Laborator 6:Functii si Clase generice (template).

Functii template

O functie template defineste un set de operatii care vor fi aplicate unor tipuri de date variate. Unei astfel de functii, tipul de date asupra caruia va opera ii va fi transmis ca parametru. Utilizand acest mecanism, poate fi aplicata aceiasi procedura unui domeniu larg de date. Multi algoritmi au aceiasi logica, indiferent de tipul de date asupra caruia opereaza, de exemplu un algoritm de sortare va functiona la fel indiferent daca este aplicat pe un vector de tip int sau de tip double. Ceea ce difera, este doar tipul de date care va fi sortat.

Creand o functie generica, putem defini natura algoritmului, indiferent de date. O data facut acest lucru, cand se executa functia, compilatorul genereaza automat codul corect pentru tipul de date folosit efectiv. In esenta, cand creati o functie generica, creati o functie care se supraincarca singura, automat.

O astfel de functie este creata cu ajutorul cuvantului cheie **template.** Forma generala a unei definitii de functie de tip **template** este urmatoarea:

template < class *Tip*> tip-returnat nume-functie (*lista parametri*)

{

//corpul functiei

}

Tip este un nume care tine locul tipului de date folosit de catre functie. Compilatorul il va inlocui autimat cu tipul de date efectiv , atunci cand va crea o versiune specifica a functiei.

Exemplu:

```
#include <iostream>
using namespace std;
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j); // swap integers
```

```
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
system("pause");
return 0;
}
```

Linia template <class X> void swapargs(X &a, X &b) spune compilatorului 2 lucruri: ca este creat un template si ca va urma definirea generica. **X** este aici un tip generic care este folosit ca substitut. Dupa zona template este declarata functia swapargs(), folosind pe **X** ca tip de data pentru valorile care vor fi inversate. In main(), functia swapargs() este apelata cu 3 tipuri de date: int, float si char. Deoarece swapargs() este o functie generica, compilatorul ii va crea automat 3 versiuni - una care va inversa valorile intregi, una care va inversa pe cele in virgula mobila si una care va inversa caractere.
Exemplul 2 :

```
#include <iostream>
using namespace std;
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
cout << x << ' ' << y << '\n';
}
int main()
{
myfunc(10, "I like C++");
myfunc(98.6, 19L);

system("pause");
return 0;
}
```

Intr-o instructiune template putem defini mai mult de un tip generic, folosind o lista separata prin virgula. In exemplul de mai sus, type1 si type2 sunt inlocuiti de catre compilator cu tipurile de date int si char*, respectiv double si long atunci cand genereaza exemplarele specifice pentru myfunc din cadrul lui main.

Supraincarcarea explicita a unei functii template

Chiar daca o functie generic a se supraincarca singura atunci cand este necesar, aceasta poate fii supraincarcata si explicit. Atunci cand o functie generica este supraincarcata, ea suprascrie functia template relativa la acea versiune specifica.

Exemplu:

```
#include <iostream>
using namespace std;
// Functia template.
template <class X> void swapargs(X &a, X &b)
{
```

```
X temp;
temp = a;
a = b;
b = temp;
cout << "In interiorul functiei template.\n";
}
// Suprascrie varianta template a lui swapargs() pentru int.
void swapargs(int &a, int &b)
{
int temp;
temp = a;
a = b;
b = temp;
cout << "In interiorul functiei suprascrise.\n";
}
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j); // se apeleaza varianta suprascrisa explicit
swapargs(x, y); // se apeleaza functia template
swapargs(a, b); // se apeleaza functia template
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
system("pause");
return 0;

}
```

!!! Restrictii ale functiilor template: o functie virtuala nu poate fi functie template si destructorii nu pot fi functii template.

Clase template (generice)

In afara de functiile template, putem defini si clase template. Cand facem asta, cream o clasa care defineste toti algoritmii folositi de ea, dar tipul de date care este manevrat efectiv va fi specificat ca un parametru la crearea obiectelor acelei clase.

Clasele template sunt folositoare cand avem de implementat o clasa ce contine caracteristici generale. Forma generala a declararii unei clasegenerice este urmatoarea:

Template <class *Tip*> class nume-clasa

{..

…}

Tip tine locul tipului ce il vom specifica cand se defineste un exemplar al clasei. Daca este necesar, putem define mai mult de un tip de date generice, folosind o lista separata prin virgule. O data ce am construit o clasa generic, putem crea o instanta a acesteia folosind forma generala:

nume-clasa<tip> ob; In acest caz, tip este numele tipului de date cu care va opera clasa. Functiile membre ale claselor template sunt automat si ele insele template.

Exemplu

```
#include <iostream>
using namespace std;
template <class Type1, class Type2> class myclass
{
Type1 i;
Type2 j;
public:
myclass(Type1 a, Type2 b) { i = a; j = b; }
void show() { cout << i << ' ' << j << '\n'; }
};
int main()
{
myclass<int, double> ob1(10, 0.23);
myclass<char, char *> ob2('X', "Templates add power.");
ob1.show(); // show int, double
ob2.show(); // show char, char *
system("pause");
return 0;
}
```

Exercitii

1. Scrieti o functie template ce realizeaza sortarea unei vector de elemente generice. Functia primeste ca parametru vectorul de elemente si numarul de elemente din vectorul respectiv.
   Creati clasa de numere rational si adaugati operatorii aferenti pentru a putea aplica functia de sortare creata.
2. Scrieti o clasa template, array, ce implementeaza functionalitatile unui vector. Creati membri private de stocare a elementelor vectorului si a dimensiunii vectorului. Creati metode publice pentru setarea si accesarea elementelor vectorlui si de asemeni o metoda publica de afisare a tuturor elementelor vectorului. Creati clasa complex si instatiati un vector in programul principal pentru a putea functiona cu elemente de tip complex. Setati valori pentru toate elementele vectorului cu numere complexe si afisati-le.