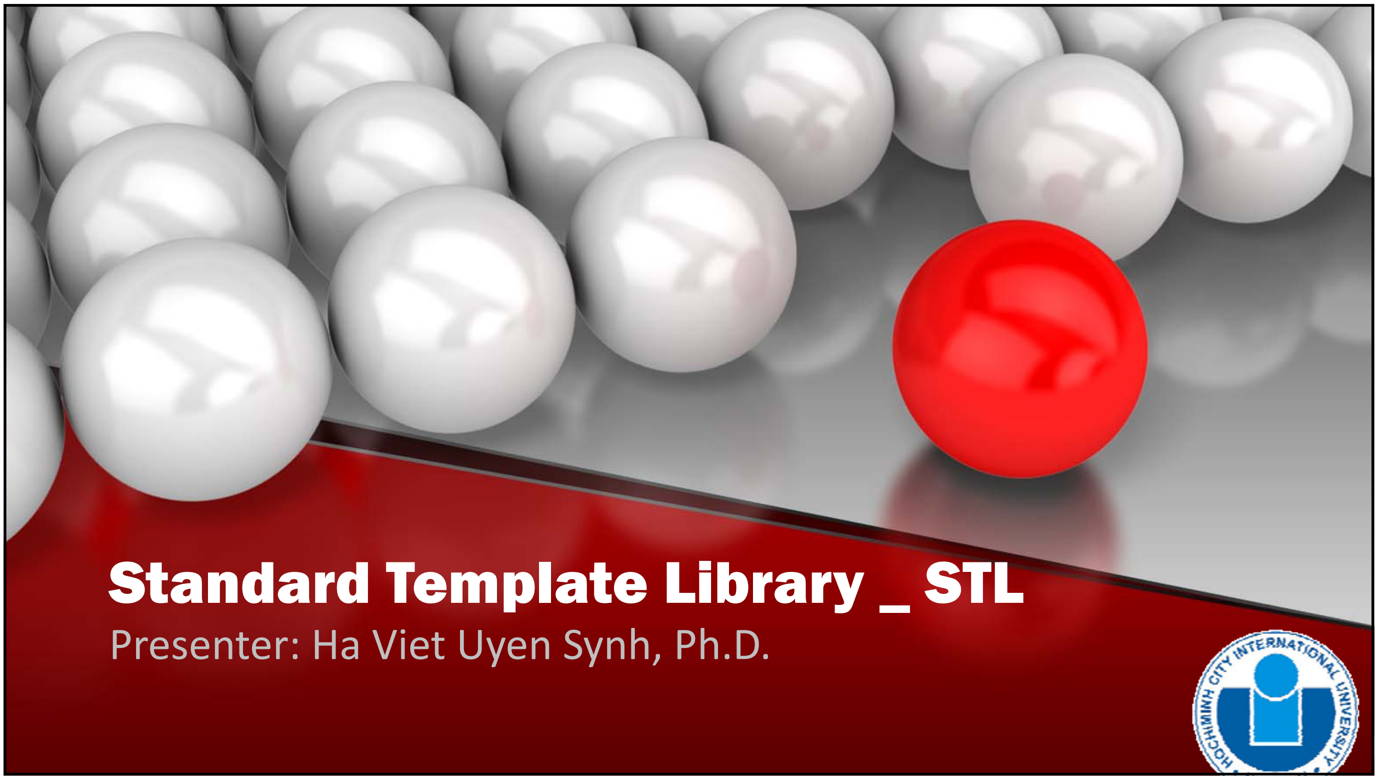
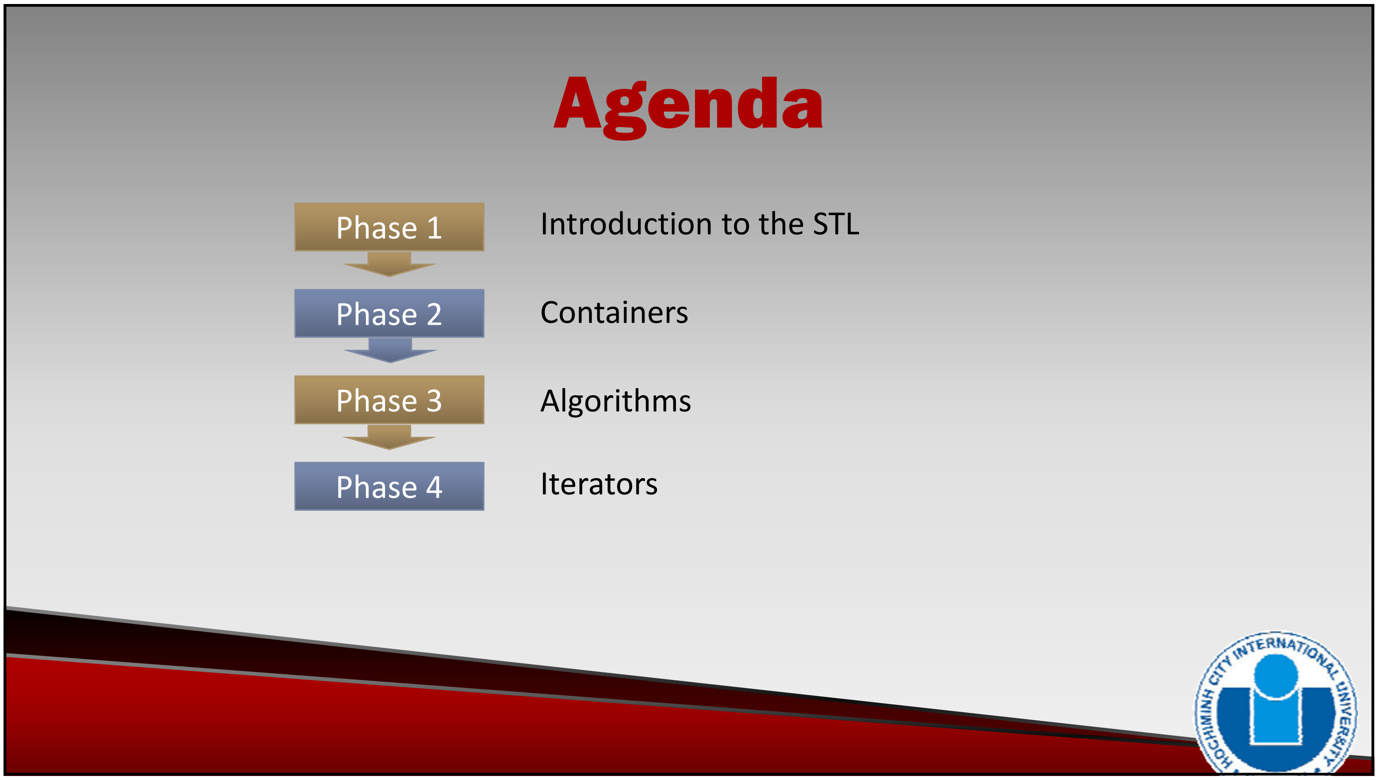


Standard Template Library _ STL

Presenter: Ha Viet Uyen Synh, Ph.D.

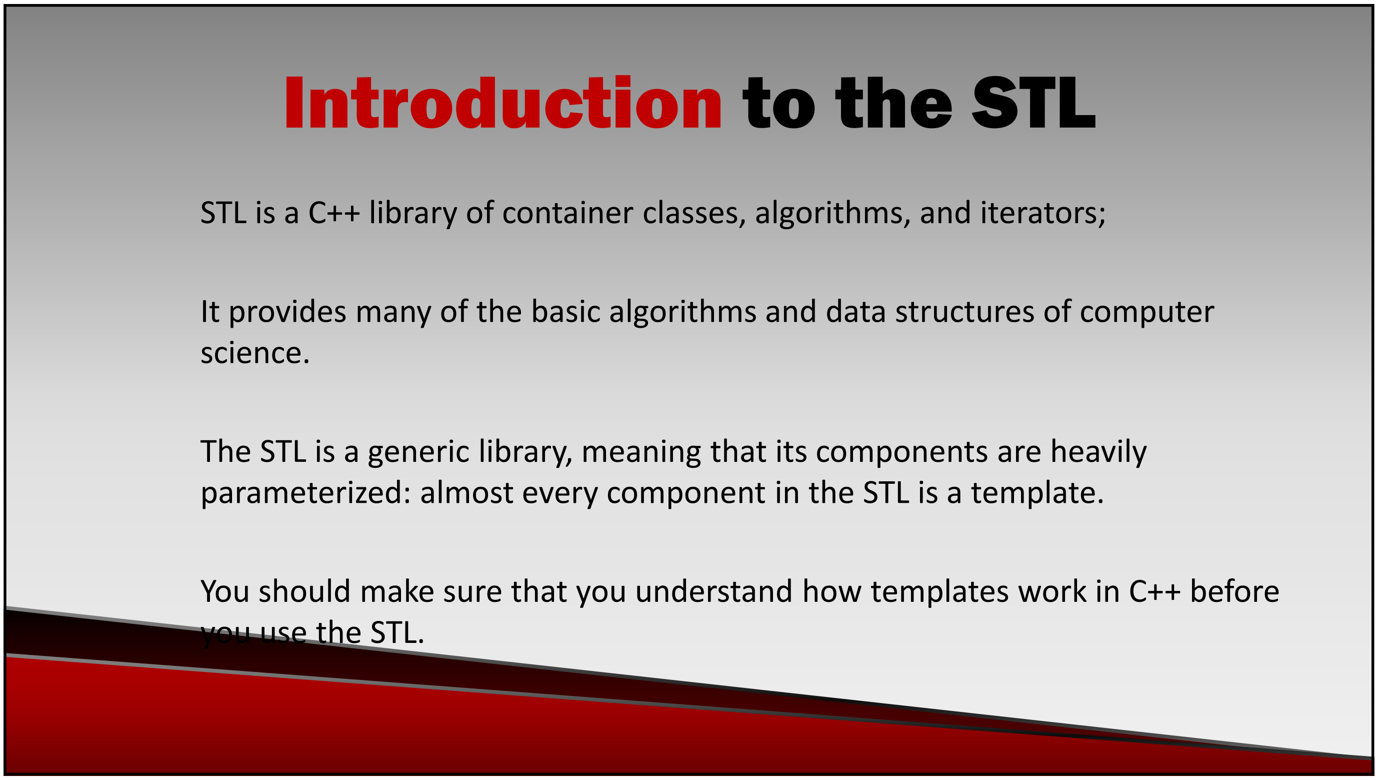


Agenda

Phase 1	Introduction to the STL
Phase 2	Containers
Phase 3	Algorithms
Phase 4	Iterators

Part I

INTRODUCTION TO THE STL



Introduction to the STL

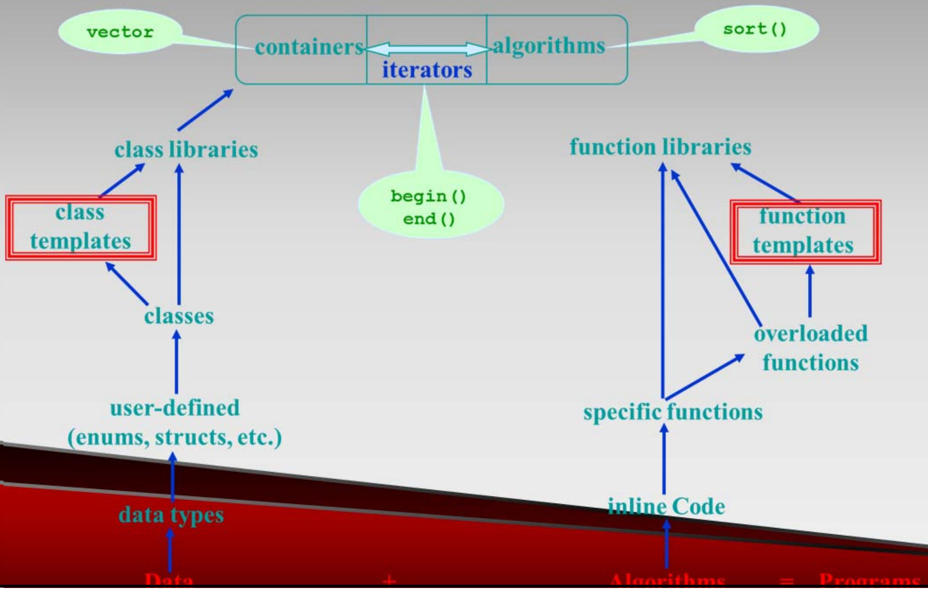
STL is a C++ library of container classes, algorithms, and iterators;

It provides many of the basic algorithms and data structures of computer science.

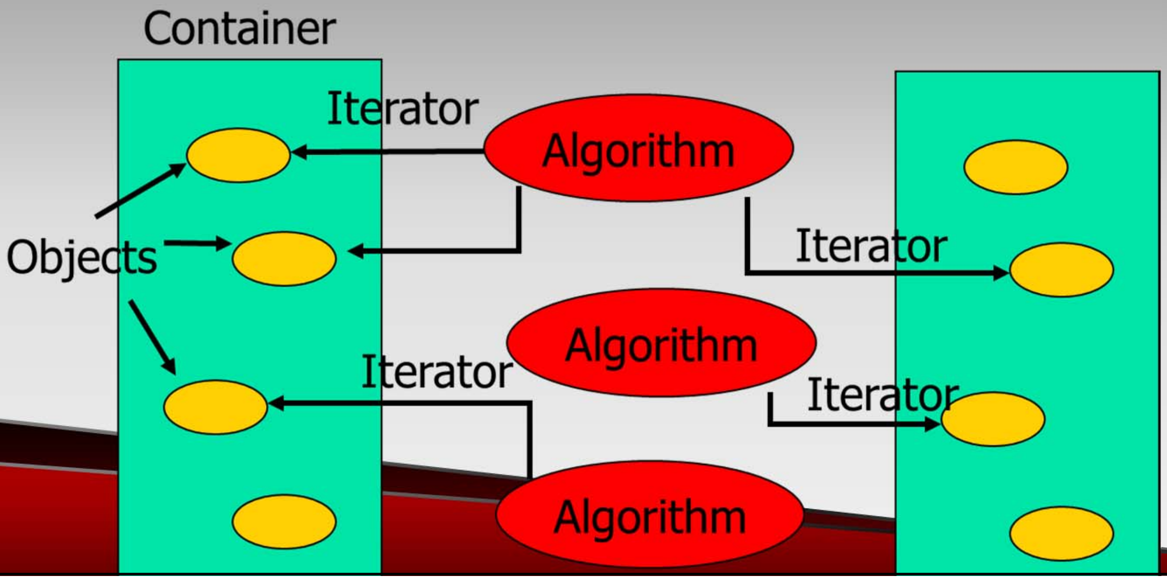
The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template.

You should make sure that you understand how templates work in C++ before you use the STL.

Standard Template Library (STL)

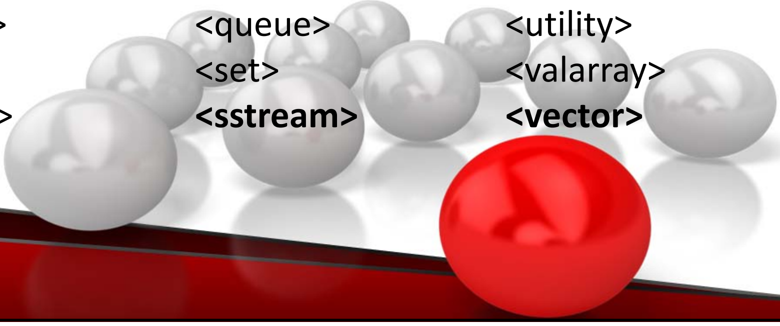


Containers, Iterators, Algorithms



Standard Template Library (STL)

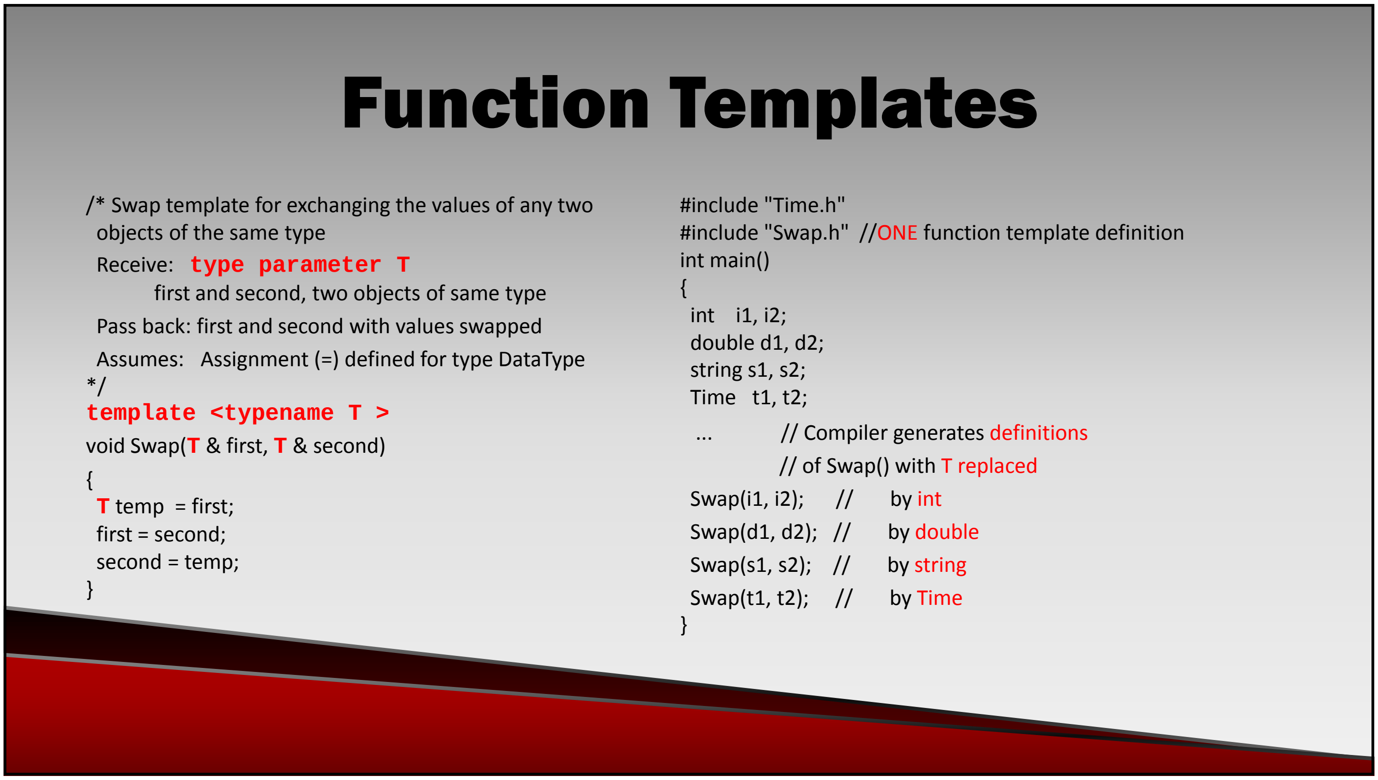
<algorithm>
<bitset>
<complex>
<deque>
<exception>
<fstream>
<functional>
<iomanip>
<ios>
<iosfwd>
<iostream>
<istream>
<iterator>
<limits>
<list>
<locale>
<map>
<memory>
<new>
<numeric>
<ostream>
<queue>
<set>
<sstream>
<stack>
<stdexcept>
<streambuf>
<string>
<typeinfo>
<utility>
<valarray>
<vector>



Class Templates

```
template < class T >
class Value {
                    T _value;
public:
                     Value ( T value ) { _value = value; }
                    T getValue ();
                    void setValue ( T value );
};

template < class T >
T Value<T>::getValue () { return _value; }

template < class T >
void Value<T>::setValue ( T value ) { _value = value; }

Value<float> values[10]; // array of values of type float
```

```
Template < class T >
class ValueList {
Value<T> * _nodes;
public:
                    ValueList ( int noElements )
                     {
                            _nodes = new Node<T>[noElements];
                    }

                    virtual ~ValueList ()
                    {
                            delete [] _nodes;
                    }
};
```



Function Templates

```
/* Swap template for exchanging the values of any two
  objects of the same type
  Receive:  type parameter T
         first and second, two objects of same type
  Pass back: first and second with values swapped
  Assumes:  Assignment (=) defined for type DataType
*/
template <typename T >
void Swap(T & first, T & second)
{
  T temp  = first;
  first = second;
  second = temp;
}
```

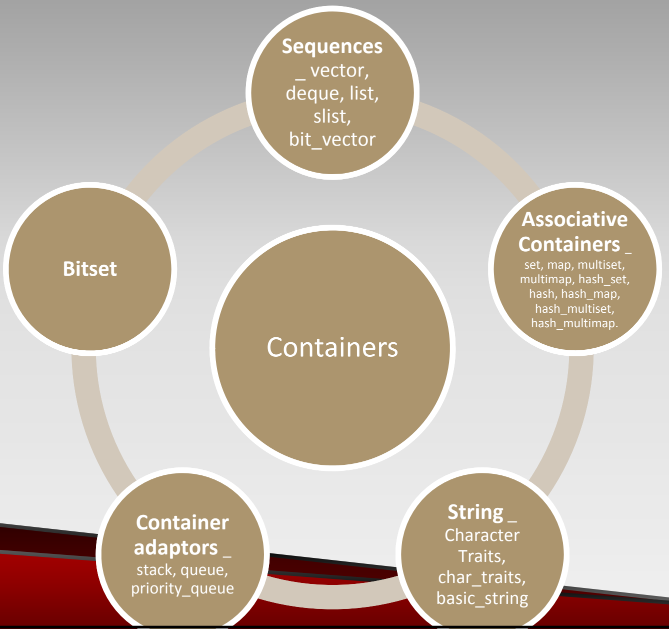
```
#include "Time.h"
#include "Swap.h"  //ONE function template definition
int main()
{
  int   i1, i2;
  double d1, d2;
  string s1, s2;
  Time   t1, t2;
   ...          // Compiler generates definitions
                // of Swap() with T replaced
  Swap(i1, i2);     //      by int
  Swap(d1, d2);  //      by double
  Swap(s1, s2);   //      by string
  Swap(t1, t2);    //      by Time
}
```

Part II

CONTAINERS

Containers

A *container* is a way that stored data is organized in memory, for example an array of elements.


Sequences _ vector, deque, list, slist, bit_vector
Associative Containers _ set, map, multiset, multimap, hash_set, hash, hash_map, hash_multiset, hash_multimap.
Containers
Bitset
Container adaptors _ stack, queue, priority_queue
String _ Character Traits, char_traits, basic_string

Sequence Containers

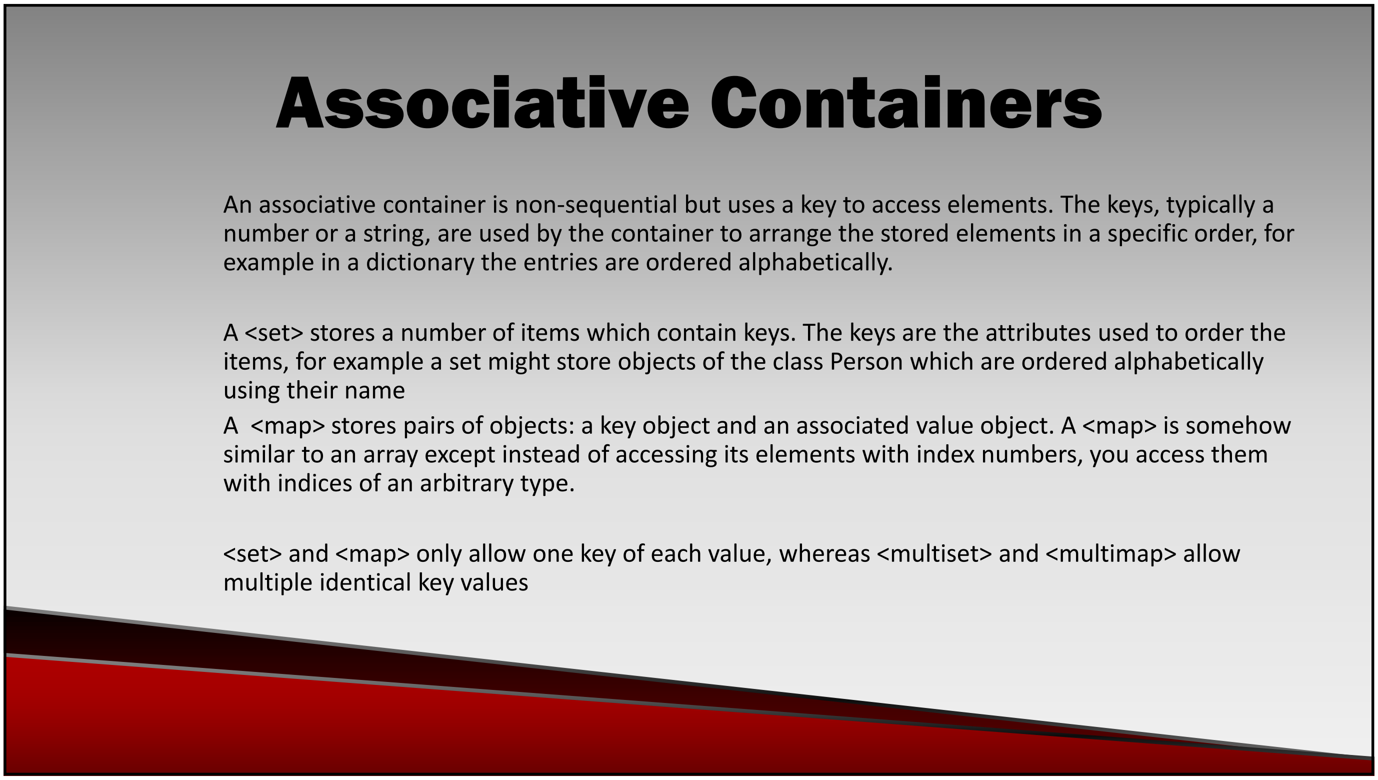
A sequence container stores a set of elements in sequence, in other words each element (except for the first and last one) is preceded by one specific element and followed by another, <vector>, <list> and <deque> are sequential containers.

In an ordinary C++ array the size is fixed and can not change during run-time, it is also tedious to insert or delete elements. Advantage: quick random access

<vector> is an expandable array that can shrink or grow in size, but still has the disadvantage of inserting or deleting elements in the middle

<list> is a double linked list (each element has points to its successor and predecessor), it is quick to insert or delete elements but has slow random access

<deque> is a double-ended queue, that means one can insert and delete elements from both ends, it is a kind of combination between a stack (last in first out) and a queue (first in first out) and constitutes a compromise between a <vector> and a <list>



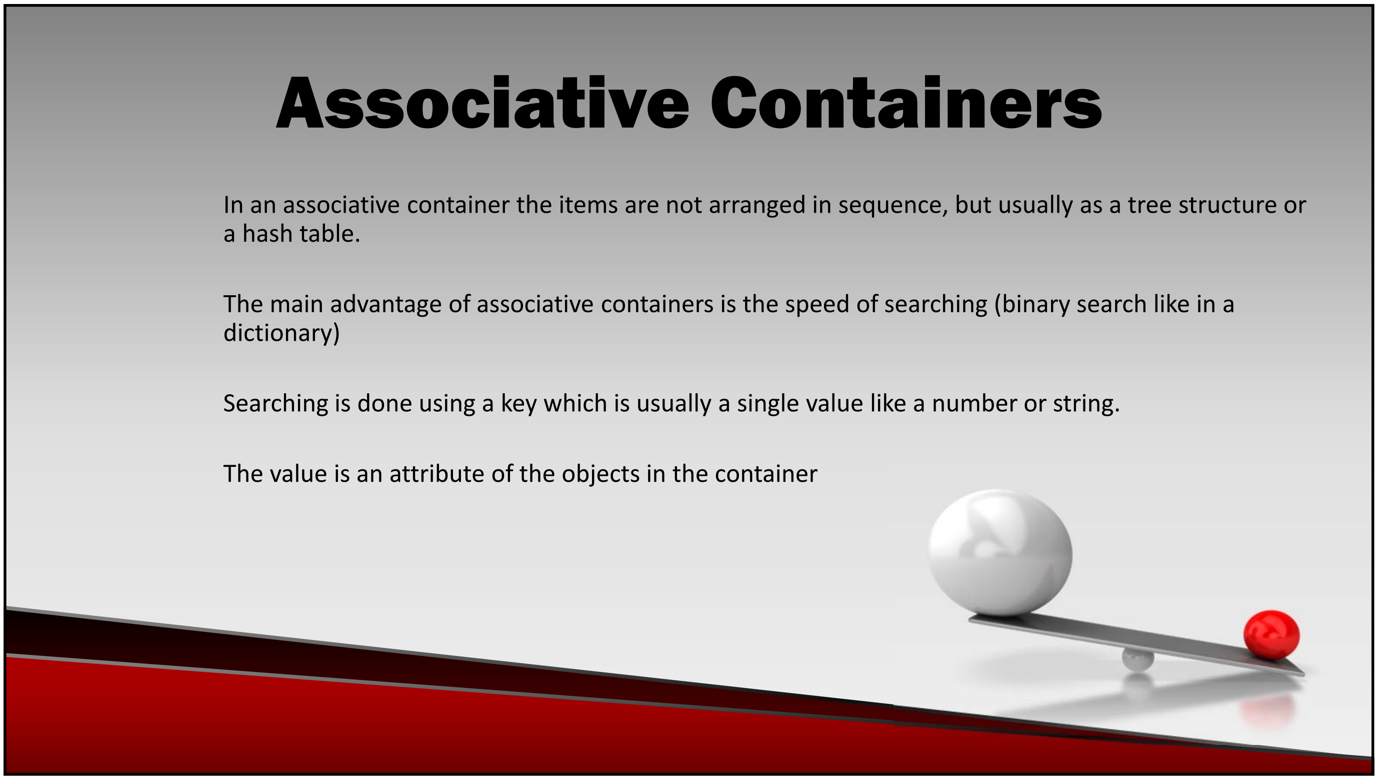
Associative Containers

An associative container is non-sequential but uses a key to access elements. The keys, typically a number or a string, are used by the container to arrange the stored elements in a specific order, for example in a dictionary the entries are ordered alphabetically.

A <set> stores a number of items which contain keys. The keys are the attributes used to order the items, for example a set might store objects of the class Person which are ordered alphabetically using their name

A <map> stores pairs of objects: a key object and an associated value object. A <map> is somehow similar to an array except instead of accessing its elements with index numbers, you access them with indices of an arbitrary type.

<set> and <map> only allow one key of each value, whereas <multiset> and <multimap> allow multiple identical key values



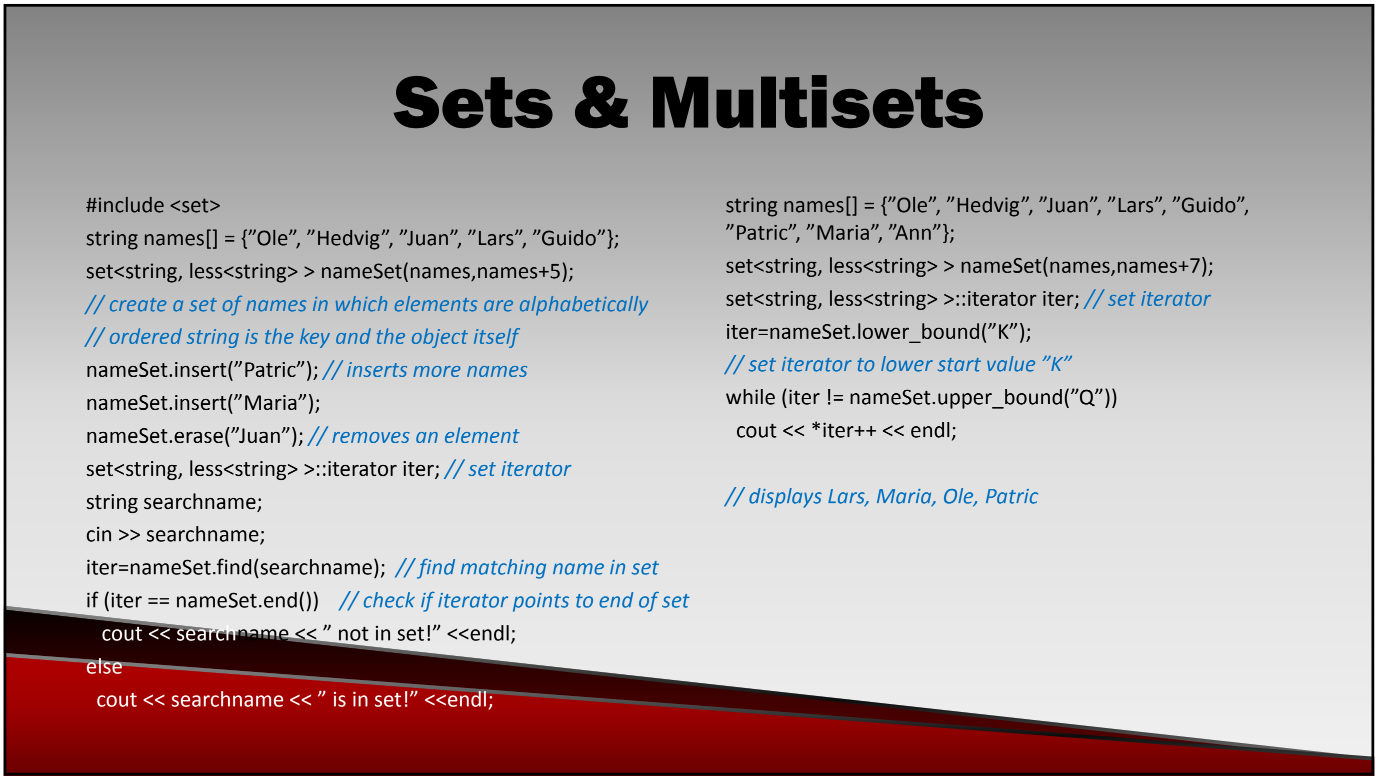
Associative Containers

In an associative container the items are not arranged in sequence, but usually as a tree structure or a hash table.

The main advantage of associative containers is the speed of searching (binary search like in a dictionary)

Searching is done using a key which is usually a single value like a number or string.

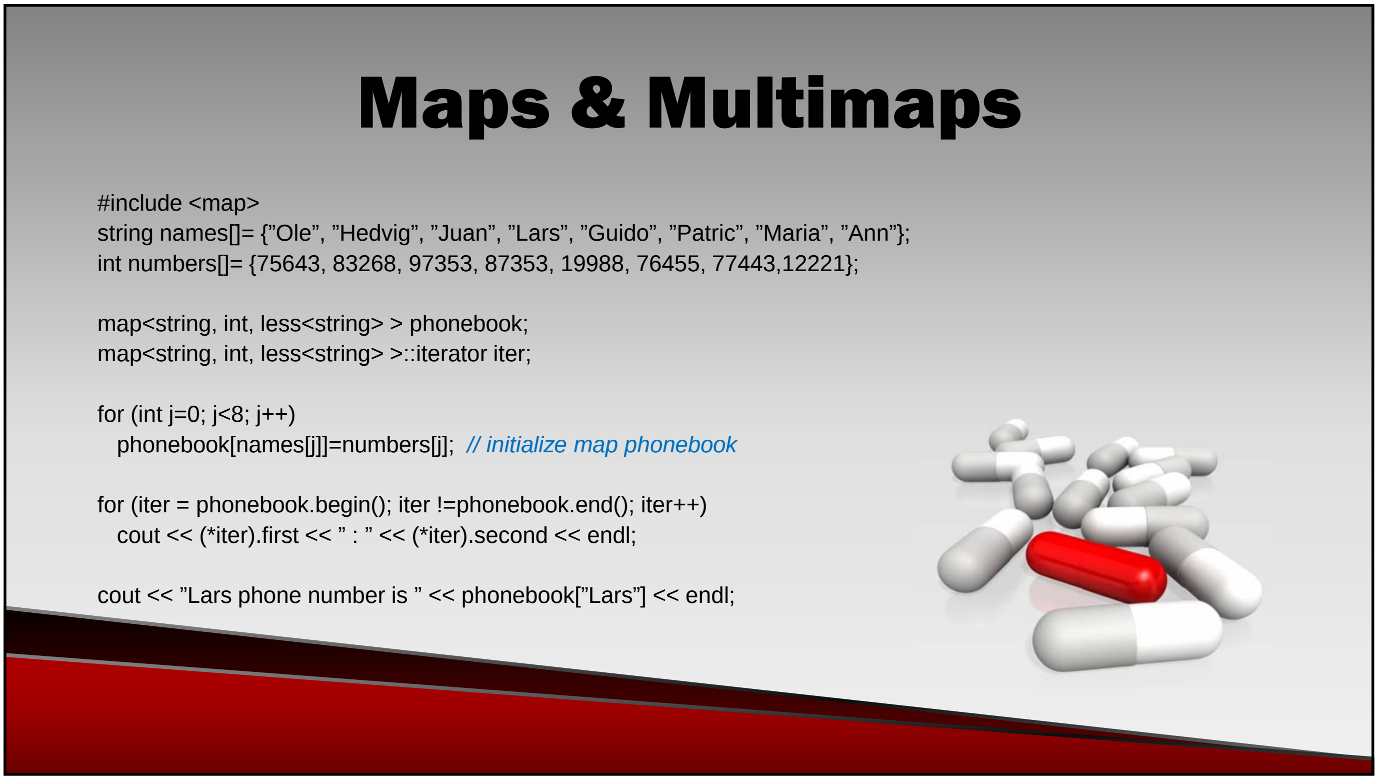
The value is an attribute of the objects in the container



Sets & Multisets

```
#include <set>
string names[] = {”Ole”, ”Hedvig”, ”Juan”, ”Lars”, ”Guido”};
set<string, less<string> > nameSet(names,names+5);
// create a set of names in which elements are alphabetically
// ordered string is the key and the object itself
nameSet.insert(”Patric”); // inserts more names
nameSet.insert(”Maria”);
nameSet.erase(”Juan”); // removes an element
set<string, less<string> >::iterator iter; // set iterator
string searchname;
cin >> searchname;
iter=nameSet.find(searchname);  // find matching name in set
if (iter == nameSet.end())    // check if iterator points to end of set
   cout << searchname << ” not in set!” <<endl;
else
  cout << searchname << ” is in set!” <<endl;
```

```
string names[] = {”Ole”, ”Hedvig”, ”Juan”, ”Lars”, ”Guido”,
”Patric”, ”Maria”, ”Ann”};
set<string, less<string> > nameSet(names,names+7);
set<string, less<string> >::iterator iter; // set iterator
iter=nameSet.lower_bound(”K”);
// set iterator to lower start value ”K”
while (iter != nameSet.upper_bound(”Q”))
  cout << *iter++ << endl;

// displays Lars, Maria, Ole, Patric
```



Maps & Multimaps

```
#include <map>
string names[]= {”Ole”, ”Hedvig”, ”Juan”, ”Lars”, ”Guido”, ”Patric”, ”Maria”, ”Ann”};
int numbers[]= {75643, 83268, 97353, 87353, 19988, 76455, 77443,12221};

map<string, int, less<string> > phonebook;
map<string, int, less<string> >::iterator iter;

for (int j=0; j<8; j++)
   phonebook[names[j]]=numbers[j];  // initialize map phonebook

for (iter = phonebook.begin(); iter !=phonebook.end(); iter++)
   cout << (*iter).first << ” : ” << (*iter).second << endl;

cout << ”Lars phone number is ” << phonebook[”Lars”] << endl;
```

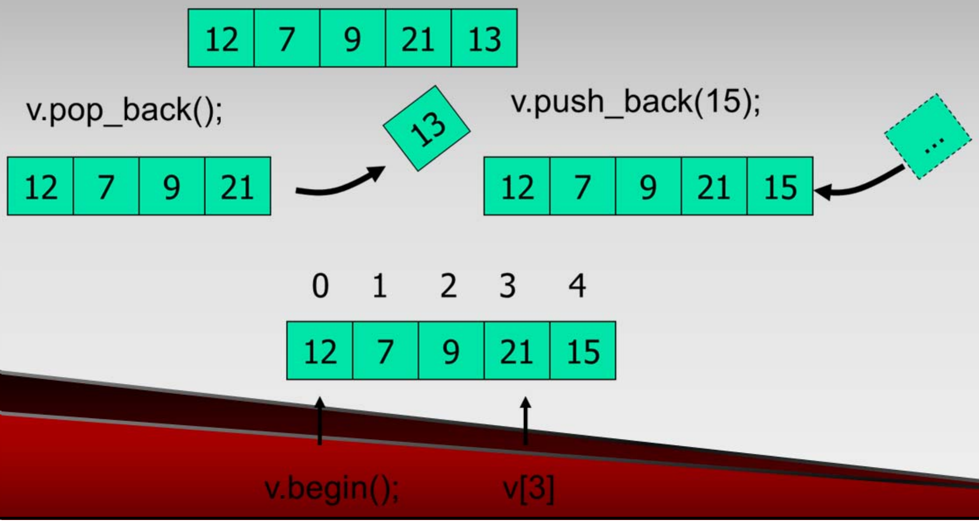
Multiset Example

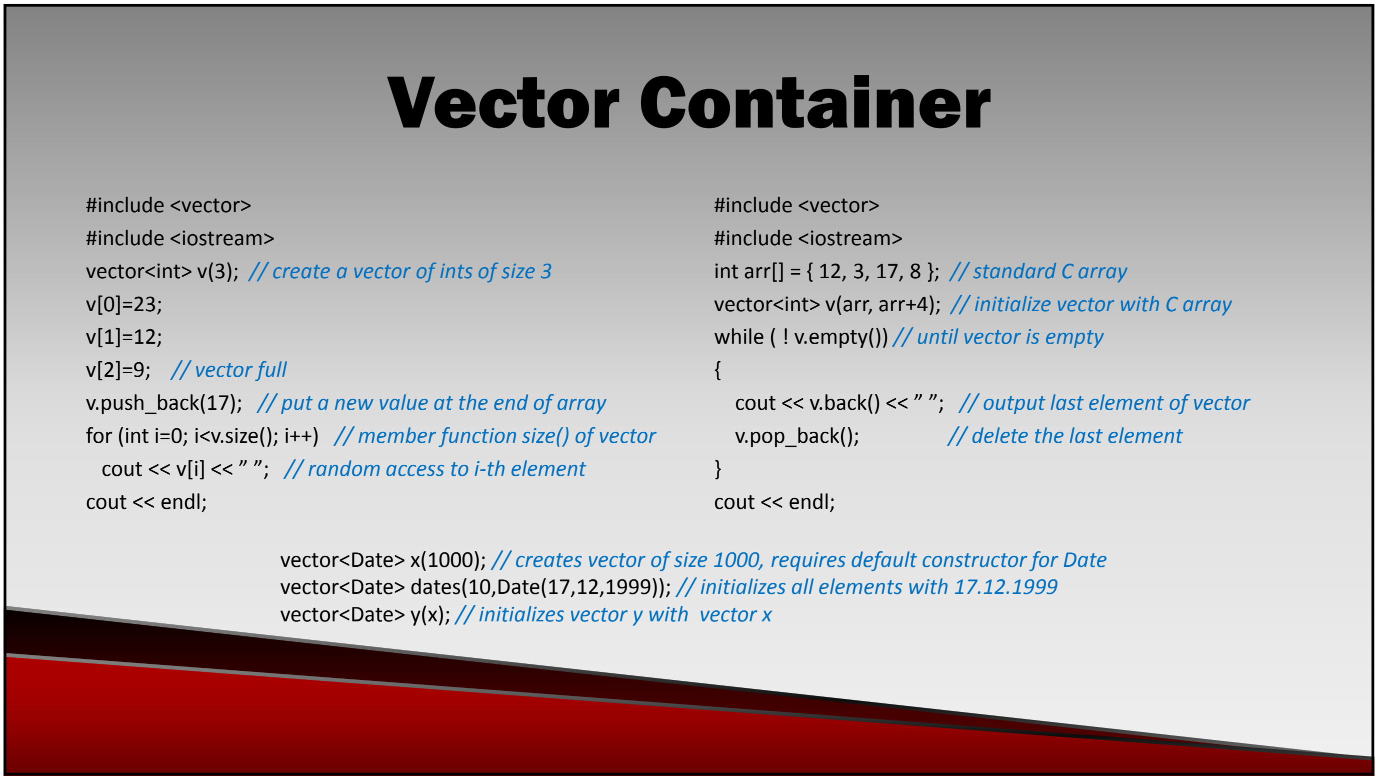
```
class person
{
   private:
     string lastName;
     string firstName;
     long phoneNumber;
   public:
      person(string lana, string fina, long pho) : lastName(lana), 
firstName(fina), phonenumber(pho) {}
   bool operator<(const person& p);
   bool operator==(const person& p);
}
```

```
person p1(”Neuville”, ”Oliver”, 5103452348);
person p2(”Kirsten”, ”Ulf”, 5102782837);
person p3(”Larssen”, ”Henrik”, 8904892921);
multiset<person, less<person>> persSet;
multiset<person, less<person>>::iterator iter;
persSet.insert(p1);
persSet.insert(p2);
persSet.insert(p3);
```

Vector Container

```
int array[5] = {12, 7, 9, 21, 13 };
vector<int> v(array,array+5);
```



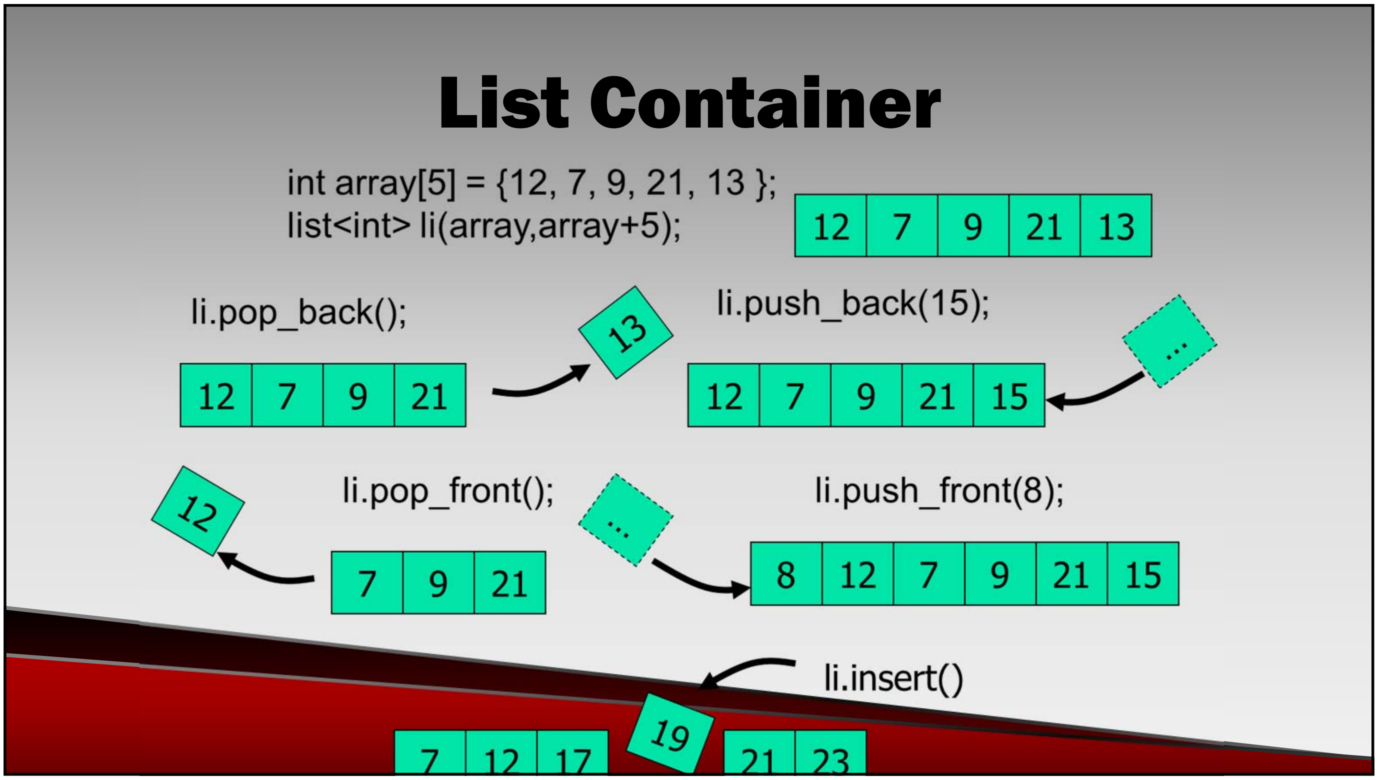


Vector Container

```
#include <vector>
#include <iostream>
vector<int> v(3);  // create a vector of ints of size 3
v[0]=23;
v[1]=12;
v[2]=9;    // vector full
v.push_back(17);   // put a new value at the end of array
for (int i=0; i<v.size(); i++)   // member function size() of vector
   cout << v[i] << ” ”;   // random access to i-th element
cout << endl;
```

```
#include <vector>
#include <iostream>
int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array
while ( ! v.empty()) // until vector is empty
{
   cout << v.back() << ” ”;   // output last element of vector
   v.pop_back();              // delete the last element
}
cout << endl;
```

```
vector<Date> x(1000); // creates vector of size 1000, requires default constructor for Date
vector<Date> dates(10,Date(17,12,1999)); // initializes all elements with 17.12.1999
vector<Date> y(x); // initializes vector y with  vector x
```



List Container

```
int array[5] = {12, 7, 9, 21, 13 };
list<int> li(array,array+5);
```

12 7 9 21 13

li.pop_back();

12 7 9 21

13

li.push_back(15);

12 7 9 21 15

...

li.pop_front();

12

7 9 21

li.push_front(8);

...

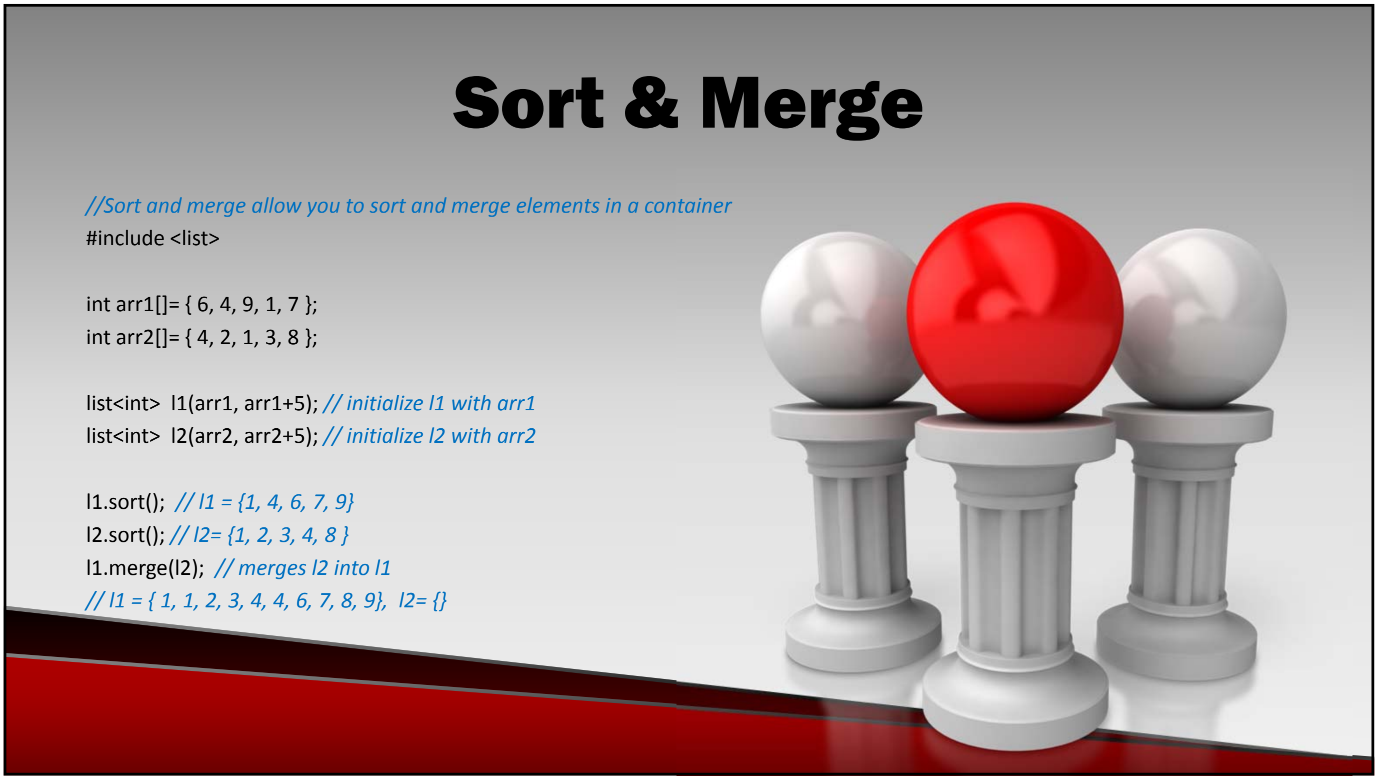
8 12 7 9 21 15

li.insert()

7 12 17

19

21 23



Sort & Merge

```
//Sort and merge allow you to sort and merge elements in a container
#include <list>

int arr1[]= { 6, 4, 9, 1, 7 };
int arr2[]= { 4, 2, 1, 3, 8 };

list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2

l1.sort();  // l1 = {1, 4, 6, 7, 9}
l2.sort(); // l2= {1, 2, 3, 4, 8 }
l1.merge(l2);  // merges l2 into l1
// l1 = { 1, 1, 2, 3, 4, 4, 6, 7, 8, 9},  l2= {}
```

Part III

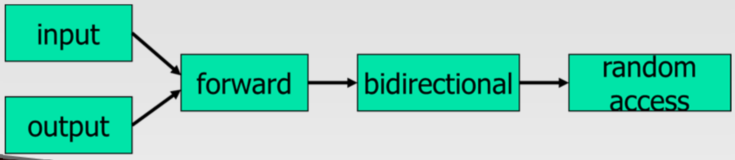
ITERATORS

Iterators

Iterators are a generalization of the concept of pointers, they point to elements in a container, for example you can increment an iterator to point to the next element in an array

Iterators are a generalization of pointers.

There are also some iterators, such as istream_iterator and ostream_iterator, that aren't associated with containers at all.



Iterators

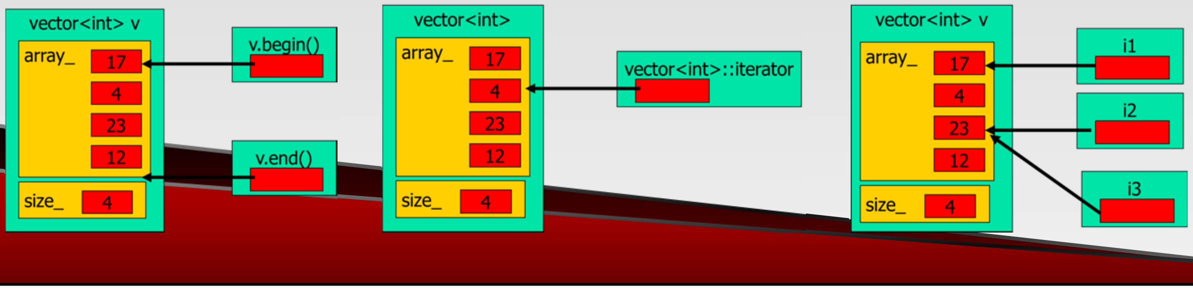
Iterators are pointer-like entities that are used to access individual elements in a container.

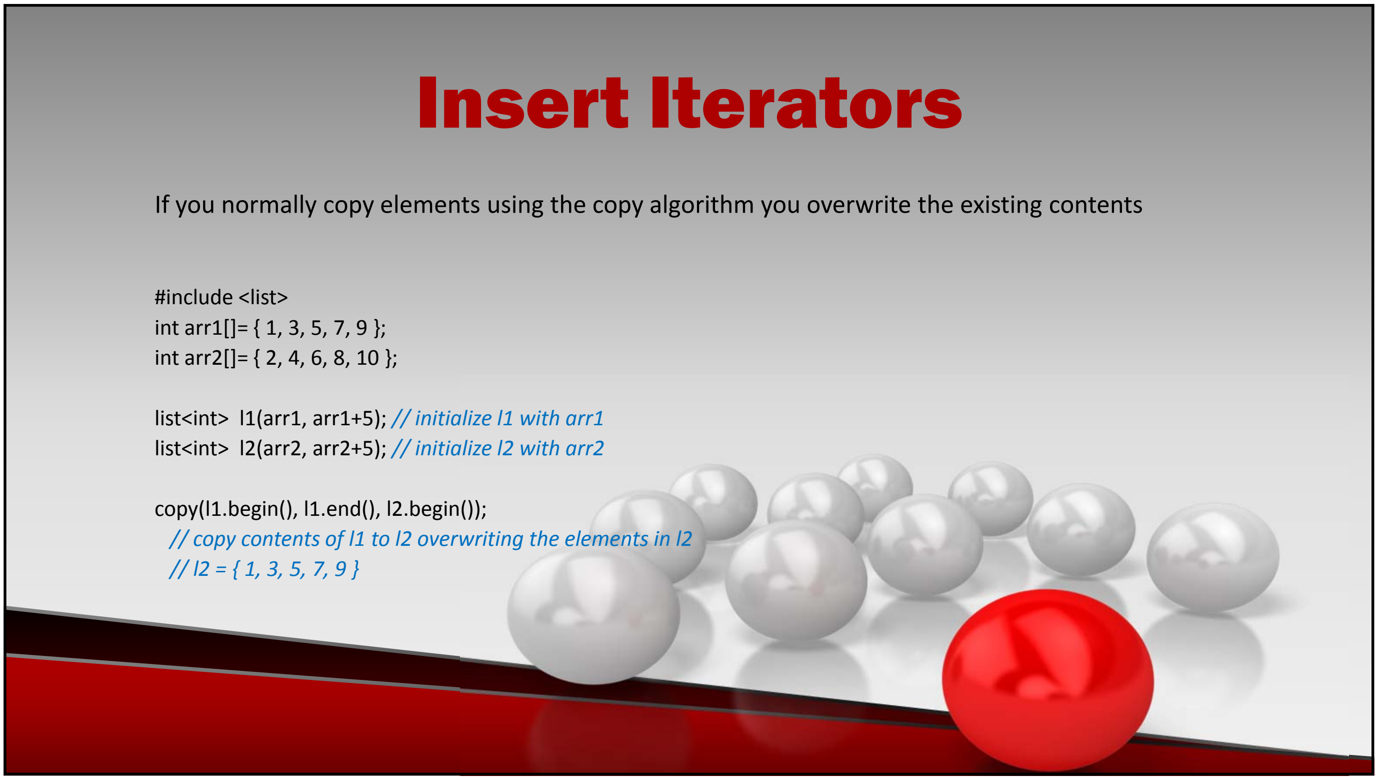
Often they are used to move sequentially from element to element, a process called *iterating* through a container.

The member functions begin() and end() return an iterator to the first and past the last element of a container

The iterator corresponding to the class vector<int> is of the type vector<int>::iterator

One can have multiple iterators pointing to different or identical elements in the container

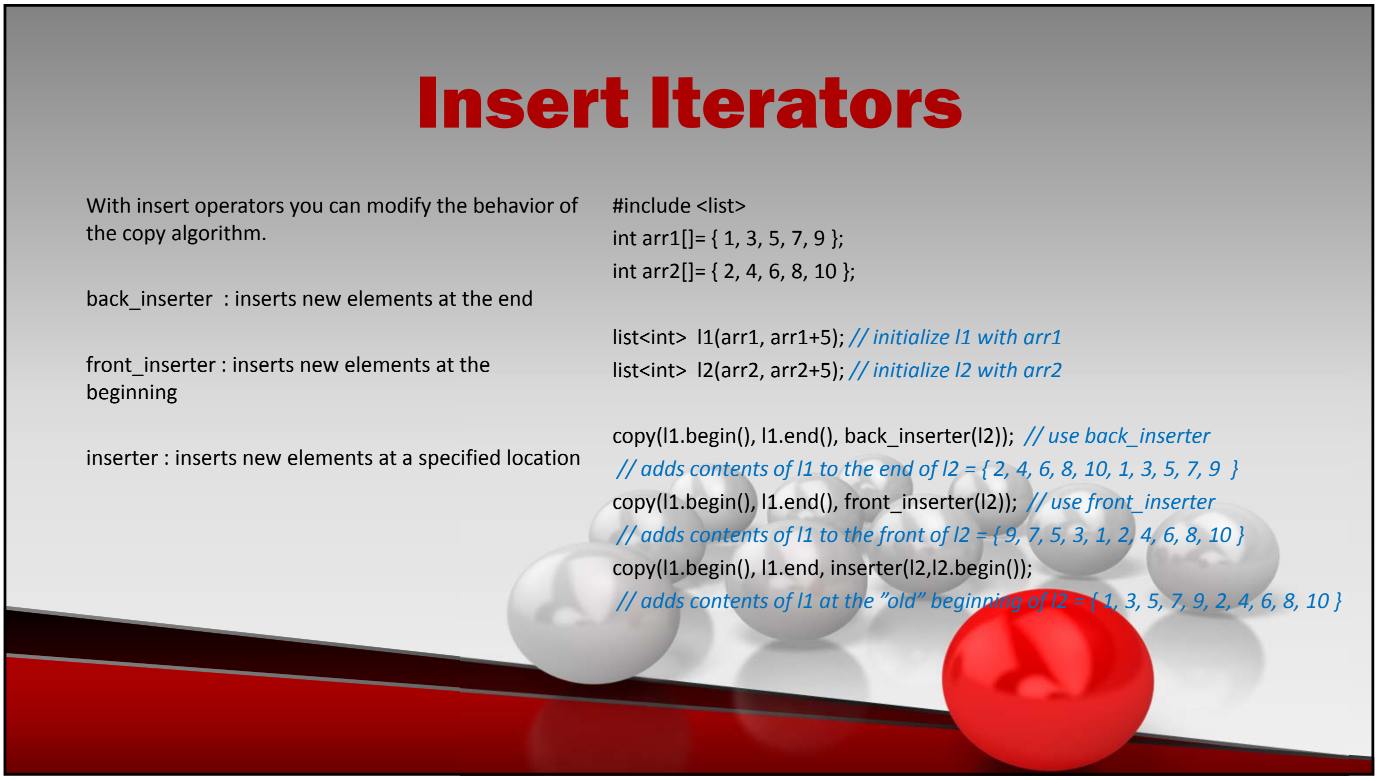




Insert Iterators

If you normally copy elements using the copy algorithm you overwrite the existing contents

```
#include <list>
int arr1[]= { 1, 3, 5, 7, 9 };
int arr2[]= { 2, 4, 6, 8, 10 };

list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2

copy(l1.begin(), l1.end(), l2.begin());
  // copy contents of l1 to l2 overwriting the elements in l2
  // l2 = { 1, 3, 5, 7, 9 }
```



Insert Iterators

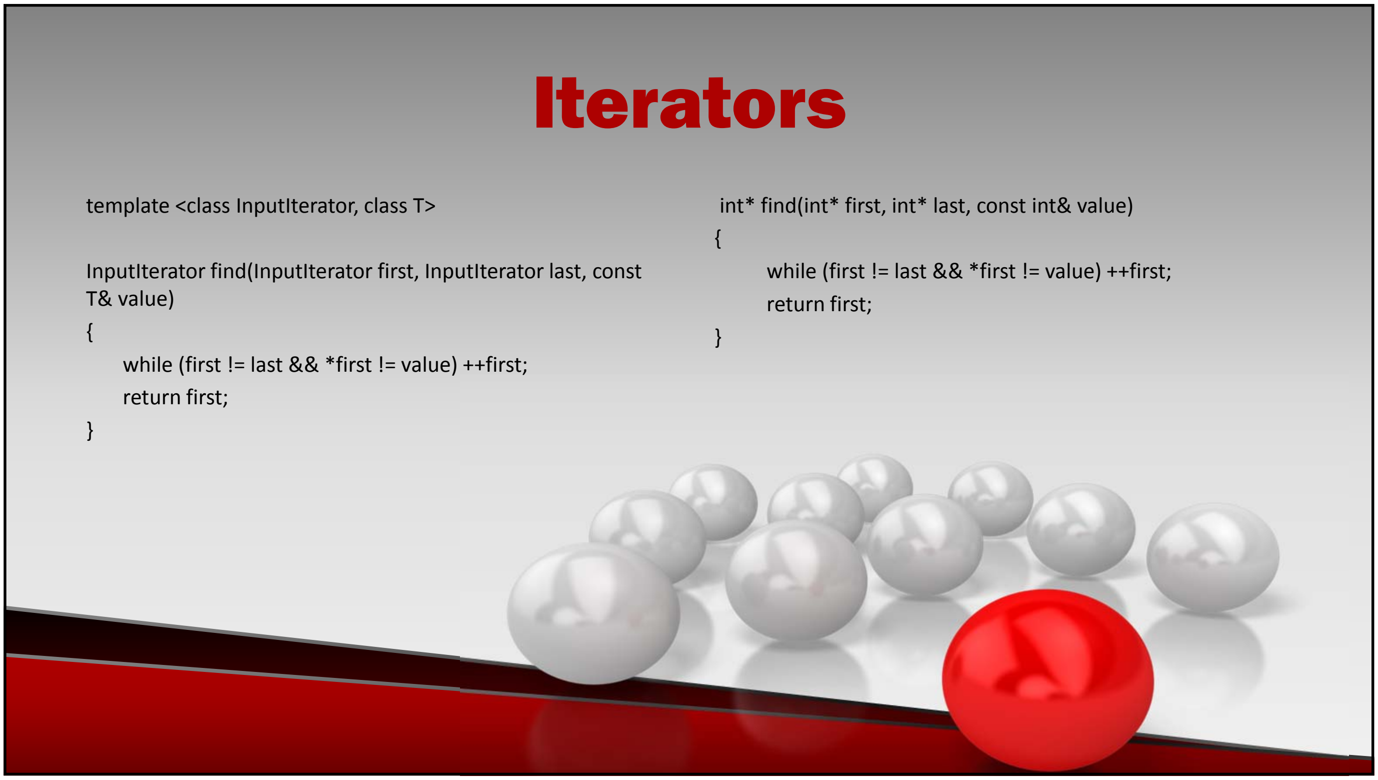
With insert operators you can modify the behavior of the copy algorithm.

back_inserter : inserts new elements at the end

front_inserter : inserts new elements at the beginning

inserter : inserts new elements at a specified location

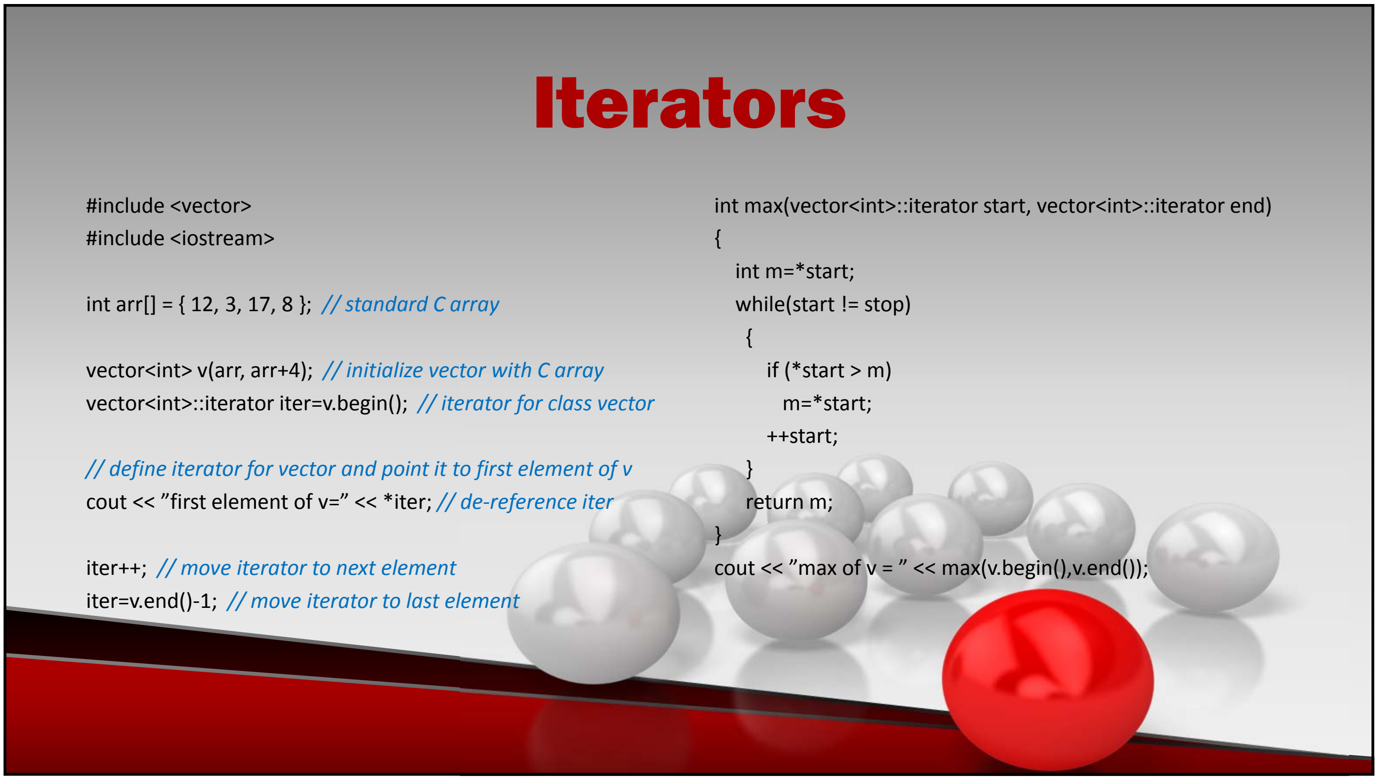
```
#include <list>
int arr1[]= { 1, 3, 5, 7, 9 };
int arr2[]= { 2, 4, 6, 8, 10 };

list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2

copy(l1.begin(), l1.end(), back_inserter(l2));  // use back_inserter
 // adds contents of l1 to the end of l2 = { 2, 4, 6, 8, 10, 1, 3, 5, 7, 9  }
copy(l1.begin(), l1.end(), front_inserter(l2));  // use front_inserter
 // adds contents of l1 to the front of l2 = { 9, 7, 5, 3, 1, 2, 4, 6, 8, 10 }
copy(l1.begin(), l1.end, inserter(l2,l2.begin());
 // adds contents of l1 at the "old" beginning of l2 = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 }
```



Iterators

```
template <class InputIterator, class T>

InputIterator find(InputIterator first, InputIterator last, const
T& value)
{
      while (first != last && *first != value) ++first;
      return first;
}
```

```
int* find(int* first, int* last, const int& value)
{
        while (first != last && *first != value) ++first;
        return first;
}
```



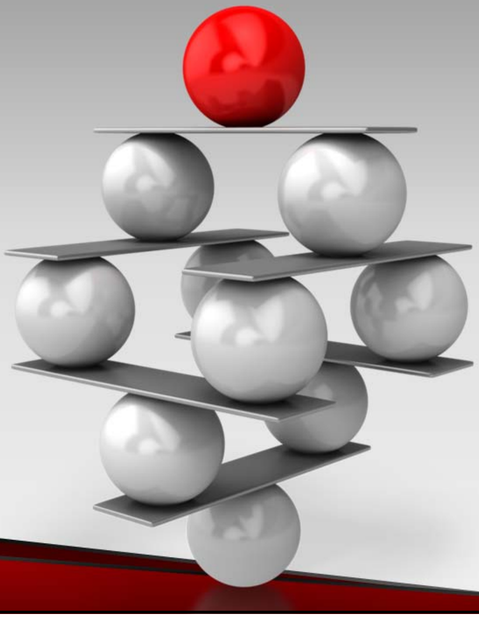
Iterators

```
#include <vector>
#include <iostream>

int arr[] = { 12, 3, 17, 8 };  // standard C array

vector<int> v(arr, arr+4);  // initialize vector with C array
vector<int>::iterator iter=v.begin();  // iterator for class vector

// define iterator for vector and point it to first element of v
cout << ”first element of v=” << *iter; // de-reference iter

iter++;  // move iterator to next element
iter=v.end()-1;  // move iterator to last element
```

```
int max(vector<int>::iterator start, vector<int>::iterator end)
{
   int m=*start;
   while(start != stop)
    {
        if (*start > m)
          m=*start;
        ++start;
    }
    return m;
}
cout << ”max of v = ” << max(v.begin(),v.end());
```

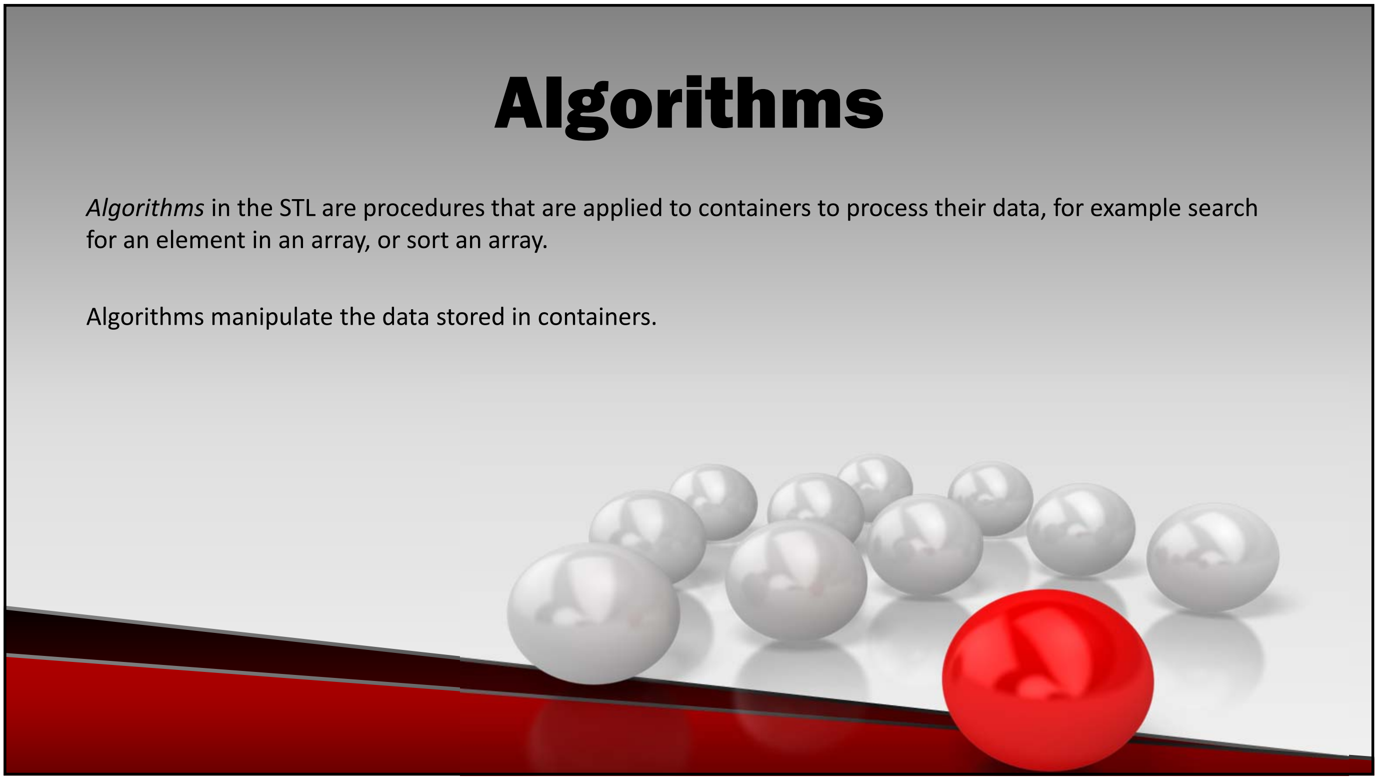
Iterators

```
#include <vector>
#include <iostream>
int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array

for (vector<int>::iterator i=v.begin(); i!=v.end(); i++)
// initialize i with pointer to first element of v
// i++ increment iterator, move iterator to next element
{
    cout << *i << " ";   // de-referencing iterator returns the
                         // value of the element the iterator points at
}
cout << endl;
```



Part IV

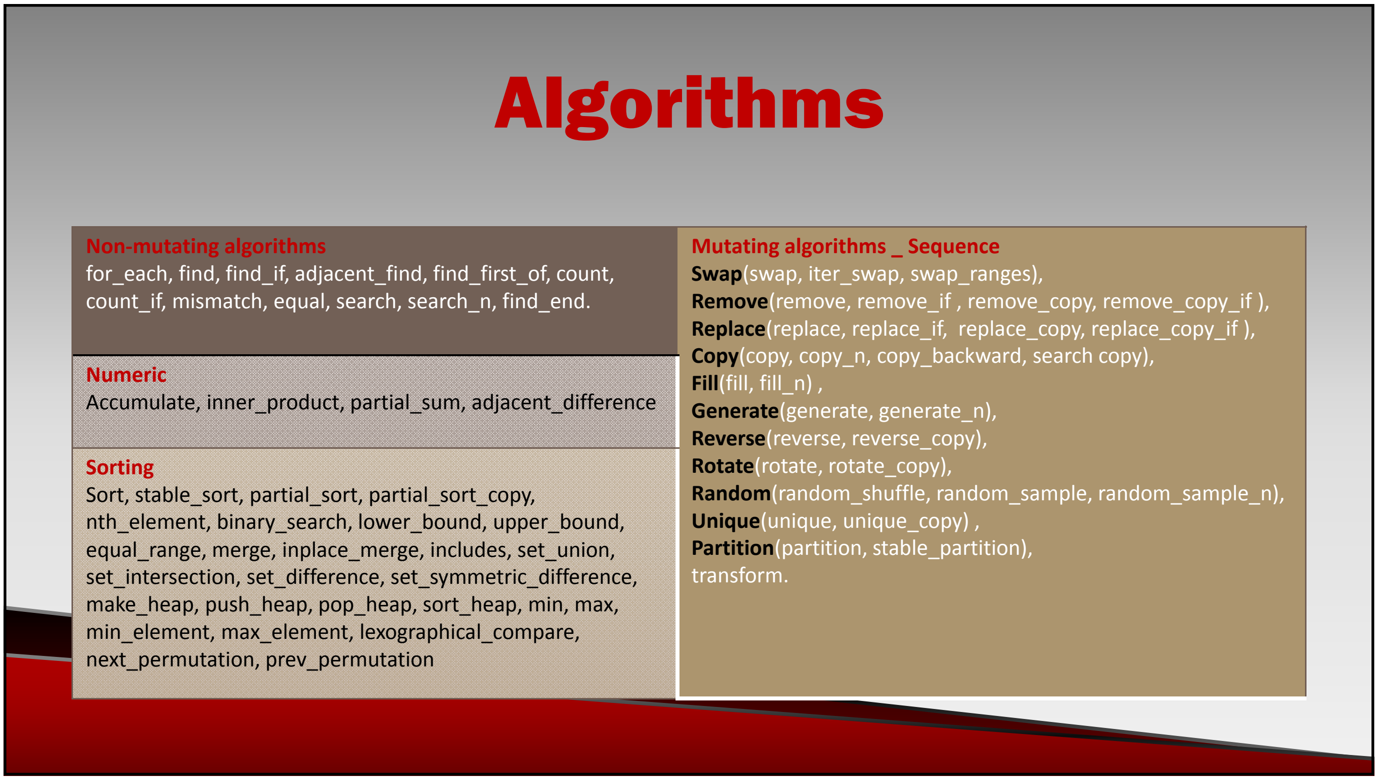
ALGORITHMS



Algorithms

Algorithms in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.

Algorithms manipulate the data stored in containers.



Algorithms

Non-mutating algorithms
for_each, find, find_if, adjacent_find, find_first_of, count, count_if, mismatch, equal, search, search_n, find_end.

Numeric
Accumulate, inner_product, partial_sum, adjacent_difference

Sorting
Sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, make_heap, push_heap, pop_heap, sort_heap, min, max, min_element, max_element, lexographical_compare, next_permutation, prev_permutation

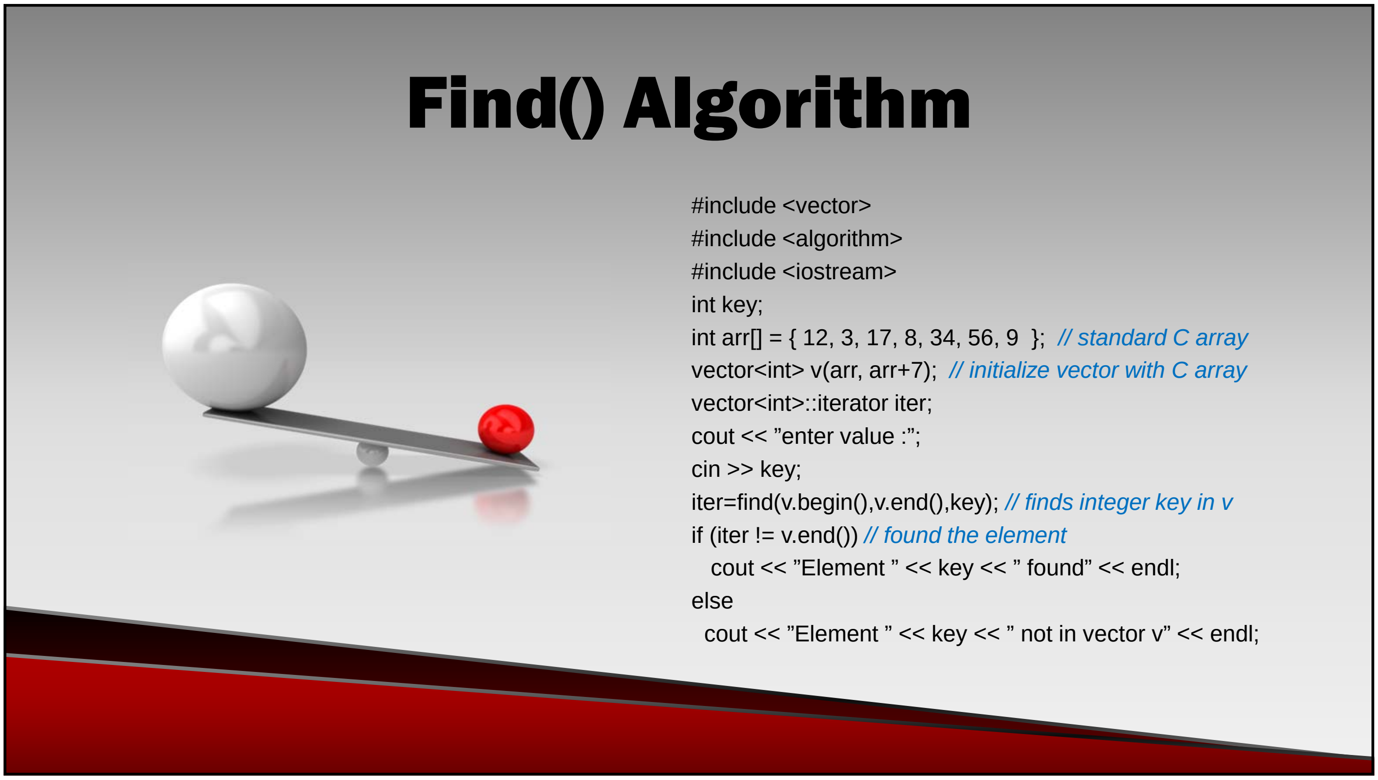
Mutating algorithms _ Sequence
Swap(swap, iter_swap, swap_ranges),
Remove(remove, remove_if , remove_copy, remove_copy_if),
Replace(replace, replace_if, replace_copy, replace_copy_if),
Copy(copy, copy_n, copy_backward, search copy),
Fill(fill, fill_n) ,
Generate(generate, generate_n),
Reverse(reverse, reverse_copy),
Rotate(rotate, rotate_copy),
Random(random_shuffle, random_sample, random_sample_n),
Unique(unique, unique_copy) ,
Partition(partition, stable_partition),
transform.

Reverse Algorithm

```
reverse(v.begin(), v.end()); // v[0] == 17, v[1] == 10, v[2] == 7

double A[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
    reverse(A, A + 6);
    for (int i = 0; i < 6; ++i)
      cout << "A[" << i << "] = " << A[i];
```

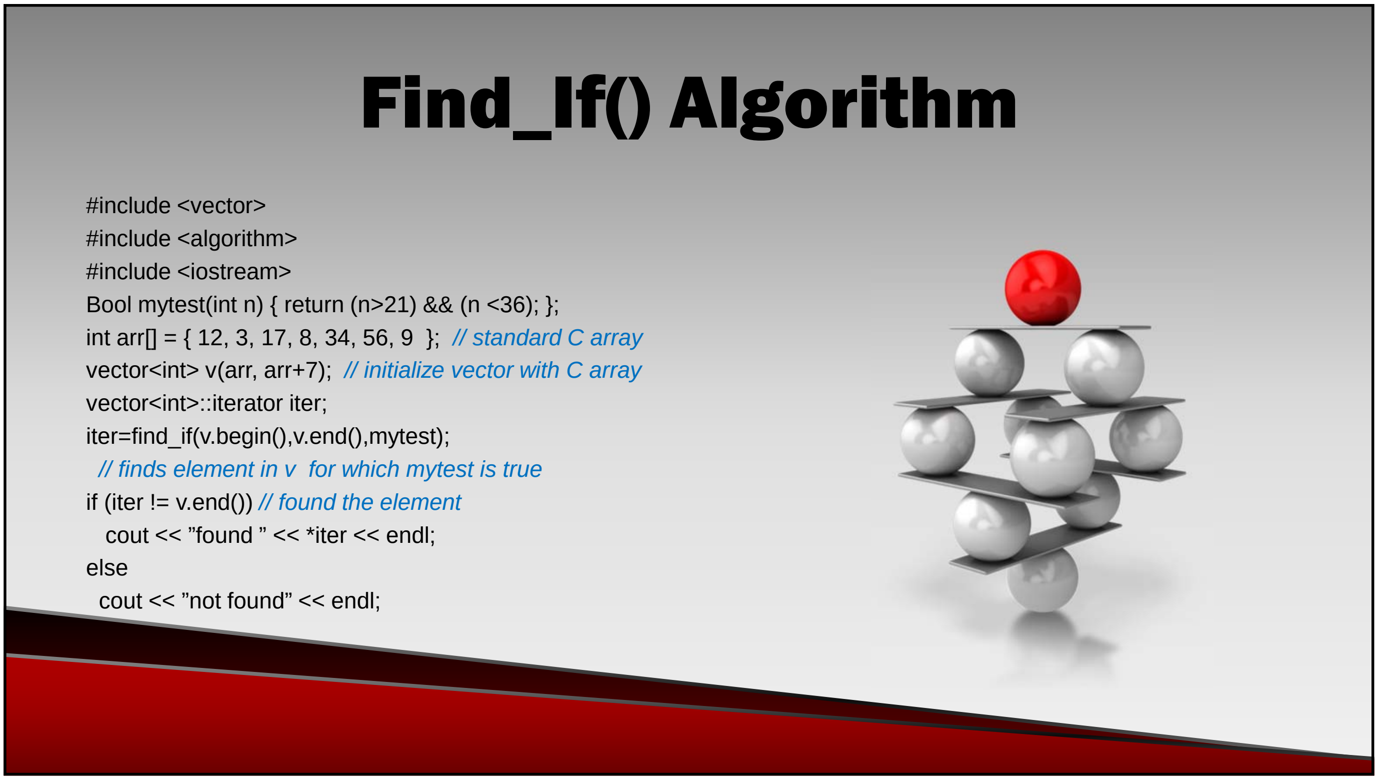
For_Each() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
void show(int n)
{
  cout << n << " ";
}

 int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array

for_each (v.begin(), v.end(), show); // apply function show
                                      // to each element of vector v
```



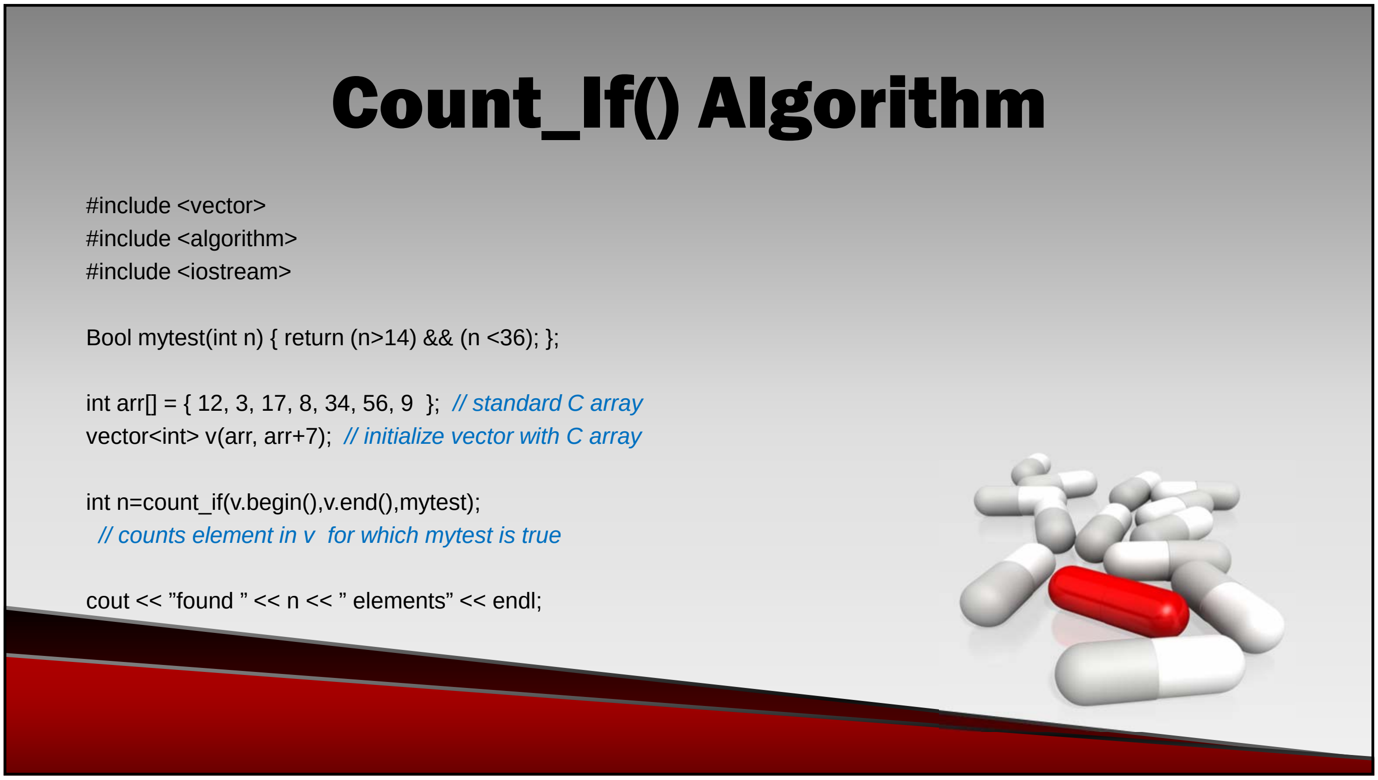
Find() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
int key;
int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array
vector<int>::iterator iter;
cout << ”enter value :”;
cin >> key;
iter=find(v.begin(),v.end(),key); // finds integer key in v
if (iter != v.end()) // found the element
   cout << ”Element ” << key << ” found” << endl;
else
  cout << ”Element ” << key << ” not in vector v” << endl;
```



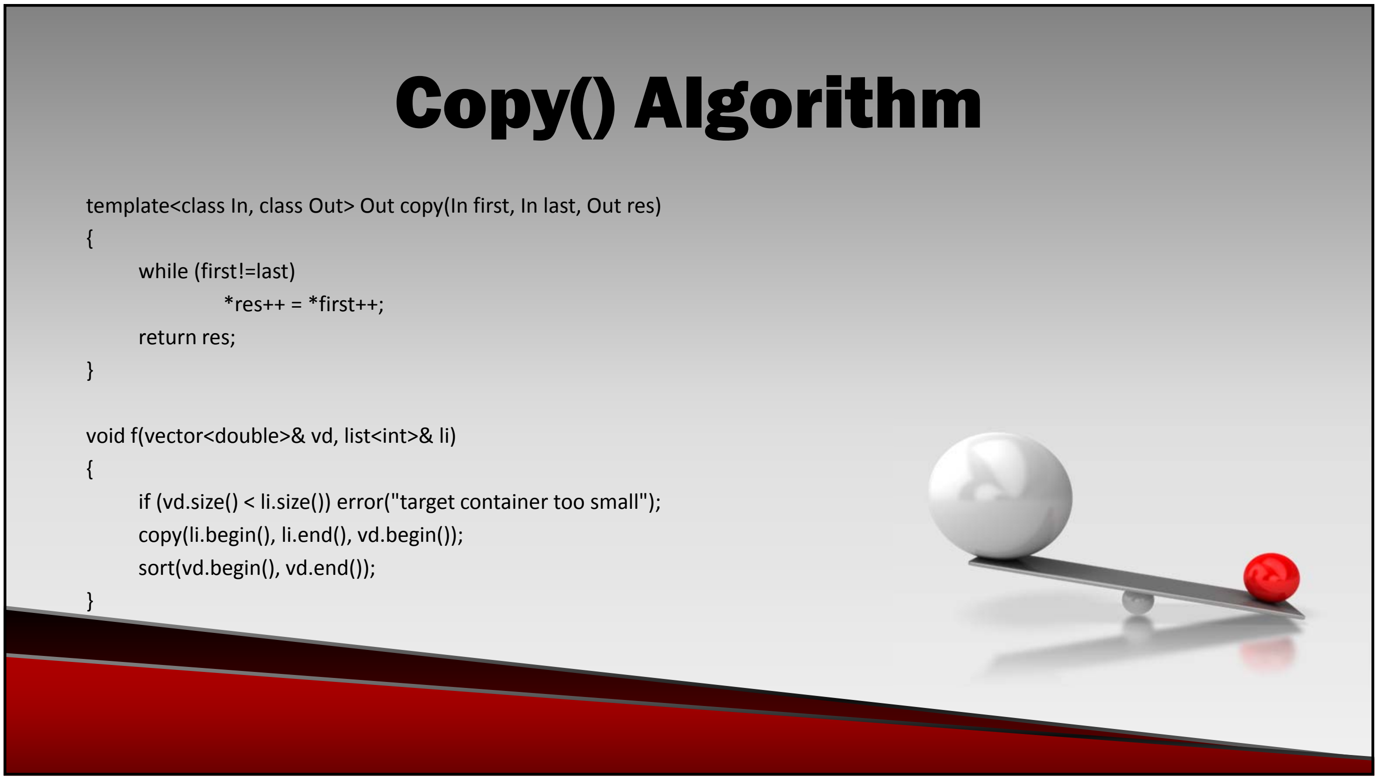
Find_If() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
Bool mytest(int n) { return (n>21) && (n <36); };
int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array
vector<int>::iterator iter;
iter=find_if(v.begin(),v.end(),mytest);
 // finds element in v  for which mytest is true
if (iter != v.end()) // found the element
   cout << ”found ” << *iter << endl;
else
  cout << ”not found” << endl;
```



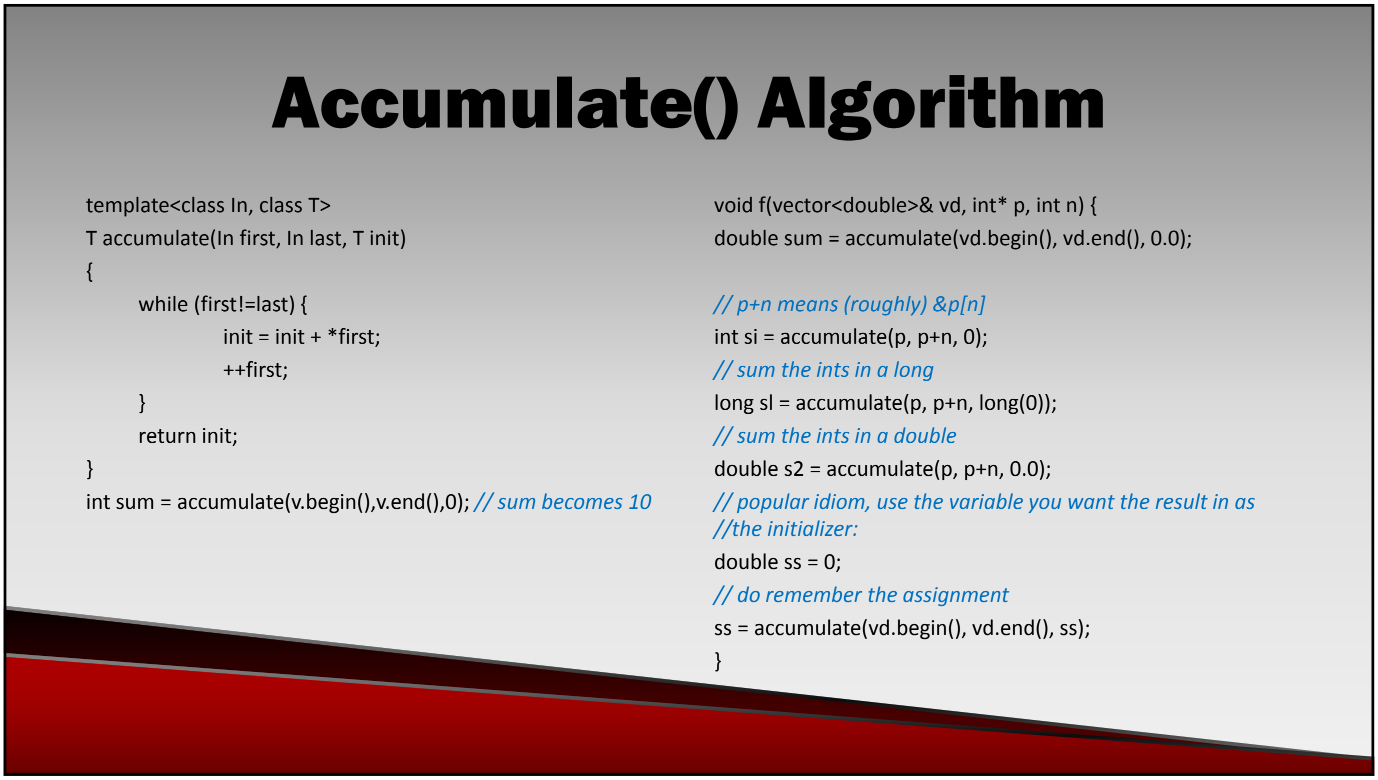
Count_If() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>

Bool mytest(int n) { return (n>14) && (n <36); };

int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array

int n=count_if(v.begin(),v.end(),mytest);
  // counts element in v  for which mytest is true

cout << ”found ” << n << ” elements” << endl;
```



Copy() Algorithm

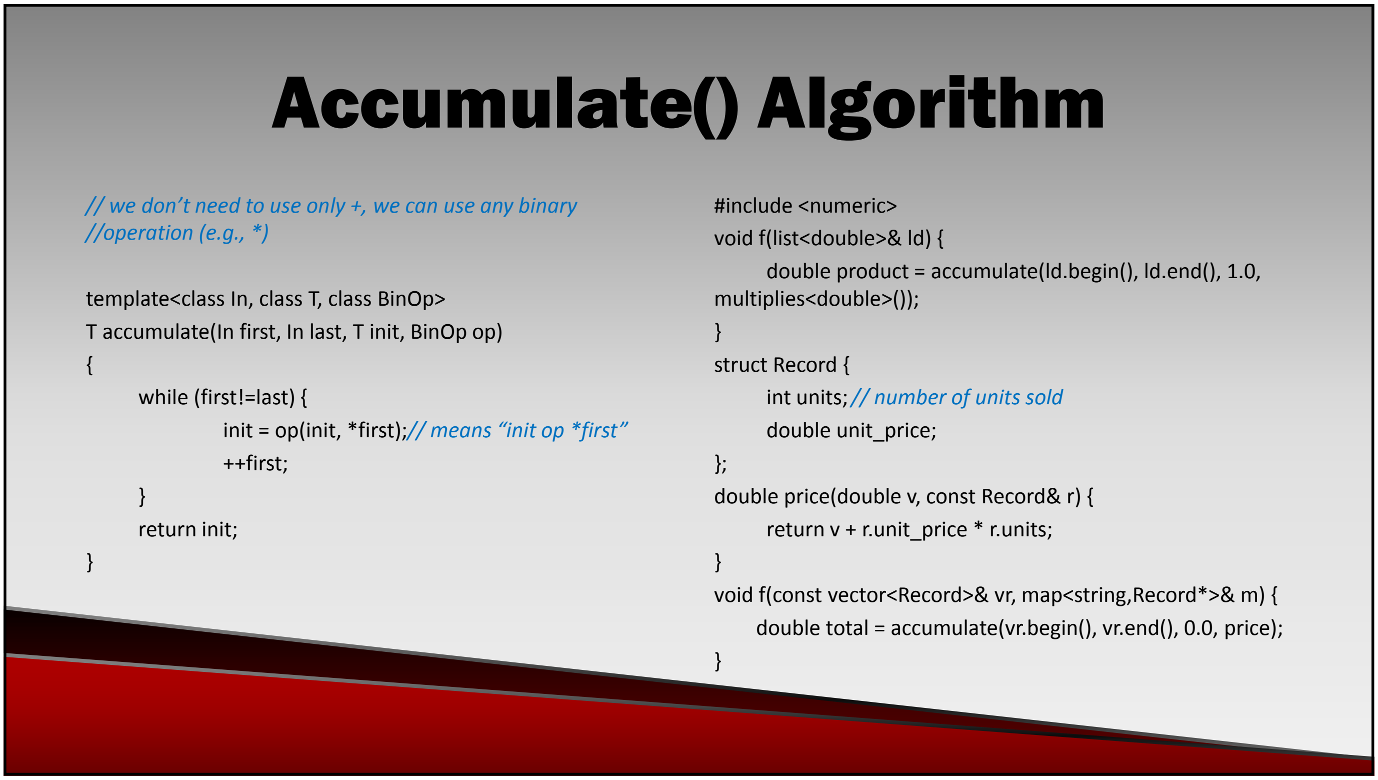
```
template<class In, class Out> Out copy(In first, In last, Out res)
{
        while (first!=last)
                    *res++ = *first++;
        return res;
}

void f(vector<double>& vd, list<int>& li)
{
        if (vd.size() < li.size()) error("target container too small");
        copy(li.begin(), li.end(), vd.begin());
        sort(vd.begin(), vd.end());
}
```



Accumulate() Algorithm

```
template<class In, class T>
T accumulate(In first, In last, T init)
{
        while (first!=last) {
                init = init + *first;
                ++first;
        }
        return init;
}
int sum = accumulate(v.begin(),v.end(),0); // sum becomes 10
```

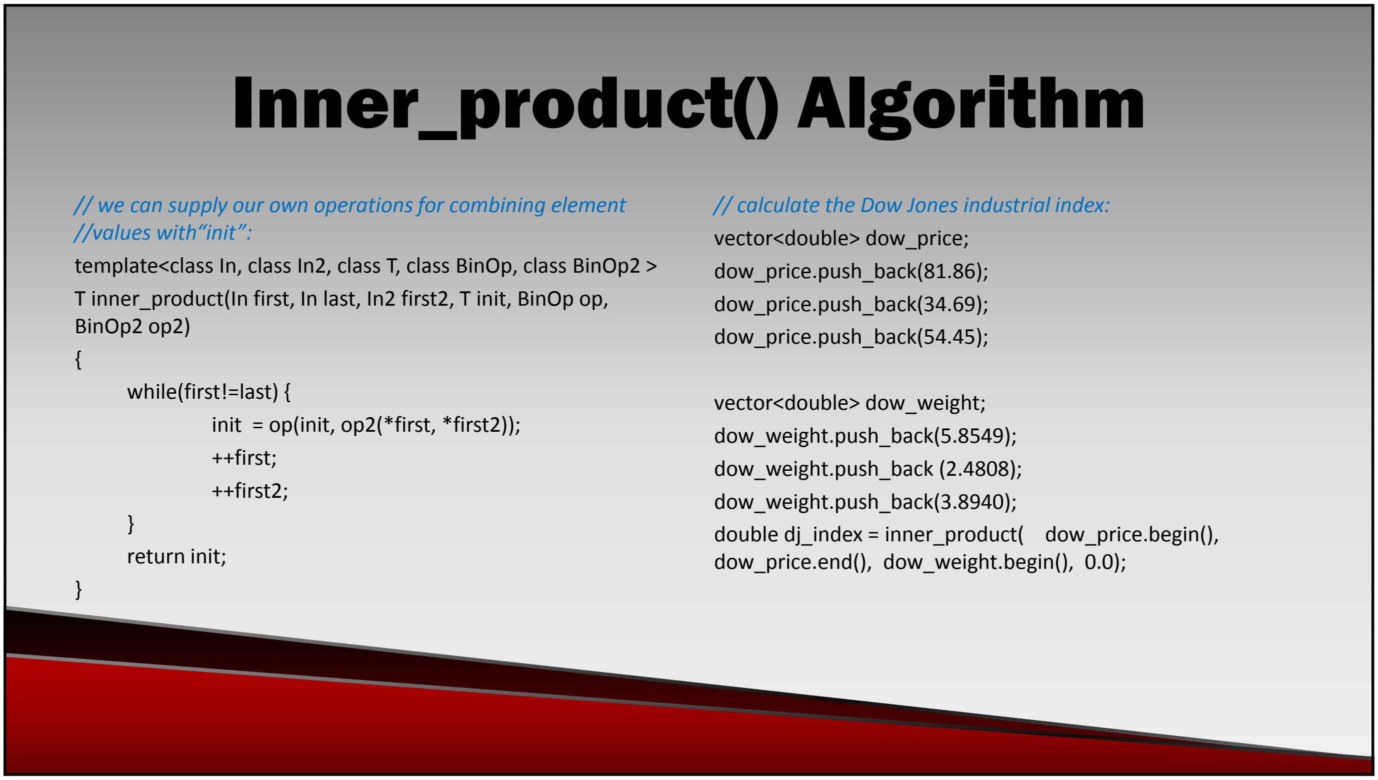
```
void f(vector<double>& vd, int* p, int n) {
double sum = accumulate(vd.begin(), vd.end(), 0.0);

// p+n means (roughly) &p[n]
int si = accumulate(p, p+n, 0);
// sum the ints in a long
long sl = accumulate(p, p+n, long(0));
// sum the ints in a double
double s2 = accumulate(p, p+n, 0.0);
// popular idiom, use the variable you want the result in as
//the initializer:
double ss = 0;
// do remember the assignment
ss = accumulate(vd.begin(), vd.end(), ss);
}
```



Accumulate() Algorithm

```
// we don't need to use only +, we can use any binary
//operation (e.g., *)

template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op)
{
        while (first!=last) {
                init = op(init, *first);// means "init op *first"
                ++first;
        }
        return init;
}
```

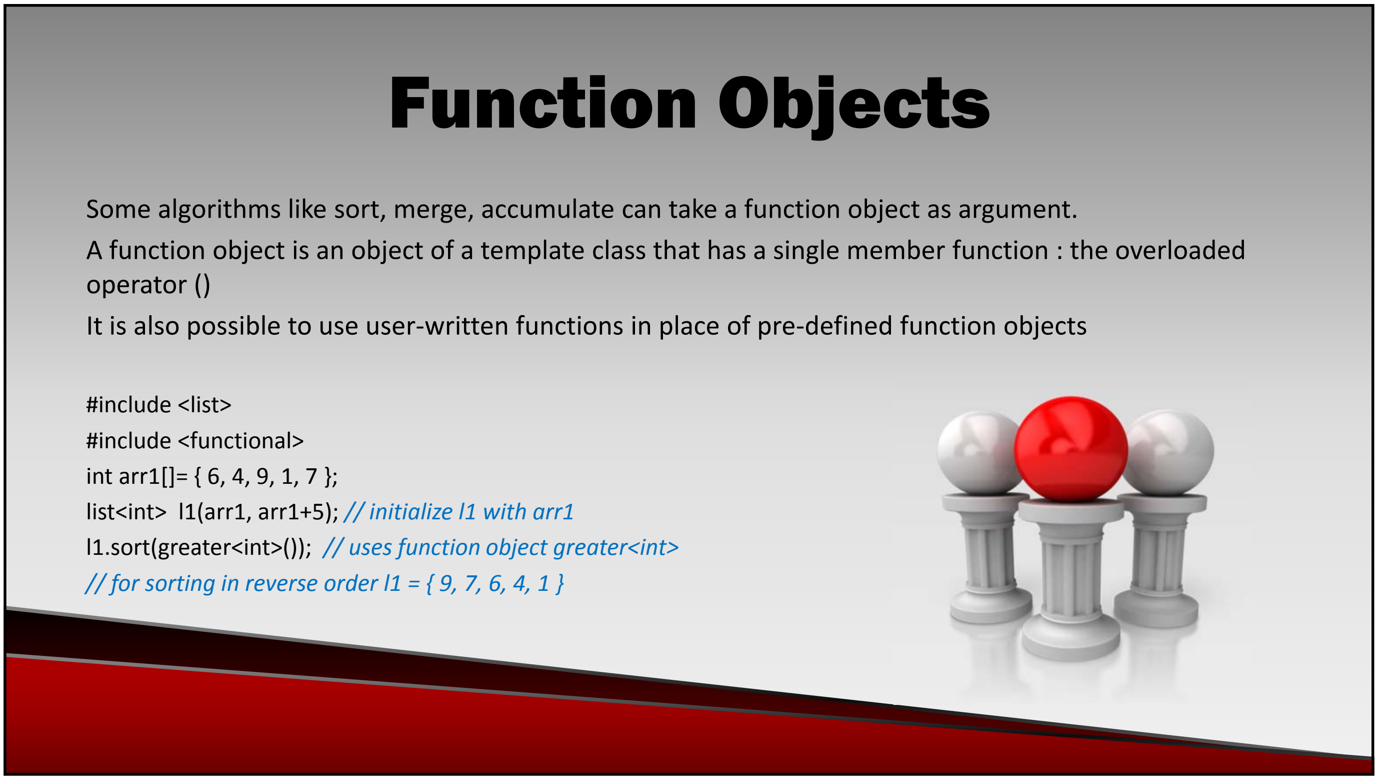
```
#include <numeric>
void f(list<double>& ld) {
        double product = accumulate(ld.begin(), ld.end(), 1.0,
multiplies<double>());
}
struct Record {
        int units;// number of units sold
        double unit_price;
};
double price(double v, const Record& r) {
        return v + r.unit_price * r.units;
}
void f(const vector<Record>& vr, map<string,Record*>& m) {
      double total = accumulate(vr.begin(), vr.end(), 0.0, price);
}
```



Inner_product() Algorithm

```
// we can supply our own operations for combining element
//values with“init”:
template<class In, class In2, class T, class BinOp, class BinOp2 >
T inner_product(In first, In last, In2 first2, T init, BinOp op,
BinOp2 op2)
{
        while(first!=last) {
                init  = op(init, op2(*first, *first2));
                ++first;
                ++first2;
        }
        return init;
}
```

```
// calculate the Dow Jones industrial index:
vector<double> dow_price;
dow_price.push_back(81.86);
dow_price.push_back(34.69);
dow_price.push_back(54.45);

vector<double> dow_weight;
dow_weight.push_back(5.8549);
dow_weight.push_back (2.4808);
dow_weight.push_back(3.8940);
double dj_index = inner_product(    dow_price.begin(),
dow_price.end(),  dow_weight.begin(),  0.0);
```



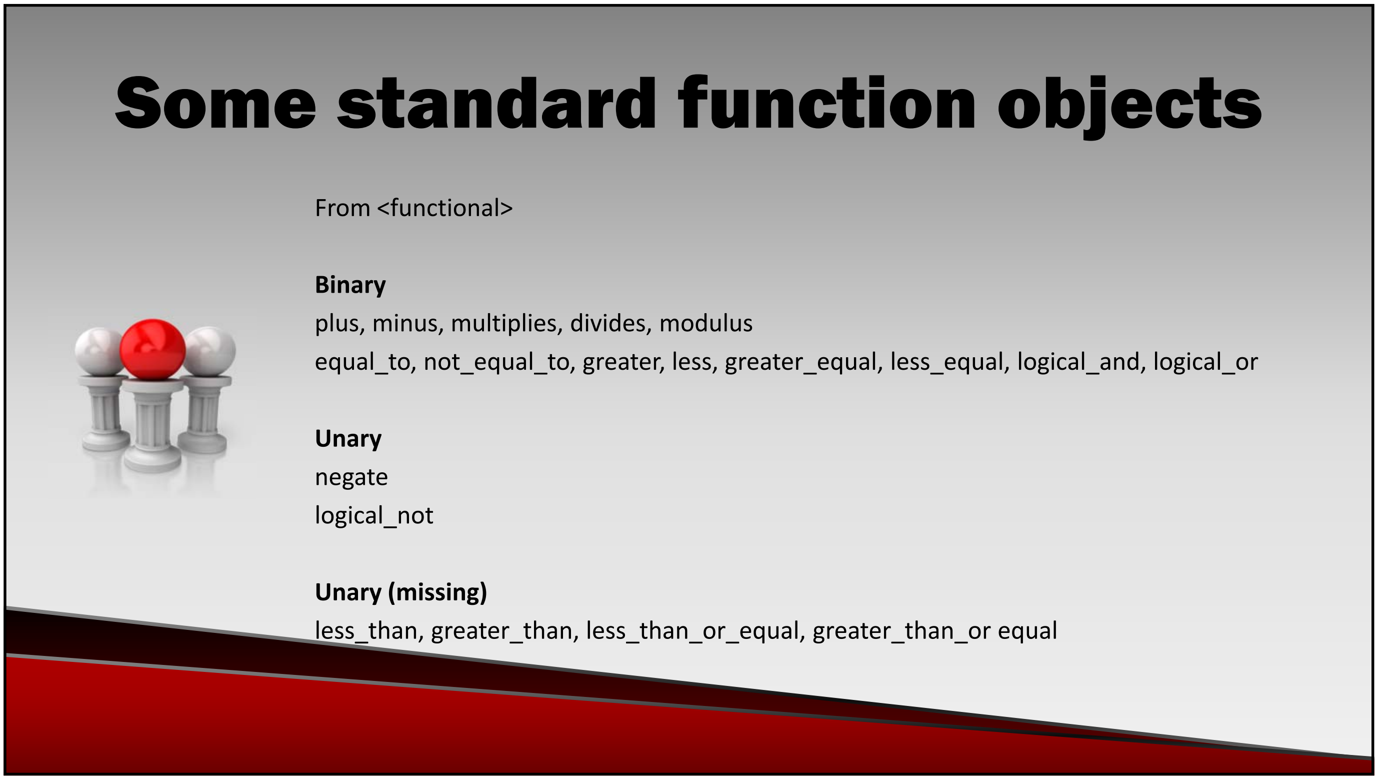
Function Objects

Some algorithms like sort, merge, accumulate can take a function object as argument.

A function object is an object of a template class that has a single member function : the overloaded operator ()

It is also possible to use user-written functions in place of pre-defined function objects

```
#include <list>
#include <functional>
int arr1[]= { 6, 4, 9, 1, 7 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
l1.sort(greater<int>());  // uses function object greater<int>
// for sorting in reverse order l1 = { 9, 7, 6, 4, 1 }
```



Some standard function objects

From <functional>

Binary

plus, minus, multiplies, divides, modulus

equal_to, not_equal_to, greater, less, greater_equal, less_equal, logical_and, logical_or

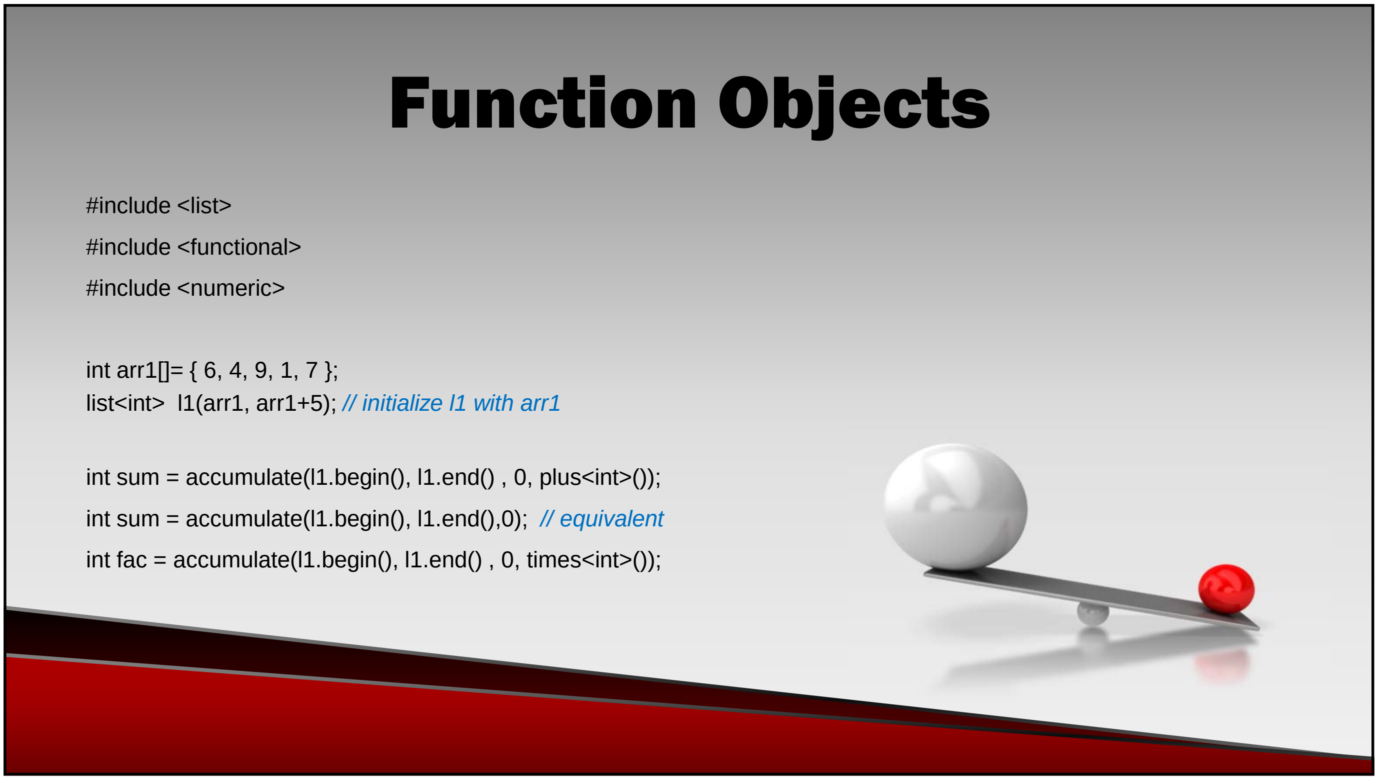
Unary

negate

logical_not

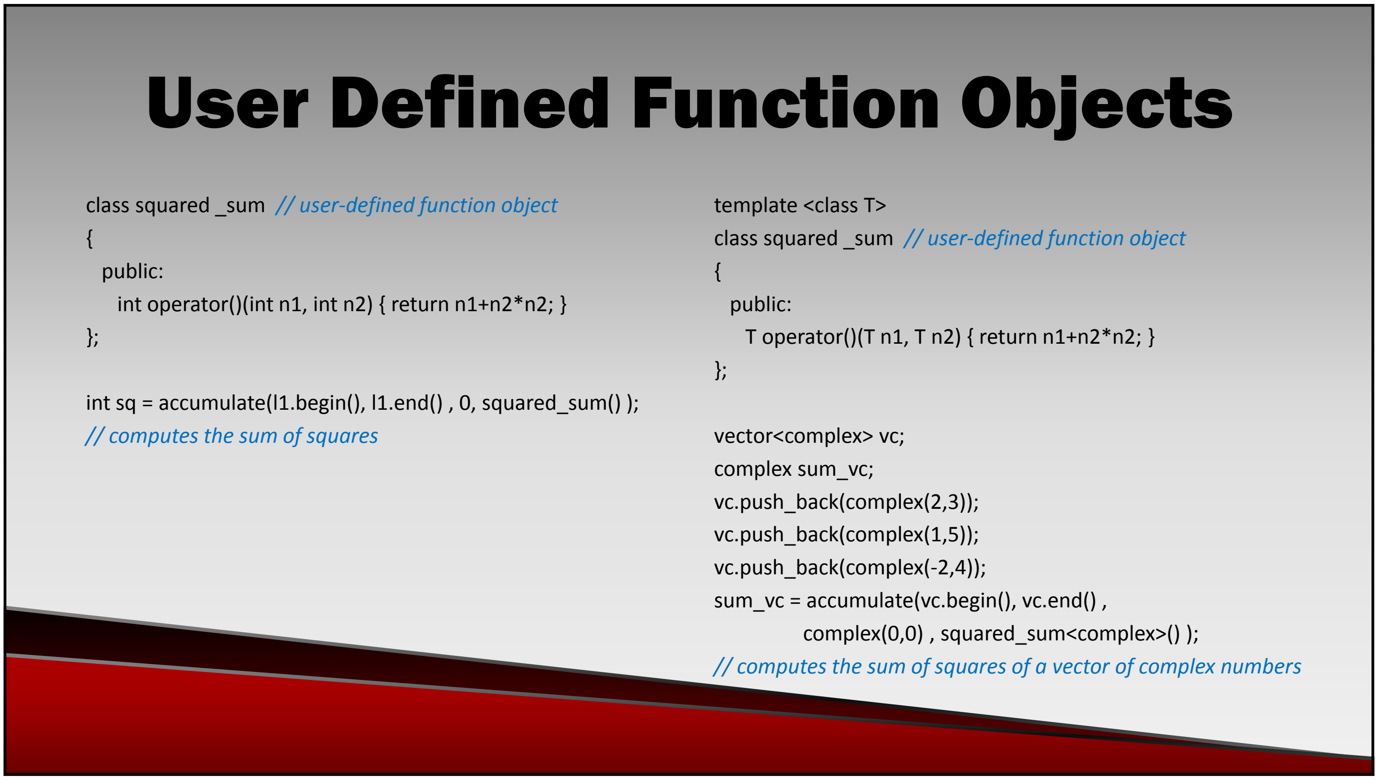
Unary (missing)

less_than, greater_than, less_than_or_equal, greater_than_or equal



Function Objects

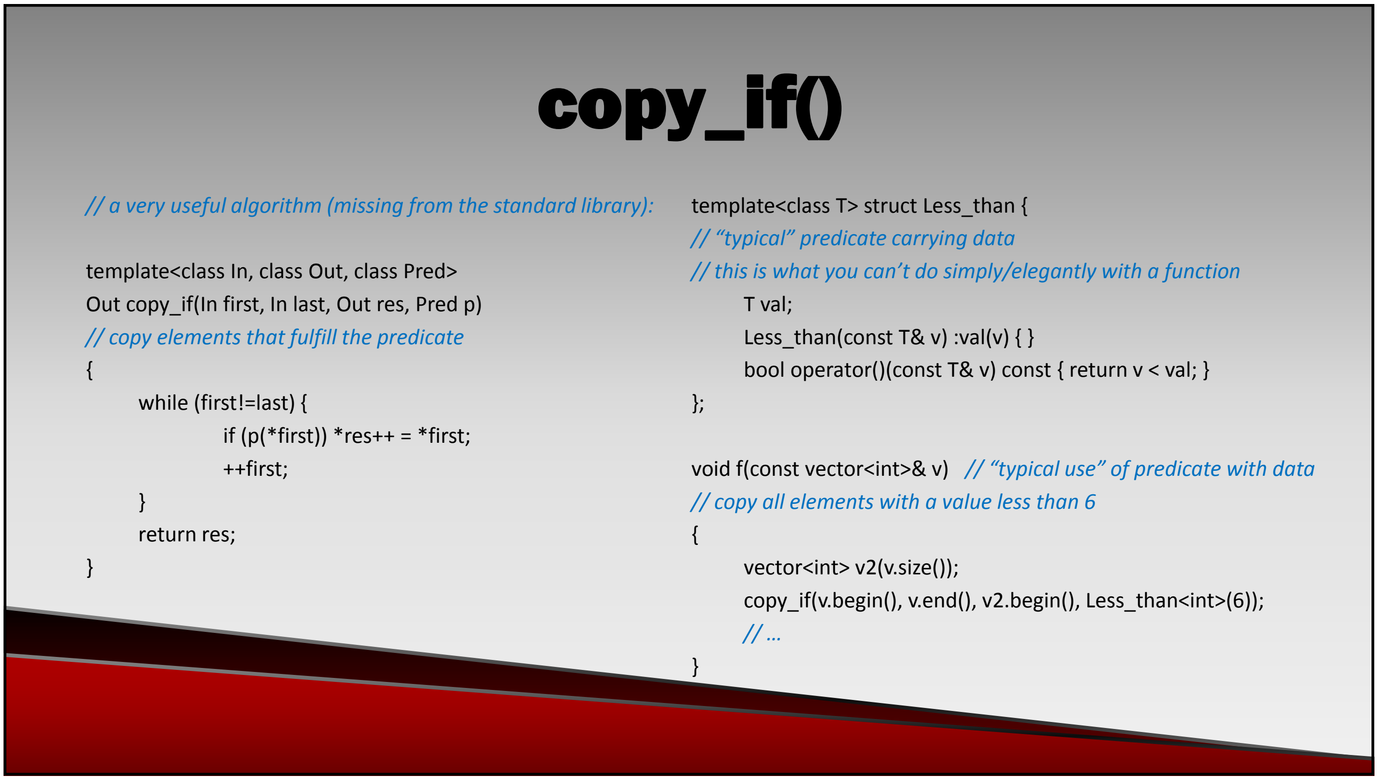
```
#include <list>
#include <functional>
#include <numeric>

int arr1[]= { 6, 4, 9, 1, 7 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1

int sum = accumulate(l1.begin(), l1.end() , 0, plus<int>());
int sum = accumulate(l1.begin(), l1.end(),0);  // equivalent
int fac = accumulate(l1.begin(), l1.end() , 0, times<int>());
```



User Defined Function Objects

```
class squared _sum  // user-defined function object
{
  public:
    int operator()(int n1, int n2) { return n1+n2*n2; }
};

int sq = accumulate(l1.begin(), l1.end() , 0, squared_sum() );
// computes the sum of squares
```

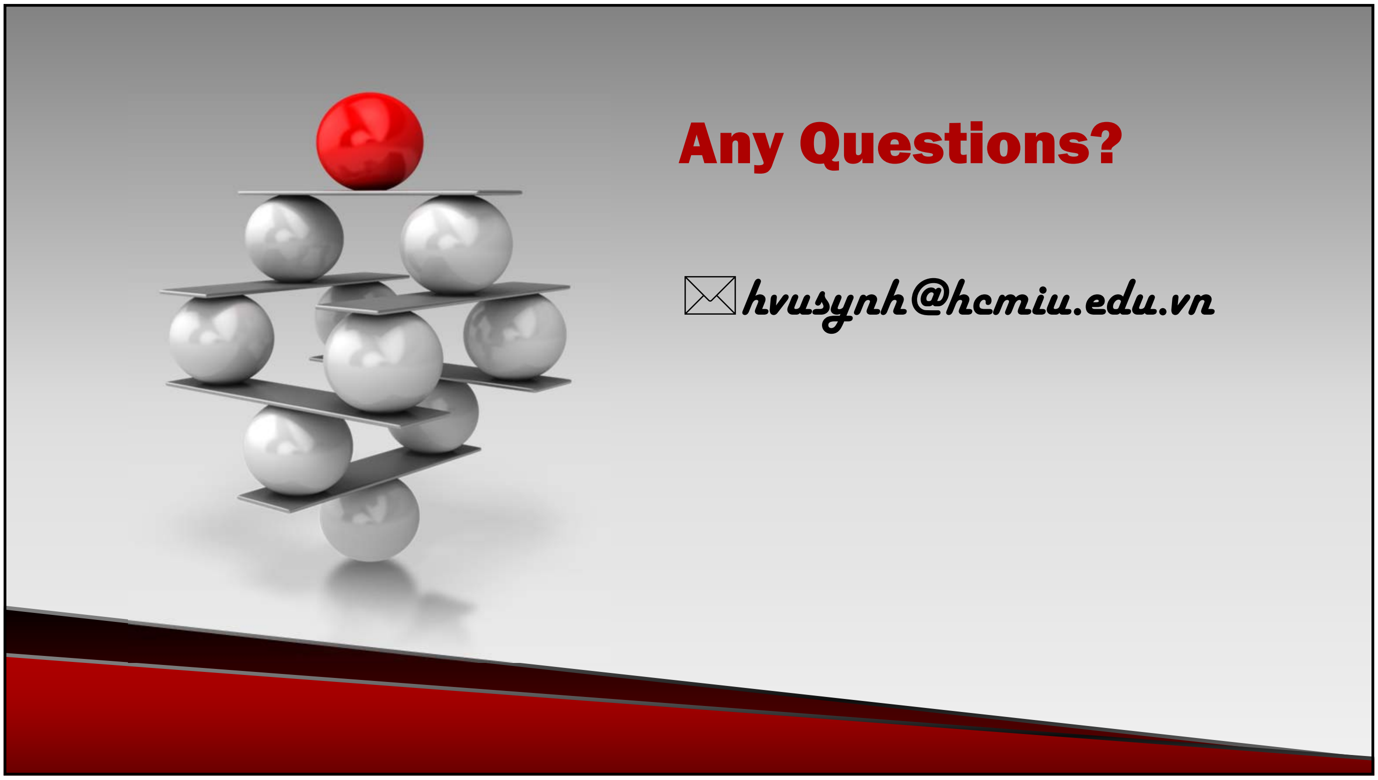
```
template <class T>
class squared _sum  // user-defined function object
{
  public:
    T operator()(T n1, T n2) { return n1+n2*n2; }
};

vector<complex> vc;
complex sum_vc;
vc.push_back(complex(2,3));
vc.push_back(complex(1,5));
vc.push_back(complex(-2,4));
sum_vc = accumulate(vc.begin(), vc.end() ,
             complex(0,0) , squared_sum<complex>() );
// computes the sum of squares of a vector of complex numbers
```



copy_if()

```
// a very useful algorithm (missing from the standard library):

template<class In, class Out, class Pred>
Out copy_if(In first, In last, Out res, Pred p)
// copy elements that fulfill the predicate
{
       while (first!=last) {
                 if (p(*first)) *res++ = *first;
                 ++first;
       }
       return res;
}
```

```
template<class T> struct Less_than {
// "typical" predicate carrying data
// this is what you can't do simply/elegantly with a function
       T val;
       Less_than(const T& v) :val(v) { }
       bool operator()(const T& v) const { return v < val; }
};

void f(const vector<int>& v)   // "typical use" of predicate with data
// copy all elements with a value less than 6
{
       vector<int> v2(v.size());
       copy_if(v.begin(), v.end(), v2.begin(), Less_than<int>(6));
       // …
}
```



Any Questions?

✉ *hvusynh@hcmiu.edu.vn*