# Standard Template Library

# Objectives

- To explain the different templates in the Standard Template Library
- To be able to choose templates from STL to solve different kinds of problems.

# The Standard Template Library

- The STL is a library of containers and algorithms to assist the programmer.
- The STL provides a set of guarantees with regards to performance, which enables programmer to choose a part of the library with confidence.
- There are 5 parts to the STL
  - Iterators; Containers; Adaptors; Algorithms; Function Objects

# Iterators

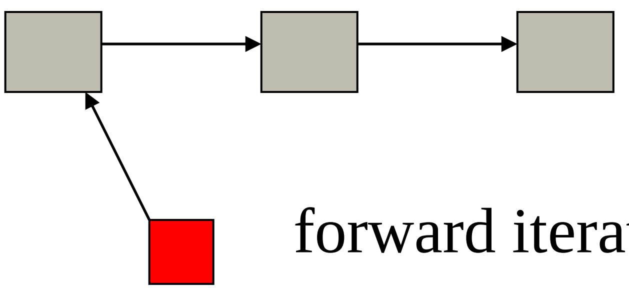
- An iterator is a class that acts in a similar manner to a pointer.  An iterator can be considered a ‘smart pointer’ to an object.
- Iterators can be dereferenced, returning a reference to the object they point to.
- Classes within the STL use iterators as a method of accessing objects stored within containers.
- They are usually parameters to the standard algorithms.

# Iterators

- There are a number of different types of iterators. The important ones are
  - Forward Iterators
  - Reverse Iterators
  - Bidirectional Iterators
  - Random Access Iterators
  - istream and ostream _iterators

# Forward Iterators

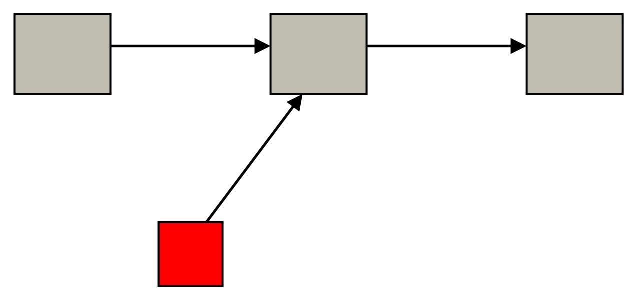
- A forward iterator allows access to a sequence of objects from start to end, in that order only.
- A forward iterator can be incremented using the ++ operator.  This makes the iterator point at the next object in the sequence.



forward iterator called i

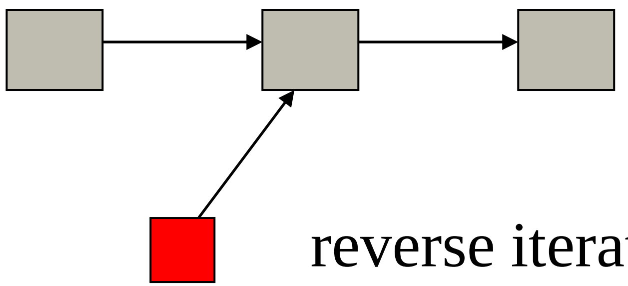
**Code issued:**

```
++i;
```



# Reverse Iterators

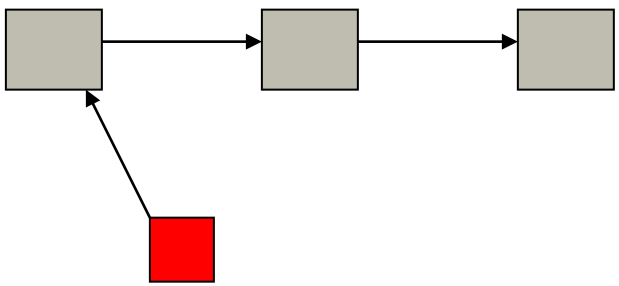
- Reverse iterators work on a sequence in reverse order.  The key point is that **incrementing** a reverse iterator causes you to go **backwards** through the sequence.



reverse iterator called i

**Code issued:**

```
++i;
```



```
#include <iostream>
#include <vector>
#include <string> using namespace std;
main() {
   vector<string> SS;
   SS.push_back("The number is 10");
   SS.push_back("The number is 20");
   SS.push_back("The number is 30");
   cout << "Loop by index:" << endl;
   int ii;
   for(ii=0; ii < SS.size(); ii++) {
            cout << SS[ii] << endl;
   }

cout << endl << "Constant Iterator:" << endl;
   vector<string>::const_iterator cii;
   for(cii=SS.begin(); cii!=SS.end(); cii++) {
            cout << *cii << endl;
   }
```

//iteratorExample.cpp

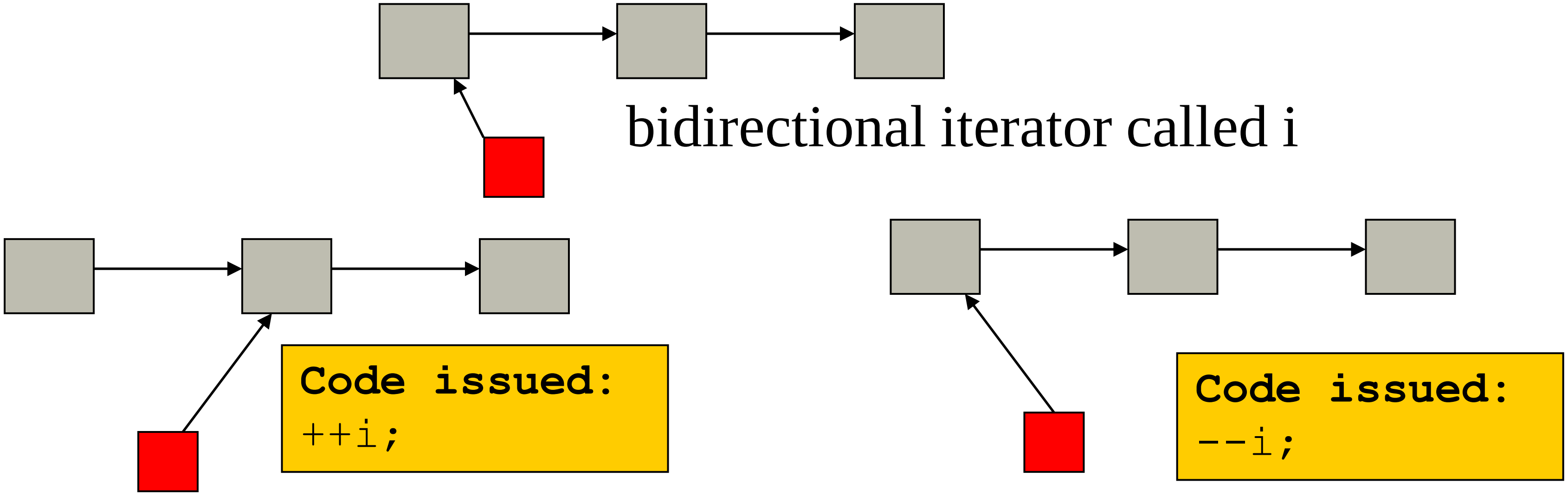
```
cout << endl << "Reverse Iterator:" << endl;
vector<string>::reverse_iterator rii;
for(rii=SS.rbegin(); rii!=SS.rend(); ++rii) {
         cout << *rii << endl;
}

cout << endl << "Sample Output:" << endl;
cout << SS.size() << endl;

cout << SS[2] << endl;
swap(SS[0], SS[2]);
cout << SS[2] << endl; }
```

---

```
Loop by index:
The number is 10
The number is 20
The number is 30
Constant Iterator:
The number is 10
The number is 20
The number is 30
Reverse Iterator:
The number is 30
The number is 20
The number is 10
Sample Output: 3
The number is 30
The number is 10
```

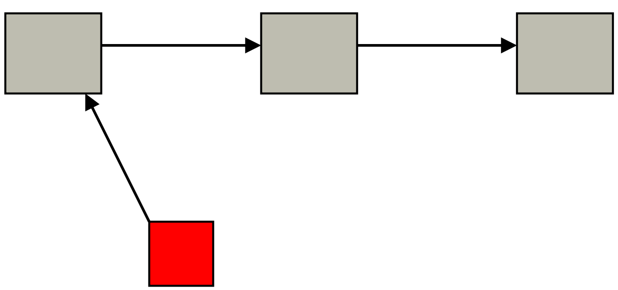
# Bidirectional Iterators

- Bidirectional iterators are like forward iterators that also allow you to go in reverse.
- Incrementing a bidirectional iterator moves forward in the sequence, and decrementing a bidirectional iterator moves backwards in the sequence.



# Random Access Iterators

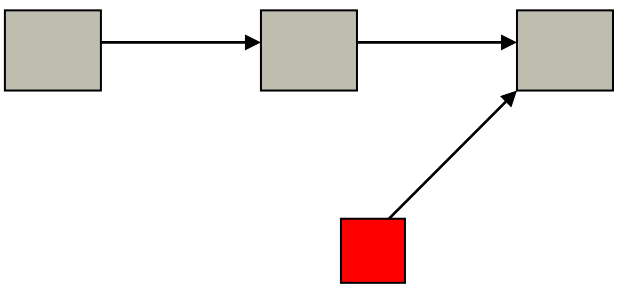
- Random Access Iterators are the closest type of iterator to a pointer.  In fact, pointers can be thought of as random access iterators.
- Random Access Iterators allow incrementing, decrementing and pointer-style arithmetic on them.



random access iterator called i

**Code issued:**

```
i += 2;
```



# istream_ and ostream_ iterators

- sequence iterators for input and output.

# Containers

- The STL provides a set of containers for you to (re)use.  These should be preferred to writing your own versions of these.
- There are two kinds of container
  - Sequential Container
  - Associative Container

# Sequential Containers

- Sequential containers are those where the objects that are located inside them naturally form a sequence, such as arrays and linked lists.
- There are three sequential containers
  - `vector`
  - `list`
  - `deque`

# Vector

- Vectors are arrays that can automatically resize to hold elements. They are intended as drop in replacements for arrays.
- They are located in the `<vector>` header, in the `std` namespace.
- A vector is instantiated as a template:

```
vector<int> vectorOfInts;
```

# Vector Access

- Objects can be retrieved from a vector using the array subscript notation:

```
int tmp = vectorOfInts[0];
```

- Array notation is **not** range checked.  To perform range checking when accessing, use the `at()` member function.

```
int tmp = vectorOfInts.at(-473); // exception
```

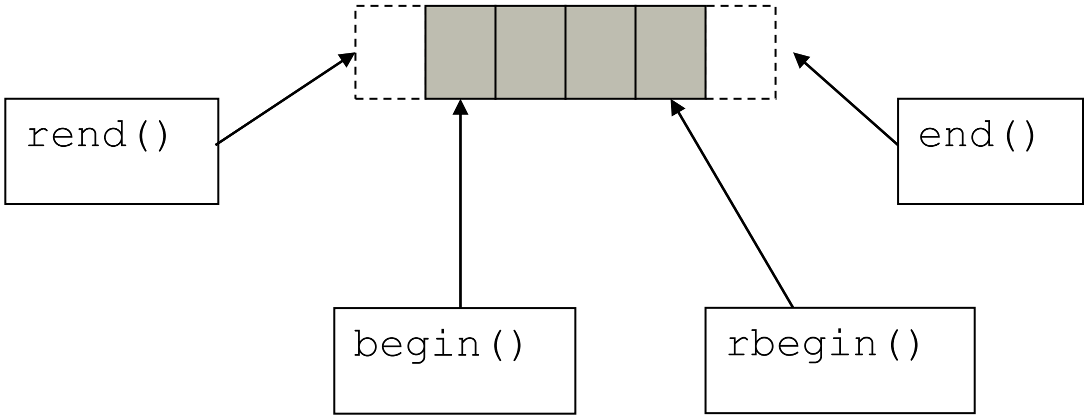
# Vector Access

- Vectors can also be accessed via iterators.
- `begin()` returns an iterator to the first element.
- `end()` returns an iterator that points ‘one past’ the end. This is not valid, so not be dereferenced.
- If `begin()` equals `end()`, the vector is empty.

```
vector<int>::iterator i=vectorOfInts.begin();
*i = 12; // same as vectorOfInts[0] = 12;
*(i+2) = 1; // vectorOfInts[2] = 1;
if (i == vectorOfInts.end())
 cout << “Iterator can’t be de’ref\n”;
```

# Vector Access

- Vectors provide `rbegin()` and `rend()`, which return reverse iterators. `rbegin()` points to the last element, and `rend()` to 'one-before' the first element.



# Vector Insertion

- Objects are inserted into a vector using the `push_back()` or `insert()` member functions.
- `push_back()` inserts at the end of the vector, and `insert()` inserts after an iterator.
- If order doesn't matter, favour `push_back()` over `insert()`.

```
vectorOfInts.push_back(12);
vectorOfInts.insert(v.begin(), 1);
```

# Vector size and capacity

- We can check the current number of objects stored in a vector using the `size()` member function.
- When inserting, there's a chance the vector might need to resize. It will resize when `size()` is equal to `capacity()`.
- You can increase the capacity of the vector using the `reserve()` member function.
- You can resize the vector by using the `resize()` member function. This will create new objects and call their default constructors.

# Deleting from a vector

- Deleting from a vector is achieved using the `erase()` function.
- Note that this moves the object to the end of the vector, and decrements `size()`.
- Erase accepts an iterator to the object to be erased.

```
vector<string> v;
v.erase(v.begin());
```

# List

- A list container is a doubly linked list of objects.
- It is located in the <list> header, in the std namespace.
- It is instantiated in a similar manner to a vector:

```
std::list<int> listOfInts;
```

# List access

- All list access is done through iterators.  You cannot use subscripted operators.
- A list provides the programmer with bi-directional iterators by default.
- Lists provide `begin()`, `end()`, `rbegin()` and `rend()` functions like a vector.

# List insertion

- A list provides the `push_back` and `push_front` member functions.
- It also provides the `insert` member function.
- These operator in the same was as for a vector.
- We can find out how many objects are in a list using the `size()` member function.

# List deletion

- Deleting from a list is performed by the `remove()` or `remove_if()` member functions.
- The `remove()` function removes all objects that match its argument.

```
list<int> l;
l.remove(1);
```

- The `remove_if()` function uses a **predicate**, which will be explained in the section on function objects.

# List example

```
// Standard Template Library example

#include <iostream>
#include <list>
using namespace std;

// Simple example uses type int

main()
{
   list<int> L;
   L.push_back(0);              // Insert a new element at the end
   L.push_front(0);             // Insert a new element at the beginning
   L.insert(++L.begin(),2);     // Insert "2" before position of first argument
                                // (Place before second argument)
   L.push_back(5);
   L.push_back(6);

   list<int>::iterator i;

   for(i=L.begin(); i != L.end(); ++i) cout << *i << " ";
   cout << endl;
   return 0;
}
```

listExample.cpp

Output: 0 2 0 5 6

# Sorting a list

- A list can be sorted by calling the `sort()` member function.
- The sort member function uses (by default) a less-than comparison.
- In order to support this comparison, the objects stored in the list must have an `operator<` defined.
- Other comparisons can be defined – we'll look at this when we look at function objects.

# Deque & Insertion

- A `deque` is a double-ended queue.
- It supports efficient insertion and deletion at the front and back of the container, like a list, but allows subscripted access like a vector.
- It supports insertion in the front, back and middle.
- Similar to vectors, inserting in the middle is costly.
- Inserting at the front (`push_front()`), or at the back (`push_back()`) are efficient.

# Deque deletion

- Deleting from a deque is similar to deleting from a vector.
- You can use the `erase()` member function that accepts an iterator
- To completely empty a deque, use the `clear()` member function.

# Associative containers

- There are four associative containers. These are
  - `map`
  - `set`
  - `multimap`
  - `multiset`
- These containers have a ‘key’ and some data. The key is used to access the data.

# Map declaration and access

- A map can be declared by including the <map> header, and specifying the key and data type:

```
std::map<std::string, Game> mapOfGames;
```

- Data within the map can be accessed via the key and subscripting operators:

```
Game g;

mapOfGames["Game 1"] = g;

Game g2 = mapOfGames["Game 1"];
```

# Map iterators

- Dereferencing a map iterator does not give you the underlying object. Instead, you get a `pair` containing the key and the data.
- A `pair` is a template that contains two public members, `first` and `second`.
- When we dereference a map iterator, `first` contains the key, and `second` contains the data.

# Map iterators

```
map<string, string> someMap;
// ..
map<string, string>::iterator i = someMap.begin();
// ..
cout << "Key is: " << i->first << ", data is: " << i->second << endl;
```

- **A point to remember**: When your map object contains pointers as the data, you will need to dereference `i->second` as well!

# Map insertion

- There are two ways to insert into a map:

```
someMap[key] = data;

someMap.insert(make_pair(key, data));
```

- I prefer the second, for two reasons:
  - It makes it obvious we are inserting a pair into the map.
  - It is more efficient than the first method as it avoids a temporary object.
- **Remember!** We can only have one piece of data for one particular key value.

# Map deletion

- Individual elements can be deleted using the `erase()` member function.

```
someMap.erase(key);
```

- The entire map can be emptied using the `clear()` member function.

```
someMap.clear();
```

- Neither of these functions calls `delete()` on any stored pointers.

# Map example

Testmap1.cpp

```
#include <string.h>
#include <iostream>
#include <map>
#include <utility>

using namespace std;

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

   // 1) Assignment using array index notation
   Employees[5234] = "Mike C.";
   Employees[3374] = "Charlie M.";
   Employees[1923] = "David D.";
   Employees[7582] = "John A.";
   Employees[5328] = "Peter Q.";

   cout << "Employees[3374]=" << Employees[3374] << endl << endl;

   cout << "Map size: " << Employees.size() << endl;

   for( map<int,string>::iterator ii=Employees.begin(); ii!=Employees.end(); ++ii)
   {
       cout << (*ii).first << ": " << (*ii).second << endl;
   }
}
```

Compile: `g++ testMap.cpp`
Run: `./a.out`

```
Employees[3374]=Charlie M.

Map size: 5
1923: David D.
3374: Charlie M.
5234: Mike C.
5328: Peter Q.
7582: John A.
```

# Map example

Testmap2.cpp

```
#include <string.h>
#include <iostream>
#include <map>
#include <utility>

using namespace std;

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

   // Examples of assigning Map container contents

   // 1) Assignment using array index notation
   Employees["Mike C."] = 5234;
   Employees["Charlie M."] = 3374;

   // 2) Assignment using member function insert() and STL pair
   Employees.insert(std::pair<string,int>("David D.",1923));

   // 3) Assignment using member function insert() and "value_type()"
   Employees.insert(map<string,int>::value_type("John A.",7582));

   // 4) Assignment using member function insert() and "make_pair()"
   Employees.insert(std::make_pair("Peter Q.",5328));

   cout << "Map size: " << Employees.size() << endl;

   for( map<string, int>::iterator ii=Employees.begin(); ii!=Employees.end(); ++ii)
   {
       cout << (*ii).first << ": " << (*ii).second << endl;
   }
}
```

Compile: `g++ testMap.cpp`
Run: `./a.out`

```
Map size: 5
Charlie M.: 3374
David D.: 1923
John A.: 7582
Mike C.: 5234
Peter Q.: 5328
```

# “Sorting” maps

- Maps are sorted by default on the key value, using `operator<`.
- Therefore, the key type must be comparable using `operator<`.
- It is possible to sort using different operations – see the later section on function objects.

# Sets

- A **set** is a special associative container where the object itself is the key to the container.
- Set does not allow duplicates
- Here, the key is not used to access the data, but can be used to test if an object exists within a set.
- The key is also used to sort the set.  Therefore, a set is similar to a sorted list.

# Set declaration

- Sets are found in the `<set>` header, and the `std` namespace, and are declared as follows:

```
using std::set;
set<string> setOfStrings;
```

- Sets are sorted by default using `operator<`. Ensure that `operator<` is defined for the type you are storing.

# Set insertion and deletion

- Inserting into a set is simple:

```
setOfStrings.insert(“Hello, World”);
```

- Deleting from a set is also simple:

```
setOfStrings.erase(“Hello, World”);
```

# Set access

- Sets are usually accessed through their iterators. This provides sorted access.
- Set iterators are **const iterators**, which does not allow to change the underlying object. You can only call **const member functions**.

```
set<string>::iterator iter = setOfStrings.begin()
iter->someConstMemberFunction();
```

- You can test for the presence of an object by using find()

```
if(setOfStrings.find("Hello")!=setOfStrings.end())
 {   // object exists in set }
```

# Multimaps and multisets

- Multimaps and multisets are similar to maps and sets, except that they allow more than one object with the same key.
- This causes potential problems when we try to find an object with multiple instances. Which one do we retrieve?

# Multimap Access

- To return all objects with the same key, we use the `equal_range()` function.
- It returns a `pair` of iterators, marking the start and end of the range of objects with the same key.
- To enhance readability, it is common to `typedef` the `pair` types. This is shown on the next slide.

# Multimap Access

```
typedef multimap<string,string>::iterator mmIter;

// ..
multimap<string, string> mMapOfStrings;

// ..
pair<mmIter, mmIter> range;
range = mMapOfStrings.equal_range("key");

mmIter i = range.first;
while (i != range.second) {
   // .. do something with iterators
   ++i;
}
```

# Sequence Adaptors

- Some containers are created from other containers.  These are called the **sequence adaptors**.
- The adaptors do not provide iterators – you should use the interfaces provided to access data.

# Stack

- The first sequence adaptor is the **stack**. It is found in the `<stack>` header.
- It replaces the `back()`, `push_back()` and `pop_back()` with the more common stack terminology of `top()`, `push()` and `pop()`.
- A stack is declared like other sequence container:
  - `stack<int> stackOfInts;`
- It is also possible to specify the underlying container – by default, a stack uses a `deque`.
  - `stack<int,vector<int> > stackOfInts;`

# Queue

- A queue allows insertion at the back, and removal of elements from the front.
- It is declared in the `<queue>` header.
- Provides `push()`, `pop()`,`front()`, `back()` funcs.
- A queue is declared in the same way we expect:
  - `queue<string> messageQueue;`
- We can change the underlying container in the same was as for a stack, but the sequence must support `pop_front()` and `push_back()`, ruling out a vector.

# Priority Queues

- Similar to a queue, but items stored in the queue have a priority.
- Items of the same priority are served on the first in, first out principle of the queue, but higher priority items are served before lower priority ones.
- Priority is defined by overriding `operator<`.
- Priority queues are defined in `<queue>`.
- They provide the same operations as for a queue.

# Other containers

- The STL provides other containers.
- These containers do not have a robust interface that we can use, but still act similar to containers.
- They are `basic_string`, `valarray`, `bitset`.
- We have already seen `basic_string` in strings.
- A `valarray` is a vector optimised for numerical operations.
- A `bitset` is a container representing a set of bits.

# Algorithms

- C++ provides more than 50 standard algorithms in `<algorithm>` for you to use on sequences.
- A sequence in this sense is a range of iterators.
- An algorithm works on different types of iterators.
- The standard algorithms are provided to prevent you writing your own versions.
- If you find yourself going to write a sorting routine, use the standard `sort()` function.
- Traversal of iterators is achieved with algorithms.

# Modifying and Nonmodifying Algorithms

- Some algorithms are meant to traverse a range of data, but perform no action or make no change, such as `find()`.
- These are called **non-modifying algorithms**.
- Others, such as `transform()`, are meant to change the underlying range. These are called **modifying algorithms**.

# Function Objects

- A function object is any class that overloads the `()` operator.
- The `()` operator is the **function dereference** or **function call** operator.
- A function name is a type of function pointer. How do you dereference that pointer?

```
functionName(); // () dereferences functionName.
```

# Function Objects

- This enables us to create an object of some type, and use it as a function:

```
class FunctionObject { };
// . .
FunctionObject fo;
fo();
```

# Predicates

- Predicates are special kinds of function objects that return a `bool` type.
- A **unary predicate** is a function object that accepts a single argument, and returns a `bool`.
- A **binary predicate** is a function object that accepts two arguments.

# Predicated Algorithms

- Predicates are used extensively with standard algorithms, performing the role of ‘Yes/No’ answers for objects.
- For example, the `find_if()` algorithm can be used to see find an object matching some criteria.
- The criteria is encoded in the predicate, and the predicate returns true if the object passed matches.

# Predicated Algorithms

```
class StartsWithHello {
public:
   bool operator() (const std::string& obj) const
  {
      return (obj.find("Hello") == 0);
   }
};
//..
vector<string> v;
vector<string>::iterator I =
      find_if(v.begin(), v.end(),
 StartsWithHello());
```

# count_if

- The algorithm `count_if()` can be used to count the number of objects in a sequence that match a predicate.

```
int cnt = count_if(v.begin(), v.end(),
 StartsWithHello());
```

# Copy

- The standard algorithm `copy` is used to copy one sequence into another.
- Care must be taken to avoid overflowing the target. In the following example, `v2` must be the same size or larger than `v1`.

```
vector<int> v1, v2;
copy(v1.begin(), v1.end(), v2.begin());
```

# back_inserter

- It is possible to use copy to insert into empty containers – to do so, you need to use a **`back_inserter`** predicate.

```
copy(v1.begin(), v1.end(), back_inserter<v2>);
```

- A `back_inserter` ensures that the container grows, and objects copied are inserted at the back.

# Merge

- Merge is used to join two range of iterators together.
- The result buffer used must contain sufficient space for both ranges.

# Transform

- Transform exists to apply the result of a function object to a range of iterators.
- The function object's `operator()` should accept a single const parameter of the same type that the iterator points to.  Eg, for `string`, a `char`.
- It should return an object of the same type.

# Transform

- The function syntax is:

```
void transform(InIter start, InIter end,
     OutIter out, FunctionObject);
```

- Parameters `start` and `end` specify the range of iterators, and `out` specifies where the output should be written to.
- You must ensure that there is sufficient space in `out` to hold the output.
- It is possible for `out` to be the same as `start` (self-modification).

# Transform

```
class ToUpper {
   public:
      char operator()(const char c) {
         return std::toupper(c);
      }
};
// ..TransformL8.cpp

std::string s = "aBcDeFg";
transform(s.begin(), s.end(), s.begin(),
 ToUpper());
```

# Find

- The `find` algorithm is provided for those containers that do not provide their own member function.
- It performs a linear search, searching for an object.

```
vector<int> v;
find(v.begin(), v.end(), 2);
```

# Sort

- `sort` is used to sort a series of random access iterators.
- By default, it uses `operator<` to sort, so this must be defined for the objects being sorted.

```
vector<int> v;
sort(v.begin(), v.end());
```

- You can also sort by different criteria via a binary predicate.
- This must follow **Strict Weak Ordering**. In SWO, if object a is less than object b, and object b is less than object c, then object a must be less than object c.
- Or, if a < b, and b < c, then a < c.

# Sort

```
class SortByPoints {
   public:
      bool operator()(const Game& lhs, const Game&
            rhs) const {
         return lhs.getPoints() < rhs.getPoints();
      }
};

vector<Game> games;
// ..
sort(games.begin(), games.end(), SortByPoints());
```

# Stable sort

- There are two other variants of `sort`.
- By default, `sort` provides an $O(n \log n)$ guarantee in the average case, but $O(n^2)$ in the worst case.
- You can use `stable_sort` to sort with a guaranteed $O(n \log n \log n)$ worst and average case.

# partial_sort

- You can also use `partial_sort` to only sort a subset of the data.
- This might be useful if you need the top ten objects, but don't care about the rest.  You only sort what's necessary.

```
partial_sort(v.begin(), v.begin()+10,
            v.end());
```

- This sorts from `v.begin()` to `v.begin()+10`.

# Common Question

- **Question:** How can a map/set find an object using only `operator<`?
- **Answer:** By applying it the other way around:

```
if (a < b) {

   // not equal, return false

}

else if (b < a) {
 // not equal, return false

}

// neither less than, nor greater than, so equal

// return true;
```

# Summary

- The STL provides a large range of containers and algorithms for you to use.
- These are functional, efficient, and relatively easy to use.
- Use them wherever appropriate.  Don't write your own.