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

# PROGRAMARE II

## Curs 6

*Template-uri*
*Return prin referință.*

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

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

```
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
```

- 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

- 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 (şabloane)

- La funcţii
  - Funcţii speciale care pot opera cu tipuri generice de date (tipul de date cu care operează funcţia este transmis ca parametru)
- La clase
  - Clase ce folosesc tipuri generice de date pentru membrii clasei

## Template-uri de funcţii

- 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 acelaşi comportament
- Al doilea mod a fost introdus pentru a nu crea confuzii între clase şi tipul parametrilor

## Template pentru funcţia 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 afişa pe ecran:

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

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

```
class PersoanaAC
{
protected:
	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;
}
```

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

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

cout << "Varsta maxima este " << m.getVarsta() << 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)
		{
			cout << "Index gresit" << 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;
}
```

```
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 sa fie o valoare stanga
- 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++)
		cout<<vect[i]<<" ";

	return 0;
}
```

```
1 2 100 4 5
```