C++

Krzysztof Woźniak, IFJ PAN

Operacje podstawowe

Operacje arytmetyczne:

```
a = b;              // podstawienie
a = b = c = 0;      // podstawienie ma także swoją wartość: a = (b = (c = 0));

x=a+b*c-d/e;        // podstawowe działania arytmetyczne + - * /
n += 5;             // to samo co n = n + 5;
                    // podobnie dla  -,  *,  /

n++;                // to samo co:   n = n + 1;
++n;                // w zasadzie to samo co n++:   n = n + 1;
    n = 2;
    k = n++;            // dodawanie 1 dokonywane jest PO podstawieniu:
                        // k ma wartość 2
    k = ++n;            // dodawanie 1 PRZED podstawieniem:
                        // k ma wartość 3

m = n % k;          // reszta z dzielenia

potęgowanie?        // brak operatora dla x^2,  x^y
                    // są funkcje biblioteczne: pow(x, y)   TMath:Power(x, y)
```

Operacje podstawowe

Porównanie wartości:

```
>,  <,  >=,  <=      // większe, mniejsze
a == b               // równe
a != b               // różne
```

Operacje logiczne:

```
stała                // 0 - fałsz, różne od 0 - prawda
a &&  b              // koniunkcja logiczna (a  i  b)
a || b               // alternatywa logiczna (a   lub  b)
!a                   // negacja logiczna
```

Operacje na bitach:

```
a & b                // koniunkcja AND
a | b                // alternatywa OR
a ^ b                // XOR (0 - gdy bity równe, 1 - gdy bity różne)
~a                   // negacja
k << n               // przesunięcie k o n bitów w LEWO (mnożenie przez 2 do potęgi  n)
k >> n               // przesunięcie k o n bitów w PRAWO (dzielenie przez 2 do potęgi  n)
```

Typy zmiennych prostych

Typy podstawowe

bool - zmienna logiczna (0- fałsz, 1-prawda)
char - pojedynczy znak
int - liczba całkowita ze znakiem (+- 2 100 000 000)
dodatkowo: short, long, longlong, ushort, uint, ulong
float - liczba zmiennoprzecinkowa (do 3* 10 do 38, 7 cyfr)
double - dokładniejsza liczba zmiennoprzecinkowa (10 do 308, 15 cyfr)

Tablice - numerowane od zera

```
char linia [250];
int x[5] = { 2, 32, 1, -3, 7 };
```

Wskaźniki

```
int *ptr;
ptr = x;              // wskaźnik do początku tablicy x
ptr = &linia[3];      // & oznacza tutaj adres, ptr wskazuje na 4 z kolei
                      // element tablicy
int i = *ptr;         // * oznacza sięgnięcie do pamięci wskazywanej przez ptr
                      // po tym podstawieniu i ma wartość  -3
```

Instrukcje złożone

Instrukcje warunkowe:

if(*warunek*) *instrukcja*;
if(*warunek*) *instrukcja1*; **else** *instrukcja2;*
if*(warunek) { instrukcja1a; instrukcja1b; ...}* ***else*** *{ instrukcja2a; instrukcja2b; ... }*

```
switch( zmienna )
{
      case wartość_A:
            instrukcjaA_1; instrukcjaA_2;  // ....
            break;
      case wartość_B:
            instrukcjaB_1; instrukcjaB_2;  // ....
            break;
        //  podobnie  dla  C, D, ....

      default:   // czyli wszystkie nie wymienione wartości
            instrukcjaDEF_1; instrukcjaDEF_2;  // ....
            break;
}

a = (b>c)  ?  b  :  c;  // podstawienie warunkowe
```

Instrukcje złożone

Instrukcje powtórzeniowe:

for(*instrukcja inicjująca; warunek zakończenia; instrukcja końca pętli*) {
    instrukcje pętli;
}

```
suma=0;
for(i=1; i<=10; i++) suma += i*i;  // suma kwadratów liczb od 1 do 10

  // każdy z elementów for może być pominięty
for(;;)  {  instrukcje  }          // warunek pusty jest zawsze fałszywy
                                   // dla przerwania tej nieskończonej pętli konieczne jest
                                   // użycie break w bloku instrukcji (lub return w funkcji)
```

while (*warunek*) *instrukcja*;

do {
  instrukcje pętli;
} **while** (*warunek*);

Proste instrukcje wejścia - wyjścia

Instrukcje wyjściowe:

std::cout << ”wartośc a = ” << a << **endl;**

printf(*format, zmienna1, zmienna2, ..., zmienna11*); // maksimum 12 parametrów
format - przemieszany tekst ze znacznikami określającymi sposób wyprowadzania zmiennych:

%5d	- liczby całkowite
%9.3f	- liczby zmiennoprzecinkowe
%c	- pojedynczy znak
%50s	- ciągi znaków
\n	- oznacza znak końca linii

fprintf(plik wyjściowy, *format, zmienna1, zmienna2, ..., zmienna10*);

sprintf(*wskaźnik do ciągu znaków, format, zmienna1, zmienna2, ..., zmienna10*);

plik wyjściowy jest deklarowany jako:
FILE *plik = fopen("nazwa_pliku.txt”, "w"); // plik do zapisywania

Proste instrukcje wejścia - wyjścia

Instrukcje wejściowe:

std::cin >> a**;**

scanf(*format, &zmienna1, &zmienna2, ..., &zmienna11*);
format - przemieszany tekst ze znacznikami określającymi sposób wyprowadzania zmiennych, zwykle bez rozmiaru pola

fscanf(*plik wejściowy, format, &zmienna1, &zmienna2, ..., &zmienna10*);

char linia[250];
fgets(linia, 249, *plik wejściowy*);
sscanf(linia, *format*, *&zmienna1, &zmienna2, ..., &zmienna10*);

plik wejściowy jest deklarowany jako:
FILE *plik = fopen(”nazwa_pliku.txt”, ”r”); // plik do odczytywania

Funkcje

Deklaracja

```
void nazwa();                    // funkcja bez parametrów i bez wyniku
typ_wyniku  nazwa();             // funkcja bez parametrów, dajaca wynik
typ_wyniku  nazwa(typ par1, typ par2, ..., typ parN);
                                 // funkcja z N parametrami, z wynikiem
```

Definicja

```
void nazwa()
{
  cout << ”komunikat” << endl;
}

typ_wyniku  nazwa()
{
   typ_wyniku x;
   x = jakieś_obliczenia;
   return x;                  // konieczne
}
```

Modyfikacja parametrów

```
void fun1(int k)              // parametr przekazywany przez wartość - k bez zmian
{  k = k+1;  }

void fun2(int *k);            // parametr wskaźnikiem do zmiennej - k może być zmienione
{  *k = *k -5;  }

void fun3(int &k);            // parametr przekazywany przez adres - k może być zmienione
{  k = k -5;  }
```

Wywołania:

```
int m = 5;
int it[4] = {1, 2, 3, 4 };
fun1(6);   fun1(m);   fun1(m/2);  fun1(it[1]-it[2]);    // prawidłowe wywołania

fun2(&m);   fun2(it);  fun2(&it[2]);          // prawidłowe wywołania
            fun2(m+2);   fun2(it[1]);         // błędne wywołania

fun3(m);   fun3(*it);  fun3(it[2]);           // prawidłowe wywołania
            fun3(m+2);                        // błędne wywołanie
```

Biblioteki funkcji standardowych

#include <nazwa>
#include ”plik”

<iostream>
cin, cout, cerr, clog

<cstdio> (stdio.h)

fopen,, fclose	- operacje otwarcia i zamkniecia pliku
getchar, getc, gets	- wczytywanie znaku lub ciągu znaków
fgetc, fgets	- wczytywanie znaku lub ciagu znaków z pliku
scanf, fscanf, sscanf	- wczytywanie danych (wg. formatu)
printf, fprintf, sprintf	- wypisywanie (wg. formatu)

<cstring> (string.h)

strcpy, strncpy	- kopiowanie ciągu znaków
strcmp, strncmp	- porównywanie ciągu znaków
srtchr, strrchr	- wyszukiwanie znaku w ciągu znaków
strstr	- wyszukiwanie ciągu znaków w drugim
strpbrk	- wyszukiwanie jednego z podanych znaków w ciągu znaków
strlen	- długość ciągu znaków

<cmath> (math.h)
podstawowe funkcje arytmetyczne (trygonometryczne, logarytmy, pow(x, y), itd.)

vector

Obiekt do przechowywania ciągów liczb (głównie) o zmiennej długości

deklaracja

```
std::vector<int> itab;
std::vector<vector<float>> *ftab2;
```

funkcje:

```
begin()          -iterator ustawiony na pierwszym elemencie
                 std::vector<int>::iterator it = itab.begin();
                 n = *it;  // element listy danych
                 // zmiana elementu wskazywanego przez it++;
end()            - iterator ustawiony na ostatnim elemencie
empty()          - 1 gdy wektor jest pusty
size()           - aktualny rozmiar wektora
front()          - wartość pierwszego elementu
back()           - wartość ostatniego elementu
at(n)            - wartość elementu n (liczenie od 0)
push_back(x)     - dodanie elementu x na końcu wektora
pop_back()       -  usunięcie ostatniego elementu z wektora
```

string

Obiekt do przechowywania ciągów znaków o zmiennej długości

deklaracja

```
std::string ctab;
```

funkcje:

```
data()              - zwraca wskaźnik do ciągu znaków (typu  char *),
                      np.     cout << ctab.data() << endl;   // wypisanie znaków z ctab
begin()             - iterator ustawiony na pierwszym elemencie
          std::string::iterator it = ctab.begin();
          char c = *it;  // element listy danych
          // zmiana elementu wskazywanego przez it++;
end()               - iterator ustawiony na ostatnim znaku
empty()             - 1 gdy ciąg znaków jest pusty
length()            - aktualna liczba znaków
front()   back()    - pierwszy znak,  ostatni znak
at(n)               - element n (liczony od 0)
push_back(c)        - dodanie znaku c na końcu (to samo przez ctab += c;)
append(parametry) - dodawanie znaków - na kilka dostępnych sposobów (za czymś)
insert(parametry) - dodawanie znaków - na kilka dostępnych sposobów (przed czymś)
clear()             - zmiana w pusty ciąg znaków
find(parametry)     - wyszukiwanie podciągu znaków
```

Klasy

Zestaw danych + funkcje operujące na nich:

```
class   A       // zliczanie do 20
{
public:
    A()       {    m_x = 0;      }          // konstruktor1
    A(int i)   {   m_x = i;   }             // konstruktor 2
    ~A()     {  }                           // destruktor
    incr()   {    m_x++;    }               // funkcja versia 1
    incr(int i)   {  m_x += i;   }          // funkcja wersja 2
    int count()     {   if(m_x > 20)  error();    return m_x;  }
private:
    error()  {   printf('error\n");         // funkcja prywatne, niedostępna z zewnątrz
    int m_x;                                // ukryta zmienna
};
```

Wykorzystanie

```
A   a;
A *b = new A(4);
a.incr();                   // odwołanie do elementu klasy zadeklarowanej statycznie
b->incr(2);                 // odwołanie do elementu klasy której wskaźnik znamy
```

Można tworzyć **hierarchię klas**, klasy wyższego rzędu **dziedziczą** własności klasy podstawowej

Klasy - dziedziczenie

Funkcje w klasach wyższego rzędu mogą być ”podmieniane”

```
class   A
{     ...
   int funa0();              // funkcja zdefiniowana tylko w klasie A, używana w klasie AA
   int funa1();              // funkcja wybierana na podstawie typu zmiennej wskaźnikowej
   virtual int funa2();      // funkcja wybierana na podstawie typu obiektu (przy wywołaniu)
      ...
};
class   AA:  public A
{     ...
   int funa1();        // inna wersja  funa1
   int funa2();        // inna wersja  funa2
   int funa3();        // dodatkowa funkcja
      ...
};

A *p1;
AA *p2 = new AA();
p1 = p2;                    // p1 i p2 wskazują na ten sam obiekt
p1->funa1()    oraz    p2->funa1()    oznaczają wywołanie różnych funkcji (z A i AA)
p1->funa2()    oraz    p2->funa2()    oznaczają wywołanie tej samej funkcji (z AA) -
                                      zapewnia to deklaracja virtual w A
```

Zmiana typu

Konwersja typów:

```
float x = 2.1;
int i;
i = (int) x;   // prawidłowe podstawienie
```

Dla klas:

```
A *p1;
AA *p2 = new AA();
p1 = p2;            // dopuszczalne: obiekt typu AA ma też pełne własności obiektu A
AA *p3;
p3 = p1;            // BŁĄD!  typ p1 nie zgadza się z typem p3
p3 = (AA *) p1;     // prawidłowe: przed podstawieniem jest sprawdzone, że w p1
                    // znajduje się odnośnik do obiektu typu AA
```

Dodatkowe informacje:
http://www.learncpp.com/
http://www.cplusplus.com/