

Universitatea Tehnică „Gheorghe Asachi” din Iași
Facultatea de Automatică și Calculatoare



PROGRAMARE II

Curs 2

Particularități ale limbajului C++.
Principiile POO.

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);
}
```

14

1.5

```
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;
}
```

Supraîncărcarea funcțiilor

- ▶ Compilatorul examinează tipurile argumentelor și decide care dintre cele două funcții `operatie` să fie apelată
 - ▶ O funcție nu poate fi supraîncărcată numai prin tipul pe care îl 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 citați 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

- Declaraarea 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

Error C3892: 'v': you cannot assign to a variable that is const

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 într-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

- Modificatorul *const* se folosește frecvent la funcții, la declararea parametrilor formali de tip pointer sau referințe, pentru a interzice funcțiilor respective modificarea datelor spre care pointează parametrii respectivi.

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

- Protecția datelor cu ajutorul *const* nu este totală pentru toate compilatoarele
-

Namespace

- ▶ Orice program constă din mai multe părți separate.
- ▶ Namespace permite gruparea unor entități cum ar fi clase, obiecte și funcții sub același 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 entitățile sunt clase, obiecte și funcții care sunt incluse în acel namespace
-

Namespace

► Exemplu:

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

- 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

- ▶ namespace-urile sunt utile atunci când există posibilitatea ca un obiect global sau funcție să aibă aceeași denumire

```
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

```
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 ține 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;
```

Principiile de bază ale POO

- ▶ **Abstractizarea datelor**
 - ▶ **Incapsularea**
 - ▶ Moștenirea
 - ▶ Polimorfismul
-

Abstractizarea datelor

- ▶ Abstractizarea - procesul de recunoaștere și de concentrare asupra caracteristicilor importante ale unui obiect
 - ▶ Concentrare asupra semnificației unui tip de date și asupra comportamentului său
 - ▶ Înlăturarea detaliilor de implementare irelevante
 - ▶ Generarea unor funcții care creează și manipulează datele și pe care se bazează toate funcțiile de acces
 - ▶ Abstractizarea datelor nu este o știință exactă: există nenumerate moduri de abstractizare a unui obiect sau a unei situații
-

Clasa

- ▶ Este o extindere a noțiunii de structură din C
 - ▶ Conține atât date, cât și **funcții** (metode)
 - ▶ Permite restricționarea accesului la membri prin folosirea specificatorilor de acces
 - ▶ Poate fi declarată utilizând atât cuvântul `class` cât și cuvântul `struct`, diferența făcând-o numai specificatorul de acces implicit
 - ▶ Un obiect este o instanțiere a unei clase
-

Incapsularea

- ▶ Posibilitatea de a ascunde utilizatorului unei clase detaliile ei de implementare se numeste incapsulare
 - ▶ O clasă va conține
 - ▶ O parte privată – implementare
 - ▶ O parte publică – interfață
 - ▶ Exemplu: clasă ce implementează o stivă
 - ▶ Implementarea se poate schimba (de la o implementare cu tablouri la o implementare cu liste înlanțuite)
 - ▶ Utilizatorul nu are acces direct la tablou sau la listă
 - ▶ Interfața rămâne aceeași (funcțiile push, pop, etc)
 - ▶ Incapsularea datelor este realizată în C++ prin utilizarea specificatorilor de acces
-

Specificatori de acces

- ▶ **Modifică drepturile de acces către membrii clasei**
 - ▶ `private`: membrii privați pot fi accesați numai de membri *aceleiași clase sau de membrii claselor sau funcțiilor prietene*
 - ▶ `protected`: membrii protejați sunt accesibili în cadrul *aceleiași clase sau dintr-o clasă prietenă, dar și din clasele derivate din acestea*
 - ▶ `public`: membrii publici sunt accesibili de oriunde din *domeniul de vizibilitate al obiectului de tip clasă*
-

Exemplu specificatori de acces

```
#include <iostream>

class Data
{
private:
    int zi, luna;
public:
    int an;
};

int main()
{
    Data d1;
    d1.zi = 1;
    d1.luna = 2;
    d1.an = 1900;

    std::cout << "Data calendaristica este" << d1.zi << "-" << d1.luna
    << "-" << d1.an;
    return 0;
}
```

Exemplu specificatori de acces

- ▶ Error 1 error C2248: 'Data::zi' : cannot access private member declared in class 'Data' [...]
 - ▶ Error 2 error C2248: 'Data::luna' : cannot access private member declared in class 'Data' [...]
-

Specificator de acces implicit

```
class Data
{
private:
    int zi, luna;
public:
    int an;
};
```

```
class Data
{
    int zi, luna;
public:
    int an;
};
```

- ▶ În cadrul unei clase, specificatorul de acces implicit este `private`
 - ▶ Membrii celor două clase au aceiași specificatori de acces
 - ▶ În cadrul unei structuri, specificatorul de acces implicit este `public`
-

Metode ale claselor

- ▶ O clasă poate să conțină funcții
- ▶ O funcție a unei clase se numește metodă a clasei
- ▶ Metodele claselor se supun restricțiilor generate de către specificatorii de acces

```
class Data
{
    int zi, luna, an;
public:
    void afisare();
    void citire();
};
```

Metode ale claselor – definiția lor

```
void Data::citire()
{
    std::cout << "Zi:";
    std::cin >> zi;
    std::cout << "Luna:";
    std::cin >> luna;
    std::cout << "An:";
    std::cin >> an;
}
```

```
void Data::afisare()
{
    std::cout << "Data calendaristica este" <<
    zi << "-" << luna << "-" << an << std::endl;
}
```

- ▶ Se folosește operatorul de rezoluție înaintea numelui funcției pentru a specifica clasa din care face parte funcția
- ▶ Ale cui sunt variabilele zi, luna, an din funcții?

Metode ale claselor – definiția lor

- ▶ De obicei, se evită definiția metodelor în interiorul clasei sau în fișierele header
 - ▶ Definițiile metodelor se scriu într-un fișier sursă separat.
 - ▶ Ca metodă de bună practică, fiecare clasă va avea două fișiere:
 - ▶ Unul header (.h sau.hpp) în care se va găsi definiția clasei
 - ▶ Unul sursă (.cpp) cu definițiile metodelor clasei
-