++ posiada bowiem aż cztery **operatory rzutowania**, które są najlepszym sposobem na realizację zamiany typów w tym języku. Będziemy sukcesywnie poznawać je wszystkie, a zaczniemy od najczęściej stosowanego – tytułowego `static_cast`.

`static_cast` ('rzutowanie statyczne') nie ma nic wspólnego z modyfikatorem `static` i zmiennymi statycznymi. Operator ten służy do przeprowadzania najbardziej pospolitych konwersji, które jednak są spotykane najczęściej. Możemy go stosować wszędzie, gdzie sposób zamiany jest oczywisty – zarówno dla nas, jak i kompilatora ;)

Najlepiej po prostu **zawsze** używać `static_cast`, uciekając się do innych środków, gdy ten zawodzi i nie jest akceptowany przez kompilator (albo wiąże się z pokazaniem ostrzeżenia).

W szczególności, możemy i powinniśmy korzystać ze `static_cast` przy rzutowaniu między typami podstawowymi. Zobaczmy zresztą, jak wyglądałoby ono dla naszego ostatniego przykładu:

```
float fX = 0.75;
int nY = static_cast<int>(fX * 3);
```

Widzimy, że użycie tego operatora od razu likwiduje nam niejednoznaczność, na którą poprzednio zwróciliśmy uwagę. Wyrażenie poddawane rzutowaniu musimy bowiem ująć w nawiasy okrągłe.
Ciekawy jest sposób zapisu nazwy typu, na który rzutujemy. Znaki < i >, oprócz tego że są operatorami mniejszości i większości, tworzą parę nawiasów ostrych. Pomiędzy nimi wpisujemy określenie docelowego typu.

Pełna składnia operatora `static_cast` wygląda więc następująco:

```
static_cast<typ>(wyrażenie)
```

Być może jest ona bardziej skomplikowana od „zwykłego" rzutowania, ale używając jej osiągamy wiele korzyści, o których mogłeś się naocznie przekonać :)

Warto też wspomnieć, że trzy pozostałe operatory rzutowania mają identyczną postać – oczywiście z wyjątkiem słowa `static_cast`, które jest zastąpione innym.

***

Tą uwagą kończymy omawianie różnych aspektów związanych z typami zmiennych w języku C++. Wreszcie zajmiemy się tytułowymi zagadnieniami tego rozdziału, czyli czynnościach, które możemy wykonywać na zmiennych.

Problem z rzutowaniem w stylu C polega na tym, iż zupełnie nie rozróżnia tych dwóch rodzajów zamiany. Pozostaje tak samo niewzruszone na niewinną konwersję z `float` na `int` oraz, powiedzmy, na zupełnie nienaturalną zmianę `std::string` na `bool`. Nietrudno domyśleć się, że zwiększa to prawdopodobieństwo występowania różnego rodzaju błędów.

[39] Jak wszystko, co dotyczy fundamentów języka C++, pochodzi ona od jego Komitetu Standaryzacyjnego.

Kalkulacje na liczbach

Poznamy teraz kilka standardowych operacji, które możemy wykonywać na danych liczbowych. Najpierw będą to odpowiednie funkcje, których dostarcza nam C++, a następnie uzupełnienie wiadomości o operatorach arytmetycznych. Zaczynajmy więc :)

Przydatne funkcje

C++ udostępnia nam wiele funkcji matematycznych, dzięki którym możemy przeprowadzać proste i nieco bardziej złożone obliczenia. Prawie wszystkie są zawarte w pliku nagłówkowym *cmath*, dlatego też musimy dołączyć ten plik do każdego programu, w którym chcemy korzystać z tych funkcji. Robimy to analogicznie jak w przypadku innych nagłówków – umieszczając na początku naszego kodu dyrektywę:

```
#include <cmath>
```

Po dopełnieniu tej drobnej formalności możemy korzystać z całego bogactwa narzędzi matematycznych, jakie zapewnia nam C++. Spójrzmy więc, jak się one przedstawiają.

Funkcje potęgowe

W przeciwieństwie do niektórych języków programowania, C++ nie posiada oddzielnego operatora potęgowania[40]. Zamiast niego mamy natomiast funkcję `pow()` (ang. *power* – potęga), która prezentuje się następująco:

```
double pow(double base, double exponent);
```

Jak widać, bierze ona dwa parametry. Pierwszym (`base`) jest podstawa potęgi, a drugim (`exponent`) jej wykładnik. W wyniku zwracany jest oczywiście wynik potęgowania (a więc wartość wyrażenia $base^{exponent}$).

Podobną do powyższej deklarację funkcji, przedstawiającą jej nazwę, ilość i typy parametrów oraz typ zwracanej wartości, nazywamy **prototypem**.

Oto kilka przykładów wykorzystania funkcji `pow()`:

```
double fX;
fX = pow(2, 8);      // ósma potęga dwójki, czyli 256
fX = pow(3, 4);      // czwarta potęga trójki, czyli 81
fX = pow(5, -1);     // odwrotność piątki, czyli 0.2
```

Inną równie często wykonywaną czynnością jest pierwiastkowanie. Realizuje ją między innymi funkcja `sqrt()` (ang. *square root* – pierwiastek kwadratowy):

```
double sqrt(double x);
```

Jej jedyny parametr to oczywiście liczba, która chcemy pierwiastkować. Użycie tej funkcji jest zatem niezwykle intuicyjne:

```
fX = sqrt(64);             // 8 (bo 8*8 == 64)
fX = sqrt(2);              // około 1.414213562373
```

[40] Znak `^`, który służy w nich do wykonywania tego działania, jest w C++ zarezerwowany dla jednej z operacji bitowych – różnicy symetrycznej. Więcej informacji na ten temat możesz znaleźć w Dodatku B, *Reprezentacja danych w pamięci*.

```
fX = sqrt(pow(fY, 2));      // fY
```

Nie ma natomiast wbudowanej formuły, która obliczałaby pierwiastek **dowolnego** stopnia z danej liczby. Możemy jednak łatwo napisać ją sami, korzystając z prostej własności:

$$\sqrt[a]{x} = x^{\frac{1}{a}}$$

Po przełożeniu tego równania na C++ uzyskujemy następującą funkcję:

```
double root(double x, double a)        { return pow(x, 1 / a); }
```

Zapisanie jej definicji w jednej linijce jest całkowicie dopuszczalne i, jak widać, bardzo wygodne. Elastyczność składni C++ pozwala więc na zupełnie dowolną organizację kodu.

Dokładny opis poznanych funkcji `pow()` i `sqrt()` znajdziesz w MSDN.

Funkcje wykładnicze i logarytmiczne

Najczęściej stosowaną w matematyce funkcją wykładniczą jest e^x, niekiedy oznaczana także jako $\exp(x)$. Taką też formę ma ona w C++:

```
double exp(double x);
```

Zwraca ona wartość stałej *e*[41] podniesionej do potęgi `x`. Popatrzmy na kilka przykładów:

```
fX = exp(0);               // 1
fX = exp(1);               // e
fX = exp(2.302585093);     // 10.000000
```

Natomiast funkcję wykładniczą o dowolnej podstawie uzyskujemy, stosując omówioną już wcześniej formułę `pow()`.

Przeciwstawne do funkcji wykładniczych są logarytmy. Tutaj mamy aż **dwie** odpowiednie funkcje :) Pierwsza z nich to `log()`:

```
double log(double x);
```

Jest to logarytm naturalny (o podstawie *e*), a więc funkcja dokładnie do odwrotna do poprzedniej `exp()`. Otóż dla danej liczby `x` zwraca nam wartość wykładnika, do którego musielibyśmy podnieść *e*, by otrzymać `x`. Dla pełnej jasności zerknijmy na poniższe przykłady:

```
fX = log(1);          // 0
fX = log(10);         // 2.302585093
fX = log(exp(x));     // x
```

Drugą funkcją jest `log10()`, czyli logarytm dziesiętny (o podstawie 10):

```
double log10(double x);
```

[41] Tak zwanej stałej Nepera, podstawy logarytmów naturalnych - równej w przybliżeniu `2.71828182845904`.

Analogicznie, funkcja ta zwraca wykładnik, do którego należałoby podnieść dziesiątkę, aby otrzymać podaną liczbę `x`, na przykład:

```
fX = log10(1000);           // 3 (bo 10^3 == 1000)
fX = log10(1);              // 0
fX = log10(pow(10, x));     // x
```

Niestety, znowu (podobnie jak w przypadku pierwiastków) nie mamy bardziej uniwersalnego odpowiednika tych dwóch funkcji, czyli logarytmu o **dowolnej** podstawie. Ponownie jednak możemy skorzystać z odpowiedniej tożsamości matematycznej[42]:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Nasza własna funkcja może więc wyglądać tak:

```
double log_a(double a, double x)       { return log(x) / log(a); }
```

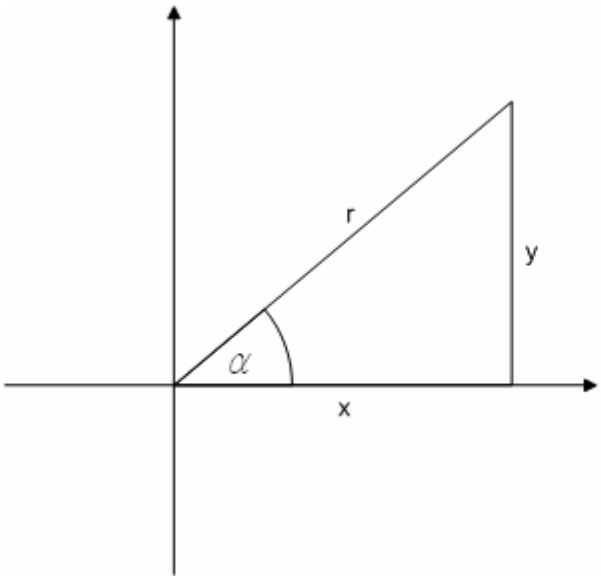
Oczywiście użycie `log10()` w miejsce `log()` jest również poprawne.

Zainteresowanych ponownie odsyłam do MSDN celem poznania dokładnego opisu funkcji `exp()` oraz `log()` i `log10()`.

Funkcje trygonometryczne

Dla nas, (przyszłych) programistów gier, funkcje trygonometryczne są szczególnie przydatne, gdyż będziemy korzystać z nich niezwykle często – choćby przy różnorakich obrotach. Wypadałoby zatem dobrze znać ich odpowiedniki w języku C++.

Na początek przypomnijmy sobie (znane, mam nadzieję :D) określenia funkcji trygonometrycznych. Posłuży nam do tego poniższy rysunek:



$$\sin\alpha = \frac{y}{r}$$

$$\cos\alpha = \frac{x}{r}$$

$$\tan\alpha = \frac{y}{x}$$

$$\cot\alpha = \frac{x}{y} = \frac{1}{\tan\alpha}$$

$$\sec\alpha = \frac{r}{x} = \frac{1}{\cos\alpha}$$

$$\csc\alpha = \frac{r}{y} = \frac{1}{\sin\alpha}$$

Rysunek 1. Definicje funkcji trygonometrycznych dowolnego kąta

[42] Znanej jako zmiana podstawy logarytmu.

Zwróćmy uwagę, że trzy ostatnie funkcje są określone jako odwrotności trzech pierwszych. Wynika stąd fakt, iż potrzebujemy do szczęścia jedynie sinusa, cosinusa i tangensa – resztę funkcji i tak będziemy mogli łatwo uzyskać.
C++ posiada oczywiście odpowiednie funkcje:

```
double sin(double alfa);   // sinus
double cos(double alfa);   // cosinus
double tan(double alfa);   // tangens
```

Działają one identycznie do swoich geometrycznych odpowiedników. Jako jedyny parametr przyjmują miarę kąta **w radianach** i zwracają wyniki, których bez wątpienia można się spodziewać :)
Jeżeli chodzi o trzy brakujące funkcje, to ich definicje są, jak sądzę, oczywiste:

```
double cot(double alfa)    { return 1 / tan(alfa); }     // cotangens
double sec(double alfa)    { return 1 / cos(alfa); }     // secant
double csc(double alfa)    { return 1 / sin(alfa); }     // cosecant
```

Gdy pracujemy z kątami i funkcjami trygonometrycznymi, nierzadko pojawia się konieczność zamiany miary kąta ze stopni na radiany lub odwrotnie. Niestety, nie znajdziemy w C++ odpowiednich funkcji, które realizowałyby to zadanie. Być może dlatego, że sami możemy je łatwo napisać:

```
const double PI = 3.1415923865;
double degtorad(double alfa)     { return alfa * PI / 180; }
double radtodeg(double alfa)     { return alfa * 180 / PI; }
```

Pamiętajmy też, aby nie mylić tych dwóch miar kątów i zdawać sobie sprawę, iż funkcje trygonometryczne w C++ używają radianów. Pomyłki w tej kwestii są dość częste i powodują nieprzyjemne rezultaty, dlatego należy się ich wystrzegać :)

Jak zwykle, więcej informacji o funkcjach `sin()`, `cos()` i `tan()` znajdziesz w MSDN. Możesz tam również zapoznać się z funkcjami odwrotnymi do trygonometrycznych – `asin()`, `acos()` oraz `atan()` i `atan2()`.

Liczby pseudolosowe

Zostawmy już te zdecydowanie zbyt matematyczne dywagacje i zajmijmy się czymś, co bardziej zainteresuje przeciętnego zjadacza komputerowego i programistycznego chleba :) Mam tu na myśli generowanie wartości losowych.
Liczby losowe znajdują zastosowanie w bardzo wielu programach. W przypadku gier mogą służyć na przykład do tworzenia realistycznych efektów ognia, deszczu czy śniegu. Używając ich możemy również kreować za każdym inną mapę w grze strategicznej czy zapewnić pojawianie się wrogów w przypadkowych miejscach w grach zręcznościowych. Przydatność liczb losowych jest więc bardzo szeroka.

Uzyskanie losowej wartości jest w C++ całkiem proste. W tym celu korzystamy z funkcji `rand()` (ang. *random* – losowy):

```
int rand();
```

Jak możnaby przypuszczać, zwraca nam ona przypadkową liczbę dodatnią[43]. Najczęściej jednak potrzebujemy wartości z określonego przedziału – na przykład w programie

[43] Liczba ta należy do przedziału `<0; RAND_MAX>`, gdzie `RAND_MAX` jest stałą zdefiniowaną przez kompilator (w Visual C++ .NET ma ona wartość `32767`).

ilustrującym działanie pętli `while` losowaliśmy liczbę z zakresu od `1` do `100`. Osiągnęliśmy to w dość prosty sposób:

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

Wykorzystanie operatora reszty z dzielenia sprawia, że nasza dowolna wartość (zwrócona przez `rand()`) zostaje odpowiednio „przycięta" – w tym przypadku do przedziału `<0; 99>` (ponieważ resztą z dzielenia przez sto może być `0`, `1`, `2`, …, `98`, `99`). Dodanie jedynki zmienia ten zakres do pożądanego `<1; 100>`.
W podobny sposób możemy uzyskać losową liczbę z jakiegokolwiek przedziału. Nie od rzeczy będzie nawet napisanie odpowiedniej funkcji:

```
int random(int nMin, int nMax)
   { return rand() % (nMax - nMin + 1) + nMin; }
```

Używając jej, potrafimy bez trudu stworzyć chociażby symulator rzutu kostką do gry:

```
void main()
{
   std::cout << "Wylosowano " << random(1, 6) << " oczek.";
   getch();
}
```

Zdaje się jednak, że coś jest nie całkiem w porządku… Uruchamiając parokrotnie powyższy program, za każdym razem zobaczymy jedną i **tą samą** liczbę! Gdzie jest więc ta obiecywana losowość?!

Cóż, nie ma w tym nic dziwnego. Komputer to tylko wielkie liczydło, które działa w zaprogramowany i **przewidywalny** sposób. Dotyczy to także funkcji `rand()`, której działanie opiera się na raz ustalonym i niezmiennym algorytmie. Jej wynik nie jest zatem w żaden sposób losowany, lecz **wyliczany** na podstawie formuł matematycznych. Dlatego też liczby uzyskane w ten sposób nazywamy **pseudolosowymi**, ponieważ tylko udają prawdziwą przypadkowość.
Wydawać by się mogło, że fakt ten czyni je całkowicie nieprzydatnymi. Na szczęście nie jest to prawdą: liczby pseudolosowe można z powodzeniem wykorzystywać we właściwym im celu – pod warunkiem, że robimy to poprawnie.

Musimy bowiem pamiętać, aby przed pierwszym użyciem `rand()` wywołać inną funkcję – `srand()`:

```
void srand(unsigned int seed);
```

Jej parametr `seed` to tak zwane ziarno. Jest to liczba, która inicjuje generator wartości pseudolosowych. Dla każdego możliwego ziarna funkcja `rand()` oblicza nam **inny** ciąg liczb. Zatem, logicznie wnioskując, powinniśmy dbać o to, by przy każdym uruchomieniu programu wartość ziarna była inna.
Dochodzimy tym samym do pozornie błędnego koła – żeby uzyskać liczbę losową, potrzebujemy… liczby losowej! Jak rozwiązać ten, zdawałoby się, nierozwiązywalny problem?…

Otóż należy znaleźć taką wartość, która będzie się zmieniać miedzy kolejnymi uruchomieniami programu. Nietrudno ją wskazać – to po prostu **czas systemowy**.
Jego pobranie jest bardzo łatwe, bowiem C++ udostępnia nam zgrabną funkcję `time()`, zwracająca aktualny czas[44] w sekundach:

[44] Funkcja ta zwraca liczbę sekund, jakie upłynęły od północy 1 stycznia 1970 roku.

```
time_t time(time_t* timer);
```

Być może wygląda ona dziwnie, ale zapewniam cię, że działa świetnie :) Wymaga jednak, abyśmy dołączyli do programu dodatkowy nagłówek *ctime*:

```
#include <ctime>
```

Teraz mamy już wszystko, co potrzebne. Zatem do dzieła! Nasza prosta aplikacja powinna obecnie wyglądać tak:

```
// Random - losowanie liczby

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

int random(int nMin, int nMax) { return rand() % nMax + nMin; }

void main()
{
   // zainicjowanie generatora liczb pseudolosowych aktualnym czasem
   srand (static_cast<unsigned int>(time(NULL)));

   // wylosowanie i pokazanie liczby
   std::cout << "Wylosowana liczba to " << random(1, 6) << std::endl;

   getch();
}
```

Kompilacja i kilkukrotne uruchomienie powyższego kodu utwierdzi nas w przekonaniu, iż tym razem wszystko funkcjonuje poprawnie.

Screen 27. Przykładowy rezultat „rzutu kostką”

Dzieje się tak naturalnie za sprawą tej linijki:

```
srand (static_cast<unsigned int>(time(NULL)));
```

Wywołuje ona funkcję `srand()`, podając jej ziarno uzyskane poprzez `time()`. Ze względu na to, iż `time()` zwraca wartość należącą do specjalnego typu `time_t`, potrzebne jest rzutowanie jej na typ `unsigned int`.
Wyjaśnienia wymaga jeszcze parametr funkcji `time()`. `NULL` to tak zwany wskaźnik zerowy, niereprezentujący żadnej przydatnej wartości. Używamy go tutaj, gdyż nie mamy nic konkretnego do przekazania dla funkcji, zaś ona sama niczego takiego od nas nie wymaga :)

Kompletny opis funkcji `rand()`, `srand()` i `time()` znajdziesz, jak poprzednio, w MSDN.

Zaokrąglanie liczb rzeczywistych

Gdy poznawaliśmy rzutowanie typów, podałem jako przykład konwersję wartości `float` na `int`. Wspomniałem też, że zastosowane w tym przypadku zaokrąglenie liczby rzeczywistej polega na zwyczajnym odrzuceniu jej części ułamkowej.

Nie jest to wszakże jedyny sposób dokonywania podobnej zamiany, gdyż C++ posiada też dwie specjalnie do tego przeznaczone funkcje. Działają one w inaczej niż zwykłe rzutowanie, co samo w sobie stanowi dobry pretekst do ich poznania :D

Owe dwie funkcje są sobie wzajemnie przeciwstawne – jedna zaokrągla liczbę w górę (wynik jest zawsze większy lub równy podanej wartości), zaś druga w dół (rezultat jest mniejszy lub równy). Świetne obrazują to ich nazwy, odpowiednio: `ceil()` (ang. *ceiling* – sufit) oraz `floor()` ('podłoga').
Przyjrzyjmy się teraz nagłówkom tych funkcji:

```
double ceil(double x);
double floor(double x);
```

Nie ma tu żadnych niespodzianek – no, może poza typem zwracanego wyniku. Dlaczego nie jest to `int`? Otóż typ `double` ma po prostu większą rozpiętość przedziału wartości, jakie może przechowywać. Ponieważ argument funkcji także należy do tego typu, zastosowanie `int` spowodowałoby otrzymywanie błędnych rezultatów dla bardzo dużych liczb (takich, jakie „nie zmieściłyby się" do `int`'a).

Na koniec mamy jeszcze kilka przykładów, ilustrujących działanie poznanych przed chwilą funkcji:

```
fX = ceil(6.2);      // 7.0
fX = ceil(-5.6);     // -5.0
fX = ceil(14);       // 14.0
fX = floor(1.7);     // 1.0
fX = floor(-2.1);    // -3.0
```

Szczególnie dociekliwych czeka kolejna wycieczka wgłąb MSDN po dokładny opis funkcji `ceil()` i `floor()` ;D

Inne funkcje

Ostatnie dwie formuły trudno przyporządkować do jakiejś konkretnej grupy. Nie znaczy to jednak, że są one mniej ważne niż pozostałe.

Pierwszą z nich jest `abs()` (ang. *absolute value*), obliczająca wartość bezwzględną (moduł) danej liczby. Jak pamiętamy z matematyki, wartość ta jest tą samą liczbą, lecz bez znaku – zawsze dodatnią.
Ciekawa jest deklaracja funkcji `abs()`. Istnieje bowiem kilka jej wariantów, po jednym dla każdego typu liczbowego:

```
int abs(int n);
float abs(float n);
double abs(double n);
```

Jest to jak najbardziej możliwe i w pełni poprawne. Zabieg taki nazywamy **przeciążaniem** (ang. *overloading*) funkcji.

Przeciążanie funkcji (ang. *function overloading*) to obecność kilku deklaracji funkcji o tej samej nazwie, lecz posiadających różne listy parametrów i/lub typy zwracanej wartości.

Gdy więc wywołujemy funkcję `abs()`, kompilator stara się wydedukować, który z jej wariantów powinien zostać uruchomiony. Czyni to przede wszystkim na podstawie przekazanego doń parametru. Jeżeli byłaby to liczba całkowita, zostałaby wywołana

wersja przyjmująca i zwracająca typ `int`. Jeżeli natomiast podalibyśmy liczbę zmiennoprzecinkową, wtedy do akcji wkroczyłby inny wariant funkcji.
Zatem dzięki mechanizmowi przeciążania funkcja `abs()` może operować na różnych typach liczb:

```
int nX = abs(-45);          // 45
float fX = abs(7.5);        // 7.5
double fX = abs(-27.8);     // 27.8
```

Druga funkcja to `fmod()`. Działa ona podobnie do operatora `%`, gdyż także oblicza resztę z dzielenia dwóch liczb. Jednak w przeciwieństwie do niego nie ogranicza się jedynie do liczb całkowitych, bowiem potrafi operować także na wartościach rzeczywistych. Widać to po jej nagłówku:

```
double fmod(double x, double y);
```

Funkcja ta wykonuje dzielenie `x` przez `y` i zwraca pozostałą zeń resztę, co oczywiście łatwo wydedukować z jej nagłówka :) Dla porządku zerknijmy jeszcze na parę przykładów:

```
fX = fmod(14, 3);           // 2
fX = fmod(2.75, 0.5);       // 0.25
fX = fmod(-10, 3);          // -1
```

Wielbiciele MSDN mogą zacierać ręce, gdyż z pewnością znajdą w niej szczegółowe opisy funkcji `abs()`[45] i `fmod()` ;)

***

Zakończyliśmy w ten sposób przegląd asortymentu funkcji liczbowych, oferowanego przez C++. Przyswoiwszy sobie wiadomości o tych formułach będziesz mógł robić z liczbami niemal wszystko, co tylko sobie zamarzysz :)

Znane i nieznane operatory

Dobrze wiemy, że funkcje to nie jedyne środki służące do manipulacji wartościami liczbowymi. Od początku używaliśmy do tego przede wszystkim operatorów, które odpowiadały doskonale nam znanym podstawowym działaniom matematycznym. Nadarza się dobra okazja, aby przypomnieć sobie o tych elementach języka C++, przy okazji poszerzając swoje informacje o nich.

Dwa rodzaje

Operatory w C++ możemy podzielić na dwie grupy ze względu na liczbę „parametrów”, na których działają. Wyróżniamy więc operatory **unarne** – wymagające jednego „parametru” oraz **binarne** – potrzebujące dwóch.

Do pierwszej grupy należą na przykład symbole `+` oraz `-`, gdy stawiamy je przed jakimś wyrażeniem. Wtedy bowiem nie pełnią roli operatorów dodawania i odejmowania, lecz **zachowania** lub **zmiany znaku**. Może brzmi to dość skomplikowanie, ale naprawdę jest bardzo proste:

```
int nX = 5;
```

[45] Standardowo dołączona do Visual Studio .NET biblioteka MSDN posiada lekko nieaktualny opis tej funkcji – nie są tam wymienione jej wersje przeciążane dla typów `float` i `double`.

```
int nY = +nX;   // nY == 5
nY = -nX;       // nY == -5
```

Operator `+` zachowuje nam znak wyrażenia (czyli praktycznie nie robi nic, dlatego zwykle się go nie stosuje), zaś `-` zmienia go na przeciwny (**neguje** wyrażenie). Operatory te mają identyczną funkcję w matematyce, dlatego, jak sądzę, nie powinny sprawić ci większego kłopotu :)
Do grupy operatorów unarnych zaliczamy również `++` oraz `--`, odpowiadające za inkrementację i dekrementację. Za chwilę przyjrzymy im się bliżej.

Drugi zestaw to operatory binarne; dla nich konieczne są dwa argumenty. Do tej grupy należą wszystkie poznane wcześniej operatory arytmetyczne, a więc `+` (dodawanie), `-` (odejmowanie), `*` (mnożenie), `/` (dzielenie) oraz `%` (reszta z dzielenia).
Ponieważ swego czasu poświęciliśmy im sporo uwagi, nie będziemy teraz dogłębnie wnikać w działanie każdego z nich. Więcej miejsca przeznaczymy tylko na operator dzielenia.

Sekrety inkrementacji i dekrementacji

Operatorów `++` i `--` używamy, aby dodać do zmiennej lub odjąć od niej jedynkę. Taki zapis jest najkrótszy i najwygodniejszy, a poza tym najszybszy. Używamy go szczególnie często w pętlach `for`.

Jednak może być on także częścią złożonych wyrażeń. Poniższe fragmenty kodu są absolutnie poprawne i w dodatku nierzadko spotykane:

```
int nA = 6;
int nB = ++nA;

int nC = 4;
int nD = nC++;
```

Od tej pory będę mówił jedynie o operatorze inkrementacji, jednak wszystkie przedstawione tu własności dotyczą także jego dekrementującego brata.

Nasuwa się naturalne pytanie: jakie wartości będą miały zmienne `nA`, `nB`, `nC` i `nD` po wykonaniu tych czterech linijek kodu?

Jeżeli chodzi o `nA` i `nC`, to sprawa jest oczywista. Każda z tych zmiennych została jednokrotnie poddana inkrementacji, zatem ich wartości są o jeden większe niż na początku. Wynoszą odpowiednio `7` i `5`.
Pozostałe zmienne są już twardszym orzechem do zgryzienia. Skupmy się więc chwilowo na `nB`. Jej wartość na pewno ma coś wspólnego z wartością `nA` - może to być albo `6` (liczba przed inkrementacją), albo `7` (już po inkrementacji). Analogicznie, `nD` może być równa `4` (czyli wartości `nC` przed inkrementacją) lub `5` (po inkrementacji).

Jak jest w istocie? Sam się przekonaj! Stwórz nowy program, wpisz do jego funkcji `main()` powyższe wiersze kodu i dodaj instrukcje pokazujące wartości zmiennych…

Cóż widzimy? Zmienna `nB` jest równa `7`, a więc została jej przypisana wartość `nA` już po inkrementacji. Natomiast `nD` równa się `4` - tyle, co `nC` przed inkrementacją.
Przyczyną tego faktu jest rzecz jasna rozmieszczenie plusów. Gdy napisaliśmy je przed inkrementowaną zmienną, dostaliśmy w wyniku wartość zwiększoną o 1. Kiedy zaś umieściliśmy je za tą zmienną, otrzymaliśmy jeszcze stary rezultat.
Jak zatem mogliśmy się przekonać, odpowiednie zapisanie operatorów `++` i `--` ma całkiem spore znaczenie.

Umieszczenie operatora `++` (`--`) **przed** wyrażeniem nazywamy **preinkrementacją** (**predekrementacją**). W takiej sytuacji **najpierw** dokonywane jest zwiększenie (zmniejszenie) jego wartości o 1. Nowa wartość jest potem zwracana jako wynik.

Kiedy napiszemy operator `++` (`--`) **po** wyrażeniu, mamy do czynienia z **postinkrementacją** (**postdekrementacją**). W tym przypadku najpierw następuje zwrócenie wartości, która dopiero **potem** jest zwiększana (zmniejszana) o jeden[46].

Czyżby trzeba było tych regułek uczyć się na pamięć? Oczywiście, że nie :) Jak większość rzeczy w programowaniu, możemy je traktować intuicyjnie.
Kiedy napiszemy plusy (lub minusy) **przed** zmienną, wtedy **najpierw** „zadziałają" właśnie one. A skutkiem ich działania będzie inkrementacja lub dekrementacja wartości zmiennej, a więc otrzymamy w rezultacie już zmodyfikowaną liczbę.
Gdy zaś umieścimy je **za** nazwą zmiennej, ustąpią jej pierwszeństwa i pozwolą, aby jej stara wartość została zwrócona. Dopiero **potem** wykonają swoją pracę, czyli in/dekrementację.

Jeżeli mamy możliwość dokonania wyboru między dwoma położeniami operatora `++` (lub `--`), powinniśmy zawsze używać wariantu prefiksowego (przed zmienną). Wersja postfiksowa musi bowiem utworzyć w pamięci kopię zmiennej, żeby móc zwrócić jej starą wartość po in/dekrementacji. Cierpi na tym zarówno szybkość programu, jak i jego wymagania pamięciowe (chociaż w przypadku typów liczbowych jest to niezauważalna różnica).

Słówko o dzieleniu

W programowaniu mamy do czynienia z dwoma rodzajami dzielenia liczb: całkowitoliczbowym oraz zmiennoprzecinkowym. Oba zwracają te same rezultaty w przypadku podzielnych przez siebie liczb całkowitych, ale w innych sytuacjach zachowują się odmiennie.
Dzielenie całkowitoliczbowe podaje jedynie całkowitą część wyniku, odrzucając cyfry po przecinku. Z tego powodu wynik takiego dzielenia może być bezpośrednio przypisany do zmiennej typu całkowitego. Wtedy jednak traci się dokładność ilorazu.
Dzielenie zmiennoprzecinkowe pozwala uzyskać precyzyjny rezultat, gdyż zwraca liczbę rzeczywistą wraz z jej częścią ułamkową. Ów wynik musi być wtedy zachowany w zmiennej typu rzeczywistego.

Większa część języków programowania rozróżnia te dwa typy dzielenia poprzez wprowadzenie dwóch odrębnych operatorów dla każdego z nich[47]. C++ jest tu swego rodzaju wyjątkiem, ponieważ posiada tylko **jeden** operator dzielący, `/`. Jednakże posługując się nim odpowiednio, możemy uzyskać oba rodzaje ilorazów.

Zasady, na podstawie których wyróżniane są w C++ te dwa typy dzielenia, są ci już dobrze znane. Przedstawiliśmy je sobie podczas pierwszego spotkania z operatorami arytmetycznymi. Ponieważ jednak powtórzeń nigdy dość, wymienimy je sobie ponownie :)

Jeżeli **obydwa** argumenty operatora / (dzielna i dzielnik) są liczbami całkowitymi, wtedy wykonywane jest dzielenie **całkowitoliczbowe**.

---

[46] To uproszczone wyjaśnienie, bo przecież zwrócenie wartości kończyłoby działanie operatora. Naprawdę więc wartość wyrażenia jest tymczasowo zapisywana i zwracana po dokonaniu in/dekrementacji.
[47] W Visual Basicu jest to `\` dla dzielenia całkowitoliczbowego i `/` dla zmiennoprzecinkowego. W Delphi odpowiednio **`div`** i `/`.

W przypadku, gdy **chociaż jedna** z liczb biorących udział w dzieleniu jest typu rzeczywistego, mamy do czynienia z dzieleniem **zmiennoprzecinkowym**.

Od chwili, w której poznaliśmy rzutowanie, mamy większą kontrolę nad dzieleniem. Możemy bowiem łatwo **zmienić typ** jednej z liczb i w ten sposób spowodować, by został wykonany inny rodzaj dzielenia. Możliwe staje się na przykład uzyskanie dokładnego ilorazu dwóch wartości całkowitych:

```
int nX = 12;
int nY = 5;
float fIloraz = nX / static_cast<float>(nY);
```

Tutaj uzyskamy precyzyjny rezultat `2.4`, gdyż kompilator przeprowadzi dzielenie zmiennoprzecinkowe. Zrobi tak, bo drugi argument operatora `/`, mimo że ma wartość całkowitą, jest traktowany jako wyrażenie typu `float`. Dzieje się tak naturalnie dzięki rzutowaniu.
Gdybyśmy go nie zastosowali i wpisali po prostu `nX / nY`, wykonałoby się dzielenie całkowitoliczbowe i ułamkowa część wyniku zostałaby obcięta. Ten okrojony rezultat zmieniłby następnie typ na `float` (ponieważ przypisalibyśmy go do zmiennej rzeczywistej), co byłoby zupełnie zbędne, gdyż i tak w wyniku dzielenia dokładność została stracona.

Prosty wniosek brzmi: uważajmy, jak i co tak naprawdę dzielimy, a w razie wątpliwości korzystajmy z rzutowania.

***

Kończący się właśnie podrozdział prezentował podstawowe instrumentarium operacyjne wartości liczbowych w C++. Poznając je zyskałeś potencjał do tworzenia aplikacji wykorzystujących złożone obliczenia, do których niewątpliwie należą także gry.

Jeżeli czujesz się przytłoczony nadmiarem matematyki, to mam dla ciebie dobrą wiadomość: nasza uwaga skupi się teraz na zupełnie innym, lecz również ważnym typie danych - tekście.

Łańcuchy znaków

Ciągi znaków (ang. *strings*) stanowią drugi, po liczbach, ważny rodzaj informacji przetwarzanych przez programy. Chociaż zajmują więcej miejsca w pamięci niż dane binarne, a operacje na nich trwają dłużej, mają wiele znaczących zalet. Jedną z nich jest fakt, iż są bardziej zrozumiałe dla człowieka niż zwykłe sekwencje bitów. W czasie, gdy moce komputerów rosną bardzo szybko, wymienione wcześniej wady nie są natomiast aż tak dotkliwe. Wszystko to powoduje, że dane tekstowe są coraz powszechniej spotykane we współczesnych aplikacjach.

Duża jest w tym także rola Internetu. Takie standardy jak HTML czy XML są przecież formatami tekstowymi.

Dla programistów napisy były od zawsze przyczyną częstych bólów głowy. W przeciwieństwie bowiem do typów liczbowych, mają one **zmienny rozmiar**, który nie może być ustalony raz podczas uruchamiania programu. Ilość pamięci operacyjnej, którą zajmuje każdy napis musi być dostosowywana do jego długości (liczby znaków) i zmieniać się podczas działania aplikacji. Wymaga to dodatkowego czasu (od programisty

i od komputera), uwagi oraz dokładnego przemyślenia (przez programistę, nie komputer ;D) mechanizmów zarządzania pamięcią.
Zwykli użytkownicy pecetów - szczególnie ci, którzy pamiętają jeszcze zamierzchłe czasy DOSa - także nie mają dobrych wspomnień związanych z danymi tekstowymi. Odwieczne kłopoty z polskimi „ogonkami" nadal dają o sobie znać, choć na szczęście coraz rzadziej musimy oglądać na ekranie dziwne „krzaczki" zamiast znajomych liter w rodzaju ą, ć, ń czy ź.

Wydaje się więc, że przed koderem piszącym programy przetwarzające tekst piętrzą się niebotyczne wręcz trudności. Problemy są jednak po to, aby je rozwiązywać (lub by inni rozwiązywali je za nas ;)), więc oba wymienione dylematy doczekały się już wielu bardzo dobrych pomysłów.
Rozszerzające się wykorzystanie standardu Unicode ograniczyło już znacznie kłopoty związane ze znakami specyficznymi dla niektórych języków. Kwestią czasu zdaje się chwila, gdy znikną one zupełnie.
Powstało też mnóstwo sposobów na efektywne składowanie napisów o zmiennej długości w pamięci komputera. Wprawdzie w tym przypadku nie ma jednego, wiodącego trendu zapewniającego przenośność między wszystkimi platformami sprzętowymi lub chociaż aplikacjami, jednak i tak sytuacja jest znacznie lepsza niż jeszcze kilka lat temu[48].
Koderzy mogą więc sobie pozwolić na uzasadniony optymizm :)

Wsparci tymi pokrzepiającymi faktami możemy teraz przystąpić do poznawania elementów języka C++, które służą do pracy z łańcuchami znaków.

Napisy według C++

Trudno w to uwierzyć, ale poprzednik C++ - język C - w ogóle nie posiadał odrębnego typu zmiennych, mogącego przechowywać napisy. Aby móc operować danymi tekstowymi, trzeba było używać mało poręcznych tablic znaków (typu `char`) i samemu dbać o zagadnienia związane z przydzielaniem i zwalnianiem pamięci.
Nam, programistom C++, nic takiego na szczęście nie grozi :) Nasz ulubiony język jest bowiem wyposażony w kilka bardzo przydatnych i łatwych w obsłudze mechanizmów, które udostępniają możliwość manipulacji tekstem.

Rozwiązania, o których będzie mowa poniżej, są częścią Biblioteki Standardowej języka C++. Jako że jest ona dostępna w każdym kompilatorze tego języka, sposoby te są najbardziej uniwersalne i przenośne, a jednocześnie wydajne. Korzystanie z nich jest także bardzo wygodne i łatwe.
Oprócz nich istnieją również inne metody obsługi łańcuchów znaków. Na przykład biblioteki MFC i VCL (wspomagające programowanie w Windows) posiadają własne narzędzia, służące temu właśnie celowi[49]. Nawet jeżeli skorzystasz kiedyś z tych bibliotek, będziesz mógł wciąż używać opisanych tutaj mechanizmów standardowych.

Aby móc z nich skorzystać, należy przede wszystkim włączyć do swojego kodu plik nagłówkowy *string*:

```
#include <string>
```

Po tym zabiegu zyskujemy dostęp do całego arsenału środków programistycznych, służących operacjom tekstowym.

[48] Dużą zasługę ma w tym ustandaryzowanie języka C++, w którym powstaje ponad połowa współczesnych aplikacji. W przyszłości znaczącą rolę mogą odegrać także rozwiązania zawarte w platformie .NET.
[49] MFC (Microsoft Foundation Classes) zawiera przeznaczoną do tego klasę `CString`, zaś VCL (Visual Component Library) posiada typ **`String`**, który jest częścią kompilatora C++ firmy Borland.

Typy zmiennych tekstowych

Istnieją dwa typy zmiennych tekstowych, które różnią się rozmiarem pojedynczego znaku. Ujmuje je poniższa tabelka:

nazwa	typ znaku	rozmiar znaku	zastosowanie
`std::string`	`char`	1 bajt	tylko znaki ANSI
`std::wstring`	`wchar_t`	2 bajty	znaki ANSI i Unicode

Tabela 6. Typy łańcuchów znaków

`std::string` jest ci już dobrze znany, gdyż używaliśmy go niejednokrotnie. Przechowuje on dowolną (w granicach dostępnej pamięci) ilość znaków, z których każdy jest typu `char`. Zajmuje więc dokładnie 1 bajt i może reprezentować jeden z 256 symboli zawartych w tablicy ANSI.

Wystarcza to do przechowywania tekstów w językach europejskich (choć wymaga specjalnych zabiegów, tzw. stron kodowych), jednak staje się niedostateczne w przypadku dialektów o większej liczbie znaków (na przykład wschodnioazjatyckich). Dlatego wykoncypowano, aby dla pojedynczego symbolu przeznaczać większą ilość bajtów i w ten sposób stworzono MBCS (Multi-Byte Character Sets - wielobajtowe zestawy znaków) w rodzaju Unicode.
Nie mamy tu absolutnie czasu ani miejsca na opisywanie tego standardu. Warto jednak wiedzieć, że C++ posiada typ łańcuchowy, który umożliwia współpracę z nim - jest to `std::wstring` (ang. *wide string* - „szeroki" napis). Każdy jego znak jest typu `wchar_t` (ang. *wide char* - „szeroki" znak) i zajmuje 2 bajty. Łatwo policzyć, że umożliwia tym samym przechowywanie jednego z aż 65536 (256^2) możliwych symboli, co stanowi znaczny postęp w stosunku do ANSI :)
Korzystanie z `std::wstring` niewiele różni się przy tym od używania jego bardziej oszczędnego pamięciowo kuzyna. Musimy tylko pamiętać, żeby poprzedzać literką `L` wszystkie wpisane do kodu stałe tekstowe, które mają być trzymane w zmiennych typu `std::wstring`. W ten sposób bowiem mówimy kompilatorowi, że chcemy zapisać dany napis w formacie Unicode. Wygląda to choćby tak:

```
std::wstring strNapis = L"To jest tekst napisany znakami dwubajtowymi";
```

Dobra wiadomość jest taka, że jeśli zapomniałbyś o wspomnianej literce `L`, to powyższy kod w ogóle by się nie skompilował ;D

Jeżeli chciałbyś wyświetlać takie „szerokie" napisy w konsoli i umożliwić użytkownikowi ich wprowadzanie, musisz użyć specjalnych wersji strumieni wejścia i wyjścia. Są to odpowiednio `std::wcin` i `std::wcout`. Używa się ich w identyczny sposób, jak poznanych wcześniej „zwykłych" strumieni `std::cin` i `std::cout`.

Manipulowanie łańcuchami znaków

OK, gdy już znamy dwa typy zmiennych tekstowych, jakie oferuje C++, czas zobaczyć możliwe działania, które możemy na nich przeprowadzać.

Inicjalizacja

Najprostsza deklaracja zmiennej tekstowej wygląda, jak wiemy, mniej więcej tak:

```
std::string strNapis;
```

Wprowadzona w ten sposób nowa zmienna jest z początku całkiem pusta - nie zawiera żadnych znaków. Jeżeli chcemy zmienić ten stan rzeczy, możemy ją **zainicjalizować** odpowiednim tekstem - tak:

```
std::string strNapis = "To jest jakis tekst";
```

albo tak:

```
std::string strNapis("To jest jakis tekst");
```

Ten drugi zapis bardzo przypomina wywołanie funkcji. Istotnie, ma on z nimi wiele wspólnego - na tyle dużo, że możliwe jest nawet zastosowanie drugiego parametru, na przykład:

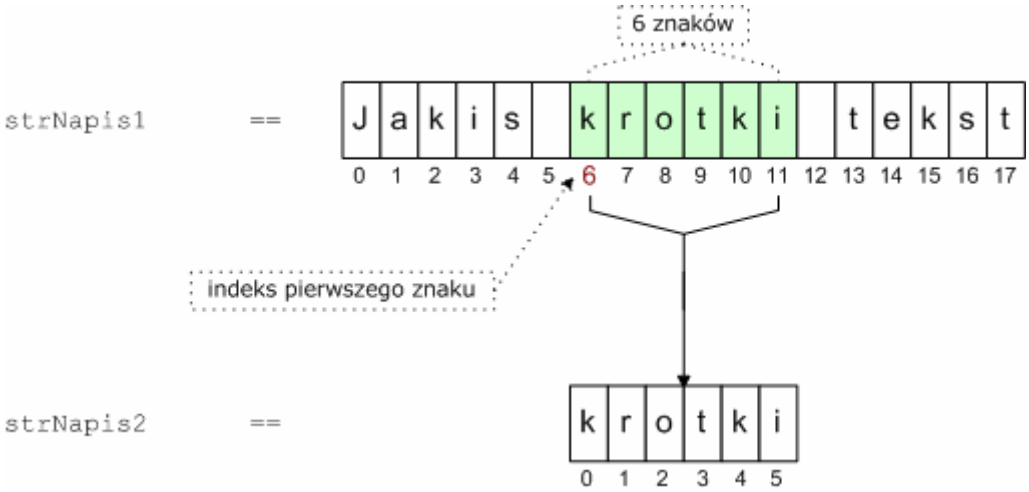
```
std::string strNapis("To jest jakis tekst", 7);
```

Jaki efekt otrzymamy tą drogą? Otóż do naszej zmiennej zostanie przypisany jedynie fragment podanego tekstu - dokładniej mówiąc, będzie to podana w drugim parametrze ilość znaków, liczonych od początku napisu. U nas jest to zatem sekwencja `"To jest"`.

Co ciekawe, to wcale nie są wszystkie sposoby na inicjalizację zmiennej tekstowej. Poznamy jeszcze jeden, który jest wyjątkowo użyteczny. Pozwala bowiem na uzyskanie ściśle określonego „kawałka” danego tekstu. Rzućmy okiem na poniższy kod, aby zrozumieć tą metodę:

```
std::string strNapis1 = "Jakis krotki tekst";
std::string strNapis2(strNapis1, 6, 6);
```

Tym razem mamy aż dwa parametry, które razem określają fragment tekstu zawartego w zmiennej `strNapis1`. Pierwszy z nich (`6`) to indeks pierwszego znaku tegoż fragmentu - tutaj wskazuje on na **siódmy** znak w tekście (gdyż znaki liczymy zawsze **od zera**!). Drugi parametr (znowuż `6`) precyzuje natomiast długość pożądanego urywka - będzie on w tym przypadku sześcioznakowy.
Jeżeli takie opisowe wyjaśnienie nie bardzo do ciebie przemawia, spójrz na ten poglądowy rysunek:



Schemat 7. Pobieranie wycinka tekstu ze zmiennej typu `std::string`

Widać więc czarno na białym (i na zielonym :)), że kopiowaną częścią tekstu jest wyraz `"krotki"`.

Podsumowując, poznaliśmy przed momentem trzy nowe sposoby na inicjalizację zmiennej typu tekstowego:

```
std::[w]string nazwa_zmiennej([L]"tekst");
std::[w]string nazwa_zmiennej([L]"tekst", ilość_znaków);
std::[w]string nazwa_zmiennej(inna_zmienna, początek [, długość]);
```

Ich składnia, podana powyżej, dokładnie odpowiada zaprezentowanym wcześniej przykładowym kodom. Zaskoczenie może jedynie budzić fakt, że w trzeciej metodzie nie jest obowiązkowe podanie *długości* kopiowanego fragmentu tekstu. Dzieje się tak, gdyż w przypadku jej pominięcia pobierane są po prostu wszystkie znaki od podanego indeksu aż do końca napisu.

Kiedy opuścimy parametr *długość*, wtedy trzeci sposób inicjalizacji staje się bardzo podobny do drugiego. Nie możesz jednak ich mylić, gdyż w każdym z nich liczby podawane jako drugi parametr znaczą coś innego. Wyrażają one albo **ilość znaków**, albo **indeks znaku**, czyli wartości pełniące zupełnie odrębne role.

Łączenie napisów

Skoro zatem wiemy już wszystko, co wiedzieć należy na temat deklaracji i inicjalizacji zmiennych tekstowych, zajmijmy się działaniami, jakie możemy nań wykonywać.

Jedną z najpowszechniejszych operacji jest złączenie dwóch napisów w jeden - tak zwana **konkatenacja**. Można ją uznać za tekstowy odpowiednik dodawania liczb, szczególnie że przeprowadzamy ją także za pomocą operatora `+`:

```
std::string strNapis1 = "gra";
std::string strNapis2 = "ty";
std::string strWynik = strNapis1 + strNapis2;
```

Po wykonaniu tego kodu zmienna `strWynik` przechowuje rezultat połączenia, którym są oczywiście `"graty"` :D Widzimy więc, iż scalenie zostaje przeprowadzone w kolejności ustalonej przez porządek argumentów operatora `+`, zaś pomiędzy poszczególnymi składnikami nie są wstawiane żadne dodatkowe znaki. Nie rozminę się chyba z prawdą, jeśli stwierdzę, że można było się tego spodziewać :)

Konkatenacja może również zachodzić między większą liczbą napisów, a także między tymi zapisanymi w sposób dosłowny w kodzie:

```
std::string strImie = "Jan";
std::string strNazwisko = "Nowak";
std::string strImieINazwisko = strImie + " " + strNazwisko;
```

Tutaj otrzymamy personalia pana Nowaka zapisane w postaci ciągłego tekstu, ze spacją wstawioną pomiędzy imieniem i nazwiskiem.

Jeśli chciałbyś połączyć dwa teksty wpisane bezpośrednio w kodzie (np. `"jakis tekst"` i `"inny tekst"`), choćby po to żeby rozbić długi napis na kilka linijek, **nie możesz** stosować do niego operatora +. Zapis `"jakis tekst" + "inny tekst"` będzie niepoprawny i odrzucony przez kompilator.
Zamiast niego wpisz po prostu `"jakis tekst" "inny tekst"`, stawiając między obydwoma stałymi jedynie spacje, tabulatory, znaki końca wiersza itp.

Podobieństwo łączenia znaków do dodawania jest na tyle duże, iż możemy nawet używać skróconego zapisu poprzez operator `+=`:

```
std::string strNapis = "abc";
strNapis += "def";
```

W powyższy sposób otrzymamy więc sześć pierwszych małych liter alfabetu - `"abcdef"`.

Pobieranie pojedynczych znaków

Ostatnią przydatną operacją na napisach, jaką teraz poznamy, jest uzyskiwanie pojedynczego znaku o ustalonym indeksie.

Być może nie zdajesz sobie z tego sprawy, ale już potrafisz to zrobić. Zamierzony efekt można bowiem osiągnąć, wykorzystując jeden ze sposobów na inicjalizację łańcucha:

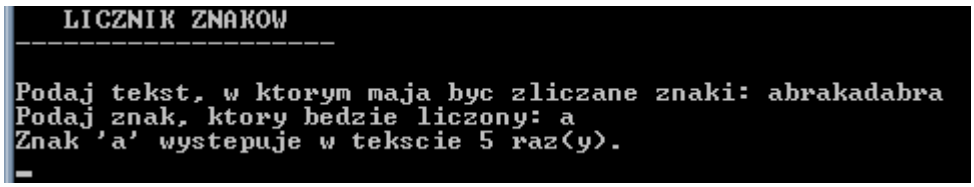
```
std::string strNapis = "przykladowy tekst";
std::string strZnak(strNapis, 9, 1);   // jednoznakowy fragment od ind. 9
```

Tak oto uzyskamy dziesiąty znak (przypominam, indeksy liczymy od zera!) z naszego przykładowego tekstu - czyli `'w'`.

Przyznasz jednak, że taka metoda jest co najmniej kłopotliwa i byłoby ciężko używać jej na co dzień. Dobry C++ ma więc w zanadrzu inną konstrukcję, którą zobaczymy w niniejszym przykładowym programie:

```
// CharCounter - zliczanie znaków

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

unsigned ZliczZnaki(std::string strTekst, char chZnak)
{
   unsigned uIlosc = 0;

   for (unsigned i = 0; i <= strTekst.length() - 1; ++i)
   {
         if (strTekst[i] == chZnak)
               ++uIlosc;
   }

   return uIlosc;
}

void main()
{
   std::string strNapis;
   std::cout << "Podaj tekst, w ktorym maja byc zliczane znaki: ";
   std::cin >> strNapis;

   char chSzukanyZnak;
   std::cout << "Podaj znak, ktory bedzie liczony: ";
   std::cin >> chSzukanyZnak;

   std::cout << "Znak '" << chSzukanyZnak <<"' wystepuje w tekscie "
             << ZliczZnaki(strNapis, chSzukanyZnak) << " raz(y)."
             << std::endl;

   getch();
}
```

Ta prosta aplikacja zlicza nam ilość wskazanych znaków w podanym napisie i wyświetla wynik.



Screen 28. Zliczanie znaków w akcji

Czyni to poprzez funkcję `ZliczZnaki()`, przyjmującą dwa parametry: napis oraz znak, który ma być liczony. Ponieważ jest to najważniejsza część naszego programu, przyjrzymy się jej bliżej :)

Najbardziej oczywistym sposobem na dokonanie podobnego zliczania jest po prostu przebiegnięcie po wszystkich znakach tekstu odpowiednią pętlą `for` i sprawdzanie, czy nie są równe szukanemu znakowi. Każde udane porównanie skutkuje inkrementacją zmiennej przechowującej wynik funkcji. Wszystko to dzieje się w poniższym kawałku kodu:

```
for (unsigned i = 0; i <= strTekst.length() - 1; ++i)
{
   if (strTekst[i] == chZnak)
         ++uIlosc;
}
```

Jak już kilkakrotnie i natarczywie przypominałem, indeksy znaków w zmiennej tekstowej liczymy od zera, zatem są one z zakresu `<0; n-1>`, gdzie `n` to długość tekstu. Takie też wartości przyjmuje licznik pętli `for`, czyli `i`. Wyrażenie `strTekst.length()` zwraca nam bowiem długość łańcucha `strTekst`.
Wewnątrz pętli szczególnie interesujące jest dla nas porównanie:

```
if (strTekst[i] == chZnak)
```

Sprawdza ono, czy aktualnie „przerabiany” przez pętlę znak (czyli ten o indeksie równym `i`) nie jest takim, którego szukamy i zliczamy. Samo porównanie nie byłoby dla nas niczym nadzwyczajnym, gdyby nie owe wyławianie znaku o określonym indeksie (w tym przypadku `i`-tym). Widzimy tu wyraźnie, że można to zrobić pisząc po prostu żądany indeks w **nawiasach kwadratowych** `[ ]` za nazwą zmiennej tekstowej.

Ze swej strony dodam tylko, że możliwe jest nie tylko odczytywanie, ale i zapisywanie takich pojedynczych znaków. Gdybyśmy więc umieścili w pętli następującą linijkę:

```
strTekst[i] = '.';
```

zmienilibyśmy wszystkie znaki napisu `strTekst` na kropki.

Pamiętajmy, żeby pojedyncze znaki ujmować w apostrofy (`''`), zaś cudzysłowy (`""`) stosować dla stałych tekstowych.

***

Tak oto zakończyliśmy ten krótki opis operacji na łańcuchach znaków w języku C++. Nie jest to jeszcze cały potencjał, jaki oferują nam zmienne tekstowe, ale z pomocą

zdobytych już wiadomości powinieneś radzić sobie całkiem nieźle z prostym przetwarzaniem tekstu.

Na koniec tego rozdziału poznamy natomiast typ logiczny i podstawowe działania wykonywane na nim. Pozwoli nam to między innymi łatwiej sterować przebiegiem programu przy użyciu instrukcji warunkowych.

Wyrażenia logiczne

Sporą część poprzedniego rozdziału poświęciliśmy na omówienie konstrukcji sterujących, takich jak na przykład pętle. Pozwalają nam one wpływać na przebieg wykonywania programu przy pomocy odpowiednich **warunków**.
Nasze pierwsze wyrażenia tego typu były bardzo proste i miały dość ograniczone możliwości. Przyszła więc pora na powtórzenie i rozszerzenie wiadomości na ten temat. Zapewne bardzo się z tego cieszysz, prawda? ;)) Zatem niezwłocznie zaczynajmy.

Porównywanie wartości zmiennych

Wszystkie warunki w języku C++ opierają się na jawnym lub ukrytym porównywaniu dwóch wartości. Najczęściej jest ono realizowane poprzez jeden ze specjalnych **operatorów porównania**, zwanych czasem **relacyjnymi**. Wbrew pozorom nie są one dla nas niczym nowym, ponieważ używaliśmy ich w zasadzie w każdym programie, w którym musieliśmy sprawdzać wartość jakiejś zmiennej. W poniższej tabelce znajdziesz więc jedynie starych znajomych :)

operator	***porównanie jest prawdziwe, gdy***
==	lewy argument jest równy prawemu
!=	lewy argument nie jest równy prawemu (jest od niego różny)
>	lewy argument ma większą wartość niż prawy
>=	lewy argument ma wartość większą lub równą wartości prawego
<	lewy argument ma mniejszą wartość niż prawy
<=	lewy argument ma wartość mniejszą lub równą wartości prawego

Tabela 7. Operatory porównania w C++

Dodatkowym ułatwieniem jest fakt, że każdy z tych operatorów ma swój matematyczny odpowiednik - na przykład dla >= jest to ≥, dla != mamy ≠ itd. Sądzę więc, że symbole te nie będą ci sprawiać żadnych trudności. Gorzej może być z następnymi ;)

Operatory logiczne

Doszliśmy oto do sedna sprawy. Nowy rodzaj operatorów, który zaraz poznamy, jest bowiem narzędziem do konstruowania bardziej skomplikowanych wyrażeń logicznych. Dzięki nim możemy na przykład uzależnić wykonanie jakiegoś kodu od spełnienia **kilku** podanych warunków lub tylko jednego z wielu ustalonych; możliwe są też bardziej zakręcone kombinacje. Zaznajomienie się z tymi operatorami da nam więc pełną swobodę sterowania działaniem programu.

Ubolewam, iż nie mogę przedstawić ciekawych i interesujących przykładowych programów na ilustrację tego zagadnienia. Niestety, choć operatory logiczne są niemal stale używane w programowaniu poważnych aplikacji, trudno o ewidentne przykłady ich głównych zastosowań - może dlatego, że stosuje się je prawie do wszystkiego? :)
Musisz więc zadowolić się niniejszymi, dość trywialnymi kodami, ilustrującymi funkcjonowanie tych elementów języka.

Koniunkcja

Pierwszy z omawianych operatorów, oznaczany poprzez `&&`, zwany jest **koniunkcją** lub **iloczynem logicznym**. Gdy wstawimy go między dwoma warunkami, pełni rolę spójnika „i". Takie wyrażenie jest prawdziwe tylko wtedy, kiedy **oba** te warunki są **spełnione**. Operator ten można wykorzystać na przykład do sprawdzania przynależności liczby do zadanego przedziału:

```
int nLiczba;
std::cout << "Podaj liczbe z zakresu 1-10: ";
std::cin >> nLiczba;

if (nLiczba >= 1 && nLiczba <= 10)
   std::cout << "Dziekujemy.";
else
   std::cout << "Nieprawidlowa wartosc!";
```

Kiedy dana wartość należy do przedziału `<1; 10>`? Oczywiście wtedy, gdy jest **jednocześnie** większa lub równa jedynce **i** mniejsza lub równa dziesiątce. To właśnie sprawdzamy w warunku:

```
if (nLiczba >= 1 && nLiczba <= 10)
```

Operator `&&` zapewnia, że całe wyrażenie (`nLiczba >= 1 && nLiczba <= 10`) zostanie uznane za prawdziwe jedynie w przypadku, gdy obydwa składniki (`nLiczba >= 1`, `nLiczba <= 10`) będą przedstawiały prawdę. To jest właśnie istotą koniunkcji.

Alternatywa

Drugi rodzaj operacji, zwany **alternatywą** lub **sumą logiczną**, stanowi niejako przeciwieństwo pierwszego. O ile koniunkcja jest prawdziwa jedynie w jednym, ściśle określonym przypadku (gdy oba jej argumenty są prawdziwe), o tyle alternatywa jest tylko w jednej sytuacji **fałszywa**. Dzieje się tak wtedy, gdy **obydwa** złączone nią wyrażenia przedstawiają **nieprawdę**.
W C++ operatorem sumy logicznej jest `||`, co widać na poniższym przykładzie:

```
int nLiczba;
std::cin >> nLiczba;

if (nLiczba < 1 || nLiczba > 10)
   std::cout << "Liczba spoza przedzialu 1-10.";
```

Uruchomienie tego kodu spowoduje wyświetlenie napisu w przypadku, gdy wpisana liczba nie będzie należeć do przedziału `<1; 10>` (czyli odwrotnie niż w poprzednim przykładzie). Naturalnie, stanie się tak wówczas, jeśli będzie ona mniejsza od `1` **lub** większa od `10`. Taki też warunek posiada instrukcja `if`, a osiągnęliśmy go właśnie dzięki operatorowi alternatywy.

Negacja

Jak można było zauważyć, alternatywa `nLiczba < 1 || nLiczba > 10` jest dokładnie przeciwstawna koniunkcji `nLiczba >= 1 && nLiczba <= 10` (co jest dość oczywiste - przecież liczba nie może jednocześnie należeć i nie należeć do jakiegoś przedziału :D). Warunki te znacznie różnią się od siebie: stosujemy w nich przecież różne działania logiczne oraz porównania. Moglibyśmy jednak postąpić inaczej.
Aby zmienić sens wyrażenia na odwrotny - tak, żeby było prawdziwe w sytuacjach, kiedy oznaczało fałsz i na odwrót - stosujemy operator **negacji** `!`. W przeciwieństwie do

poprzednich, jest on unarny, gdyż przyjmuje tylko jeden argument: warunek do zanegowania.
Stosując go dla naszej przykładowej koniunkcji:

```
if (nLiczba >= 1 && nLiczba <= 10)
```

otrzymalibyśmy wyrażenie:

```
if (!(nLiczba >= 1 && nLiczba <= 10))
```

które jest prawdziwe, gdy dana liczba **nie należy** do przedziału `<1; 10>`. Jest ono zatem równoważne alternatywnie `nLiczba < 1 || nLiczba > 10`, a o to przecież nam chodziło :)

W ten sposób (niechcący ;D) odkryliśmy też jedno z tzw. praw de Morgana. Mówi ono, że zaprzeczenie (negacja) koniunkcji dwóch wyrażeń równe jest alternatywnie wyrażeń przeciwstawnych. A ponieważ `nLiczba >= 1` jest odwrotne do `nLiczba < 1`, zaś `nLiczba <= 10` do `nLiczba > 10`, możemy naocznie stwierdzić, że prawo to jest słuszne :)

Czasami więc użycie operatora negacji uwalnia od konieczności przekształcania złożonych warunków na ich przeciwieństwa.

Zestawienie operatorów logicznych

Zasady funkcjonowania operatorów logicznych ujmuje się często w tabelki, przedstawiające ich wartości dla wszystkich możliwych argumentów. Niekiedy nazywa się je **tablicami prawd** (ang. *truth tables*). Nie powinno więc zabraknąć ich tutaj, zatem czym prędzej je przedstawiam:

a	***b***	***a && b***	***a || b***
prawda	prawda	prawda	prawda
prawda	fałsz	fałsz	prawda
fałsz	prawda	fałsz	prawda
fałsz	fałsz	fałsz	fałsz

a	***!a***
prawda	fałsz
fałsz	prawda

Tabele 8 i 9. Rezultaty działania operatorów koniunkcji, alternatywy oraz negacji

Oczywiście, nie ma najmniejszej potrzeby, abyś uczył się ich na pamięć (a już się bałeś, prawda? :D). Jeżeli uważnie przeczytałeś opisy każdego z operatorów, to tablice te będą dla ciebie jedynie powtórzeniem zdobytych wiadomości.

Najważniejsze są bowiem proste reguły, rządzące omawianymi operacjami. Powtórzmy je zatem raz jeszcze:

Koniunkcja (`&&`) jest **prawdziwa** tylko wtedy, kiedy **oba** jej argumenty są **prawdziwe**.

Alternatywa (`||`) jest **fałszywa** jedynie wówczas, gdy **oba** jej argumenty są **fałszywe**.

Negacja (`!`) powoduje **zmianę** prawdy na fałsz lub fałszu na prawdę.

Łączenie elementarnych wyrażeń przy pomocy operatorów pozwala na budowę dowolnie skomplikowanych warunków, regulujących funkcjonowanie każdej aplikacji. Gdy zaczniesz używać tych działań w swoich programach, zdziwisz się, jakim sposobem mogłeś w ogóle kodować bez nich ;)

Ponieważ operatory logiczne mają niższy priorytet niż operatory porównania, nie ma potrzeby stosowania nawiasów w warunkach podobnych do tych zaprezentowanych. Jeżeli jednak będziesz łączył większą liczbę wyrażeń logicznych, pamiętaj o używaniu nawiasów - to zawsze rozstrzyga wszelkie nieporozumienia i pomaga w uniknięciu niektórych błędów.

Typ `bool`

Przydatność wyrażeń logicznych byłaby dość ograniczona, gdyby można je było stosować tylko w warunkach instrukcji `if` i pętli. Zdecydowanie przydałby się sposób na zapisywanie wyników obliczania takich wyrażeń, by móc je potem choćby przekazywać do i z funkcji.

C++ dysponuje rzecz jasna odpowiednim typem zmiennych, nadającym się to tego celu. Jest nim tytułowy `bool`[50]. Można go uznać za najprostszy typ ze wszystkich, gdyż może przyjmować jedynie dwie dozwolone wartości: prawdę (`true`) lub fałsz (`false`). Odpowiada to prawdziwości lub nieprawdziwości wyrażeń logicznych.

Mimo oczywistej prostoty (a może właśnie dzięki niej?) typ ten ma całe multum różnych zastosowań w programowaniu. Jednym z ciekawszych jest przerywanie wykonywania zagnieżdżonych pętli:

```
bool bKoniec = false;

while (warunek_pętli_zewnętrznej)
{
   while (warunek_pętli_wewnętrznej)
   {
         kod_pętli

         if (warunek_przerwania_obu_pętli)
         {
               // przerwanie pętli wewnętrznej
               bKoniec = true;
               break;
         }
   }

   // przerwanie pętli zewnętrznej, jeżeli zmienna bKoniec
   // jest ustawiona na true
   if (bKoniec) break;
}
```

Widać tu klarownie, że zmienna typu `bool` reprezentuje wartość logiczną - możemy ją bowiem bezpośrednio wpisać jako warunek instrukcji `if`; nie ma potrzeby korzystania z operatorów porównania.

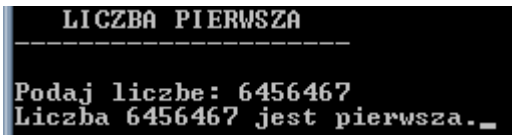
W praktyce często stosuje się funkcje zwracające wartość typu `bool`. Poprzez taki rezultat mogą one powiadamiać o powodzeniu lub niepowodzeniu zleconej im czynności albo sprawdzać, czy dane zjawisko zachodzi, czy nie.
Przyjrzyjmy się takiemu właśnie przykładowi funkcji:

```
// IsPrime - sprawdzanie, czy dana liczba jest pierwsza
```

[50] Nazwa pochodzi od nazwiska matematyka George'a Boole'a, twórcy zasad logiki matematycznej (zwanej też algebrą Boole'a).

```
bool LiczbaPierwsza(unsigned uLiczba)
{
   if (uLiczba == 2) return true;

   for (unsigned i = 2; i <= sqrt(uLiczba); ++i)
   {
         if (uLiczba % i == 0)
               return false;
   }

   return true;
}

void main()
{
   unsigned uWartosc;
   std::cout << "Podaj liczbe: ";
   std::cin >> uWartosc;

   if (LiczbaPierwsza(uWartosc))
         std::cout << "Liczba " << uWartosc << " jest pierwsza.";
   else
         std::cout << "Liczba " << uWartosc<< " nie jest pierwsza.";

   getch();
}
```

Mamy tu funkcję `LiczbaPierwsza()` o prostym przeznaczeniu - sprawdza ona, czy podana liczba jest pierwsza[51], czy nie. Produkuje więc wynik, który może być sklasyfikowany w kategoriach logicznych: prawdy (liczba jest pierwsza) lub fałszu (nie jest). Naturalne jest zatem, aby zwracała wartość typu `bool`, co też czyni.



Screen 29. Określanie, czy wpisana liczba jest pierwsza

Wykorzystujemy ją od razu w odpowiedniej instrukcji `if`, przy pomocy której wyświetlamy jeden z dwóch stosownych komunikatów. Dzięki temu, że funkcja `LiczbaPierwsza()` zwraca wartość logiczną, wszystko wygląda ładnie i przejrzyście :)

Algorytm zastosowany tutaj do sprawdzania „pierwszości” podanej liczby jest chyba najprostszy z możliwych. Opiera się na pomyśle tzw. sita Eratostenesa i, jak widać, polega po prostu na sprawdzaniu po kolei wszystkich liczb jako potencjalnych dzielników, aż do wartości pierwiastka kwadratowego badanej liczby.

Operator warunkowy

Z wyrażeniami logicznymi ściśle związany jest jeszcze jeden, bardzo przydatny i wygodny, operator. Jest on kolejnym z licznych mechanizmów C++, które czynią składnię tego języka niezwykle zwartą.

[51] Liczba pierwsza to taka, która ma tylko dwa dzielniki - jedynkę i samą siebie.

Mowa tu o tak zwanym **operatorze warunkowym** `?:`. Użycie go pozwala na uniknięcie, nieporęcznych niekiedy, instrukcji `if`. Nierzadko może się nawet przyczynić do poprawy szybkości kodu.
Jego działanie najlepiej zilustrować na prostym przykładzie. Przypuśćmy, że mamy napisać funkcję zwracają większą wartość spośród dwóch podanych[52]. Ochoczo zabieramy się więc do pracy i produkujemy kod podobny do tego:

```
int max(int nA, int nB)
{
   if (nA > nB) return nA;
   else return nB;
}
```

Możemy jednak użyć operatora ?:, a wtedy funkcja przyjmie bardziej oszczędną postać:

```
int max(int nA, int nB)
{
   return (nA > nB ? nA : nB);
}
```

Znikła nam tu całkowicie instrukcja `if`, gdyż zastąpił ją nasz nowy operator. Porównując obie (równoważne) wersje funkcji `max()`, możemy łatwo wydedukować jego działanie.

Wyrażenie zawierające tenże operator wygląda bowiem tak:

```
warunek ? wartość_dla_prawdy : wartość_dla_fałszu
```

Składa się więc z trzech części - dlatego `?:` nazywany jest czasem **operatorem ternarnym**, przyjmującym trzy argumenty (jako jedyny w C++).
Jego funkcjonowanie jest nadzwyczaj proste. Sprowadza się do obliczenia *`warunku`* oraz podjęcia na jego podstawie odpowiedniej decyzji. Jeśli będzie on prawdziwy, operator zwróci *`wartość_dla_prawdy`*, w innym przypadku - *`wartość_dla_fałszu`*.
Działalność ta jest w oczywisty sposób podobna do instrukcji `if`. Różnica polega na tym, że operator warunkowy manipuluje **wyrażeniami**, a nie instrukcjami. Nie zmienia więc przebiegu programu, lecz co najwyżej wyniki jego pracy.

Kiedy zatem należy go używać? Odpowiedź jest prosta: wszędzie tam, gdzie konstrukcja `if` wykonuje **te same instrukcje** w obu swoich blokach, lecz operuje na **różnych wyrażeniach**. W naszym przykładzie było to zawsze zwracanie wartości przez funkcję (instrukcja `return`), jednak sam rezultat zależał od warunku.

***

I to już wszystko, co powinieneś wiedzieć na temat wyrażeń logicznych, ich konstruowania i używania we własnych programach. Umiejętność odpowiedniego stosowania złożonych warunków przychodzi z czasem, dlatego nie martw się, jeżeli na razie wydają ci się one lekką abstrakcją. Pamiętaj, ćwiczenie czyni mistrza!

Podsumowanie

Nadludzkim wysiłkiem dobrnęliśmy wreszcie do samego końca tego niezwykle długiego i niezwykle ważnego rozdziału. Poznałeś tutaj większość szczegółów dotyczących zmiennych oraz trzech podstawowych typów wyrażeń. Cały ten bagaż będzie ci bardzo

[52] Tutaj ograniczymy się tylko do liczb całkowitych i typu `int`.

przydatny w dalszym kodowaniu, choć na razie możesz być o tym nieszczególnie przekonany :)

Uzupełnieniem wiadomości zawartych w tym rozdziale może być Dodatek B, *Reprezentacja danych w pamięci*. Jeżeli czujesz się na siłach, to zachęcam do jego przeczytania :)

W kolejnym rozdziale nauczysz się korzystania ze złożonych struktur danych, stanowiących chleb powszedni w poważnym kodowaniu - także gier.

Pytania i zadania

Nieubłaganie zbliża się starcie z pracą domową ;) Postaraj się zatem odpowiedzieć na poniższe pytania oraz wykonać zadania.

Pytania

1. Co to jest zasięg zmiennej? Czym się różni zakres lokalny od modułowego?
2. Na czym polega zjawisko przesłaniania nazw?
3. Omów działanie poznanych modyfikatorów zmiennych.
4. Dlaczego zmienne bez znaku mogą przechowywać większe wartości dodatnie niż zmienne ze znakiem?
5. Na czym polega rzutowanie i jakiego operatora należy doń używać?
6. Który plik nagłówkowy zawiera deklaracje funkcji matematycznych?
7. Jak nazywamy łączenie dwóch napisów w jeden?
8. Opisz funkcjonowanie operatorów logicznych oraz operatora warunkowego

Ćwiczenia

1. Napisz program, w którym przypiszesz wartość `3000000000` (trzy miliardy) do dwóch zmiennych: jednej typu `int`, drugiej typu `unsigned int`. Następnie wyświetl wartości obu zmiennych. Co stwierdzasz?
   (**Trudne**) Czy potrafisz to wyjaśnić?
   Wskazówka: zapoznaj się z podrozdziałem o liczbach całkowitych w Dodatku B.
2. Wymyśl nowe nazwy dla typów `short int` oraz `long int` i zastosuj je w programie przykładowym, ilustrującym działanie operatora `sizeof`.
3. Zmodyfikuj nieco program wyświetlający tablicę znaków ANSI:
   a) zamień cztery wiersze wyświetlające pojedynczy rząd znaków na jedną pętlę `for`
   b) zastąp rzutowanie w stylu C operatorem `static_cast`
   c) (**Trudniejsze**) spraw, żeby program czekał na dowolny klawisz po całkowitym zapełnieniu okna konsoli - tak, żeby użytkownik mógł spokojnie przeglądnąć całą tablicę
   Wskazówka: możesz założyć „na sztywno”, że konsola mieści 24 wiersze
4. Stwórz aplikację podobną do przykładu `LinearEq` z poprzedniego rozdziału, tyle że rozwiązującą równania kwadratowe. Pamiętaj, aby uwzględnić wartość współczynników, przy których równanie staje się liniowe (możesz wtedy użyć kodu ze wspomnianego przykładu).
   Wskazówka: jeżeli nie pamiętasz sposobu rozwiązywania równań kwadratowych (wstyd! :P), możesz zajrzeć na przykład do encyklopedii WIEM.
5. Przyjrzyj się programowi sprawdzającemu, czy dana liczba jest pierwsza i spróbuj zastąpić występującą tam instrukcję `if-else` operatorem warunkowym `?:`.