Universitatea Tehnică „Gheorghe Asachi" din Iaşi
Facultatea de Automatică şi Calculatoare

PROGRAMARE II

Curs 2

Particularităţi ale limbajului C++.

Operaţii de intrare – ieşire în C++

- `cin` pentru citirea de la tastatură
- `cout` pentru afişarea pe monitor
- Fişier header `<iostream>`
- `using namespace std;`
- Exemplu `cin`:

```
int c;
char j[40];
cin >> c;
cin >> j;
```

- Exemplu `cout`:

```
cout << j;
cout << "Nr citit este " << c << endl;
```

Tipul referinţă în C++

- Sinonim pentru o altă variabilă

```
int i=8;
int &j = i;
j=5;
cout << i << endl;
```

- Pe ecran se va afişa valoarea 5
- Transmiterea parametrilor funcţiilor prin:
  - Valoare (C, C++)
  - Referinţă (C++)
- O funcţie poate returna o variabilă prin referinţă

Funcție interschimbare a doi întregi

C

```
void Schimba(int* a, int* b)
{
        int aux;
        aux = *a;
        *a = *b;
        *b = aux;
}

  int x=1,y=2;
  Schimba(&x,&y);
```

C++

```
void Schimba1(int & a, int & b)
{
        int aux;
        aux = a;
        a = b;
        b = aux;
}

int x=1,y=2;
Schimba1(x,y);
```

```
Afisare x afisare y
```

Mesajul afișat:

x=2 y=1

Alocarea şi dealocarea dinamică de memorie

- In C++ alocarea și dealocarea dinamică de memorie se face cu ajutorul operatorilor:
  - new
  - delete

Alocarea si dealocarea tablourilor

- Pentru alocare de memorie se folosește operatorul new
- Pentru dealocare se foloseste operatorul delete []

```
int *dv;
dv = new int [10];

delete [] dv;
```

Funcții cu parametri impliciți

- Un parametru implicit este un parametru pentru care s-a specificat o valoare implicită
- Daca utilizatorul nu specifică o valoare pentru acest parametru la apelul functiei, valoarea implicită este utilizată
- Trebuie să se regasească în extrema dreaptă a listei parametrilor din antetul funcției

```
void afis(int a, int b =0)
{
        cout << "a=" << a << " b=" << b <<endl;
}
```

Funcții cu parametri impliciți

- Pentru un apel de forma

```
int a=7,b=2;
afis(a,b);
```

- Se va afișa

```
a=7 b=2
```

- Pentru un apel

```
afis(a)
```

- Se va afișa

```
a=7 b=0
```

Supraîncărcarea funcțiilor

- In C++, două sau mai multe funcții pot avea același nume dacă tipul sau numărul parametrilor este diferit

Supraîncărcarea funcțiilor

```
int operatie (int a, int b)
{
  return (a*b);
}

float operatie (float u, float v)
{
  return (u/v);
}

int main ()
{
  int x=7, y=2;
  float n=3.0f, m=2.0f;
  cout << operatie (x,y) << endl;
  cout << operatie (n,m) << endl ;
  return 0;
}
```

```
14
1.5
```

Supraîncărcarea funcțiilor

- Compilatorul examinează tipurile argumentelor şi decide care dintre cele două funcţii `operatie` să fie apelată
- O funcție nu poate fi supraîncarcată numai prin tipul pe care il returnează
- Dacă nu se găsește o funcție care să se potrivească exact cu tipul argumentelor transmise, compilatorul încearcă să aplice unele conversii
- Toate conversiile standard au aceeași prioritate

Supraîncărcarea funcțiilor

```
int main ()
{
  int x=7, y=2;
  float n=3.0f, m=2.0f;
  cout << operatie (x,y) << endl;
  cout << operatie (n,m) << endl;

  double p = 3.0, q= 2.0;
  cout << operatie (p,q) << endl ;

  return 0;
}
```

```
error C2668: 'operatie' : ambiguous call to overloaded
function
```

Constante

- C++ oferă conceptul de constantă definită de utilizator, `const`, pentru a exprima noțiunea că o valoare nu se schimbă direct
- Utilizări ale constantelor:
  - Variabile care nu își modifică valoarea după inițializare
  - Constantele simbolice permit o întreținere mai ușoară a programului
  - Majoritatea pointerilor sunt folosiți pentru a se citi o valoare(adresă) și nu pentru a fi scrișI
  - Majoritatea parametrilor unei funcții sunt citiți nu scriși
- Cuvântul `const` poate fi adăugat unei declarații pentru a face obiectul respectiv constant; obiectul trebuie inițializat

```
const int model = 90;            //model este const
const int v[] = { 1, 2, 3, 4 };  //v[i] este const
const int x;         //error C2734: 'x' : const object
                     //must be initialized if not extern
```

Constante

- Declararea unui obiect `const` va asigura ca valoarea acestuia nu se modifică în cadrul blocului.

```
void f( void )
{
        model = 200;
        v[2]++;
}
```

```
error C3892: 'model' : you
cannot assign to a variable that
is const
```

- Obs. Cuvântul cheie `const` modifică un tip în sensul că restricționează modurile în care un obiect poate fi folosit.

```
void g(const X* p)
{
        //(*p) nu poate fi modificat
}
void h(void)
{
        X val; //val poate fi modificat
        g(&val);
}
```

Constante

- Funcție de compilator, se poate rezerva spațiu pentru variabila respectivă sau nu

```
const int c1 = 1;
const int c2 = 2;
const int c3 = my_f(3);    //valoarea lui c3 nu este
                           cunoscuta la compilare
extern const int c4;//valoarea lui c4 nu este cunoscuta la
                    compilare
const int *p = &c2; //este necesar a se aloca spatiu
                    pentru c2
```

- Pentru vectori se alocă memorie deoarece compilatorul nu poate ști ce element al vectorului este folosit intr-o expresie.

Constante

- Constantele sunt utilizate în mod frecvent ca limite pentru vectori sau *case labels*:

```
const int a = 42;
int b = 99;
const int max = 128;
int v[max];
void f(int i)
{
        switch(i)
        {
                case a:
                ...
                case b:
                ...
        }
}
```

error C2051: case expression not constant

Pointeri și constante

- Prin folosirea unui pointer sunt implicate două obiecte: pointerul însuși și obiectul pointat
- Precedarea declarației unui pointer cu un `const` face obiectul (nu pointerul) constant.
- Declararea unui pointer constant presupune utilizarea *`const` și nu doar *.

Pointeri și constante

```
void f1(char* p){
        char s[] = "obiect";
        const char* pc = s; //pointer catre constant
        pc[3] = 'g';        //error: pc pointeaza catre constant
        pc = p;             //ok
        char *const cp = s; //pointer constant
        cp[3] = 'a';        //ok
        cp = p;             //error: cp este constant
        const char *const cpc = s; //pointer constant catre
                                   constant
        cpc[3] = 'a';       //error: cpc pointeaza catre
                            constant
        cpc = p;            //error: cpc este constant
}
```

Pointeri şi constante

- Operatorul care face un pointer constant este *const. Declaratorul const* este considerat că face parte din tipul de bază.

```
char *const cp;      //pointer constant catre char
char const* pc;      //pointer catre char constant
const char* pc2;     //pointer catre char constant
```

Pointeri şi constante

- Modificatorul const se foloseşte frecvent la funcţii, la declararea parametrilor formali de tip pointer sau referinţe, pentru a interzice funcţiilor respective modificarea datelor spre care pointează parametrii respectivi.

```
char* strcpy(char* p, const char* q);  //nu poate modifica (*q)
```

- Protecţia datelor cu ajutorul const nu este totală pentru toate compilatoarele

Namespace

- Orice program constă din mai multe părţi separate.
- Ex.
  - Programul "Hello, world!" implică cel puţin două părţi: cererea utilizatorului de a tipări *Hello , world!* şi sistemul I/O care va realiza tipărirea
- *Namespace* permite gruparea unor entităţi cum ar fi clase, obiecte şi funcţii sub acelaşi nume. Astfel, domeniul global poate fi divizat în subdomenii, fiecare cu numele său.
- Formatul este:

```
namespace identificator
{
        entitati
}
```

- Unde *identificator* este un nume de identificare valid iar *entitatile* sunt clase, obiecte şi funcţii care sunt incluse în acel namespace

Namespace

- Exemplu:

```
namespace myFirstNamespace
{
        int a = 15;
        int b = 17;
}
//accesarea variabilelor
myFirstNamespace::a;
myFirstNamespace::b;
```

- În exemplul de mai sus *a* și *b* sunt variabile declarate în namespace-ul *myFirstNamespace*. Pentru a accesa variabilele respective din afara *myFirstNamespace* se va folosi operatorul de domeniu.

```
namespace mySecondNamespace
{
        int a = 25;
        int b = 27;
}
```

Namespace

- Funcționalitatea namespace-urilor este utilă atunci când există posibilitatea ca un obiect global sau funcție să aibă aceeași denumire cauzând erori de redefinire.

```
int main (void) {
    cout << myFirstNamespace::a << endl;
    cout << mySecondNamespace::a << endl;
    return 0;
}
```

- Cuvântul cheie *using* este folosit pentru a introduce un nume/entitate dintr-un namespace în regiunea curentă de declarații

```
int main (void) {
    using myFirstNamespace::a;
    using mySecondNamespace::b;
    cout << a << endl;
    cout << b << endl;
    cout << myFirstNamespace::b << endl;
    cout << mySecondNamespace::a << endl;
    return 0;
}
```

Namespace

- Cuvântul cheie *using* poate fi folosit pentru a introduce un întreg namespace

```
int main (void) {
    using namespace myFirstNamespace;
    cout << a << endl;
    cout << b << endl;
    cout << mySecondNamespace::a << endl;
    cout << mySecondNamespace::b << endl;
    return 0;
}
```

Namespace

- Se pot utiliza mai multe namespace-uri dacă se ţine cont de domeniu

```
int main (void) {
    {
        using namespace first;
        cout << x << endl;
    }
    {
        using namespace second;
        cout << x << endl;
    }
    return 0;
}
```

- Se pot declara nume alternative pentru namespace-uri existente astfel:

```
namespace new_name = current_name;
```