LUCRAREA DE LABORATOR NR7

Tema: Şabloane

Scopul lucrării:

- Studierea necesităţii şabloanelor;
- Studierea regulilor de definire şi utilizare a şabloanelor;
- Studierea specializării şabloanelor;
- Studierea potenţialelor probleme rezolvate cu ajutorul şabloanelor;

Noţiuni generale

Motivarea

Şabloanele reprezintă cea mai puternică construcţia a limbajului C++, dar în acelaşi timp, unul din cele mai puţin studiate şi rar utilizat. Cauza este ascunsă în faptul că el este foarte complicat şi are o sintaxă neobişnuită.

Aşa deci, şabloanele reprezintă prin sine un mecanism ce permite să scrii un algoritm, care nu este legat de un tip anumit. Cel mai des, şabloanele sunt utilizate pentru crearea container şi algoritmi abstracţi. Containerele sunt obiecte, care conţin alte obiecte sau date, cel mai des o cantitate nedeterminată, aşa cum sunt masivele, stivele, liste asociative, etc. Prin algoritm abstract este necesar de înţeles studierea bună a metodelor de prelucrare a datelor, ca sortarea, căutarea, etc., înscrise fără prezentarea tipului datelor.

Şabloanele sunt clase şi funcţii. Şabloanele au venit sa schimbe macrourile, aşa cum ultimele foarte des duc la găsirea complicată a erorilor, deoarece compilatorul nu verifică, dar nici nu are posibilitatea să le verifice de erori sintactice.

Programatorul, scriind şabloanele, creează aprovizionarea, care, ca urmare, se utilizează deja cu tipurile de date specificate. Adică, la baza şabloanelor compilatorul creează funcţii normale. Dacă şabloanele sunt utilizate cu câteva tipuri de date diferite, compilatorul creează un codul necesar pentru fiecare tip în parte. Cu alte cuvinte, şabloanele nu micşorează compilarea modulului, mai degrabă chiar invers, dar simţitor micşorează codul de ieşire, care duce la, micşorarea cantităţii de erori, micşorează introducerea modificărilor în cod şi micşorează prezentarea programelor în general, aşa cum se micşorează calitatea tipurilor şi funcţiilor date.

Definirea

Şabloanele sunt definite cu ajutorul cuvântului rezervat *template*:

```
template <typename T>
T& searchmax(T* ptr, int size)
{

template <typename T>
class Stack{
    T mas[10];
public:
...
};
```

Din exemplu se vede specificul definirii şablonului, pentru crearea lui este necesar de prezentat ЗС *template,* în paranteze unghiulare cuvântul *class*[1] şi un tip abstract, care se va utiliza în definirea şablonului. Istoric aşa sa întâmplat, că cel mai des, se utilizează identificatorul *T,* de la cuvântul *type*. Pentru clasă, la definirea funcţiilor în afara clasei, înainte de fiecare funcţie este necesar de scris, din nou, cuvântul *template*.

[1] În corespundere cu noile standarde, la fel poate fi utilizat ЗС "*typename*" în locul *"class"*.

Utilizarea

Funcţiile se utilizează practic aşa ca şi funcţiile obişnuite.

```
void main(){
 int    masi[10];
 float  masf[20];
 cout<<searchmax(masi,10);
 cout<<searchmax(masf,20);
}
```

Se vede că, sintaxa, apelării coincide cu cele obişnuite. Cunoscând tipurile parametrilor funcţiilor, compilatorul generează funcţii obişnuite. După care are posibilitatea să supraîncarce funcţia. În cazul claselor lucrurile sunt puţin mai complicate:

```
void main(){
 Stack<int>    sti;
 Stack<float>  stf;
}
```

Este necesar de prezentat tipul datelor, aşa cum, în acest caz, la această etapă de translare, compilatorul nu este în stare să determine, pentru care tip de date este necesar de generat codul.

Specializarea

Câte odată nu ne satisface lucrul şabloanelor pentru determinarea tipurilor datelor. Ca de exemplu:

```
template <class T>
T& max(T& a, T& b){
 if(a>b)return a;
 return b;
}
```

Acest exemplu lucrează excelent pentru tipurile încorporate, aşa ca int, float şi altele. Dar pentru şiruri – nu. Motivul constă în aceia, că în acest caz se vor compara pointerii la şiruri, dar nu conţinutul şirului. Pentru alte tipuri de date, posibil nu este definit operatorul >. Rezolvări pot fi câteva: se poate de interzis utilizarea pointerilor la şiruri şi să utilizezi tipul *String*, pentru care este definit operatorul >, atunci în acest caz se complică procesul de dezvoltare şi regulile de utilizare. Mai mult ca atât, interzicerea poate fi numai informativă, adică dacă utilizatorul ştie, că nu trebuie de utilizat pointeri. Însăşi limbajul nu are posibilitatea să interzică utilizarea specificării unui tip de date special. Altă rezolvare constă în utilizarea specializării, care reprezintă înscrierea încă a unei funcţii pentru un tip determinat. În cazul funcţiilor această de obicei nu este o funcţie şablon cu acelaşi nume şi cu parametri predefiniţi. Această funcţie poate avea un avantaj mai mare decât şabloanele. În cazul claselor se poate desigur de definit o clasă neşablon cu acelaşi nume, dar aceasta nu este interesant, aşa cum deosebirile pot fi minimale. În acest caz poate fi utilizat specializarea metodei clasei. Specializat poate fi numai metoda definit în afara clasei. Ca de exemplu:

```
template <class T>
class Stack{
public:
  void push(T& t);
  void sort();
friend ostream& operator<<(ostream& os, Stack<T>& s);
};
template <class T>
void Stack<T>::sort(){
...
// aici se înscrie un algoritm abstract
}
```

```
        void Stack<char*>::sort(){
        ... // dar aici unul specializat
        }
        template <class T>
        ostream& operator<<(ostream& os, Stack<T>& s){
           return os; // afişarea conţinutului stivei
         }
        void main(){
         Stack<int> si;
         si.push(5);
         Stack<char*> sc;
         sc.push("Hello");
         si.sort();    // Apelarea funcţiei abstracte
         sc.sort();    //Apelarea funcţiei specializate
         cout<<si<<sc;
// Apelarea operatorului de supraîncărcare a fluxului de ieşire
        }
```

Şabloanele clasei pot fi moştenite, aşa ca şi cele obişnuite, cu aceasta, şi acel de bază, aşa şi a cel derivat pot fi clase obişnuite.

```
class One{
};
template <class T>
class Two: public One{
};
template <class T>
class Three: public Two<T>{
};
class Four: public Three<int>{
};
template <class T>
class Five: public T{
};
```

Un interes deosebit reprezintă prin sine ultimul tip de moştenire, aşa cum are loc moştenirea de la parametrii şablonului. În acest caz *T* desigur nu trebuie să fie clasă ori structură.

Cum se vede din cele relatate mai sus, şabloanele reprezintă prin sine un mecanism interesant şi puternic. Sunt greutăţi şi complicaţii în studierea lor, dar ele se răscumpără, aşa cum permit crearea unui cod frumos şi compact. Mai mult ca atât, şabloanele nu sunt realizate în alte limbaje de programare moderne utilizate pe larg, dar la fel permit realizarea unor noi rezolvări tehnice, aşa cum pointeri deştepţi, susţinerea tranzacţiei, dirijarea memoriei, etc.

Întrebări de control:

1. Ce reprezintă prin sine şabloanele?
2. Care sunt avantajele utilizării şabloanelor?
3. Pentru ce se utilizează specializarea?
4. Ce poate fi un şablon?
5. Care sunt problemele utilizării şabloanelor?
6. Utilizarea şabloanelor duce la economia memoriei?
7. Dar a timpului de executare a compilării codului?
8. Poate o clasă şablon să moştenească una obişnuită şi invers?

Sarcina
Varianta 1

a) Creaţi o funcţie şablon, care schimbă ordinea elementelor în felul următor: prima parte a listei se amestecă la urmă, dar a doua la început. De exemplu: 1 2 3 4 5 6 - 4 5 6 1 2 3. Funcţia trebuie să lucreze cu masive de lungimi diferite. Dacă numărul de elemente este impar, atunci elementul mijlociu nu trebuie de prelucrat.

b) Creaţi clasa parametrizată *Stack*. Clasa trebuie să conţină constructorii, destructorii, şi deasemenea funcţiile *push, pop, empty, full* şi operatorii de intrare/ieşire. Pentru alocarea memoriei să se utilizeze operatorul *new*.

Varianta 2

a) Creaţi o funcţie şablon, care schimbă ordinea elementelor după perechi. De exemplu: 1 2 3 4 5 6 - 2 1 4 3 6 5. Funcţia trebuie să lucreze cu masive de lungimi diferite. Dacă numărul de elemente este impar, atunci ultemul element nu trebuie de prelucrat.

b) Creaţi clasa parametrizată *Vector*. Clasa trebuie să conţină constructorii, destructorii, şi deasemenea funcţiile *getLength*, operatorii *[]*, +, - şi operatorii de intrare/ieşire. Pentru alocarea memoriei să se utilizeze operatorul *new*.

Varianta 3

a) Creaţi o funcţie şablon, care calculează cantitatea de repetări a unui parametru în listă. De exemplu: lista - 0 2 3 4 3 6, parametrul - 3, rezultatul - 2. Funcţia trebuie să lucreze cu masive de lungimi diferite.

b) Creaţi clasele parametrizate *List* şi *ListItem*. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *add, in, remove, getLength*, operatorii *[]* şi operatorii de intrare/ieşire.

Varianta 4

a) Creaţi o funcţie şablon, care caută o cheie dată. Funcţia întoarce poziţia primului element întâlnit. De exemplu: lista - 0 2 3 4 3 6, parametrul - 2, rezultatul - 1. În cazul când lipseşte elementul necesar să se întoarcă codul erorii. Funcţia trebuie să lucreze cu masive de lungimi diferite.

b) Creaţi clasa parametrizată *Queue* - coadă. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *add, in, get, getLength*, operatorii *[]* şi operatorii de intrare/ieşire.

Varianta 5

a) Creaţi o funcţie şablon, care caută al doilea element după dimensiune în elementele listei. De exemplu: lista - 0 2 3 4 3 6, rezultatul - 4. Funcţia trebuie să lucreze cu masive de lungimi diferite.

b) Creaţi clasa parametrizată *Set* - mulţimea. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *add, in, remove*, operatorii "+", "*","-" şi operatorii de intrare/ieşire.

Varianta 6

a) Creaţi o funcţie şablon, de sortare în creştere după metoda bulelor. Funcţia trebuie să lucreze cu masive de lungimi diferite.

b) Creaţi clasa parametrizată *Map* – listă asociativă, care conţine cheia câmpului şi valoarea. Unei chei îi corespunde o valoarea. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *add, removeByKey, getLength, getByKey, getByValue*, operatorii *[]* şi operatorii de intrare/ieşire.

Varianta 7

a) Creaţi o funcţie şablon, care schimbă după perechi elementele masivelor în felul următor: primul element va avea valoarea sumei perechii, dar al doilea diferenţei perechii. De exemplu: lista- 0 2 3 4 3 6, rezultatul 2 –2 7 –1 9 –3.

b) Creaţi clasa parametrizată *Matrix* – matrice. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *getRows, getCols,* operatorii *[],* +, -, * şi operatorii de intrare/ieşire.

Varianta 8

a) Creaţi o funcţie şablon, care modifică ordinea elementelor invers. De exemplu: 1 2 3 4 5 6 - 6 5 4 3 2 1. Funcţia trebuie să lucreze cu masive de lungimi diferite.

b) Creaţi clasa parametrizată *Tree* – arbore binar. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *add, in,* funcţia de eludare a arborelului şi operatorii de intrare/ieşire.

Varianta 9

a) Creaţi o funcţie şablon, care calculează cantitatea de elemente, valoarea cărora este mai mare decât a parametrului dat. De exemplu: lista - 0 2 3 4 3 6, parametrul - 5, rezultatul - 1. Funcţia trebuie să lucreze cu masive de lungimi diferite.

b) Creaţi clasa parametrizată *Stack*. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *push, pop, empty* şi operatorii de intrare/ieşire. Pentru alocarea memoriei să se utilizeze operatorul *new,* pe *n* elemente.

Varianta 10

a) Creaţi o funcţie şablon, de sortare a elementelor unui masiv în descreştere prin metoda de introducere. Funcţia trebuie să lucreze cu masive de lungimi diferite.

b) Creaţi clasa parametrizată *MultiMap* – listă multi-asociativă, care conţine cheia câmpurilor şi lista de valori. Adică, unei chei pot sa-i aparţină mai multe valori. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *add, removeByKey, getLength, getByKey, getByValue,* şi operatorii *[]* şi de intrare/ieşire.

Varianta 11

a) Creaţi o funcţie şablon, care calculează cantitatea de elemente, valoarea cărora este mai mică decît a unui element dat. De exemplu: lista - 0 2 6 4 3 3, parametrul - 3, rezultatul - 2. Funcţia trebuie să lucreze cu masive de lungimi diferite.

b) Creaţi clasa parametrizată *PriorityQueue* – coadă cu prioritate. Fiecare element al cozii conţine o prioritate definită şi la extragerea se alege un element cu un prioritate mai mare, dar elementele cu priorităţi asemănătoare – după timpul adăugării. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *add, in, get, getLength*, şi operatorii *[]* şi de intrare/ieşire.

Varianta 12

a) Creaţi o funcţie şablon, de căutare a elementului al doilea minimal după mărime din elementele listei. De exemplu: lista - 0 2 3 4 3 6, rezultatul - 2. Funcţia trebuie să lucreze cu masive de lungimi diferite.

b) Creaţi clasa parametrizată *Matrix* – matrice. Clasa trebuie să conţină constructorii, destructorii şi funcţiile *getRows, getCols,* operatorii *[],* +=, -=, *= şi de intrare/ieşire.