20. STANDARD TEMPLATE LIBRARY – STL

20.1 Structura STL

Structura STL cuprinde trei elemente principale: *containerele, iteratorii şi algoritmii.*

Containerele sunt diferite tipuri de obiecte care conţin alte obiecte, de tipuri predefinite sau definite de programator.

Iteratorii sunt obiecte care se comportă asemănător pointerilor şi care sunt utilizaţi pentru a accesa elementele unui container.

Algoritmii furnizează funcţionalităţi de acces şi prelucrare asupra elementelor containerelor.

În tabelul 20.1 este prezentată structura STL.

Tabelul nr. 20.1 *Structura STL*

Containere	**Secvenţiale**	**vector**	Implementează vectori alocaţi dinamic.	#include <vector>
		list	Lista liniară dublu înlănţuită	#include < list >
		deque	Asemănător containerului vector, operaţiile putându-se realiza la ambele capete.	#include < deque >
	Asociative	**set**	Mulţime sortată de elemente unice.	#include < set >
		multiset	Mulţime sortată de elemente.	#include < set >
		map	Stochează perechi sortate de tip <cheie, valoare> în care o cheie identifică în mod unic o valoare.	#include < map >
		multimap	Stochează perechi sortate de tip <cheie, valoare> în care o cheie identifică una sau mai multe valori.	#include < map >
	Adaptive	**stack**	Structură de tip stivă.	#include < stack >
		queue	Structură de tip coadă	#include < queue >
		priority_queue	Structură de tip coadă în care elementelor le sunt asociate priorităţi.	#include < queue >
Iteratori	**Acces aleatoriu**		Stochează şi regăseşte valori. Elemente pot fi accesate aleatoriu.	
	Bidirecţionali		Stochează şi regăseşte valori. Iteratorul poate înainta şi reveni.	
	Înainte		Stochează şi regăseşte valori. Iteratorul poate doar înainta.	
	De intrare		Regăseşte dar nu stochează valori. Iteratorul poate doar înainta.	
	De ieşire		Stochează dar nu regăseşte valori. Iteratorul poate doar înainta.	
Algoritmi	Funcţii globale care oferă servicii generale cum ar fi sortări, reordonări, modificări, copieri, căutări etc.			#include <algorithm>

În paragrafele următoare sunt exemplificate modalităţi de lucru utilizând diferitele tipuri de containere, iteratori şi algoritmi.

20.2 Containere STL

Containerele sunt încadrate în trei mari categorii: *secvenţiale, asociative* şi *adaptive*.

Containerele secvenţiale sunt colecţii liniare şi ordonate de date în care accesul se face pe baza poziţiei elementului în cadrul containerului. Cele trei clase predefinite de tip container secvenţial sunt: *vector*, *list* şi *deque*. Ordinea în care se găsesc elementele în cadrul acestor containere este ordinea în care ele sunt adăugate. De reţinut că, in cazul şabloanelor *list* şi *deque* elementele pot fi adăugate pe la ambele capete.

Containerele asociative se diferenţiază de celelalte prin faptul că stocarea elementelor se face pe baza unor chei. Accesul în acest caz se poate realiza după cheie, deci în mod direct. Elementele sunt reţinute în ordinea dată de modul de aranjare a cheilor. Din această categorie de containere fac parte *set*, *multiset*, *map* şi *multimap*.

Containerele adaptive adaugă funcţionalităţi containerelor secvenţiale. Acestea nu pot fi parcurse cu ajutorul iteratorilor întrucât nu sunt folosite în mod independent. Pentru a folosi un astfel de container, programatorul trebuie să aleagă mai întâi containerul de bază căruia să îi fie aplicat un container adaptiv. Astfel containerul *stack* poate adapta containerele *vector***,** *list* şi *deque*, containerul *queue* poate fi implementat pentru *list* şi *deque*, iar *priority_queue* poate fi implementat pentru *vector* şi *deque*.

Pentru a exemplifica modul de lucru cu *containerele secvenţiale* atât pe tipuri de date predefinite, cât şi pentru tipuri de date definite programator, este propus următorul exemplu.

```
#include <iostream>
#include <vector>
#include <list>
#include <deque>

using namespace std;

class Student
{
private:
	int nrMat;
	char nume[20];
public:
	Student(int nr = 0, char* n = "Student"):nrMat(nr)
	{
		strcpy(nume, n);
		cout<<"Constructor clasa Student"<<endl;
	}

	int getNrMat()
	{
	return nrMat;
	}

	char* getNume()
	{
		return nume;
```

```
	}

	void setNume(char* n)
	{
		strcpy(this->nume,n);
	}

	Student(const Student& s)
	{
		this->nrMat = s.nrMat;
		strcpy(this->nume, s.nume);
		cout<<"Constructor de copiere clasa Student"<<endl;
	}

	bool operator<(Student& s)
	{
		return (this->nrMat < s.nrMat);
	}

	friend ostream& operator<<(ostream& o, Student s)
	{
		o<<s.getNrMat()<<" "<<s.getNume()<<endl;
		return o;
	}
};

void main()
{
	Student s1(1, "Gigel");
	Student s2(2, "Ionel");
	Student s3(3, "Balanel");

////////////////////////////////////////////////////////////////

	//vector de tip de data predefinit
	vector<int> vectInt;

	vectInt.push_back(10);
	vectInt.push_back(1);
	vectInt.push_back(25);
	vectInt.push_back(12);

	for(int i=0; i<vectInt.size(); i++)
		cout<<vectInt[i]<<" ";
	cout<<endl;

	//-------------------------------------------------

	//vector de tip de data definit de programator
	vector<Student> vectStud;

	vectStud.push_back(s3);
	vectStud.push_back(s2);
	vectStud.push_back(s1);

	for(int i=0; i<vectStud.size(); i++)
		cout<<vectStud[i]<<" ";
	cout<<endl;

////////////////////////////////////////////////////////////////
```

```
	// lista de tip de data predefinit
	list<int> listInt;

	listInt.push_back(10);//inserare la sfarsit cu ajutorul metodei
push_back()
	listInt.push_front(1);//inserare la inceput prin push_front()
	listInt.insert(listInt.end(),25);//inserare prin insert()
	listInt.insert(listInt.begin(),12);//inserare prin insert()


	//declararea unui iterator cu care va fi traversata lista
	//operatorul [] nu este implementat pentru list
	list<int>::iterator itInt;

	for(itInt=listInt.begin(); itInt!=listInt.end(); itInt++)
		cout<<*itInt<<" ";
	cout<<endl;

	listInt.sort();

	cout<<"Lista sortata"<<endl;
	for(itInt=listInt.begin(); itInt!=listInt.end(); itInt++)
		cout<<*itInt<<" ";
	cout<<endl;

	//-------------------------------------------------------------

	// lista de tip de data definit de programator
	list<Student> listStud;

	//inserare la sfarsit cu ajutorul metodei push_back()
	listStud.push_back(s3);
	listStud.push_front(s1);//inserare la inceput prin push_front()
	listStud.insert(listStud.end(),s2);//inserare prin insert()


	//declararea unui iterator cu care va fi traversata lista
	//operatorul [] nu este implementat pentru list
	list<Student>::iterator itStud;

	for(itStud=listStud.begin(); itStud!=listStud.end(); itStud++)
		cout<<*itStud<<" ";
	cout<<endl;

	listStud.sort();

	cout<<"Lista de studenti sortata"<<endl;
	for(itStud=listStud.begin(); itStud!=listStud.end(); itStud++)
		cout<<*itStud<<" ";
	cout<<endl;


////////////////////////////////////////////////////////////////

	//deque pentru tipuri de date predefinite
	deque<int> deqInt;

	deqInt.push_front(10);
	deqInt.push_back(1);
	deqInt.push_front(25);
	deqInt.push_back(12);
```

```
	deque<int>::iterator iInt;

	for(iInt = deqInt.begin(); iInt!= deqInt.end(); iInt++)
		cout<<*iInt<<" ";
	cout<<endl;

	//------------------------------------------------------------

	//deque pentru tipuri de date definite de programator
	deque<Student> deqStud;

	deqStud.push_front(s1);
	deqStud.push_back(s3);
	deqStud.push_front(s2);

	deque<Student>::iterator iStud;

	for(iStud = deqStud.begin(); iStud!= deqStud.end(); iStud++)
		cout<<*iStud<<" ";
	cout<<endl;
}
```

Pentru tipul de date *int* şi tipul definit *Student* sunt construite câte un container de tip *vector*, unul de tip *list* şi unul de tip *deque*. In fiecare dintre acestea sunt adăugate elemente de tip *int* sau *Student*, după caz, utilizând metode specifice acestor şabloane.

Clasa *vector* permite adăugarea elementelor pe la un singur capăt, pe când clasele *list* şi *deque* permit operaţii pe la ambele capete. Metoda *push_back()* pentru toate cele trei clase şi metoda *push_front()* pentru *list* şi *deque* copiază obiectul într-unul dintre elementele clasei. Dacă pentru tipurile predefinite de date copierea valorii obiectului nu pune nici o problemă, copierea unui tip definit de programator necesită supraîncărcarea constructorului de copiere, în special în cazul în care obiectul conţine atribute cu extensie în memoria dinamică. Clasa Student supraîncarcă constructorul de copiere şi operatorul >>, pentru tipărirea pe ecran a unui obiect de tip Student. Vectorul de elemente int şi cel de elemente Student pot fi parcurşi cu ajutorul unui contor ce ia valori de la 0 la dimensiunea masivului, dimensiune dată de metoda *size()* a vectorului. Această parcurgere se datorează faptului că vectorul ocupă un spaţiu contiguu de memorie dinamică, ceea ce permite calculul deplasării unui element faţă de începutul vectorului. Operatorul [] nu este definit şi pentru clasele *list* şi *deque*, de aceea este necesară definirea unui iterator pentru a realiza accesul la date. Acest iterator poate parcurge masivul de la început – dat de metoda *begin()* până la sfârşit, dat de metoda *end().*

Clasa *list* permite sortarea elementelor containerului (sortare crescătoare implementată de metoda *sort()* sau sortare descrescătoare, implementată de metoda *reverse()*), motiv pentru care, în moment ce containerul este utilizat pentru stocarea unor obiecte de tip definit de programator, trebuie supraîncărcat operatorul<. Pe baza acestuia este stabilită relaţia de ordine în clasa respectivă.

Şablonul *deque* combină facilităţile oferite de *vector* cu cele oferite de *list*. Acesta este proiectat ca un vector ce poate creşte în ambele direcţii, deci metodele *push_back()/pop_back()* şi *push_front()/pop_front()* sunt valide. Adăugarea de elemente în interiorul unui *deque* este mai lentă decât

pe o poziţie interioară într-o listă întrucât în cazul *deque* toate elementele trebuie mutate.

Pentru exemplificarea modului de lucru cu *containerele asociative* se consideră clasa *Student* şi prelucrările corespunzătoare din exemplul următor.

```
#include <iostream>
#include <set>

using namespace std;

class Student
{
private:
	int nrMat;
	int varsta;
	char nume[20];
public:
	Student(int nr = 0, int v=0, char* n = "Student"):nrMat(nr),
			varsta(v)
	{
		strcpy(nume, n);
		cout<<"Constructor clasa Student"<<endl;
	}

	int getNrMat(){return nrMat;}
	char* getNume(){return nume;}
	int getVarsta(){return varsta;}
	void setNume(char* n){strcpy(this->nume,n);}

	Student(const Student& s){
		this->nrMat = s.nrMat;
		this->varsta = s.varsta;
		strcpy(this->nume, s.nume);
		cout<<"Constructor de copiere clasa Student"<<endl;
	}

	bool operator<(const Student& s)const{
		return (this->nrMat < s.nrMat);
	}

	friend ostream& operator<<(ostream& o, Student s){
		o<<s.getNrMat()<<" "<<s.getNume()<<endl;
		return o;
	}
};
void main()
{
	Student s1(1, 19, "Gigel");
	Student s2(2, 20, "Ionel");
	Student s3(3, 19, "Balanel");
	Student s4(4, 19, "Ilie");

//////////////////////
//sablon set //
//////////////////////
set<Student> setStud;

setStud.insert(s4);
setStud.insert(s2);
```

```
setStud.insert(s3);
setStud.insert(s1);

cout<<"Comparatie dupa cheie:";
cout<<s4.getNume()<<" este"<<(setStud.key_comp()(s4,s2)?" inaintea ":"
dupa ")<<s2.getNume()<<endl;
cout<<"Comparatie dupa valoarea elementului:";
cout<<s4.getNume()<<"  este"<<(setStud.value_comp()(s4,s2)?"  inaintea
":" dupa ")<<s2.getNume()<<endl;

cout<<"Comparatie dupa cheie:";
cout<<s3.getNume()<<" este"<<(setStud.key_comp()(s3,s1)?" inaintea ":"
dupa ")<<s1.getNume()<<endl;
cout<<"Comparatie dupa valoarea elementului:";
cout<<s3.getNume()<<"  este"<<(setStud.value_comp()(s3,s1)?"  inaintea
":" dupa ")<<s1.getNume()<<endl;

cout<<"Comparatie dupa cheie:";
cout<<s1.getNume()<<" este"<<(setStud.key_comp()(s1,s4)?" inaintea ":"
dupa ")<<s4.getNume()<<endl;
cout<<"Comparatie dupa valoarea elementului:";
cout<<s1.getNume()<<"  este"<<(setStud.value_comp()(s1,s4)?"  inaintea
":" dupa ")<<s4.getNume()<<endl;

set<Student>::iterator itStud;

itStud = setStud.find(2);
while(itStud!=setStud.end())
{
      cout<<itStud->getNume()<<" ";
      itStud++;
}
}
```

Întrucât toate containerele asociative reţin elementele sortate pe baza cheilor, clasa Student supraîncarcă operatorul< pentru a defini relaţia de ordine. Clasa Student utilizată în lucrul cu şablonul *set* defineşte relaţia de ordine pe baza numărului matricol al studentului, număr ce identifică în mod unic un student. Metodele *key_comp()* şi *value_comp()* au acelaşi efect în lucrul cu şabloane set întrucât fiecare valoare a unui element corespunde unei singure chei. Aşadar, valoarea şi cheia sunt identice. Rezultatele rulării programului confirmă ordinea impusă prin supraîncărcarea operatorului<. Containerele asociative dispun şi de metode de regăsire directă a elementelor, adică după valoare cheii. Această metodă se numeşte *find()* şi primeşte ca parametru valoarea cheii, întorcând un pointer către elementul respectiv. In exemplul prezentat, este definit un iterator care adresează într-o primă fază elementul ce are valoarea cheii egală cu 2. Acest element este obţinut cu ajutorul metodei *find().* De la respectiva poziţie şi până la sfârşitul mulţimii aceasta este parcursă, iar toate numele ce aparţin acestui interval sunt tipărite pe ecran.

Pentru a exemplifica lucrul cu *multiset* va fi stabilită o nouă relaţie de ordine a clasei Student, pe baza vârstei. Astfel, o cheie nu va identifica în mod unic neapărat doar un element.

```
class Student
{
...
//supraincarcare operator< a clasei Student modificata
bool operator<(const Student& s)const{
            return (this->varsta < s.varsta);
}
...
}; //sfarsit clasa Student
//////////////

void main()
{
      Student s1(1, 19, "Gigel");
      Student s2(2, 20, "Ionel");
      Student s3(3, 19, "Balanel");
      Student s4(4, 19, "Ilie");
      Student s5(5, 20, "Maricica");
      Student s6(6, 20, "Lenuta");

      Student vectStud[] = {s6, s1, s5, s4, s2, s3};
////////////////////////////
//lucrul cu sablon multiset //
////////////////////////////
multiset<Student> mSetStud;

for(int i=0; i<6; i++)
      mSetStud.insert(vectStud[i]);

multiset<Student>::iterator itStud;
std::pair<multiset<Student>::iterator,multiset<Student>::iterator>
itVarsta;

cout<<"Persoane de aceeasi varsta cu "<<s1.getNume()<<": ";
itVarsta = mSetStud.equal_range(s1);
for(itStud=itVarsta.first; itStud!= itVarsta.second; itStud++)
      cout<<itStud->getNume()<<" ";
cout<<endl;

cout<<"Persoane de aceeasi varsta cu "<<s2.getNume()<<": ";
itVarsta = mSetStud.equal_range(s2);
for(itStud=itVarsta.first; itStud!= itVarsta.second; itStud++)
      cout<<itStud->getNume()<<" ";
cout<<endl;
}
```

Pentru a putea identifica în cadrul mulţimii de studenţi pe ci cu aceeaşi vârstă este definită o pereche de iteratori ai multisetului. Această pereche este încărcată cu valori prin apelul metodei *equal_range(obiect),* primul dintre ei adresând primul obiect ce satisface condiţia, iar al doilea adresând primul obiect ce numai satisface condiţia. Cu ajutorul unui nou iterator, *itStud*, sunt parcurse toate adresele ce se încadrează în intervalul precizat, iar numele studenţilor sunt tipărite pe ecran.

Containerele *set* şi *multiset* sunt implementate în C++ cu ajutorul arborilor de căutare, astfel regăsirea fiind foarte rapidă.

Utilizarea containerul asociativ *map* este prezentat într-un exemplu de agendă telefonică. Se consideră o adresă telefonică, în care vor fi inserate perechi de şiruri de caracter sub forma *<nume, număr de telefon>*. Pentru a putea insera într-un container *map* valori, acestea trebuie să fie

furnizate mai întâi ca parametru constructorului clasei *pair<string, string>*. Numele persoanei este cheie unică, iar fiecărei persoane îi este asociat un număr de telefon. Agenda permite căutarea unui număr de telefon după numele persoanei. Pentru căutarea în agendă este folosit un iterator.

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

using namespace std;

int main()
{
    map<string, string> agenda;
      typedef pair<string, string> String_pair;

      agenda.insert(String_pair("Gigel", "021111111"));
      agenda.insert(String_pair("Georgiana", "023666777"));
      agenda.insert(String_pair("Ionel", "021333456"));
      agenda.insert(String_pair("Balanel", "021999888"));
      agenda.insert(String_pair("Maricica", "021333444"));
      agenda.insert(String_pair("Lenuta", "023444555"));

      map<string, string>::iterator itAgenda;

      cout<<"Numele valabile: ";
      for(itAgenda = agenda.begin(); itAgenda != agenda.end();
itAgenda++)
            cout<<(*itAgenda).first<<"  "; /* scrie prima valoare
stocata, adica numele */
      cout<<endl;
      cout<<"-----------------------------------------------"<<endl;

      cout<<"Introduceti numele: ";
      string nume;
      cin>>nume;
      while(nume != "IESIRE")
      {
            cout<<endl;
            map<string, string>::iterator itNume = agenda.find(nume);
            if(itNume != agenda.end()){
                  cout<<(*itNume).first<<"          "<<(*itNume).second
<<endl<<endl;
            }
            else{
                  cout<<"Acest contact nu exista in agenda"
<<endl<<endl;
            }
            cout<<"Introduceti numele: ";
            cin>>nume;
      }
}
```

Numele persoanei, care este cheie, este prima valoare stocată din cadrul perechii şi poate fi referit ca *(*numeIterator).first*. Numărul de telefon este a doua valoare memorată şi poate fi referit cu ajutorul iteratorului astfel: *(*numeIterator).second*. Căutarea unui număr de telefon după numele persoanei este realizată cu ajutorul metodei *find(valoare_cheie)* a şablonului **map**. Această funcţie returnează un

iterator, a cărui valoare indică fie elementul pentru care valoarea cheii este egală cu valoarea căutată, fie o adresă egală cu cea furnizată de metoda *end()* a clasei *map*, în cazul în care nu există nici un element în container pentru care valoarea cheii să fie egală cu valoarea dorită.

Containerul asociativ *multimap* permite identificarea unei sau mai multor valori pe baza unei valori a cheii. Mai exact, cheia nu este unică, pot exista oricâte perechi pentru care cheia să fie egală.

Modul de lucru cu *containere adaptive* este pus în evidenţă de exemplul următor.

```
#include <iostream>
#include <stack>
#include <queue>
#include <vector>
#include <deque>
#include <list>

using namespace std;

template <class T> popStiva(T& st)
{
	while(!st.empty()){
		cout<<st.top()<<" ";
		st.pop();
	}
}

template <class T> popCoada(T& cd)
{
	while(!cd.empty()){
		cout<<cd.front()<<" ";
		cd.pop();
	}
}

void main()
{

	//stiva adaptata pentru container deque -> implicit
	stack<char, deque<char> > stivaDeq; /* echivalent cu stack<int>
stivaDeq; */
	//stiva adaptata pentru container vector
	stack<char, vector<char> > stivaVect;
	//stiva adaptate pentru container list
	stack<char, list<char> > stivaList;

	//coada adaptata pentru container deque
	queue<char, deque<char> > coadaDeq;
	//coada adaptata pentru conatainer list
	queue<char, list<char> >coadaList;

	for(char i=57; i>47; i--)
	{
		stivaDeq.push(i);
		coadaDeq.push(i);
		stivaVect.push(i+17);
		stivaList.push(i+49);
		coadaList.push(i+49);
	}
```

```
	cout<<"Stiva deque: ";
	popStiva(stivaDeq);
	cout<<endl;

	cout<<"Coada deque: ";
	popCoada(coadaDeq);
	cout<<endl;

	cout<<"Stiva vector: ";
	popStiva(stivaVect);
	cout<<endl;

	cout<<"Stiva list: ";
	popStiva(stivaList);
	cout<<endl;

	cout<<"Coada list: ";
	popCoada(coadaList);
	cout<<endl;
}
```

In exemplul de mai sus este creat un container adaptiv de tip stivă pentru fiecare container secvenţial. Aşadar, *stivaDeq* corespunde implementării unei stive pentru un container de tip *deque*, *stivaVect* este o adaptare a clasei vector la o structură de stivă iar *stivaList* este implementarea stivei pentru o listă. Totodată au fost create şi două containere de tip coadă pentru cele două containere secvenţiale ce permit această adaptare: *coadaDeq* – adaptarea unui *deque* pentru a implementa o structură de tip coadă şi *coadaList* – adaptarea unei liste. In stiva şi coada corespunzătoare şablonului *deque* sunt introduse cifrele de la 0 la 9. Afişarea elementelor din cadrul stivei şi cozii se face cu ajutorul metodelor *top(),* respectiv *front().* Elementele din cadrul cozii vor fi afişate în ordine inversă faţă de cele din cadrul stivei. In stiva corespunzătoare şablonului *vector* sunt inserate caracterele de la J la A. Afişarea se face în ordine inversă, deci aceste vor apărea pe ecran de la A la J. Similar, în coada şi stiva corespunzătoare şablonului *list* sunt inserate caracterele de la j la a. Afişarea pe baza cozii se face în ordinea în care au fost introduse, deci de la j la a, iar afişarea pe baza stivei se face în ordine inversă, deci de la a la j.

Următorul program exemplifică utilizarea containerului asociativ *multimap*.

```
#include "stdafx.h"
#include <map>
#include <string>
#include <iostream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{

	multimap<string, string> agenda;

	agenda.insert(pair<string,string>("Alina","0711111111"));
	agenda.insert(pair<string,string>("Alina","0211111111"));
```

```
      agenda.insert(pair<string,string>("Alina","0311111111"));
      agenda.insert(pair<string,string>("Dragos","0711111112"));
      agenda.insert(pair<string,string>("Dragos","0311111112"));
      agenda.insert(pair<string,string>("Vlad","0711111112"));
      agenda.insert(pair<string,string>("Ionut","0711111114"));
      agenda.insert(pair<string,string>("Marian","0711111115"));

      multimap<string, string>::iterator it;

      //Parcurgere intreaga agenda
      for (it = agenda.begin(); it != agenda.end(); it++)
            cout << "Persoana de contact " << (*it).first << ", are
numarul: " << (*it).second<< endl;


      cout<<"Dragos are " <<agenda.count("Dragos")<<" intrari in
agenda"<<endl;


      cout<<"Numerele de telefon ale lui Dragos sint:" << endl;
    for (it = agenda.lower_bound("Dragos");it !=
agenda.upper_bound("Dragos"); it++)
            cout << "\t" << it->second << endl;

      agenda.erase("Dragos");
      cout<<"Dragos are " <<agenda.count("Dragos")<<" intrari in
agenda"<<endl;

      cin.get();
      return 0;
}
```

În programul de mai sus s-a utilizat clasa *multimap* pentru a crea o agendă de contacte. Cheia de căutare este numele persoanei, o persoană putând avea asociate mai multe numere de telefon. Sunt exemplificate pentru clasa *multimap* operaţiile de inserare, numărare a elementelor, ştergere şi parcurgere.

Un adaptor *priority-queue* furnizează un subset restrâns de funcţionalităţi containerului secvenţial corespunzător. Aceste funcţionalităţi se referă la inserţia de noi elemente, consultarea şi extragerea/ştergerea de elemente. O coadă cu priorităţi nu poate fi parcursă prin intermediul unui iterator. Containerul asociat în mod implicit este *vector*. Elementul din vârful unei cozi cu priorităţi este elementul cu cea mai mare prioritate.

```
#include <iostream>
#include <queue>

using namespace std;

void main()
{
  priority_queue<char> Q;

  Q.push('A');
  Q.push('C');
  Q.push('B');
  Q.push('E');
  Q.push('D');
  Q.push('F');
```

```
  while(!Q.empty())
 {
        cout<<Q.top()<<" ";
        Q.pop();
  }

  //iesire: F E D C B A
}
```

Metoda *top()* returnează o referinţă către elementul cu valoarea cea mai mare din coadă. Acest lucru este realizat astfel datorită definirii implicite a unui *Comparator*. În exemplu, deşi literele nu au fost introduse în coadă astfel încât să fie tipărite de la mare la mic – de la F la A, stocarea lor coada cu priorităţi a rezultat în afişarea lor de la F la A.

20.3 Iteratori STL

Iteratorii se aseamănă cu pointerii, dar sunt de fapt obiecte ce adresează alte obiecte. Cu ajutorul lor pot fi adresate elemente ale containerelor care aparţin anumitor intervale. Iteratorii reprezintă interfaţa de comunicaţie între algoritmi şi containere, fiind preluaţi ca parametrii de către algoritmi. Containerele le furnizează algoritmilor o cale de acces către elementele lor prin intermediul iteratorilor.

In exemplele prezentate anterior au fost utilizaţi pentru parcurgerea containerelor iteratori definiţi de programator după modelul *container<tip_data>::iterator numeIterator;*. STL furnizează şi iteratori predefiniţi, a căror utilizare presupune includerea fişierului *iterator*.

Un *iterator de tip istream*, notat *istream_iterator<T,distanta>* formatează intrarea provenită de la un obiect de tip T dintr-un flux de intrare. In momentul în care fluxul ajunge la sfârşit iteratorul ia o valoare de sfârşit specială pentru a specifica sfârşitul datelor. Acest iterator permite modificarea elementului pe care îl referă.

Un *iterator de tip ostream*, notat *ostream_iterator<T>* este un iterator de ieşire ce realizează legătura unui iterator cu un flux de ieşire. De menţionat că elementul referit de acest tip de iterator nu poate fi modificat, ci doar consultat.

```
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

void main()
{
      vector<char> vect;
      istream_iterator<char>inIterator(cin);
      ostream_iterator<char>outIterator(cout, "\n");

      //varianta a - stream intrare
      copy(istream_iterator<char>(cin),
            istream_iterator<char>(), back_inserter(vect));

      //varianta b - stream intrare
```

```
	/* while(*inIterator)
	{
		vect.push_back(*inIterator);
		inIterator++;
	} */

	//varianta a - stream iesire
	copy(vect.begin(),
		vect.end(), ostream_iterator<char>(cout, "\n"));

	//varianta b - stream iesire
	/* while(!vect.empty())
	{
		*outIterator = vect.back();
		vect.pop_back();
	} */
}
```

În exemplul de mai sus este prezentat modul de lucru cu iteratori de tip *istream_iterator* şi *ostream_iterator*. La declararea unui iterator de intrare, respectiv de ieşire, trebuie specificate tipul dedate al elementelor din colecţie (exemplul utilizează elemente de tip char) şi fluxul de date – *cin*, respectiv *cout*. ”\n” este o constantă care va fi afişată automat după afişarea unui element. Practic elementele vor fi afişate câte unul pe linie. Varianta a din exemplul, atât pentru fluxul de intrare cât şi pentru cel de ieşire este echivalentă cu varianta b. In varianta a însă, nu s-au folosit iteratori declaraţi anterior. Funcţia *copy(InputIterator prim, InputIterator ultim, OutputIterator rezultat)* copiază elementele care aparţin intervalului *[prim, ultim)* în intervalul *[rezultat, rezultat + (ultim - prim))*.

Un *iterator invers*, de tip *reverse_iterator* este folosit pentru parcurgerea în ordine inversă a elementelor unei colecţii.

```
#include <iostream>
#include <vector>
#include <iterator>

using namespace std;

void main()
{
	vector<char> vect;
	vector<char>::reverse_iterator reverseIt;

	for(int i=57; i>47; i--)
		vect.push_back(i);

	for(reverseIt=vect.rbegin(); reverseIt != vect.rend();
reverseIt++)
		cout<<*reverseIt<<" ";
}
```

Metoda *rbegin()* este echivalentă cu metoda *end()* a vectorului, iar metoda *rend()* este echivalentă cu *begin()*. Metodele *rbegin()* şi *redn()* sunt utilizate pentru a lucra cu iteratori inverşi, iar operatorul **++** semnifică de fapt decrementarea acestuia. In vector sunt introduse cifrele de la 9 la 0, iar parcurgerea elementelor cu ajutorul iteratorului invers afişează pe ecran

cifrele de la 0 la 9. Practic, afişarea se face ca şi cum vectorului i-ar fi aplicat un adaptor de tip stivă, aceasta fiind golită prin metoda *pop()*.

Un *iterator de inserare*, *insert_iterator<container<T> >* este un iterator specializat pe inserări în cadrul containerelor. Parametrul primit de către acest tip de iterator este tipul containerului în care se face inserarea, iar tipul containerului va primi ca parametru tipul de dată al elementelor. Există şi două forme specializate ale acestui iterator, *back_insert_iterator* pentru inserări la sfârşitul containerelor şi *front_insert_iterator* pentru inserări la începutul containerelor.

```
#include <iostream>
#include <list>
#include <iterator>

using namespace std;

void main()
{
	list<char> lista;
	lista.push_front('D');

	//definesc un iterator cu care inserez la inceputul containerului
	insert_iterator<list<char>	>	insertItBegin(lista, lista.begin());
	//inserez in lista
	*insertItBegin = 'A'; *insertItBegin = 'B';*insertItBegin = 'C';

	//afisez pe ecran
	copy(lista.begin(),lista.end(),ostream_iterator<char>(cout," "));
}
```

Constructorul iteratorului de inserare primeşte ca parametru containerul în care vor fi inserate elemente şi poziţia din cadrul containerului unde se vor efectua inserările.

20.4 Algoritmi STL

Pe lângă metodele care aparţin de clasele container, STL pune la dispoziţie funcţii globale ce oferă posibilitatea efectuării unor prelucrări asupra mai multor containere prin includerea fişierului *algorithm*. Aceste funcţii primesc ca parametri iteratori ai diferitelor containere, acesta fiind modul de realizare a accesului la elementele containerelor.

Algoritmii STL se împart în patru mari categorii:

A. *Algoritmi care modifică ordinea elementelor în container – modifying sequence operations***.** Dintre cei mai folosiţi algoritmi care fac parte din această categorie sun amintiţi: *copy()*, *replace()*, *transform()* şi *remove()*.

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

using namespace std;

void main()
{
	vector<char> vect;
	list<char> lista(6);

	vect.push_back('A');
	vect.push_back('B');
	vect.push_back('C');
	vect.push_back('D');
	vect.push_back('A');
	vect.push_back('A');

	//functia COPY - pentru copierea elementelor
	//dintr-un container in altul
	copy(vect.begin(), vect.end(), lista.begin());

	//functia COPY pentru tiparirea pe ecran
	//a elementelor copiate
	copy(lista.begin(), lista.end(), ostream_iterator<char>(cout, "
"));
	cout<<endl;

	//functia REPLACE - pentru inlocuirea
	//caracterelor A cu a
	replace(lista.begin(), lista.end(), 'A', 'a');
	cout<<"Lista dupa efectuarea inlocuirilor:"<<endl;
	copy(lista.begin(), lista.end(), ostream_iterator<char>(cout, "
"));
}
```

Funcţiilor le sunt furnizate ca parametrii intervale de acţiune asupra elementelor prin intermediul a doi iteratori: unul care indică de unde începe acţiunea funcţiei, unul unde se termină. Funcţia *copy()* este utilizată în cadrul exemplului pentru a copia elemente dintr-un container în altul şi pentru a afişa pe ecran elementele containerului în care s-au efectuat modificări. Pe lângă cei doi iteratori ce definesc intervalul căruia îi aparţin elementele, funcţia mai primeşte ca parametru şi un iterator ce defineşte destinaţia copierilor. Pentru copierea elementelor dintr-un container în altul, al treilea parametru este poziţia din containerul destinaţie unde se doresc să fie inserate elementele, adică *lista.begin()* (la începutul listei). Pentru tipărirea pe ecran a elementelor, al treilea parametru este dat de un iterator de ieşire, care leagă copierea valorilor de fluxul standard de ieşire, ecranul: *ostream_iterator<char>(cout, " ").* Funcţia *replace()* efectuează înlocuirea unei valori cu altă valoare, pentru elementele ce aparţin unui interval dat. Aşadar, în lista din exemplu, valoarea **A** este înlocuită cu **a**, de la începutul listei şi până la sfârşitul ei.

B. *Algoritmi care nu modifică ordinea elementelor în container – non-modifying sequence operations*. Cei mai întâlniţi din această categorie sunt: *for_each()*, *find()*, *count()* şi *equal()*.

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

using namespace std;

void scrie(char c)
{
	cout<<++c<<"/";
}

void main()
{
	list<char> lista;

	lista.push_back('A');
	lista.push_back('B');
	lista.push_back('C');
	lista.push_back('D');
	lista.push_back('A');
	lista.push_back('A');

	for_each(lista.begin(), lista.end(), scrie);

	cout<<"Numar de elemente egale cu A: "<<count(lista.begin(),
lista.end(), 'A')<<endl;
	cout<<"Elemente dupa D:";
	list<char>::iterator it = find(lista.begin(), lista.end(), 'D');
	it++;
	for( ;it != lista.end(); it++)
		cout<<*it<<" ";
	cout<<endl;
}
```

Funcţia *for_each()* iterează de-a lungul intervalului delimitat de cei doi iteratori primiţi ca primi doi parametrii şi prelucrează fiecare element inclus în acest interval. In exemplul de mai sus, funcţia *for_each()* tipăreşte pe ecran caracterele incluse în lista, incrementate. Dacă lista conţine *A*, atunci în locul acestuia va fi tipărit *B*. Funcţia *count()* numără câte elemente dintr-un interval dat al containerului au o valoare egală cu o valoare dată. Funcţia *find()* returnează un iterator către poziţia din cadrul containerului unde există primul element ce are o valoare egală cu valoarea primită ca parametru. Această funcţie este utilizată în exemplu pentru a tipări toate caracterele din listă care îi urmează caracterului *D*.

C. *Algoritmi de sortare şi operaţii similare.* Dintre cei mai cunoscuţi algoritmi incluşi în această categorie, îi reamintim pe următorii: *sort()***,** *equal_range()***,** *merge()* şi *includes()*.

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

using namespace std;

bool less(int n1, int n2){return n1<n2;}
```

```
bool greater(int n1, int n2){return n1>n2;}

void main()
{
	vector<int> vect;

	vect.push_back(10);
	vect.push_back(5);
	vect.push_back(7);
	vect.push_back(1);
	vect.push_back(13);

	sort(vect.begin(), vect.end());
	copy(vect.begin(), vect.end(), ostream_iterator<int>(cout, "
"));
	cout<<endl;
	sort(vect.begin(), vect.end(), less);
	copy(vect.begin(), vect.end(), ostream_iterator<int>(cout, "
"));
	cout<<endl;
	sort(vect.begin(), vect.end(), greater);
	copy(vect.begin(), vect.end(), ostream_iterator<int>(cout, "
"));
	cout<<endl;
}
```

Funcţia *sort()* sortează elementele unui container după o condiţie dată prin intermediul unui pointer la o funcţie. Această funcţie trebuie să primească două variabile de tipul elementelor containerului ca parametrii şi să întoarcă un boolean. Dacă ceea ce întoarce este egal cu *false*, funcţia *sort()* interschimbă elementele. In exemplul de mai sus, sunt definite două funcţii, *less* şi *greater*, cu ajutorul cărora elementele vectorului *vect* sunt sortate crescător, respectiv descrescător. Funcţia *sort()* sortează elementele în mod implicit (dacă nu primeşte şi pointer către o funcţie ca parametru) crescător. Dacă tipul obiectelor stocate în container este unul definit de programator, este necesară supraîncărcarea operatorului < pentru a defini o relaţie de ordine în cadrul containerului. Totodată, dacă se doreşte tipărirea pe ecran a elementelor containerului, este necesară supraîncărcarea operatorului <<.

D. *Algoritmi generali pentru operaţii numerice*, precum *min()* şi *max()*.

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

using namespace std;

void main()
{

	int a = 10;
	int b = 11;
	int c = 12;

	cout<<"Minimul dintre "<<a<<" si "<<b<<" este: "<<
min(a,b)<<endl;
	cout<<"Minimul dintre "<<a<<", "<<b<<" si "<<c<<" este: "<<
min(min(a,b),c)<<endl;
```

```
      cout<<"Maximul dintre "<<a<<" si "<<b<<" este: "<<
max(a,b)<<endl;
      cout<<"Maximul dintre "<<a<<", "<<b<<" si "<<c<<" este: "<<
max(max(a,b),c)<<endl;
}
```

Funcţiile *min()* şi *max()* puse la dispoziţie de STL returnează minimul, respectiv maximul dintre două numere. Ele pot fi utilizate şi în cascadă, pentru a calcula minimul, respectiv maximul dintre mai multe numere.

În practică, biblioteca standard STL este suportată de toate compilatoarele C++ (Intel, Microsoft, Unix) pe toate tipurile de platforme – UNIX, Windows.