C++ STL Containers

Sandra Batista, Mark Redekopp, and David Kempe

1.1–1.2

List

- 2 specialized List ADTs:
- Queues
- Stacks

Dictionary/Map

Set

Maps / Dictionaries

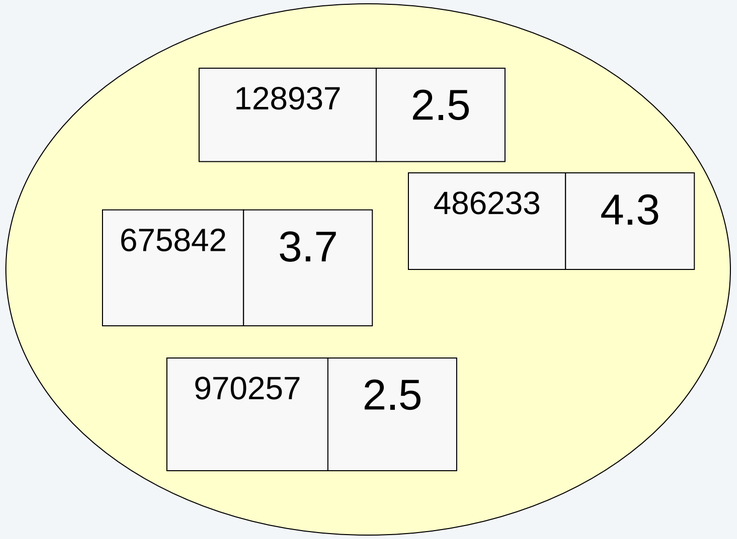
Stores key and value pairs

- Example: Map student IDs to their GPA

Keys must be unique

No constraints on the values

No inherent ordering between key value pairs



Map / Dictionary Operations

Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key,value pair to the dictionary (assuming its not there already)	Key, Value	
Remove	Remove the key,value pair with the given key	Key	
Get / lookup	Lookup the value associated with the given key or indicate the key,value pair doesn't exist	Key	Value associated with the key
In / Find	Check if the given key is present in the map	Key	bool (or ptr to pair/NULL)
empty	Returns true if there are no keys in the map		bool
size	Returns the number of keys in the map		int

Set

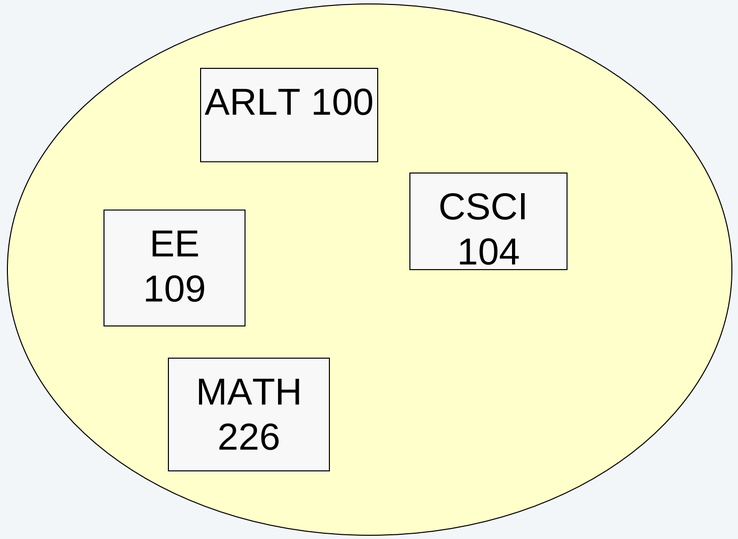
A set is a dictionary with only keys

- Example: All the courses taught at a university

Keys must be unique

- No duplicate keys (only one occurrence)

Not indexing or ordering



Set Operations

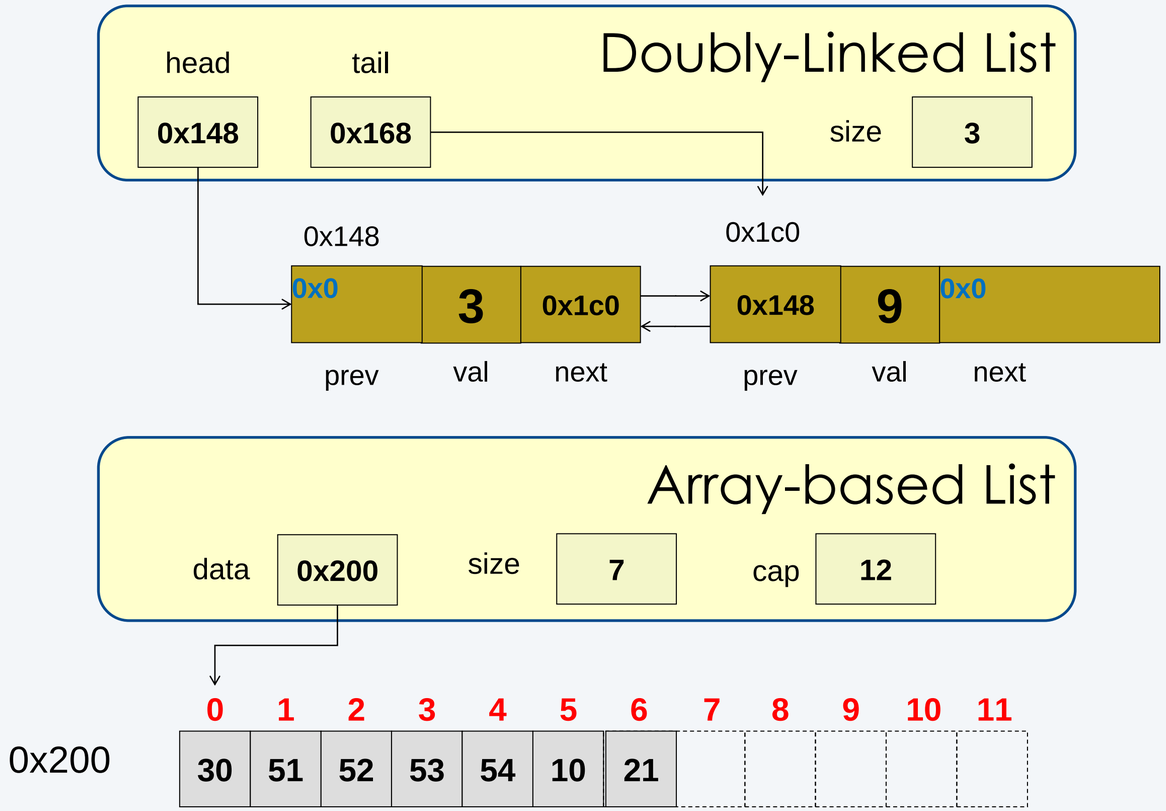
Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key to the set (assuming its not there already)	Key	
Remove	Remove	Key	
In / Find	Check if the given key is present in the set	Key	bool (or value)
empty	Returns true if there are no keys in the set		Bool
size	Returns the number of keys in the set		Int
intersection	Returns a new set with the common elements of the two input sets	Set1, Set2	New set with all elements that appear in both set1 and set2
union	Returns a new set with all the items that appear in either set	Set1, Set2	New set with all elements that appear in either set1 and set2
difference	Returns a set with all items that are just in set1 but not set2	Set1, Set2	New set with only the items in set1 that are not in set2

Overview

STL CONTAINERS

C++ Standard Template Library provides implementations of several sequential containers

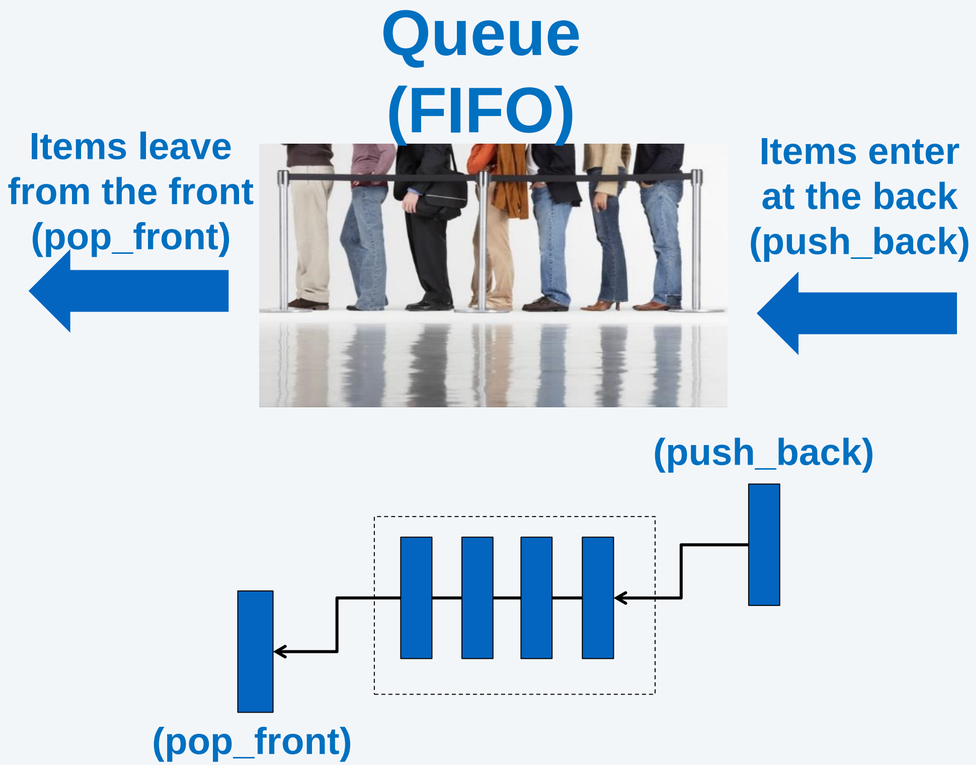
- DynamicArrayList => C++: std::vector<T>
- LinkedList => C++: std::list<T>

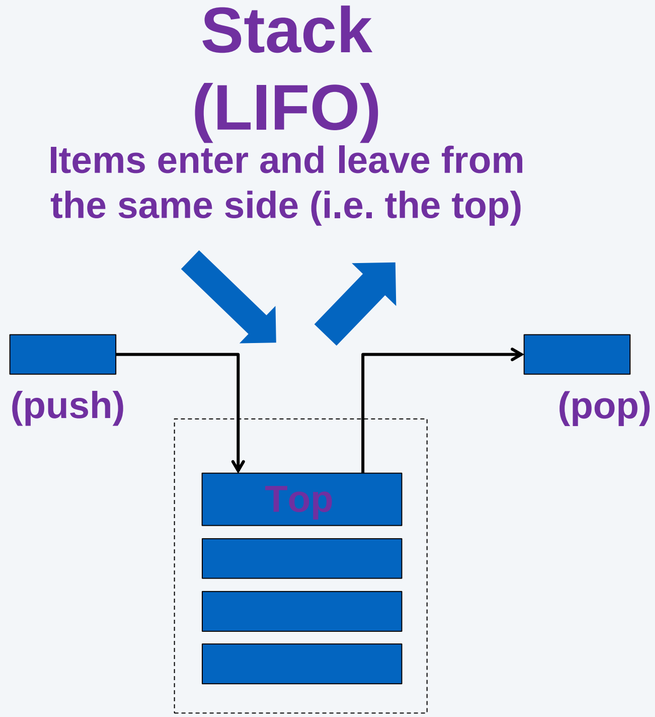


STL Container Adaptor Classes

C++ Standard Template Library provides implementations of several adaptor containers

- Queues => C++: std::queue<T>
- Stacks => C++: std::stack<T>

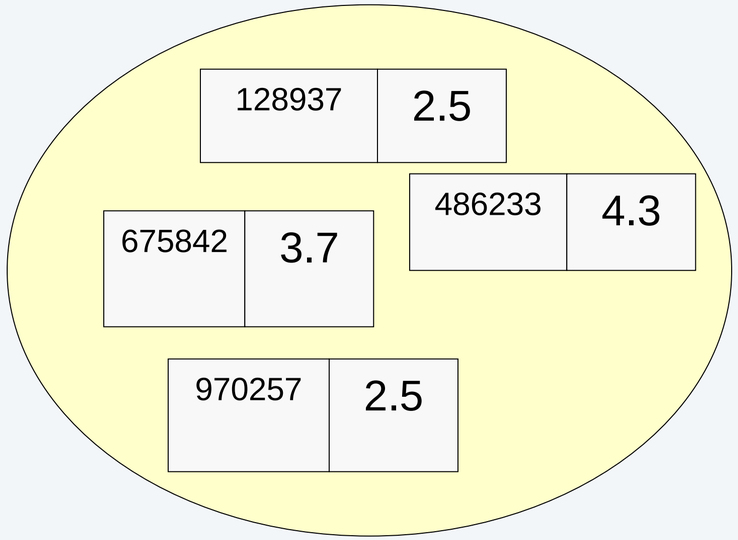


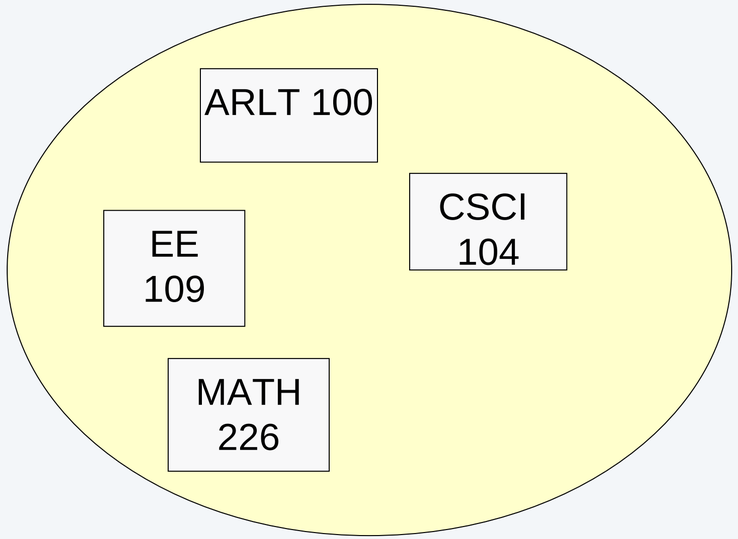


STL Associative Container Classes

C++ Standard Template Library provides implementations of several associative containers

- Sets => C++: std::set<T>
- Maps => C++: std::map<K,V>





How to traverse a container

STL ITERATORS

Iteration

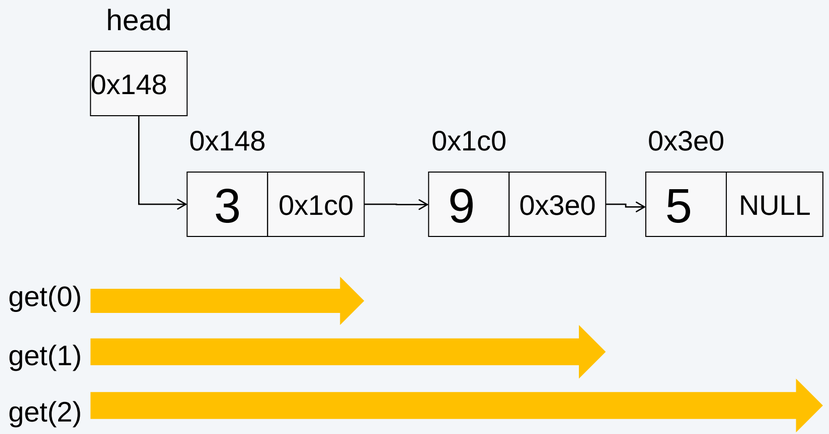
Consider how to traverse a list

For an array list?

```
vector<int> mylist;
...
for(int i=0; i < mylist.size(); ++i)
{
  cout << mylist[i]<< endl;
}
```

For a linked list?

```
list<int> mylist;
...
for(int i=0; i < mylist.size(); ++i)
{
  // How to get the i-th item without
  // traversing from the beginning?
}
```



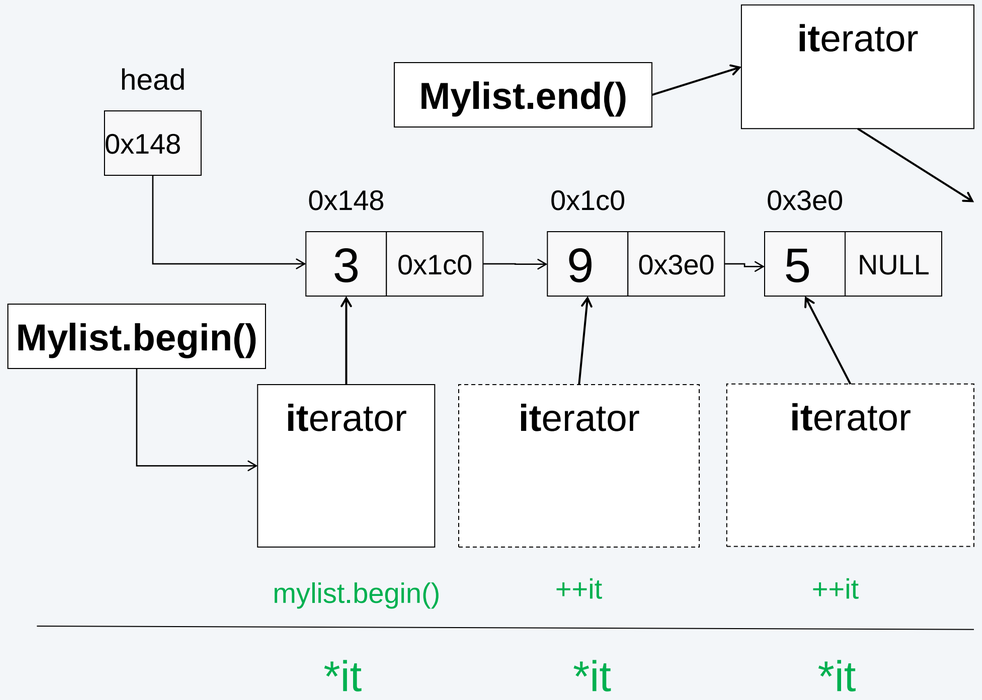
Iteration: A Better Approach

Use an **iterator**

Iterator tracks the internal location of each successive item

Iterators provide the semantics of a pointer

- begin() returns an iterator to the beginning item
- end() returns an iterator one item beyond the last item
- ++it moves iterator to next item




```
list<int> mylist;
...
iterator it = mylist.begin()
for(it = mylist.begin();
    it !=  mylist.end();
    ++it)
{
  cout << *it << endl;
}
```

Iterators

Iterators are a new class type defined **in the scope of each container**

- Type is ***container*::iterator** (vector<int>::iterator is a type)

1) Initialize them with *objname*.begin()
2) Check whether they are finished by comparing with *objname.end()*
3) Move to the next item with ++ operator

```
// vector.h
template<class T>
class vector
{
  class iterator {

  };
};
```

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  vector<int> my_vec(5); // 5 = init. size
  for(int i=0; i < 5; i++){
    my_vec.push_back(i+50);
  }
  vector<int>::iterator it;
  for(it = my_vec.begin() ; it != my_vec.end(); ++it){
     // Do work on items here
  }
}
```

Iterator has pointer semantics on an item in the container

- Use * to dereference and get the actual item

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  vector<int> my_vec(5); // 5 = init. size
  for(int i=0; i < 5; i++){
    my_vec.push_back(i+50);
  }
  for(vector<int>::iterator it = my_vec.begin() ; it != my_vec.end(); ++it){
     cout << *it << endl;
  }
  return 0;
}
```

C++ STL Algorithms

Many useful functions defined in <algorithm> library

- http://www.cplusplus.com/reference/algorithm/sort/
- http://www.cplusplus.com/reference/algorithm/count/

These functions accept iterator(s) to elements in a container

```
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;

int main()
{
  vector<int> my_vec(5); // 5 = init. size
  for(int i=0; i < 5; i++){
    my_vec.push_back(rand());
  }
  sort(my_vec.begin(), myvec.end());
  for(vector<int>::iterator it = my_vec.begin() ; it != my_vec.end(); ++it){
     cout << *it << endl;
  }
  return 0;
}
```

ASSOCIATIVE CONTAINERS

Maps / Dictionaries

Stores key and value pairs

- Example: Map student IDs to their GPA

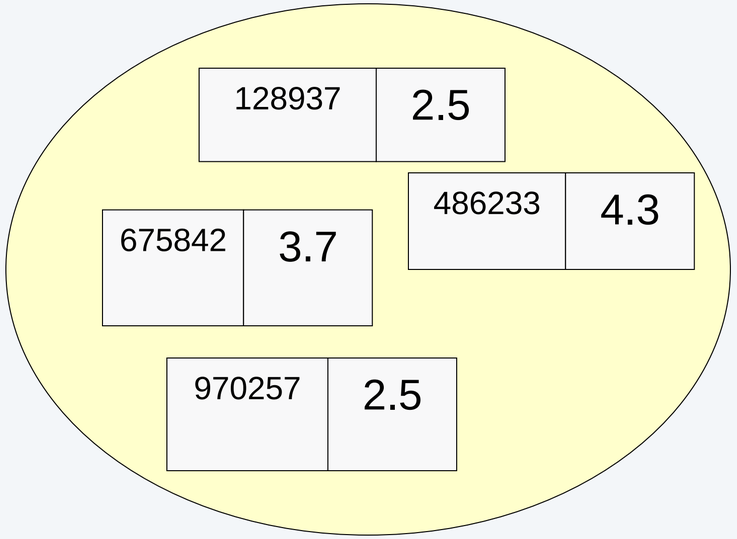
Keys must be unique

No constraints on the values

No inherent ordering between key,value pairs

Operations:

- Insert
- Remove
- Find/Lookup



C++ Pair Struct/Class

C++ library defines a struct pair holds two values

C++ map stores its key as the first value and value as second value in ***pair***

To use pair:

1. Instantiate a pair
2. Use `make_pair()` to do it

```
template <class T1, class T2>
struct pair {
  T1 first;
  T2 second;
}
```

```
#include <iostream>
#include <utility>
#include <string>
using namespace std;

void func_with_pair_arg(pair<char,double> p)
{   cout << p.first << " " << p.second <<endl;  }

int main()
{
  string mystr = “Cat";
  pair<string, int> p1(mystr, 1);
  cout << p1.first << " " << p1.second <<endl;


  func_with_pair_arg( pair<char,double>('c', 2.3) );

 func_with_pair_arg( make_pair('c', 2.3) );
}
```

Cat 1
c 2.3
c 2.3

STL Map

Maps store (key,value) pairs where:

- key = label to access the associated value
- Stored value is associated data

Keys must be unique

Value type should have a default constructor

Key type must have less-than (<) operator defined for it

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main()
{
  map<string,double> bodega_inventory;


  bodega_inventory[“plaintains”] = .99;
  bodega_inventory.insert(pair<string, double>(“mangos”, 2));
  bodega_inventory.insert(make_pair(“bread”, 3.99));


  double mango_price = bodega_inventory[“mangos”];


  bodega_inventory.erase( “plantains" );
  cout << “No plaintains this week.”;
  cout << endl;
}
```

bodega_inventory is a map that associates C++ strings for foods (keys) with doubles for prices(values)

Maps & Iterators

To iterate through all key value pairs in the map **using an iterator for the map type**

This iterator will point to a pair struct

- it->first is the key
- it->second is the value

```
#include <map>
#include <string>
using namespace std;

int main()
{
  map<string,double> bodega;

  bodega[“plaintains”] = .99;
  bodega.insert(pair<string, double>(“mangos”, 2));
  bodega.insert(make_pair(“bread”, 3.99));


  cout << “This week’s inventory: ” << endl;

  map<string,double>::iterator it;


  for(it = bodega.begin(); it != bodega.end(); ++it){

    cout << “Inventory item: “ << it->first;

    cout << “ Price: “ <<  it->second << endl;

  }

}
```

```
Output:
This week’s inventory:


Inventory item: bread Price: 3.99


Inventory item: mangos Price: 2


Inventory item: plaintains Price: 0.99
```

Map Membership [Find()]

Check/search whether key is in the map object using *find()* function

Pass a key as an argument

Find returns an iterator

If key is IN the map

- Returns an iterator/pointer to that (key,value) pair

If key is NOT IN the map

- Returns an iterator equal to *end()'s* return value

Efficient at finding specified key/value and testing membership ($O(\log_2 n)$)

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main()
{
  map<string,double> bodega;

  bodega[“plaintains”] = .99;
  bodega.insert(pair<string, double>(“mangos”, 2));
  bodega.insert(make_pair(“bread”, 3.99));



  map<string,double>::iterator it;

  it = bodega.find(“apples”);

  if (it == bodega.end()){
    cout << “No apples this week! “ << endl;
  } else {
    cout << it->first << “ weekly price: “ << it->second <<endl;
  }

}
```

Set Class

C++ STL set class has only keys

Keys are unique

insert() to add a key to the set

erase() to remove a key from the set

Very efficient at testing membership (**$O(\log_2 n)$**)

Key type must have a less-than (<) operator defined for it

Iterators to iterate over all elements in the set

Find() to test membership

```
#include <set>
#include <string>
#include <iostream,>
using namespace std;

int main()
{
  set<string> fruits;

  fruits.insert(“apples");
  fruits.insert(“watermelons");
  string f3= “grapes";
  fruits.insert(f3);

  for(set<string>::iterator it=fruits.begin();
      it != fruits.end();
      ++it){
    cout << “Fruit: " << *it << endl;
  }

 if(fruits.find(“cabbage”) != fruits.end()){
    cout<< “Cabbage is a fruit" << endl;
  }
  else {
    cout<< “Cabbage is not a fruit" << endl;
  }
  fruits.erase(“watermelons”);
  return 0;
}
```

Maps and sets use balanced binary search trees to store keys

This allows logarithmic find runtime

This is why the less-than (<) operator needs to be defined for the data type of the key