



# Week 5

---



# Chapter 10: The Standard Template Library

---

## Outline:

- C++ Standard Library
- STL
- Iterators
- Containers
  - Adapters: stacks, queues, priority queues.
  - Sequence Containers: Vector, List
  - Associative Containers: map, multimap, set, multiset.
- Algorithms



# C++ Standard Library

---

- The C++ standard library provides many useful routines for mathematics, I/O, etc.
- To use the routines in the standard library the **relevant header files must be included** in the user source files before compilation.
- Some routines in C++ standard library correspond to C standard library. The C++ header file is `<cXYZ>` if the corresponding C header file is `<XYZ.h>`.
- A significant part of the standard library uses templates and is called the C++ **Standard Template Library (STL)**.



# Standard Library Categories

<i>Category</i>	<i>Header Files</i>
Language support	cstddef, limits, climits, cfloats, new, typeinfo, exception, cstdint, csetjmp, csignal, cstdlib
Diagnostics	stdexcept, cassert, cerrno
General Utilities	utility, functional, memory, cstring, ctime
Strings	string, cctype, cwctype, cstring, cwstring, cstdlib
Localization	locale, clocale
Containers	deque, list, queue, stack, vector, map, set, bitset
Iterators	iterator
Algorithms	algorithm, cstdlib
Numerics	complex, valarray, numeric, cmath, cstdlib
Input and output	iosfwd, iostream, ios, streambuf, istream, ostream, iomanip, sstream, fstream, cstdio

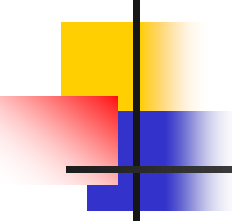
[http://msdn.microsoft.com/en-us/library/a7tkse1h\(vs.80\).aspx](http://msdn.microsoft.com/en-us/library/a7tkse1h(vs.80).aspx)



# Prototypes of Standard Library Functions

---

- The system header file `iostream` already contains the prototypes of many standard library functions.
- Thus it may not be necessary to include the specific header file for a particular library function's prototype.



# Standard Library Functions

## Example (1)

---

- The simple random number generator `rand()` generates a sequence of non-negative random numbers ( $< \text{RAND\_MAX}$ ).
- When the same seed is used in calling `srand(seed)`, `rand()` will produce the same sequence of random numbers.

# Standard Library Functions

## Example (2)

```
$ cat rand.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main(int ac, char *av[]) {
```

```
    int seed = 0;
```

```
    if(ac>1) seed = atoi(av[1]);
```

```
    srand(seed);
```

```
    int i, c[10], N=1<<20;
```

```
    for(i=0; i<10; i++) c[i] = 0;
```

```
    for(i=0; i<N; i++) c[rand()%10]++;
```

```
    for(i=0; i<10; i++)
```

```
        cout << c[i]/float(N) << '\n';
```

```
}
```

```
$ g++ rand.cpp
```

```
$ a.out
```

```
0.0999527
```

```
0.100307
```

```
0.0999861
```

```
0.0997295
```

```
0.0998964
```

```
0.0998697
```

```
0.100133
```

```
0.100414
```

```
0.100032
```

```
0.099679
```



# Standard Library Functions

## Example (3)

---

```
$ cat ctype.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main(int ac, char *av[]) {
```

```
    int c;
```

```
    if(ac<2) return -1; else c = atoi(av[1]);
```

```
    if(isprint(c)) cout << char(c) << ": ";
```

```
    else cout << "code " << c << " nonprintable";
```

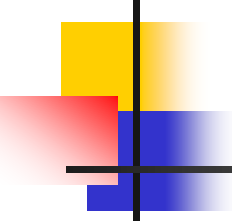


# Standard Library Functions

## Example (5)

```
if(isalnum(c)) cout << " isalnum";  
if(isalpha(c)) cout << " isalpha";  
if(iscntrl(c)) cout << " iscntrl";  
if(isdigit(c)) cout << " isdigit";  
if(isgraph(c)) cout << " isgraph"; // isprint() exclude blank  
if(islower(c)) cout << " islower";  
if(isprint(c)) cout << " isprint";  
if(ispunct(c)) cout << " ispunct";  
if(isspace(c)) cout << " isspace";  
if(isupper(c)) cout << " isupper";  
if(isxdigit(c)) cout << " isxdigit";  
cout << endl;
```

```
}
```



# Standard Library Functions

## Example (6)

---

```
$ for c in 33 44 55 66 77 88 99
```

```
> do
```

```
> a.out $c
```

```
> done
```

```
!: isgraph isprint ispunct
```

```
,: isgraph isprint ispunct
```

```
7: isalnum isdigit isgraph isprint isxdigit
```

```
B: isalnum isalpha isgraph isprint isupper isxdigit
```

```
M: isalnum isalpha isgraph isprint isupper
```

```
X: isalnum isalpha isgraph isprint isupper
```

```
c: isalnum isalpha isgraph islower isprint isxdigit
```

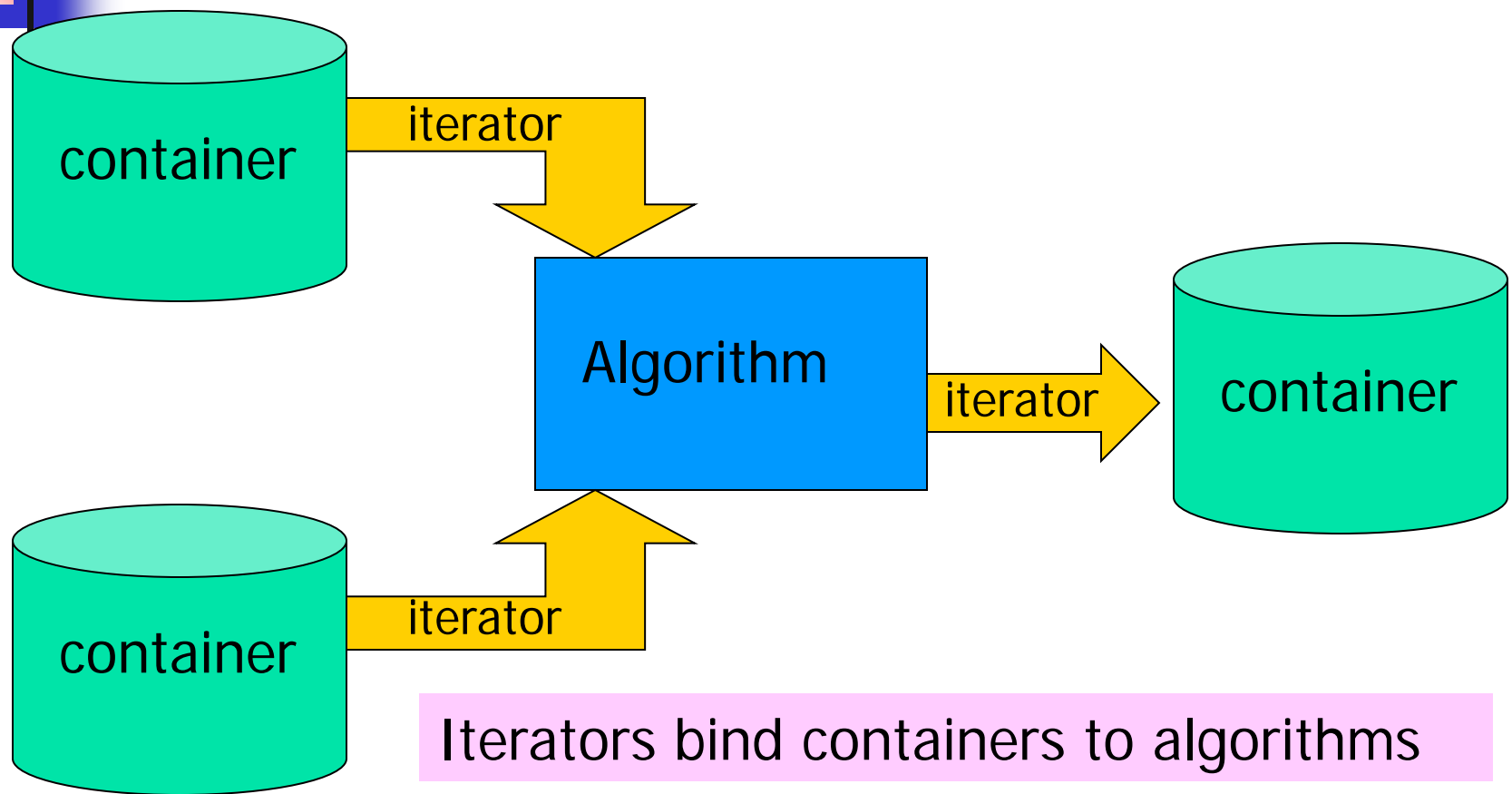


# C++ Standard Template Library (STL)

---

- The C++ STL is a subset of the C++ standard library.
- It is a collection of generic algorithms and class templates.
- The generic **algorithms** provides standard functions for sorting, searching, and iterating.
- The class templates provides standard data structures such as queues and stacks. These STL data structures are known as C++ **containers**.
- The STL algorithms work on the STL classes instantiated from the STL class templates.

# STL Main components





# Iterators

---

An **iterator** is a “general pointer” that “points” to a particular element in a **container**.

We want an iterator to be able to do what a pointer can do.



# What can a pointer do?

---

```
int a[] = {1,2,3,4,5,6,7,8,9};  
// access the elements through pointer, auto-increment  
// operator ++, and dereference operator *  
int *p;  
for (p = a; p != a+9; ++p) {  
    cout << *p << endl;  
}
```

A pointer can be

1. initialized to point to the **begin** of the container (the array)
2. compared with another pointer for inequality to see whether it has come to the **end** of the container
3. incremented (**++**) to point to the next element in the container
4. dereferenced (**\***) to access the value of the element in the container



# Operations on Iterators

---

Let *iter* be an iterator of a container

**\***: Accesses the value of the item pointed to by the iterator.

*\*iter*

**++**: Moves the iterator to the next item in the container.

*iter++* or *++iter*

**--**: Moves the iterator to the previous item in the container.

*iter--* or *--iter*

**==**: Takes two iterators as operands and returns true when they both point at the same item in the container.

*iter1 == iter2*

**!=**: Returns true when the two iterators do not point at the same item in the container.

*iter1 != iter2*

*\*iter++* is equivalent to *\*(iter++)*



# begin, end

---

- All container classes (adaptors excluded) provide their own iterators
- iter, an iterator of a container is described as `container::iterator iter`;
- `begin()` return an iterator representing the beginning of the container
- `end()` return an iterator pointing after the end of the container



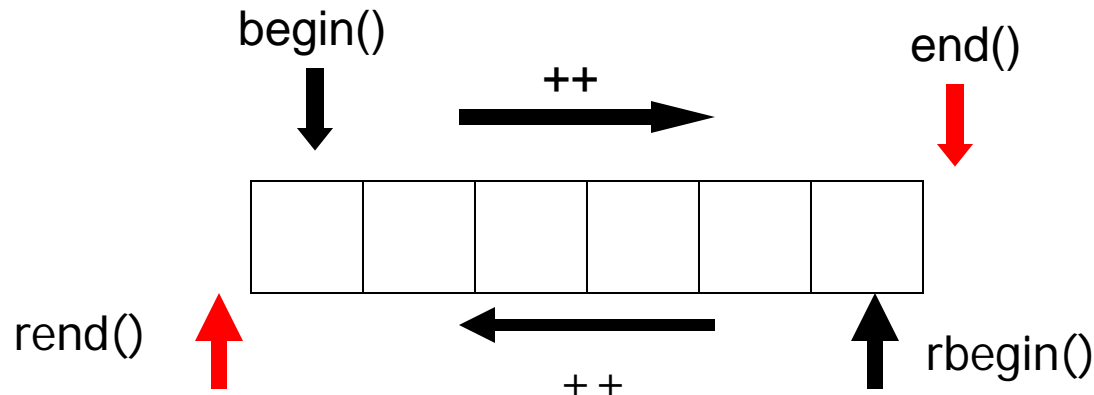
# Iterators, const\_iterator, reverse\_iterators

```
for (list<int>::iterator iter = int_list.begin();  
    iter != int_list.end(); ++iter)  
    cout << *iter << endl;
```

`const_iterator` treats a container as a constant and does not allow modification

reverse direction

```
for (list<int>::reverse_iterator iter =  
    int_list.rbegin(); iter != int_list.rend();  
    ++iter) std::cout << *iter << endl;
```





# Types of Iterators

---

- There are many types of iterator:
  - forward iterators,
  - bidirectional iterators,
  - random access iterators,
  - input iterators,
  - output iterators.
- Almost all STL algorithms and data structures use iterators



# Container classes

---

## Sequences

- vector
- list

## Container adaptors (no iterator)

- stack
- queue
- priority\_queue

## Associative Containers

- set
- map
- multiset
- multimap

All container classes are template classes



# The STL Vector Container

---

The vector container is an extensible array implemented as a class with many useful operations.

# Vectors are Extensible Arrays

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v; // allocate an empty vector
    cout << "size\tcapacity" << endl;
    cout << v.size() << "\t" << v.capacity() << endl;
    /* v[0] = 0; this triggers a segmentation fault */
    for(int i=0; i<9; i++) {
        v.push_back(i);
        cout << v.size() << "\t" << v.capacity() << endl;
    }
}
```

```
$ g++ vec1.cpp
$ a.out
size    capacity
0        0
1        1
2        2
3        4
4        4
5        8
6        8
7        8
8        8
9       16
```

# Random Vector Access and Range Checking (1)



```
$ cat vec2.cpp
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<int> a(6);
```

```
    int i;
```

```
    for(i=0; i<a.size(); i++) a[i]=i*i;
```

```
    for(i=0; i<=a.size(); i++) cout << a[i] << ' '; cout << '\n';
```

```
    try {
```

```
        for(i=0; i<=a.size(); i++) cout << a.at(i) << ' ';
```

```
    } catch(...) { cout << "exception\n"; }
```

```
}
```



# Random Vector Access and Range Checking (2)

---

```
$ g++ vec2.cpp
```

```
$ a.out
```

```
0 1 4 9 16 25 262247
```

```
0 1 4 9 16 25 exception
```

Note:

- Even though 6 exceeds the range 0..5, using `a[6]` will not cause an exception and garbage was printed.
- `a.at(6)` will throw an out-of-range exception



# Forward/Backward Vector Iteration

```
$ cat vec3.cpp
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<char> c(6);
```

```
    c[0]='c'; c[1]='s'; c[2]='1';
```

```
    c[3]='2'; c[4]='8'; c[5]='0';
```

```
    vector<char>::iterator f;
```

```
    for(f=c.begin(); f!=c.end(); f++) cout << *f;
```

```
    cout << endl;
```

```
    vector<char>::reverse_iterator b;
```

```
    for(b=c.rbegin(); b!=c.rend(); b++) cout << *b;
```

```
    cout << endl;
```

```
}
```

```
$ g++ vec3.cpp
```

```
$ a.out
```

```
cs1280
```

```
0821sc
```





# Grow/Shrink at Vector Tail (1)

To grow or shrink at vector tail is efficient

```
$ cat vec4.cpp
```

```
// To change vector's content from "rite" to "ritual"
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<char> c(4);
```

```
    c[0]='r'; c[1]='i'; c[2]='t'; c[3]='e';
```

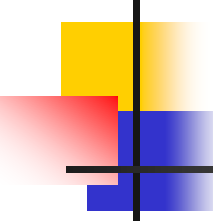
```
    cout << c.front() << c.back() << '\n';
```



# Grow/Shrink at Vector Tail (2)

```
c.push_back('s');  
cout << c.front() << c.back() << '\n';  
c.pop_back(); cout << c.back() << '\n';  
c.pop_back(); cout << c.back() << '\n';  
c.push_back('u'); c.push_back('a');  
c.push_back('l');  
for(int i=0; i<c.size(); i++) cout << c[i];  
cout << endl;  
}
```

```
$ g++  
    vec4.cpp  
$ a.out  
re  
rs  
e  
t  
ritual
```



# Grow/Shrink a Vector Anywhere (1)

To grow or shrink at anywhere of a vector is inefficient

```
$ cat vec5.cpp
```

```
// To change vector's content from "rite" to "right"
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<char> c(4);
```

```
    c[0]='r'; c[1]='i'; c[2]='t'; c[3]='e';
```

```
    for(int i=0; i<c.size(); i++) cout << c[i];
```

```
    cout << '\n';
```

# Grow/Shrink a Vector Anywhere (2)

```
c.erase( c.end()-1 ,c.end());  
c.insert( c.begin()+2 ,'h');  
c.insert(c.begin()+2,'g');  
for(int i=0; i<c.size(); i++)  
    cout << c[i];  
cout << endl;  
}
```

```
$ g++ vec5.cpp  
$ a.out  
rite  
right
```



# Searching/Sorting a Vector (1)

---

```
$ cat vec6.cpp
```

```
#include <iostream>
```

```
#include <vector>
```

```
#include <string>
```

```
#include <algorithm>
```

```
using namespace std;
```

```
int main(int ac, char *av[]) {
```

```
    vector<string> s;
```

```
    if(ac<2) return -1;
```

```
    for(int i=1; i<ac; i++)
```

```
        s.push_back(av[i]);
```

```
// populate the vector with av[]
```



# Searching/Sorting a Vector (2)

---

```
vector<string>::iterator t;  
string aim;  
while( cin >> aim ) {  
    t = find(s.begin(), s.end(), aim);           // search a vector  
    if( t != s.end() )  
        cout << *t << " found\n";  
    else  
        cout << aim << " not found\n";  
}
```



# Searching/Sorting a Vector (3)

```
sort(s.begin(),s.end());    // sort a vector
for(int i=0; i<s.size(); i++) cout << s[i] << ' ';
cout << endl;
}
```

```
$ g++ vec6.cpp
$ a.out lion cat tiger dog wolf fox elephant
wolf
wolf found
eagle
eagle not found
^D
cat dog elephant fox lion tiger wolf
```



# STL List Container

---

The list container is a doubly linked list.





# Simple List Container Example (1)

---

```
$ cat list1.cpp
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<char> c;
    c.push_back('0');    // insert at tail
    c.push_front('1');   // insert at head
```



# Simple List Container Example (2)

---

```
// insert right following head
```

```
c.insert(++c.begin(),'8');
```

```
c.insert(++c.begin(),'2');
```

```
c.push_back('S');
```

```
// insert at tail
```

```
c.push_front('s');
```

```
// insert at head
```

```
c.push_front('c');
```

```
// insert at head
```

```
for( list<char>::iterator i=c.begin(); i!= c.end(); ++i)
```

```
    cout << *i;
```

```
cout << endl;
```

```
}
```

```
$ g++ list1.cpp
```

```
$ a.out
```

```
cs1280S
```



# Another List Example (1)

---

```
$ cat list2.cpp
#include <iostream>
#include <list>
using namespace std;
class rec {
    string name;
    double wage;
    int year;
```



## Another List Example (2)

---

```
friend ostream &operator<<(ostream &out,const rec &rhs) {  
    return out  
        << rhs.name << '\t'  
        << rhs.wage << '\t'  
        << rhs.year;  
}
```



## Another List Example (3)

---

public:

```
rec(string n="unknown", double w=0.0, int y=2008): name(n),  
    wage(w), year(y) {}
```

```
bool operator<(const rec &rhs) {    // for sort  
    return (year < rhs.year) ||  
        ((year == rhs.year) && (wage < rhs.wage)) ||  
        (year==rhs.year && wage==rhs.wage &&  
            name<rhs.name);  
}
```

```
};
```



# Another List Example (4)

---

```
int main() {  
    list<rec> team;  
    team.push_back( rec("James", 1280.0, 1990) );  
    team.push_back( rec("James", 2180.0, 1983) );  
    team.push_back( rec("David", 2180.0, 1983) );  
    team.push_back( rec("Frank", 2810.0, 2002) );  
    team.push_back( rec("David", 8210.0, 2002) );  
}
```



# Another List Example (5)

```
list<rec>::iterator i;
for(i=team.begin(); i!=team.end(); i++)
    cout<<*i<<endl;
cout << "****" << endl;
team.sort();
for(i=team.begin(); i!=team.end(); i++)
    cout<<*i<<endl;
}
```

```
$ g++ list2.cpp
$ a.out
James 1280 1990
James 2180 1983
David 2180 1983
Frank 2810 2002
David 8210 2002
***
David 2180 1983
James 2180 1983
James 1280 1990
Frank 2810 2002
David 8210 2002
```



# Vector versus List

---

- Efficiency

- Vector: constant time for random access and appending item, linear time for insertion.
- List: constant time for insertion at ends, linear time for random access.

- Functionality

- Vector: no `push_front()`, no `pop_front()`, no `merge()`, no `remove()`, no `remove_if()`, no `reverse()`, no `sort()`, no `splice()`, no `unique()`.
  - Have to use the missing functions found in `<algorithm>`
- List: no `operator[]`, no `at()`.
  - Some functions in `<algorithm>` are adapted as List's member functions





# Adapters

---

An adapter makes an underlying container (vector or list) behave like another data structure. Container adapters are stacks, queues, priority queues.



# stack

---

```
template <class T>
class stack {
public:
    bool empty() const;
    size_type size() const; // unsigned int
    T& top(); // Dangerous! top of stack exposed!
    void push(const T& t);
    void pop();
};
```



# queue

---

```
template <class T>
class queue {
public:
    bool empty() const;
    size_type size() const;
    T& front();
    const T& front() const;
    T& back();
    const T& back() const;
    void push (const T& t); // enqueue
    void pop(); // dequeue
};
```



# Code to check for palindrome

---

```
bool palindrome (string v) {  
    stack <char> s ;  
    queue <char> q ;  
    int len = v.size ();  
    // push string into stack and queue  
    for (int j=0; j < len; j++) {  
        s.push (v[j] );  
        q.push (v[j] );  
    }  
    while (!s.empty()) {  
        if (s.top() != q.front()) return false;  
        s.pop();  
        q.pop();  
    }  
    return true;  
}
```



# Priority Queue

---

A Special form of queue from which items are removed according to their designated priority and not the order in which they entered.



# STL priority\_queue

---

To use STL priority\_queue,

```
#include <queue>
```

To create an empty priority queue.

e.g. `priority_queue<int> pq;`

```
bool empty() const;
```

```
void push (const T& item);
```

```
int size() const;
```

```
T& top();           // Return a reference to the item having  
                   // the highest priority.
```

```
void pop();         // Remove the item of highest priority
```



# Priority Queue Example

---

```
#include <queue>
using namespace std;
int main() {
    priority_queue<int> pq;
    pq.push(10);
    pq.push(20);
    while (!pq.empty()){
        cout << pq.top() << " ";    // 20 10
        pq.pop();
    }
    return 0;
}
```



# Associative Containers

---

STL associative containers are

map, multimap, set, multiset.

The elements of an associative container can be accessed with a key rather than an index.





# Sets –(1)

---

`#include <set>`

- Elements are sorted based on < (less than)
- Implemented as **balanced binary search trees**.  
Therefore, perform search well.
- *Cannot* change the value of an element directly.  
Must remove old element first and then insert new element.
- **Multiset** is similar to the **set** but allows duplicates.



## Sets – (2)

---

- Creating a set of integers:  
`set<int> int_set;`  
`multiset<int> m_set;`
- Insert an element:  
`int_set.insert(10);`
- Get the size of the set:  
`int_set.size();`
- Comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`  
`int_set1 == int_set2`



## Sets – (3)

---

- Assign elements of one set to another:

```
int_set1 = int_set2;
```

- Remove all elements with key value 5 and return the number of removed elements:

```
int num_removed = int_set.erase(5);
```

- Count all elements with value 5:

```
int num_fives = int_set.count(5);
```

- Remove all elements:

```
int_set.clear();
```

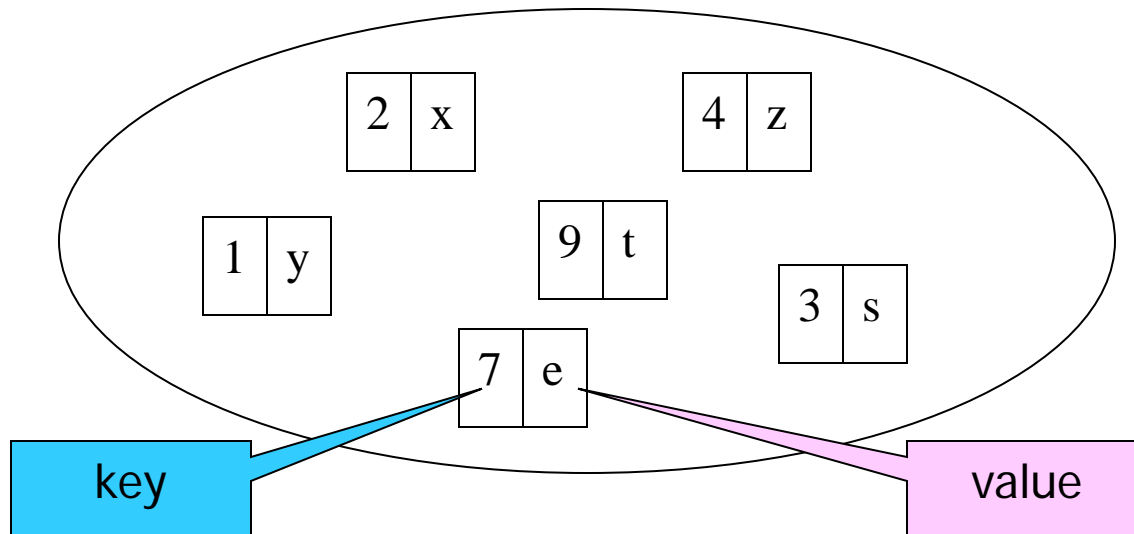
- Find an element:

```
if(int_set.find(5) == int_set.end())  
    cout << "5 is not in the set";
```

# Maps – (1)

`#include <map>`

- Manage key/value pairs as elements.
  - A pair has 2 public attributes: `first` and `second`
- Elements sorted based on the keys.





## Maps – (2)

---

- `Multimap` allows duplicates, but `map` does not.
- Create a map:  
`map<string, double> pricelist;`
- Insert an element (either update an existing entry or create a new entry):  
`pricelist["apple"] = 0.65;`



## Maps – (3)

---

- Size of the map:  
`pricelist.size();`
- Assignment:  
`pricelist1 = pricelist;`
- Comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`  
`pricelist1 == pricelist2;`
- Get the number of elements having a particular key:  
`pricelist.count("apple");`



# In map, key are sorted

---

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main(){
    map<int,char*> m;           // <position, message>
    m.insert(make_pair(3,"C++")); // m[3]="C++"
    m.insert(make_pair(1,"I"));   // m[1] = "I"
    m.insert(make_pair(2,"like")); // m[2] = "like"
    for (map<int,char*>::iterator iter = m.begin();
        iter != s.end(); ++iter) {
        cout << iter->second << ' '; // (*iter).second
    }
    cout << endl;
}
```



# erase

## through key, through iterator

```
int main() {
    map<string, int> word_count;           // <word, count>
    string word;
    while (cin >> word)    ++word_count[word];
    string removal_word = "the";           // erase through key
    if (word_count.erase(removal_word))    // no. of pairs erased
        cout << "ok: " << removal_word << " removed\n";
    else cout << "oops: " << removal_word << " not found!\n";
    map<string,int>::iterator where;        // erase through iterator
    removal_word = "a";
    where = word_count.find(removal_word);
    if (where == word_count.end())
        cout << "oops: " << removal_word << " not found!\n";
    else {
        word_count.erase(where);
        cout << "ok: " << removal_word << " removed!\n";
    }
    return 0; }
```





# Map Example (1)

---

```
$ cat aa.cpp
```

```
#include <iostream>
```

```
#include <map>
```

```
#include <string>
```

```
using namespace std;
```

```
int main() {
```

```
    map<string,int> DoB;
```

```
    map<string,int>::iterator i;
```

```
    DoB["Confucius"] = -551;
```

```
    DoB["George-Washington"] = 1732;
```

```
    DoB["Bill-Gates"] = 1955;
```

```
    DoB["Mahatma-Gandhi"] = 1869;
```

```
    DoB["Leonardo-da-Vinci"] = 1452;
```



## Map Example (2)

```
for(i=DoB.begin(); i != DoB.end(); ++i)
    cout << i->first << '\t'
        << i->second << endl;
cout << "\n->Enter a Famous Name\n\n";
string name;
while( cin >> name ) {
    cout << name << '\t'
        << DoB[name] << endl;
}
}
```

What will be printed if you enter your name?

```
$ g++ aa.cpp
$ a.out
Bill-Gates          1955
Confucius           -551
George-Washington  1732
Leonardo-da-Vinci  1452
Mahatma-Gandhi     1869
->Enter a Famous Name
Leonardo-da-Vinci
Leonardo-da-Vinci  1452
Confucius
Confucius           -551
Bill-Gates
Bill-Gates          1955
^D
```



# Function object

---

- A function object is
  - an object of a class that defines `operator()` so that its objects act like a function. Also called a **functor**.
  - an ordinary bool function.
- It has many uses e.g. to customize the sorting algorithms.
- Many template function objects are provided in STL:
  - `equal_to`
  - `not_equal_to`
  - `less`
  - `less_equal`
  - `greater`
  - `greater_equal`
- To use these function objects,  
`#include <functional>`



# Function object usage

---

```
#include <iostream>
using namespace std;
class compare {
public:
    int operator()(int x, int y) const { return x > y; }
};
int main()
{
    compare v;
    cout << v(2, 15) << endl;           // Output: 0
    cout << compare() (5, 3) << endl;    // create object, then ()
                                         // Output: 1
    cout << less<int>() (5, 3) << endl;  // Use system functor
    return 0;
}
```



# Algorithms

---



# <algorithm>

---

`#include <algorithm>`

- There are about 80 algorithms
- All STL algorithms process one or more iterator ranges.
- Some algorithms allow user defined operations.



# find, to search in an array

---

```
int main(){
    // use find to search an array
    int ia[6] = {27, 210, 12, 47, 109, 83};
    int search_value = 83;
    int *result = find (ia, ia + 6, search_value);
    cout << "The value " << search_value
        << (result == ia + 6 ? " is not present" : " is present")
        << endl;
```



# find, to search in a vector

---

```
vector<int> vec (ia, ia+6);  
// value we'll look for  
search_value = 42;  
// call find to see if that value is present  
vector<int>::const_iterator result =  
    find (vec.begin(), vec.end(), search_value);  
// report the result  
cout << "The value " << search_value  
    << (result == vec.end() ? " is not present" : " is present")  
    << endl;
```





# find, to search in a list

---

```
list<int> lst (ia, ia+6);
search_value = 47;
// call find to look through elements in a list
list<int>::const_iterator result =
    find (lst.begin(), lst.end(), search_value);
cout << "The value " << search_value
    << (result == lst.end() ? " is not present" : " is present")
    << endl;
return 0;
}
```



# sort

---

```
int main() {  
    // store words in a vector  
    vector<string> words;  
    string next_word;  
    while (cin >> next_word) {  
        words.push_back(next_word);  
    }  
    // sort words into ascending order  
    sort(words.begin(), words.end());  
    // sort words into descending order  
    sort(words.begin(), words.end(), greater<string>());  
}
```