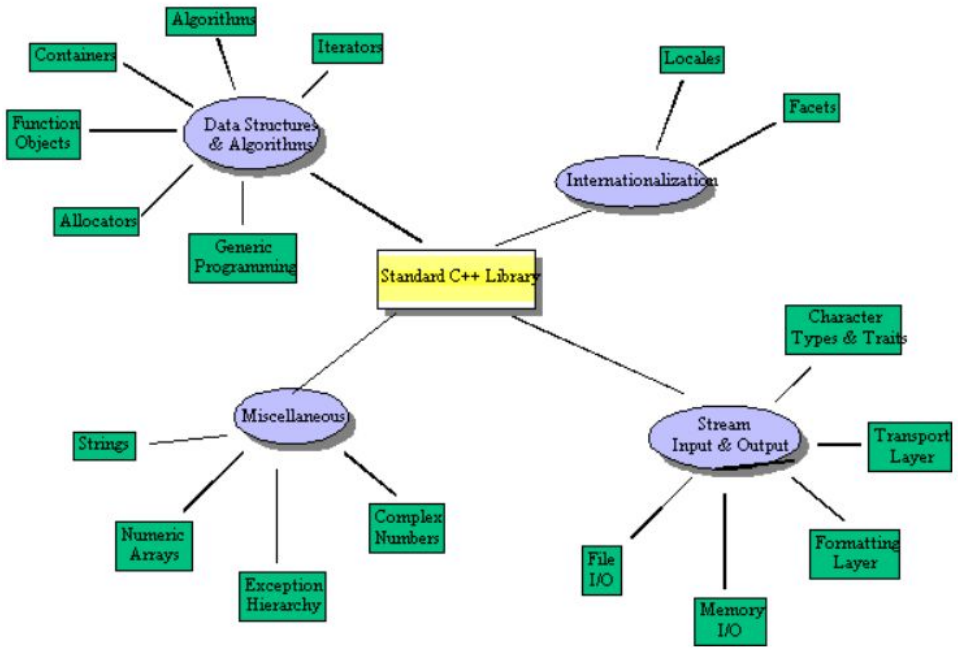
C++ Standard Library & STL

Lecture 14

"In our lives, change is unavoidable, loss is unavoidable. In the adaptability and ease with which we experience change, lies our happiness and freedom."

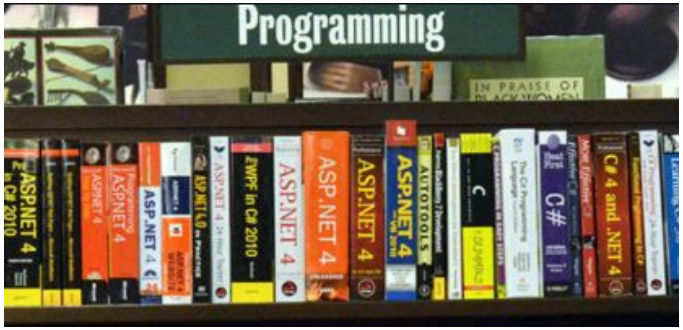
Overview

- Library
- STL
  - Containers
  - Iterators
  - Algorithms
  - Function Objects
- C++ Standard Library
- Guidelines



Library

- A **library** is a collection of implementations of behavior.
  - Having a well-defined interface.
- To use certain functionality, people can use a library instead of implementing those over and over again.
- Able to reuse in multiple independent programs.




```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
using namespace std;
```

Library

- Static library: code of the library is accessed during the build of the invoking program
- Dynamic library: library is connected after the program executed (loaded at runtime).
- Most compiled languages have a standard library although programmers can also create their own custom libraries.
- Using only the bare language, every task is tedious (in any language). Using a suitable library any task can be reasonably simple..

Directive #include

- For #include <filename> the preprocessor normally searches from directories pre-designated by the compiler/IDE, including standard library header files.
- For #include "filename" the preprocessor searches first in the same directory as the file, and then follows the search path from there.
  - Normally used to include programmer-defined header files.

```
#include <iostream>
#include <iomanip>
```

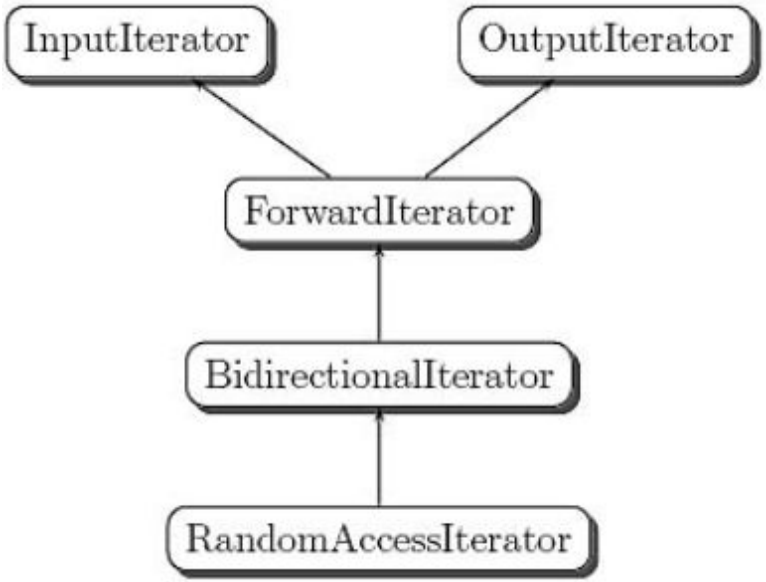
```
#include "app.h"
#include "network.h"
```

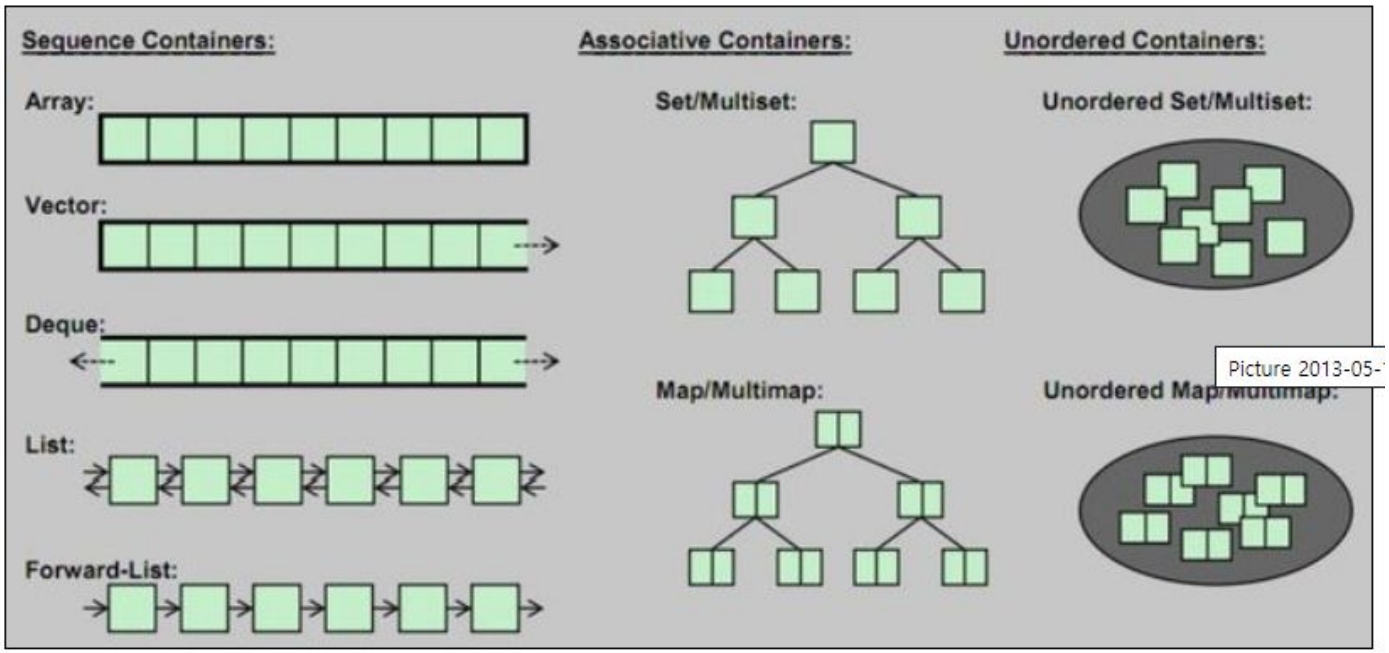
The C++ Standard Template Library (STL)

- A library consisting of set of C++ template classes to provide data structures and functions such as lists, stacks, arrays, etc.
  - Can be used with any type supporting some elementary operations (such as copying and assignment).

Components of STL

- Containers
- Iterators
- Algorithms
- Functions





Containers

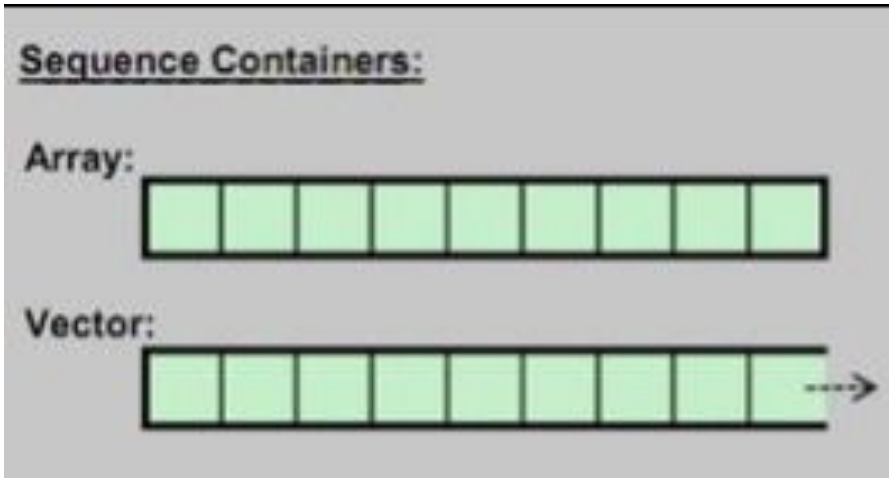
- Objects that store a collection of data.
  - Allow programmers to easily implement common data structures.
  - Sequence containers
    - array, vector, deque, and list.
  - Associative containers
    - set, multiset, map, multimap, unordered_set, unordered_map, unordered_multiset and unordered_multimap.
  - Container adaptors
    - queue, priority_queue, and stack.

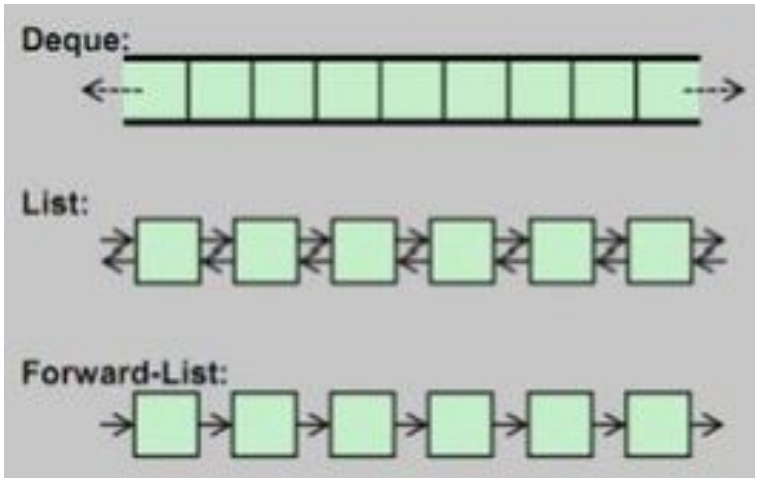
Containers

- Simplify development and reduce the burden of custom re-implementation of a data structure.
  - Implemented correctly, so no need to spend time debugging custom ones.
  - Fast, and likely more efficient than custom ones.
  - Share common interfaces, so simple to utilize different containers.
  - Well-documented and easily understood by other developers.

Sequence Containers

- Sequence containers implement data structures which can be accessed sequentially.
  - array : static contiguous array
  - vector : dynamic contiguous array
  - deque : double-ended queue
  - list : doubly-linked list

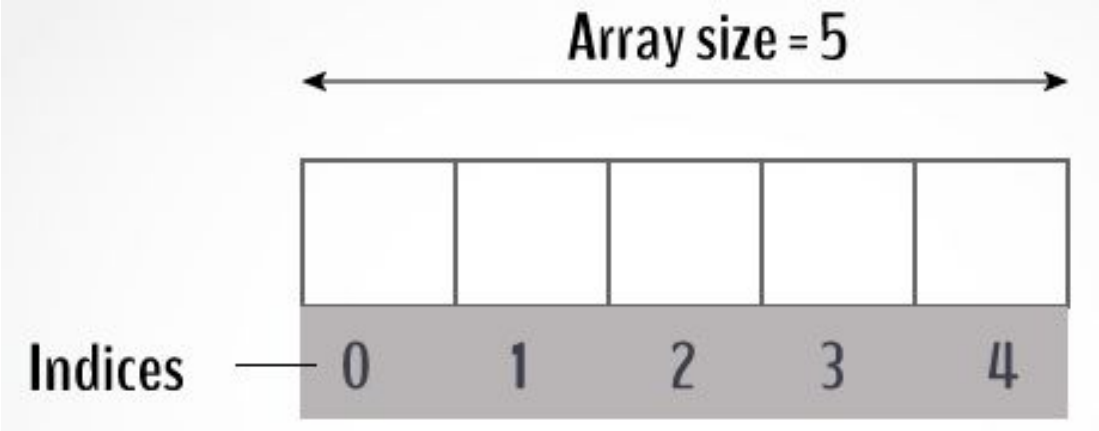




Sequence Containers

- array : static contiguous array
  - Just a container version of the array we used to know

```
array <int, 128> intarray;
// Element type and the size of array specified at definition
```



Sequence Containers

- array : static contiguous array

```
// Using C array
#include <iostream>
using namespace std;
int main() {
 int myarray[3] = {10,20,30};
 for (int i=0; i<3; ++i)
   ++myarray[i];
 for (int elem : myarray)
   cout << elem << ' ';
}
```

```
// Using array container
#include <iostream>
#include <array>
using namespace std;
int main() {
 array<int,3> myarray {10,20,30};
 for (int i=0; i<myarray.size(); ++i)
   ++myarray[i];
 for (int elem : myarray)
   cout << elem << ' ';
}
```

```
11 21 31
```

Sequence Containers

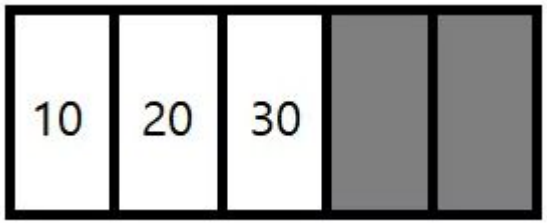
- Can get first and last element with front(), back()
- Can iterate through array using begin(), end()

```
#include <iostream>
#include <array>
using namespace std;
int main() {
array<int,3> myarray {10,20,30};
cout << myarray.front() << myarray.back() <<'\n'
for (auto iter = myarray.begin();
          iter != myarray.end(); ++iter)
  ++myarray[i];
for (int elem : myarray)
  cout << elem << ' ';
}
```

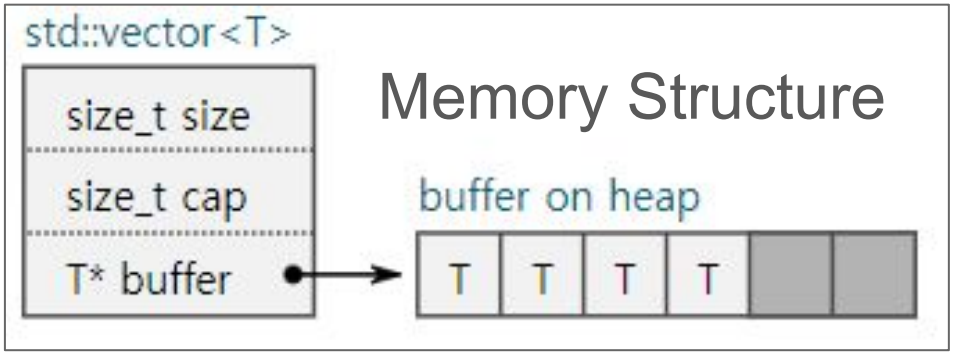
```
10 30
11 21 31
```

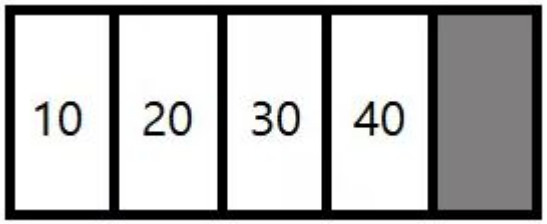
Sequence Containers

- vector : dynamic contiguous array

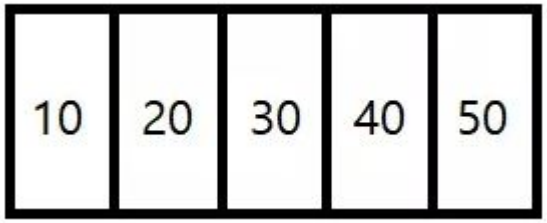


vec.push_back(40);

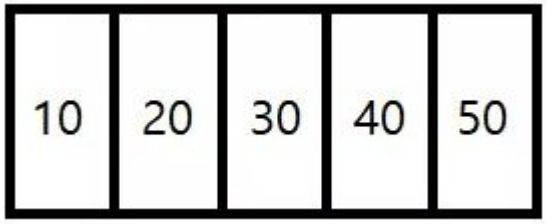


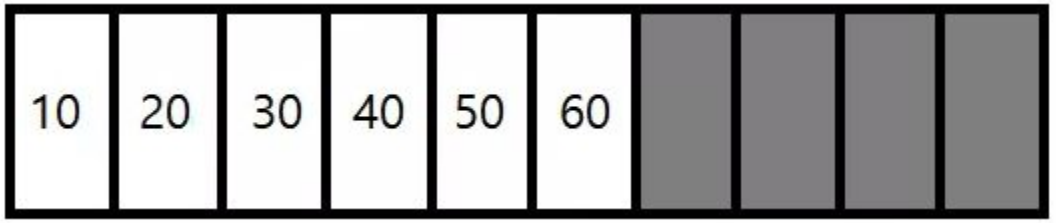


vec.push_back(50);



vec.push_back(60);





Sequence Containers

- vector : dynamic contiguous array

```
int main() {
 std::vector<int> vec;
 vec.push_back(10);
 vec.push_back(20);
 vec.push_back(30);
 vec.push_back(40);
 print_vector(vec);
 vec.insert(vec.begin() + 2, 15);
 print_vector(vec);
 vec.erase(vec.begin() + 3);
 print_vector(vec);
}
```

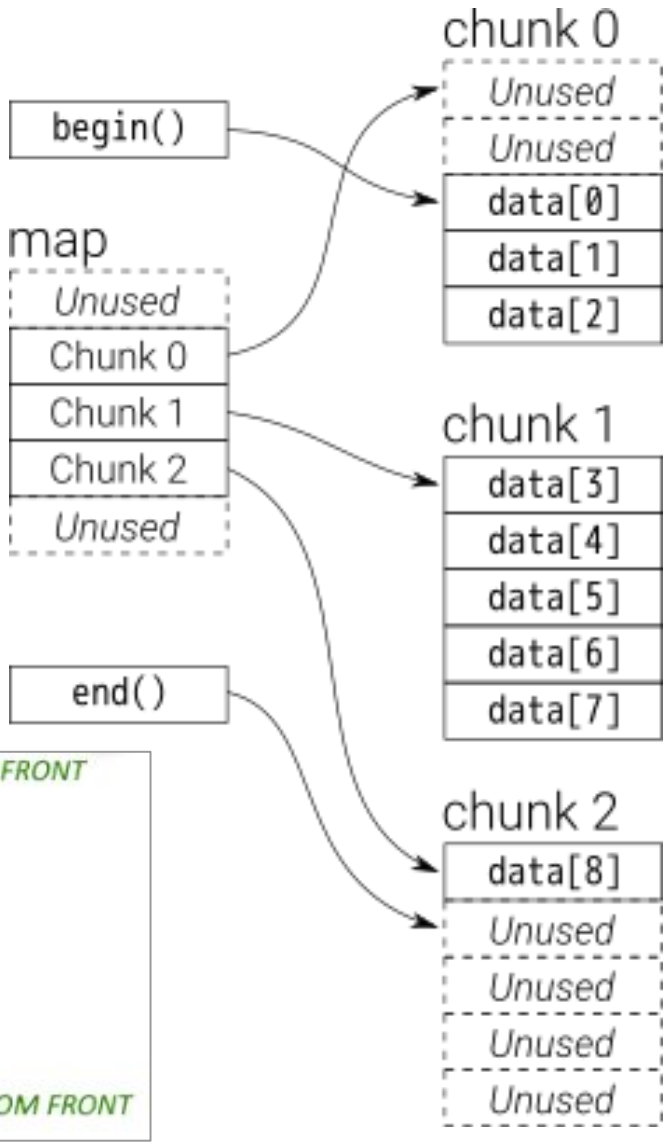
```
#include <vector>
#include <iostream>
using namespace std;
template <typename T>
void print_vector(vector<T>& vec) {
 for (auto itr = vec.begin();
      itr != vec.end(); ++itr) {
 cout << *itr << " ";
 } cout << "\n";
}
```

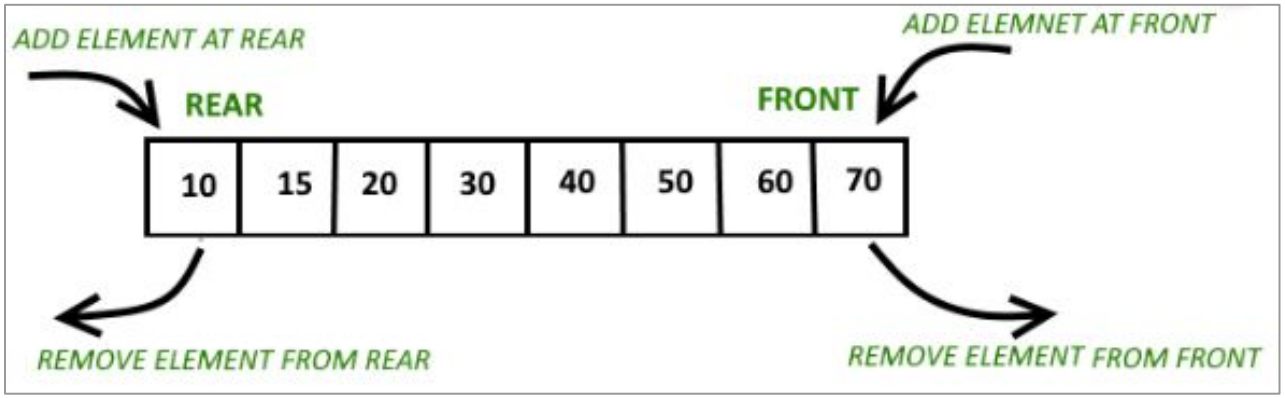
```
10 20 30 40
10 20 30 15 40
10 20 30 40
```

Sequence Containers

- deque : double-ended queue
  - Can add / delete element at the front and the rear of the container.

```
deque <int> mydeq;
// Element type specified in template
```





Sequence Containers

- deque : double-ended queue

```
int main() {
 deque <int> mydeq;
 mydeq.push_back(10); mydeq.push_front(20);
 mydeq.push_back(30); mydeq.push_front(15);
 cout << "deque : ";   showdq(mydeq);
 cout << "\nsize() : " << mydeq.size();
 cout << "\nmax_size() : " << mydeq.max_size();
 cout << "\nat(2) : " << mydeq.at(2);
 cout << "\npop_front() : ";  mydeq.pop_front();
 showdq(mydeq);
 cout << "\npop_back() : "; mydeq.pop_back();
 showdq(mydeq); return 0;
}
```

Sequence Containers

- deque : double-ended queue

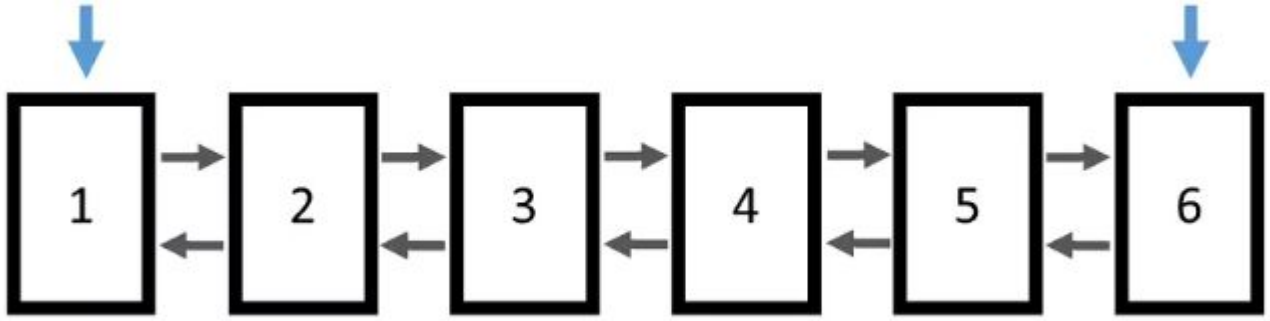
```
#include <iostream>
#include <deque>
 using namespace std;
 void showdq(deque <int> g) {
   for (auto it = g.begin(); it != g.end(); ++it)
       cout << ' ' << *it;
   cout << '\n';
}
```

```
deque : 15 20 10 30
size() : 4
max_size() : ???
at(2) : 10
pop_front() : 20 10 30
pop_back() : 20 10
```

Sequence Containers

- list : doubly-linked list

```
list <int> mylist;
// Element type specified in template
```



Sequence Containers

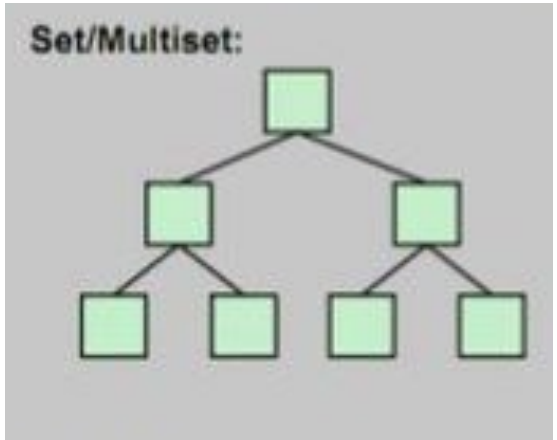
- list : doubly-linked list

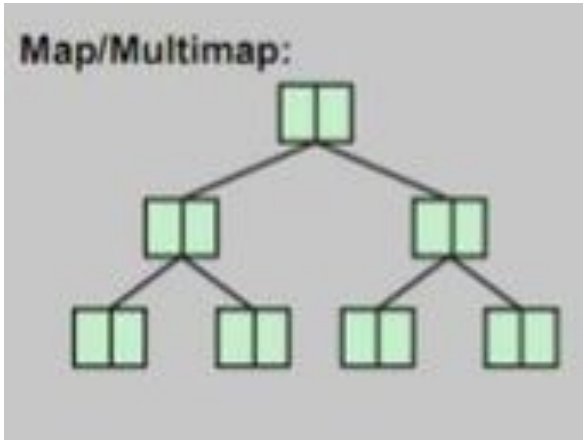
```
#include <iostream>
#include <list>
int main() {
 std::list<int> lst;
 lst.push_back(10); lst.push_back(20);
 lst.push_back(30); lst.push_back(40);
 for (auto itr = lst.begin(); itr != lst.end(); ++itr) {
   std::cout << *itr << std::endl;
 }
}
```

```
10
20
30
40
```

Associative Containers

- Associative containers implement sorted data structures that can be quickly searched.
  - set : collection of unique keys, sorted by keys
  - map : collection of key-value pairs, sorted by keys, keys are unique
  - multiset : collection of keys, sorted by keys
  - multimap : collection of key-value pairs, sorted by keys





Associative Containers

- set : collection of unique keys, sorted by keys
  - Element type should be specified with a template.
  - Comparison operator can be feeded as a second value of the template for the order of the elements.

```
set <string> sset; // Element type specified
set <int, greater <int> > myset1; // custom comparator
```

  - For user-defined types, the operator< should be defined on that type, or should provide the comparator.

```
set<myclass> myset2; // User-defined type with operator< overriden
set<myclass,mycomp> myset3;
// User-defined type and comparator functor
```

Associative Containers

- set : collection of unique keys, sorted by keys

```
int main(){
// empty set container
set <int, greater <int> > mset1;
// insert elements in random order
mset1.insert(40); mset1.insert(30);
mset1.insert(60); mset1.insert(20);
mset1.insert(50); mset1.insert(50);
// only one 50 will be added to the set
mset1.insert(10);
// printing set
printset(mset1)
```

```
// copying the elements
 set <int> mset2(mset1.begin(),
 mset1.end());
 printset(mset2)
 // remove all elements up to 30
 mset2.erase(mset2.begin(),
 mset2.find(30));
 printset(mset2)
 // remove element with value 50
 mset2.erase (50);
 printset(mset2)
 return 0;
}
```

Associative Containers

- set : collection of unique keys, sorted by keys

```
#include <iostream>
#include <set>
#include <functional> // greater
#include <iterator>
using namespace std;
void printset(set <int> g) {
  for (auto it = g.begin(); it != g.end(); ++it)
      cout << ' ' << *it;
  cout << '\n';
}
```

```
60 50 40 30 20 10
10 20 30 40 50 60
30 40 50 60
30 40 60
```

Associative Containers

- set : collection of unique keys, sorted by keys
  - For user-defined types, the operator< should be defined on that type, or should provide the comparator.

```
class Job {
 int priority; string job_desc;
public:
 Job(int priority, string job_desc)
   : priority(priority), job_desc(job_desc) {}
 bool operator<(const Job& t) const { // Ordering of Jobs
   if (priority == t.priority) {
     return job_desc < t.job_desc;
   }
   return priority > t.priority;
}
```

Associative Containers

- set : collection of unique keys, sorted by keys
  - For user-defined types, the operator< should be defined on that type, or should provide the comparator.

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
void print_set(set<Job>& s) {
for (const auto& elem : s) {
 cout << " [ " << td.priority << "] " <<
td.job_desc << ", " ;
 }
}
```

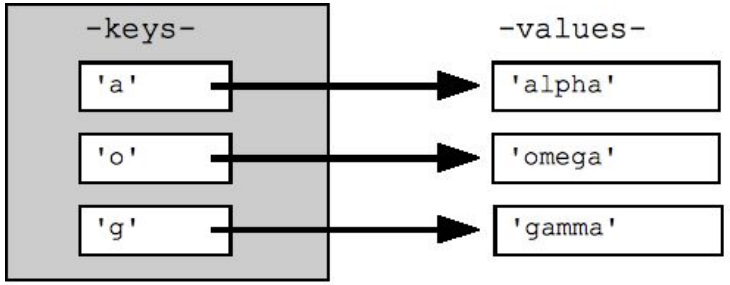
```
int main() {
 set<Job> jobs;
 jobs.insert(Job(1, "Exercise"));
 jobs.insert(Job(2, "Study"));
 jobs.insert(Job(4, "Programming"));
 jobs.insert(Job(1, "Chores"));
 print_jobs(jobs); return 0;
}
```

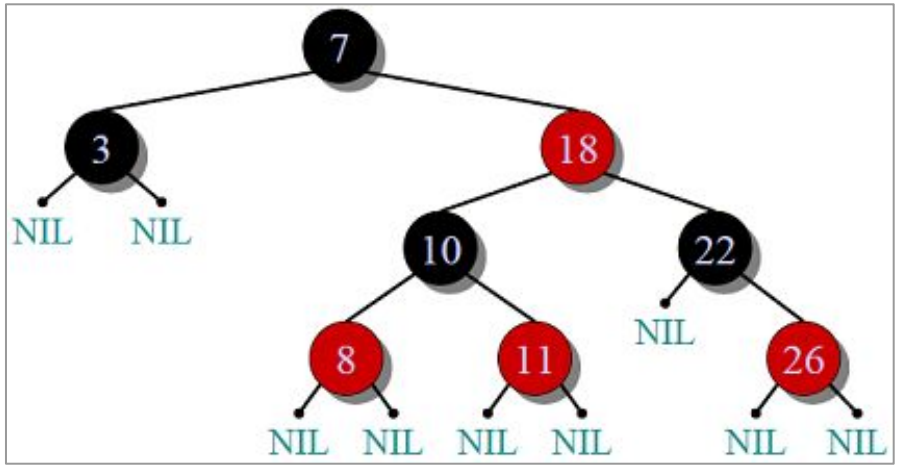
```
 [4] Programming, [2] Study, [1]
Chores, [1] Exercise
```

Associative Containers

- map : collection of key-value pairs, sorted by keys, keys are unique.
  - Implemented with Red-Black tree.

```
map <int, string> id_name;
// The type of key and value are specified specified in template
```





Associative Containers

- map : collection of key-value pairs, sorted by keys, keys are unique.

```
int main() {
 // empty map container
 map<int, int> dict1;
 // insert elements in random order
 dict1.insert(pair<int, int>(1, 40));
 dict1.insert(pair<int, int>(2, 30));
 dict1.insert(pair<int, int>(3, 60));
 dict1.insert(pair<int, int>(4, 20));
 dict1.insert(pair<int, int>(5, 50));
 // printing map dict1
 printmap(dict1)
```

```
// copying the elements
map<int, int> dict2(dict1.begin(),
dict1.end());
 // remove all elements up to key=3
 dict2.erase(dict2.begin(),
dict2.find(3));
cout << "\ndict2 after removal of";
printmap(dict2)
// remove all elements with key = 4
dict2.erase(4);
cout << "\ndict2.erase(4) : ";
printmap(dict2)
return 0; }
```

Associative Containers

- map : collection of key-value pairs, sorted by keys, keys are unique.

```
#include <iostream>
#include <iterator>
#include <map>
 using namespace std;
void printmap(map <int> g) {
cout << "\tKEY\tELEM\n";
for(auto itr=g.begin();itr!= g.end();++itr) {
 cout<<'\t'<<itr->first <<'\t'<<itr->second<<'\n';
}
cout << endl;
}
```

```
KEY ELEM
1       40
2       30
3       60
4       20
5       50
KEY ELEM
3       60
4       20
5       50
KEY ELEM
3       60
5       50
```

Associative Containers

- map : collection of key-value pairs.
  - For user-defined key types, the operator< should be defined on the key type.

```
class User {
// User Defined Type
string m_id; string m_name;
public:
User(string name, string id)
:m_id(id), m_name(name){}
const string& getId() const {
  return m_id;
}
```

```
const string& getName() const {
  return m_name;
}
bool operator< (const User&
userObj) const {
  if(userObj.m_id > this->m_id)
    // Ascending order
    return true;
}
};
```

Associative Containers

- map : collection of key-value pairs.
  - For user-defined key types, the operator< should be defined on the key type.

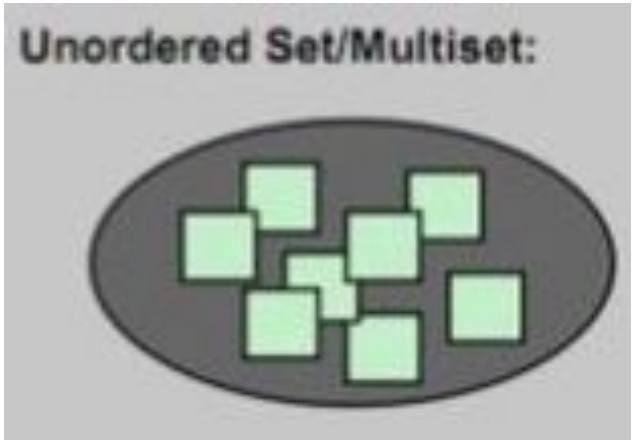
```
int main(){
map<User, int> userMoneyMap;
userMoneyMap.insert(
make_pair<User, int>(User("Henry",
"3"), 100) );
userMoneyMap.insert(
make_pair<User, int>(User("David",
"1"), 120) );
userMoneyMap.insert(
make_pair<User,
int>(User("Raynold", "2"), 300) );
```

```
for(auto it = userMoneyMap.begin();
it != m_UserInfoMap.end(); it++){
  cout<<it->first.getName()<<" |
"<<it->second<<endl;
}
}
```

```
David | 120
Raynold | 300
Henry | 100
```

Unordered Associative Containers

- Unordered associative containers implement unsorted data structures that can be quickly searched.
  - Unordered counterparts for each associative containers
    - unordered_set, unordered_map, unordered_multiset, unordered_multimap



Unordered Associative Containers

- To use user-defined types in the unordered associative containers, the operator== and the hash function should be defined.

```
class Job {
 int priority; string job_desc;
public:
 Job(int priority, string job_desc)
   : priority(priority), job_desc(job_desc) {}
 bool operator==(const Job& t) const {
// Ordering of Jobs
 return priority == t.priority && job_desc
== t.job_desc;
}
```

```
template <>
struct hash<Job> {
 size_t operator()(const Job& t)
const {
   std::hash<string> hash_func;
   return t.priority ^
(hash_func(t.job_desc));
 }
}
```

Unordered Associative Containers

- To use user-defined types in the unordered associative containers, the operator== and the hash function should be defined.

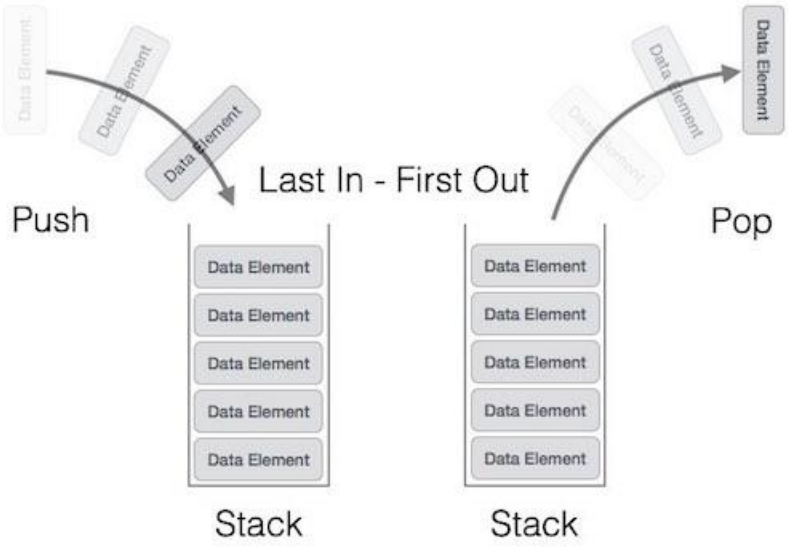
```
void print_jobs(unordered_set
<Job>& s) {
for (const auto& elem : s) {
 cout << " [ " << td.priority << "] " <<
td.job_desc << ", " ;
 }
} // Feed user-defined type and hash
as a template
int main() {
 unordered_set<Job,Hash> jobs;
```

```
 jobs.insert(Job(1, "Exercise"));
 jobs.insert(Job(2, "Study"));
 jobs.insert(Job(4, "Programming"));
 jobs.insert(Job(1, "Chores"));
 print_jobs(jobs); return 0;
}
```

```
 [1] Exercise, [2] Study, [1] Chores,
[4] Programming
```

Container Adaptors

- Container adaptors provide a different interface for sequential containers.
  - stack : Last-In-First-Out data structure
  - queue : First-In-First-Out data structure
  - priority_queue



Containers Guideline

- Prefer using STL array or vector instead of a C array.
  - C arrays are less safe (misused pointers) , and have no advantages over array and vector.
  - For a fixed-length array, use std::array.
  - For a variable-length array, use std::vector, which additionally can change its size and handles memory allocation.

```
int v[SIZE];              // BAD
std::array<int, SIZE> w;  // ok
```

```
int* v = new int[initial_size];
// BAD, owning raw pointer
delete[] v;
// BAD, manual delete
std::vector<int> w(initial_size); // ok
```

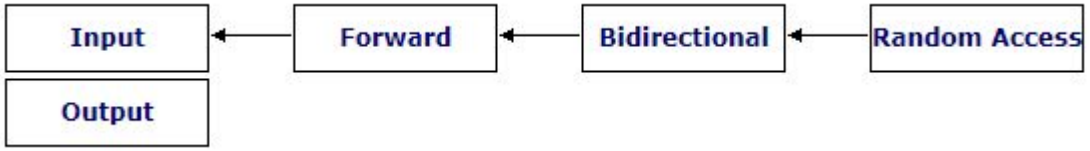
Containers Guideline

- Prefer using STL vector by default unless you have a reason to use a different container.
  - Fastest access (including sequential / random access).
  - Fastest default access pattern (begin-to-end or end-to-begin is prefetcher-friendly).
  - Lowest space overhead (contiguous memory layout has zero per-element overhead).
  - Easy to add and remove elements from the container., so use vector by default; if you don’t need to modify the container’s size, use array.

Iterators

- An iterator is a pointer that points to some element in a range of elements (e.g. Container).
  - Ability to iterate through the elements of that range using a set of operators.

Iterators



- The STL implements five different types of iterators.
  - Input iterators : can only read a sequence of values
  - Output iterators : can only write a sequence of values
  - Forward iterators : can be read, written to, and move forward
  - Bidirectional iterators : like forward iterators, but can also move backwards
    - map, set, List, multimap, multiset
  - Random access iterators : can move freely any number of steps in one operation
    - vector, deque

Iterators

- begin() : iterator pointing the beginning position of the container.
- end() : iterator pointing the position *after* the end of the container.

```
#include<iostream>
#include<iterator>
#include<vector>
using namespace std;
int main() {
vector<int> ar = { 1, 2, 3, 4, 5 };
// Declaring an iterator of a vector
vector<int>::iterator ptr;
// Displaying vector elements using
begin() and end()
for (ptr = ar.begin(); ptr < ar.end(); ptr++)
 cout << *ptr << " ";
return 0;
}
```

```
1 2 3 4 5
```

Iterators

- Iterate through them using the increment operator (++)
- Get the value where the iterator is pointing by dereference (*).

```
#include<iostream>
#include<iterator>
#include<vector>
using namespace std;
int main() {
vector<int> ar = { 1, 2, 3, 4, 5 };
// Declaring an iterator of a vector
vector<int>::iterator ptr;
```

```
// Displaying vector elements using
begin() and end()
for (ptr = ar.begin(); ptr < ar.end(); ptr++)
 cout << *ptr << " ";
return 0;
}
```

```
1 2 3 4 5
```

Iterators

- next() : The iterator after advancing the positions mentioned in its arguments.
- prev() : The iterator after decrementing the positions mentioned in its arguments.

```
#include<iostream>
#include<iterator>
#include<vector>
using namespace std;
int main() {
vector<int> ar = { 1, 2, 3, 4, 5 };
vector<int>::iterator ptr = ar.begin();
vector<int>::iterator ftr = ar.end();
```

```
auto it = next(ptr, 3); // points to 4
auto it1 = prev(ftr, 3); // points to 3
// Value at iterator position
cout << *it << " " << *it1 << endl;;
return 0;
}
```

```
4 3
```

Iterators

- advance() : increment the iterator position itself, not creating the new iterator.

```
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main() {
   vector<int> ar = { 1, 2, 3, 4, 5 };
   // Declaring iterator to a vector
   vector<int>::iterator ptr = ar.begin();
   advance(ptr, 3); // points to 4
   cout << *ptr << " ";
   return 0;
}
```

```
4
```

Algorithms

- Common algorithms performing activities such as searching and sorting on ranges of elements are provided in the STL
  - Sorting
  - Searching
  - Partition Operations
  - Numeric
  - sequence operations
  - set operations ...

Sorting

- Sorts a container (with random access iterator).
  - We can also write our own “comparator” function and pass it as a third parameter.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool greater(int i,int j) { return (i>j); }
void printvec(vector<int> vec) {
for (auto it=vec.begin(); it!=vec.end(); ++it)
 cout << ' ' << *it;
cout << '\n';
}
```

```
int main () {
int ints[] =
{32,71,12,45,26,80,53,33};
vector<int> vec (ints, ints+8);
sort (vec.begin(), vec.begin()+4);
// descending from 4th element
sort (vec.begin()+4, vec.end(),
greater);
printvec(myvector); return 0;
}
```

```
12 32 45 71 80 53 33 26
```

Min, Max

- max : returns the greater of the given values
- min : returns the smaller of the given values

```
#include <iostream>
#include <algorithm>
using namespace std;
int main () {
cout << min(1,2) << ' ' <<  min(2,1)
<< ' '<< min('a','z') << ' ' <<
min(3.14,2.72)
<< '\n';
return 0;
}
```

```
2 2 z 3.14
```

```
#include <iostream>
#include <algorithm>
using namespace std;
int main () {
cout << max(1,2) << ' ' <<  max(2,1)
<< ' '<< max('a','z') << ' ' <<
max(3.14,2.72) << '\n';
return 0;
}
```

```
1 1 a 2.72
```

Function Objects(Functors)

- Instances of classes that overload the function call operator (operator()).
  - Functors are objects that can be treated as a function.

```
 // A Function Object
class increment{
      int num;
public:
      increment(int n) : num(n) {  }
// overloading operator function () to make it callable
      int operator () (int arr_num) const {
            return num + arr_num;
      }
};
```

Function Objects(Functors)

- Instances of classes that overload the function call operator (operator()).
  - *std::transform* requires a unary function(a function taking only one argument).

```
#include <algorithm>
#include <iostream>
using namespace std;

int main() {
int arr[] = {1, 2, 3, 4, 5};
int n = sizeof(arr)/sizeof(arr[0]);
int to_add = 5;
```

```
// transform : applies a function to
// each elements of the array
transform(arr, arr+n, arr,
          increment(to_add));
for (int i=0; i<n; i++)
  cout << arr[i] << " ";
}
```

```
6 7 8 9 10
```

Function Objects(Functors)

- STL functions often utilize functors as an argument.
  - Algorithms like find_if take a unary predicate that operates on the elements of a sequence.

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
class modular{       int mod;
public :
modular(int mod_in):mod(mod_in){}
bool operator()(int x){
     return !(x % mod);
}}
```

```
int main() {
 vector<int> myvector;
 myvector.push_back(10);
 myvector.push_back(25);
 myvector.push_back(55);
 auto it =  find_if(myvector.begin(),
      myvector.end(), modular(2));
 // find element satisfying the
predicate
 return 0;}
```

Function Objects(Functors)

- STL functions often utilize functors as an argument.
  - Algorithms like sort use a binary predicate as a comparator.

```
#include <algorithm>
#include <vector>
#include <iostream>
class comparator{
 bool operator()(int a, int b){  return a > b; }
}
void printvec(vector<int> g){
 for (auto elem : g) { cout << elem << " " ; }
 cout << endl;
}
```

```
int main(){
 vector<int> g;
 g.push_back(9);
 g.push_back(20);
 printvec(g)
 sort(g.begin(), g.end(),
comparator)
 printvec(g)
}
```

C++ Standard Library

- The C++ Standard Library is a standardized library of C++, based upon conventions introduced by the STL.
- The C++ Standard Library provides collection of all the tools and utilities available in C++,
  - Containers & functions to utilize and manipulate these.
  - strings
  - streams (including interactive and file I/O)
  - maths
  - e.t.c.

C++ Standard Library

- Refined syntax and semantics of generic algorithms.
- Ensures performance.
  - introsort, the best sorting algorithm around is used for sort().
- Stable, well-maintained, widely available since it underwent ISO standardization as a part of the C++ ISO Standardization effort.

Numerics Library

- Common mathematical functions and types, as well as optimized numeric arrays.
  - Mathematical functions : <cmath>
    - std::fabs, std::sqrt, std::sin, std::beta, std::hermite, ,std::cyl_bessel_i, e.t.c.
  - Complex number arithmetic : <complex>
  - Numeric arrays : <valarray>
    - numeric arrays, array masks and array slices

Numerics Library

- Numeric algorithms : <numeric>
  - gcd : returning the greatest common divisor of two integers
  - lcm : returning the least common multiple of two integers
  - iota : fills a range with successive increments of the starting value

Random Number Generation

- Pseudo-random number generation
  - The header <random> defines several pseudo-random number generators and numerical distributions.

```
#include <iostream>
#include <random>
using namespace std;
int main () {
  random_device example;
  cout << "Random number between "
       << example.min() << " and "
       << example.max() << " : "
       << example() << endl;
       return 0;
}
```

```
Random number between
0 and 4294967295 :
3705944883
```

String

- The header <string> provides support for objects that represent sequences of characters, just like java.lang.String.

```
#include <string>
#include <iostream>
using namespace std;
 int main() {
string str("first string");
string str2(str.begin(), str.begin() + 5);
cout << str2.length() << endl;
cout << str2.at(2) << endl;
str2.append(" extension");
cout << str2 << endl;
```

```
cout << str2.substr(7, 3) << endl;
str2.replace(2, 7, "cest la vie");
cout << str2 <<endl;
return 0;
}
```

```
6
r
first extension
xte
ficest la vieension
```

String

- <regex> provides utilities for pattern matching strings, based on ‘regular expression’.

```
#include <iostream>
#include <string>
#include <regex>
int main (){
 string s ("subject");
 string fals(" reject");
 regex e ("(sub)(.*)");
 if (regex_match ("subject", e))
   cout << "string literal matched\n";
 if (regex_match (s,e))
   std::cout << "string object
matched\n";
 if ( not regex_match (fals,e))
   std::cout << "string object not
matched\n";
 return 0;
}
```

```
string literal matched
string object matched
string object not matched
```

Chrono

- <chrono> is used to deal with date and time.
  - Time elements, such as std::chrono::duration, std::chrono::time_point.
  - Functions for outputting time in various units.

```
#include <iostream>
#include <chrono>
using namespace std::chrono;
using namespace std;
int main() {
  time_point<system_clock> start, end;
```

```
start = system_clock::now();
  // Procedures you want to
measure the latency
end = system_clock::now();
  duration<double>
elapsed_seconds = end - start;
  cout << "elapsed time: " <<
elapsed_seconds.count() << "s\n";
}
```

Chrono

- time_point expresses a current point in time.
- duration expresses the timespan between two time_points.

```
#include <iostream>
#include <chrono>
using namespace std::chrono;
using namespace std;
int main() {
  time_point<system_clock> start, end;
  start = system_clock::now();
  // Procedures you want to measure
the latency
end = system_clock::now();
  duration<seconds>
elapsed_seconds = end - start;
  cout << "elapsed time: " <<
elapsed_seconds.count() << "s\n";
duration<millisecond>
elapsed_seconds = end - start;
  cout << "elapsed time (ms): " <<
elapsed_seconds.count() << "s\n";
}
```

Chrono

- clock : system_clock, steady_clock, high_resolution_clock
  - high_resolution_clock provides the smallest possible tick period.

```
#include <iostream>
#include <chrono>
using namespace std::chrono;
using namespace std;
int main() {
 time_point<high_resolution_clock>
start, end;
   start = high_resolution_clock::now();
```

```
   // Procedures you want to
measure the latency
end = high_resolution_clock::now();
   duration<nanoseconds>
elapsed_seconds = end - start;
 cout << "elapsed time (ns): " <<
elapsed_seconds.count() << "s\n";
```

Libraries from C

- The C language library is also included as a subset of the C++ Standard library.
  - Each header file has the same name as the C language version but with a "c" prefix and no extension.
    - For example, the C++ equivalent for the C language header file <stdlib.h> is <cstdlib>.
  - Nevertheless, for compatibility with C, the header names name.h (like stdlib.h) are also provided.
- Mixed use of C and C++, or backward compatibility with C.

Libraries from C

- <cstdio> (stdio.h) : C library to perform Input/Output operations
- <cstdlib> (stdlib.h) : C Standard General Utilities Library (header)

```
#include <cstdio>
using namespace std;
int main(){
 printf("Hello World!"); // print on console output
 printf("Characters: %c %c \n", 'a', 65);
 printf("Decimals: %d %ld\n", 1977, 650000L);
 return 0;
}
```

```
Hello World!
Characters: a A
Decimals: 1977
650000
```

Guidelines

- Use libraries wherever possible
  - Save time. Don’t replicate the work of others.
  - Benefit from other people’s work when they make improvements. Help other people when you make improvements.

Guidelines

- Prefer the standard library to other libraries
  - It is more likely to be stable, well-maintained, and widely available than your own code or most other libraries.