Lucrarea de laborator nr. 6.

Tema: Templates: Template pentru clase şi funcţii.

Scopul lucrării: familiarizarea studenţilor cu clase şi funcţii generice.

Consideraţiile teoretice necesare:

Clase şi funcţii generice

Template-ul implementează aşa-zisul concept de "*tip parametrizat*" ("*parametrized type*"). Un *template* reprezintă o familie de tipuri sau funcţii, cu alte cuvinte, un şablon sau model. Acest concept a fost introdus, în primul rînd, pentru a creşte gradul de reutilizare a codului. De exemplu, pentru a implementa o listă de numere întregi este necesară realizarea unei clase speciale (să spunem *ListOfIntegers*), iar pentru o listă de şiruri altă clasă (să spunem *ListOfStrings*). Conceptul de *template* permite realizarea unei clase generale (să spunem *List*), care să accepte orice tip de element, inclusiv tipuri necunoscute la momentul implementării acesteia. Tipul *template* este stabilit în momentul instanţierii sale. Template-urile sînt foarte utile pentru realizarea de biblioteci care trebuie să ofere metode generice de prelucrare a datelor. Sintaxa generală de declarare a unui template este următoarea:

template < ListaDeParametri > Declaraţie

unde *Declaraţie* reprezintă declararea sau definirea unei clase sau unei funcţii, definirea unui membru static al unei clase template, definirea unei clase sau funcţii membri al unei clase template, sau definirea unui membru template al unei clase.

Clasele parametrizate (sau clasele template) se declară astfel:

```
template <class NumeParametru>
class NumeClasa
{  // ...
// definirea clasei
}
```

Stabilirea tipului clasei *template* se face prin intermediul unei construcţii de genul:

NumeClasa <NumeParametru>

unde *NumeParametru* reprezintă tipul obiectului.

Funcţiile template se declară astfel:

```
template <class NumeParametru>
//
...
// declaraţia funcţiei
```

Să considerăm în continuare ca exemplu implementarea unei stive generice folosind template-uri.

```
#include <iostream.h>
template <class T> class StackItem
{ public:
    StackItem *Next;
    Tip  *Data;
    StackItem(Tip  Data_new, StackItem <Tip> *Next_new)
    {  Data = new Tip(Data_new);
       Next = Next_new;  }
};
```

```
template <class Tip> class Stack
{public:
    Tip  pop()
    {  Tip  result = *(Data->Data);
       StackItem <Tip> *temp = Data;
      Data = Data->Next;
      delete temp;
       return result;
    }
    Tip  top()
    {  return *(Data->Data);  }
    void push(Tip  Data_new)
    {  Data = new StackItem <Tip>(Data_new, Data);   }
    int isEmpty()
    {       return Data == 0;  }
    Stack()
    {       Data = 0;  }
private:
    StackItem <Tip> *Data;
};
void main()
{    Stack <int> anIntegerStack;
    anIntegerStack.push(5);
    anIntegerStack.push(7);
    if(anIntegerStack.isEmpty())
      cout << "Stiva goala" << endl;
    else
      cout << anIntegerStack.pop() << endl;
     cout << anIntegerStack.top() << endl;
}
```

Rezultatul îndeplinirii programului:

```
7
5
```

În exemplul următor a fost implementată o listă generică (*List*). Ca elemente a listei s-au folosit obiecte de tip *Point.* Pentru parcurgerea uşoară a listei a fost implementată o clasă de tip "*iterator*", care poate fi considerată ca fiind un "cursor" care străbate lista. Funcţia *List.begin()* returnează un iterator poziţionat pe primul element al listei, *List.end()* pe ultimul element al listei. Saltul la următorul element al listei se face cu ajutorul operatorului ++ din clasa *Iterator*.

```
#include <iostream.h>
class Point
{friend ostream& operator << (ostream& output, Point p);
protected:
unsigned x, y;
public:
Point()  {x = 0;y = 0;}
Point(unsigned X, unsigned Y) {x = X;y = Y;}
```

```
~Point() {}
unsigned GetX() {return x;}
unsigned GetY() {return y;}
void SetX(unsigned X) {x = X;}
void SetY(unsigned Y) {y = Y;}};
ostream& operator << (ostream& output, Point p)
{output<<"(" << p.x <<", " << p.y << ")"; return output;}
template <class Tip> class Item
{public:
Item *Next;
Tip *Data;
Item(Tip __Data, Item <Tip> *__Next)
{Data = new Tip(__Data); Next = __Next;}
};
template <class Tip> class List
{public:
Tip pop_front()
{Tip result = *(Data->Data);
Item <Tip> *temp = Data;
Data = Data->Next;
delete temp;
return result;}
Tip front() {return *(Data->Data);}
void push_front(Tip __Data)
{Data = new Item <Tip>(__Data, Data);}
int empty() {return Data == 0;}
List() {Data = 0;}
class Iterator
{  friend class List <Tip>;
protected:
Item <Tip> *Current;
Iterator(Item <Tip> *x) {Current = x;}
public:
Iterator() {}
int operator == (Iterator& x){return Current == x.Current;}
int operator != (Iterator& x){return Current != x.Current;}
Tip operator *(){return *(Current->Data);}
Iterator& operator ++(int)
{Current = Current->Next; return *this;}
};
Iterator begin(){return Iterator(Data);}
Iterator end()
{Item <Tip> *temp;
for(temp = Data; temp; temp = temp->Next);
return Iterator(temp);}
private:
```

```
Item <Tip> *Data;
};
void main()
{List <Point> anPointList;
List <Point>::Iterator index, end;
anPointList.push_front(Point(1, 1));
anPointList.push_front(Point(3, 14));
index = anPointList.begin(); end = anPointList.end();
if(anPointList.empty()) cout << "Lista vida" << endl;
else
for(; index != end; index++)
cout << *index << " ";
cout << endl;
}
```

Rezultatul îndeplinirii programului:

(3, 14) (1, 1)

Clasele template pot avea trei tipuri de *prieteni* (*friends*):

- o clasă sau funcţie care nu este de tip template;
- o clasă sau funcţie template;
- o clasă sau funcţie template avînd tipul specificat.

Dacă este necesară particularizarea unei funcţii template sau a unei funcţii membri a unei clase template pentru un anumit tip, funcţia respectivă poate fi supraîncărcată pentru tipul dorit.

Trebuie de remarcat, de asemenea, că în cazul în care o clasă *template* conţine membri statici, fiecare instanţă a template-ului în cauză va conţine propriile date statice.

Sarcina pentru lucrări de laborator:

1. Să se scrie un program care implimentează o clasă generică și care realizează algoritmul de sortare: Bubble Sort [interschimbare]
2. Să se scrie un program care implimentează o clasă generică și realizează algoritmul de sortare: Selection Sort [selectie]
3. Să se scrie un program care implimentează o clasă generică și realizează algoritmul de sortare: Insertion Sort [inserare]
4. Să se scrie un program care implimentează o clasă generică și realizează algoritmul de sortare: Shell Sort [inserare]
5. Să se scrie un program care implimentează o clasă generică și realizează algoritmul de sortare: Merge Sort [interclasare]
6. Să se scrie un program care implimentează o clasă generică și realizează algoritmul de sortare: Heap Sort [selectie]
7. Să se scrie un program care implimentează o clasă generică și realizează algoritmul de sortare: Quick Sort [partitionare]

8. Fie **v** un tablou unidimensional de dimensiune **n** şi k un element de acelaşi tip cu elementele tabloului. (**k**=cheie de căutare). Căutarea lui **k** revine de a găsi dacă există sau nu, un element din **v** de valoare k. Să se scrie un program care implimentează o clasă generică și realizează algoritmul de căutare secvenţială – nu se specifică nimic despre elementele tabloului v.

9. Fie **v** un tablou unidimensional de dimensiune **n** şi k un element de acelaşi tip cu elementele tabloului. (**k**=cheie de căutare). Căutarea lui **k** revine de a găsi dacă există sau nu, un element din **v** de valoare k. Să se scrie un program care implimentează o clasă generică și realizează algoritmul de căutare secvenţială în tabloul ordonat.

10. Fie **v** un tablou unidimensional de dimensiune **n** şi k un element de acelaşi tip cu elementele tabloului. (**k**=cheie de căutare). Căutarea lui **k** revine de a găsi dacă există sau nu, un element din **v** de valoare k. Să se scrie un program care implimentează o clasă generică și realizează algoritmul de căutare binară (caz în care tabloul este ordonat).

11. Să se scrie un program care implimentează o clasă generică pentru o listă simplu înlănțuită.

12. Să se scrie un program care implimentează o clasă generică pentru o listă dublu înlănțuită.

13. Să se scrie un program care implimentează o clasă generică pentru o stivă.

14. Să se scrie un program care implimentează o clasă generică pentru o coadă.

15. Să se scrie un program care implimentează o clasă generică pentru o listă circulară.