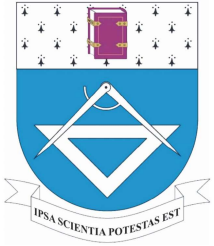

Universitatea Tehnică „Gheorghe Asachi” din Iași
Facultatea de Automatică și Calculatoare

Programarea Calculatoarelor II

Curs 5

Functii inline. Compoziția și agregarea. Template-uri.

Funcții membre constante

- Fie clasa:

```
class Data
{
    int an, zi, luna;
 public:
    Data() :zi(0), luna(0), an(0) {};
    int getZi() const
    {
        return zi;
    }
    int getLuna() const
    {
        return luna;
    }
    int getAn() const
    {
        return an;
    }
    int addAn() const;
};
```

Funcții membre constante

- Prin prezența cuvântului cheie `const` se transmite compilatorului faptul că funcțiile respective nu pot modifica conținutul obiectului *Data*.

```
int Data::addAn() const
{
    an++;
    return an;
}
```

```
error C3490: 'an' cannot be modified because it is being
accessed through a const object
```

- O funcție membră constantă definită în afara clasei necesită repetarea sufixul `const`.

Funcții membre constante

- O funcție membră poate fi apelată atât de obiecte constante cât și de obiecte non-constante.
- Se modifică: `int addAn();`

```
int Data::addAn()
{
    an++;
    return an;
}

void f(Data d, const Data cd)
{
    int i = d.getAn();//ok
    d.addAn();//ok
    int j = cd.getAn();//ok
    cd.addAn();
    //error C2662 : 'int Data::addAn(void)' : cannot convert
    //'this' pointer from 'const Data' to 'Data &'
}
```

Funcții inline

- Funcțiile `inline` sunt funcții care forțează compilatorul să le expandeze inline
- Cu alte cuvinte, compilatorul va insera tot corpul funcției acolo unde este apelată
- Se renunță astfel la instrucțiunile de salt, care implică un timp mare de execuție
- Crește dimensiunea programului
- Unele compilatoare ignoră funcțiile inline definite de utilizator și decid singure ce funcții vor fi inline și care nu
- Funcțiile de dimensiuni mici și care se apelează foarte des trebuie să fie `inline`
- Majoritatea compilatoarelor nu vor face `inline` o funcție recursivă

Funcții inline C++

- Funcțiile declarate și definite în clase sunt implicit funcții inline
- Se preferă folosirea cuvantului cheie `inline` la definirea funcției declarate în clasă
- Există diferite metode de a forța o anumită funcție să fie `inline` (`pragma inline_recursion(on)`, `__forceinline`, `etc`)

Compoziția și agregarea

- În lumea reală, obiectele mari sunt de obicei compuse din mai multe obiecte mici (o mașină are roți, motor, etc)
- Procesul de construire a obiectelor complexe din mai multe sub-obiecte poartă numele de compoziție sau agregare
- C++ permite utilizarea claselor ca obiecte membre ale altor clase

Compoziția

- Compoziția
  - Toți membrii au aceeași durată de viață ca și clasa în care au fost incluși
  - Obiectul părinte "are" (deține) (*eng. "has a"*) obiectele de tipul claselor folosite

Compoziția

```
class Adresa
{
        char * strada;
        int nr;
        //...
};
```

```
class Persoana
{
        Adresa *m_adresa;
public:
        Persoana ()
        {
                m_adresa = new Adresa;
        }
        ~Persoana()
        {
                delete m_adresa;
        }
};
```

- Adresa îi aparține persoanei
- Alt exemplu:
  - Relația casă - cameră

Agregarea

- Este un tip special de compoziție
- Obiectele membre pot avea durate de viață diferite față de clasa părinte
- Compoziția este un caz special de agregare

Agregarea

```
class Profesor
{
        char * nume;
        int vechime;
        //...
};
```

```
class Departament
{
        Profesor * profesor;
public:
        Departament(Profesor * prof)
                :profesor(prof){};
        ~Departament();
};
```

- Profesorii nu sunt detinuți de departamentul in care se află
- Alt exemplu:
  - Relația companie - angajat

Template-uri

- Funcție pentru determinarea maximului dintre două numere

```
int maxim (int a, int b)
{
      return a > b ? a : b;
}
...
int x=16, y=6;
std::cout<< "Max(x,y)" << maxim (x,y)<< std::endl;
```

```
Max(x,y) = 16
```

Template-uri

```
float m=10.5f, n=10.7f;
cout<< "Max(m,n) = " << maxim (m,n)<<endl;
```

```
Max(m,n) = 10
```

- Supraîncarcarea funcției maxim pentru a putea prelucra și float-uri

```
float maxim (float a, float b)
{
      return a > b ? a : b;
}
```

```
cout<< "Max(m,n) = " << maxim (m,n)<<endl;
```

```
Max(m,n) = 10.7
```

Template-uri

- Se poate scrie doar funcția pentru float-uri pentru prelucrarea datelor de tip int?

- Dacă se comentează funcția maxim pentru int-uri:

```
cout<< "Max(x,y) = " << maxim (x,y)<<endl;
```

```
warning C4244: 'argument' : conversion from 'int' to
'float', possible loss of data
```

Conversii int-float

- Când se pierd date la conversiile între int și float?

```
int x=1677721789;
cout << "int - x = " << x << endl;

float aux = x;
cout << "float - x = " << aux << endl;

int i = aux;
cout << "int - float - int x = " << i << endl;
```

```
int - x = 1677721789
float - x = 1.67772e+009
int - float - int x = 1677721728
```

- Se pierd date când există biți de 1 în afara mantisei în reprezentarea pe 32 de biți în virgulă mobilă

Template-uri

- Pentru majoritatea tipurilor de date introduse, ar trebui scrisă câte o funcție
- Se supradimensionează programul scris

- Există soluții pentru a scrie numai o singură funcție care să aibă ca parametru și tipul datelor?

Template-uri (șabloane)

- La funcții
  - Funcții speciale care pot opera cu tipuri generice de date (tipurile de date cu care operează funcția sunt transmise ca parametri)
- La clase
  - Clase ce folosesc tipuri generice de date pentru membrii clasei

Template-uri de funcții

- Se definesc utilizând unul dintre următoarele apeluri:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

- Din punctul de vedere al compilatorului, cele două moduri de definire au același comportament
- Al doilea mod a fost introdus pentru a nu crea confuzii între clase și tipul parametrilor

Template pentru funcția maxim

```
template <typename T>
T maxim (T a, T b)
{
        return a > b ? a : b;
}
...
cout<< "Max(x,y) = " << maxim <int> (x,y)<<endl;
cout<< "Max(m,n) = " << maxim <float> (m,n)<<endl;
```

- Se va afișa pe ecran:

```
Max(x,y) = 16
Max(m,n) = 10.7
```

Template argument deduction

- Compilatorul poate deduce tipul funcției template apelată din tipul parametrilor

- Pentru apeluri simple, se poate evita transmiterea tipului funcției template ca parametru:

```
cout<< "Max(x,y) = " << maxim (x,y)<<endl;
cout<< "Max(m,n) = " << maxim (m,n)<<endl;
```

Template-uri de funcții

- Se poate apela funcția maxim și pentru tipuri de dată definite de utilizator?

```
class PersoanaAC
{
private:
	int m_ivarsta;
public:
	PersoanaAC (int varsta): m_ivarsta(varsta){}
	int getVarsta(){return m_ivarsta;}
	bool operator> (PersoanaAC aux);
};

bool PersoanaAC::operator>(PersoanaAC aux)
{
	if (m_ivarsta > aux.m_ivarsta)
		return true;
	else
		return false;
}
```

Template-uri de funcții

```
PersoanaAC p1(10), p2(20);

PersoanaAC m = maxim <PersoanaAC> (p1, p2);

std::cout << "Varsta maxima este " << m.getVarsta() <<
        std::endl;
```

```
Varsta maxima este 20
```

Template-uri de clase - Vector de întregi

```
class IntVector
{
private:
	int m_iNr;
	int * m_piData;

public:
	IntVector (int n);
	~IntVector();
	int& operator[](int idx)
	{
		if (idx < 0 || idx >= m_iNr)
		{
			std::cout << "Index gresit" <<
				std::endl;
			exit(-1);
		};
		return m_piData[idx];
	}
};
```

Vector de întregi

```
IntVector :: IntVector (int n)
{
	m_iNr = n;
	m_piData = new int[n];
}

IntVector :: ~IntVector ()
{
	if (m_piData)
		delete [] m_piData;
}
```

Vector de întregi

```
int main ()
{
        int n = 10;
        IntVector vec1(n);

        for (int i = 0; i < n; i++)
                vec1[i] = 2 * i;

        for (int i = 0; i < n; i++)
                cout << i << " = " << vec1[i] << endl;

        return 0;
}
```

```
0 = 0
1 = 2
2 = 4
3 = 6
4 = 8
5 = 10
6 = 12
7 = 14
8 = 16
9 = 18
```

- Cum ar trebui schimbată clasa dacă s-ar dori crearea unui vector de valori `double`? Dar dacă dorim să avem 2 vectori, unul de întregi și unul de valori double?

Clasa vector generic

```
template <typename Tip>
class Vector
{
private:
	int m_iNr;
	Tip * m_piData;

public:
	Vector (int n);
	~Vector();
	Tip& operator[](int idx)
	{
		if (idx < 0 || idx >= m_iNr)
		{
			cout << "Index gresit" << endl;
			exit(-1);
		};
		return m_piData[idx];
	}
};
```

```
template <typename Tip>
Vector <Tip>:: Vector (int n)
{
        m_iNr = n;
        m_piData = new Tip[n];
}

template <typename Tip>
Vector <Tip>:: ~Vector ()
{
        if (m_piData)
                delete [] m_piData;
}
```

- Implementarea (definiția) funcțiilor dintr-un template la clasă se face în același fișier cu template-ul

```
int main ()
{       int n=10;
        Vector <double> vecDouble (n);

        for (int i = 0; i<n; i++)
                vecDouble[i] = 2.5 * i;

        for (int i = 0; i<n; i++)
                cout << i << " = " << vecDouble[i] << endl;

        cout<<endl;
        Vector <int> vecInt(n);

        for (int i = 0; i<n; i++)
                vecInt[i] = 2.5 * i;

        for (int i = 0; i<n; i++)
                cout << i << " = " << vecInt[i] << endl;

        return 0;
}
```

```
0 = 0
1 = 2.5
2 = 5
3 = 7.5
4 = 10
5 = 12.5
6 = 15
7 = 17.5
8 = 20
9 = 22.5
```

```
0 = 0
1 = 2
2 = 5
3 = 7
4 = 10
5 = 12
6 = 15
7 = 17
8 = 20
9 = 22
```

Return prin referință

- Când o variabilă e returnată prin referință, se creează o referință (sinonim) la obiectul returnat
- Utilizatorul poate utiliza această referință pentru a modifica obiectul referențiat
- Altfel spus, se utilizează return prin referință atunci când tipul funcției trebuie să fie o valoare stânga
- Trebuie avut în vedere domeniul de valabilitate al variabilei referențiate:

```
int& testReferinta (int a)
{
        int aux = a*2;
        return aux;
}
```

```
warning C4172: returning address of local variable or
temporary
```

Return prin referință

```
int& setVal (int * vect, int idx)
{
	return vect[idx];
}


int main ()
{
	int vect[] = {1, 2, 3, 4, 5};

	setVal(vect, 2) = 100;

	for (int i = 0; i < 5; i++)
		std::cout << vect[i] << " ";

	return 0;
}
```

```
1 2 100 4 5
```