Променливи. Инициализация на променливи.

По време на изпълнението на дадена програма е необходимо да оперира с някакви стойности.

Стойностите с които оперира програмата се наричат променливи.

Променливите служат като контейнери за данни от някакъв характер целочислен, булев, символен или друг.

Променливата физически представлява една или повече клетки в оперативната памет на системата. Информацията във всяка клетка е достъпна от нейният адрес. За да се достигне до нея трябва да се знае адреса в паметта, но този начин за достъп е много трудоемък и неясен. Това е довело до кръщаването на променливите с дадени имена носещи повече полезна информация за съхраняваната в тях информация. Променливите могат да съдържат информация, която се съхранява само и единствено по време на изпълнението на програма, след това цялата информация бива директно изтривана автоматично. Друго нещо, което е много важно да запомните е че променливите могат да съдържат различни типове информация като цели числа, реални числа символи.

Деклариране на променливите

Променливите имат три характеристики: тип, име и стойност. Преди да бъдат използвани, променливите трябва да бъдат дефинирани. Общият вид на дефиницията на една променлива е следният:

<тип> <име на променлива> [= < стойност>];

Задължителните елементи на дефиницията са типа и името на променливата. Дефинирането променливите може да бъде съпроводено с инициализация, т.е. задаване на начална стойност. За тази цел се използва знака за присвояване. Ето някои примери за дефиниции на променливи:

```
int    x;         // Дефиниция на променлива x от тип int
int    y, z = 2; // Дефиниция y и z и инициализация на z
float pi = 3.14; // Дефиниция и инициализация на pi
double d;        // Дефиниция на променлива x от тип double
```

Чрез тези дефиниции се създават променливите **x**, **y** и **z** от тип **int**, променливата **pi** от тип **float** и променливата **d** от тип **double**. Променливите z и pi са инициализирани с начални стойности 2 и 3.14. Променливите могат да участват в аритметични изрази, а техните стойности могат да се променят чрез оператора за присвояване.

Както сами можете да се досетите в една програма, едва ли ще има само една единствена променлива. Ето заради това езикът C++

ни предоставя две различни възможности за декларация на променливи. Когато две или повече променливи са от един и същи тип, те могат да бъдат записани на един ред като всяко се отделя със запетая. Ето и как точно става това:

```
int nRow,nCol,i,j;
```

Разбира се всички тези променливи могат да се декларират по обикновения начин, всяка на нов ред:

```
int nRow;
int nCol;
int i;
int j;
```

Наистина е много важно да запомните, че само и единствено променливи, които са от един и същи тип могат да се декларират на едни ред, както демонстрирах в първия пример. Във всеки останал случай се използва вторият метод.

Внимание: C++ е чувствителен език спрямо главните и малките букви. "NCOL" – написано по този начин, името на променливата не е еквивалентно на "ncol". Това се смятат за две напълно различни променливи. Същото правило важи и за променливите включващи главна и малки букви "nCol"!

Инициализация на променливите

C/C++ позволява по време на деклариране на променливите, те да се инициализират с желана стойност. Форматът е:

<Тип> <ИмеНаПроменлива> = <Стойност>;

При инициализиране на елементи, изискващи повече от една стойност (масиви, структури), поредицата се огражда във фигурни скоби,а стойностите вътре се отделят чрез запетаи.

Пример:

```
int x = 1, y = 2;       // Дефиниране на целочислени променливи
char name[] = "Камен"; // Дефиниране на стринг
char answer = 'У';      // Дефиниране на символ
char key = 3;           // Дефиниране на число
char list [2][10] = {"Първи","Втори"};//Дефиниране на масив от стрингове
float Array[3][3] = {{1, 2, 3}, // Дефиниране на двумерен масив
                     {4, 5, 6},
                     {7, 8, 9}};
```

Съхраняване на променливите

Езикът C/C++ дефинира няколко класа на съхранение на променливите (**auto**, **extern**, **static**, **register**). Глобалните променливи (декларирани извън функциите, включително извън main) по подразбиране се приемат за клас **external**, което означава, че те се инициализират със стойност **0** при стартиране на програмата (освен ако програмистът не ги инициализира със друга стойност). Променливите, декларирани във функции (включително в главната main), по подразбиране

се приемат за локални - клас **auto**. Те не се инициализират и загубват стойностите си между две последователни обръщения към функцията (такава е променливата i от следващия пример). Предвидена е възможност да се запази стойността на променлива между отделните обръщения към функцията. За целта се използва клас **static**. Те ще бъдат инициализирани със стойност **0** при стартиране на програмата и ще запазват стойностите си между отделните обръщения към функцията (това се отнася за променливата **count** от следващия пример).

Ето и самият пример:

```
int test(void)
{
 int i;
 static int count;
   ...
}
```

Класове памет

Програмният език C++ поддържа следните класове памет:

auto - автоматична (локални и формални променливи)

static - статична памет

extern - външна памет

register - регистрова памет

В C/C++ всички променливи които са декларирани в дадена функция (локални и формални променливи) са от тип **auto**. Променливите от тип **auto** се съхраняват в стека на програмата. Тази тяхна особеност определя и времето им на съществуване, което съвпада с времето за изпълнение на функцията. За разлика от тях, глобалните променливи (тези които са декларирани извън всички функции) се разполагат в областта за статични данни на програмата наречена **heap**. Тази област се създава при стартирането на програмата и се унищожава при излизане от нея. Това обуславя и времето за съществуване на глобалните променливи и тяхната видимост.

За да се промени времето на съществуване на локалните променливи е необходимо, те да се декларират като статични. Това се осъществява като при декларацията се използва декларацията **static**.

Пример:

```
int MinElem() {
   int i;
   static int minelem;
   for ( i=0 ; i<MaxBr ; i++)
    {
       if ( Array[i] < minelem )
            minelem = Array[i];
    }
   return minelem;
}
```

При първото извикване на функцията **MinElem**. Променливата **minelem** ще има стойност 0. При следващото извикване стойността на променливата ще бъде тази от предишното извикване на функцията. Забележете променливата **minelem** е декларирана като локална променлива, но е разположена в областта на глобалните променливи, но за разлика от тях, тя е видима само в рамките на функцията в която е декларирана.

Глобалните променливи също могат да бъдат декларирани като **static**. При тях действието на оператора **static** е коренно различно. Тук не се променя времето за живот на променливите, а се променя видимостта й. По-точно казано видимостта на променливата от външни модул (капсулиране на променливите). При това положение до променливата не може да се достигне от други модули посредством декларацията **extern**.

Когато програмата, която се разработва е много голяма, е удачно тя да се раздели на модули. След като се програмират и настроят отделните модули те се обединяват в един общ проект. При това част от глобалните променливи ще бъдат използвани от няколко модула. За да може да се осъществи това се използва декларацията **extern**.

Пример:

Първи модул.

```
extern int y;         // имен а променлива декларирана в другия модул.
extern const  start; // имен а константа декларирана в другия модул.
```

Втори модул.

```
int x;              // Деклариране на променлива от цял тип
int y = 0;          // Деклариране и инициализиране на цяла променлива
const  start = 1; // Деклариране и  инициализиране на константа
struct person       // Дефиниране на структура и деклариране на променлива
{                   // от този тип
   long phone;
   char *name;
} customer;
```

В някой ситуации бързодействието на програмата е от особена важност. За подобряване на бързодействието се използват различни методи, като писане на оптимален код по бързодействие или използване на регистрите за съхраняване на някой от променливите. Назначаването на регистър за променлива се извършва, като се декларира съответната променлива като регистрова.

Пример:

```
int Sum()
{
  register int i;
  register int Sum = 0;
  for(i=0;i<100;Sum+=i, i++);
  return Sum;
}
```

При това деклариране на променливите, ако в даденият момент има свободен регистър, той ще бъде назначен от компилатора за съхраняване на променливите **i** и **Sum**.

Дефиниране на потребителски типове

Една удобна езикова конструкция в C++ е дефинирането на потребителски типове. Това се извърша с операцията **typeccdef** по следния начин:

typedef <стар_тип> <нов_тип>

Където **стар_тип** е име на допустим за езика тип на променлива, например **int**, **char**, **double** и др.;

Освен това може да бъде и име на тип дефиниран преди това чрез **typedef**.

нов_тип - ново име, заместващо името, описано в стар_тип;

Новото име е произволен допустим за езика C++ идентификатор. Името определено от **typedef** е еквивалентно на името на името на съществуващия тип.

Например:

```
typedef unsigned short int WORD;
typedef char BYTE;
typedef float EXS[100];
typedef char *STRING;
typedef struct
 {
      unsigned  int F11:1;
      unsigned  int F12:3;
      unsigned  int F13:1;
      unsigned  int F14:3;
      unsigned  int F15:8;
  } FLAGS;
```

При тези заместванията декларацията **WORD i;** е еквивалентна на декларацията **unsigned short int i;.** А декларацията **EXS f;** на **float a[100];** и т.н.

Използуване на typedef

При класическия стил, дефинираните от потребителя типове обикновено не се именуват. Изключение правят типовете **struct** и **union**, но и при тях декларацията трябва да се предшества от ключовата дума **struct** или **union**. При модерния стил, когато се използува директивата **typedef**, за скриване на информацията се използува друго ниво. Това позволява да се свърже определен тип

данни с име (включително и при `struct` и `enum`). След това вече може да се декларират променливи от така дефинирания нов тип. **Например:**

```
typedef int *ptrInt;
typedef char namestr[30];
typedef enum { male, female, unknown } sex;
typedef struct
 {
   namestr last, first;
   char   ssn[9];
   sex    gender;
   short  age;
   float  gpa;
  } TClas;
typedef student clas[100];
TClas  hist104, ps102;
student valedictorian;
ptrInt iptr;
```

Дефинирането на потребителски типове данни прави програмите по-четими. Освен това дава възможност да се направи при нужда лесно предефиниране на група променливи, като се смени само дефиницията `typedef` за тях.

Предимството от използването на новите имена се състои в това, че те са по-кратки като запис и по-разбираеми.

Операторът `typedef` не води до създаване на нови обекти и заделяне на памет, а само до деклариране на нови имена. В този смисъл, декларирането на ново име на даден тип и дефинирането на променлива от даден тип са качествено различни операции, въпреки синтактичното им сходство.

Определяне размера на обекти (оператор SIZEOF)

Операторът **sizeof** има две форми:

sizeof <унарен израз>

sizeof <име_на_тип>

Резултатът от изпълнение на оператора **sizeof** е число от тип **int**, представляващо количеството памет в байтове, необходимо за типа на операнда. Например **sizeof(int)** е 4, а **sizeof(double)** е 8. Когато операндът е израз, резултатът от **sizeof** е количеството памет, необходимо за типа на резултата от изчислението на израза. Но самият израз не се изчислява, т.е. няма странични ефекти. Например, ако x е променлива от тип **int**, то **sizeof(x + 2 * 3)**

ще бъде 4, тъй като резултатът от изчислението на израза ще бъде от тип **int**, който се представя в два байта, а **sizeof**(x + 2.3 - 3) ще бъде 8, тъй като резултатът от израза ще бъде от тип double, който се представя в 8 байта. Операторът sizeof може да се прилага не само върху основните типове, но и върху типове, дефинирани от потребителя.

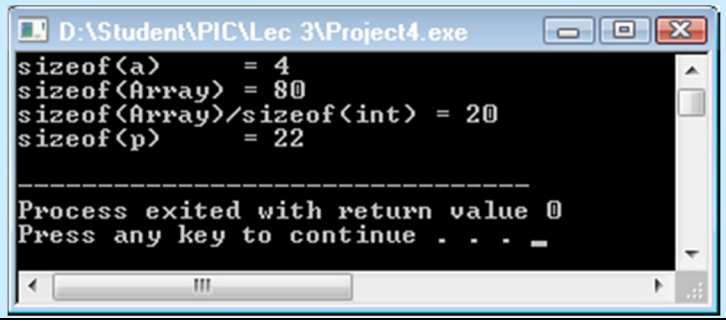
Пример:

Програмен код

```
#include <iostream>
using namespace std;
int main(int argc, char** argv) {
  int a; // Деклариране на цяло число
  int Array[20]; // Деклариране на масив от 20 елемента
  char p[] = "Това е примерен текст"; // дефиниране на стринг
  // Размер на променливата a
  cout << "sizeof(a)      = " << sizeof(a)     << endl;
  // Размер на заеманото място в паметта от масива Array
  cout << "sizeof(Array) = " << sizeof(Array) << endl;
  // Размер (брой елементи) на масива Array
  cout << "sizeof(Array)/sizeof(int) = "
       << sizeof(Array)/sizeof(int)     << endl;
  // Размер на заеманото място в паметта от стринга p
  cout << "sizeof(p)      = " << sizeof(p)     << endl;

  return 0;
}
```

Резултат



На ред 5 е декларирана променливата a. Тя е от ти **int**. размера на тип **int** в 32-битов компилатор е 4-ри байта. При изпълнението на командата **sizeof(**a**)** трябва да бъде равен с размера на типът **int**. На ред 6 е деклариран масивът Array. Под масив се разбира последователност от N елемента от даден тип. В случая типът на масива е **int**. Броят на елементите на масива се задават в квадратните скоби ([]). В нашия пример 20. Заеманото място в паметта от масива се определя от броят на елементите му и неговият тип. В случая 20 * 4 = 80. Това е и резултатът върнат от командата **sizeof(**Array**)**.

А случаите когато желаем да разберете размера на един масив можете да използвате следният програмен код:

sizeof(Array**)/sizeof(int)**

Тук **sizeof(**Array**)** връща размера на масива в байтове, а **sizeof(int)** размера на един елемент.

По-особен е случая с определяне размера на стринга деклариран на ред 7. Текста "Това е примерен текст" съдържа 21 символа, а резултата върнат от командата **sizeof(**p**)** е 22. При положение, че размерът на char е 1 байт, логично е върнатата стойност да бъде 21. Този един байт идва от байтът със стойност 0, който се поставя в края на всеки стринг.