Tratarea erorilor în C++. Excepţii.

- Tratarea erorilor în limbajul C se realizează printr-un mecanism ad-hoc ce presupune testarea prin intermediul instrucţiunilor ***if*** a condiţiilor de eroare apărute în diverse funcţii. Acest mecanism este neproductiv deoarece „poluează" codul aferent aplicaţiei în sine şi, în plus, nu permite luarea deciziei privitoare la tratarea erorii într-un cadru mai larg.
- În C++ a fost creat un mecanism specializat de tratare a erorilor, ce foloseşte conceptul de excepţie
- O excepţie poate fi privită şi ca fiind un obiect ce este trimis dinspre zona de cod unde a apărut o eroare către o zonă de cod care trebuie să o trateze.
- Implementarea mecanismului foloseşte trei cuvinte cheie : ***try***, ***throw*** şi ***catch***
- Blocul ***try*** încadrează secvenţa de cod în care este posibil să apară erori.
- În cazul apariţiei unei erori, aceasta este „lansată" folosind clauza ***throw***
- Eroarea este „captată" de un bloc ***catch***
- Observaţie: excepţiile se folosesc pentru tratarea erorilor de tip sincron. În cazul erorilor de tip asincron (evenimente produse de mediul extern programului), se foloseşte mecanismul întreruperilor.
- Execuţia clauzei ***throw*** presupune crearea unui obiect de un tip precizat (se pot arunca orice tipuri de date), obiect ce nu ar fi fost creat în condiţiile execuţiei normale (fără erori) a programului.
- Funcţia în care s-a produs eroarea returnează acest obiect, chiar dacă nu are prevăzut în antet/prototip un tip de date corespunzător tipului obiectului astfel creat.
- Continuarea execuţiei programului după execuţia clauzei ***throw*** este preluată de clauza ***catch*** care are ca parametru acelaşi tip de date cu cel aruncat de excepţie.
- La părăsirea unei funcţii prin aruncarea unei excepţii se generează cod corespunzător pentru distrugerea obiectelor locale create şi eliberarea memoriei alocate în funcţie.

- Clauzele ***catch*** sunt mutual exclusive.

- Sintaxa pentru clauza ***try*** este :

```
try {
   // cod
   throw TipExceptie;

 }
```

- Sintaxa pentru clauza ***throw*** este:

```
throw TipExceptie;
```

- Sintaxa pentru clauza ***catch*** este:

```
catch(TipExceptie [parametru])
{
      // cod tratare exceptie
}
```

- Clauza ***throw*** fără parametri pasează eroarea secţiunii apelante.
- După un bloc ***try***, pot urma unul sau mai multe blocuri ***catch***.
- Daca tipul excepţiei corespunde cu unul din tipurile precizate într-o clauză ***catch*** ce îi urmează, aceasta este tratată de clauză ***catch*** corespunzătoare.
- După ce clauza ***catch*** este executată, programul continuă cu instrucţiunea imediat următoare blocului ***try***.
- Daca nu există definită nici o clauză ***catch*** corespunzătoare excepţiei, este apelată rutina predefinită `terminate()`, definită în biblioteca standard, care încheie execuţia programului în curs. Implicit, funcţia `terminate()` apelează funcţia `abort()`, indicând un comportament anormal al aplicaţiei
- După încheierea execuţiei clauzei ***catch***, dacă aceasta nu conţine instrucţiunea ***return***, execuţia programului continuă cu instrucţiunea ce urmează după ultima clauză ***catch*** din listă.
- În situaţia în care nu s-a produs nici o eroare, clauza ***catch*** nu se execută.

Exemplul 1. Eroarea poate fi prinsă şi eventual tratată chiar de către funcţia în sine:

```
……
        void f(int test)
        {
            ……
            try
           {
            ……
              if(test)
                 throw test;
              else
                 throw “Eroare”;
           }
           catch(int i)
           {
               cout<<”Am prins numarul”<<i<<endl;
           }
           catch(char *p)
           {
              cout<<”Am prins sirul”<<p<<endl;
           }
       }
```

Exemplul 2. Excepţia poate fi „pasată”

```
#include <iostream>

void f1()
{
    try {
    throw "f1";
    }
    catch(char *p)
    {
          throw;
    }
}


void f2()
{
    try{
          f1();
     }
    catch(char *p)
    {
        throw;
    }
}
```

```
void f3()
{
    try{
       f2();
    }
    catch(char *p)
    {
        throw;
    }
}

int main(void)
{
    try{
        f3();
    }
    catch(char *p)
    {
         cout<<p<<endl;
    }
    cin.get();
    return 0;
}
```

Exemplul 3. Tratarea erorilor in cazul clasei Stiva

```
#include<iostream>
const int MAX=10;
class Stack
{
    int st[MAX];
    int top;

public:
    class Full
    {
    };
    class Empty
    {
    };
    Stack()
    {
        top=-1;
    }
    int push(int val)
    {
        if(top>=MAX-1)
            throw Full(); //Constructor!
        return st[++top]=val;
    }
    int pop()
    {
        if(top<0)
            throw Empty(); //Constructor!
        return st[top--];
    }
};
int main(void)
{
    Stack s;
    // int imax=20;
    int imax=10;
    try
    {
        int i;
        for(i=0;i<imax;i++)
            cout<<s.push(i)<<'\t';
        cout<<endl;
        for(i=0;i<20;i++)
            cout<<s.pop()<<'\t';

    }
    catch(Stack::Full)
    {
        cout<<"\nEXCEPTIE :Stiva plina\n";
    }
    catch(Stack::Empty)
    {
        cout<<"\nEXCEPTIE :Stiva goala\n";
    }
    return 0;
}
```

```
/* imax=20
0    1    2    3    4    5    6    7    8    9
EXCEPTIE :Stiva plina

imax=10
0    1    2    3    4    5    6    7    8    9
9    8    7    6    5    4    3    2    1    0
EXCEPTIE :Stiva goala  */
```

Clauza catch- all

- Există situaţii în care se doreşte tratarea unor aspecte comune mai multor clauze ***catch*** sau rezolvarea unor probleme privitoare la eliberarea resurselor proprii secţiunii ce lansează excepţia înainte de lansarea propriu-zisă, ca în exemplul de mai jos:

```
void manip() {
   resource res;
   res.lock();     // alocă o resursă
  // foloseşte res
  // acţiuni ce pot declanşa lansarea de excepţii
   res.release(); // neapelat daca se lansează excepţii
}
```

- Pentru a garanta eliberarea resurselor se poate folosi clauza ***catch***-all care are sintaxa:

```
catch ( ... ) {
  //
}
```

- În clauza ***catch***-all se intră indiferent de tipul excepţiei aruncate şi ea se utilizează în mod obişnuit în combinaţie cu clauza ***throw***

Exemplul de mai sus se rescrie >

```
void manip() {
   resource res;   res.lock();
   try {
      //
   }
   catch (...) {
        res.release();
        throw;
   }
   res.release(); // pentru cazul în care nu s-au lansat excepţii
}
```

Template-uri în C++.

- Pentru a permite o utilizare optimă a codului, limbajul C++ permite crearea unor şabloane (*template*) pe baza cărora compilatorul poate crea adecvat clase sau funcţii.
- Prin intermediul acestei facilităţi, clasele şi/sau funcţiile sunt capabile să opereze cu aşa-numite tipuri generice, ceea ce elimină necesitatea scrierii de cod separat pentru fiecare tip în parte .
- *Template-ul* implementează aşa-numitul concept de "*tip parametrizat*" ("*parametrized type*").
- Şabloanele (template-urile) nu au făcut parte din specificaţiile originale ale limbajului C++, ci au fost adăugate in 1990, fiind definite de standardul ANSI C++ şi preluate de standardul ISO.
- În acest context, biblioteca standard de template-uri C++ Standard Template Library, inclusă în Standard C++ Library, furnizează clase şi algoritmi extrem de versatili
- Specificarea utilizării şablonului se face în general cu o construcţie de forma :

template **<*class*** tip >

Template-uri de funcţii

- În C++ template-urile de funcţii sunt şabloane ce servesc la crearea unor funcţii similare
- Ideea de bază este aceea de a putea crea funcţii fără a fi nevoie să se specifice tipul exact al unor sau eventual al tuturor variabilelor.
- Un template de funcţie se comportă în principiu ca o funcţie, cu excepţia faptului că template-ul poate avea argumente de tipuri diferite
- Cu alte cuvinte, un template de funcţie reprezintă o familie de funcţii
- Atunci când o funcţie sau o clasă este instanţiată dintr-un template, compilatorul creează o specializare a template-ului pentru setul de argumente utilizat
- Specificarea utilizării template-ului de funcţie se face astfel:

template **<*class*** tip> type func_name(tip arg1, ...);

Template-uri de clase

- În C++ există posibilitatea de a scrie *template*-uri de clase, astfel încât clasele să poată avea membri ce folosesc parametrii template-ului drept tipuri de date
- Specificarea utilizării template-ului de clasă se face astfel:

template **<*class*** tip> ***class*** clasa {...};

Exemplul 1. Să se proiecteze un *template* pentru o funcţie care să calculeze valoarea maximă dintre elementele unui vector cu nr elemente de tipul generic T

```
#include<iostream>

template <class T>
T max_t(T* array, int nr)
{
    T max=array[0];
    for (int i  = 1;i<nr;i++)
      if(max<array[i])
        max=array[i];
    return max;
}

int max_t(int,int);
double max_t(double,int);

int main(void)
{
    int ti[]={1,2,3,4};
    double td[]={.1,.2,.3,.4};
    cout<<"max_int="<<max_t(ti,4)<<endl;
    cout<<"max_double="<<max_t(td,4)<<endl;
    cin.get();
    return 0;
}
```

Exemplul 2. Să se proiecteze un template pentru o clasă care să implementeze o structură de date de tip stivă cu elemente de tipul generic T

```
#include<iostream>

template <class T>
class Stack
{
    T *buff;
    int isempty;
    int isfull;
    int nr_elem;
    int stack_size;
   public:
     Stack(int size);
     T push(T value);
     T pop();
     int is_empty(){
         return isempty;}
     int is_full(){
         return isfull;}
};

template <class T>
Stack<T>::Stack(int size)
{
      buff=new T[size];
      nr_elem=0;
      isempty=1;
      isfull=0;
      stack_size=size;
}

template <class T>
T  Stack<T>::pop()
{
    if(is_empty())
      return 0;
    if(--nr_elem==0)
        isempty=1;
    isfull=0;
    return buff[nr_elem];
}
```

```
template <class T>
T  Stack<T>::push(T value)
{
    if(is_full())
        return 0;
    buff[nr_elem++]=value;
    if(nr_elem==stack_size)
        isfull=1;
    isempty=0;
    return value;
}

int main()
{
  Stack<int>  intstack(10);
  Stack<double> dblstk(15);
  for(int i=0;
   !intstack.is_full();i++)
        intstack.push(i);
  intstack.push(i);

while(!intstack.is_empty())

cout<<intstack.pop()<<endl;

  cin.get();

  return 0;
}
```

Exemplul 3. Să se proiecteze un template pentru o clasă care să implementeze o structură de date de tip listă dublu înlănţuită cu elemente de tipul generic T

```
#include<iostream>
template <class T>
struct Node
{
    T value;    Node *next;    Node *prev;
};
template <class T>
class LinkList
{
        Node<T> *first;    Node<T> *end;
   public:
        LinkList();
        void show_f();    void show_e();
        Node<T> *append(T);
};
template <class T>
LinkList<T>:: LinkList()
{
     first=NULL;    end=NULL;
}
template<class T>
void LinkList<T>::show_f()
{
        Node<T> *node;
        node=first;
        while(node)
        {
                cout<<node->value<<endl;   node=node->next;
    }
}
```

```
template<class T>
void LinkList<T>::show_e()
{
    Node<T> *node;
    node=end;
    while(node)
    {
        cout<<node->value<<endl;  node=node->prev;
    }
}
template<class T>
Node<T> * LinkList<T>::append(T value)
{
    Node<T> *ptr=end;
    end=new Node<T>;
    if(first==NULL)
        first=end;
    else
         ptr->next=end;
    if(end)
    {
        end->next=NULL;    end->prev=ptr;
        end->value=value;
    }
    return end;
}
int main()
{
    LinkList<int> list;
    for(int i=0;i<10;i++)
        list.append(i);
    list.show_f();    cout<<endl;
    list.show_e();     cin.get();
    return 0;
}
```

Standard Template Library (STL)

- Biblioteca Standard Template Library este o bibliotecă generică ce oferă soluţii de prelucrare a unor colecţii de date prin intermediul unor algoritmi moderni şi eficienţi.
- Ca paradigmă de programare se utilizează programarea generică, ce implică existenţa unui aşa-numit polimorfism parametric ; în C++ acest mecanism se implementează prin intermediul ***template*** - urilor .
- Programarea generică reprezintă un stil de programare în care algoritmii sunt concepuţi în termenii unor tipuri de date ce urmează a fi specificate ulterior, prin instanţieri adecvate tipurilor furnizate ca parametri
- Alexander Stepanov, un pionier al programării generice, scria:

  "Generic programming is about abstracting and classifying algorithms and data structures. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream. "

- STL furnizează clase specializate împreună cu algoritmii ce operează asupra lor, componente incluse în spaţiul de nume ***std***.
- Toate componentele bibliotecii sunt *template*-uri, astfel încât ele se pot utiliza pentru orice tip de date.
- Biblioteca Standard Template Library este inclusă în biblioteca standard a limbajului C++, ***C++ Standard Library***

- În concluzie, biblioteca standard C++ are trei mari componente:

  - biblioteca de funcţii C

  - biblioteca de stream-uri de intrare / ieşire

  - Standard Template Library (STL).

- Fişierele header puse la dispoziţie de limbajul C++ sunt:

C++ Standard Library

- ios
- iostream
- iomanip
- fstream
- sstream

Standard Template Library

- vector
- deque
- list
- map
- set
- stack
- queue
- bitset
- algorithm
- functional
- iterator

C Standard Library

- cassert
- cctype
- cerrno
- climits
- clocale
- cmath
- csetjmp
- csignal
- cstdarg
- cstddef
- cstdio
- cstdint
- cstdlib
- cstring
- ctime

Componentele STL

1. **Containere** - sunt destinate stocării unor colecţii de obiecte, gestionând în mod adecvat spaţiul de memorie alocat şi furnizând modalităţi adecvate de acces la elementele lor.
2. **Iteratori -** se utilizează pentru parcurgerea elementelor unui container Avantajul major al iteratorilor este acela că oferă o interfaţă minimală comună tuturor tipurilor de containere, independent de structura internă a acestora. Iteratorii se comportă în principiu ca şi pointerii
3. **Algoritmi -** se utilizează pentru procesare elementelor colecţiei (căutare, sortare etc)

- Conceptul de bază al bibliotecii STL este cel al separării datelor de operaţii : datele sunt gestionate de containere, iar operaţiile asupra lor se efectuează prin algoritmi configurabili adecvat. Iteratorii reprezintă "liantul" dintre aceste două componente.

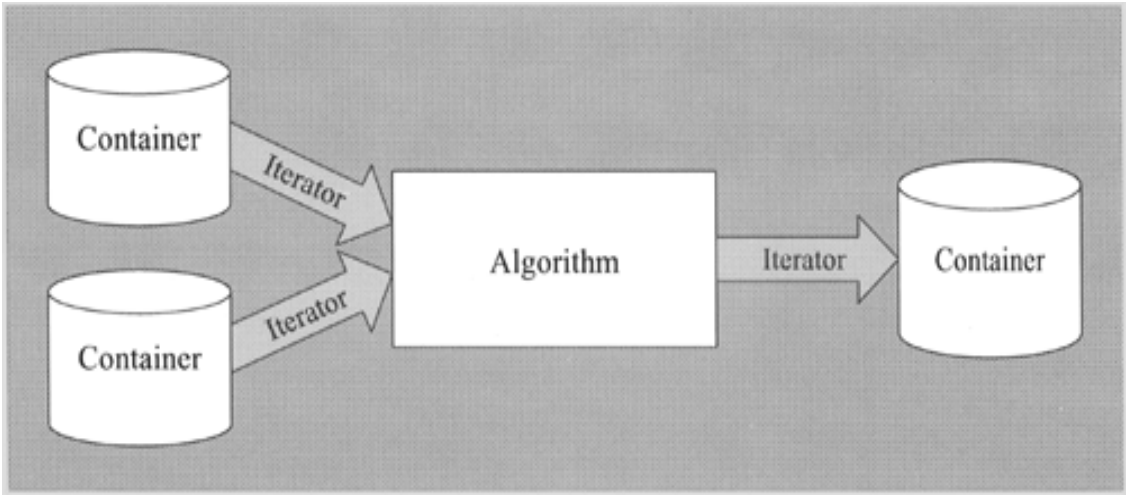

Figura 1. Componentele STL

1. Containere

- Un container este un obiect care păstrează o colecţie de alte obiecte.
- Containerele sunt implementate ca şi ***template***-uri de clase
- Containerele sunt replici ale unor structuri de date frecvent utilizate în programare :
  - Tablouri dinamice (***vector***)
  - Cozi (***queue***)
  - Cozi cu două capete (***deque***)
  - Stive (***stack***)
  - Liste înlănţuite (***list***)
  - Mulţimi/arbori (***set***)
  - Tablouri asociative (***map***)

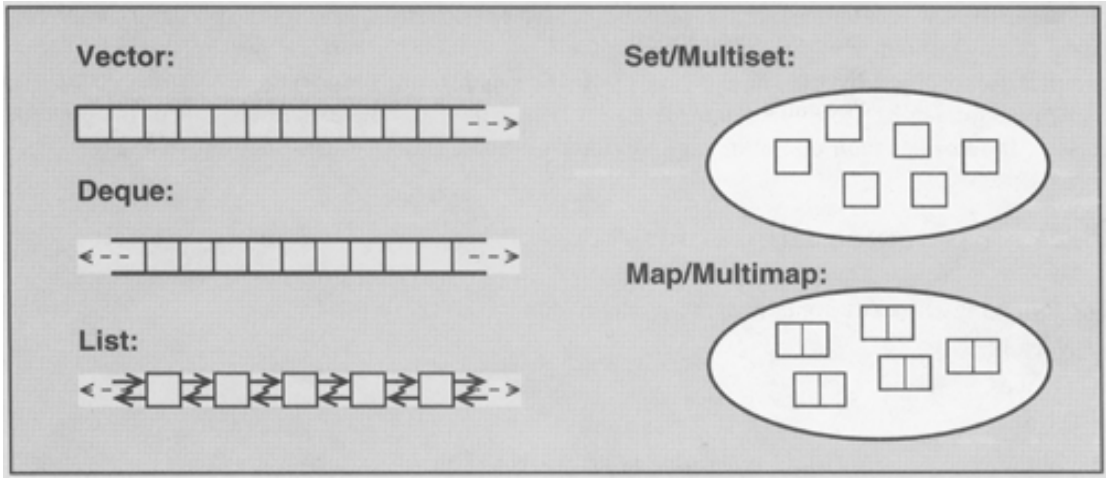

Figura 2.Câteva tipuri de containere

- Containerele pot fi:
  - **Containere** secvenţiale – colecţii ordonate, în care fiecare element are o anumită poziţie ce nu depinde de valoarea sa (***vector, deque*** şi ***list***).
  - **Containere** asociative – colecţii sortate , în care fiecare element are o anumită poziţie ce depinde de valoarea sa, în funcţie de un anumit criteriu de sortare (***set, multiset, map*** şi ***multimap***).

- O clasificare mai riguroasă a containerelor este cea din diagrama de mai jos:

Template-uri de clase container (Container class templates)

Containere simple (Simple containers):

Pair	Pair (pereche)

Containere secvenţiale (Sequence containers):

vector	Vector
deque	Double ended queue
list	List

Adaptori de containere - Container adaptors:

stack	LIFO stack
queue	FIFO queue (coadă)
priority_queue	Priority queue

Containere associative - Associative containers:

set	Set
multiset	Multiple-key set
map	Map
multimap	Multiple-key map
bitset	Bitset

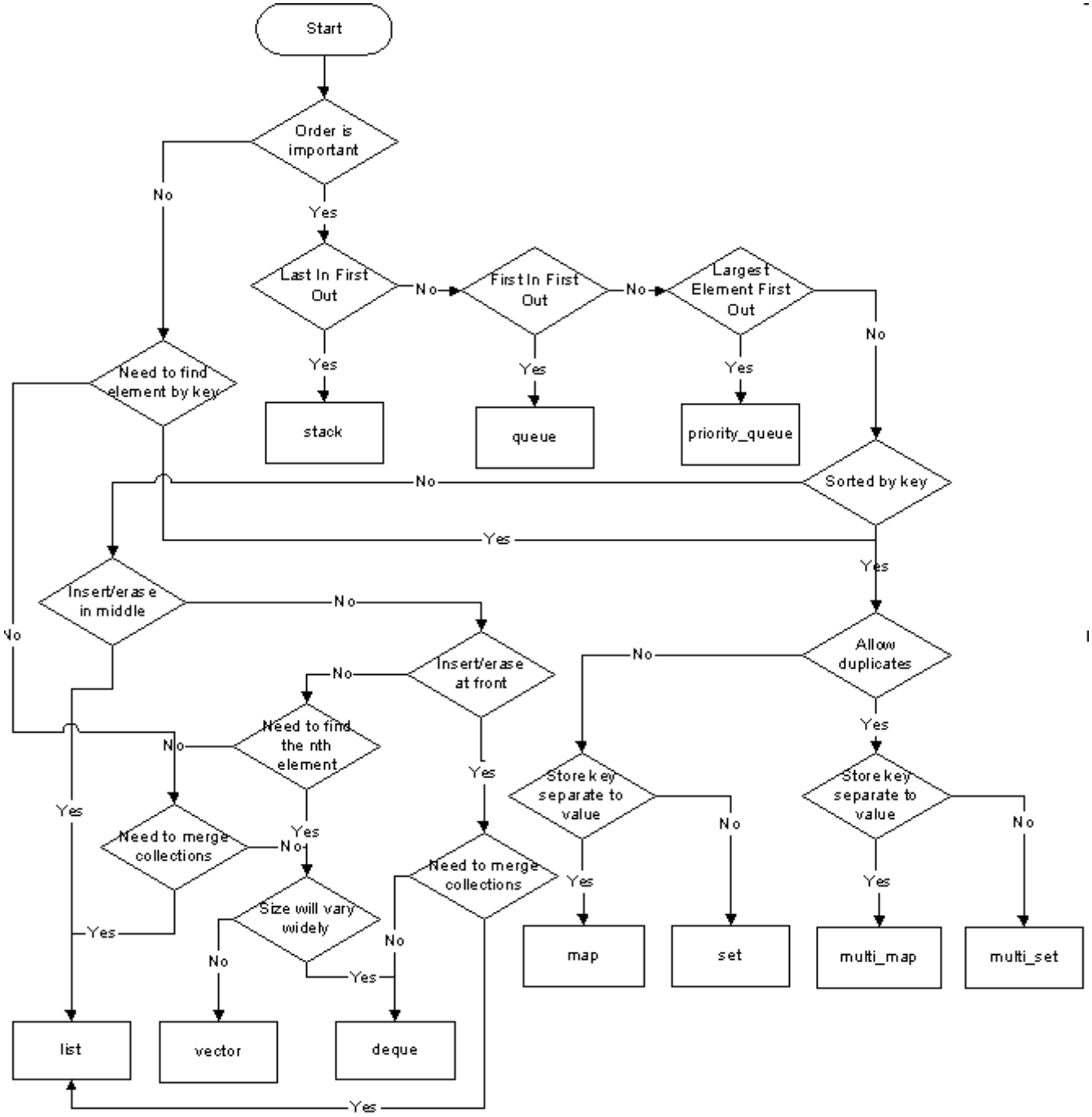

Figura 3. Diagrama de selecţie a tipului adecvat de container

Exemple:

1.1. Vectori

- Un vector gestionează elementele unui tablou unidimensional dinamic, permiţând acces aleator la elementele sale, prin intermediul indexului.
- Adăugarea şi eliminarea elementelor la sfârşitul tabloului sunt operaţii extrem de rapide.
- Inserarea unui element la începutul sau în interiorul tabloului sunt operaţii ce necesită mai mult timp, deoarece este necesară crearea de spaţiu suplimentar pentru noul element şi translarea corespunzătoare a elementelor plasate după poziţia de inserare.

În exemplul de mai jos se defineşte un vector de elemente întregi, se adaugă 6 elemente, care se afişează apoi:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> coll;   // container de tip vector de intregi
    // se adauga elemente cu valori de la 1 la 6
    for (int i=1; i<=6; ++i) {
        coll.push_back(i);
    }
    for (i=0; i<coll.size( ); ++i) {
        cout << coll[i] << ' ';
    }
    cout << endl;
    return 0;
}
```

1.2. Liste

- Listele în STL sunt implementate ca liste dublu înlănţuite, ceea ce înseamnă că fiecare element are un predecesor şi un succesor.
- Avantajul listelor este cel oferit de inserţia şi respectiv ştergerea rapidă a unui element din orice poziţie

Exemplu : Crearea şi afişarea unei liste de caractere

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<char> coll;   // container de tip lista de caractere
    // se adauga elemente de la 'a' la 'z'
    for (char c='a'; c<= 'z'; ++c) {
        coll.push_back(c);
    }
    // afiseaza si sterge primul element
    while (! coll.empty()) {
        cout << coll.front() << ' ';
        coll.pop_front();
    }
    cout << endl;
}
```

2. Iteratori

- Iteratorii sunt obiecte care "iterează" (navighează) printre elementele unui container.
- Iteratorul reprezintă o anumită poziţie într-un container.
- Comportamentul unui iterator este definit de următorii operatori:

- Operatorul *	- returnează elementul aflat pe poziţia curentă. Dacă elementele respective au membri, aceştia pot fi accesaţi direct din iterator prin intermediul operatorului ->.
- Operatorul ++	- avansează poziţia iteratorului peste un element. Majoritatea iteratorilor acceptă şi deplasarea înapoi peste un element, prin intermediul operatorului --.
- Operatorii == şi !=	- testează condiţia ca doi iteratori să refere aceeaşi poziţie.
- Operatorul =	- atribuie unui iterator poziţia referită

- Operaţiile cu iteratori sunt similare operaţiilor cu pointeri; iteratorii pot fi priviţi ca pointeri "inteligenţi" (*smart pointers* — pointeri ce iterează pe structuri de date complexe).
- Comportamentul intern al unui iterator depinde de tipul structurii de date peste care "navighează".
- Fiecare tip de container îşi defineşte propriul său tip de iterator ca o clasă inclusă.
- Ca urmare, iteratorii expun aceeaşi interfaţă, dar pot avea tipuri diferite.
- Acesta este de altfel conceptul **programării generice** : operaţiile utilizează aceeaşi interfaţă, dar tipuri diferite, astfel încât se pot defini ***template***-uri pentru a formula operaţii generice ce se pot efectua asupra unor tipuri arbitrare ce satisfac condiţiile impuse de interfaţă.
- Toate clasele de tip container furnizează aceleaşi funcţii membru de bază ce permit iteratorilor să le parcurgă elementele.

- STL implementează cinci tipuri diferite de iteratori
  - iteratori de intrare (input - folosiţi doar pentru citirea unei secvenţe de valori)
  - iteratori de ieşire (output - folosiţi doar pentru scrierea unei secvenţe de valori)
  - iteratori înainte (forward- folosiţi pentru citire şi scriere ; se pot deplasa înainte)
  - iteratori bidirecţionali (bidirecţional - se pot deplasa înainte şi înapoi)
  - iteratori cu acces aleator (ce se pot deplasa cu orice număr de paşi într-o singură operaţie).

- Cele mai importante funcţii ce operează asupra iteratorilor sunt :

  - *begin*() - returnează un iterator ce reprezintă poziţia de început a elementelor containerului (poziţia primului element, dacă acesta există).

  - *end*() - returnează un iterator ce reprezintă sfârşitul elementelor containerului (poziţia "dincolo" de ultimul element). Un astfel de iterator se numeşte *past-the-end iterator*

- ***begin***() şi ***end***() definesc un domeniu semideschis ce include primul element, dar îl exclude pe ultimul.

- Domeniul semideschis are următoarele avantaje :

  - oferă un criteriu simplu pentru testarea sfârşitului pentru ciclurile ce iterează peste elementele unui container (practic se continuă atâta timp cât nu s-a atins sfârşitul).

  - oferă o modalitate simplă de tratare a domeniilor vide (în acest caz, *begin*() este egal cu *end*().

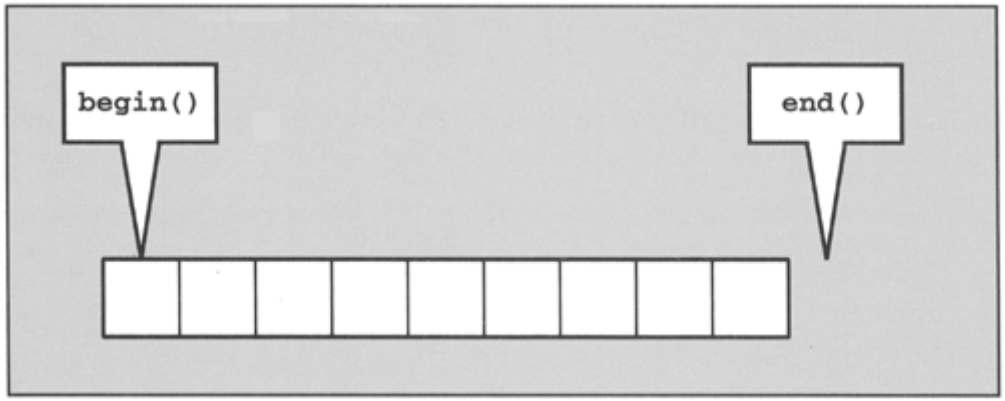

Figura 4. ***begin()*** şi ***end()*** pentru containere

- În mod analog se pot defini şi funcţiile ***rbegin()*** şi respectiv ***rend()*** pentru parcurgerea în ordine inversă a elementelor containerului.
- Din punctul de vedere al posibilităţii modificării elementului pe care îl referă, există două tipuri de iteratori:

  - *container::* `iterator` pentru exploatare read/write.

  - *container::* `const_iterator` pentru exploatare read-only.

De exemplu, în clasa ***list*** definiţia ar putea fi următoarea:

```
namespace std {
    template <class T>
    class list {
      public:
        typedef ... iterator;
        typedef ... const_iterator;
        ...
    };
}
```

Exemplul 2.1 . Parcurgerea unei liste de caractere cu iteratori

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<char> coll;
    // adauga elemente de la 'a' la 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }
    list<char>::const_iterator pos;   // iterator constant - read-only
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ' ;
    }
    list<char>::iterator pos1;      // iterator ne-constant - read/write
    for (pos1 = coll.begin(); pos1 != coll.end(); ++pos1) {
         *pos1 = toupper(*pos1);
         cout << *pos1 << ' ' ;
     }
    cout << endl;
    return 0;
 }
```

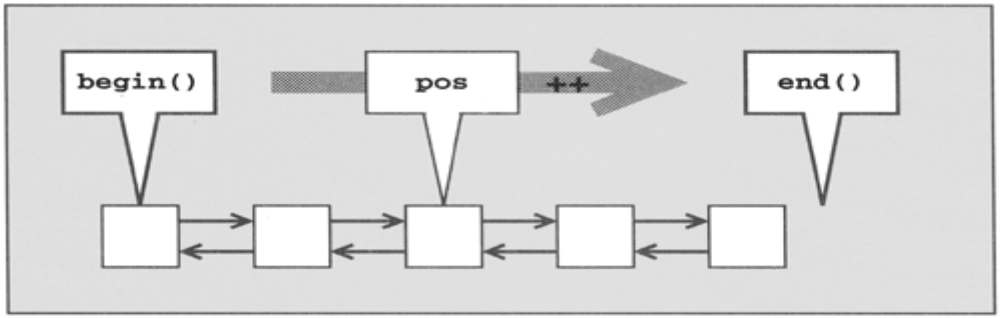

Figura.5. Iteratorul *pos* navigând peste elementele unei liste

Algoritmi

- Biblioteca STL pune la dispoziţia programatorilor un set de template-uri de funcţii ce pot opera cu tipuri de date generice
- Câteva dintre cele mai frecvent utilizate funcţii sunt : ***sort***, ***find, reverse, max_element***

Exemplul 3.1 Crearea unui vector de caractere si parcurgerea sa, ilustrarea algoritmului ***sort***

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

using namespace std;
int main()
{
    vector<char> coll;
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    vector<char>::const_iterator pos; // iterator read-only

    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ' ;
    }
    cout<<endl;

    vector<char>::iterator pos1;     // iterator  read/write
    for (pos1 = coll.begin(); pos1 != coll.end(); ++pos1) {
         *pos1 = toupper(*pos1);
         cout << *pos1 << ' ' ;
     }
     cout << endl;

     sort(coll.rbegin(),coll.rend());
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
      cout << *pos << ' ' ;
    }
    cout << endl;
    return 0;
 }
```

Rezultate

a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

Exemplul 3.2 Crearea unei matrici de double, crearea unui ***deque*** (coadă cu două capete) şi ilustrarea algoritmului ***max_element***

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


using namespace std;

int main(void)
{

    int i,j;
    vector<vector<double> > mat(4, vector<double>(3,-1));
 // matrice 4x3 initializata cu -1
    for(i=0;i<4;i++)
    {
        for(j=0;j<3;j++)
                    cout<<mat[i][j]<<' ';
        cout<<endl;
    }

    cout<<endl;

    deque<int> d;
    d.push_back(10);
    d.push_front(100);

    cout << *max_element(d.begin(), d.end()) << endl;

    return 0;
}
```