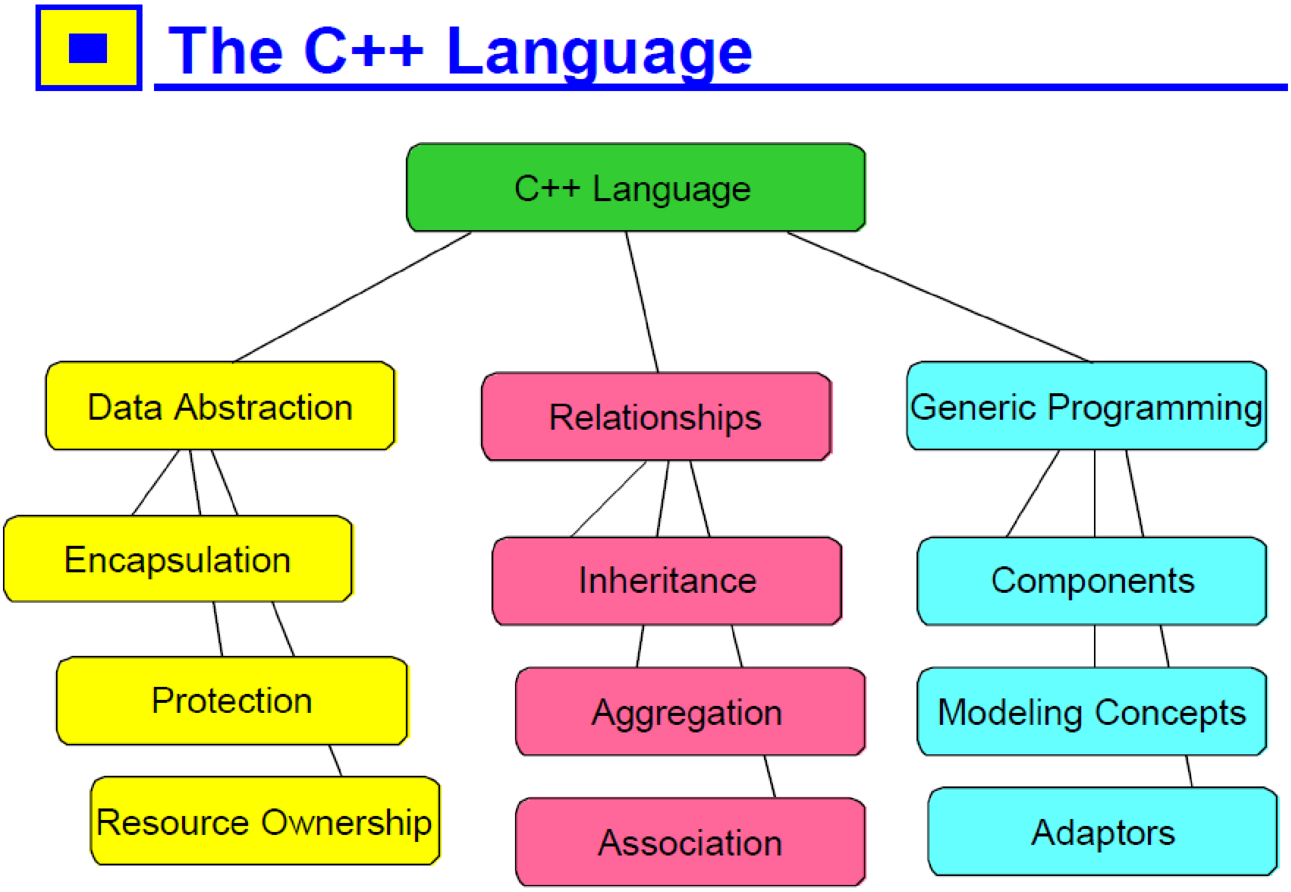
# Table of Contents

C++ Overview....3
C++ with Standard Template Library....4
History....5
Overview - STL....6
Overview containers, iterators, algorithms....7
Iterators Preview....8
Iterator Sort Example....9
Iterator....10
Iterator categories....10
Iterator Classes....12
Primitives....12
Adaptors....12
Stream Iterators....12
Functions Adaptors....13
Functions Operations....13
Range....13
Non-member operators....13
Iterators Example....15
Iterator adaptors....16
istream_iterator....16
Iterator Adaptor Example....17
Templates....18
Defining a Template....19
Function Template Definition Example....20
Function templates Example....21
Function Template Variations Example....22
Class templates....24
Class Template Example....25
introducing the vector template....26
STL components....27
Containers....27
Hierarchy....28
Sequences....28
Vectors....28
Lists....28
Deques....28
Stacks and queues....29
Associative containers....29
Sets....29
Multisets....29
Maps....29
Multimaps....29
priority queue,....29
bit vector,....29
queue....29
Example using container algorithms....30
Container Summary Table....31
Simple Containers....31
Container Adaptors....32

Containers – Associative and other types....33
Algorithms....34
Algorithms library....34
Non-modifying sequence operations....34
Modifying sequence operations....35
Partitioning operations....36
Sorting operations (on sorted ranges)....36
Binary search operations (on sorted ranges)....37
Set operations (on sorted ranges)....37
Heap operations....38
Minimum/maximum operations....38
Numeric operations....39
C library....39
Functors....40
C++ Standard Library....41
C++ Standard Library....42

## C++ Overview



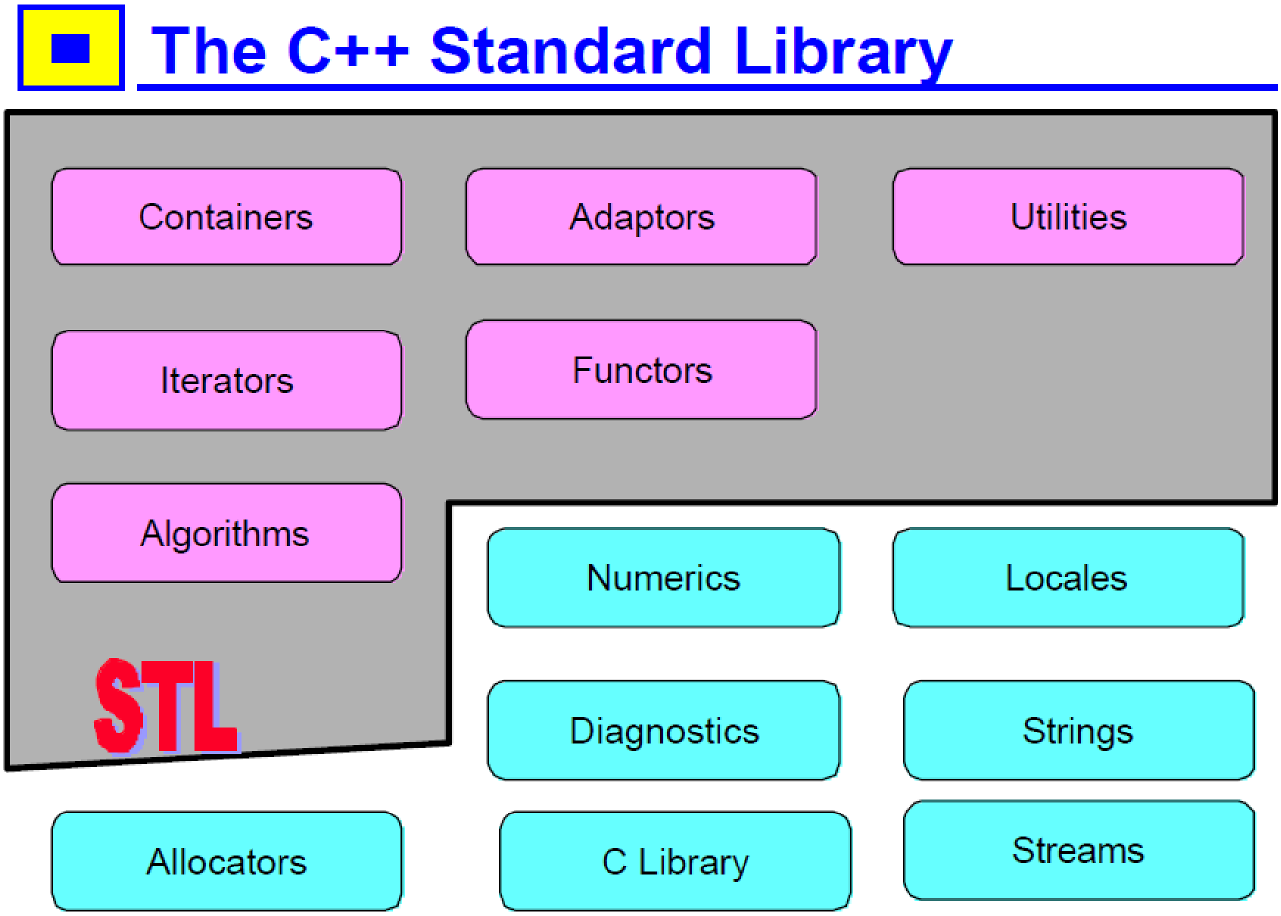

cs.brown.edu

Data abstraction separates the interface (how the object is used) from the implementation (how it works Inside).

Relationships between objects and types are fundamental in object-oriented programming; inheritance defines a relationship between types corresponding to “is-a”, aggregation corresponds to “has-a”, and associations are more general relationships

Generic programming is a notion that has gained a lot of popularity in the last few years; originally used in Ada, the Standard Template Library (roughly a subset of the standard C++ library) made it popular in C++.

## C++ with Standard Template Library




Containers are objects that contain other objects, e.g., vector, set, map.
2. Iterators represent locations in a container. Each container has its own iterator type.
3. Algorithms are operations on containers, e.g., find, sort, random_shuffle.
4. Functors are operations on objects, e.g., less, plus.
5. Adaptors are objects that change an interface, e.g., not1. (That's a one at the end, not an ell; there is also a not2.)
6. Utilities are components such as pairs, operations like comparison, etc. In the ANSI standard, allocators are included in the utilities section.
7. Diagnostics are provided to deal with exceptions.
8. Locales facilitate internationalization.
9. Numerics are container types that are optimized for speed, less general than containers, e.g., valarray, complex.
10. Strings replace C’s character arrays.
11. Streams are used for input and output.
12. Allocators customize memory allocation, e.g., malloc_alloc.

## History

The architecture of STL is largely the creation of one person, Alexander Stepanov.

In 1979 he began working out his initial ideas of generic programming and exploring their potential for revolutionizing software development. Although David Musser had developed and advocated some aspects of generic programming already by year 1971, it was limited to a rather specialized area of software development (computer algebra).

Stepanov recognized the full potential for generic programming and persuaded his then-colleagues at General Electric Research and Development (including, primarily, David Musser and Deepak Kapur) that generic programming should be pursued as a comprehensive basis for software development. At the time there was no real support in any programming language for generic programming.

The first major language to provide such support was Ada, with its generic units feature. By 1987 Stepanov and Musser had developed and published an Ada library for list processing that embodied the results of much of their research on generic programming.

The reason for turning to C++, which Stepanov was the C/C++ model of computation which allows very flexible access to storage via pointers is crucial to achieving generality without losing efficiency.

Stepanov experimented with many architectural and algorithm formulations, first in C and later in C++. Musser collaborated in this research and in 1992 Meng Lee joined Stepanov's project at HP and became a major contributor.

Andrew Koenig of Bell Labs had not become aware of the work and asked Stepanov to present the main ideas at a November 1993 meeting of the ANSI/ISO committee for C++ standardization. The committee's response was overwhelmingly favorable and led to a request from Koenig for a formal proposal in time for the March 1994 meeting.

Subsequently, the Stepanov and Lee document 17 was incorporated into the ANSI/ISO C++ draft standard (1, parts of clauses 17 through 27). It also influenced other parts of the C++ Standard Library, such as the string facilities, and some of the previously adopted standards in those areas were revised accordingly

The prospects for early widespread dissemination of STL were considerably improved with Hewlett-Packard's decision to make its implementation freely available on the Internet in August 1994. This implementation, developed by Stepanov, Lee, and Musser during the standardization process, became the basis of many implementations offered by compiler and library vendors today.

# Overview - STL

The STL provides a ready-made set of common classes for C++,:

- containers and associative arrays,
- that can be used with any built-in type or with any user-defined type
- supports operations such as copying and assignment

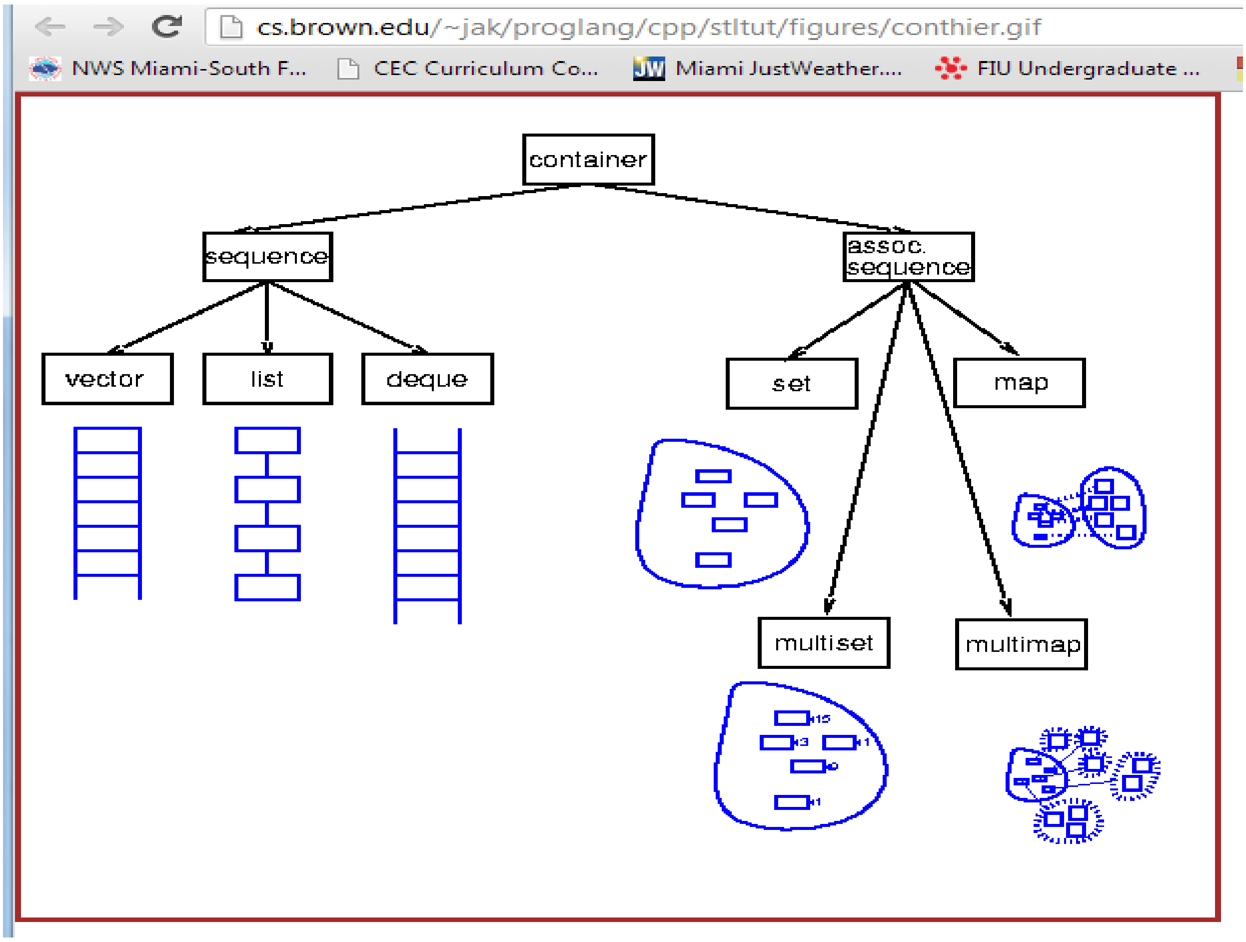
STL algorithms are independent of containers - significantly reduces the complexity of the library.
The STL achieves its results through the use of templates
This approach provides

- compile-time polymorphism
- is often more efficient than traditional run-time polymorphism.

The STL was created as the first library of generic algorithms and data structures for C++, with four ideas in mind:

1. generic programming,
2. abstractness without loss of efficiency,
3. the Von Neumann computation model,
4. and value semantics.

http://www.thefullwiki.org/Standard_Template_Library




http://cs.brown.edu/~jak/proglang/cpp/stltut/tut.html

## Overview containers, iterators, algorithms

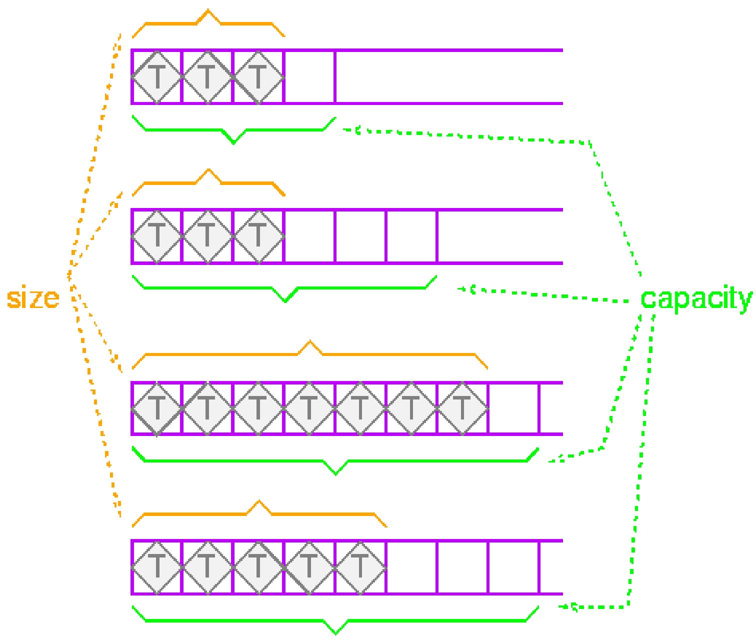
STL provides a number of container types, representing objects that contain other objects.

One of these containers is a class called **vector** that behaves like an array, but can grow itself as necessary. One of the operations on vector is **push_back**, which pushes an element onto the end of the vector (growing it by one).

A vector contains a block of **contiguous** initialized elements -- if element index k has been initialized, then so have all the ones with indices less than k.

A vector can be **presized**, supplying the size at construction, and you can ask a vector how many elements it has with **size**. This is the **logical** number of elements -- the number of elements up to the highest-indexed one you have used. There is also a notion of **capacity** -- the number of elements the vector can hold before reallocating.

http://cs.brown.edu/~jak/proglang/cpp/stltut/tut.html



## Iterators Preview

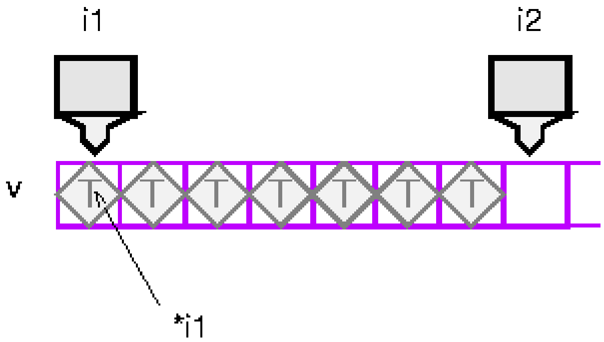
Iterators provide a way of specifying a position in a container.

An iterator can be **incremented** or **dereferenced**, and two iterators can be **compared**. There is a special iterator value called "past-the-end".

You can ask a vector for an iterator that points to the first element with the message **begin**. You can get a past-the-end iterator with the message **end**. The code

```
vector<int> v;
// add some integers to v
vector::iterator i1 = v.begin();
vector::iterator i2 = v.end();
```

will create two iterators like this:




Operations like sort take two iterators to specify the source range.

To get the source elements, they increment and dereference the first iterator until it is equal to the second iterator. Note that this is a semi-open range: it includes the start but not the end.

Two vector iterators compare equal if they refer to the same element of the same vector.

## Iterator Sort Example

Putting this together, here is the new program:

```
#include <string.h>
#include <algo.h>
#include <vector.h>
#include <stdlib.h>
#include <iostream.h>

main ()
{
  vector<int> v;  // create an empty vector of integers
  int input;
  while (cin >> input)    // while not end of file
    v.push_back (input);  // append to vector

  // sort takes two random iterators, and sorts the elements between
  // them.  As is always the case in STL, this includes the value
  // referred to by first but not the one referred to by last; indeed,
  // this is often the past-the-end value, and is therefore not
  // dereferenceable.
  sort(v.begin(), v.end());

  int n = v.size();
  for (int i = 0; i < n; i++)
    cout << v[i] << "\n";
}
```

# Iterator

<iterator>

Iterator definitions

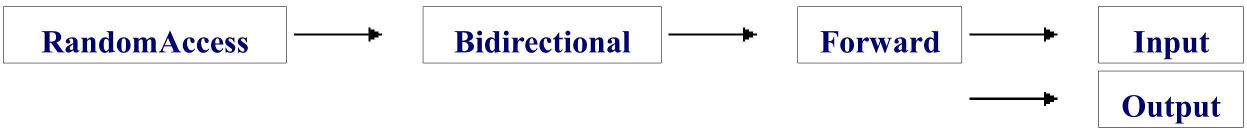
In C++, an iterator is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (at least, the increment (++) and dereference (*) operators).

The most obvious form of iterator is a *pointer*: A pointer can point to elements in an array, and can iterate through them using the increment operator (++). But other forms of iterators exist. For example, each container type (such as a vector) has a specific *iterator* type designed to iterate through its elements in an efficient way.

Notice that while a pointer is a form of iterator, not all iterators have the same functionality a pointer has; To distinguish between the requirements an iterator shall have for a specific algorithm, five different *iterator categories*exist:

## *Iterator categories*

Iterators are classified in five categories depending on the functionality they implement:




In this graph, each iterator category implements the functionalities of all categories to its right:

Input and output iterators are the most limited types of iterators, specialized in performing only sequential input or output operations.

Forward iterators have all the functionality of input and output iterators, although they are limited to one direction in which to iterate through a range.

Bidirectional iterators can be iterated through in both directions. All standard containers support at least bidirectional iterators types.

Random access iterators implement all the functionalities of bidirectional iterators, plus, they have the ability to access ranges non-sequentially: offsets can be directly applied to these iterators without iterating through all the elements in between. This provides these iterators with the same functionality as standard pointers (pointers are iterators of this category).

http://www.cplusplus.com/reference/iterator/

The characteristics of each category of iterators are:

<table>
<tr><th colspan="4">category</th><th>characteristic</th><th>valid expressions</th></tr>
<tr><td colspan="4" rowspan="2">all categories</td><td>Can be copied and copy-constructed</td><td><code>X b(a);</code><br><code>b = a;</code></td></tr>
<tr><td>Can be incremented</td><td><code>++a</code><br><code>a++</code><br><code>*a++</code></td></tr>
<tr><td rowspan="9">Random Access</td><td rowspan="5">Bidirectional</td><td rowspan="4">Forward</td><td rowspan="2">Input</td><td>Accepts equality/inequality comparisons</td><td><code>a == b</code><br><code>a != b</code></td></tr>
<tr><td>Can be dereferenced as an rvalue</td><td><code>*a</code><br><code>a->m</code></td></tr>
<tr><td>Output</td><td>Can be dereferenced to be the left side of an assignment operation</td><td><code>*a = t</code><br><code>*a++ = t</code></td></tr>
<tr><td></td><td>Can be default-constructed</td><td><code>X a;</code><br><code>X()</code></td></tr>
<tr><td colspan="2"></td><td>Can be decremented</td><td><code>--a</code><br><code>a--</code><br><code>*a--</code></td></tr>
<tr><td colspan="3" rowspan="4"></td><td>Supports arithmetic operators + and -</td><td><code>a + n</code><br><code>n + a</code><br><code>a - n</code><br><code>a - b</code></td></tr>
<tr><td>Supports inequality comparisons (<, >, <= and >=) between iterators</td><td><code>a < b</code><br><code>a > b</code><br><code>a <= b</code><br><code>a >= b</code></td></tr>
<tr><td>Supports compound assignment operations += and -=</td><td><code>a += n</code><br><code>a -= n</code></td></tr>
<tr><td>Supports offset dereference operator ([])</td><td><code>a[n]</code></td></tr>
</table>

Where `X` is an iterator type, `a` and `b` are objects of this iterator type, `t` is an object of the type pointed by the iterator type, and `n` is an integer value.

Random access iterators have all characteristics. Bidirectional iterators have a subset of random access iterators's.Forward iterators have a subset of bidirectional iterators's. And input and output have each their own subset of forward iterator's.

# Iterator Classes

### *Primitives*

| | |
|---|---|
| **iterator** | the basic iterator<br>(class template) |
| **input_iterator_tag** | |
| **output_iterator_tag** | |
| **forward_iterator_tag** | empty class types used to indicate iterator categories<br>(class) |
| **bidirectional_iterator_tag** | |
| **random_access_iterator_tag** | |
| **iterator_traits** | provides uniform interface to the properties of an iterator<br>(class template) |

### *Adaptors*

| | |
|---|---|
| **reverse_iterator** | iterator adaptor for reverse-order traversal<br>(class template) |
| **move_iterator**<br>(C++11) | iterator adaptor which dereferences to an rvalue reference<br>(class template) |
| **back_insert_iterator** | iterator adaptor for insertion at the end of a container<br>(class template) |
| **front_insert_iterator** | iterator adaptor for insertion at the front of a container<br>(class template) |
| **insert_iterator** | iterator adaptor for insertion into a container<br>(class template) |

### *Stream Iterators*

| | |
|---|---|
| **istream_iterator** | input iterator that reads from `std::basic_istream`<br>(class template) |
| **ostream_iterator** | output iterator that writes to `std::basic_ostream`<br>(class template) |
| **istreambuf_iterator** | input iterator that reads<br>from `std::basic_streambuf`<br>(class template) |
| **ostreambuf_iterator** | output iterator that writes to `std::basic_streambuf`<br>(class template) |

## Functions *Adaptors*

| | |
|---|---|
| **make_move_iterator** (C++11) | creates a `std::move_iterator` of type inferred from the argument (function template) |
| **front_inserter** | creates a `std::front_insert_iterator` of type inferred from the argument (function template) |
| **back_inserter** | creates a `std::back_insert_iterator` of type inferred from the argument (function template) |
| **inserter** | creates a `std::insert_iterator` of type inferred from the argument (function template) |

## Functions *Operations*

| | |
|---|---|
| **advance** | advances an iterator by given distance (function) |
| **distance** | returns the distance between two iterators (function) |
| **next** (C++11) | increment an iterator (function) |
| **prev** (C++11) | decrement an iterator (function) |
| **Range** | |
| **begin** (C++11) | returns an iterator to the beginning of a container or array (function) |
| **end** (C++11) | returns an iterator to the end of a container or array (function) |

## *Non-member operators*

| | |
|---|---|
| **operator==** **operator!=** **operator<** **operator<=** **operator>** **operator>=** | compares the underlying iterators (function template) |
| **operator+** | advances the iterator (function template) |
| **operator-** | computes the distance between two iterator adaptors (function template) |

| | |
|---|---|
| operator==<br>operator!=<br>operator<<br>operator<=<br>operator><br>operator>= | compares the underlying iterators<br>(function template) |
| **operator+** | advances the iterator<br>(function template) |
| **operator-** | computes the distance between two iterator adaptors<br>(function template) |
| **operator==**<br>**operator!=** | compares two istream_iterators<br>(function template) |
| **operator==**<br>**operator!=** | compares two istreambuf_iterators<br>(function template) |

## *Iterators Example*

http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html

```
#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;
   }

   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;
}
```

### *Iterator adaptors*

In addition to iterating through containers, iterators can iterate over streams, either to read elements or to write them.

An input stream like *cin* has the right functionality for an input iterator:

- it provides access to a sequence of elements.

The trouble is, it has the wrong interface for an iterator:

- operations that use iterators expect to be able to increment them and dereference them.

STL provides **adaptors**, types that transform the interface of other types. This is very much how electrical adaptors work.

One very useful adaptor is istream_iterator.

This is a template type; you parameterize it by the ***type*** of object you want to read from the stream. i.e. int, char, float, etc..

**istream_iterator**

In this case we want integers, so we would use an istream_iterator<int>.

- Istream iterators are initialized by giving them a stream
- dereferencing the iterator reads an element from the stream,
- incrementing the iterator has no effect.

An istream iterator that is created with the default constructor:

- has the past-the-end value,

as does an iterator whose stream has reached the end of file.

# Iterator Adaptor Example

```
// include library for each feature used HW
// cppreference.com

#include <algorithm>  // sort
#include <vector>  // vector
#include <iostream>  //cin cout
#include <iterator>  //iterators


int main ()
{
  using namespace std;

  // create a vector to hold numbers typed in
  vector<int> v;
  cout << "Please enter numbers (ctrl-D) ends sequence" << endl;
  istream_iterator<int> start (cin);  //input iterator from stream
  istream_iterator<int> end;   // end of stream iterator
  back_insert_iterator<vector<int> > dest (v);  // append integers to vector

  copy (start, end, dest);  // copy cin numbers to vector
  sort(v.begin(), v.end());  // sort the vector
  copy (v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));  // copy vector to cout

  return 0;
}
```

the vector is

- copied into memory,
- sorted, and
- copied out again.

# Templates

Are a feature of the C++ programming language that allow functions and classes to operate with generic types.

- char
- int
- float
- double
- structure
- class

This allows a function or class to work on many different data types without being rewritten for each one.

Templates are of great utility to programmers in C++, especially when combined with multiple inheritance and operator overloading.

The C++ Standard Library provides many useful functions within a framework of connected templates.

http://www.thefullwiki.org/Template_(programming)

### *Defining a Template*

Imagine that we want to write a function to compare two values and indicate whether the first is less than, equal to, or greater than the second.

In practice, we'd want to define several such functions, each of which will compare values of a given type.

Our first attempt might be to define several overloaded functions:

```
//  returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
int compare(const string &v1, const string &v2)
{
if (v1 < v2) return -1;
if (v2 < v1) return 1;
return 0;
}
int compare(const double &v1, const double &v2)
{
if (v1 < v2) return -1;
if (v2 < v1) return 1;
return 0;
}
```

These functions are nearly identical: The only difference between them is the type of their parameters.

The function body is the same in each function.

Having to repeat the body of the function for each type that we compare is tedious and error-prone.

More importantly, we need to know when we write the program all the types that we might ever want to compare. This strategy cannot work if we want to be able to use the function on types that our users might supply.

Lippman, Lajoie, Moo – C++11

Rather than defining a new function for each type, we can define a function **template.**

A function template is a formula from which we can generate **type-specific** versions of that function.

The template version of compare looks like

### *Function Template Definition Example*

```
template <typename T>
int compare(const T &v1, const T &v2)
{
     if (v1 < v2) return -1;
     if (v2 < v1) return 1;
     return 0;
}
```

A **template definition** starts with the

**keyword template** followed by a

**template parameter list**,

which is a comma-separated list of one or more template parameters

bracketed by the less-than (<) and greater-than (>) tokens.

Our compare function declares one type parameter named T.

Inside compare, we use the name T to refer to a type.

Which actual type T represents is determined at compile time based on how compare is used.

## *Function templates Example*

```
1.  #include <iostream>  // cin, cout, endl
2.
3.  using std::cout;
4.  using std::endl;
5.  using std::string;
6.
7.  template <typename T>
8.  const T& max(const T& x, const T& y)
9.  {
10.   if(y < x)
11.     return x;
12.   return y;
13. }
14.
15. int main()
16. {
17.
18.   // This will call max <int> (by argument deduction)
19.   cout << max(3, 7) << endl;
20.   // This will call max<double> (by argument deduction)
21.   cout << max(3.0, 4.0) << endl;
22.   // This type is ambiguous; explicitly instantiate max<double>
23.   cout << max<double>(3, 8.0) << endl;
24.   // This type is character by explicit instantiation
25.   cout << max<char>('A', 'C') << endl;
26.
27.
28.   return 0;
29. }
```

- In 19 and 21, the template argument T is automatically deduced by the compiler to be int and double, respectively.
- In 23, case deduction fails because the type of the parameters must in general exactly match the template arguments.
- This function template can be instantiated with any copy-constructible type for which the expression (y < x) is valid.
- **For user-defined types, this implies that the less-than operator must be overloaded.**

## *Function Template Variations Example*

```
// function template variations
#include <vector>
#include <iterator>
#include <iostream>
#include <cmath>
#include <string>
#include <utility>
#include <cstring>
using std::vector;
using std::cout;
using std::endl;
using std::string;
using std::iter_swap;
template <typename T>
void myReverse(T& input)
{
    typename T::iterator it;
    typename T::iterator et;
    for (it=input.begin(), et=input.end(); it< et; it++, et--)
    {
        iter_swap(it, et-1);
    }
    return;
}
template <typename T>
void showContents(T& input)
{
    typename T::iterator it;
    for (it=input.begin(); it != input.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}
int main()
{
    int x[] = {1, 2, 3, 4, 5};
    vector<int> MyVec(x, x+5);
    showContents(MyVec);
    myReverse(MyVec);
    showContents(MyVec);

    string str = "This is a C++ string";
    vector<char> data(str.begin(), str.end());
    showContents(data);
    myReverse(data);
    showContents(data);

    string MyString ("This is a C++ string container");
    showContents(MyString);
    myReverse(MyString);
    showContents(MyString);

    char Cstr[] = "This is a C string char array";
    vector<char> data1(Cstr, Cstr+(strlen(Cstr)));
    showContents(data1);
    myReverse(data1);
    showContents(data1);

    vector<string> fruit;  // This is a string vector
    fruit.push_back("apple");
    fruit.push_back("banana");
    fruit.push_back("orange");
    fruit.push_back("strawberry");
    showContents((fruit));
    myReverse(fruit);
    showContents((fruit));
}
```

### *Class templates*

A class template provides a specification for generating classes based on parameters.

Class templates are commonly used to implement containers.

A class template is instantiated by passing a given set of types to it as template arguments.

The C++ Standard Library contains many class templates, in particular the containers adapted from the Standard Template Library, such as vector.

The basic syntax for declaring a templated class is as follows:

```
template <class a_type>

class a_class {...};
```

The keyword 'class' above simply means that the identifier a_type will stand for a datatype.

Note: a_type is not a keyword; it is an identifier that during the execution of the program will represent a single datatype. For example, you could, when defining variables in the class, use the following line:

```
a_type a_var;
```

and when the programmer defines which datatype 'a_type' is to be when the program instantiates a particular instance of a_class, a_var will be of that type.

When declaring an instance of a templated class, the syntax is as follows:

```
a_class<int> an_example_class;
```

## *Class Template Example*

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

## *introducing the vector template*

```
// vect1.cpp -- introducing the vector template HW
#include <iostream>
#include <string>
#include <vector>
#define NUM 5
int main()
{
using std::vector;
using std::string;
using std::cin;
using std::cout;
using std::endl;

vector<int> ratings(NUM);
vector<string> titles(NUM);
cout << "You will do exactly as told. You will enter"
<< NUM << " book titles and your ratings (0-10).\n";

int i;
for (i = 0; i < NUM; i++)
{
  cout << "Enter title #" << i + 1 << ": ";
  getline(cin,titles[i]);
  cout << "Enter your rating (0-10): ";
  cin >> ratings[i];
  cin.get();
}
cout << "Thank you. You entered the following:\n"
<< "Rating\tBook\n";

for (i = 0; i < NUM; i++)
{
  cout << ratings[i] << "\t" << titles[i] << endl;
}
return 0;
}
```

# STL components

## *Containers*

Containers are objects that conceptually contain other objects.

They use certain basic properties of the objects (ability to copy, etc.) but otherwise do not depend on the type of object they contain.

STL containers may contain pointers to objects, though in this case you will need to do a little extra work.

vectors, lists, deques, sets, multisets, maps, multimaps, queues, stacks, and priority queues, are all provided.

Perhaps more importantly,

built-in containers (C arrays) and

user-defined containers

can also be used as STL containers;.

This is generally useful when applying operations to the containers, e.g., sorting a container.

Using user-defined types as STL containers can be accomplished by satisfying the requirements listed in the STL container requirements definition.

If this is not feasible, you can define an adaptor class that changes the interface to satisfy the requirements.

All the types are "templated", of course,

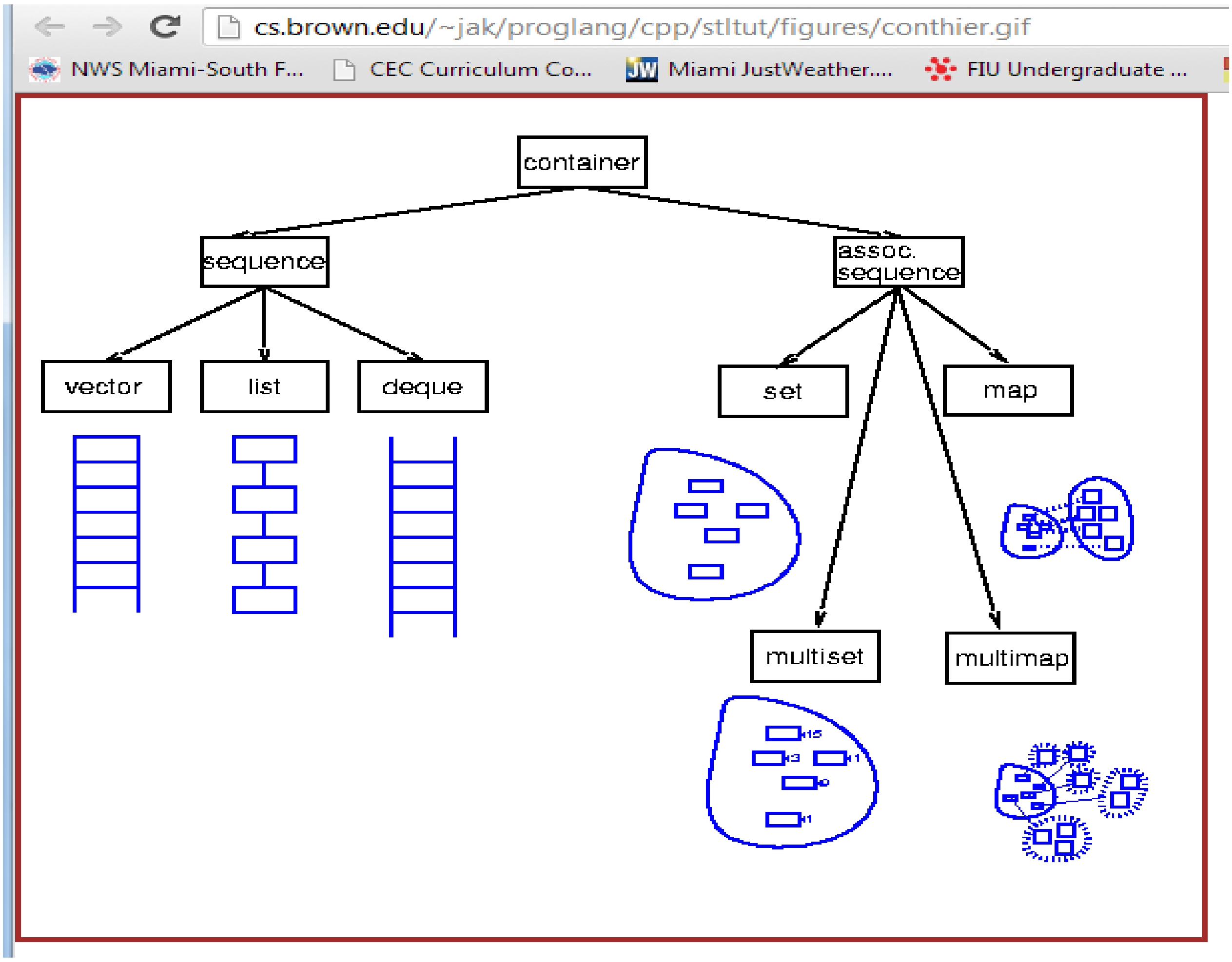
so you can have a vector of ints

or Windows

or a vector of vector of sets of multimaps of strings to Students.

Sweat, compiler-writers, sweat! - Make the compiler do the work.

## *Hierarchy*




### Sequences

Contiguous blocks of objects; you can insert elements at any point in the sequence, but the performance will depend on the type of sequence and where you are inserting.

#### Vectors

Fast insertion at end, and allow random access.

#### Lists

Fast insertion anywhere, but provide only sequential access.

#### Deques

Fast insertion at either end, and allow random access. Restricted types, such as stack and queue, are built from these using adaptors.

#### Stacks and queues

Provide restricted versions of these types, in which some operations are not allowed.

### Associative containers

Associative containers are a generalization of sequences. Sequences are indexed by integers; associative containers can be indexed by any type.

The most common type to use as a key is a string; you can have a set of strings, or a map from strings to employees, and so forth.

It is often useful to have other types as keys; for example, if I want to keep track of the names of all the Widgets in an application, I could use a map from Widgets to Strings.

#### Sets

Sets allow you to add and delete elements, query for membership, and iterate through the set.

#### Multisets

Multisets are just like sets, except that you can have several copies of the same element (these are often called bags).

#### Maps

Maps represent a mapping from one type (the *key* type) to another type (the *value* type). You can associate a value with a key, or find the value associated with a key, very efficiently; you can also iterate through all the keys.

#### Multimaps

Multimaps are just like maps except that a key can be associated with several values.

<u>other containers:</u>

**priority queue,**

**bit vector,**

**queue**

### *Example using container algorithms*

Here is a program that generates a random permutation of the first n integers, where **n** is specified on the command line.

```
#include <iostream.h>
#include <vector.h>
#include <algo.h>
#include <iterator.h>

main (int argc, char *argv[])
{
  int n = atoi (argv[1]); // argument checking removed for clarity

  vector<int> v;
  for (int i = 0; i < n; i++)            // append integers 0 to n-1 to v
    v.push_back (i);

  random_shuffle (v.begin(), v.end());                         // shuffle
  copy (v.begin(), v.end(), ostream_iterator<int> (cout, "\n")); // print
}
```

This program creates an empty vector and fills it with the integers from 0 to **n**. It then shuffles the vector and prints it out.

# Container Summary Table

## *Simple Containers*

| Container | Description |
|---|---|
| **Simple Containers** | |
| pair | The pair container is a simple associative container consisting of a 2-tuple of data elements or objects, called 'first' and 'second', in that fixed order. The STL 'pair' can be assigned, copied and compared. The array of objects allocated in a map or hash_map (described below) are of type 'pair' by default, where all the 'first' elements act as the unique keys, each associated with their 'second' value objects. |
| **Sequences (Arrays / Linked Lists) - ordered collections** | |
| vector | a dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object. Inserting and removing an element to/from back of the vector at the end takes amortized constant time. Inserting and erasing at the beginning or in the middle is linear in time.<br>A specialization for type bool exists, which optimizes for space by storing bool values as bits. |
| list | a doubly-linked list; elements are not stored in contiguous memory. Opposite performance from a vector. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time). |
| deque (double ended queue) | a vector with insertion/erase at the beginning or end in amortized constant time, however lacking some guarantees on iterator validity after altering the deque. |

## Container Adaptors

| Container | Description |
|---|---|
| **Container adaptors** | |
| queue | Provides FIFO queue interface in terms of `push/pop/front/back` operations.<br>Any sequence supporting operations `front()`, `back()`, `push_back()`, and `pop_front()` can be used to instantiate queue (e.g. list and deque). |
| priority_queue | Provides priority queue interface in terms of `push/pop/top` operations (the element with the highest priority is on top).<br>Any random-access sequence supporting operations `front()`, `push_back()`, and`pop_back()` can be used to instantiate priority_queue (e.g. vector and deque). Elements should additionally support comparison (to determine which element has a higher priority and should be popped first). |
| stack | Provides LIFO stack interface in terms of `push/pop/top` operations (the last-inserted element is on top).<br>Any sequence supporting operations `back()`, `push_back()`, and `pop_back()` can be used to instantiate stack (e.g. vector, list, and deque). |

## *Containers – Associative and other types*

| Container | Description |
|---|---|
| **Associative containers - unordered collections** | |
| set | a mathematical set; inserting/erasing elements in a set does not invalidate iterators pointing in the set. Provides set operations union, intersection, difference, symmetric difference and test of inclusion. Type of data must implement comparison operator < or custom comparator function must be specified. Implemented using a self-balancing binary search tree. |
| multiset | same as a set, but allows duplicate elements. |
| map | an associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator < or custom comparator function must be specified. Implemented using a self-balancing binary search tree. |
| multimap | same as a map, but allows duplicate keys. |
| hash_set<br>hash_multiset<br>hash_map<br>hash_multimap | similar to a set, multiset, map, or multimap, respectively, but implemented using a hash table; keys are not ordered, but a hash function must exist for the key type. Thesecontainers are not part of the C++ Standard Library, but are included in SGI's STL extensions, and are included in common libraries such as the GNU C++ Library in the__gnu_cxx namespace. These are scheduled to be added to the C++ standard as part ofTR1, with the slightly different names of unordered_set, unordered_multiset,unordered_map and unordered_multimap. |
| **Other types of containers** | |
| bitset | stores series of bits similar to a fixed-sized vector of bools. Implements bitwise operations and lacks iterators. Not a Sequence. |
| valarray | another C-like array like vector, but is designed for high speed numerics at the expense of some programming ease and general purpose use. It has many features that make it ideally suited for use with vector processors in traditional vector supercomputers and SIMD units in consumer-level scalar processors, and also ease vector mathematics programming even in scalar computers. |

# Algorithms

## **algorithm**: Routines to find, count, sort, search, ... elements in container classes

A large number of algorithms to perform operations such as searching and sorting are provided in the STL,

each implemented to require a certain level of iterator (and therefore will work on any container which provides an interface by iterators).

### *Algorithms library*

C++

**Algorithm library**

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as `[first, last)` where `last` refers to the element *past*the last element to inspect or modify.

### *Non-modifying sequence operations*

Defined in header `<algorithm>`

| | |
|---|---|
| **all_of**<br>**any_of**<br>**none_of**<br>(C++11)<br>(C++11)<br>(C++11) | checks if a predicate is `true` for all, any or none of the elements in a range<br>(function template) |
| **for_each** | applies a function to a range of elements<br>(function template) |
| **count**<br>**count_if** | returns the number of elements satisfying specific criteria<br>(function template) |
| **mismatch** | finds the first position where two ranges differ<br>(function template) |
| **equal** | determines if two sets of elements are the same<br>(function template) |
| **find**<br>**find_if**<br>**find_if_not**<br><br>(C++11) | finds the first element satisfying specific criteria<br>(function template) |
| **find_end** | finds the last sequence of elements in a certain range<br>(function template) |
| **find_first_of** | searches for any one of a set of elements |

| | |
|---|---|
| | (function template) |
| **adjacent_find** | finds two identical (or some other relationship) items adjacent to each other<br>(function template) |
| **search** | searches for a range of elements<br>(function template) |
| **search_n** | searches for a number consecutive copies of an element in a range<br>(function template) |

### *Modifying sequence operations*

Defined in header `<algorithm>`

| | |
|---|---|
| **copy**<br>**copy_if**<br>(C++11) | copies a range of elements to a new location<br>(function template) |
| **copy_n**<br>(C++11) | copies a number of elements to a new location<br>(function template) |
| **copy_backward** | copies a range of elements in backwards order<br>(function template) |
| **move**<br>(C++11) | moves a range of elements to a new location<br>(function template) |
| **move_backward**<br>(C++11) | moves a range of elements to a new location in backwards order<br>(function template) |
| **fill** | assigns a range of elements a certain value<br>(function template) |
| **fill_n** | assigns a value to a number of elements<br>(function template) |
| **transform** | applies a function to a range of elements<br>(function template) |
| **generate** | saves the result of a function in a range<br>(function template) |
| **generate_n** | saves the result of N applications of a function<br>(function template) |
| **remove**<br>**remove_if** | removes elements satisfying specific criteria<br>(function template) |
| **remove_copy**<br>**remove_copy_if** | copies a range of elements omitting those that satisfy specific criteria<br>(function template) |
| **replace**<br>**replace_if** | replaces all values satisfying specific criteria with another value<br>(function template) |
| replace_copy<br>replace_copy_if | copies a range, replacing elements satisfying specific criteria with another value |

| | |
|---|---|
| | (function template) |
| **swap** | swaps the values of two objects (function template) |
| **swap_ranges** | swaps two ranges of elements (function template) |
| **iter_swap** | swaps the elements pointed to by two iterators (function template) |
| **reverse** | reverses the order elements in a range (function template) |
| **reverse_copy** | creates a copy of a range that is reversed (function template) |
| **rotate** | rotates the order of elements in a range (function template) |
| **rotate_copy** | copies and rotate a range of elements (function template) |
| **random_shuffle** **shuffle** (C++11) | randomly re-orders elements in a range (function template) |
| **unique** | removes consecutive duplicate elements in a range (function template) |
| **unique_copy** | creates a copy of some range of elements that contains no consecutive duplicates (function template) |

### *Partitioning operations*

Defined in header `<algorithm>`

| | |
|---|---|
| **is_partitioned** (C++11) | determines if the range is partitioned by the given predicate (function template) |
| **partition** | divides a range of elements into two groups (function template) |
| **partition_copy** (C++11) | copies a range dividing the elements into two groups (function template) |
| **stable_partition** | divides elements into two groups while preserving their relative order (function template) |
| **partition_point** (C++11) | locates the partition point of a partitioned range (function template) |

### *Sorting operations (on sorted ranges)*

Defined in header `<algorithm>`

| | |
|---|---|
| **is_sorted** (C++11) | checks whether a range is sorted into ascending order (function template) |

| | |
|---|---|
| **is_sorted_until** (C++11) | finds the largest sorted subrange (function template) |
| **sort** | sorts a range into ascending order (function template) |
| **partial_sort** | sorts the first N elements of a range (function template) |
| **partial_sort_copy** | copies and partially sorts a range of elements (function template) |
| **stable_sort** | sorts a range of elements while preserving order between equal elements (function template) |
| **nth_element** | partially sorts the given range making sure that it is partitioned by the given element (function template) |

### *Binary search operations (on sorted ranges)*

Defined in header `<algorithm>`

| | |
|---|---|
| **lower_bound** | returns an iterator to the first element *not less* than the given value (function template) |
| **upper_bound** | returns an iterator to the first element *greater* than a certain value (function template) |
| **binary_search** | determines if an element exists in a certain range (function template) |
| **equal_range** | returns range of elements matching a specific key (function template) |

### *Set operations (on sorted ranges)*

Defined in header `<algorithm>`

| | |
|---|---|
| **merge** | merges two sorted ranges (function template) |
| **inplace_merge** | merges two ordered ranges in-place (function template) |
| **includes** | returns true if one set is a subset of another (function template) |
| **set_difference** | computes the difference between two sets (function template) |
| **set_intersection** | computes the intersection of two sets (function template) |
| **set_symmetric_difference** | computes the symmetric difference between two sets (function template) |

| | |
|---|---|
| **set_union** | computes the union of two sets<br>(function template) |

### *Heap operations*

Defined in header <algorithm>

| | |
|---|---|
| **is_heap** | checks if the given range is a heap<br>(function template) |
| **is_heap_until**<br>(C++11) | finds the largest subrange that is heap<br>(function template) |
| **make_heap** | creates a heap out of a range of elements<br>(function template) |
| **push_heap** | adds an element to a heap<br>(function template) |
| **pop_heap** | removes the largest element from a heap<br>(function template) |
| **sort_heap** | turns a heap into a sorted range of elements<br>(function template) |

### *Minimum/maximum operations*

Defined in header <algorithm>

| | |
|---|---|
| **max** | returns the larger of two elements<br>(function template) |
| **max_element** | returns the largest element in a range<br>(function template) |
| **min** | returns the smaller of two elements<br>(function template) |
| **min_element** | returns the smallest element in a range<br>(function template) |
| **minmax**<br>(C++11) | returns the larger and the smaller of two elements<br>(function template) |
| **minmax_element**<br>(C++11) | returns the smallest and the largest element in a range<br>(function template) |
| **lexicographical_compare** | returns true if one range is lexicographically less than another<br>(function template) |
| **is_permutation**<br>(C++11) | determines if a sequence is a permutation of another sequence<br>(function template) |
| **next_permutation** | generates the next greater lexicographic permutation of a range of elements<br>(function template) |
| **prev_permutation** | generates the next smaller lexicographic permutation of a range of elements<br>(function template) |

## *Numeric operations*

Defined in header `<numeric>`

| | |
|---|---|
| **iota** (C++11) | fills a range with successive increments of the starting value (function template) |
| **accumulate** | sums up a range of elements (function template) |
| **inner_product** | computes the inner product of two ranges of elements (function template) |
| **adjacent_difference** | computes the differences between adjacent elements in a range (function template) |
| **partial_sum** | computes the partial sum of a range of elements (function template) |

## *C library*

Defined in header `<cstdlib>`

| | |
|---|---|
| **qsort** | sorts a range of elements with unspecified type (function) |
| **bsearch** | searches an array for an element of unspecified type (function) |

# Functors

The STL includes classes that overload the function operator (operator())

They are useful for keeping and retrieving state information in functions passed into other functions.

Function objects (aka "functors"). Functors are objects that can be treated as though they are a function or function pointer--you could write code that looks like this:

```
myFunctorClass functor;
functor( 1, 2, 3 );
```

This code works because C++ allows you to overload operator(), the "function call" operator. The function call operator can take any number of arguments of any types and return anything it wishes to. (It's probably the most flexible operator that can be overloaded)

```
#include <iostream>

class myFunctorClass
{
   public:
      myFunctorClass (int x) : _x( x ) {}
      int operator() (int y) { return _x + y; } //<<<<<<<<<<<
   private:
      int _x;
};

int main()
{
   myFunctorClass addFive( 5 );
   std::cout << addFive( 6 );

   return 0;
}
```

http://www.cprogramming.com/tutorial/functors-function-objects-in-c++.html

## C++ Standard Library

**C++ Standard Library**

- ios
- iostream
- iomanip
- fstream
- sstream

***C++ Standard Library***

# Standard Template Library

ALL wxWidgets Classes

- vector
- deque
- list
- map
- set
- stack
- queue
- bitset
- algorithm
- functional
- iterator

- WxVector

wxList
wxHashMap
wxHashSet
wxStack