Course STL

Overview

- **Introduction in STL**
- **STL containers**
- **Iterators**
- **Algorithms**
- **Function objects**

Introduction in STL

- STL (Standard Template Library) as a whole is a component library (basically a **framework** of containers and algorithms of ISO C++), which ensures **interoperability** between **predefined components** (language included) and **user-defined** components.
- Thus, an **STL algorithm** can also operate on **user-defined containers and user-defined algorithms can also work on STL containers** if they meet certain requirements.
- The library was designed by Alex Stepanov and Meng Lee at HP's Palo Alto laboratories. The STL library was included in the ANSI C++ standard in 1994 and accepted in 1998.
- STL as a major positive element leads to:

- Separate compilation of templates

- Better support for generic programming

- Generalized list initialization mechanism, etc.

- Versions C++ 1y/2z brings new improvements to basic containers as well as additional features

STL structure

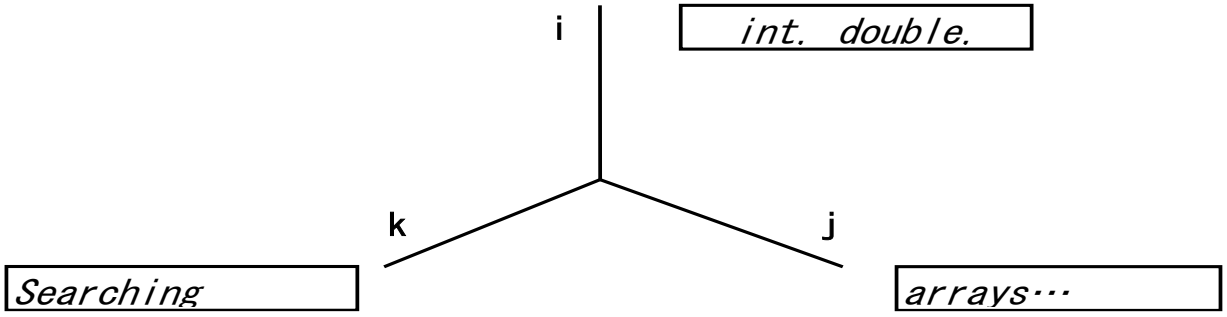
Consider the following situation where **software components** are imagined as a **multi-dimensional space**:

- a dimension represents **data types** (*int, char, float, double,* ...) **(i)**
- another dimension is **containers** (*arrays, linked lists, ...*) **(j)**
- and the last dimension is **algorithms** (*searching, sorting, ...*) **(k)**
- In this case, *i * j * k* code versions must be designed to cover all possible situations.

If we use the template **(generic) functions/classes**, the "i" axis may be missing and ***j * k*** code versions are needed. For example, we will have a single implementation of the *linked list* for all types of data.

The next step is to make algorithms work for **different types of containers** (arrays, lists, ...). This will only require ***j + k*** code versions.

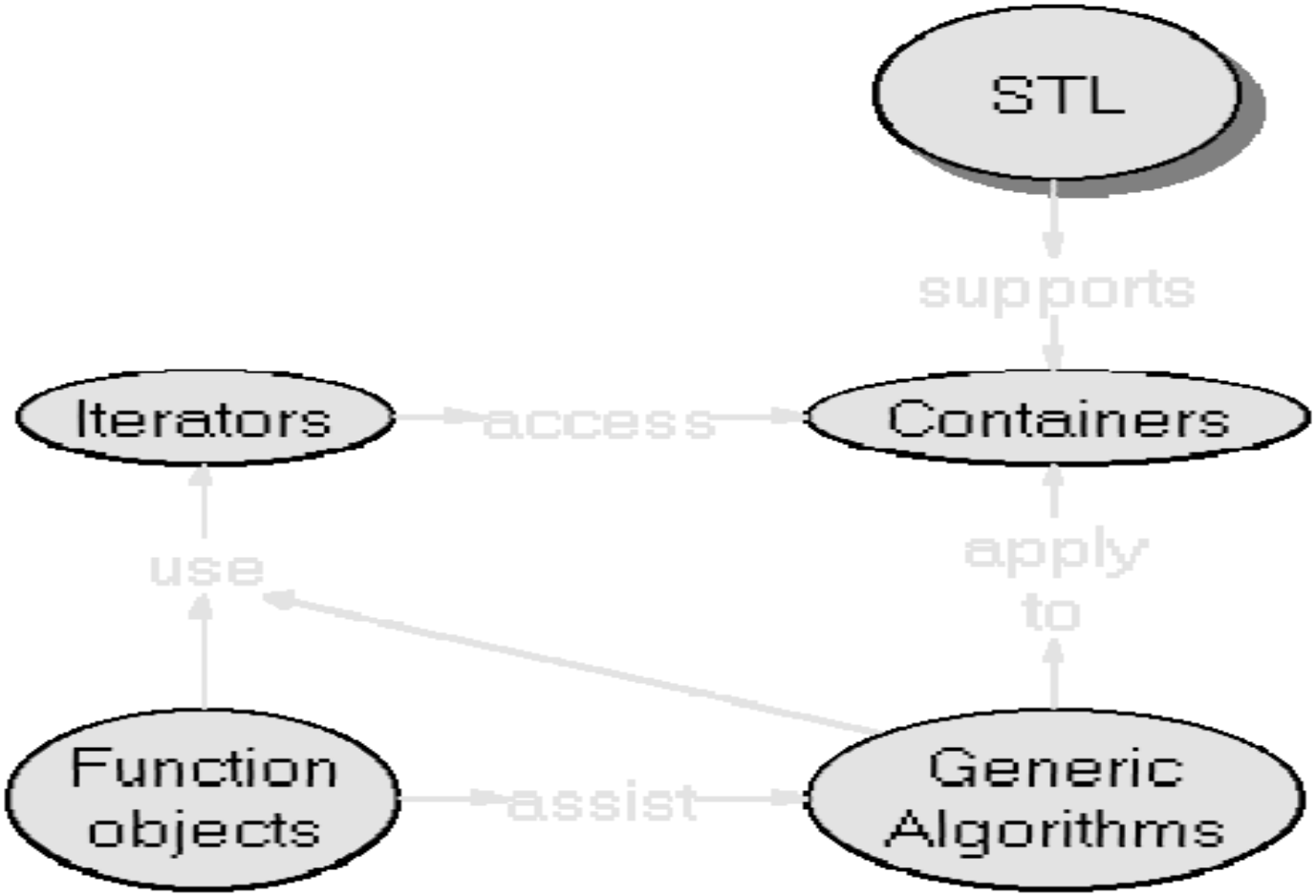
- STL incorporates this concept and, as a result, **simplifies the application development process** by reducing creation time, simplifying debugging and increasing portability.



STL components

- The main components of STL are:
- - **containers**: objects that can store and manage objects; contain the data structures supported by the STL; for this, defines template classes that contain these structures and methods for data manipulation;
- - **algorithms:** procedures that can operate on different containers;
- - **iterators**: abstractions of algorithm access to containers so that the algorithms can operate on different containers (in fact they are pointers to containers but have less flexibility than ordinary pointers, flexibility given by the nature of the containers);
- - **function objec**ts: class data that have overloaded the function call operator, ( );
- - **adapter**: encapsulates a component to provide another interface (for example, to get a *stack* from a *list*).

STL diagram


STL
supports
Iterators
access
Containers
use
apply
to
Function
objects
assist
Generic
Algorithms

How is used STL

STL usage:

- everything related to the STL is placed in the ***standard namespace***, *std*; this must be specified by a *using* directive:

using namespace std;

- there is also the possibility to use **only a section** of this namespace as follows:

*using namespace **std:: string**;*

Starting from the standard C++ library with 51 header files, 13 header files make up the original STL library.

These are: *algorithm, deque, functional, iterator, vector, list, map, memory, numeric, queue, set, stack, utility*.

- Are used by:

#include <algorithm>

using namespace std;

The STL Library **allows expansion**, being based by only one core. Extensions are defined based on imposed rules.

STL containers

There are types of abstract data (classes) that can be used to build data collections of the same type.

Characteristics :

- All **containers** are *parameterized* by the ***type*** they contain
- Each container declares an ***iterator*** and special methods for iterators

Categories :

1. **Sequence container**: The data is ordered in a linear fashion and allows *searches* based on the ***key*** (***array***, *vector,* ***forward_list****, list, deque*). There are ordered collections in which each element has a certain position. The term "ordered" does not mean ascending or descending but refers to a particular position. This *position depends* on the *time* and *place* of insertion but is independent of the element's value.
2. **Associative containers**: data is kept in appropriate data structures for *associative searches* (*set, map, multiset, multimap*). Are *sorted collections* where the actual position of an item depends on its value due to a particular sorting criterion. *C++1y/2z* introduced *Unordered Associative Containers* that are *unsorted collections (Hash collections)*.
3. **Adapters**: Provides different but specific interfaces for the above containers. They are built on other containers and are used to force access rules that do not support iterators. *Stack* and *queue* containers are made from *deque*.
4. **Special containers**: are almost containers with some limitations (*string, bitset, valarray*), and are usually not considered as a separate category. They are especially used to provide *additional facilities* to manage this data effectively in the program development process.

STL containers

Categorie	Container	Caracteristici
Secvenţiale	vector	Pastrează datele în mod liniar şi într-o zonă contiguă de memorie. Similar cu tablourile (liste simplu inlănţuite). Permit inserări rapide dar numai la sfârşit.
	list	Lista dublu inlanţuită ce permite inserări rapide în orice poziţie.
	deque	Organizare liniară dar stocare neliniară. Permite inserări rapide la extremităţi.
Asociative	multiset	Implementare pentru tipul set dar sunt permise duplicări. Permite căutări asociative rapide.
	set	Implementare pentru tipul set dar fără duplicări. Permite căutari asociative rapide.
	multimap	Permite implementarea unei structuri în care se foloseşte o mapare cheie-> mai multe valori (1 to *).
	map	Permite implementarea unei structuri în care se foloseşte o mapare cheie-> o valoare (1 to 1).
Adaptoare	stack	Implementarea unei structuri Last In First Out (LIFO).
	queue	Implementarea unei structuri First In First Out (FIFO).
	priority_queue	O coadă ce păstrează elementele sortate.

STL containers: https://www.cplusplus.com/reference/stl/

Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

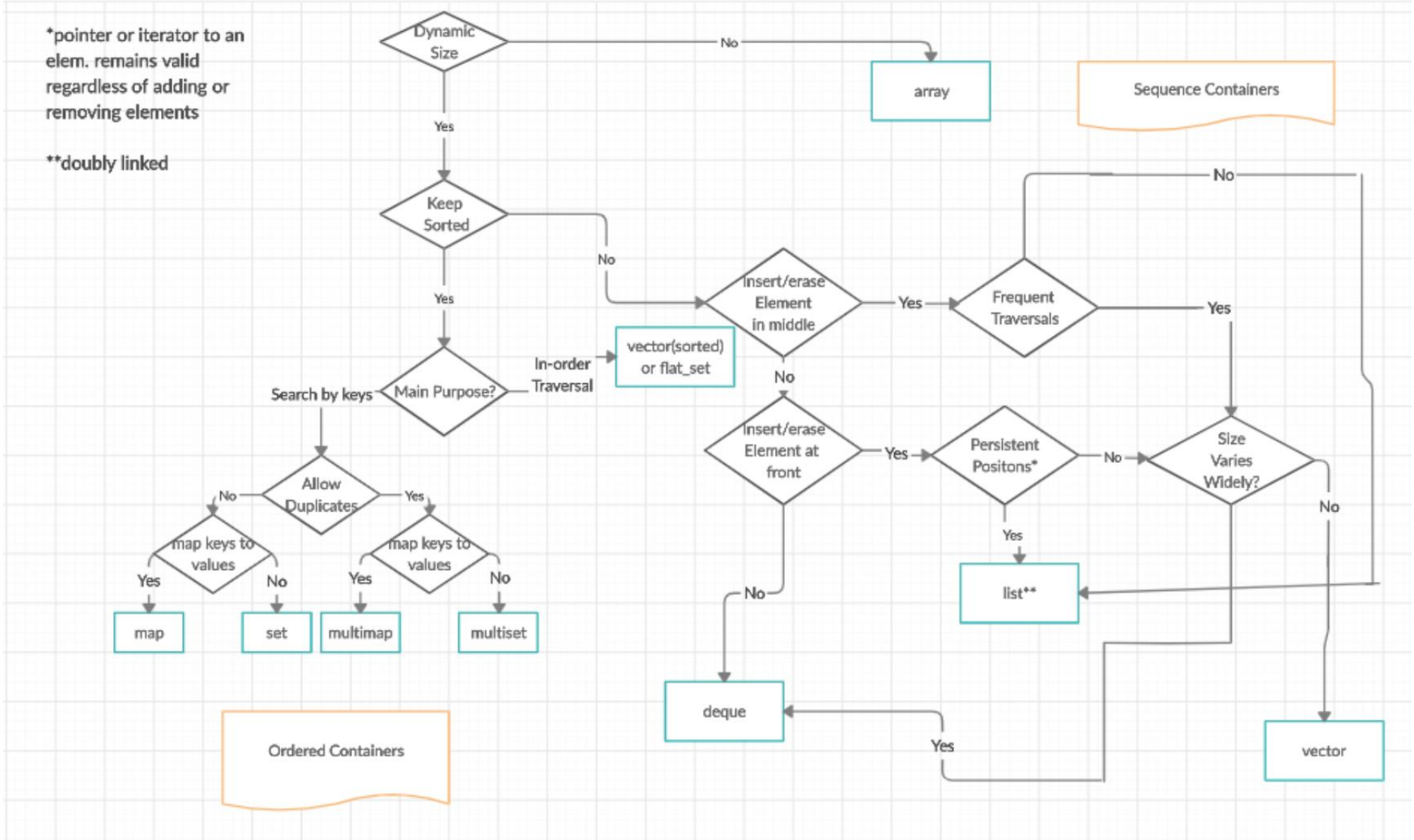
set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

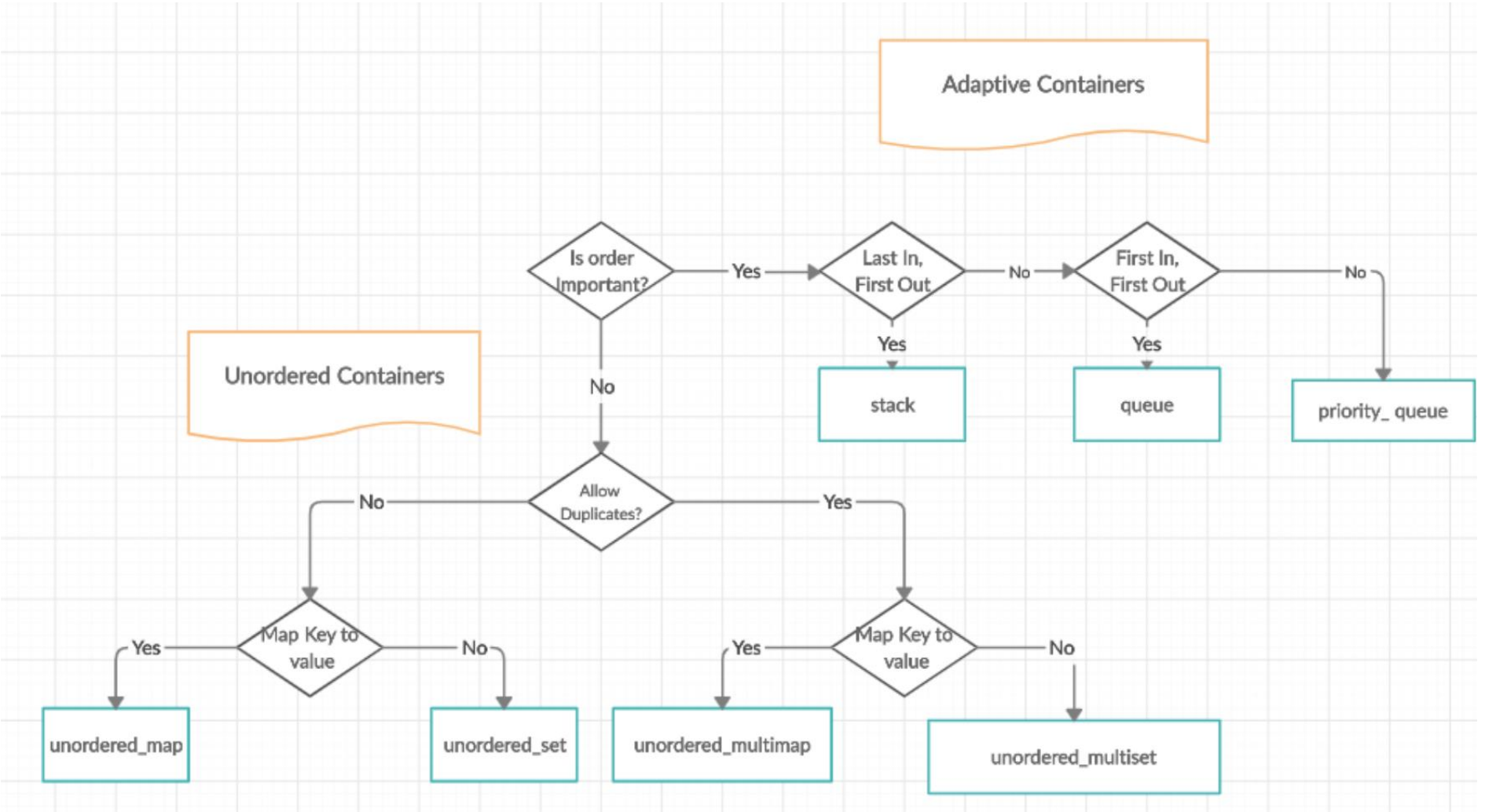
unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Other:
Two class templates share certain properties with containers, and are sometimes classified with them: `bitset` and `valarray`.

STL ordered and sequence containers - https://www.geeksforgeeks.org/the-c-standard-template-library-stl/



STL Adaptive and unordered containers:



```
//Sorting  (qsort) example –Special string container
//Student.h

const int dim_note = 3;

class Student {
        string name;
        string surname;
        int *marks;
        int group;
        double avg_mark;
public:
        Student( ){
                name= "Unknown";
                surname= "Unknown";
                group = 1;
                marks = new int[dim_note];
                for (int i = 0; i < dim_note; i++) *(marks + i) = 10;
                avg_mark = 10;
        }
```

```
Student(string n, string p, int gr, int *m)
        {       name= n;
                surname= p;
                group = gr;
                marks = new int[dim_note];
                for (int i = 0; i < dim_note; i++) {*(marks + i) = m[i];  }
                avg_mark = media( );
        }
~Student( ){delete[ ]marks;     }
double media( ){
                int s = 0;
                for (int i = 0; i < dim_note; i++) s += marks[i];
                return ((double)s / dim_note);
        }
void setName(string n){name= n; }
string getName( ){return name;}
void setSurname(string p){name= p;}
string getSurname( ){return surname;}
void setGroup(int gr){group=gr;}
int getGroup( ){return group;}
double getMedia( ){ return avg_mark;}
};//Student class
```

```
int cmp_int(Student *a, Student *b) {
        return (int)(((Student *)b)->getGroup( ) - ((Student *)a)->getGroup( ));//valori intregi
}
int cmp_double(void *a, void *b) {//varianta cu parametrii void *
        if (((Student *)b)->getMedia( ) > ((Student *)a)->getMedia( )) return 1;
        else if (((Student *)b)->getMedia( ) == ((Student *)a)->getMedia( )) return 0;
        else return -1;
}
int cmp_str(Student *a, Student *b) {
        if (a->getName( )> b->getName( )) return 1;
        else        if (a->getName( )< b->getName( )) return -1;
        if (a->getSurname( )> b->getSurname( )) return 1;
        else        if (a->getSurname( )< b->getSurname( )) return -1;
        return 0;
}

//main
//#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#include <string>
#include "Student.h"
int main( ){
        int n;
        string na, sur;
        int group, me[dim_note];
        cout << "\nEnter number of students: ";
        cin >> n;
        Student *tab = new Student[n];
```

```
for (int i = 0; i < n; i++) {cout << "\nEnter name: ";
                            cin >> na;
                            cout << "\nEnter surname: ";
                            cin >> sur;
                            cout << "\nEnter group: ";
                            cin >> group;
                            for (int i = 0; i < dim_note; i++){
                                    cout << "\nEnter mark " << i + 1 << ": ";
                                    cin >> *(me + i);}
                            tab[i] = Student(na, sur, group, me);
                }
                cout << "\nInitial array: \n";
for (int i = 0; i < n; i++)             cout << i + 1 << ". " << tab[i].getName( ) << " " <<
tab[i].getSurname( ) << " Group:" << tab[i].getGroup( ) << " Average mark: " << tab[i].getMedia( ) <<
endl;
                qsort(tab, (size_t)n, sizeof(*tab), (int(*)(const void*, const void*))cmp_double);
                cout << "\nSorted array by media: \n";
for (int i = 0; i < n; i++)             cout << i + 1 << ". " << tab[i].getName( ) << " " <<
tab[i].getSurname( ) << " Group:" << tab[i].getGroup( ) << " Average mark: " << tab[i].getMedia( ) <<
endl;
                qsort(tab, (size_t)n, sizeof(*tab), (int(*)(const void*, const void*))cmp_int);
                cout << "\nSorted array by Group: \n";
for (int i = 0; i < n; i++)             cout << i + 1 << ". " << tab[i].getName( ) << " " <<
tab[i].getSurname( ) << " Group:" << tab[i].getGroup( ) << " Average mark: " << tab[i].getMedia( ) <<
endl;
                qsort(tab, (size_t)n, sizeof(*tab), (int(*)(const void*, const void*))cmp_str);
                cout << "\nSorted array by Name & Surname: \n";
for (int i = 0; i < n; i++)             cout << i + 1 << ". " << tab[i].getName( ) << " " <<
tab[i].getSurname( ) << " Group:" << tab[i].getGroup( ) << " Average mark: " << tab[i].getMedia( ) <<
endl;
                }//main
```

Other containers elements

From the initial STL framework **new elements** appeared as:

-*forward_list* as a sequential container for **single linked lists, non-contiguous** memory allocation as for *list,* that it is a **double linked list**

-*unordered_map, unordered_set*, as **hash associative containers** that can be *multimap* and *multiset*

-*array* as an unidimensional fixed-size container

-*bitset*, as bool array container.

C++98 introduced a special container called *valarray* to hold and provide **mathematical operations** on arrays efficiently.

It supports element-wise mathematical operations and various forms of generalized subscript operators, slicing and indirect access.

As compare to vectors, *valarray*-s are efficient in certain mathematical operations than vectors.

The declaration of a container is as follows:

Container <Type_concrete> c; // container object

The specific *Type_concrete* (user type) must provide the **copy constructor** and the **overloading of the assignment operator**.

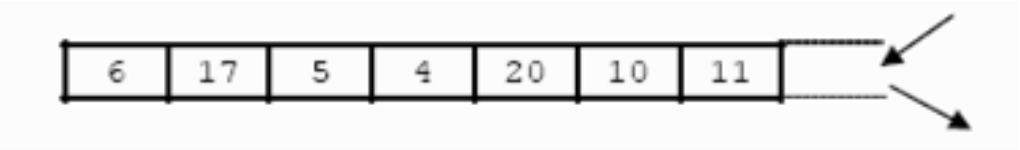
Example:

vector < int> tab1(10) ;

vector <float> tab2(20) ;

vector container

- A *vector* manages its elements in a dynamic array using contiguous memory locations. This allows random access. Adding and removing items at the end of the array is very fast.
- The structure of a *vector* looks like this:



A *vector* is described by a template class with implicit parameters, as follows:

template < typename T, typename Alloc = allocator<T> > class vector; *//generic template prototype*

-*T* is the type of the elements in the container

-*Alloc*, class allocator that accepts as default template argument, *allocator*<*T*>.

-***deque*** and ***list*** classes have the same header template, only the names are different.

For a *vector* container type, it is sufficient to specify only its type, the allocator being the default provided by the STL.

vector <int> v ;

vector <bool> b; is a specialization of the vector container, introduced in later versions of C++.

Main *vector* methods

begin ( ) returns an iterator to the first element

end ( ) returns an iterator after the last element

rbegin ( ) returns an inverse iterator, reverse iterator that points to the last element in the vector

rend ( ) returns an inverse iterator, reverse iterator that points to the theoretical element preceding the first element of the vector

cbegin ( ) returns a constant iterator to the first element

cend ( ) returns a constant iterator after the last element

push_back (...) adds an element to the end of the vector

pop_back (...) extracts an element at the end

swap (...) changes two vector elements, and ***swap (,)*** changes the elements of two containers

insert (,) inserts an element

erase (,) delete an item (or more)

size ( ) returns the number of elements in the vector

capacity ( ) gives the capacity (number of elements) before makes a new reallocation

reserve ( ) allocates space for a number of elements

resize (,) resizes a vector

empty ( ) returns *True* if the vector is empty

[ ] access operator

...

The following example defines an integer value *vector*, inserts 10 elements, and prints the *vector* elements:

```
//vector simple example
#include <iostream>
//vector header file
#include <vector>
using namespace std;
const int dim = 10;

int main( ) {
        //vector container for integer elements
        //declaration
        vector<int> coll;
        //append elements with values from 1 to dim
        for (int i = 1; i <= dim; ++i)
                coll.push_back(i);
        //print all elements separated by a space
        for (int unsigned i = 0; i < coll.size( ); ++i)
                cout << coll[i] << ' ';
        cout << endl;
        return 0;
}
```

The following example defines an integer value *vector*, managed with for-range:

```
//vector for-range example
#include <iostream>
#include <vector>
//using namespace std;

int main( ) {
std::vector<int> v = { 0, 1, 2, 3, 4, 5 };
for (const int& i : v) // access by reference to const values
std::cout << i << ' ';
std::cout << '\n';
for (auto i : v) // access by value, the type of i is int
std::cout << i << ' ';
std::cout << '\n';
for (auto&& i : v) // access by forwarding reference, the type of i is int&
std::cout << i << ' ';
std::cout << '\n';

const auto& cv = v;
for (auto&& i : cv) //access by forward reference, the type of i is const int&
std::cout << i << ' ';
std::cout << '\n';
}
```

The following example analyzes the operators within a *vector*:

```
//vector, operators
#include <vector>
#include <iostream>
using namespace std;
const int dim =10;

int main( )
{//vector container for integer elements
                unsigned int i;
               vector<int> vec1, vec2, vec3;
cout<<"vec1 data: ";
//append elements with values from 1 to dim
for(i=1; i<=dim; ++i)
vec1.push_back(i);
//print all elements separated by a space
for(i=0; i<vec1.size( ); ++i)
cout<<vec1[i]<<' ';
cout<<endl;
cout<<"vec2 data: ";
//append elements with values 1 to dim
```

```
for(i=11; i<=2*dim; ++i)
vec2.push_back(i);
//print all elements separated by a space
for(i=0; i<vec2.size( ); ++i)
cout<<vec2[i]<<' ';
cout<<endl;
cout<<"vec3 data: ";
//append elements with values 1 to dim
for(i=1; i<=dim; ++i)
vec3.push_back(i);
//print all elements separated by a space
for(i=0; i<vec3.size( ); ++i)
cout<<vec3[i]<<' ';
cout<<"\n\n";
cout<<"Operation: vec1 != vec2"<<endl;
if(vec1 != vec2)
cout<<"vec1 and vec2 is not equal."<<endl;
else
cout<<"vec1 and vec2 is equal."<<endl;
cout<<"\nOperation: vec1 == vec3"<<endl;
if(vec1 == vec3)
cout<<"vec1 and vec3 is equal."<<endl;
else
cout<<"vec1 and vec3 is not equal."<<endl;
```

```
cout<<"\nOperation: vec1 < vec2"<<endl;
if(vec1 < vec2)
cout<<"vec1 less than vec2."<<endl;
else
cout<<"vec1 is not less than vec2."<<endl;
cout<<"\nOperation: vec2 > vec1"<<endl;
if(vec2 > vec1)
cout<<"vec2 greater than vec1."<<endl;
else
cout<<"vec2 is not greater than vec1."<<endl;
cout<<"\nOperation: vec2 >= vec1"<<endl;
if(vec2 >= vec1)
cout<<"vec2 greater or equal than vec1."<<endl;
else
cout<<"vec2 is not greater or equal than vec1."<<endl;
cout<<"\nOperation: vec1 <= vec2"<<endl;
if(vec1 <= vec2)
cout<<"vec1 less or equal than vec2."<<endl;
else
cout<<"vec1 is not less or equal than vec2."<<endl;
return 0;
}
```

Vector, swap( ) method for vectors change

```
//vector, swap( )
//vector, swap( )
#include <vector>
#include <iostream>
using namespace std;

int main( ){
vector <int> vec1, vec2;
vec1.push_back(4);
vec1.push_back(7);
vec1.push_back(2);
vec1.push_back(12);
cout << "vec1 data: ";
for (auto &i: vec1) cout << i << ' ';
cout << endl;
vec2.push_back(11);
vec2.push_back(21);
vec2.push_back(30);
cout << "vec2 data: ";
for (auto &i : vec2) cout << i << ' ';
cout << endl;
```

```
cout << "The number of elements in vec1 = " << vec1.size( ) << endl;
cout << "The number of elements in vec2 = " << vec2.size( ) << endl;
cout << endl;
cout << "Operation: vec1.swap(vec2)\n" << endl;
vec1.swap(vec2);//swap vectors
cout << "The number of elements in v1 = " << vec1.size( ) << endl;
cout << "The number of elements in v2 = " << vec2.size( ) << endl;
cout << "vec1 data: ";
for (auto &i :vec1) cout << i << ' ';
cout << endl;
cout << "vec2 data: ";
for (auto &i :vec2) cout << i << ' ';
cout << endl;
return 0;
}
```

Vector, User data example

```
//Factura.h
const int NL=5;

class  Factura {
    string  factura;
    int  nrLucrari;
    vector<double>  preturi;
    double  pretMediuFacturi;

public:
 Factura( ) {
        nrLucrari = NL;
        factura = "Nespecificat";
    }//constructor fara parametrii
    Factura(const  string  f, int  nr, vector<double>pret, double  p) :
        factura{ f }, nrLucrari{ nr }, preturi{ pret }, pretMediuFacturi{ p }{ }//constructor cu parametrii
    Factura(const  Factura& ot) :
        factura{ ot.factura }, nrLucrari{ ot.nrLucrari }, preturi{ ot.preturi }, pretMediuFacturi{ ot.pretMediuFacturi }
    {  }  //  constructor  de  copiere
    string  getFactura( ) {
        return  factura;
    }
    int  getNrLucrari( ) {
        return  nrLucrari;
    }
    void  setFactura(const  string  f) {
        factura = f;
    }
    void  setLucrari(int  l) {
        nrLucrari = l;
    }
```

```
void  setPreturi(vector<double>  v) {
        preturi = v;
    }
    const  vector  <double>  getPreturi() {
        return  preturi;
    }
    double  pret(vector<double>  v) {
        double  p = 0;
        for (const auto f : v) p += f;
        return  p;
    }
    void  setPretMediu(double  p) {
        pretMediuFacturi = p;
    }
    void  medie(vector<Factura>  v) {
        for (auto f : v) f.setPretMediu(pret(f.preturi));
    }
    double  TVA(Factura  f, int  tva) {
        double  p = pret(f.preturi);
        double  q = (tva / 100.) * p;
        p = p + q;
        return  p;
    }
};//Factura
```

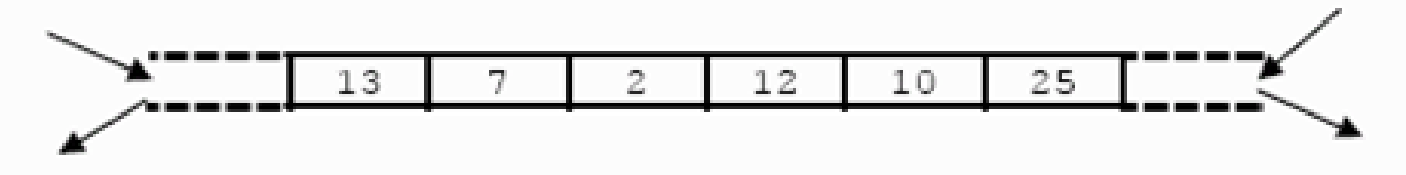
```
//main
#include<iostream>
#include  <vector>
#include  <string>
using  namespace  std;
#include "Factura.h"

const int dimP = 4;
const int dimL = 4;
const int TVAF = 20;

int  main( ) {
    vector<Factura>  v;//declarare  vectorul  de  facturi
    vector<double>  p1;//declarare vector  de  preturi
    p1.push_back(15);
    p1.push_back(30);
    p1.push_back(30);
    p1.push_back(45);
    Factura  f1;
    f1.setFactura("Constructii");
    f1.setLucrari(dimL);
    f1.setPreturi(p1);
    vector<double>  p2;//declarare vector  de  preturi
    p2.push_back(15);
    p2.push_back(10);
    p2.push_back(20);
    p2.push_back(30);
    Factura  f2;
    f2.setFactura("Montaj");
    f2.setLucrari(dimL);
    f2.setPreturi(p2);
```

```
vector<double>  p3;//declarare vector  de  preturi
   p3.push_back(15);
   p3.push_back(11);
   p3.push_back(22);
   Factura  f3;
   f3.setFactura("Amenajari");
   f3.setLucrari(dimL - 1);
   f3.setPreturi(p3);

   vector<double>  p4;//declarare vector  de  preturi
   p4.push_back(33);
   p4.push_back(44);
   Factura  f4;
   f4.setFactura("Pavaje");
   f4.setLucrari(dimL - 2);
   f4.setPreturi(p4);

   v.push_back(f1);//atasare facturi vectorul  de  facturi
   v.push_back(f2);
   v.push_back(f3);
   v.push_back(f4);

   cout << "\nAfisare obiecte vector facturi:";
   for (auto f : v) {
      cout << "\nServiciu: " << f.getFactura( ) << " Nr.lucrari: " << f.getNrLucrari( ) << " Preturi: ";
      for (const auto h : f.getPreturi( )) cout << h << "    ";
   }
cout << "\n---------------------------------------";
```

```
double  mediu = 0;
    for (auto f : v) mediu = mediu + f.pret(f.getPreturi( ));
    mediu = mediu / dimP;
    cout << "\nPret  mediu: " << mediu;
    cout << "\n---------------------------------------";
  vector<Factura>  newvec;
    for (int j = 1; j <= dimP; j++) {//sortare  desc.  dupa  pret
        Factura  mare = v[0];
        int  i = 0;
        int  el = 0;
        for (auto f : v) {
            if (f.pret(f.getPreturi( )) > mare.pret(mare.getPreturi( ))) {
                mare = f;
                el = i;    }
            i++;
        }
        newvec.push_back(mare);
        v.erase(v.begin( ) + el);
    }
    cout << "\nAfisare sortare descrescatoare dupa pret: ";
    for (auto x : newvec)
        cout << "\nServiciu: " << x.getFactura( ) << " Pret factura: " << x.pret(x.getPreturi( ));
    cout << "\n-----------------------------------------------";

    Factura  newobj{ "Montaj",dimL,p2,20 };
    Factura  cpyobj = newobj;//apel constructor de copiere
    cout << "\nObiectul  creat  prin  copiere din Montaj este, Serviciu: " << cpyobj.getFactura( ) << "
NrLucrari: " << cpyobj.getNrLucrari( ) << " Preturi: ";
    for (const auto h : newobj.getPreturi( )) cout << h << "    ";
    cout << "\nPretul cu TVA pentru noua factura = " << cpyobj.TVA(cpyobj, TVAF);
}
```

deque containers

- The term ***deque*** (pronounced "deck") is an abbreviation for the "two-end queue". It is a dynamic array that is implemented so that it can grow in both directions.
- Putting the elements at the end and at the beginning is fast. However, inserting the middle elements takes time for the elements to move. The *deque* structure can be described as follows:



They are *similar to vectors*, but are ***more efficient*** *in case of insertion and deletion of elements*. Unlike vectors, **contiguous storage** allocation *may not be guaranteed*.

The following example declares a *deque* of floating point values, inserts elements from 1.2 to 12 at the front of the container, and prints all *deque* elements:

```
#include <iostream>
#include <deque>
using namespace std;
const int dim = 10;

int main( ) {
        //deque container for double-point elements declaration
        deque<double> elem, elem1;
        //insert the elements each at the front
        cout << "push_front( )\n";
        int i;
        for (i = 1; i <= dim; ++i)
                elem.push_front(i * (1.2));
        //print all elements separated by a space
        for (unsigned i = 0; i < elem.size(); ++i)
                cout << elem[i] << ' ';
        cout << endl;
        //insert the elements each at the back
        cout << "\npush_back( )\n";
        for (i = 1; i <= dim; ++i)
                elem1.push_back(i * (1.2));
        //print all elements separated by a space
        for (unsigned i = 0; i < elem1.size(); ++i)
                cout << elem1[i] << ' ';
        cout << endl;
        return 0;
}
```

deque operations with iterators

```
//deque, constructors
#include <deque>
#include <iostream>
using namespace std;
const int dim =10;

int main( ){
deque <int>::iterator deq0Iter, deq1Iter, deq2Iter, deq3Iter, deq4Iter, deq5Iter, deq6Iter;
//Create an empty deque deq0
deque <int> deq0;
//Create a deque deq1 with dim elements of default value 0
deque <int> deq1(dim);
//Create a deque deq2 with 7 elements of value 10
deque <int> deq2(7, 10);
//Create a deque deq3 with 4 elements of value 2 and with the
//allocator of deque deq2
deque <int> deq3(4, 2, deq2.get_allocator( ));
//Create a copy, deque deq4, of deque deq2
deque <int> deq4(deq2);
//deque deq5 a copy of the deq4(_First, _Last) range
deq4Iter = deq4.begin( );
deq4Iter++;
deq4Iter++;
deq4Iter++;
deque <int> deq5(deq4.begin( ), deq4Iter);
```

```
//Create a deque deq6 by copying the range deq4 (_First, _Last) and the allocator of deque deq2
deq4Iter = deq4.begin( );
deq4Iter++;
deq4Iter++;
deq4Iter++;
deque <int> deq6(deq4.begin( ), deq4Iter, deq2.get_allocator( ));
cout<<"Operation: deque <int> deq0\n";
cout<<"deq0 data: ";
for(deq0Iter = deq0.begin( ); deq0Iter != deq0.end( ); deq0Iter++)
cout<<*deq0Iter<<" ";
cout<<endl;
cout<<"\nOperation: deque <int> deq1(dim)\n";
cout<<"deq1 data: ";
for(deq1Iter = deq1.begin( ); deq1Iter != deq1.end( ); deq1Iter++)
cout<<*deq1Iter<<" ";
cout<<endl;
cout<<"\nOperation: deque <int> deq2(7, 3)\n";
cout<<"deq2 data: ";
for(deq2Iter = deq2.begin( ); deq2Iter != deq2.end( ); deq2Iter++)
cout<<*deq2Iter<<" ";
cout<<endl;
```

```
cout<<"\nOperation: deque <int> deq3(4, 2, deq2.get_allocator( ))\n";
cout<<"deq3 data: ";
for(deq3Iter = deq3.begin( ); deq3Iter != deq3.end( ); deq3Iter++)
cout<<*deq3Iter<<" ";
cout<<endl;
cout<<"\nOperation: deque <int> deq4(deq2);\n";
cout<<"deq4 data: ";
for(deq4Iter = deq4.begin( ); deq4Iter != deq4.end( ); deq4Iter++)
cout<<*deq4Iter<<" ";
cout<<endl;
cout<<"\nOperation1: deq4Iter++...\n";
cout<<"Operation2: deque <int> deq5(deq4.begin( ), deq4Iter)\n";
cout<<"deq5 data: ";
for(deq5Iter = deq5.begin( ); deq5Iter != deq5.end( ); deq5Iter++)
cout << *deq5Iter<<" ";
cout << endl;
cout<<"\nOperation1: deq4Iter = deq4.begin( ) and deq4Iter++...\n";
cout<<"Operation2: deque <int> deq6(deq4.begin( ), \n"
" deq4Iter, deq2.get_allocator( ))\n";
cout<<"deq6 data: ";
for(deq6Iter = deq6.begin( ); deq6Iter != deq6.end( ); deq6Iter++)
cout<<*deq6Iter<<" ";
cout<<endl;
return 0;
}
```

- The other containers have similar properties that can be analyzed separately for each individual.
- The ***stack*** and ***queue*** containers use the ***deque <T>*** container as the default adapter.

template <typename T, typename Container = deque<T> > class stack;

template <typename T, typename Container = deque<T> > class queue;

Sorted queue, *prority_queue* is so defined:

template <typename T, typename Container = vector<T>, typename Compare = less<typename Container::value_type> > class priority_queue;

For **associative sorted containers**, the elements in the containers are referenced by the ***key*** and *not by their absolute position* in the container. Here we have:

1) set, are containers that store single items in a certain order:

```
template < typename T,                  // set::key_type/value_type
typename Compare = less<T>,     // set::key_compare/value_compare
typename Alloc = allocator<T> > // set::allocator_type
  class set;
```

Compare is the ordering criterion in the *set* with the default argument, *less <T>*.

- ***2) multiset***, allows duplicate keys with the same header.
- ***3) map***, are associative containers that store the elements formed by a combination of a ***key*** value and a ***mapped*** value, following a specific order:

```
template < typename Key,                    // map::key_type
           typename T,                      // map::mapped_type
           typename Compare = less<Key>,    // map::key_compare
         typename Alloc = allocator<pair<const Key,T> >
//map::allocator_type
           class map;
```

In a *map*, ***key values*** are generally used **to sort** and uniquely identify the items, while ***mapped values*** store the associated content to that key.

Key value types and ***mapped*** value may differ, and are grouped into the *value_type* member, which is a pair type combining both by:

typedef pair<const Key, T> value_type;

4) multimap is defined with the same header and allows ***keys*** with duplicate values.

Iterators

- ***Iterators*** act as intermediates between *algorithms* and *containers*, providing access to objects stored in a container without knowing the type of the elements.
- They are generalizers of the pointers and allow the unitary treatment of different types of data.
- They can be used to cross collections of data stored in containers.
- The iterators of a particular collection (container) are defined in the class associated with the collection, in the form of *typedef* constructions:

std::vector<std::string>::iterator it;

std::vector<std::string>::const_iterator cit;

- Containers have methods that return iterators:
- - *begin( )*: returns an iterator to the first element
- - *end( )*: returns an iterator after the last element; this iterator can be used as a sentinel when marking the collection and is also called ***past the end***
- Two iterators are considered equal if they indicate the same element or indicate the value ***next to the last element*** (*past the end).* The compiler does not check the iterator domains, i.e. if two iterators indicate on the same container or not.

- The *iterator* has the *scope operator* that precedes it:

vector <int> :: itereator p;

For:

vector <int> vi;

the first occurrence of a value is determined by:

p= find (vi.begin( ), vi.end( ), 7) ;

if (p!= vi.end( ){cout<<"Found val. 7 in vi";}

else {cout <<"Not Found val. 7 in vi ";}

Using an *iterator* is done as follows:

*Container_name ::**iterator** first, last;//declaration*

first = Container_name.begin( );//assign

last = Container_name.end( );

or:

*Container_name ::**iterator** first= Container_name.begin( );//init*

*Container_name ::**iterator** last= Container_name.end( );*

Types of Iterators:

https://www.cplusplus.com/reference/iterator/
https://www.geeksforgeeks.org/iterators-c-stl/

- Input, *InputIterator*: Reads an item at a time in the forward direction.
- Output, *OutputIterator*: Write an item at a time in the forward direction (ostream).
- Forward, *ForwardIterator*: Reads or writes an item at a time forward (forward_list).
- Bidirectional, *BidirectionalIterator*: Read or write, forward or backward (*list, map, set*).
- Random access, *RandomAccessIterator*: Same as bidirectional, plus jumps at any distance within the collection (*vector*).
- Besides these categories, there are also *adapters iterators* that only make backward (reverse) traverses, *insert iterators*, *iterators* that are *read-only*, etc.

Iterators operators

- We consider that *i, j* are iterators and *n* is an integer.
- Operators sharing all types of iterators:

++*i* Advances a position and returns the new *i* value

i++ Advances a position and returns the old *i* value

- **Input Iterators**:

**i* Returns a read-only reference to the element in the position given by *i*

i == j Returns TRUE if the two iterators are positioned on the same element (or after the last item in the collection)

i! = j Returns TRUE if i and j are positioned on different elements

- **Output Iterators**:

**i* Returns a reference to the element in the position given by *i*

i = j Sets for the same position as for *j*

- **Bidirectional iterators:**

--*i* Withdraw a position and return the new value for *i*

i-- Withdraw a position and return the old value for *i*

- **Random access iterators:**

i + = *n* Advances n positions, returns the new value for *i*

i- = *n* Withdraw n positions, reintroduces the new value for *i*

i + *n* Returns an iterator positioned over n elements after *i*

i – *n* Returns an iterator positioned over *n* elements in front of *i*

i [n] Returns a reference to element *n* in the collection

Simple iterators example using list container:

```
//iterator simple example
#include <iostream>
#include <list>
using namespace std;

int main( ){
//lst, list container for character elements
list<char> lst;
//append elements from 'A' to 'Z'
//to the list lst container
for(char chs='A'; chs<='Z'; ++chs)
lst.push_back(chs);
//iterate over all elements and print,
//separated by space
list<char>::const_iterator pos;
for (auto &pos: lst)
cout << pos << ' ';
//for (pos = lst.begin( ); pos != lst.end( ); ++pos)
//cout << *pos << '  ';
cout<<endl;
return 0;
}
```

Simple iterators example using multiset container:

```
//iterator, multiset example
#include <iostream>
#include <set>
using namespace std;

int main( ){
//multiset container of int data type
multiset<int> tst;
//insert elements
tst.insert(12);
tst.insert(21);
tst.insert(32);
tst.insert(31);
tst.insert(9);
tst.insert(14);
tst.insert(21);
tst.insert(31);
tst.insert(7);
//iterate over all elements and print, separated by space
multiset<int>::const_iterator pos;
//preincrement and predecrement are fast than postincrement and postdecrement.
for(pos = tst.begin( ); pos != tst.end( ); ++pos)
cout<<*pos<<' ';
cout<<endl;
return 0;
}
```

Simple iterators example using multimap container:

```
//iterator, multimap simple example
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main( ){
//type of the collection
multimap<int, string> mmp;
//set container for int/string values ,insert some elements in arbitrary order, notice a value of key 1
mmp.insert(make_pair(5,"learn"));
mmp.insert(make_pair(2,"map"));
mmp.insert(make_pair(1,"Testing"));
mmp.insert(make_pair(7,"tagged"));
mmp.insert(make_pair(4,"strings"));
mmp.insert(make_pair(6,"iterator!"));
mmp.insert(make_pair(1,"the"));
mmp.insert(make_pair(3,"tagged"));
//iterate over all elements and print, element member second is the value
multimap<int, string>::iterator pos;
for(pos = mmp.begin( ); pos != mmp.end( ); ++pos)
cout<<pos->second<<' ';
cout<<endl;
return 0;
}
```

Iterators can be analyzed according to their types, allowed operators, etc.

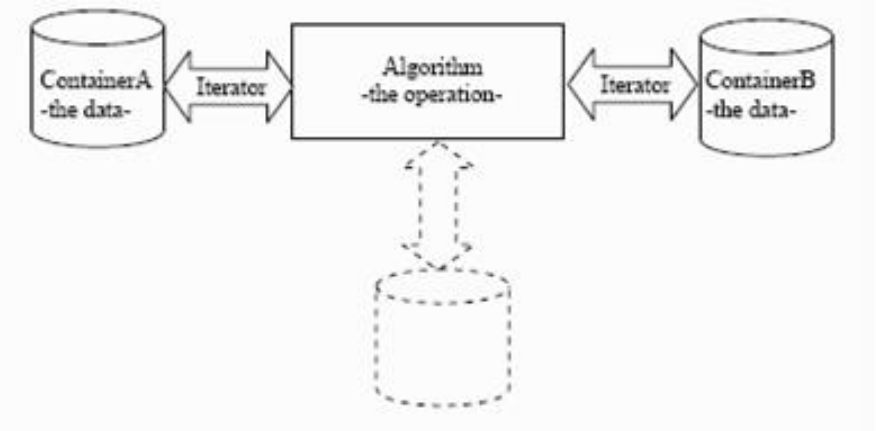
Algorithms

- STL algorithms are **generic member functions** that operate on containers.
- In order to be able to work with different types of containers, these functions have no containers as arguments but only iterators specifying the part or the containers in their entirety.
- Thus, algorithms **can also work on data types that are not containers**.
- This makes a decoupling between *algorithms* and *containers* via *iterators*.
- It is very important that those containers support the necessary iterators for an algorithm (see documentation).
- In this way, any container can be combined with any type of algorithm. All components work with arbitrary types, being a good example of the generic programming concept.

Example:

template <typename ForwardIterator> ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

- This algorithm requires a container that supports at least one "Forward" iterator. The attempt to use an algorithm on a container that does not provide the necessary iterators leads to errors, sometimes strange.
- The algorithms are in the <*algorithm*> library, more than 60 algorithms have been integrated, and now there are approximately 80 algorithms.



Types of Algorithms

- The main types of STL algorithms are:
- Algorithm
  - Sorting
  - Searching
  - Important STL Algorithms
  - Useful Array algorithms
  - Partition Operations
- Numeric
  - valarray class

Simple algorithm:

```
//Algorithm, simple example
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

//predicate, which returns whether an integer is a prime number
bool isPrimeNum(int number);//function used as function_object in find_if

int main( ){
list<int> lst1;
//insert elements from 10 to 20
for(int i=10; i<=20; ++i)
lst1.push_back(i);
//search for prime number
list<int>::iterator pos;
cout<<"The list lst1 data:\n";
for(pos=lst1.begin( ); pos!=lst1.end( ); pos++)
cout << *pos << " ";
cout<<endl<<endl;
```

```
pos = find_if(lst1.begin( ), lst1.end( ),isPrimeNum); //range predicate
if(pos != lst1.end( ))//found
cout<<*pos<<" is the first prime number found"<<endl;
else //not found
cout<<"no prime number found"<<endl;
return 0;
}//main

bool isPrimeNum(int number){
//ignore negative sign
number = abs(number);
//0 and 1 are prime numbers
if(number == 0 || number == 1)return true;
//find divisor that divides without a remainder
int divisor;
for(divisor = (number/2); (number%divisor) != 0; --divisor){ }
//if no divisor greater than 1 is found, it is a prime number
return (divisor == 1);
}
```

Types of algorithms

- **Sequence operations that do not cause changes:**

-Apply a function to all elements (*for_each ...)*

-Look for an element that satisfies a condition (*find ...)*

-Counts the elements that satisfy a condition (*count ...)*

-Look for the first mismatch between two sequences (*mismatch ...)*

-Check whether two sequences are equal (*equal ...)*

-Look for the first matching of a subsequence in another sequence (*search ...)*

- **Generalized numerical operations**

-Sum all elements of a sequence (*accumulate)*

-Calculates the scalar product of two sequences (*inner_product)*

-Calculates partial amounts (*partial_sum)*

-Calculates a new sequence starting from partial sums in another sequence (*adjacent_difference)*

Sequence operations that cause changes:

-Copy a sequence to another sequence (*copy ...)*

-Interchange values or ranges (*swap ...)*

-Transforms a sequence or two sequences into a new sequence (*transform ...)*

-Replace the specified elements (*replace ...)* and (*replace_if ( ))* replace the elements that satisfy a predicate

-Populate a domain with a value (*fill, fill_n)*

-Populate a field with generated values (*generated, generated_n),* replaces all, or *n* elements with those generated

-Delete items (*remove ...)*

-Delete duplicates (*unique, unique_copy)*

-Inverse a sequence (*reverse, reverse_copy)*

-Rotation of elements (*rotated, rotate_copy*)

-Randomly blend items (*random_suffle)*

-Partitioning with elements that satisfy a predicate (*partition, stable_partition)*

Sorting and associated operations:

-Domain sorting (*sort ...)*

-Placing the element *n* in the final position that would result from sorting (*nth_element)*

-Look for the limits, or position of a value in a sorted sequence (*lower_bound, upper_bound, equal_range, binary_search)*

-Interlace two sorted sequences (*merge, inplace_merge)*

-Set operations on sorted sequences (*includes, set_union, set_intersection, set_difference, set_symmetric_difference)*

-Heap operations (*push_heap, pop_heap, make_heap, sort_heap)*

-Look for the minimum or maximum element (*min, max, min_element, max_element)* of two or one sequence

-Determine lexicographic order of two sequences (*lexicographical_compare)*

-Generate permutations for a sequence (*next_premutation, prev_permutation)* in lexicographical order

Sorting algorithm:

```
//algorithm, sort( )
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

//Return whether first element is greater than the second
bool userdefgreater(int elem1, int elem2);//used as function object in sort
const int dim =15;

int main( ){
vector <int> vec1; //container
vector <int>::iterator Iter1; //iterator
int k;
for(k = 0; k <= dim; k++)
vec1.push_back(k);
random_shuffle(vec1.begin( ), vec1.end( ));
```

```
cout<<"Original random shuffle vector vec1 data:\n";
for(Iter1 = vec1.begin( ); Iter1 != vec1.end( ); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
sort(vec1.begin( ), vec1.end( ));
cout<<"\nSorted vector vec1 data:\n";
for(Iter1 = vec1.begin( ); Iter1 != vec1.end( ); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
//To sort in descending order, specify binary predicate
sort(vec1.begin( ), vec1.end( ), greater<int>( ));
cout<<"\nRe sorted (greater) vector vec1 data:\n";
for(Iter1 = vec1.begin( ); Iter1 != vec1.end( ); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
//A user-defined binary predicate can also be used
sort(vec1.begin( ), vec1.end( ), userdefgreater);
cout<<"\nUser defined re sorted vector vec1 data:\n";
for(Iter1 = vec1.begin( ); Iter1 != vec1.end( ); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
return 0;
}//main

bool userdefgreater(int elem1, int elem2)
{return elem1 > elem2;}
```

```
//Sorting Example -STL Objects.
//Person.h

class Person {
        // Left out making a constructor for simplicity's sake.
        string name;
        int age;
        string favoriteColor;
public:
        const string getName( ) { return name; }
        int getAge( ) { return age; }
        string getFavoriteColor( ) { return favoriteColor; }

        static void introdu(vector <Person> &p, int n) {
                for (vector<Person>::size_type i = 0; i != n; ++i){
                cout << "Person #" << i + 1 << " name: ";
                cin >> p[i].name;
                cout << "Person #" << i + 1 << " age: ";
                cin >> p[i].age;
                cout << "Person #" << i + 1 << " favorite color: ";
                cin >> p[i].favoriteColor; }
            }
        };
// Sort Container by name compare function
        bool sortByName(Person &lhs, Person &rhs) { return lhs.getName( ) < rhs.getName( ); }
// Sort Container by age compare function
        bool sortByAge(Person &lhs, Person &rhs) { return lhs.getAge( ) < rhs.getAge( ); }
// Sort Container by favorite color compare function
        bool sortByColor(Person &lhs, Person &rhs) { if(lhs.getFavoriteColor( ) < rhs.getFavoriteColor( ))
                                                        return true;
                                                        else return false; }
```

```
//main
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
#include "Person.h"

int main( ) {		int n;
			cout << "\nEnter no. of peoples: " << endl;
			cin >> n;
			// Make a vector that holds n blank Person Objects
			vector<Person> people(n);
			cout << "\nEnter values\n";
			Person::introdu(people, n);
		cout << "\nSort by name\n";
			sort(people.begin( ), people.end( ), sortByName);
			for (Person &n : people)
			cout << n.getName( ) << " ";
		cout << "\nSort by age\n";
			 sort(people.begin( ), people.end( ), sortByAge);
			for (Person &n : people)
			cout << n.getAge( ) << " ";
		cout << "\nSort by color\n";
			sort(people.begin( ), people.end( ), sortByColor);
			for (Person &n : people)cout << n.getFavoriteColor( ) << " ";
			return 0;
		}
```

- Functional behavior is something that can be called by using brackets and arguments, e. g.:

function(arg1, arg2); //a function call

- So if we want the objects to behave in this way, we must make it possible to call them, using the brackets and the arguments.
- So all you need to do is **define the *operator( )*** with the appropriate parameter, for example:

class XYZ {

public:

//define "function call" operator

return-value operator( ) (arguments) const*;*

...

};

Now you can use objects in this class to behave as a function and can call:

XYZ foo;

...

//call operator( ) for function object foo

foo(arg1, arg2); *//object called as a function*

Calling equivalent with:

//call operator( ) for function object foo

foo.operator( ) (arg1, arg2);

Base function object example:

```
//function object example
//PrintSomething.h
//simple function object that prints the passed argument
class PrintSomething
{
public:
void operator( ) (int elem) const  {
cout << elem << " ";
                }
};//class

//main
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
const int dim =10;
#include "PrintSomething.h"

int main( ){
vector<int> vec;
//insert elements from 1 to dim
for(int i=1; i<=dim; ++i)
vec.push_back(i);
//print all elements
for_each (vec.begin( ), vec.end( ), PrintSomething( )); //range operation
cout<<endl;
}
```

- The *PrintSomething* class defines objects for which the operator can call *( )* with an *int* argument.
- Expression:

PrintSomething( )

- In construction:

for_each(vec.begin( ), vec.end( ), PrintSomething( ));

- Create a temporary object of this class, which is passed to the *for_each( )* algorithm as an argument. The *for_each( )* algorithm is written as follows:

```
namespace std{
template <typename Iterator, typename Operation>
Operation for_each(Iterator act, Iterator end, Operation op)
{
                    while(act != end)
          { //as long as not reached the end
          op(*act); //call op( ) for actual element
          act++; //move iterator to the next element
          }
return op;
}
```

- The main uses of **Functions objects** are related to ***data generation***, ***data testing*** **(predicates)** and ***operations***. They are faster than normal functions.
- There are also **predefined function objects** such as sorting, or numerical processing.
- **Functions objects** are *generalizations of the concept of function*;
- They *take the place of pointers to functions* from traditional C/C++ programming;
- They are frequently used as the *generic parameter* of an algorithm to indicate an operation that is executed for certain elements in the data structure.
- **Generators**
- There are algorithms that go through a domain by calling a function object at each step and assigning the result to the current element. In this case we have a generator.
- **Predicates**
- They are used to test certain conditions. The *( )* bracket must be overloaded to return something that can be tested.
- Algorithms that have the **suffix** ***_if*** use a function object to test each element for a condition.
- A *simple predicate* dereferences a single element for tests and a *binary predicate* (BinaryPredicate) deferens two elements to compare them.
- Finding an element that satisfies a predicate can be defined as follows:

```
vector <int> vi;
vector <int> :: iterator p= find_if (vi.begin( ), vi.end( ), Less_than <int> (7));
if (p!= vi.end( ){cout<<"Am gasit val. < 7 in vi";}
        else {cout <<"Nu am gasit val. < 7 in vi";}
```