Lab 6 – P2

Cuprins

1. Template-uri de funcții

Template-urile de funcții sunt funcții speciale ce pot opera cu tipuri generice de date. Prin utilizarea template-urilor de funcții se creează funcții ce pot fi folosite ușor pentru mai multe tipuri de date. În general, se utilizează template-urile de funcții pentru a nu scrie câte o funcție pentru fiecare tip de dată ce se va folosi în program. Template-urile de funcții se utilizează în situațiile în care operațiile realizate nu depind în mod necesar de tipul de dată al parametrilor implicați. Spre exemplu, metoda de calcul a maximului dintre două numere este aceeași pentru valori întregi, reale sau orice alt tip de dată definit de utilizator. În mod normal, ar trebui sa scriem câte o funcție pentru fiecare tip de dată implicat, după modelul următor:

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

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

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

Se observă faptul că toate funcțiile au aceeași definiție, făcând același lucru, fiind suficientă compararea celor doi parametri. Template-urile permit evitarea repetării codului aferent acelorași funcții pentru tipuri de dată diferite. Prin utilizarea template-urilor, se poate crea un singur template de funcție, căruia ii vom transmite tipul de dată ca parametru, dupa modelul urmator:

```
template <typename T>
T maxim(T a, T b)
{
	return (a > b) ? a : b;
}
```

Toate template-urile de funcții trebuie să înceapă cu o instrucțiune de forma

```
template <typename T>
```

Această instrucțiune specifică faptul că definiția funcției ce urmează este un template, iar T va fi un tip de dată transmis de utilizator la momentul apelării funcției. Pentru funcția maxim definită anterior, apelul ei se poate face utilizând apelurile prezentate mai jos:

```
void main()
{
	int x = 2, y = 4;
	float a = 3.3, b = 1.5;

	int maxI = maxim <int> (x, y);
	float maxF = maxim <float> (a, b);
}
```

La apelul funcției ce are definit un template, se specifică tipul de date prin transmiterea acestuia între parantezele unghiulare <>.

Tipul de dată al unui template poate fi oricare, inclusiv unul definit de utilizator:

```
class Masina
{
	char culoare[20];
	int an;

public:
	Masina(char c[], int a)
	{
		int len = strlen(c);
		strcpy_s(culoare, len + 1, c);
		an = a;
	}

	Masina(const Masina &m)
	{
		int len = strlen(m.culoare);
		strcpy_s(culoare, len + 1, m.culoare);
		an = m.an;
	}

	void afiseaza()
	{
		std::cout << culoare << " " << an << std::endl;
	}
};
```

```
int main()
{

    Masina m1("alb", 2012), m2("negru", 2018);
    Masina m3 = maxim <Masina> (m1, m2);//eroare! cum comparam doua masini?
    m3.afiseaza();

    return 0;
}
```

În acest caz, pentru ca funcția maxim() să lucreze corect, trebuie specificată o modalitate de comparare a două obiecte din acea clasă, de exemplu prin supraîncărcarea operatorului de comparare **<** . Presupunem că dorim sa determinăm masina cu cel mai recent (mai mare) an de fabricație:

```
class Masina
{
      char culoare[20];
      int an;

public:
      Masina(char c[], int a)
      {
            int len = strlen(c);
            strcpy_s(culoare, len + 1, c);
            an = a;
      }

      Masina(const Masina &m)
      {
            int len = strlen(m.culoare);
            strcpy_s(culoare, len + 1, m.culoare);
            an = m.an;
      }

      void afiseaza()
      {
            cout << culoare << " " << an << endl;
      }

      bool operator<(const Masina &m)
      {
            return (an < m.an) ? true : false;
      }
};
```

```
void main()
{
	Masina m1("alb", 1999), m2("negru", 2000);

	//in acest caz se compara m1.an si m2.an
	Masina m3 = maxim<Masina>(m1, m2);
	m3.afiseaza();
}
```

Același lucru e valabil și în cazul altor operatori (+, *, =, ==, etc.). Ori de câte ori se dorește utilizarea lor în cadrul unei funcții, trebuie ca rolul lor să fie clar stabilit atunci când se lucreaza cu tipuri de dată definite de programator (clase). Acest lucru este cu atât mai ușor de trecut cu vederea în cazul template-urilor, unde tipurile de dată utilizate nu sunt specificate, fiind astfel mai dificil de anticipat dacă e nevoie ca o anumită operație sa fie definită explicit.

Template-urile pot apărea, de asemenea, și ca membri ai unei clase (template-uri de metode):

```
class A
{
	int val;

public:
	A(int v):val(v){}

	template <typename T>
	T Produs(T x)
	{
		return x * val;
	}
};
```

```
void main()
{
	A  a(2);
	float b = 3;

	cout << a.Produs<float>(b) << endl;
}
```

2. Template-uri de clase

Clasele pot fi, de asemenea, specificate prin intermediul template-urilor. Un template de clasă are unul sau mai multe tipuri, date ca parametri. Acești parametri pot fi folosiți în interiorul clasei, urmând să se specifice la crearea unui obiect din acea clasă.

Considerăm ca exemplu o clasă care definește un punct în plan, cu cele două coordonate ale sale, x, y. La definirea clasei, nu interesează deocamdată dacă acele coordonate sunt întregi sau reale. Acest lucru se va specifica doar în momentul în care vom crea obiecte de tip Punct:

```
template <typename T>
class Punct
{
	T x, y;
public:
	Punct(T xcoord, T ycoord) : x(xcoord), y(ycoord){}
	void afiseaza()
	{
		cout << x << ", " << y << endl;
	}
};
```

```
void main()
{
	Punct<int> punctI(2, 3);
	Punct<float> punctF(2.3f, 3.4f);
	Punct<double> punctD(4.5, 5.6);

	punctI.afiseaza();
	punctF.afiseaza();
	punctD.afiseaza();
}
```

Un caz frecvent de utilizare a template-urilor de clasă îl constituie încapsularea și gestiunea unei structuri de date (ex. un vector). Ca și in cazul anterior, nu interesează tipul de dată efectiv al elementelor din acea structură de date. Dorim sa scriem o singură clasă care, teoretic, să funcționeze cu date de orice tip. Acel tip va fi specificat doar la crearea obiectelor din acea clasă:

```
template <typename T>
class MyVector
{
	T* elem;
	int nrElem;
public:
	MyVector(int n)
	{
		nrElem = n;
		elem = new T[n];
	}
	~MyVector()
	{
		if(elem)
			delete[] elem;
	}

	//supraincarcarea operatorului de indexare:
	T& operator[](int index)
	{
```

```
            return elem[index];
        }
};
```

```
void main()
{
        int n = 10;
        MyVector<int> vectorI(n); //un vector de numere intregi
        MyVector<float> vectorF(n); //un vector de numere reale

        for(int i = 0; i < n; i++)
        {
                /* accesam elementele din membrul privat elem
                prin intermediul operatorului [] supraincarcat in clasa */
                vectorI[i] = i + 1;
                vectorF[i] = 1.5 * i;
        }
}
```

Template-urile se utilizează în cardul unui stil de programare numit "programare generică" (*generic programming)*, unde se dorește separarea algoritmilor și metodelor de tipul și caracteristicile datelor cărora le vor fi aplicate. De exemplu, un algoritm de sortare se aplică datelor de orice tip, fiind suficient să existe metode de comparare și interschimbare a oricăror două elemente de acel tip.

3. Exerciţii

1. Scrieți un template de funcție care să adune două elemente primite ca parametru. Testați funcția atât pe tipurile de datre standard, cât și pe un tip de date definit de utilizator (de exemplu Complex).

2. Scrieți un template de clasă Multime, care să conțină un vector alocat static (vectorul cu elementele mulțimii) și un număr întreg (numărul de elemente din mulțime).

- Definiți constructori de inițializare și copiere și destructorul clasei. Testați-i pentru mulțimi de numere întregi și reale;

- Supraîncărcați operatorul ! (semnul exclamării) pentru a obține cel mai mare element din mulțime;

- Supraîncărcați operatorul + pentru a aduna două mulțimi element cu element;

- Scrieți un template de funcție Aduna, care primește doi parametri de un tip oarecare și returnează rezultatul adunării lor. Utilizați template-ul pentru a aduna două multimi cu elemente de tip double.

3. Scrieți un template de funcție care să sorteze elementele unui vector și să returneze vectorul astfel sortat. Funcția primește ca parametri un pointer la un tip oarecare și un număr întreg (numărul de elemente ale vectorului). Se poate utiliza orice metodă de sortare (eg. *bubble sort*). Aplicați funcția pentru vectori de diferite tipuri (int, float, etc). Pentru codul celor mai folosiți algoritmi de sortare accesați pagina https://www.geeksforgeeks.org/sorting-algorithms/#algo