

Lab 6 – P2

Cuprins

1. Template-uri de funcții	1
2. Template-uri de clase	4
3. Exerciții	6

1. Template-uri de funcții

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 sau reale. În mod normal, ar trebui să scriem câte o funcție pentru fiecare tip de dată implicat (int, float sau double):

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

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

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

Observăm faptul că toate trei funcțiile fac același lucru, fiind suficientă doar existența unei modalități de comparare a celor doi parametri. Template-urile permit evitarea implementării acelorași operații pentru tipuri de dată multiple. Astfel, vom crea un singur template de funcție, căruia îi vom transmite tipul de dată ca parametru, după modelul următor:

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

Tipul de dată T este încă nespecificat. Vom specifica acest tip la apelul funcției:

```
void main()
{
    int x = 1, y = 2;
    float a = 2.3, b = 1.5;

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

Tipul de dată al unui template poate fi oricare, inclusiv o clasă creată de utilizator:

```
class Masina
{
    char *culoare;
    int an;

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

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

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

```
void main()
{
    Masina m1("alb", 1999), m2("negru", 2000);
    Masina m3 = maxim<Masina>(m1, m2); //eroare! cum comparam doua masini?
    m3.afiseaza();
}
```

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 să determinăm mașina cu cel mai recent (mai mare) an de fabricație:

```
class Masina
{
    char *culoare;
    int an;

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

    Masina(const Masina &m)
    {
        int len = strlen(m.culoare);
        culoare = new char[len + 1];
        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 lucrează 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 să 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 în cazul anterior, nu interesează tipul de dată efectiv al elementelor din acea structură de date. Dorim să 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;
    }

    //supraincercarea 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 cadrul 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. Exerciții

1. 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).

2. Scrieți un template de clasă Multime, care să conțină un pointer la un tip oarecare (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 mulțimi cu elemente de tip double.