Podstawy programowania w C++

Zmienne i operacje na zmiennych

Opracował: Andrzej Nowak

Bibliografia:

Symfonia C++ (Tom I, II i III); J. Grębosz
Nauka programowania dla początkujących; A. Struzińska-Walczak / K. Walczak
http://eduinf.waw.pl/inf/alg/001_search/0001.php
CPA: PROGRAMMING ESSENTIALS IN C++ https://www.netacad.com

Zmienna (ang. variable) jest fragmentem pamięci komputera, w którym program przechowuje określoną informację.
Przed pierwszym użyciem zmiennej musimy ją zadeklarować, czyli określić rodzaj przechowywanej w niej informacji oraz nazwę, poprzez którą będziemy się odwoływali w programie do tej informacji.
Zmienne wykorzystujemy w programach do przetwarzania danych.

Deklaracja zmiennej w języku C++ jest następująca:

```
[modyfikator] typ_zmiennej nazwa_zmiennej;
```

`modyfikator`

Zmienne typów podstawowych mogą posiadać tzw. modyfikator, czyli instrukcję która dokładniej opisuje kompilatorowi właściwości zmiennej

`signed` - modyfikator określający czy dana zmienna będzie mogła przechowywać liczby ujemne

`unsigned` - modyfikator określający czy dana zmienna będzie mogła przechowywać tylko liczby dodatnie, ale o podwojonym zakresie

Uwaga:

Jeśli nie podamy modyfikatora kompilator domyślnie przypisuje modyfikator **`signed`** do nowo zadeklarowanej zmiennej.

`typ_zmiennej`

określa rodzaj (typ danych) przechowywanej w zmiennej informacji.

`nazwa_zmiennej`

jest napisem, poprzez który uzyskujemy dostęp do informacji przechowywanej w zmiennej.

Nazwy zmiennych zbudowane są z liter małych i dużych, cyfr oraz znaku podkreślenia. Pierwszym znakiem nazwy zmiennej powinna być litera (nie może być nim cyfra!). Zmienna

nie powinna posiadać nazwy dłuższej od 31 znaków (tyle zwykle zapamiętują kompilatory C++).
Nie wolno również zmiennym nadawać nazw identycznych ze słowami kluczowymi języka C++ (np. return, int, unsigned, itp.).
Nazwa zmiennej powinna kojarzyć się ze spełnianą funkcją w programie:
lepiej użyć nazw

Wynik, **nazwisko**, **konto**

niż

Q12XC5, **r74SY8** czy **a_7dnm99VCD**

Typy danych

Typ – opis rodzaju, struktury i zakresu wartości, jakie może przyjmować dany literał, zmienna, stała, argument, wynik funkcji lub wartość.

1. Typ całkowity

Powszechnie używanym do reprezentacji liczb całkowitych w systemach komputerowych jest kod U2. Dla liczb tylko dodatnich używa się kodu NBC (naturalny kod binarny).

`int`

32 bitowe liczby całkowite w **kodzie U2**.
Zakres: $-2^{31}...2^{31} - 1$, -2147483648...2147483647.
Każda zmienna **int** zajmuje w pamięci komputera 4 bajty.

`unsigned int`

32 bitowe liczby całkowite w **naturalnym kodzie binarnym (NBC)**.
Zakres: $0...2^{32} - 1$, 0...4294967295.
Każda zmienna **unsigned int** zajmuje w pamięci komputera 4 bajty.

`short int`

16 bitowe liczby całkowite w **kodzie U2**.
Zakres: $-2^{15}...2^{15} - 1$, -32768...32767.
Każda zmienna **short int** zajmuje w pamięci komputera 2 bajty.

`unsigned short int`

16 bitowe liczby całkowite w **naturalnym kodzie binarnym (NBC)**.
Zakres: $0...2^{16} - 1$, 0...65535.
Każda zmienna **unsigned short int** zajmuje w pamięci komputera 2 bajty.

`long long int`

64 bitowe liczby całkowite w **kodzie U2**.
Zakres: $-2^{63}...2^{63} - 1$, -9223372036854775808...9223372036854775807.
Każda zmienna **long long int** zajmuje w pamięci komputera 8 bajtów.

`unsigned long long int`

64 bitowe liczby całkowite w **naturalnym kodzie binarnym (NBC)**.
Zakres: $0...2^{64} - 1$, 0...18446744073709551615.
Każda zmienna **unsigned long long int** zajmuje w pamięci komputera 8 bajtów.

2. Typ zmiennoprzecinkowy (rzeczywisty)

Do przechowywania liczb ułamkowych używamy zmiennych typu zmiennoprzecinkowego (ang. floating point type). W typach zmiennoprzecinkowych

ważniejszym parametrem od zakresu jest tzw. precyzja, która określa dokładność zapamiętywania liczb.

`float`

32 bitowe liczby zmiennoprzecinkowe o pojedynczej precyzji.
Precyzja 7-8 cyfr.
Każda zmienna **`float`** zajmuje w pamięci komputera 4 bajty.

`double`

64 bitowe liczby zmiennoprzecinkowe o podwójnej precyzji.
Precyzja 15 cyfr.
Każda zmienna **`double`** zajmuje w pamięci komputera 8 bajtów.

`long double`

80 bitowe liczby zmiennoprzecinkowe o rozszerzonej precyzji.
Precyzja 20 cyfr.
Każda zmienna **`long double`** zajmuje w pamięci komputera 12 bajtów, z których na dane wykorzystuje się tylko 10, a 2 są uzupełnieniem do granicy słów 4-bajtowych. Słowa 4 bajtowe procesor Pentium pobiera z pamięci w jednym cyklu i stąd takie rozwiązanie.

3. **Typ znakowy**

`char`

8 bitowe liczby całkowite w **kodzie U2**.
Zakres: $-2^7 ... 2^7 - 1$, -128...127.
Każda zmienna **char** zajmuje w pamięci komputera 1 bajt. Typ ten jest interpretowany przez operacje wejścia wyjścia jako typ znakowy - jedna zmienna char może przechować jeden znak ASCII. Poza tym zmienne char możemy traktować jak zwykłe zmienne całkowite.

`unsigned char`

8 bitowe liczby całkowite w **naturalnym kodzie binarnym (NBC)**.
Zakres: 0 do $2^8 - 1$, 0...255.
Każda zmienna **unsigned char** zajmuje w pamięci komputera 1 bajt. Typ ten jest interpretowany przez operacje wejścia wyjścia jako typ znakowy.

4. **Typ tekstowy**

`string`

Typ tekstowy posiada bibliotekę standardową z zaimplementowaną uogólnioną klasą napisów zwaną **string**. Taka klasa daje jednolity, niezależny od systemu i bezpieczny interfejs do manipulowania napisami.

Aby móc korzystać z klasy **string** należy dołączyć plik nagłówkowy:

```
#include <string>
```

5. Typ logiczny

bool

1 bitowa dana logiczna.
Każda zmienna **bool** zajmuje w pamięci komputera 1 bajt. Wartość zmiennej jest przechowywana w najmłodszym bicie. Pozostałe 7 bitów jest niewykorzystywane.

Przy przypisywaniu wartości zmiennej kompilator C++ dokonuje niejawnej konwersji:

```
zmienna = wyrażenie;
```

Wartość wyrażenia jest obliczana, a następnie dokonana zostaje konwersja tej wartości zgodnie z typem zmiennej. Dla zmiennych **bool** wartość wyrażenia różna od 0 oznacza prawdę - zostanie ustawiony ostatni bit zmiennej i w efekcie otrzymamy wartość 1:

Przykłady deklaracji i zmiennych:

```
// Przykładowy program w C++
//---------------------------

#include <iostream>


using namespace std;


int main()
{
    int to_jest_zmienna;
    unsigned int tylkoDodatnie;

    float zmiennoprzecinkowa;

    char jeden_znak;
    unsigned char rowniez_znak;

    int abc = - 53;

    tylkoDodatnie = 22;
    zmiennoprzecinkowa = 12.42;
    rowniez_znak = 'c';

    cout << "Wypisujemy zmienne:" << endl;
    cout << "  to_jest_zmienna = " << to_jest_zmienna << endl;
    cout << "  tylkoDodatnie = " << tylkoDodatnie << endl;
    cout << "  abc = " << abc << endl;
    cout << "  zmiennoprzecinkowa = " << zmiennoprzecinkowa << endl;
    cout << "  jeden_znak = " << jeden_znak << endl;
    cout << "  rowniez_znak = " << rowniez_znak << endl;
    return 0;
}
```

Efekt działania powyższego programu może wyglądać np. tak:

```
Wypisujemy zmienne:
  to_jest_zmienna = 2293728
  tylkoDodatnie = 22
  abc = -53
  zmiennoprzecinkowa = 12.42
  jeden_znak =
  rowniez_znak = c

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

Zwróć uwagę na wypisane wartości zmiennych, którym nie została przypisana wartość początkowa. Wartości są przypadkowe. Warto więc **inicjalizować** wszystkie zmienne początkową wartością.

Co to jest inicjalizacja

Inicjalizacja jest to nadawanie początkowej wartości zmiennej w chwili jej tworzenia.

Zakres zmiennej

Aby sprawdzić ile w rzeczywistości zajmuje określony typ danych dla Twojego kompilatora, wystarczy napisać prosty program i wykorzystać w nim słowo kluczowe **sizeof**.

```
// Przykładowy program w C++
//--------------------------
#include <iostream>
#include <cstdlib>

using namespace std;


int main()
{
   cout << "sizeof(bool) = " << sizeof( bool ) <<endl;
   cout << "sizeof(char) = " << sizeof( char ) <<endl;
   cout << "sizeof(unsigned char) = " << sizeof( unsigned char ) <<endl;
   cout << "sizeof(wchar_t) = " << sizeof( wchar_t ) <<endl;
   cout << "sizeof(short) = " << sizeof( short ) <<endl;
   cout << "sizeof(unsigned short) = " << sizeof( unsigned short ) <<endl;
   cout << "sizeof(int) = " << sizeof( int ) <<endl;
   cout << "sizeof(unsigned int) = " << sizeof( unsigned int ) <<endl;
   cout << "sizeof(long) = " << sizeof( long ) <<endl;
   cout << "sizeof(unsigned long) = " << sizeof( unsigned long ) <<endl;
   cout << "sizeof(long long) = " << sizeof( long long ) <<endl;
   cout << "sizeof(float) = " << sizeof( float ) <<endl;
   cout << "sizeof(double) = " << sizeof( double ) <<endl;
   cout << "sizeof(long double) = " << sizeof( long double ) <<endl;
    return 0;
}
```

Wcześniej zapoznaliśmy się z pojęciem typu danych i dowiedzieliśmy się, że jednym z podstawowych typów znanych w języku C ++ jest typ integer o nazwie `int`.

Teraz pora omówić inny typ, zaprojektowanym do reprezentowania i przechowywania liczb, które (jak by powiedział matematyk) mają niepustą część dziesiętną.

Są to liczby, które mają (lub mogą mieć) ułamkową część po przecinku dziesiętnym, i chociaż jest to bardzo uproszczona definicja, wystarcza do naszych celów.

Ilekroć używamy terminu "dwa i pół" lub "kropka zero cztery", myślimy o liczbach, które komputer uważa za liczby zmiennoprzecinkowe.

2.5

"Dwa i pół" zapisane z przecinkiem jest nie do zaakceptowania przez kompilator.

Kompilator albo go nie zaakceptuje, albo (w bardzo rzadkich okolicznościach) źle zrozumie twoje intencje, ponieważ sam przecinek ma własne zarezerwowane znaczenie w języku C ++.

Jeśli chcesz użyć wartości "dwa i pół", powinieneś zapisać ją tak, jak pokazano na obrazku powyżej.

.4

W języku C++ zapis "kropka cztery" jest dopuszczalny - możesz pominąć zero, gdy jest to jedyna cyfra przed przecinkiem dziesiętnym lub po nim.

W zasadzie powinieneś wpisać wartość 0.4

Jeśli chcesz używać dowolnych liczb, które są bardzo duże lub bardzo małe, możesz użyć tak zwanej notacji naukowej.

Weźmy na przykład prędkość światła wyrażoną w metrach na sekundę. Zapisana bezpośrednio będzie wyglądała następująco:

```
300000000  m/s
```

Aby uniknąć nudnej pracy pisania tak wielu zer, podręczniki fizyki używają skróconej formy, którą prawdopodobnie już widziałeś:

$3x10^8$

Oznacza to: trzy razy dziesięć do potęgi ósmej.

W języku C ++ ten sam efekt osiąga się w nieco innej formie - spójrz:

```
3E8
```

Litera E (można również użyć małej litery e - pochodzi od wyrazu wykładniczego) jest zwięzłą reprezentacją frazy razy dziesięć do potęgi.

Uwaga:

wykładnik (wartość po "E") musi być liczbą całkowitą.

podstawa (wartość przed "E") może, ale nie musi być liczbą całkowitą.

Zobaczmy, w jaki sposób konwencja ta jest używana do rejestrowania liczb, które są bardzo małe (w sensie ich bezwzględnej wartości, która jest bliska zeru).

Stała fizyczna o nazwie stałej Plancka (oznaczona jako h), zgodnie z podręcznikiem, ma wartość

```
6.62607 · 10-34
```

Jeśli chciałbyś jej użyć w programie, napisałbyś to w ten sposób:

```
6.62607E-34
```

Wróćmy do wartości zmiennoprzecinkowych.
Wiemy już, czym jest zmienna, a także wiemy, jak zadeklarować zmienną całkowitą, więc teraz czas zadeklarować zmienne typu zmiennoprzecinkowego.
Robimy to za pomocą słowa kluczowego **`float`**.
Przykład:
Możemy zadeklarować dwie zmiennoprzecinkowe zmienne, o nazwie **`PI`** i **`Pole`**.

```
float PI, Pole;
```

zadanie

jakie wartości zostaną wypisane dla zmiennej **i** i **x** ?

```
int i;
float x;

i = 10/4;
x = 10.0 / 4.0;
```

Przeliczanie wartości całkowitych na zmiennoprzecinkowe i odwrotnie

Co się dzieje, gdy musimy przeliczyć wartości całkowite na wartości zmiennoprzecinkowe lub odwrotnie?

Zawsze możemy przekształcić z **`int`** w **`float`**, ale może to prowadzić do utraty dokładności.

Rozważmy przykład:

```
int i;
float f;

i = 100;
f = i;
```

Po zmianie z int na float wartość zmiennej f wynosi 100.0, ponieważ wartość typu int (100) jest automatycznie konwertowana na zmienną (100.0).

Transformacja wpływa na wewnętrzną (maszynową) reprezentację tych wartości, ponieważ komputery używają różnych metod przechowywania w swoich pamięciach elementów zmiennoprzecinkowych i całkowitych.

Rozważmy teraz sytuację odwrotną.

```
int i;
float f;

f = 100.25;
i = f;
```

Jak zapewne się domyślacie, taka zamiana spowodują utratę dokładności - wartość zmiennej `i` wynosi `100`. Dwadzieścia pięć setnych zostanie pominięta, bo typ `int` przechowuje tylko liczby całkowite. Ponadto przekształcenie `float-a` w `int` nie zawsze jest możliwe.

Wynika to z ograniczeń przedziałów liczbowych dla, których są określone typy `int` i `float`.

Operatory

Operator przypisania

Do przetwarzania danych w programie C++ używany jest **operator przypisania** (ang. assignement operator). Posiada on następującą postać:

```
zmienna = wyrażenie;
```

Komputer oblicza wartość wyrażenia i wynik umieszcza w podanej zmiennej.

Znaku = nie traktuj jako równość matematyczną, jest to symbol operacji przypisania.

Przykład:

```
// Program  przeliczający stopnie
// Celsjusza na stopnie Fahrenheita.
//----------------------------------

#include <iostream>

using namespace std;

int main()
{
    int tc,tf;

    cout << "Przeliczanie stopni Celsjusza na stopnie Fahrenheita\n"
            "-----------------------------------------------------\n\n"
            "Temperatura w stopniach Celsjusza?  : ";
    cin  >> tc;

    tf = (9 * tc) / 5 + 32;

    cout << "Temperatura w stopniach Fahrenheita : " << tf << endl << endl;

    return 0;
}



// Program  obliczający drogę w ruchu jednostajnie
// przyspieszonym.
//----------------------------------

#include <iostream>
```

```
using namespace std;

int main()
{
    int s,V0,t,a;

    cout << "Droga w ruchu jednostajnie przyspieszonym\n"
            "----------------------------------------\n\n"
            "Vo = "; cin >> V0;
    cout << "t  = "; cin >> t;
    cout << "a  = "; cin >> a;

    s = t * (V0 + a * t / 2);

    cout << "\n-----------------------------------------------------\n"
            "droga w ruchu jednostajnie przyspieszonym s = " << s << endl;;

    return 0;
}
```

Wyrażenie z operatorem przypisania posiada wartość równą wartości wyrażenia po prawej stronie operatora.

Przykład:

```
cout << (a = 5) << endl;
```

Wynikiem wykonania tej instrukcji jest liczba 5, ponieważ to stoi po prawej stronie operatora =. Dzięki tej własności możemy w języku C++ przypisać tę samą wartość do kilku zmiennych.

Zamiast pisać:

```
a = 5;
b = 5;
c = 5;
```

możemy zapisać prościej:

```
a = b = c = 5;
```

Operatory arytmetyczne

Mnożenie

*

Gwiazdka * to operator mnożenia.

zadanie

jakie wartości zostaną wypisane dla zmiennej **k** i **z** ?

```
int i,j,k;
float x,y,z;

i = 10;
j = 12;
k = i * j;
x = 1.25;
y = 0.5;
z = x * y;
```

Dzielenie

/

Ukośnik / jest operatorem dzielenia.

zadanie

jakie wartości zostaną wypisane dla zmiennej **k** i **z** ?

```
int i,j,k;
float x,y,z;
```

```
i = 10;
j = 5;
k = i / j;
x = 1.0;
y = 2.0;
z = x / y;
```

Dzielenie przez zero

Jak się pewnie domyślacie, dzielenie przez zero jest surowo zabronione, ale kara za naruszenie tej zasady przyjdzie do was w różnych momentach.

Przykład:

```
float x;

x = 1.0/0.0;
```

Jeśli odważysz się napisać coś takiego, kompilator oszaleje - dostaniesz błąd kompilacji, błąd wykonania lub wiadomość w czasie wykonywania.

Zasadniczo nie należy dzielić przez zero.

Przykład:

```
float x,y;

x = 0.0;

y = 1.0 /x;
```

W powyższym przykładzie kompilator nic ci nie powie, ale kiedy spróbujesz wykonać ten kod, wynik operacji może być zaskakujący.

To nie jest liczba. Jest to specjalna funkcja o nazwie **inf**

Zasadniczo tego rodzaju nielegalna operacja jest tak zwanym wyjątkiem.

Kiedy znajdziesz wyjątki w swoim programie, powinieneś odpowiednio zareagować.

Dodawanie

+

Operatorem dodawania jest znak + (plus), który większość z nas zna.

zadanie

jakie wartości zostaną wypisane dla zmiennej **k** i **z** ?

```
int i,j,k;
float x,y,z;

i = 100;
j = 2;
k = i + j;
x = 1.0;
y = 0.02;
z = x + y;
```

Odejmowanie

−

Operatorem odejmowania jest oczywiście znak - (minus), chociaż należy zauważyć, że ten operator ma również inne znaczenie - może zmienić znak liczby.

zadanie

jakie wartości zostaną wypisane dla zmiennej **k** i **z** ?

```
int i,j,k;
float x,y,z;

i = 100;
j = 200;
k = i - j;
x = 1.0;
y = 1.0;
z = x - y;
```

Operator reszty z dzielenia całkowitego %

Operator ten nie ma odpowiednika wśród tradycyjnych operatorów arytmetycznych.

Jest operatorem binarnym (wykonuje operację modulo) i oba argumenty nie mogą być liczba mi zmiennoprzecinkowymi (**nie zapominajcie o tym!).**

Spójrz na przykład:

```
int i,j,k;

i = 13;

j = 5;

k = i % j;
```

Zmienna k wynosi 3 (ponieważ 2 * 5 + 3 = 13).

UWAGA:

Nie możesz obliczyć reszty z właściwym argumentem równym zero.

Operator dzielenia /

Wynik dzielenia uzależniony jest od typu zmiennej wynik – w tym wypadku k jest typu całkowitego, dlatego wynik będzie liczbą całkowitą.

Spójrz na przykład:

```
int i,j,k;

i = 13;

j = 5;

k = i / j;
```

Operatory - jednoargumentowe

Jednoargumentowy minus −

W aplikacjach "odejmujących" operator minus oczekuje dwóch argumentów:

- lewego (minuend)
- prawego (subtrahend).

Z tego powodu operator odejmowania jest uważany za jeden z operatorów binarnych, podobnie jak operatory dodawania, mnożenia i dzielenia.

Ale operator minus może być użyty w inny sposób - spójrz na fragment kodu:

```
int i,j;

i = -100;

j = -i;
```

Jak się pewnie domyślasz, zmiennej `j` zostanie przypisana wartość `100`.

Użyliśmy operatora minus jako operatora jednoargumentowego, ponieważ oczekuje on tylko jednego argumentu - prawego.

Jednoargumentowy plus +

Ta sama dwoistość może być wyrażona przez operator `+`, który możemy również wykorzystać jako operator jednoargumentowy - jego rolą jest zachowanie znaku.

Spójrz na fragment kodu:

```
int i,j;

i = 100;

j = +i;
```

Chociaż taka konstrukcja jest poprawna pod względem składni, użycie jej nie ma większego sensu.

Operatory – relacji (porównań)

==	równy	przykład: `a == 15`
>	większy	przykład: `a + 15 > c`
>=	większy lub równy	przykład: `a >= b + c`
<	mniejszy	przykład: `a < 15`
<=	mniejszy lub równy	przykład: `b + 5 <= a`
!=	różny	przykład: `a != b`

Wyrażenia logiczne

W logice występują jedynie dwie wartości: prawda i **fałsz**. W języku C++ zdefiniowane są specjalne stałe do reprezentowania wartości logicznych:

prawda = **true** = 1

fałsz = **false** = 0

Przykład:

```
// Wartosci logiczne
//---------------------------

#include <iostream>

using namespace std;

int main()
{
    cout << "false = " << false << endl
         << "true  = " << true  << endl;

    return 0;
}
```

Oczywiście zamiast **true** można stosować 1, a zamiast **false** 0 - jednakże posługiwanie się tymi stałymi zwiększa znacznie czytelność programu.

Wyrażenie logiczne (ang. logical expresion) jest wyrażeniem, które może przyjmować tylko wartości logiczne **true** lub **false**.

Tego typu wyrażenia powstają z operatorami relacji (porównań).

Przykład zastosowania operatorów:

```
// Operator porownan - rowny
//--------------------------

#include <iostream>

using namespace std;

int main()
{
    int a = 4;

    cout << (a == 4) << endl
         << (a == 3) << endl;

    return 0;
}


// Operatory porownan - rozny
//---------------------------

#include <iostream>

using namespace std;

int main()
{
    int a = 4;

    cout << (a != 4) << endl
         << (a != 3) << endl;

    return 0;
}


// Operatory porownan - mniejszy
//---------------------------

#include <iostream>

using namespace std;

int main()
{
    int a = 4;
```

```
    cout << (a < 5) << endl
         << (a < 3) << endl;

    return 0;
}
// Operatory porownan - większy rowny
//---------------------------

#include <iostream>

using namespace std;

int main()
{
    int a = 4;

    cout << (a >= 5) << endl
         << (a >= 3) << endl;

    return 0;
}
```

Operatory logiczne

- **&&** koniunkcja (iloczyn zdań)
- **||** alternatywa (suma zdań)
- **!** negacja (zaprzeczenie zdań)

Operatory – inkrementacji

Język C++ posiada dwa użyteczne operatory do zwiększania i zmniejszania zawartości zmiennej:

Operator inkrementacji

++ - zwiększa zawartość zmiennej o 1

Operator dekrementacji

-- - zmniejsza zawartość zmiennej o 1

Operatory te można stosować na dwa sposoby:

++ zmienna (preinkrementacja) lub **zmienna ++** (postinkrementacja)
-- zmienna (predekrementacja) lub **zmienna --** (postdekrementacja)

Jeżeli operacja zwiększania zmiennej jest samodzielną instrukcją, to nie ma znaczenia, który z tych sposobów użyjemy.

Poniższe dwa programy dają identyczne wyniki:

```
// Modyfikacja zmiennej
//---------------------

#include <iostream>

using namespace std;

int main()
{
    int a = 125, b = 30;

    ++a;  --b;
    cout << a << " " << b << endl;

    system("pause");
    return 0;
}
```

```
// Modyfikacja zmiennej
//---------------------

#include <iostream>

using namespace std;

int main()
{
  int a = 125, b = 30;

  a++;  b--;
  cout << a << " " << b << endl;

  system("pause");
  return 0;
}
```

W obu programach tworzone są dwie zmienne *a* i *b*. Zmiennej *a* nadajemy wartość 125, a zmiennej *b* 30. Następnie program zwiększa zawartość zmiennej *a* o 1 do 126, a zawartość zmiennej *b* zmniejsza o 1 do 29.

Na koniec zawartości zmiennych są kolejno wyświetlane i otrzymujemy liczby 126 i 29. W pierwszym programie operatory ++ i -- stosujemy przed zmiennymi, a w drugim po zmiennych.

Jeśli operator zwiększania ++ lub zmniejszania -- zastosujemy do zmiennej użytej w wyrażeniu, to bardzo istotne jest, którą z form wybierzemy:

- **++ zmienna, -- zmienna** : najpierw modyfikuje (zmniejsza o 1) zawartość zmiennej, a następnie wynik jest używany do obliczeń w wyrażeniu
- **zmienna ++, zmienna --** : w wyrażeniu stosowana jest bieżąca zawartość zmiennej, a następnie po wyliczeniu wyrażenia zmienna jest modyfikowana (zmniejszana o 1).

Przykład:

Poniższe dwa programy dają różne wyniki:

```
// Modyfikacja zmiennej
//----------------------

#include <iostream>

using namespace std;

int main()
{
  int a = 125, b = 30,c,d;

  c = ++a;   d = --b;

  cout << a << " " << b << " "
       << c << " " << d << endl;

  system("pause");
  return 0;}
```

```
// Modyfikacja zmiennej
//----------------------

#include <iostream>

using namespace std;

int main()
{
  int a = 125, b = 30,c,d;

  c = a++;   d = b--;

  cout << a << " " << b << " "
       << c << " " << d << endl;

  system("pause");
  return 0;}
```

Programy różnią się jedynie sposobem umieszczenia operatorów **++** oraz **--** w stosunku do modyfikowanej zmiennej.

W obu przypadkach zmienna ***a*** przyjmuje na końcu wartość **126**, a zmienna ***b*** **29**. Jednakże w pierwszym programie do zmiennej ***c*** trafi **126** i do ***d*** **29**, a w drugim do ***c*** trafi **125**, a do ***d*** **30**.

W pierwszym programie zmienna ***a*** w wyrażeniu **c = ++a**; jest najpierw zwiększana z **125** na **126** i ten wynik trafia do zmiennej ***c***. podobnie zmienna ***b*** w wyrażeniu **d = --b;** zostanie najpierw zmniejszona do **29** i wynik ten trafi do zmiennej ***d***.

W drugim programie operatory **++** i **--** umieszczone są za zmienną. Zatem w wyrażeniu **c = a++;** do zmiennej ***c*** trafi najpierw zawartość zmiennej ***a***, czyli **125**, a następnie zmienna ***a*** zostanie zwiększona do **126**.

Podobnie w wyrażeniu **d = b--;** do zmiennej ***d*** trafi zawartość ***b***, czyli **30**, a następnie zmienna ***b*** zostanie zmniejszona do **29**.

UWAGA:

Na opisane wyżej zjawisko należy bardzo uważać w programach C++, gdyż często prowadzi ono do trudno wykrywalnych błędów. Najbezpieczniej jest modyfikować zmienną poza wyrażeniami, lecz często robi się to wewnątrz wyrażeń w celu optymalizacji kodu.

Wybór jest twój - zostałeś ostrzeżony.

Operatory – modyfikacji zmiennej

Poza prostymi operatorami zwiększania lub zmniejszania o 1 zawartości zmiennej, język C++ posiada bogatą gamę operatorów modyfikacji:

zmienna **+=** wyrażenie; - zwiększa zawartość zmiennej o wartość wyrażenia:

```
a += 2;  // zmienne a zostaje zwiększona o 2
c += a + b; // zmienna c zostaje zwiększona o sumę zmiennych a i b
```

zmienna **-=** wyrażenie; - zmniejsza zawartość zmiennej o wartość wyrażenia:

```
a -= 3;  // zmienna a zostaje pomniejszona o 3
b -= 2 * c; // zmienna b zostaje pomniejszona o 2 * c
```

zmienna ***=** wyrażenie; - mnoży zawartość zmiennej przez wartość wyrażenia

```
a *= 3;   // zmienna a zostaje pomnożona przez 3
b *= c - d; // zmienna b zostaje pomnożona przez różnicę zmiennych c i d
```

zmienna **/=** wyrażenie; - dzieli zawartość zmiennej przez wartość wyrażenia

```
a /= 4;   // zawartość zmiennej a zostaje podzielona przez 4
b /= c--; // zawartość zmiennej b zostaje podzielona przez c, po czym c zostaje zmniejszone o 1
```

zmienna **%=** wyrażenie; - w zmiennej zostanie umieszczona reszta z dzielenia tej zmiennej przez wartość wyrażenia:

```
a %= 10; // w zmiennej a znajdzie się reszta z dzielenia a przez 10, czyli ostatnia cyfra dziesiętna
a %= 2;  // w zmiennej a będzie 0, jeśli a było parzyste, lub 1, jeśli a było nieparzyste
```

Podane powyżej operatory modyfikacji (jak również instrukcje przypisania) mogą wystąpić w wyrażeniach, lecz zdecydowanie odradzam takie rozwiązania poza szczególnymi wyjątkami - prowadzą one do bardzo zawiłych kodów, które trudno później analizować:

```
a = 1; b = 2; c = 3;
```

Kolejność działań – hierarchia priorytetów

Do tej pory traktowaliśmy każdy operator tak, jakby nie miał on żadnego związku z innymi. Oczywiście w prawdziwym programowaniu nic nie jest tak proste.

Ponadto bardzo często musimy zastosować więcej niż jeden operator w wyrażeniu.

Rozważ następujące wyrażenie.

Jeśli twoja pamięć Cię nie zawiodła, powinieneś pamiętać ze szkoły, że mnożenie poprzedza dodawanie. Prawdopodobnie pamiętasz, że musisz pomnożyć 3 przez 5, zachować 15 w pamięci, dodać do 2, uzyskując wynik 17.

Zjawisko, które powoduje, że niektóre operatory działają przed innymi, jest znane jako hierarchia priorytetów.

Język C ++ precyzyjnie określa priorytety wszystkich operatorów i zakłada, że operator o większym (wyższym) priorytecie wykonuje swoje operacje przed operatorem o niższym priorytecie.

Lista priorytetów

Ponieważ jesteśmy na początku nauki języka C ++, przedstawienie pełnej listy priorytetów operatorów może nie być dobrym pomysłem. Zamiast tego poznamy jej skróconą formę i będziemy ją konsekwentnie rozwijać podczas wprowadzania nowych operatorów.

Lista priorytetów od najwyższego do najmniejszego wygląda następująco:

+ -	jednoargumentowe
/ % *	
+ -	binarne

Zadanie

Wykonaj następujące wyrażenie:

Oba operatory (`*` i `%`) mają ten sam priorytet

Nawiasy

Zgodnie z regułami arytmetycznymi podwyrażenia w nawiasach są zawsze obliczane jako pierwsze. Nawiasy mogą zmienić naturalną kolejność obliczeń.

Możesz użyć tyle nawiasów, ile potrzebujesz i często używamy ich do poprawy czytelności wyrażenia, nawet jeśli nie zmieniają one kolejności operacji.

Oto przykład wyrażenia zawierającego wiele nawiasów

```
int i,j,k,l;

i = 100;

j = 25;

k = 13;

l = (5 * ((j % k) + i) / (2 * k)) / 2;
```

Teraz spróbuj ustalić wartość zmiennej `l`.

Liczby zmiennoprzecinkowe

Wprowadzenie

Poznaliśmy dotychczas dwa typy danych liczbowych w języku C++:

`int` – liczby 32 bitowe ze znakiem o zakresie około ± 2 mld.
`unsigned` – liczby 32 bitowe bez znaku o zakresie od 0 do około 4 mld.

Oba powyższe typy pozwalają stosować jedynie liczby całkowite. Tymczasem w obliczeniach naukowych musimy operować wartościami niecałkowitymi.

Dlatego język C++ został wyposażony w nowy typ danych – liczby rzeczywiste.

Zanim do nich przejdziemy, zapoznajmy się ze sposobami zapisu liczb rzeczywistych w komputerze. Odpowiedzmy sobie na proste pytanie – czy wykorzystując tylko liczby całkowite, można zapisywać również liczby ułamkowe?

Na fizyce zapewne spotkałeś się wielokrotnie z zapisem bardzo dużych lub bardzo małych wartości.

Robiono to na przykład tak:

$2{,}5 \times 10^{36}$

lub tak:

$3{,}3 \times 10^{-31}$

Ogólnie możemy to zapisać następująco:

$W = m \times p^c$

gdzie:

- W – wartość liczby
- m – mantysa
- p – podstawa (w systemie dziesiętnym – 10)
- c – cecha (wykładnik potęgi)

Podstawa p jest zwykle stała, w systemie dziesiętnym wynosi **10**. Zatem nie musimy jej zapamiętywać – po prostu wiemy, że wynosi **10**.

Pozostają do zapamiętania dwie pozostałe liczby: m – **mantysa** i c – **cecha**.

Znając je możemy obliczyć wartość liczby **W**. Zwróć uwagę, że daną wartość W otrzymasz dla wielu zestawów m i c.

Przykłady:

$c = 1$
$m = 3$
$W = 3 \times 10^1 = 3 \times 10 = 30$

$c = 0$
$m = 30$
$W = 30 \times 10^0 = 30 \times 1 = 30$

$c = 2$
$m = 0,3$
$W = 0,3 \times 10^2 = 0,3 \times 100 = 30$

Jeżeli naszą liczbę zapiszemy w postaci pary liczb (c,m), to następujące pary odpowiadają tej samej wartości W: **(1,3) (0,30) (2, 0,3).**

Cecha jest zawsze liczbą całkowitą.

Mantysa jest liczbą ułamkową.

Aby standaryzować zapis, możemy się umówić, że mantysa jest **ZAWSZE** ułamkiem właściwym o mianowniku np. 1000. Skoro tak, to licznik tego ułamka jest liczbą całkowitą od -999 do 999. Skoro mianownik jest ustalony, to nie musimy go zapamiętywać – wystarczy nam licznik, aby odtworzyć wartość mantysy.

Przykład:

Niech m oznacza u nas licznik mantysy, mianownik mantysy ma stałą wartość **1000**

$c = 2$
$m = 3$
$W = {}^3/_{1000} \times 10^2 = {}^3/_{1000} \times 100 = {}^3/_{10} = 0,3$ **– liczba ułamkowa dla pary (2,3).**

Pomimo iż pamiętamy jedynie liczby całkowite, wartość reprezentowanej przez nie liczby jest ułamkowa.

Zatem liczby rzeczywiste możemy zapamiętywać w postaci pary dwóch liczb całkowitych

- cechy
- licznika mantysy.

Tak właśnie komputer zapamiętuje liczby rzeczywiste – w postaci dwóch liczb całkowitych, które wspólnie razem tworzą tzw. kod zmiennoprzecinkowy (ang floating point code).

Komputer pracuje w systemie dwójkowym, zatem mantysa jest ułamkiem binarnym – mianownik jest zawsze potęgą liczby **2** (w systemie praktycznym jest to dosyć duża potęga, np. 2^{53}).

Komputer w kodzie zmiennoprzecinkowym zapamiętuje jedynie licznik tego ułamka.

Podstawa p wynosi **2**. Znając mianownik ułamka mantysy 2^x oraz jego licznik m i cechę c możemy wyliczyć wartość dowolnej liczby zmiennoprzecinkowej:

$$W = m / 2^x \times 2^c$$

m – licznik mantysy, liczba całkowita
2^x – mianownik mantysy, stała wartość, nie pamiętana w kodzie zmiennoprzecinkowym
c – cecha

Przykład:

Niech mianownik ułamka mantysy wynosi 16 (2^4). Zatem licznik może przybierać wartości od -16 do 15. Obliczmy wartości kilku binarnych liczb zmiennoprzecinkowych:

$$c = 2$$
$$m = 1$$
$$W = 1 / 2^4 \times 2^2 = 1 / 2^2 = {}^1/_4 = 0{,}25$$

$$c = -3$$
$$m = 5$$
$$W = 5 / 2^4 \times 2^{-3} = 5 / 2^4 \times 1 / 2^3 = 5 / 2^7 = {}^5/_{128}$$

$$c = 1$$
$$m = 12$$
$$W = 12 / 2^4 \times 2^1 = 12 / 2^3 = {}^{12}/_8 = 1\ {}^4/_8 = 1\ {}^1/_2 = 1{,}5$$

Widzimy, że obliczenia nie są skomplikowane. Wyznaczenie licznika mantysy *m* i cechy *c* jest równie proste. Wykorzystujemy tu proste przekształcenia matematyczne w celu sprowadzenia mantysy do ułamka właściwego. Obliczmy przykładowo *m* i *c* dla liczby 2,5.

Najpierw zapisujemy liczbę 2,5 w postaci ułamka o mianowniku 16:

$$2{,}5 = {}^5/_2 = {}^{40}/_{16}$$

Teraz zapisujemy mantysę i cechę wstępną równą 0:

$$2{,}5 = {}^{40}/_{16} \times 2^0$$

Mantysę musimy sprowadzić do ułamka właściwego. Zatem licznik dzielimy przez 2, a do cechy dodajemy 1. Operację tę kontynuujemy do momentu, aż mantysa stanie się ułamkiem właściwym:

$${}^{40}/_{16} \times 2^0 = {}^{20}/_{16} \times 2^1 = {}^{10}/_{16} \times 2^2.$$

Dostaliśmy:

$$c = 2 \text{ i } m = 10.$$

Pewnych liczb nie da się dokładnie przedstawić w tym systemie. Spróbujmy zapisać liczbę 17 przy założeniu, że mantysa jest ułamkiem właściwym o mianowniku 16.

$$17 = {}^{272}/_{16} \times 2^0 = {}^{136}/_{16} \times 2^1 = {}^{68}/_{16} \times 2^2 = {}^{34}/_{16} \times 2^3 = {}^{17}/_{16} \times 2^4 = {}^{8}/_{16} \times 2^5$$

W ostatnim działaniu przyjęliśmy 8 jako wynik dzielenia 17 przez 2, ponieważ licznik ułamka musi być liczbą całkowitą. To zaokrąglenie spowodowało, że pierwotna liczba 17 zostaje sprowadzona do liczby:

$$c = 5$$
$$m = 8$$
$$W = 8/2^4 \times 2^5 = 8 \times 2^1 = 8 \times 2 = 16!$$

Wniosek – jeśli mantysa jest ułamkiem o mianowniku 16, to liczby 17 nie da się przedstawić dokładnie. Gdyby mantysa była ułamkiem o mianowniku większym, np. 32, problem ten nie wystąpiłby dla liczby 17, ale pojawiłby się z kolei dla liczby 33. Zatem jest to stała cecha tego zapisu.

W rzeczywistym systemie mantysa jest ułamkiem o mianowniku bardzo dużym, np. 2^{56}. Pozwala to zapisywać liczby z dużą dokładnością. Ale błędy zaokrągleń dają czasami znać o sobie, co zobaczymy w dalszych przykładach.

Zmienne typu rzeczywistego

W języku C++ stosowane są trzy typy danych rzeczywistych (zmiennoprzecinkowych). Różnią się one tzw. precyzją, czyli dokładnością zapisu liczb. Wiąże się to z ilością bitów, które w kodzie zmiennoprzecinkowym są przeznaczone na zapis licznika mantysy. Im więcej bitów, tym ułamek mantysy może dokładniej reprezentować w zapisie zmiennoprzecinkowym daną wartość.

Typy zmiennoprzecinkowe:

float – dane zmiennoprzecinkowe 32 bitowe, pojedynczej precyzji. Dokładność 7-8 cyfr znaczących.

double – dane zmiennoprzecinkowe 64 bitowe podwójnej precyzji. Dokładność 15 cyfr znaczących.

long double – dane zmiennoprzecinkowe 80 bitowe o rozszerzonej precyzji. Dokładność 20 cyfr znaczących.

Typ **float** jest najmniej dokładnym typem danych rzeczywistych. Jedyną jego zaletą jest mały rozmiar – 32 bity. Dzisiaj nie zaleca się jego stosowania.

Typ **double** jest standardowym typem rzeczywistym. Jeśli nie będą istniały specjalne powody, to będziemy stosować w programach tylko typ double.

Typ **long double** jest typem danych, które wewnętrznie wykorzystuje koprocesor arytmetyczny – jest to część procesora Pentium, która wykonuje operacje zmiennoprzecinkowe. Typ ten pozwala zminimalizować błędy zaokrągleń i zachować dużą precyzję obliczeń. Jednakże nie będziemy z niego korzystać, ponieważ może nie być dostępny na innych platformach sprzętowych – koprocesory mają różne standardy w różnych systemach.

Przykład:

```
// Obliczanie pola i obwodu prostokąta
//-------------------------------------

#include <iostream>

using namespace std;

int main()
{
    double a,b,pole,obwod;

    cout << "a = "; cin >> a;
    cout << "b = "; cin >> b;

    pole  = a * b;
    obwod = 2 * (a + b);
```

```
    cout << endl
         << "Obwód = " << obwod << endl
         << "Pole  = " << pole << endl;

    return 0;}
```

Powyższy program, chociaż działa doskonale posiada kilka wad z punktu widzenia użytkownika. Uruchom go i wprowadź poniższe dane:

```
a    =    2.3
b    =   4.12

Obwod = 12.84
Pole  = 9.476
```

Zwróć uwagę, że wynik nie jest odpowiednio wyrównany. Kropki dziesiętne znajdują się w różnych kolumnach. Liczba miejsc po przecinku jest różna. A teraz wpisz takie dane:

```
a     =     33212356
b     =     98777878

Obwod  =  2.6398e+008
Pole   = 3.28065e+015
```

Ponieważ wynik jest dużą liczbą, to komputer przedstawia go w postaci naukowej

$$\text{Obwód} = 2{,}6398 \times 10^{8}$$
$$\text{Pole} = 3.28065 \times 10^{15}$$

Manipulatory

Aby mieć pełną kontrolę nad sposobem prezentacji liczb zmiennoprzecinkowych przez konsolę, musimy użyć tzw. manipulatorów strumienia. W tym celu należy do programu dołączyć plik nagłówkowy **`iomanip`**, który zawiera definicję tych manipulatorów. Manipulatory przesyłamy do strumienia jak zwykłe dane. W poniższej tabelce zebraliśmy podstawowe manipulatory strumienia **`cout`**.

Manipulator	Opis
`endl`	Przenosi wydruk na początek nowego wiersza.
`setw(n)`	Ustawia szerokość wydruku liczby. Jeśli liczba posiada mniej cyfr niż wynosi n, to reszta pola jest wypełniana spacjami. Manipulator **`setw(n)`** działa jedynie na następną liczbę przesłaną do strumienia. Kolejne dane nie będą nim już objęte. Cyfry liczby standardowo dosuwane są do prawej krawędzi pola wydruku. **`cout << setw(6) << a << endl;`**
`left`	Umieszcza cyfry liczby po lewej stronie pola wydruku. Stosuje się tylko po manipulatorze **`setw(n)`**. **`cout << setw(6) << left << a << endl;`**
`right`	Umieszcza cyfry liczby po lewej stronie pola wydruku. Stosuje się tylko po manipulatorze **`setw(n)`**. **`cout << setw(6) << left << a << endl;`**

setfill(*ch*)	Manipulator używany tylko po **setw(n)**. Ustawia on znak, którym zostanie wypełnione puste miejsce w polu wydruku liczby – jeśli liczba posiada mniej cyfr niż wynosi szerokość pola, to puste miejsca są zwykle wypełniane spacjami. Manipulator **setfill()** pozwala zmienić spacje na inny znak. Poniższy przykład formatuje wydruk liczb całkowitych na 6 cyfr z wiodącymi zerami, np. zamiast 173 otrzymamy 000173: **cout << setw(6) << setfill('0') << a << endl;**
setprecision(*n*)	Manipulator ustawia liczbę cyfr po przecinku przy wydruku liczb zmiennoprzecinkowych. Stosuje się do wszystkich liczb zmiennoprzecinkowych, które po manipulatorze trafią do strumienia wyjściowego.
fixed	Manipulator powoduje, iż kolejne liczby zmiennoprzecinkowe będą wyświetlane ze stałą liczbą cyfr po przecinku. Liczbę cyfr ustala manipulator **setprecision().** Jeśli nie był wcześniej zastosowany, to standardowo otrzymamy 6 cyfr po przecinku: **cout << fixed << x << endl;**
scientific	Po zastosowaniu tego manipulatora liczby zmiennoprzecinkowe będą wyświetlane w postaci naukowej: **1.56E-2 odpowiada liczbie $1.56 \times 10^{-2} = 1,56 \times 0,01 = 0,0156$**

Nasz program po zastosowaniu manipulatorów wygląda następująco:

```
// Obliczanie pola i obwodu prostokąta
//------------------------

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    double a,b,pole,obwod;

// ustawiamy stałoprzecinkowe wyświetlanie liczb rzeczywistych
// z czterema cyframi po przecinku

    cout << fixed << setprecision(4);

    cout << "a = "; cin >> a;
    cout << "b = "; cin >> b;

    pole  = a * b;
    obwod = 2 * (a + b);

    cout << endl
         << "Obwód = " << setw(12) << obwod << endl
         << "Pole  = " << setw(12) << pole << endl;

    return 0;
}
```

Liczby zmiennoprzecinkowe są liczbami przybliżonymi. Typ **double** pozwala reprezentować dokładnie tylko 15 cyfr znaczących. Jeśli liczba ma ich więcej, to tylko 15 pierwszych cyfr będzie dokładne. Pozostałe już nie.

```
// Precyzja liczby zmiennoprzecinkowej
//--------------------------------------

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    double x;

    cout << fixed;

    x = 12345678901234567890.0; // x ma dwadzieścia cyfr

    cout << x << endl;

    return 0;
}
```

```
// Precyzja liczby zmiennoprzecinkowej
//--------------------------------------

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    double x;

    cout << fixed << setprecision(20);

    x = 0.12345678901234567890; // x ma dwadzieścia cyfr

    cout << x << endl;

    return 0;
}
```

Niektóre liczby nie będą nigdy reprezentowane dokładnie, chociaż posiadają mniej niż 15 cyfr znaczących. Typowym przykładem jest ułamek 0,1. Ponieważ mantysa liczby zmiennoprzecinkowej jest ułamkiem dwójkowym (mianownik tego ułamka jest potęgą liczby 2), to wartości 0,1 nie da się nigdy przedstawić dokładnie, zawsze będzie istniał pewien błąd.

Poniższe ułamki dwójkowe są prawie równe 0,1. Ale "prawie" nie oznacza wcale, że są równe:

$$^{1}/_{8},\quad ^{1}/_{16},\quad ^{3}/_{32},\quad ^{6}/_{64},\quad ^{12}/_{128},\quad ^{25}/_{256},\quad ^{102}/_{1024},\quad ^{6553}/_{65535}\ \ldots$$

Ułamek dziesiętny 0,1 posiada w systemie dwójkowym nieskończone rozwinięcie.

W naszym systemie dziesiętnym podobną własność mają ułamki $^{1}/_{3}$, $^{1}/_{6}$, $^{1}/_{7}$, $^{1}/_{9}$ – ułamków tych nie da się przedstawić dokładnie za pomocą skończonej ilości cyfr w systemie dziesiętnym. Tak samo w systemie dwójkowym, ułamka $^{1}/_{10}$ nie da się przedstawić dokładnie za pomocą mantysy o skończonej liczbie bitów. Konsekwencje tego faktu prezentuje poniższy prosty program:

```
// Niedokładny ułamek
//-----------------------

#include <iostream>

using namespace std;

int main()
{
    double x;
    x = 0.1;

    x += 0.1; // 0,2
    x += 0.1; // 0,3
    x += 0.1; // 0,4
    x += 0.1; // 0,5
    x += 0.1; // 0,6
    x += 0.1; // 0,7
    x += 0.1; // 0,8
    x += 0.1; // 0,9
    x += 0.1; // 1

    cout << "x = " << x << endl;

    if(x == 1) cout << "Dobrze!";
    else       cout << "Źle!!!";

    cout << endl;

    return 0;
}
```

Program dodaje do zmiennej x ułamek 0,1. Po wykonaniu 9 takich dodawań x powinno osiągnąć wartość 1 i na ekranie powinien pojawić się tekst Dobrze!.

Tymczasem po uruchomieniu programu pojawia się ten drugi tekst, pomimo że program wyświetla wartość x jako 1. Powodem jest to, że po wykonaniu 9 dodawań 0,1 w zmiennej x nie była dokładna wartość 1, tylko wartość bardzo zbliżona do 1.

Niestety, operator == traktuje ją jako różną od 1.

Z programu powyższego wynika BARDZO WAŻNY wniosek – liczb zmiennoprzecinkowych NIE WOLNO przyrównywać do wartości dokładnych. Zamiast sprawdzania, czy dwie liczby zmiennoprzecinkowe a i b są równe:

`a == b`

powinniśmy zbadać ich różnicę. Jeśli ta różnica jest dostatecznie mała, to przyjmiemy, że liczby a i b są równe. Różnica może być dodatnia lub ujemna. Aby nie rozważać zatem dwóch różnych przypadków, będziemy badać wartość bezwzględną różnicy:

`| a - b | < wartość graniczna`

Wartość bezwzględna liczby zmiennoprzecinkowej oblicza funkcja **`fabs(x)`**. Dostęp do tej funkcji uzyskamy po dołączeniu do programu pliku nagłówkowego **`cmath`**. Za wartość graniczną

przyjmiemy **0.0000001**. W tym celu w programie zdefiniujemy stałą **EPS** o takiej właśnie wartości. W pętli **while** mamy warunek **x** różne od **1**. Otrzymamy go następująco:

```
fabs(x - 1) > EPS
```

A oto zmodyfikowany program, który teraz działa wg oczekiwań:

```
// Niedokładny ułamek
// ulepszony
//------------------------

#include <iostream>
#include <cmath>

using namespace std;

const double EPS = 0.0000001;

int main()
{
    double x;

    x = 0.1;

    x += 0.1; // 0,2
    x += 0.1; // 0,3
    x += 0.1; // 0,4
    x += 0.1; // 0,5
    x += 0.1; // 0,6
    x += 0.1; // 0,7
    x += 0.1; // 0,8
    x += 0.1; // 0,9
    x += 0.1; // 1

    cout << "x = " << x << endl;

    if(fabs(x - 1) < EPS) cout << "Dobrze!";
    else                  cout << "Źle!!!";

    cout << endl;

    return 0;
}
```

Na liczby zmiennoprzecinkowe trzeba bardzo uważać w programowaniu. Musimy pamiętać, że są to wartości przybliżone. Błąd w stosunku do wartości dokładnej jest zwykle bardzo mały, ale może powodować błędne działanie programu, jeśli nie weźmiemy tego faktu pod uwagę.

Zadanie:

Napisz program znajdujący pierwiastki równania kwadratowego:

$$ax^2 + bx + c = 0$$

Algorytm znajdowania pierwiastków jest następujący:

Obliczamy wyróżnik równania:

$$\Delta = b^2 - 4ac$$

W zależności od wartości wyróżnika mamy trzy przypadki:

1. $\Delta = 0$

Istnieje pierwiastek podwójny, który obliczamy ze wzoru:

$$x_1 = x = -\frac{b}{2a}$$

2. $\Delta > 0$

Istnieją dwa różne pierwiastki, które obliczamy ze wzorów:

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \qquad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

3. $\Delta < 0$

Nie ma pierwiastków rzeczywistych.

Algorytm wymaga obliczania pierwiastka kwadratowego, który udostępni nam funkcja sqrt(x), dostępna po dołączeniu pliku nagłówkowego **cmath**.

```
// Równanie kwadratowe
//------------------------

#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

const double EPS = 0.0000001;

int main()
{
  double a,b,c,delta,x1,x2;

  cout << fixed << setprecision(4);

  cout << "Rozwiązywanie równania kwadratowego\n"
          "-----------------------------------\n\n"
```

```
          "    2\n"
          "  a(x*x) + bx + c = 0\n\n";

  cout << "  a = "; cin >> a;
  cout << "  b = "; cin >> b;
  cout << "  c = "; cin >> c;

  cout << endl;

  delta = b*b - 4*a*c;

  if(fabs(delta) < EPS)
  {
    cout << "  Jeden pierwiastek podwójny:\n\n";

    x1 = -b / 2 / a;

    cout << "  x = " << x1 << endl;
  }
  else if(delta > 0)
  {
    cout << "  Dwa różne pierwiastki:\n\n";

    x1 = (-b - sqrt(delta)) / 2 / a;
    x2 = (-b + sqrt(delta)) / 2 / a;

    cout << "  x1 = " << setw(14) << x1 << endl
         << "  x2 = " << setw(14) << x2 << endl << endl;
  }
  else cout << "  Brak pierwiastków rzeczywistych\n\n";

  return 0;
}
```

Ćwiczenie

Program sprawdź dla trzech poniższych równań kwadratowych:

$x^2 - 2x + 1 = 0$, pierwiastek podwójny $x_1 = x_2 = 1$
$x^2 - 3x + 2 = 0$, dwa pierwiastki różne: $x_1 = 1$, $x_2 = 2$
$x^2 + x + 1 = 0$, brak pierwiastków.