Template

Template-uri (template=sablon,model,tipar)

Template-urile C++ permit definirea de functii si clase care au parametri pentru numele de tipuri.

Exemplu:

```
  void schimba (Tip_Var& var1, Tip_Var& var2) {
    Tip_Var temp;

    temp=var1;
    var1=var2;
    var2=temp;
  }
```

unde Tip_Var poate fi **int**, **char**, **double** si altele.

Singura diferenta dintre aceste functii este ca difera tipul parametrilor formali. Aceste functii se pot scrie intr-una singura in urmatorul program C++:

Exemplu:

```
// Program ce ilustreaza o functie template

#include <iostream.h>

template <class T>
void schimba(T& var1, T& var2);
// interschimba valorile lui var1 si var2

int main() {
  int i1 = 1, i2 = 2;
  cout << "Initial, i1 = " << i1 << ", iar i2 = " << i2 << endl;
  schimba(i1, i2);
  cout << "Dupa schimbare, i1 = " << i1 << ", iar i2 = " << i2 << endl;

  char c1 = 'a', c2 = 'b';
  cout << "Initial, c1 = " << c1 << ", iar c2 = " << c2 << endl;
  schimba(c1, c2);
  cout << "Dupa schimbare, c1 = " << c1 << ", iar c2 = " << c2 << endl;
  return 0;
 }

template <class T>
void schimba(T& var1, T& var2) {
  T temp;
  temp = var1;
  var1 = var2;
  var2 = temp;
}
```

Definirea de parametri tip se face folosind cuvantul rezervat *template* urmat de <*class T*>, adica:

```
template <class T>
```

Aceasta constructie se mai numeste **prefix template** si spune compilatorului ca definitia sau prototipul care urmeaza este un template si ca T este un **parametru tip**. (In acest context, **class** inseamna de fapt **tip**). Definitia unei functii template este, de fapt, o multime de definitii de functii. Pentru exemplul nostru, parametrul de tip T se inlocuieste cu un nume de tip.

Cand folosim functii template, putem spune ca exprimam un algoritm general in C++. Acesta este un exemplu simplu de **algoritm abstract**. Algoritmii abstracti ignora detaliile, continand partea semnificativa a algoritmului. Daca functia are mai multi parametri tip atunci declaratia template-ului se generalizeaza imediat.

Pentru doi parametri, scriem:

```
template <class T1, class T2>
```

Exemplu:

```
// Ilustrarea unor situatii neobisnuite

#include <iostream.h>

template <class T>
void f(T a, T b) {
  cout << a << b;
}

int main() {
  f(40000, 6);
  f(1, 2.3);
  f(1, 'a');
  f(2.3, 1);
}
```

Template-uri pentru clase abstracte

Sintaxa pentru template-urile claselor este aceeasi ca template-urile functiilor. Ele sunt precedate de **template <class Parametr_Tip>.**

Exemplu:

Consideram o clasa cu obiecte perechi de valori de tip T, unde T este tip abstract de date (putem pune alt cuvant in loc de T, dar asa se face in mod traditional).

```
template <class T>
class Pereche  {
    public:
     Pereche();
     Pereche(T val1, T val2);
     void seteaza_element(int pozitie, T valoare);
     //Preconditie: pozitie este 1 sau 2
     //Postconditie: Pozitia indicata a fost setata catre valoare

     T obtine_element(int pozitie) const;
```

```
     //Preconditie: pozitie este 1 sau 2
     //Returneaza valoarea pozitiei indicate

    private:
     T primul;
     T al_doilea;
};
```

Declararea unor obiecte de tip Pereche <class T> pot fi:

```
Pereche <int> scor;
Pereche <char> locuri;
```

De exemplu setarea scorului Romania-Germania (3-0) se face astfel:

```
scor.seteaza_element(1,3);
scor.seteaza_element(2,0);
```

Functiile membru pentru o clasa template sunt definite in acelasi mod ca si functiile membru pentru clasele obisnuite. Singura diferenta este ca definitiile functiilor membre sunt ele insele template-uri.

Exemplu:

```
  //definim functia membru seteaza_element si constructorul cu 2
argumente

  template <class T>
  void pereche <T>::seteaza_element (int pozitie, T valoare) {
    if (pozitie == 1)
     primul = valoare;
    else
     if (pozitie == 2)
      al_doilea=valoare;
     else {
       cout << "Eroare: Pozitie inlegala in pereche\n";
       exit(1);
     }
  }

  template <class T>
  Pereche <T>::Pereche(T val1, T val2) {
    primul = val1;
    al_doilea = val2;
  }
```

Observatie: Numele clasei din fata operatorului rezolutiei de domeniu este **Pereche <T>** si nu simplu **Pereche**.

Exemplu:

In continuare, prezentam un program C++ in care se descrie o clasa template a caror obiecte sunt liste. Listele sunt structuri inlantuite de orice tip (int, double, char, ...)

Obiecte de tip Lista pot fi elemente de orice tip pentru care sunt definiti operatorii << si =. Toate articolele unei liste trebuie sa fie de acelasi tip. O lista poate avea "max" elemente de tip Nume_Tip, dupa cum urmeaza:

Lista <Nume_Tip> obiect(max);

```
// Fisierul "lista.cpp"

#include <iostream.h>
#include <stdlib.h>


template <class T>
class Lista {
  public:
    Lista(int max);
    // Initializeaza obiectul la lista vida care poate avea maxim "max" elemente
de tip T

    ~Lista();
    // Returneaza toata memoria dinamica folosita de obiect in heap

    int length() const;
    // Returneaza numarul de elemente ale listei

    void adauga(T element_nou);
    // Preconditie: Lista nu este completa
    // Postconditie: "element_nou" va fi adaugat la lista

    int completa() const;
    // Returneaza true daca lista este completa (adica nu mai sunt elemente de
adaugat)

    void sterge();
    // Sterge toate elementele listei astfel incat lista devine vida

    friend ostream& operator <<(ostream& iesire,
      const Lista <T>& lista);
    // Supraincarca operatorul << care va fi folosit la afisarea listei.
Articolele sunt afisate cate unul pe linie.
    // Preconditie: Daca "iesire" este un flux fisier de iesire, atunci "iesire"
este deja conectat la un fisier.

  private:
    T *element;   // pointer catre un sir dinamic care memoreaza lista
    int lungime_maxima; // numarul maxim de elemente permise de lista
    int lungime_curenta; // numarul de elemente curente ale listei
 };


// Foloseste stdlib.h
template <class T>
Lista <T> :: Lista(int max) {
  lungime_maxima = max;
  lungime_curenta = 0;
  element = new T[max];
  if (element == NULL) {
    cout << "Eroare: Memorie insuficienta.\n";
    exit(1);
  }
}

template <class T>
Lista <T> :: ~Lista() {
```

```
  delete [] element;
}

template <class T>
int Lista <T> :: length() const {
  return (lungime_curenta);
}

// Foloseste iostream.h si stdlib.h
template <class T>
void Lista <T> :: adauga(T element_nou) {
  if (completa()) {
    cout << "Eroare: adaugare la o lista completa.\n";
    exit(1);
  }
  else {
    element[lungime_curenta] = element_nou;
    lungime_curenta ++;
  }
}

template <class T>
int Lista <T> :: completa() const {
  return (lungime_curenta == lungime_maxima);
}

template <class T>
void Lista <T> :: sterge() {
  lungime_curenta = 0;
}

// Foloseste iostream.h
template <class T>
ostream& operator <<(ostream& iesire, const Lista <T>& lista) {
  for (int i = 0; i < lista.lungime_curenta; i++)
    cout << lista.element[i] << endl;
  return iesire;
}
```

```
// Program care demostreaza folosirea clasei template Lista

#include <iostream.h>
#include <stdlib.h>
#include "lista.cpp"


void main(void) {
  Lista <int> prima_lista(2);
  prima_lista.adauga(7);
  prima_lista.adauga(10);
  cout << "prima_lista = \n" << prima_lista;

  Lista <char> a_doua_lista(10);
  a_doua_lista.adauga('T');
  a_doua_lista.adauga('E');
  a_doua_lista.adauga('M');
  a_doua_lista.adauga('P');
  a_doua_lista.adauga('L');
  a_doua_lista.adauga('A');
  a_doua_lista.adauga('T');
```

```
  a_doua_lista.adauga('E');
  cout << "a_doua_lista = \n" << a_doua_lista;

  return 0;
}
```

Supraincarcarea parantezelor patrate ale indexului unui sir

Putem supraincarca parantezele patrate [ ] pentru o clasa astfel incat parantezele patrate pot fi folosite cu obiecte dintr-o clasa. Daca dorim sa folosim [ ] intr-o expresie din partea stanga a operatorului de asignare, atunci operatorul trebuie definit sa intoarca o referinta, care se indica prin adaugarea lui & la tipul returnat (am vazut lucrul acesta si la supraincarcarea operatorilor << si >>).

Cand supraincarcam [], operatorul [] trebuie sa fie functie membru; operatorul [] supraincarcat nu poate fi operator friend (am vazut asta la supraincarcarea operatorului =).

Exemplu:

Definim o clasa **Pereche** a caror obiecte se comporta ca sirurile de caractere care au indecsii 1 si 2 (nu 0 si 1).

```
class Pereche  {
  public:
   Pereche();
   Pereche(char val1, char val2);
   char &operator [] (int index);
  private:
   char primul;
   char al_doilea;
 };
```

Tipul parametrului nu trebuie neaparat sa fie int, deci putem avea indecsi de orice tip. Definitia functiei membru [ ] poate fi:

```
char &Pereche::operator [] (int index) {
  if (index == 1)
    return primul;
  else
    if (index == 2)
      return al_doilea;
    else {
      cout << "Valoare de index ilegala\n";
      exit(1);
    }
 }
```

Exemplu de declarare si apelare a operatorului [ ]

```
Pereche a;

a[1] = 'A';
a[2] = 'B';

cout << a[1] << a[2] << endl;
```

De exemplu "a[1]" inseamna ca "a" este obiectul apelat si 1 este argumentul functiei membru [ ].

Pointer-ul this

Cand definim functiile membre ale unei clase, dorim uneori sa ne referim la obiectul apelat. Pointerul **this** este un pointer predefinit care pointeaza la obiectul apelat.

Exemplu:

```
class Exemplu {
  public:
   ...
   void f();
   ...
  private:
   int a;
   ...
};
```

Urmatoarele doua descrieri de functii sunt echivalente:

```
void Exemplu::f() {
  cout << a;
}
```

si

```
void Exemplu::f() {
  cout << (this->a);
}
```

Observam ca this nu este numele obiectului apelat, dar este numele unui pointer care pointeaza catre obiectul apelat. Pointer-ul this nu-si poate schimba valoarea: acesta intotdeauna pointeaza catre obiectul apelat.

Cu alte cuvinte, **this** intoarce adresa de inceput a obiectului apelat, iar ***this** inseamna referirea la intregul obiect. De aceea, referirea la datele acestuia se face astfel:
(*this).a sau, echivalent **this->a**.

S-a vazut in exemplul precedent ca nu avem nevoie de this pentru a scrie acea functie. Cu toate acestea, exista situatii cand se recomanda folosirea acestuia.

Exemplu:
Supraincarcarea operatorului de asignare =

```
class String  {
  public:
    ...
    String &operator = (const String& parte_dreapta);
    ...
  private:
    char *a; //sir dinamic cu ultimul element '\0'
};
```

Vom supraincarca operatorul = pentru a putea fi folosit in asignari de forma:

```
s1 = s2 = s3;
```

Acest lant de asignari inseamna: s1=(s2=s3);

Practic asignarea s1=s2, inseamna:

- s1 este obiectul din clasa String care se apeleaza
- = operator (functie membru) supraincarcat
- s2 este argumentul valoare a functiei membru =

Definitia operatorului de asignare supraincarcat foloseste pointer-ul **this** pentru returnarea obiectului din partea stanga a operatorului =

```
String& String::operator = (const String & parte_dreapta) {
  delete [ ]a;

  a = new char[strlen(parte_dreapta.a) + 1];
  strcpy(a, parte_dreapta.a);
  return *this;
}
```

Instructiunea "**return *this**" intoarce intregul obiectul s2 ca rezultat al evaluarii s2 = s3. Apoi se face si cealalta asignare s1=(s2=s3);

Cu toate acestea, codul de mai sus are o problema. Daca acelasi obiect apare in ambele parti ale operatorului de asignare (cum ar fi s=s;) atunci sirul membru va fi sters. Pentru evitarea acestei probleme, putem folosi pointer-ul **this** pentru testarea acestui caz special dupa cum urmeaza:

```
String & String::operator = (const String& parte_dreapta)  {
  if (this == &parte_dreapta)
   return *this;
  else {
    delete []a;
    a = new char[strlen(parte_dreapta.a) + 1];
    strcpy(a, parte_dreapta.a);
    return *this;
  }
}
```

Exercitii propuse spre implementare

1. Scrieti un program C++ in care sa definiti o clasa Stiva (care sa descrie crearea unei stive, stergerea varfului stivei, test de stiva vida, adaugarea unui element in stiva (push), scoaterea unui element din stiva (pop), afisarea stivei). In vederea utilizarii stivei cu elemente de orice tip utilizati template-uri.

2. Folosind template, scrieti un program C++ care sa poata sorta siruri de intregi, caractere, double si alte tipuri. Puteti folosi metoda quicksort (cu si fara folosirea functiei predefinite qsort).