11.Šabloni

Nodaļas saturs:

11.1.Šablons kā mehānisms

11.1.1.Šablonu vispārīgs apraksts

Blakus daudzkāršai mantošanai un operatoru pārslogošanai **šabloni** jeb **veidnes** (*templates*) ir vēl viens mehānisms, kas nav sastopams daudzās citās objektorientētās programmēšanas valodās. Šabloni nodrošina pirmkoda vairākkārtēju izmantošana pēc nedaudz līdzīga principa kā mantošana, tomēr dod iespēju izveidot daudz efektīvākas konstrukcijas nekā, izmantojot citas metodes.

Šablonu mehānisms ir ļoti spēcīgs, bet tajā pašā laikā arī ļoti sarežģīts. Šī iemesla dēļ tas valodā C++ nav ieviests no pašiem pirmsākumiem – galvenokārt realizācijas problēmu dēļ kompilatoros. Savukārt kopš pagājušā gadsimta beigām, kad kompilatori bija kļuvuši pietiekoši inteliģenti, šablonu pielietojums strauji palielinājās, un tagad tie ir pamatā ļoti lielai daļai standarta bibliotēku.

Piemērs šablonu izmantošanai standarta bibliotēkās ir t.s. **standarta šablonu bibliotēka** (*standard template library*, ***STL***), kas ietver sevī daudzu datu struktūru un algoritmu realizācijas. Tajā pat laikā jaunajās realizācijās arī ievada un izvada bibliotēkas (tādas kā *iostream*, *fstream*) ir būvētas uz šablonu mehānisma bāzes.

Praktiski veidojot programmas, programmētājiem visbiežāk nāksies izmantot dažādas standarta bibliotēkas uz šablonu bāzes, nevis pašam veidot šablonus, tomēr paša šablonu mehānisma iepazīšana noteikti palīdz efektīvāk izmantot šīs bibliotēkas.

Šablons ir mehānisms, kas reprezentē shēmu, pēc kuras var tikt veidotas klases vai funkcijas.

No viena šablona veidotās konstrukcijas savā starpā atšķiras ar atšķirīgu datu tipu komplektu – līdzīgi kā atšķiras divas funkcijas ar vienādu nosaukumu un parametru skaitu, bet atšķirīgiem parametru tipiem.

Klases šablona piemērs varētu būt “kastīte kaut kam”, bet klases, kas no šī šablona varētu tikt veidotas, būtu, piemēram, “kastīte gurķiem” vai “kastīte tomātiem”.

Šablonu mehānisms ir pielietojams divās formās:

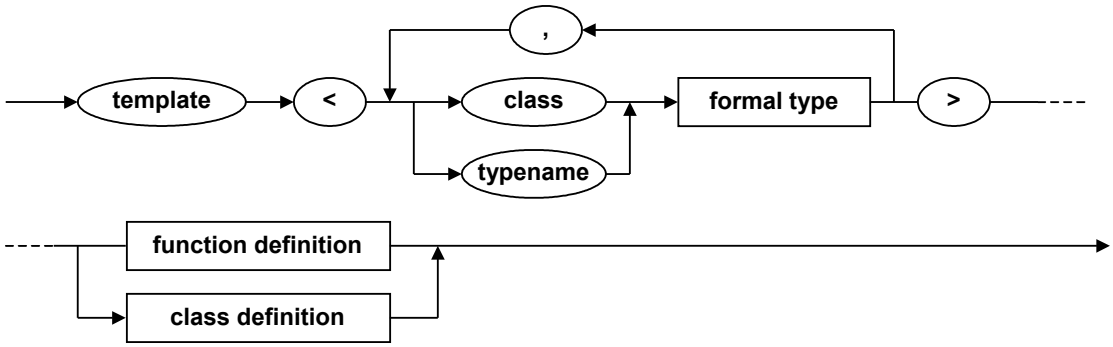
- funkciju šabloni,
- klašu šabloni.

No viena šablona veidotās klases atšķiras pēc pielietotā datu tipu komplekta, tādējādi datu tipi ir šablona parametru vērtības, un klašu šablonus sauc arī par **parametrizētajiem tipiem**. Plašs klašu šablonu pielietojums ir dažādu datu struktūru realizācijā, kuru elementi var būt dažādu tipu vērtības, tāpēc klašu šabloni tiek saukti arī par **konteineru klasēm**.

Šablons programmas tekstā no parastas funkcijas/ klases atšķiras pēc 2 pazīmēm:

- virs funkcijas/klases ir papildus rindiņa – šablona galva, kas deklarē, ka dotā funkcija/klase ir šablons, un satur formālo tipu sarakstu,
- funkcija/klase blakus parastiem tipiem (kā parametru, lauku vai lokālo mainīgo tipiem) satur arī formālos tipus, respektīvi, identifikatorus, kas vēlāk būs jāprecizē par konkrētiem tipiem.

Sintakse 11.1. ***general scheme of template*** **(šablona vispārēja shēma)**



Pēc sintaktiskās diagrammas 11.1 redzams, ka pirms katra formālā tipa ir vai nu atslēgas vārds *class* vai *typename*. Šie abi atslēgas vārdi dotajā kontekstā ir sinonīmi – abi ir pielietojami gan funkciju, gan klašu šablonu veidošanai.

11.1.2.Funkciju šabloni

Funkciju šablons ir shēma funkciju veidošanai. Funkciju šablons ir vienkāršākais piemērs šablonu mehānisma demonstrēšanai.

Pirmkoda piemērā 11.1 parādīta funkciju šablona *add()* izveidošana un izmantošana:

- funkciju šablona izveidošana ir līdzīga parastas funkcijas izveidošanai ar nekonkretizētiem datu tipiem, kam pievienota šablona galva (rindas 4-8),
- funkciju šablona izmantošana pilnīgi neatšķiras no parastas funkcijas izmantošanas (rindas 14,15).

Jāatceras, ka funkciju šablons nav universāla funkcija (kaut arī no izmantošanas viedokļa tā varētu likties), bet shēma funkciju izveidošanā, t.i. uz tā bāzes tiek izveidots tik daudz "īstu" funkciju, cik nepieciešams. Pirmkoda piemērā 11.1 uz šablona bāzes tiek izveidotas divas funkcijas, pie kam kompilators, sastopot to izsaukumus (rindās 14,15), pats automātiski noskaidro, kādas:

```
int add (int a, int b)
string add (string a, string b)
```

No tā izriet, ka

- funkciju šablona (arī klašu šablona) izmantošana nodrošina efektīva izpildāmā koda ģenerēšanu, jo katram gadījumam tiek izveidota sava funkcija (vai klase) (nevis viena universāla),
- tajā pašā laikā šabloni šī paša iemesla dēļ nenodrošina kompakta izpildāmā koda ģenerēšanu.

Lai šablonu varētu izmantot noteiktam tipu komplektam, dotajiem tipiem ir jābūt definētām visām darbībām kas pielietotas šablonā (piemēram, pirmkodā 11.1 funkciju šablonu *add()* var izmantot tipiem, kam definēta saskaitīšanas operācija – sk. rindu 7; tipi *int* un *string* tādi ir – rindas 14,15).

Pirmkods 11.1. Funkciju šablons (*tpl1fun.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 template <typename T>
05 T add (T a, T b)
06 {
07     return a + b;
08 };
09
10 int main ()
11 {
12     int x=2, y=3, z;
13     string s="Hello, ", t="World!", u;
14     z = add (x, y);
15     u = add (s, t);
16     cout << z << endl;
17     cout << u << endl;
18     return 0;
19 }
```

Programmas darbības piemērs:

```
5
Hello, World!
```

11.1.3.Klašu šabloni

Klašu šablona izveidošana notiek pēc līdzīgiem principiem, kā funkciju šablona izveidošana – šablonu klasi ievada identiska šablona galva, un klases definēšanā tiek izmantoti arī nekonkretizētie tipi, tomēr klašu šablona izmantošana objektu veidošanai ir sarežģītāka nekā funkciju šabloniem – kompilators pēc konteksta nevar noteikt, ar kādu aktuālo tipu komplektu jāveido objekts, tāpēc tas jāuzrāda tiešā veidā (pirmkoda piemērs 11.2, rindas 23,25).

Aktuālo tipu komplekta uzrādīšana pie klases objekta deklarēšanas var pasliktināt programmas lasāmību, bet izeja ir konstrukcijas *typedef* lietošana (pirmkoda piemērs 11.2, rinda 19), kas ļauj no klases šablona izveidoto klasi nosaukt ar konkrētu vārdu un tādējādi padarīt tālāko šīs klases izmantošanu baudāmāku (pirmkoda piemērs 11.2, rinda 27: precizētā klase *person*).

Pirmkoda piemērā 11.2 parādīta klašu šablona *two* izveidošana un izmantošana:

- klašu šablona izveidošana ir līdzīga parastas klases izveidošanai ar nekonkretizētiem datu tipiem, kam pievienota šablona galva (rindas 4-18),
- deklarējot mainīgo, kas reprezentēs kādas šablonu klases objektu, ir jāuzrāda aktuālo tipu komplekts (rindas 23,25), kas ir klašu šablonu izmantošanas atšķirība no funkciju šabloniem. Šajā gadījumā programmas lasāmībai ērti izmantot konstrukciju *typedef* (rinda 19).
- Pēc tam, kad deklarēts šablonu klasi reprezentējošs mainīgais, šablonu klases izmantošana neatšķiras no parastas klases izmantošanas.

Pirmkoda piemērā 11.2 uz šablona bāzes tiek izveidotas trīs dažādas klases:

```
two<int,int>
two<string,string>
two<string,int>
```

Klases šablons *two* nodrošina tādu klašu veidošanu, kas satur divas dažādu tipu vērtības un prot tās izdrukāt.

Lai šablonu varētu izmantot noteiktam tipu komplektam, dotajiem tipiem ir jābūt definētām visām darbībām kas pielietotas šablonā (piemēram, pirmkodā 11.2 klašušablonu *two* var izmantot tipiem, kam pārdefinēts izdrukas operators pie klases *ostream* – sk. rindu 16; tipi *int* un *string* tādi ir – rindas 24,26,28, kur tiek izmantota funkcija *print()*).

Pirmkods 11.2. Klašu šablons (*tpl2class.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 template <class T1, class T2>
05 struct two
06 {
07     T1 first;
08     T2 second;
09     two (const T1 &f, const T2 &s)
10     {
11         first = f;
12         second = s;
13     };
14     void print ()
15     {
16         cout << first << " " << second << endl;
17     }
18 };
19 typedef two<string,int> person;
20
21 int main ()
22 {
23     two<int,int> t (3,4);
24     t.print ();
25     two<string,string> s ("Hello,", "World!");
26     s.print ();
27     person p ("Liz", 19);
28     p.print ();
29     return 0;
30 }
```

Programmas darbības piemērs:

```
3 4
Hello, World!
Liz 19
```

Pirmkoda piemērā 11.3 definētais šablons *dynamicarray* ļauj izveidot vienkāršotus dinamiskus masīvus dažādiem datu tipiem, kur atmiņas vadība ir paslēpta šablona konstrukcijā. Bez konstruktora un destruktora dinamismu nodrošina arī funkcija append(), kas pievieno masīvam jaunu elementu galā, automātiski to palielinot.

Pirmkods 11.3. Dinamiska masīvs kā klašu šablons (*tpl3array.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 template <class T>
05 class dynamicarray
06 {
07     int Size;
08     T *Array;
09 public:
10     dynamicarray (int s=0)
11     {
12         Size = s;
13         if (Size > 0) Array = new T[Size];
14     };
15     ~dynamicarray ()
16     {
17         if (Size > 0) delete[] Array;
18     };
19     void append (const T &value)
20     {
21         T *tmparray = new T[Size+1];
22         for (int i=0; i<Size; i++) tmparray[i] = Array[i];
23         tmparray[Size] = value;
24         Size++;
25         Array = tmparray;
26     };
27     void print ()
28     {
29         for (int i=0; i<Size; i++) cout << Array[i] << endl;
30     };
31     T& operator[] (int i) { return Array[i]; };
32 };
33 typedef dynamicarray<int> intarray;
34 typedef dynamicarray<string> stringarray;
35
36 int main ()
37 {
38     intarray arri(2);
39     arri[0] = 11;
40     arri[1] = 22;
41     arri.append (33);
42     arri.print ();
43     stringarray arrs;
44     arrs.append ("Hello,");
45     arrs.append ("Word!");
46     arrs.print ();
47     return 0;
48 }
```

Programmas darbības piemērs:

```
11
22
33
Hello,
```

```
Word!
```

11.2.Standarta šablonu bibliotēka (STL)

11.2.1.STL vispārīgs apraksts

Standarta šablonu bibliotēkas (*standard template library*, ***STL***) izveidošana bija veiksmīgs solis C++ bibliotēku standartizēšanā, jo ilgu laiku praktiski vienīgās C++ standarta bibliotēkas bija tās, kas bija mantotas no C (ja neskaita ievada un izvada bibliotēkas).

STL sastāvā ietilpst 3 komponentu kategorijas:

- **Konteineri** – tās ir datu struktūras, kas veido STL pamatu (piemēram, *vector*, *stack*), un pārējo divu kategoriju komponentes nodarbojas ar to apkalpošanu.
- **Iteratori** – specializētas norādes piekļūšanai konteineru klašu vērtībām.
- **Algoritmi** – funkciju šabloni, kas apkalpo konteineru datus (piemēram, *sort()*).

STL konteineru klases.

STL satur daudzas konteineru klases, kas reprezentē vispārzināmas datu struktūras. Tas ir ļoti spēcīgs līdzeklis, tomēr prasa pietiekoši plašas zināšanas algoritmos un datu struktūrās, lai tās izmantotu "pēc pilnas programmas". Tabulā 11.1 parādītas STL pieejamās konteineru klases, dažas no kurām tiks nedaudz apskatītas vēlāk.

Tabula 11.1. STL konteineru klases

Iekļaujamais bibliotēkas fails	Konteinera klases	Apraksts
<bitset>	`bitset`	bitu kopas
<vector>	`vector`	dinamiskie masīvi
<list>	`list`	saraksti
<map>	`map` `multimap`	kartes vai attēlojumi – līdzīgi masīviem, bet indeksa lomu šeit var spēlēt arī citu tipu vērtības
<stack>	`stack`	steki – lineāra struktūras, kas organizēta pēc principa "pēdējais ienācis, pirmais izgājis" (LIFO)
<queue>	`queue` `priority_queue`	rindas – lineāra struktūras, kas organizēta pēc principa "pirmais ienācis, pirmais izgājis" (FIFO)
<deque>	`deque`	divvirzienu rindas
<set>	`set` `multiset`	kopas

STL iteratori.

Iteratori ir mehānismi, kas paredzēti piekļūšanai konteineru elementiem, kas lielā mērā (gan idejiski, gan sintaktiski) atgādina norādes (*pointers*). Iteratorus deklarē, izmantojot tipu *iterator*, kas definēts konteineru klasēs (nevis, pielietojot papildus * simbolu, kā tas pierasts pie norādēm). Divi svarīgi daudzu konteineru klašu iebūvētie iteratori ir *begin()* un *end()*, kas norāda attiecīgi uz konteinera pirmo elementu un elementu aiz pēdējā (pēc līdzības ar beigu simbolu zema līmeņa simbolu virknēs). Iteratori var būt vairāku tipu – patvaļīgas pieejas (*random access*), divvirzienu (*bidirectional*) u.c., un to, kāds iterators tiek izmantots, nosaka konteineru klases tips un konteksts. Iteratoru darbības princips tiks demonstrēts, apskatot konkrētas konteineru klases.

STL algoritmi.

STL piedāvā vairāku algoritmu realizācijas, kas apkalpo konteineru klases. Šajā mācību materiālā tiks apskatīta kārtošanas algoritma *sort()* izmantošana.

11.2.2.STL klase 'pair'

STL klase *pair* nodrošina divu dažādu vērtību glabāšanu vienā struktūrā, kas atgādina klasi *two*, kas definēta pirmkoda piemērā 11.2.

Klase *pair* ietver divus laukus: *first* un *second*, un to izmanto arī citi konteineri, piemēram, *map*, kas tiks apskatīts šajā materiālā.

Pirmkoda piemērs 11.4 demonstrē konteinera *pair* izmantošanu.

Pirmkods 11.4. STL klase 'pair' (*tpl4pair.cpp*)

```
01 #include <iostream>
02 using namespace std;
03
04 typedef pair<string,int> person;
05
06 int main ()
07 {
08     person p ("Liz", 19);
09     cout << p.first << " " << p.second << endl;
10     return 0;
11 }
```

Programmas darbības piemērs:

```
Liz 19
```

11.2.3.STL klase 'vector'

STL klase *vector* (vektors) realizē dinamisku masīvu ar daudzām papildus īpašībām, kas atgādina klasi *dynamicarray*, kas definēta pirmkoda piemērā 11.3.

Klase *vector* ir viena no populārākajām STL konteineru klasēm. Tās izmantošanas ērtums balstās uz piekļuves iespēju elementu vērtībām, izmantojot indeksu. Tajā pat laikā jāņem vērā, ka, lai nodrošinātu piekļuvi pēc indeksa, šajā struktūrā nepieciešami mehānismi, kas prasa papildus skaitļošanas un atmiņas resursus šī servisa nodrošināšanai.

Pirmkoda piemērs 11.5 parāda, ka piekļuve vektora elementiem iespējama, lietojot gan pierasto masīva sintaksi (rindas 10,11,13), gan iteratorus (rindas 17-22) (sk. arī attēlu 11.4). Pirmkoda piemērā 11.5 izmantoto klases *vector* īpašību apraksts:

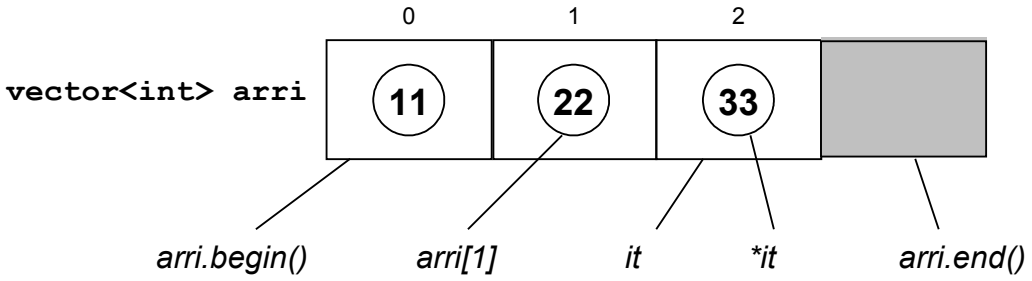
- *begin()* – iterators (norāde) uz pirmo elementu,
- *end()* – iterators (norāde) uz virtuālu elementu aiz pēdējā,
- *size()* – funkcija, kas atgriež vektora garumu,
- *push_back()* – funkcija, kas pievieno vektoram elementu tā beigās,
- *vector<T>::iterator* – vektora iteratora tips,
- *it++* – iteratora pāreja uz nākošo elementu,
- *[]* – piekļuve vektora elementa vērtībai (nevis elementam!) pēc indeksa,
- **it* – piekļuve vektora elementa vērtībai, izmantojot iteratoru.

Pirmkods 11.5. STL klase 'vector' (*tpl5vector.cpp*)

```
01 #include <vector>
02 #include <iostream>
03 using namespace std;
04
05 typedef vector<int> intarray;
06
07 int main ()
08 {
09     intarray arri(2);
10     arri[0] = 11;
11     arri[1] = 22;
12     arri.push_back (33);
13     for (int i=0; i<arri.size(); i++) cout << arri[i] << endl;
14     vector<string> arrs;
15     arrs.push_back ("Hello,");
16     arrs.push_back ("Word!");
17     vector<string>::iterator it = arrs.begin ();
18     while (it != arrs.end ())
19     {
20         cout << *it << endl;
21         it++;
22     };
23     return 0;
24 }
```

Programmas darbības piemērs:

```
11
22
33
Hello,
Word!
```



Attēls 11.4. STL klase 'vector' un piekļūšana tās elementiem un to vērtībām (atbilstoši pirmkoda piemēram 11.5)

11.2.4.STL klase 'list'

STL klase *list* (saraksts) ir līdzīga klasei *vector* ar to atšķirību, ka tajā nav pieejama piekļuve elementu vērtībām, izmantojot indeksu. Tomēr, ja tas nav nepieciešams, tad labāk vektora vietā lietot tieši sarakstu, jo šī konstrukcija ir mazāk prasīga pēc resursiem struktūras izmēra maiņas gadījumā (piemēram, pievienojot jaunu elementu). Tai pat laikā klasē *list* ir pieejamas daudzas ļoti vērtīgas metodes, piemēram, elementa iespraušana.

Vienkāršots klases *list* piemērs redzams pirmkoda piemērā 11.6. Šajā piemērā izmantoto klases *list* īpašību apraksts:

- *begin()* – iterators (norāde) uz pirmo elementu,
- *end()* – iterators (norāde) uz virtuālu elementu aiz pēdējā,
- *push_back()* – funkcija, kas pievieno sarakstam elementu tā beigās,
- *list<T>::iterator* – saraksta iteratora tips,
- *iti++, its++* – iteratora pāreja uz nākošo elementu,
- **iti, *its* – piekļuve vektora elementa vērtībai, izmantojot iteratoru.

Pirmkods 11.6. STL klase 'list' (*tpl4list.cpp*)

```
01 #include <list>
02 #include <iostream>
03 using namespace std;
04
05 typedef list<int> intlist;
06 typedef list<string> stringlist;
07
08 int main ()
09 {
10     intlist arri;
11     arri.push_back (11);
12     arri.push_back (22);
13     arri.push_back (33);
14     intlist::iterator iti = arri.begin ();
15     while (iti != arri.end ())
16     {
17         cout << *iti << endl;
18         iti++;
19     };
20     stringlist arrs;
21     arrs.push_back ("Hello,");
22     arrs.push_back ("Word!");
23     stringlist::iterator its = arrs.begin ();
24     while (its != arrs.end ())
25     {
26         cout << *its << endl;
27         its++;
28     };
29     return 0;
30 }
```

Programmas darbības piemērs:

```
11
22
33
Hello,
Word!
```

11.2.5.STL algoritms 'sort'

Viens no redzamākajiem STL algoritmiem ir kārtošanas algoritms *sort()*. Ar to var kārtot ne tikai vektora, bet arī citu struktūru (piemēram, saraksta) elementus.

Pirmkoda piemērs 11.7 parāda kārtošanas algoritma izmantošanu divos veidos:

- Standarta variants, izmantojot standarta salīdzināšanas ("mazāk") kritēriju, kāds pieejams struktūrā glabājamiem objektiem (rinda 25) – šajā gadījumā skaitļi tiks sakārtoti augošā secībā.
- Specializētais variants (rinda 28), izmantojot speciāli izveidotu salīdzināšanas kritēriju (rindas 8-15) – šajā gadījumā skaitļi tiks sakārtoti augošā secībā pēc pēdējā cipara.

Kārtošanas algoritma izmantošana standarta variantā.

Tiek izsaukta funkcija *sort()* ar diviem parametriem – (1)sākuma iterators un (2)beigu iterators (jāatceras, ka beigu iteratoram jānorāda nevis uz beidzamo kārtojamo elementu, bet nākošo aiz tā).

Kārtošanas algoritma izmantošana specializētajā variantā.

Tiek izsaukta funkcija *sort()* ar trīs parametriem – (1)sākuma iterators, (2)beigu iterators un (3) salīdzināšanas kritērijs.

Salīdzināšanas kritērijs ir funkcija, kas par diviem objektiem, kādus var glabāt dotā struktūru, pasaka – vai pirmais ir mazāks par otro. Funkcija, kas definē salīdzināšanas kritēriju, ir jānoformē noteiktā formātā – kā speciāli izveidotas klases operatora *operator()* pārdefinēšana (pārslogošana). Kad šāda klase izveidota, tad, izsaucot kārtošanas funkciju *sort()*, trešajā parametrā tiek padots klases nosaukums ar tukšām iekavām.

Pirmkods 11.7. STL algoritms 'sort' (*tpl7sort.cpp*)

```
01 #include <algorithm>
02 #include <vector>
03 #include <iostream>
04 using namespace std;
05
06 typedef vector<int> intarray;
07
08 class lastdigit_compare
09 {
10 public:
11     bool operator() (const int &v1, const int &v2) const
12     {
13         return v1%10 < v2%10;
14     }
15 };
16
17 int main ()
18 {
19     intarray arri(4);
20     arri[0] = 52;
21     arri[1] = 28;
22     arri[2] = 47;
23     arri[3] = 35;
24     for (int i=0; i<arri.size(); i++) cout << arri[i] << endl;
25     sort (arri.begin(), arri.end());
26     cout << endl;
```

```
27      for (int i=0; i<arri.size(); i++) cout << arri[i] << endl;
28      sort (arri.begin(), arri.end(), lastdigit_compare());
29      cout << endl;
30      for (int i=0; i<arri.size(); i++) cout << arri[i] << endl;
31      return 0;
32  }
```

Programmas darbības piemērs:

```
52
28
47
35

28
35
47
52

52
35
47
28
```

Pirmkoda piemērs 11.7 demonstrē šādas darbības:

- skaitļu masīva izveidošana un izdrukāšana,
- skaitļu masīva sakārtošana augošā secībā un izdrukāšana,
- skaitļu masīva sakārtošana augošā secībā pēc pēdējā cipara un izdrukāšana.

11.2.6.STL klase ‘map’

STL klase *map* (karte) nodrošina vērtību pāru glabāšanu, kā arī piekļūšanu elementiem pēc pirmās vērtības. No izmantošanas viedokļa karte līdzinās vārdnīcai. Ja elementa pirmās vērtības tips ir *int*, tad *map* praktiski nodrošina vektora funkcionalitāti. Pievienojot jaunu elementu kartei, tiek automātiski nodrošināta *map* elementu sakārtošana pēc elementu pirmajām vērtībām. Tā kā klase *map* nodrošina vērtību pāru glabāšanu, tā izmanto STL klasi *pair* viena elementa vērtību glabāšanai (sk. pirmkoda piemēru 11.8, rindu 18).

Vienkāršots klases *map* piemērs redzams pirmkoda piemērā 11.8. Šajā piemērā izmantoto klases *map*īpašību apraksts:

- *begin()* – iterators (norāde) uz pirmo elementu,
- *end()* – iterators (norāde) uz virtuālu elementu aiz pēdējā,
- *[]* – piekļuve elementa vērtībai (struktūrai *pair*) pēc pirmās vērtības; vai elementa pievienošana struktūrai, automātiski nodrošinot tās sakārtošanu,
- *map<T1,T2>::iterator* – kartes iteratora tips,
- *it++* – iteratora pāreja uz nākošo elementu,
- *it->first, *it.first* – piekļuve elementa pirmajai vērtībai, izmantojot iteratoru,
- *it->second, *it.second* – piekļuve elementa otrajai vērtībai, izmantojot iteratoru,
- *find()* – funkcija, kas atgriež iteratoru uz elementu ar doto pirmo vērtību (ja tāds elements neeksistē, tad atgriež iteratoru *end()*).

Pirmkods 11.8. STL klase 'map' (*tpl8map.cpp*)

```
01 #include <map>
02 #include <iostream>
03 using namespace std;
04
05 typedef map<string,string> dictionary;
06 typedef dictionary::iterator dictit;
07
08 int main ()
09 {
10     dictionary d;
11     d["day"] = "giorno";
12     d["street"] = "strada";
13     d["italian"] = "italiano";
14     d["red"] = "rosso";
15     dictit it = d.begin();
16     while (it != d.end())
17     {
18         cout << it->first << " " << it->second << endl;
19         it++;
20     };
21     cout << "FIND:" << endl;
22     it = d.find ("red");
23     if (it != d.end()) cout << it->first << " " << it->second <<
                endl;
24     else cout << "NOT FOUND";
25     it = d.find ("blue");
26     if (it != d.end()) cout << it->first << " " << it->second <<
                endl;
27     else cout << "NOT FOUND";
28     return 0;
29 }
```

Programmas darbības piemērs:

```
day giorno
italian italiano
red rosso
street strada
FIND:
red rosso
NOT FOUND
```

Pirmkoda piemērs 11.8 demonstrē šādas darbības:

- aizpilda karti *dictionary* ar dažām angļu-itāļu vārdnīcas vērtībām un izdrukā,
- mēģina atrast elementu ar pirmo vērtību "red" – atrod un izdrukā to,
- mēģina atrast elementu ar pirmo vērtību "blue" – neatrod un izdrukā "NOT FOUND".