STL KONTEINERIAI

Sąrašai.

STL konteinerinė klasė *list* apibrėžta vardų srities *std* antraštiniame faile <*list*>. Ši klasė įgyvendina dviejų krypčių sąrašą, kurio pagrindinė paskirtis – suteikti programuotojui tokį nuoseklų konteinerį, kuriame būtų labai efektyviai vykdomos elementų įterpimo ir šalinimo operacijos (į bet kurią poziciją / iš bet kurios pozicijos). Sąrašo klasė turi beveik visas vektoriaus klasės funkcijas ir dar papildomas funkcijas. Skirtingai nuo vektoriaus sąraše nėra realizuotas tik elementų indeksavimo metodas, tai padaryta būtent dėl prieigos efektyvumo užtikrinimo.

Papildomos sąrašo funkcijos

void push_front(const T& x); ir *void pop_front();*

yra analogiškos jau minėtoms funkcijoms *push_back()* ir *pop_back(),* bet jos prideda ir šalina elementus į sąrašo pradžią (iš pradžios), o funkcija

void remove(const T& value);

šalina elementą iš sąrašo.

Surūšiuoti sąrašą galima funkcija

void sort();

kuri rūšiuoja sąrašą pagal operatorinės funkcijos *operator<()* užduodamą tvarką. Sąrašo sukūrimui galima naudoti kelių pavidalų konstruktorius. Sąrašas neturintis elementų kuriamas konstruktoriumi

explicit list(const Allocator& alloc = Allocator());

o sąrašas iš *n* elementų, turintis T tipo objektų reikšmių (pagal nutylėjimą) kopijas, konstruktoriumi

explicit list(size_type n);

Tipas T turi turėti konstruktorių pagal nutylėjimą. Konstruktorius

list(size_type n, const T& value, const Allocator& alloc = Allocator());

sukurs *n* elementų sąrašą, kuriame būtų T tipo objektų reikšmės *value* kopijos. Visais atvejais atminties skirstymui naudojamas objektas *alloc*.

Pavyzdys 3.30. STL sąrašo sukūrimas ir rūšiavimas.

```
#include <iostream>
#include <list>
#include <cstdlib>
```

```
using namespace std;

typedef list<char> CList;
int main()
{
	CList clist;
	for (int i=0; i<10; i++)
		clist. push_back('A' + (rand() % 26));
	 cout<<"Pradinis sąrašas:  ";

	CList::iterator iter =clist.begin();
	while (iter != clist.end())
	{
		cout<< *iter;
		iter++;
	}
      cout<<"\n\n" ;

	clist.sort();
	cout<<"Surūšiuotas sąrašas:  ";
	iter =clist.begin();
	while (iter != clist.end())
	{
		cout<< *iter;
		iter++;
	}
     cout<<"\n\n" ;
     getchar(); getchar();
     return 0;
}
```

Šios programos vykdymo rezultatų pavyzdys (pradiniai sąrašo duomenys – atsitiktinai sugeneruotos didžiosios raidės nuo 'A' iki 'Z'):

Pradinis sąrašas: PHQGHUMEAY

Surūšiuotas sąrašas: AEGHHMPQUY

Dėkai (dėklai).

STL konteinerinė klasė *deque* apibrėžta vardų srities *std* antraštiniame faile <*deque*>. Ši, Donaldo Knuth‘o pasiūlytą, struktūra įgyvendina dvipusę eilę (jos pavadinimas tai ir reiškia – **d**uoble-**e**nded **que**ue). STL dėklo pagrindinė paskirtis – suteikti programuotojui tokį nuoseklų konteinerį, kuriame būtų efektyviai apjungtos vektorių ir sąrašų suteikiamos galimybes. Tad, dėkle realizuotas ir elementų indeksavimo

metodas, ir vykdomos elementų įterpimo ir šalinimo operacijos (į bet kurią poziciją / iš bet kurios pozicijos). Įgyvendinti tai padeda atsitiktinis iteratorius, kuris leidžia judėti laisvai, t.y. nurodyti bet kurį dėklo elementą ir judėti šiame konteineryje abejomis kryptimis. Reikėtų pabrėžti, kad elementų įterpimo ir šalinimo į dėklo pradžią ir pabaigą operacijos, kurias įgyvendino funkcijos

void push_front(const T& x); void pop_front();

ir

void push_back(const T& x); void pop_back();

vykdomos efektyviai, tuo tarpu įterpimo ir šalinimo į bet kurią kitą poziciją operacijos *insert()* vykdomos žymiai lėčiau.

Skirtingai nuo vektoriaus, kurio elementai užima nepertraukiamą (ir potencialiai didelį) atminties bloką, dėkas naudoja tam tikrą atminties blokų skaičių. Į tai reikėtų atsižvelgti pasirenkant konteinerio tipą konkrečiai programai, pvz., dėklas būtų geresnis pasirinkimas programuojant ribotų išteklių platformai (įterptinės, mobilios ir kt. sistemos).

Dėklo skelbimas analogiškas sąrašo skelbimui, jo sukūrimui taip pat galima naudoti kelių pavidalų konstruktorius. Pvz., konstruktorius

```
template<class InputIterator>
deque(InputIterator first, InputIterator last,
      const Allocator& alloc = Allocator());
```

sukurs dėklą iš *(last - first)* skaičiaus elementų, kuris bus užpildytas reikšmėmis į kurias rodo iteratoriai iš intervalo nuo *first* iki *last*. Atminties skirstymui šis konstruktorius naudoja objektą *alloc*.

Dėkle perkrauti tam tikri operatoriai, tarp kurių svarbų vaidmenį atlieka priskyrimo operatorius

```
deque<T, Allocator>&
operator=(const deque<T, Allocator>& x);
```

Šis operatorius, visų pirma, šalina visus elementus iš dėklo, o po to kopijuoja į jį kiekvieno dėklo *x* elemento reikšmę. Nuorodos operatoriai

```
reference operator[ ](size_type n);
const_reference operator[ ](size_type n) const;
```

grąžina nuorodą į elementą su indeksu *n*.

Pavyzdys 3.31. Dėko (dėklo) taikymo pavyzdys.

```
#include <iostream>
#include <deque>
#include <cstdlib>

using namespace std;

int main()
{
        deque<int> deklas;
        int sk;

        cout<<"Įveskite natūrinius skaičius, pabaigoje 0: \n";
        while (cin>>sk, sk!=0)
        {
                if (sk % 2 !=0)  deklas.push_front(sk);
                        else  deklas.push_back(sk);
        }
        deque<int>::iterator i;
        cout<<"Dėklo turinys:  \n";
        for (i=deklas.begin();i!=deklas.end(); i++)
                cout<< *i <<' ';
        cout<<"\n\n";
      getchar(); getchar();
      return 0;
}
```

Čia vartotojo įvedami natūriniai skaičiai talpinami į dėklą, nelyginiai į pradžią, o lyginiai į pabaigą. Šios programos vykdymo rezultatų pavyzdys:

Dėklo turinys: 11 9 7 5 3 1 2 4 6 8 10 12

KONTEINERIŲ ADAPTERIAI

Stekai ir eilės.

Stekai ir eilės nėra STL konteinerinės klasės, tai vadinamieji konteinerių adapteriai. Kitaip kalbant šios standartinės klasės tik leidžia kitam konteineriui elgtis taip kaip elgiasi steko (LIFO) arba eilės (FIFO) tipo duomenų struktūros. Todėl apibrėžiant steką arba eilę reikia nurodyti ne tik elementų tipą, bet ir tipą konteinerio, kuriame bus saugomi šie elementai ir kuriam bus adaptuojama tokia duomenų struktūra.

Steko standartinė klasė *stack* apibrėžta vardų srities *std* antraštiniame faile <*stack*>. Pagal nutylėjimą stekas naudoja dėklo konteinerį, taip pat gali būti naudojami sąrašo arba vektoriaus konteineriai. Dėklo konteineris dirba greičiau, tačiau vektorius užima mažiau atminties. Steko elementams turi būti apibrėžtos santykio operacijos mažiau < ir lygu ==.

Steko skelbimų pavyzdžiai:

stack<int> stekas;

stack<double, list<double> > stekas_sarasas;

Pastaba. Kai kuriuose kompiliatoriuose būtina palikti tarpą tarp skliaustų > > , kad ši konstrukcija nebūtų tapatinama su postūmio operacija >>.

Klasė stekas realizuoja tokias standartines funkcijas: empty(); size(); top(); pop(); push(new_elem);

Eilės standartinė klasė *queue* apibrėžta vardų srities *std* antraštiniame faile <*queue*>. Pagal nutylėjimą eilė naudoja dėklo konteinerį, taip pat gali adaptuoti bet kurį konteinerį, kuris realizuoja operacijas front(), back(), push_back() ir pop_front(), pvz., dvikryptį sąrašą. Kaip ir steko atveju eilės elementams turi būti apibrėžtos santykio operacijos mažiau < ir lygu ==.

Pavyzdys 3.32. Konteinerių adapterio (eilės) taikymo pavyzdžiai.

```
#include <iostream>
#include <string>
#include <queue>
#include <list>

using namespace std;

int main()
{
	queue<int> eile_1;
	eile_1.push(10);
	eile_1.push(20);
	cout<< eile_1.front() <<endl;
	eile_1.pop();
	cout<< eile_1.front() <<endl;
	eile_1.pop();

	queue<string, list<string> > eile_2;
	int i;
```

```
        for (i=0; i<10; i++) {
                eile_2.push(string(i+1,'c'));
        }
        for (i=0; i<10; i++) {
                cout<< eile_2.front() << endl;
                eile_2.pop();
        }
    getchar(); getchar();
    return 0;
}
```

Šios programos vykdymo rezultatai:

```
10
20
c
cc
ccc
cccc
ccccc
cccccc
ccccccc
cccccccc
ccccccccc
cccccccccc
```

ASOCIATYVŪS KONTEINERIAI.

Kaip jau minėta, visi nuoseklūs konteineriai įgyvendina prieiga prie elementų arba indeksuojant juos, arba naudojant iteratorius nuosekliam (tiesinis ir dvikryptis iteratoriai) ir atsitiktiniam elementų apdorojimui. Skirtingai nuo nuosekliųjų, asociatyvūs konteineriai skirti greitai prieigai prie konteinerio elementų naudojant raktą, t.y. įgyvendina vadinamąjį tiesioginį adresavimą (angl. *direct addressing, open addressing).* STL bibliotekoje yra 4 rūšių asociatyvūs konteineriai: atvaizdas (*map*), multiatvaizdas (*multimap*), aibė (*set*) ir multiaibė (*multiset*).

Atvaizdo (*map*) šabloninė klasė suteikia prieigą prie kiekvieno konteinerio elemento pagal unikalų to elemento raktą, t.y. atvaizduoja visą konteinerį kaip porų „raktas-reikšmė“ seką. Dėl to atvaizdai dar vadinami „žodynais“ arba „asociatyviais sąrašais“.

Šabloninė klasė *map* turi keturis parametrus: būtini parametrai − tai rakto tipas ir reikšmės tipas, neprivalomi parametrai − tai klasės, apibrėžiančios raktų lyginimo būdą (santykio operaciją) ir atminties skirstymo būdą.

```
template<class K, class T, class Cmp =less<K>, class A=allocator<T> >
class std::map
```

Svarbu pabrėžti, kad šio konteinerio elementai – tai ne viena reikšmė, bet reikšmių pora „raktas-reikšmė“, kuriai sukurti naudojama šabloninė klasė *pair*

```
typedef pair<const K,T> value_type;
```

čia pseudonimas *value_type* apibrėžia šią reikšmių porą, todėl visi atvaizdo iteratoriai ir standartiniai metodai dirba būtent su reikšmių poromis. Įterpiant arba šalinant atvaizdo elementus taip pat dirbama su poromis „raktas-reikšmė“, tipinė konstrukcija gali atrodyti taip:

```
map_Object[key_value] = object_value;
```

Į pirmą poros reikšmę galima kreiptis vardu *first*, į antrą – *second*. Pavyzdžiui, konteinerio elementų reikšmių sumą galima būtų apskaičiuoti taip:

```
map<string, int > m;
. . .
m<string, int>::iterator it=m.begin();
int sum=0; sk=0;
while (it=!m.end()) {
        sum+=it ->second; sk++;
}; vid= sum/sk
```

Trečiam neprivalomam atvaizdo parametrui STL bibliotekoje apibrėžta šabloninė klasė *less*, kuri raktų lyginimui pagal nutylėjimą taiko operaciją < „mažiau“. Papildomi atvaizdo metodai leidžia indeksuoti elementus pagal raktą ir atlikti elemento paiešką pagal raktą. Pavyzdžiui, taikant atvaizdą galima sukurti žodyną, kuriame kiekvienam mėnesiui būtų nurodytas atitinkamas dienų skaičius:

```
map<string, int> MenDienos ;
MenDienos[“sausis“]=31;
. . .
MenDienos[“balandis“]=30;
MenDienos[“gegužė“]=31;
. . .
MenDienos[“gruodis“]=31;

for (map<string, int>::iterator p= MenDienos.begin();
                        p!= MenDienos.end(); ++p)
```

```
{
cout<< “Mėnuo“ << p->first << “turi“<< p->second<<“dienų“<< endl;
}
```

Pateiksime užbaigtą programą, realizuojančią asociatyvų konteinerį − atvaizdą, kurioje poroms „raktas-reikšmė“ sudaryti naudojama šabloninė klasė *pair*. Čia raktai – tai didžiosios raidės, o reikšmės sveikieji skaičiai pradedant 1.

Pavyzdys 3.33. STL asociatyvaus sąrašo (atvaizdo) taikymas.

```
#include <iostream>
#include <map>

using namespace std;

int main()
{
        char c;
        map<char, int> atv;
        //Atvaizdo reikšmių porų inicializacija
        for (int i=0; i<10; i++) {
                atv.insert(pair<char, int>(‘A‘+ i, i+1));
        }
        //Atvaizdo reikšmės paieška pagal raktą
        cout << “Įveskite raktą“ <<endl;
        cin >> c;
        map<char, int>::iterator it;
        it=atv.find(c);
        if (it !=atv.end())
                cout<< it->second << endl;
                else cout << “Tokio elemento nėra“ <<endl;;
        }
    getchar(); getchar();
    return 0;
}
```

ALGORITMAI.

STL bibliotekoje yra apie 60 universalių algoritmų, skirtų darbui su konteineriais. Daugiausiai algoritmų apibrėžta vardų srities *std* antraštiniame faile <*algorithm*>. Algoritmai *accumulate(), inner_product(), partial_sum(), adjacent_difference()* yra apibrėžti antraštiniame faile <*numeric*>.

Su universaliais algoritmais naudojami vadinamieji **objektai-funkcijos**. Tai klasės, kurioje perkrautas funkcijos iškvietimo operatorius, t.y. *operator(),* objektai. Toks objektas iškviečiamas kaip funkcija. STL bibliotekoje apibrėžta daug standartinių objektų-funkcijų, kurios gali naudoti programuotojas apibrėžiant savo objektus-funkcijas. Bibliotekos šabloninių algoritmų parametrais gali būti tiek rodyklės į funkcijas, tiek objektai-funkcijos. Objektai-funkcijos, kurie apibrėžti vardų srities *std* antraštiniame faile <*functional*>, gali būti unariniai (klasė *unary_function*) ir binariniai (klasė *binary_function*). Pirmuosiuose reikia aprašyti argumento (*argument_type*) ir rezultato (*result_type*) tipus, antruose – dar ir antro argumento tipą (*first_argument_type*), (*second_argument_type*).

Universalūs STL algoritmai gali būti taikomi darbui ne tik su standartiniais, bet ir su programuotojo apibrėžtais konteineriais. Visi algoritmai skirstomi į dvi bendras grupes: algoritmai nekeičiantys konteinerio elementų reikšmių (pvz., *for_each(), count(), find_if(), max_element(), search()*) ir algoritmai, keičiantys konteinerio turinį (pvz., *fill(), generate(), copy(), sort(), replace(), transform(), remove()*). Pateiksime kiekvienos grupės algoritmų panaudojimo programų pavyzdžius.

Algoritmas *for_each()* turi tokią iškvietimo forma:

```
template <class InputIterator, class Function>
void  for_each(InputIterator first, InputIterator last, Function f);
```

Šis algoritmas taiko funkcija *f* visiems sekos iš intervalo nuo *first* iki *last* nariams.

Pavyzdys 3.34. Universalaus algoritmo ir objekto-funkcijos taikymas.

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
```

```
using namespace std;

//Objektų-funkcijų klasė
template <class Arg>
class out_times_x : private unary_function<Arg,void>
{
private:
        Arg daugiklis;
public:
        out_times_x(const Arg& x) : daugiklis(x) {}
        void operator()(const Arg& x)
                {cout << x*daugiklis << "  " <<endl;}
};

int main()
{
int seka[5]={1,2,3,4,5};
vector<int> v(seka,seka+5);

// Objekto-funkcijos kūrimas
out_times_x<int> fdaug(3);

// Algoritmo taikymas objektui-funkcijai
for_each (v.begin(), v.end(), fdaug);

getchar(); getchar();
return 0;
}
```

Šios programos vykdymo rezultatai:

3
6
9
12
15

Pailiustruokime algoritmų, kurie keičia konteinerio turinį, taikymą. Algoritmas *fill()* turi tokią iškvietimo forma:

```
template <class ForwardIterator, class T>
void  fill(ForwardIterator first, ForwardIterator last, const T& value);
```

Šis algoritmas užpildo konteinerio elementus iš intervalo nuo *first* iki *last* reikšmėmis *value*.

Pavyzdys 3.35. Algoritmų, keičiančių konteinerio turinį taikymas.

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
int duom[5]={1,2,3,4,5};

// Dviejų vektorių kūrimas
vector<int> v1(duom,duom+5), v2(duom,duom+5);

// Tuščio vektoriaus kūrimas
vector<int> v3,

// Dviejų vektorių užpildymas
fill(v1.begin(), v1.end(), 11);
fill_n(v2.begin(), 4, 22);

//Iteratoriaus taikymas trečio vektoriaus užpildymui
fill_n(back_inserter(v3), 3, 33);

ostream_iterator<int,char> out(cout,"  ");
copy(v1.begin(), v1.end(), out);
cout <<endl;
copy(v2.begin(), v2.end(), out);
cout <<endl;
copy(v3.begin(), v3.end(), out);
cout <<endl;
getchar(); getchar();
return 0;
}
```

Šios programos vykdymo rezultatai:

```
11  11  11  11  11
22  22  22  22  5
33 33 33
```

Sėkmės taikant STL biblioteką.