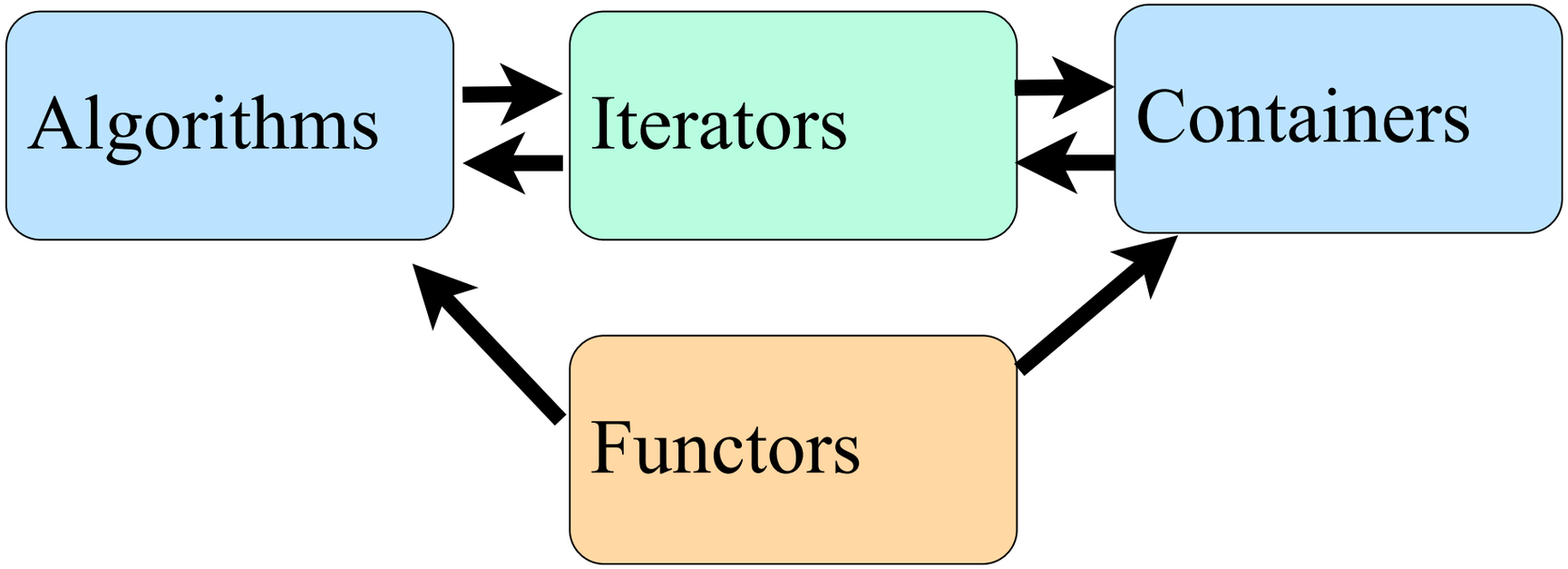
Easy to use code written by someone else.

STL
Standard Template Library



A C++ 11 STL reference can be found at:
http://en.cppreference.com/w/cpp
Another C++ reference can be found at:
http://www.cplusplus.com/reference/

Motivation for the STL Algorithms

Computers do very simple operations: add, shift by two, etc.
Compilers take the C++ language into machine code.
But we still have to write a large number of steps compared to typical pseudo code
STL algorithm take one more step to being able to write high level commands.

Find the average exams score

```
ifstream input("exam1.txt");
vector<double> exam_scores;

int score;
while ( input >> score )
    exam_scores.push_back(score);



// Compute the average
double total = 0;
for (vector<double>::iterator itr = exam_scores.begin(), itr != exam_scores.end(); ++itr)
     total += *itr;
cout << "Average score for exam 1 is " << total/exam_scores.size();
```

To use accumulate you need to add #include <numeric>

Instead Use a STL Algorithm

Find the average exams score.

```
ifstream input("exam1.txt");
vector<double> exam_scores;

int score;
while ( input >> score )
    exam_scores.push_back(score);



// Compute the average
cout << accumulate(exam_scores.begin(), exam_scores.end(), 0.0)/exam_scores.size();
```

#include<algorithm>
#include<numeric>

STL Algorithms

Algorithms typically have requirements for the type of iterator passed as a parameter. Bidirectional iterators can be used whenever forward iterators are required/used. Random access iterators can be used whenever bidirectional iterators are required/used.

- Iterator-based template functior
- Types of algorithms: non-modifying sequence operators, mutating sequence operators, sorting etc, and numeric operation.

Soooo simple!
You can and will write these yourself!
Reasons to use the STL Algorithms
- speed
- correct
- clarity

Typical STL Algorithm

```
template <class ForwardItr, class T>
T accumulate (ForwardItr first, ForwardItr last, T init )
{
    while (first!=last)
        init = init + *first++;
    return init;
}
```

- Range accessed in find is [first, last)
  - round parenthesis means boundary not included

To use accumulate you need to add #include <numeric>

Using the STL Algorithm

Find the average exams score.

```
template <class ForwardItr, class T>
T accumulate (ForwardItr first, ForwardItr last, T init )
{
    while (first!=last)
        init = init + *first++;
    return init;
}
```

```
ifstream input("exam1.txt");
vector<double> exam_scores;

int score;
while ( input >> score )
    exam_scores.push_back(score);
```

```
// Compute the average
cout << accumulate(exam_scores.begin(), exam_scores.end(), 0.0)/exam_scores.size();
```

accumulate with a specified binary operation to compute the result

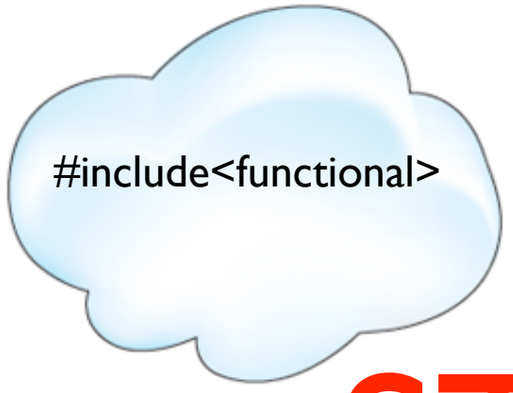
```
template <class ForwardIterator, class T, class BinaryOperation>
   T accumulate (ForwardIterator first, ForwardIterator last, T init, BinaryOperation binary_op)
{
    while (first!=last) {
        init=binary_op(init,*first);
        ++first;
     }
    return init;
}
```

To use accumulate you need to add #include <numeric>

Find the average gpa

vector<student> class_list;

```
template <class ForwardIterator, class T, class BinaryOperation>
   T accumulate (ForwardIterator first, ForwardIterator last, T init, BinaryOperation binary_op)
{
    while (first!=last) {
        init=binary_op(init,*first);
        ++first;
     }
    return init;
  }


  //create a functor!
  class add_gpa
  {
  public:
      double operator( )(double total, const student & s) { return total + s.get_gpa(); }
  };


// Compute the average
cout << accumulate(class.begin(), class.end(), 0.0, add_gpa() )/class.size();
```



STL function objects

STL Function Objects

STL function objects are *classes* that contain an operator( )

- Generator function objects don't take a parameter they return a value (e.g. *rand*, the random number generator functor.)
- Unary function objects take one parameter
- Binary function objects take two parameters

A special kind of functor is a predicate functor: function that returns a bool

- Examples of binary predicate objects in the STL

  less encapsulates operator<

  greater encapsulates operator>

  equal_to encapsulates operator==

  not_equal_to encapsulates operator!=

  greater_equal encapsulates operator>=

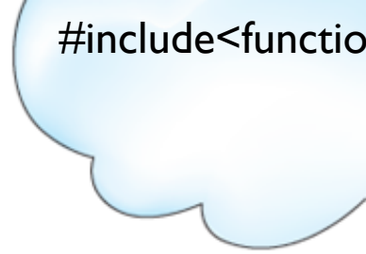
  less_equal encapsulates operator<=

#include<functional>

STL function object examples less

```
template <class Object>
class less
{ public:
    bool operator()(const Object& lhs, const Object& rhs) const
      {return lhs < rhs;}
};
```

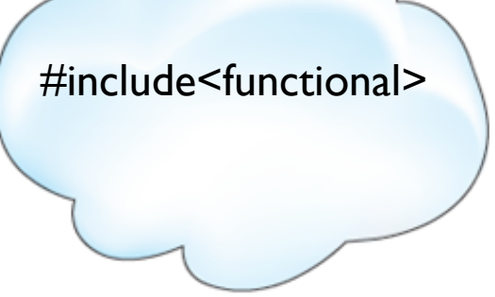
STL function object example




```
template <class Object>
class less
{ public:
  bool operator()(const Object& lhs, const Object& rhs)const
      {return lhs < rhs;}
};
```

```
// less example
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main () {
  int foo[]={10,20,5,15,25};
  int bar[]={15,10,20};
  sort (foo, foo+5, less<int>() );   // 5 10 15 20 25
  sort (bar, bar+3, less<int>() );  //   10 15 20
  return 0;
}
```

code modified from http://www.cplusplus.com/reference/functional/less/



greater_equal

```
template <class T>
class greater_equal
{
  public:
      bool operator() (const T& lhs, const T& rhs) const
        {return lhs >= rhs;}
};
```

#include<functional>

minus

```
template <class T>
class minus
{
    public:
        T operator() (const T& lhs, const T& rhs) const
            {return lhs-rhs;}
};
```

lower_bound

- lower_bound does binary search on range [first,last)
  - –container must be sorted
  - –need random access iterator for runtime O(log n)
  - –Code (Figure 7.9, p. 244) for random access iterator (STL code is slightly different)
  - –computation of middle iterator uses iterator subtraction
  - –returns iterator to leftmost element in [first,last) containing element >= x (if none exists, returns last)

STL Style Binary Search Algorithm

```
template<class RandomIterator, class Object, class Compare>
RandomIterator lower_bound( const RandomIterator begin, const RandomIterator end,
   const Object & x, const Compare lessThan)
{
     RandomIterator low=begin;
     RandomIterator mid;
     RandomIterator high = end;
    while (low < high)
        {
          mid = low + (high - low) /2;
          if(lessThan(*mid, x))
               low = mid +1;
          else
               high=mid;
         }
       return low;
}
```

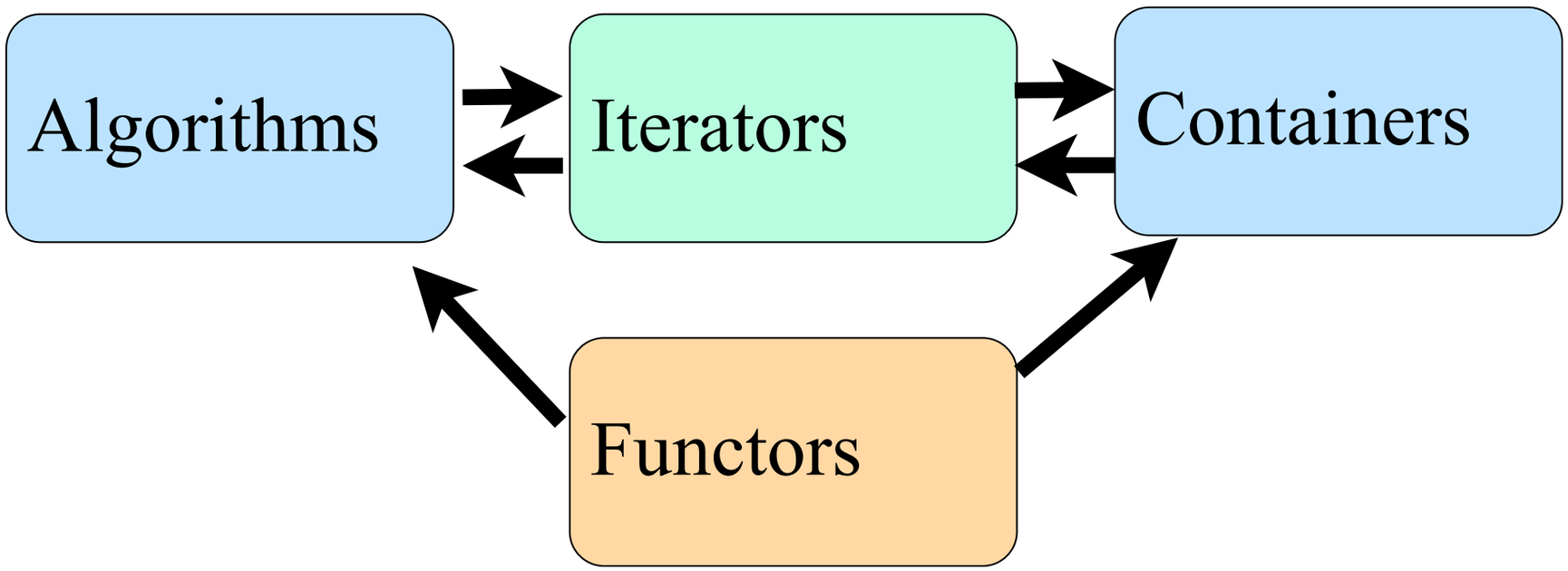
Pair of iterators define search space

```
template<class Object>
class less
{
public:
     bool operator()(const Object& x,const Object& y) const  { return (x < y); }
};
```

Running Time?

O(log(n)) where n is the number of items in [first, last)

STL
Standard Template Library



Dictionaries (ADT), SET (ADT)

- Data structures that supports **find**, **insert**, **delete**
- Many applications
- Item referred to by a **key**. In a dictionary, keys have records associated with them
- Many choices for implementing dictionary/set

  Programmer must choose best one, based on how the program will use the dictionary

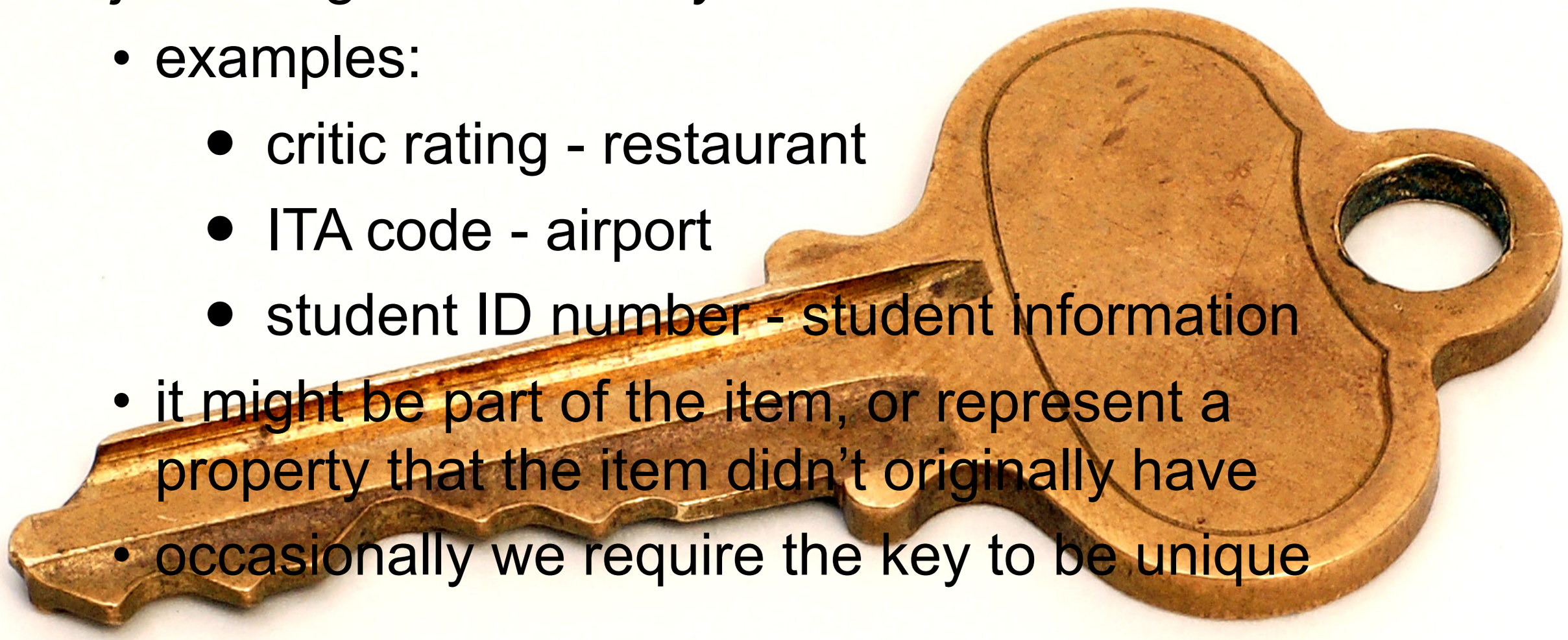
  – static versus dynamic

  – many find operations versus few find operations

keys

object assigned to *identify* an item or *rank* an item

- examples:
  - critic rating - restaurant
  - ITA code - airport
  - student ID number - student information
- it might be part of the item, or represent a property that the item didn't originally have
- occasionally we require the key to be unique

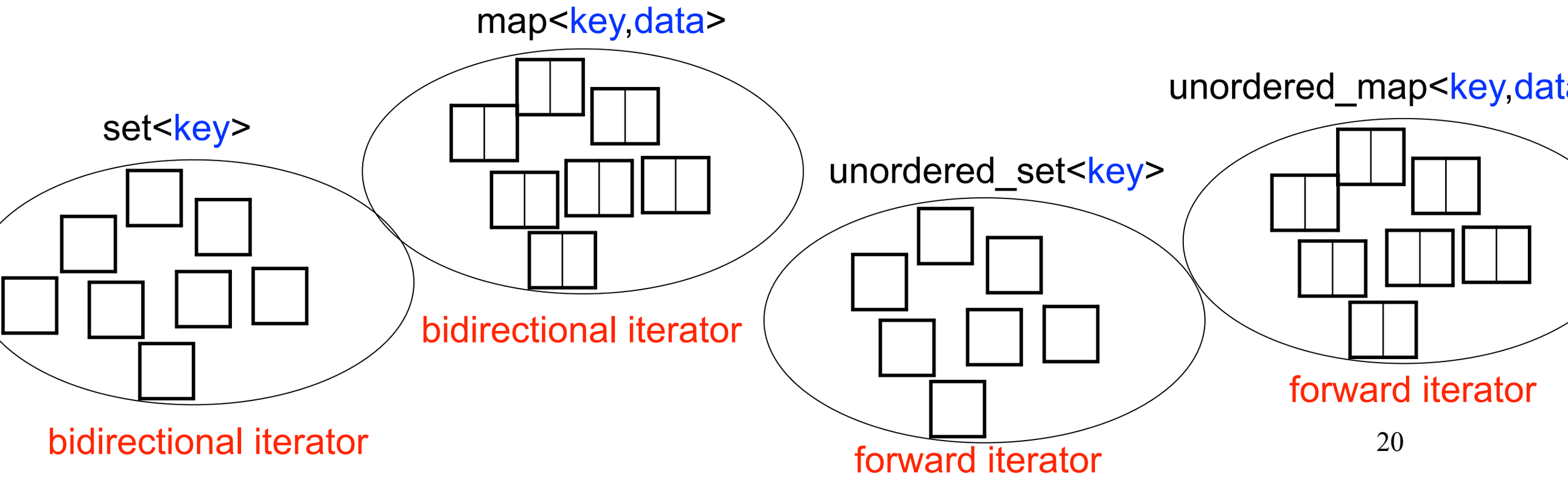


keys with ≤ have a total order: **reflexive**, **antisymmetric**, **transitive**

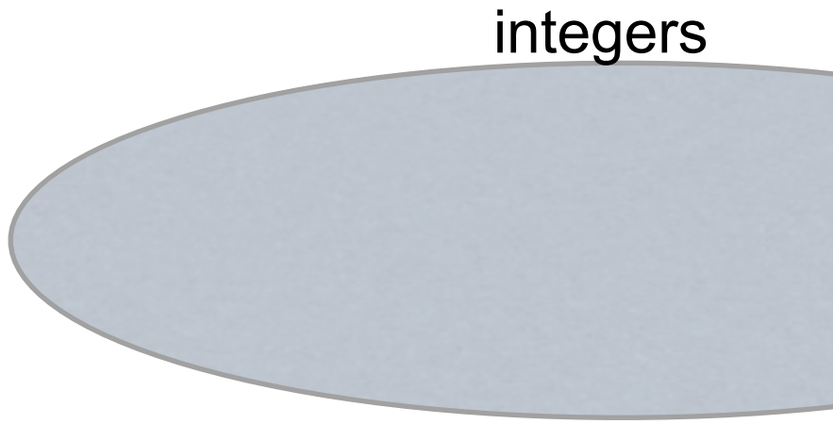
https://en.wikipedia.org/wiki/Key_(lock)#/media/File:Standard-lock-key.jpg

Ordered/Unordered Associative containers

- Can't insert element into particular position (order of insertion doesn't matter)
- Elements stored have key and (maybe) value, access by key
  - map, unordered_map: key, value pair. Efficient access by key
    - E.g. *Key* is Social Security Number, *Value* is employee data
  - set, unordered_set: Elements stored by key, but no value, access by key



Some set and unordered_set members



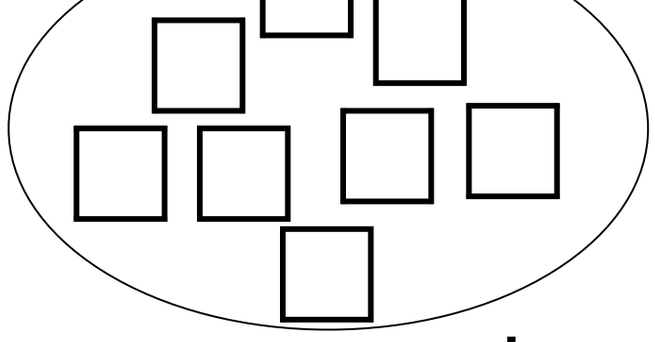
pair<iterator, bool> insert(const value_type& x)
// if x is in the set, returns false
// else inserts it and returns <iterator to x, true>

size_type erase(const key_type& k);
// removes element whose key is k and returns
// number of elements removed (0 or 1)

void erase(iterator pos);

iterator find(const key_type& k) const ;

```
unordered_set<int> setOfIntegers;
unordered_set<int>::iterator itrS;

setOfIntegers.insert(5);
setOfIntegers.insert(5);
setOfIntegers.erase(5);
setOfIntegers.insert(3);
setOfIntegers.insert(6);
setOfIntegers.insert(7);
setOfIntegers.insert(9);

itrS = setOfIntegers.find(6);
if (itrS == setOfIntegers.end())
    cout << ”6 not in set”;
else
    cout << ”6 in set”;
```



set<key, key compare>

bidirectional iterator

- Unique key (no duplicates)
- Supports insertion, deletion, and find in O(log n) time
- range [first, last) is sorted
- How's it implemented?
  - vector or linked list can't meet all the time bounds
  - Answer: balanced binary search tree – we'll study these later

unordered_set<key, key compare>

forward iterator

- Unique key (no duplicates)
- Supports insertion, deletion, and find in O(1) time on average
- range [first, last) is unsorted
- How's it implemented?
  - vector or linked list can't meet all the time bounds
  - Answer: …

Pair

```
template<class Type1, class Type2>
struct pair
{
public:
    Type1 first;
    Type2 second;
    pair (const Type1 & f=Type1(),const Type2 & s = Type2())
    : first(f), second( s){}
};
```

```
pair<int, int> mypair1;
mypair1.first = 3;
mypair1.second= 9;
cout << mypair1.first<< mypair1.second;
```

```
pair<int, string> mypair2;
mypair2.first = 3;
mypair2.second = “shoes”;
cout << mypair2.first << “ “ << mypair2.second;
```

```
pair<string, list<string> > packingList;
packingList.first = “Grand Canyon”;
packingList.second.push_front(“shoes”);
packingList.second.push_front(“sun screen”);
```

For some compilers
>> is always an operator
so you must write
pair<string, list<string> >
not
pair<string, list<string>>

Pair

```
template<class Type1, class Type2>
struct pair
{
public:
    Type1 first;
    Type2 second;
    pair (const Type1 & f=Type1(),const Type2 & s = Type2())
    : first(f), second( s){}
};
```

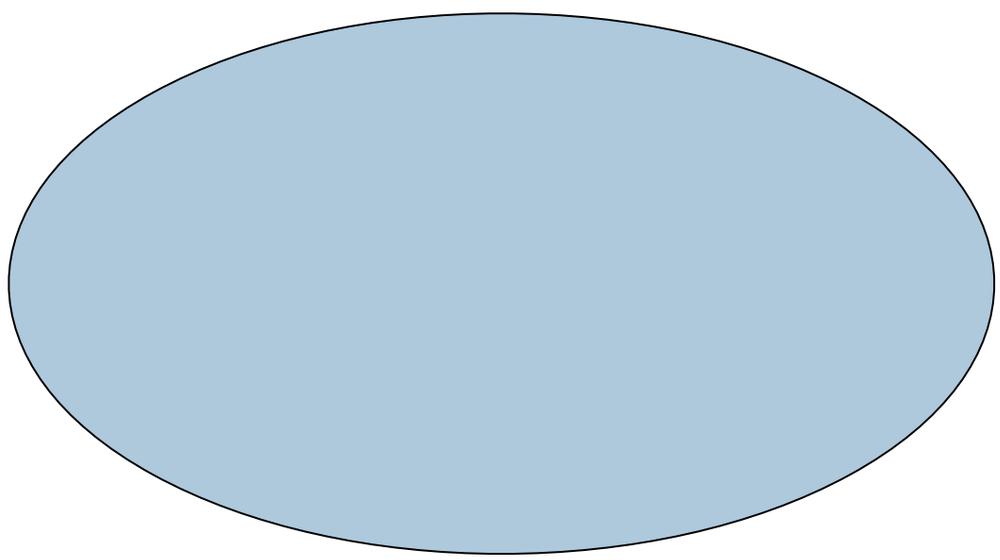
```
pair<string, string> airportCode1;
airportCode1.first = “ABR”;
airportCode1.second= “Aberdeen, SD”;
cout << airportCode1.first<< airportCode1.second;

pair<string, string> airportCode2;
airportCode2.first = “ABI”;
airportCode2.second= “Abilene, TX”;
cout << airportCode2.first<< airportCode2.second;
```

Aberdeen, SD (ABR)
Abilene, TX (ABI)
Adak Island, AK (ADK)
Akiachak, AK (KKI)
Akiak, AK (AKI)
Akron/Canton, OH (CAK)
Akuton, AK (KQA)
Alakanuk, AK (AUK)
Alamogordo, NM (ALM)
Alamosa, CO (ALS)
Albany, NY (ALB)
Albany, OR - Bus service (CVO)
Albany, OR - Bus service (QWY)
Albuquerque, NM (ABQ)
Aleknagik, AK (WKK)
Alexandria, LA (AEX)
Allakaket, AK (AET)
Allentown, PA (ABE)
Alliance, NE (AIA)
Alpena, MI (APN)
Altoona, PA (AOO)
Amarillo, TX (AMA)
Ambler, AK (ABL)
Anaktueuk, AK (AKP)
Anchorage, AK (ANC)
Angoon, AK (AGN)
Aniak, AK (ANI)
Anvik, AK (ANV)
Appleton, WI (ATW)
Arcata, CA (ACV)

Important map/Unordered_map Member functions

- pair<iterator, bool> insert(const value_type& x)
  Inserts x into the map.
  - Won't insert if there's already an element with that key in the map.
  - Return value.second indicates whether insertion was successful
- iterator find(const key_type& k)
  Finds an element whose key is k.
  - Returns end( ) if not found
  - Caller should check whether returned iterator is valid
- void erase(iterator pos)
  Erases the element pointed to by pos.
- size_type erase(const key_type& k)
  Erases the element whose key is k.

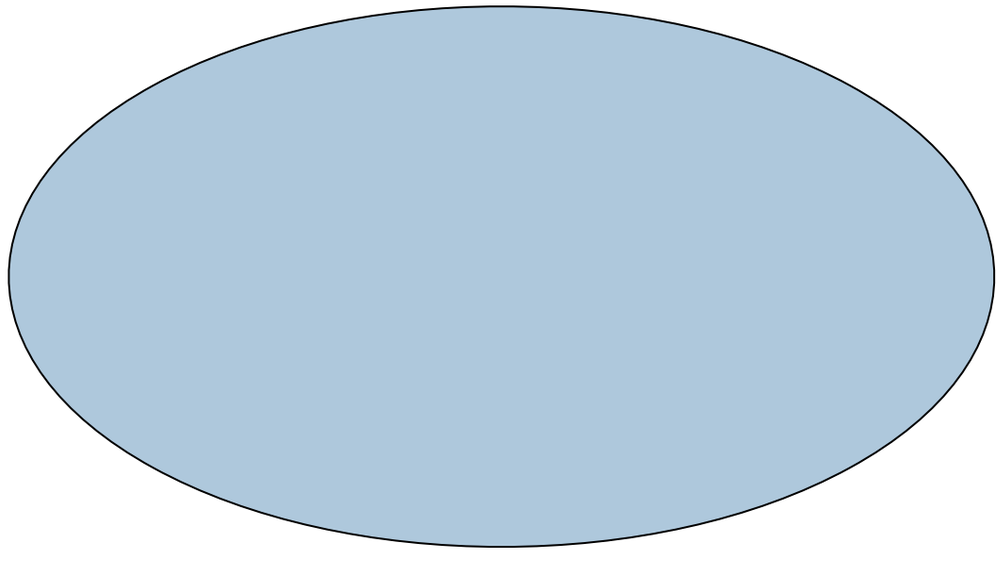
<string,string>



```
pair<string, string> airportCode1;
airportCode1.first = "ABR";
airportCode1.second= "Aberdeen, SD";

map<string,string> mymap;
map<string,string>::iterator mItr;

mymap.insert(airportCode1);
mymap.insert(pair<string, string>("ADK", "Adak Island, AK"))
mymap["JFK"] = "New York, NY - Kennedy";

mItr = mymap.find("ABR");
if ( mItr == mymap.end( ))
    cout << "ABR is not in the map";
else
    mymap.erase( mItr );
```

```
pair<string, string> airportCode1;
airportCode1.first = “ABR”;
airportCode1.second= “Aberdeen, SD”;


unordered_map<string,string> mymap;
unordered_map<string,string>::iterator mItr;

mymap.insert(airportCode1);
mymap.insert(pair<string, string>("ADK", ”Adak Island, AK”));
mymap[“JFK”] = “New York, NY - Kennedy”;

mItr = mymap.find(”ABR");
if ( mItr == mymap.end( ))
    cout << ”ABR is not in the map”;
else
    mymap.erase( mItr );
```

<string,string>



map::operator[ ]
unordered_map::operator[ ]

- data_type& operator[ ](const key_type& k)
  Returns a reference to the object that is associated with a particular key. If the map does not already contain such an object, operator[ ] inserts the default object data_type().
  - m[k] is equivalent to the "simple" ☺ (according to STL docs) expression

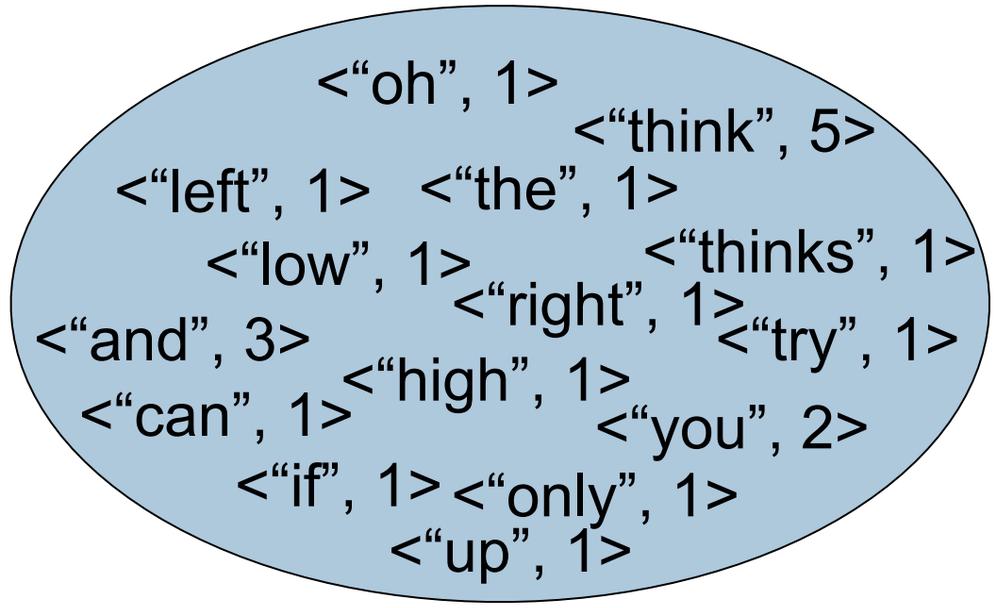
    (*((m.insert(value_type(k, data_type()))).first)).second

  - Notation suggests array indexed by key values (but that's not how it's implemented)
  - If side effect of adding new object when key is not found is not wanted, instead use:

```
it = m.find(k);
if (it != m.end())
  { // access or update it->second};
else
  {//handle case where k is not found}
```

  - Similar situation if update of data for an existing key is not wanted

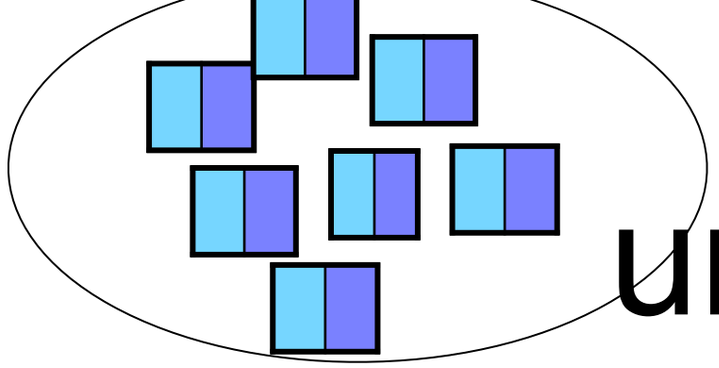
Quote by Dr. Seuss: “think left and think right and think low and think high oh the thinks you can think up if only you try”




```
// Word frequencies -- using map
// Fred Swartz 2001-12-11
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    string word;
    map<string, int> freq;
    // map of words and their frequencies

    // input buffer for words.
    //--- Read words/tokens from input stream
    while (cin >> word)
        { freq[word]++; }
    //--- Write the count and the word.
    map<string, int>::const_iterator iter;
    for (iter = freq.begin(); iter ! = freq.end(); ++iter)
        { cout << iter->second << " " << iter->first << endl; }
    return 0;
}
```

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

int main()
{
    unordered_map<string, int> freq;
    string word;
    // map of words and their frequencies

    // input buffer for words.
    //--- Read words/tokens from input stream
    while (cin >> word)
        { freq[word]++; }
    //--- Write the count and the word.
    unordered_map<string, int>::const_iterator iter;
    for (iter = freq.begin(); iter ! = freq.end(); ++iter)
        { cout << iter->second << " " << iter->first << endl; }
    return 0;
}
```

map



unordered_map

map<key,value, key_compare>

bidirectional iterator

- Unique keys (no duplicates)
- Supports insertion, deletion, and find in O(log n) time
- range [first, last) is sorted by key
- How's it implemented?
  - vector or linked list can't meet all the time bounds
  - Answer: balanced binary search tree – we'll study these later

unordered_map<key,value, key_compare>

forward iterator

- Unique keys (no duplicates)
- Supports insertion, deletion, and find in O(1) time on average
- range [first, last) is unsorted
- How's it implemented?
  - vector or linked list can't meet all the time bounds
  - Answer: …

data structure	build	insert	find
vector	O(n)	O(1) Into the back	O(n)
sorted vector	O(n log n)	O(n)	O(log n)
set or map	O(n log n)	O(log n)	O(log n)
unordered_set unordered_map	O(n) ave. $O(n^2)$ worst	O(1) ave O(n) worst	O(1) ave O(n) worst

Additional Information

Simple to adapt to a new search criteria!

```
template<class InputIterator, class UnaryPredicate>
  InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
{
  while (first!=last) {
    if (pred(*first)) return first;
    ++first;
  }
  return last;
}
```

```
class gpa_between
{
public:
   gpa_between(double l, double u):lower(l),upper(u){};
   bool operator()(student& record){return ((lower<= record.get_gpa()) &&
(record.get_gpa() <= upper)); }
private:
   double lower;
   double upper;
};
```

```
int main ()
{
  vector<student> classList;
  vector<student>::iterator itr;
  //some code to fill the vector, etc
  itr = find_if(classList.begin(), classList.end(), gpa_between(3.0,4.0));
  cout << endl<< (*itr).get_name()<< endl;
```

find_if

```
template<class InputIterator, class UnaryPredicate>
  InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
{
  while (first!=last) {
    if (pred(*first)) return first;
    ++first;
  }
  return last;
}
```

```
class gpaIs
  {
  public:
     gpaIs(const double value):value(value){}
     bool operator()(student& rhs){return ( value == rhs.get_gpa() );}
  private:
     double value;
  };
```

classList

George	Thomas	Adam	William	Abagail		
2.2	3.3	2.3	3.8	4		

```
int main ()
{
 vector<student> classList;
 vector<student>::iterator itr;
 gpaIs gpaIs3p3(3.3);
 //some code to fill the vector, etc
 itr = find_if(classList.begin(), classList.end() gpaIs3p3);
```

for_each

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function fn)
{
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;
}
```

Code using a STL algorithm for_each and a non-STL functor

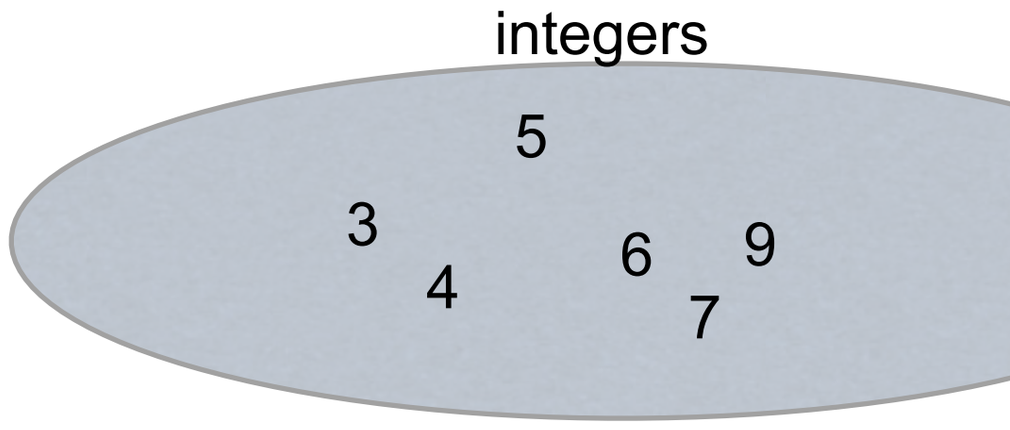
Conversion operator is a member function. It cannot modify the member variables. Note that the syntax is odd. It has no return type:
operator type()const;

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function fn)
{
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;
}

    class Sum {
        int val;
  public:
        Sum(int i) :val(i) { }
        operator int() const { return val; }            // extract value

        int operator()(int i) { return val+=i; }   // application
  };

void f(vector<int> v)
{
      Sum s = 0;     // initial value 0
      s = for_each(v.begin(), v.end(), s);    // gather the sum of all elements
      cout << "the sum is " << s << "\n";

      // or even:
      cout << "the sum is " << for_each(v.begin(), v.end(), Sum(0)) << "\n";
}
```

functor
Capable of maintaining a state. The state can be examined from the outside (static variables cannot be examined from the outside.)

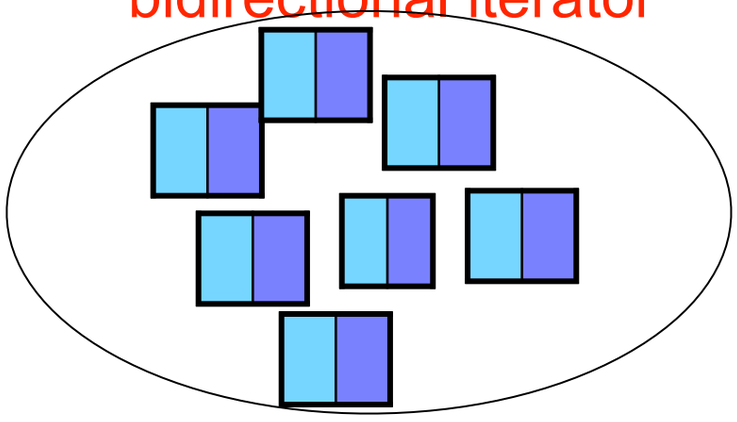
Modified code from http://www.stroustrup.com/bs_faq2.html#fns

```
#include<unordered_set>

unordered_set<int> setOfIntegers;
unordered_set<int>::iterator itrS;
          for(int i=0; i<10; ++i)
            setOfIntegers.insert(rand()%10);

          cout << setOfIntegers.size() << "items inserted into the set" << endl;
          for(itrS=setOfIntegers.begin(); itrS!=setOfIntegers.end(); ++itrS)
             cout << *itrS << " ";
          cout << endl;
```

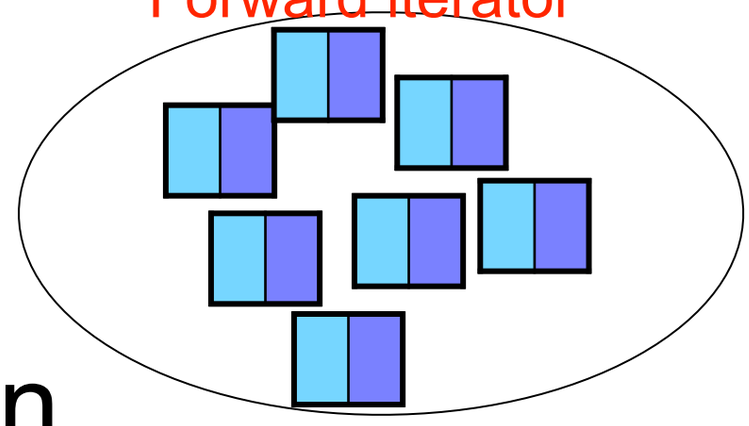



```
itrS= setOfIntegers.find(3);// if 3 is found returns an iterator to 3
if (itrS != setOfIntegers.end()) // if 3 isn't found returns an iterator
{                                // to end( )
  setOfIntegers.erase(3);
  for(itrS=setOfIntegers.begin(); itrS!=setOfIntegers.end(); ++itrS)
   cout << *itrS << " ";
  cout << endl;
}

 setOfIntegers.erase(13);//returns 0 since 13 did not exist,
```

map<key,data>
bidirectional iterator



unordered_map<key,data>
Forward iterator



Some Types used in

map< key_type, data_type, key_compare>
unordered_map< key_type, data_type, key_compare>

- key_type : The map's key type (Key). Cannot be changed
- data_type : The type of object associated with the keys (Data). Can be changed
- value_type : The type of object,

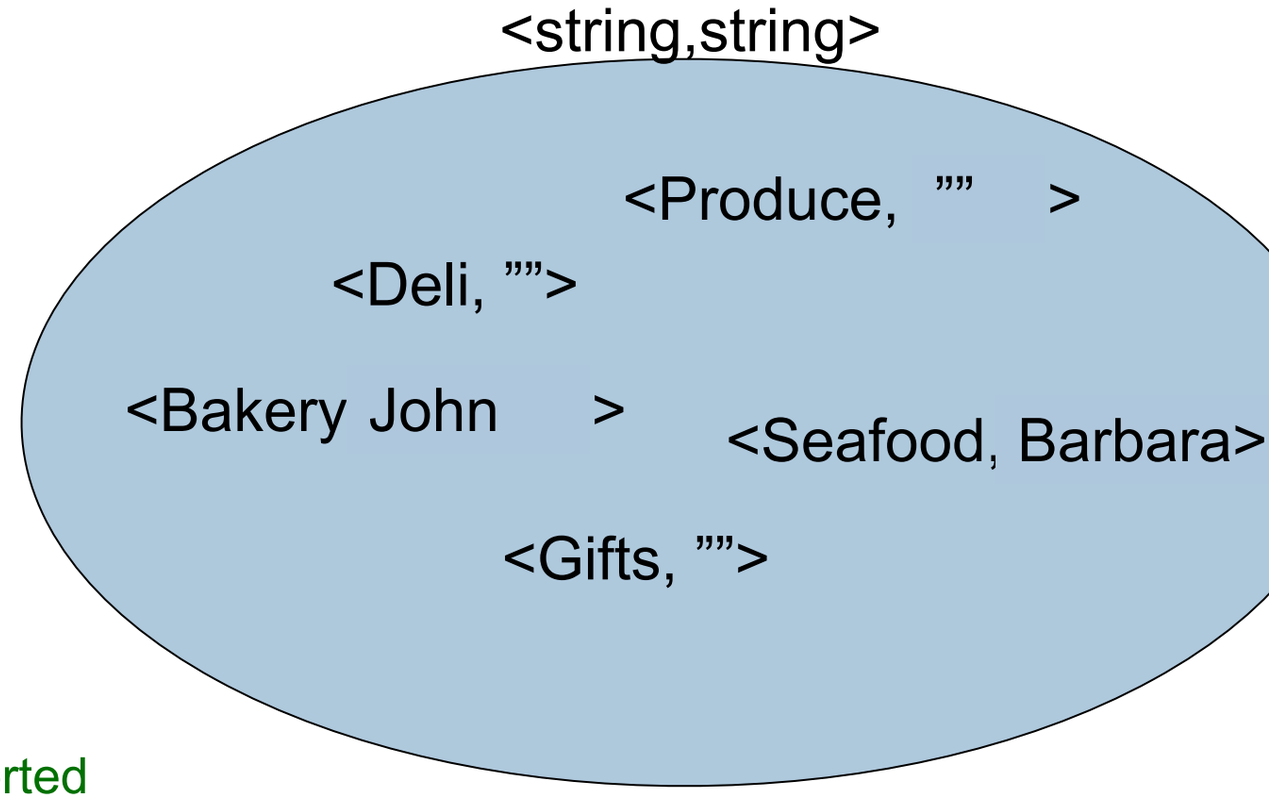
  pair<const key_type, data_type>, stored in the map.
- key_compare : function object that compares two keys for ordering (Compare)
- const and non-const iterators
  - *it is not mutable, but it->second is mutable

unordered_map example

```
// unordered_map::insert
#include <iostream>
#include <string>
#include <unordered_map>

int main ()
{
  std::unordered_map<std::string,double>
               myrecipe,
               mypantry = {{"milk",2.0},{"flour",1.5}};

std::pair<std::string,double> myshopping ("baking powder",0.3);

myrecipe.insert (myshopping);                        // copy insertion
myrecipe.insert (std::make_pair<std::string,double>("eggs",6.0)); // move insertion
myrecipe.insert (mypantry.begin(), mypantry.end());  // range insertion
myrecipe.insert ( {{"sugar",0.8},{"salt",0.1}} );    // initializer list insertion
```

From http://www.cplusplus.com/reference/unordered_map/unordered_map/insert/

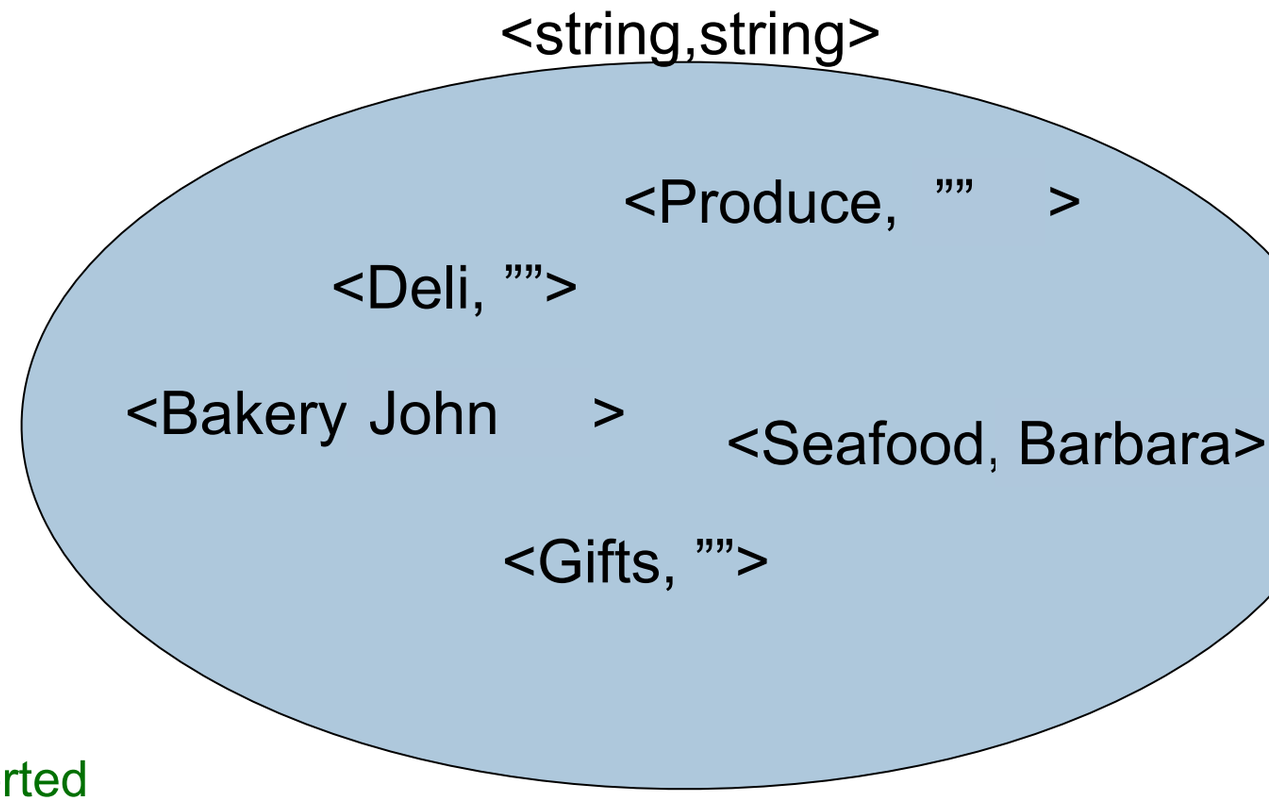
map example




```
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main ()
{
  map<string,string> mymap;

  mymap["Bakery"]="Barbara";  // new element inserted
  mymap["Seafood"]="Lisa";    // new element inserted
  mymap["Produce"]="John";    // new element inserted

  string name = mymap["Bakery"];   // existing element accessed (read)
  mymap["Seafood"] = name;              // existing element accessed (written)

  mymap["Bakery"] = mymap["Produce"];   // existing elements accessed (read/written)

  name = mymap["Deli"];      // non-existing element: new element "Deli" inserted!

  mymap["Produce"] = mymap["Gifts"];    // new element "Gifts" inserted, "Produce" written
}
```

Modified from http://www.cplusplus.com/reference/unordered_map/unordered_map/operator[]/

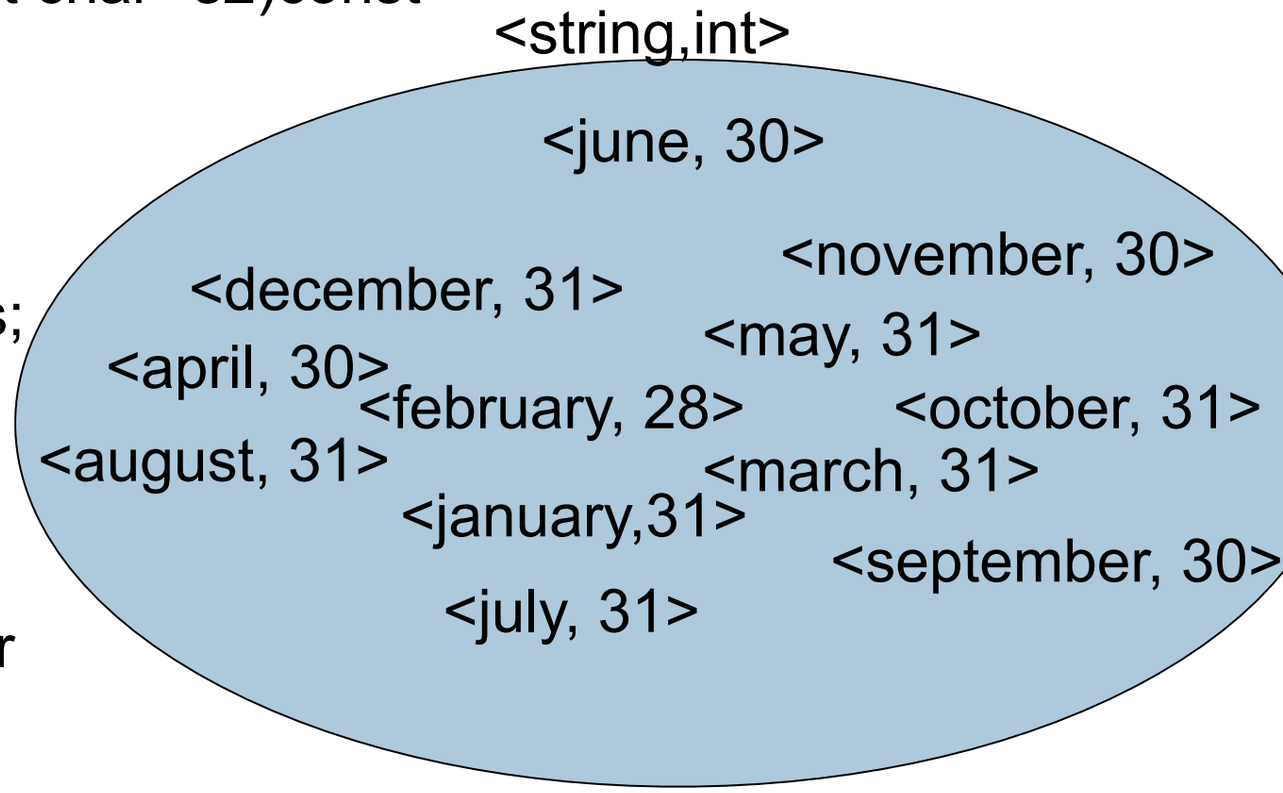
unordered_map example




```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

int main ()
{
  unordered_map<string,string> mymap;

  mymap["Bakery"]="Barbara";  // new element inserted
  mymap["Seafood"]="Lisa";    // new element inserted
  mymap["Produce"]="John";    // new element inserted

  string name = mymap["Bakery"];   // existing element accessed (read)
  mymap["Seafood"] = name;              // existing element accessed (written)

  mymap["Bakery"] = mymap["Produce"];   // existing elements accessed (read/written)

  name = mymap["Deli"];      // non-existing element: new element "Deli" inserted!

  mymap["Produce"] = mymap["Gifts"];    // new element "Gifts" inserted, "Produce" written
}
```

Modified from http://www.cplusplus.com/reference/unordered_map/unordered_map/operator[]/

```
// Example from SGI STL documentation
struct ltstr{
 bool operator()(const char* s1,const char* s2)const
   {    return strcmp(s1, s2) < 0;  }
 };
int main()  {
unordered_map<const char*, int, ltstr> months;
  months["january"] = 31;
  months["february"] = 28;
  . . .
unordered_map<const char*, int, ltstr>::iterator
     cur  = months.find("june");
unordered_map<const char*, int, ltstr>::iterator prev = cur;
unordered_map<const char*, int, ltstr>::iterator next = cur;
  ++next;
  --prev;
  cout << "Previous (in alphabetical order) is " <<    (*prev).first << endl;
  cout << "Next (in  alphabetical order) is " << (*next).first << endl;
}
```



#include <set>

set

Bidirectional Iterator

s.find(key)	O(log(n))
s.lower_bound(key)	O(log(n))
s.upper_bound(key)	O(log(n))
s.size()	O(1)
s.empty()	O(1)
s.insert(k)	O(log(n))
s.begin()	O(1)
s.end()	O(1)
s.erase(iterator) & s.erase(key)	O(1) amortized & O(log(n))
s.clear()	O(n)

#include <unordered_set>

unordered_set

Forward Iterator

	average case	worst case
s.find(key)	O(1)	O(n)
s.size()	O(1)	O(1)
s.empty()	O(1)	O(1)
s.insert(k)	O(1)	O(n)
s.begin()	O(1)	O(1)
s.end()	O(1)	O(1)
s.erase(iterator) & s.erase(key)	O(1), & O(1)	O(n), &O(n)
s.clear()	O(n)	O(n)

Note: all these times do not include constructor/destructor times which many vary according to the type

#include<map>

Great to use when you need to access the elements by key.

This list is not complete. Check expert-level resource for more info.

#include<unordered_map>

map - Bidirectional Iterator

- m.insert(pair) O(log(n))
- m.find(key) O(log(n))
- m.size() O(1)
- m.begin() O(1)
- m.end() O(1)
- m.lower_bound(key) O(log(n))
- m.upper_bound(key) O(log(n))
- m[key] O(log(n))
- m.clear() O(n)
- m.erase(key) & m.erase(iterator)
  O(log(n)) & O(1) amortized

unordered_map - Forward Iterator

- u.insert(pair) O(1) ave, 0(n) worst case
- u.find(key) O(1) ave, 0(n) worst case
- u.size() O(1)
- u.begin() O(1)
- u.end() O(1)
- m[key] O(1) ave, 0(n) worst case
- m.clear() O(n)
- m.erase(key) & m.erase(iterator)