The C++ Standard Template Library

Student Manual

Module Table of Contents

Introduction

About this course

This material is designed to teach experienced C and C++ programmers about the Standard Template Library.

Typographical syntax

Examples in this text of commands will appear in **bold** text and the output of the commands will appear in *italic* text. The commands and the output of the commands will be placed in a box to separate them from other text. Example:

```
[student@linux1 student]$ pwd
/home/student
```

Note: "[student@linux1 student]$" is a **prompt**, a method the shell uses to say “I’m ready for a new command”.

Bold text within a sentence will indicate an important term or a command. Files and directories are highlighted by being placed in `courier` font.

Using this manual while in class

In many ways, typename manuals are different from textbooks. Textbooks are often filled with lengthy paragraphs that explain a topic in detail. Unfortunately, this style doesn't work well in a classroom environment.

Class manuals often are much more concise than textbooks. It's difficult to follow the instructor's example and read lengthy paragraphs in a book at the same time. For this purpose, typename manuals are often more terse, similar to a presentation.

Lab Exercises

The lab exercises provided in this class are intended to provide practical, hands on experience with programming in C++ using the STL. Students are strongly encouraged to perform the labs provided at the end of each Unit to reinforce the knowledge provided in class.

Unit One
Basic Concepts

Unit topics:

Notes:

1.1 Origin and History of the Standard Template Library

Alexander Stepanov is the person whose ideas initiated what has become the Standard Template Library. He was educated in math but liked real world applications of math and so became a programmer. He also likes to think abstractly and so always tried to make abstract concepts work in a program. One of his first jobs was to program a minicomputer to be used to control large hydroelectric power stations. He started thinking about general algorithms and data structures even at that time.

Abstract data types (templates) and generic programming are both results of his desire to use math and abstract concepts in the real world.

Notes:

The following is a brief a timeline of the STL's development:

- 1971: David Musser created some generic algorithms for computer algebra
- 1979: Stepanov began working on generic programming with Dave Musser and Deepak Kapur at GE Research and Development.
- 1985: Stepanov and Musser developed a generic library for the Ada language using generic programming, however C looked more like a language that everyone would use so they started exploring C/C++ programming shortly before joining Hewlett-Packard (HP)
- 1992: Meng Lee joined Stepanov's project at HP
- 1993: Andrew Koenig at Bell Labs found out about this work and asked them to present their ideas at the Nov. meeting of the ANSI/ISO committee for C++ standardization.
- 1994: HP made STL implementation available free on the Internet and it was adopted into the draft standard at the July ANSI/ISO C++ Standards Committee meeting.
- 1995: Stepanov went Silicon Graphics to further develop STL

1.2 Why use the STL?

- **Efficiency**. No inheritance or virtual functions. Runs quickly.

- **Reusability**. Creating templates once or using the predefined templates means programmers can focus on the unique aspects of their programs rather than reinventing the standards algorithms and containers that the STL provides.

- **Type safety**. Extensive use of C++ templates makes STL type safe.

- **Consistency**. STL container classes, functors and algorithms all use iterators which ties the parts of the STL together well while allowing each part to be used on its own if desired.

- **Extensibility**.
  1. STL's iterators are powerful extensions of regular pointers into pointer objects; thus they not only keep a value like regular pointers but can also keep several types of data and perform defined functions.

  2. STL algorithms are standalone functions that use iterators to operate on data. This data can be in STL containers or functors (or even in a programmer's own defined data structures and functors).

Notes:

Notes:

3. Like regular functions, STL's functors allow calculations, but also allow a function to be 'smart', that is to hold data along with the function. They can be used with a user defined data structures or custom own algorithms.

4. STL memory management does not directly use the new and delete operators, but uses special allocator objects to allocate and de-allocate storage. These are classes that a programmer can use in order to maximize memory usage and pointer cleanup.

1.3 What is the Standard Template Library?

In general, the Standard Template Library is a subset of the C++ Standard Library header files, consisting of headers that define container, function and iterator classes along with algorithms and other utilities that work with these classes.

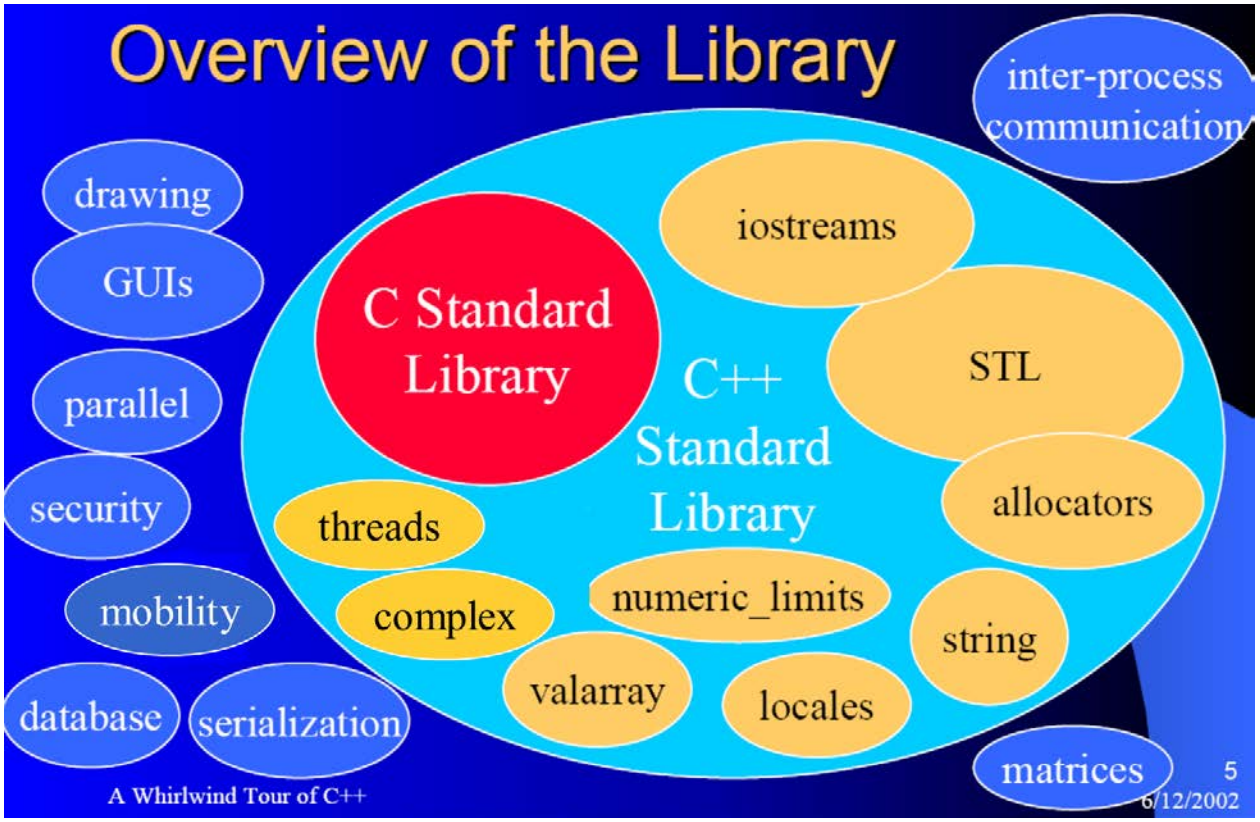
These classes, implemented as C++ structs, provide public access to their data members and functions, and all are written in a generic way to support both specific C++ built-in datatypes as well as user defined objects.

A few of the STL classes are related by inheritance; however in general, the STL is not object oriented. There are no virtual functions, and few related classes. The classes provide limited thread handling support, nor do most of them have any exception or error checking.

Standard C++ data structures and pointers can be used with the STL components.

Notes:

Notes:


Overview of the Library
inter-process communication
drawing
GUIs
parallel
security
mobility
database
serialization
C Standard Library
iostreams
C++ Standard Library
STL
allocators
threads
numeric_limits
complex
string
valarray
locales
matrices
A Whirlwind Tour of C++
5
6/12/2002

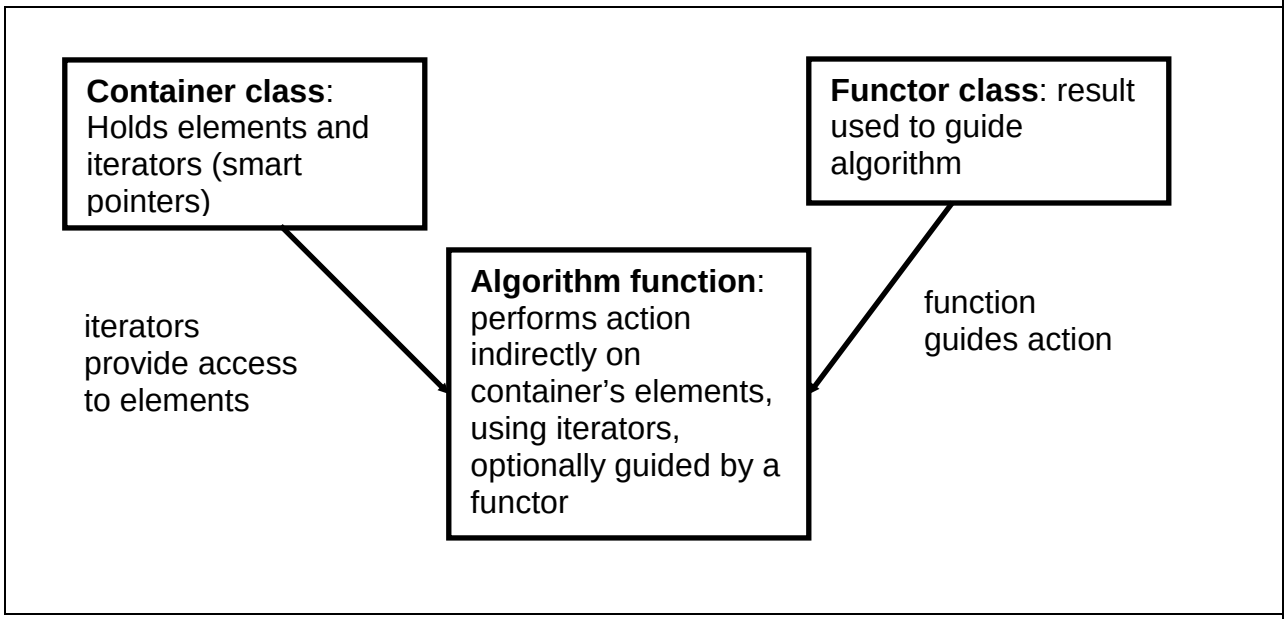
Notes:

The Standard C++ Library consists of many headers, but not all of them are considered to be the Standard Template Library. We think of the following headers as constituting the Standard Template Library:

Header	Definition
<algorithm>	Defines templates that implement useful algorithms
<bitset>	Defines a template class that administers sets of bits
<deque>	Defines a template class that implements a deque
<functional>	Defines several functor templates for predicates used with the templates defined in <algorithm> and <numeric>
<iterator>	Defines templates that help define and manipulate iterators
<list>	Defines a template class that implements a doubly linked list
<map>	Defines template classes that implement associative containers that map keys to values using a red-black tree. It contains both map and multimap.
<memory>	Defines templates that allocate and free storage
<numeric>	Defines functor templates that implement numeric functions
<queue>	Defines a template class that implements a queue container. It contains both queue and priority_queue.
<set>	Defines template classes that implement associative containers in a red-black tree. It contains both set and multiset.
<stack>	Defines a template class that implements a stack container
<utility>	Defines several templates of general utility
<vector>	Defines a template class that implements a vector container

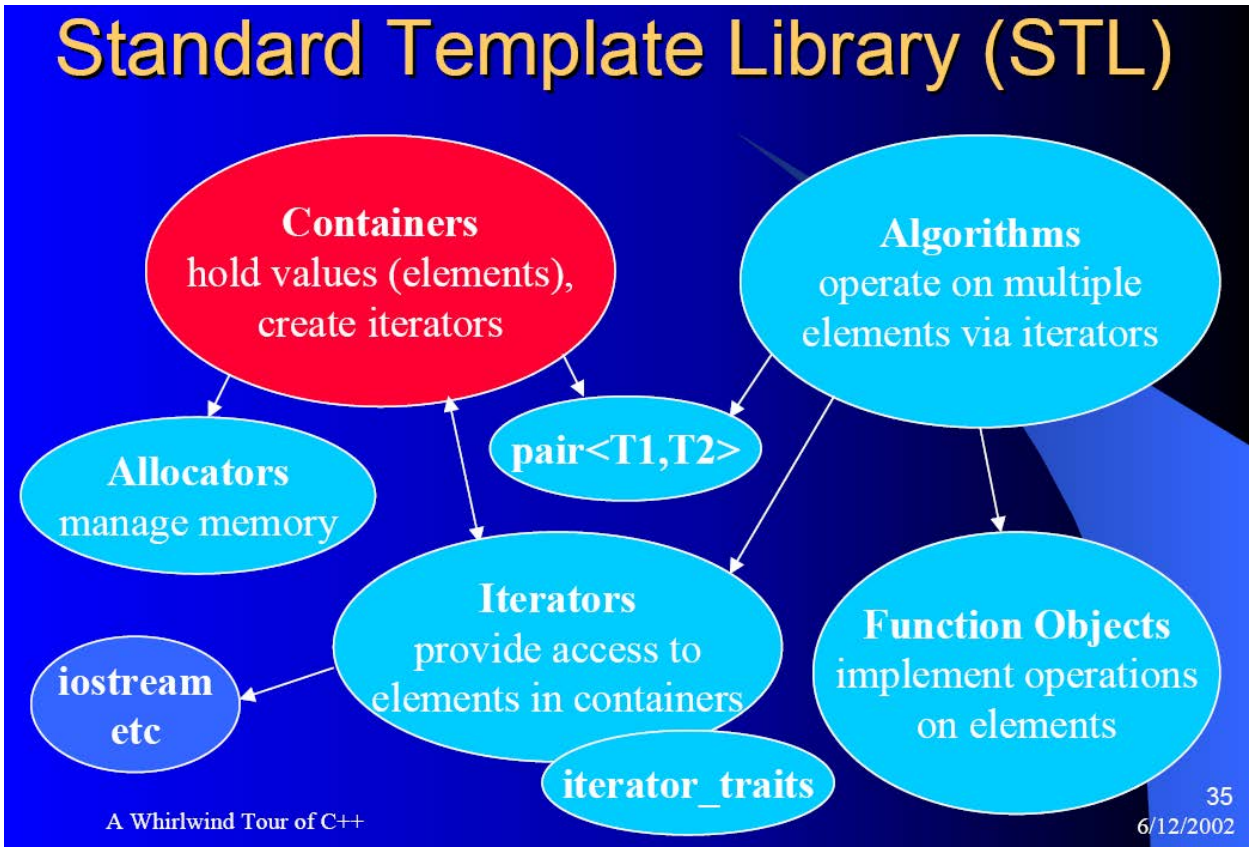
Notes:

Conceptual Picture of STL components at work



To use these components:

1. #include the necessary headers
2. Declare an object of a container class, providing a data type.
3. Declare an object of the container's Iterator class, with same datatype.
4. Add some elements to the container.
5. Perform some actions on those elements using algorithms and iterators, optionally guiding or limiting the algorithms' actions based on functors.


Standard Template Library (STL)
Containers
hold values (elements),
create iterators
Algorithms
operate on multiple
elements via iterators
Allocators
manage memory
pair<T1,T2>
Iterators
provide access to
elements in containers
Function Objects
implement operations
on elements
iostream
etc
iterator_traits
A Whirlwind Tour of C++
35
6/12/2002

Notes:

Notes:

The STL was created to support these goals:

1. **Generic programming**, which lets data types to be defined generically without needing to specify the exact type of data an algorithm, functor, or container will use until a program is compiled/linked/running. The type of data being passed when the STL template container, function or algorithm is invoked defines how it will work and ensures it will work correctly with the type of data being processed.

2. **Abstractness without loss of efficiency** due to generalized parameters, but done in a standard manner.

3. Use of the **Von Neumann computation model,** which means a running program is dynamically invoking these STL objects as it runs (i.e., everything is data to the computer, even the program).

4. **Value semantics**, where the values of parameters are copied to functions; not pointers – runs faster, but cannot use polymorphism with these containers without using object-oriented design patterns, such as Bridge.

Notes:

1.4 Foundational C++ Concepts

Namespace std

Using large numbers of modules and libraries in C and C++ sometimes creates class or function name clashes. The C++ standard solved this by introducing the concept of namespaces. A namespace is a scope that can go across many classes and even many files. It is open for extensions, unlike a class. All identifiers in the C++ Standard Library are defined to be part of namespace std.

There are 3 options when using a component of the std namespace:

1. Qualify the identifier directly as in

```
#include <iostream>

std::cout << “Hello World”  << std::endl;
```

2. Use a ‘using declaration’ - as in:

```
#include <iostream>
using std::cout;
using std::endl;

cout << “Hello World”  << endl;
```

Notes:

3. Using a 'using directive' - as in placing the phrase using namespace std; at the top of the program.

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

cout << "Hello World"  << endl;
```

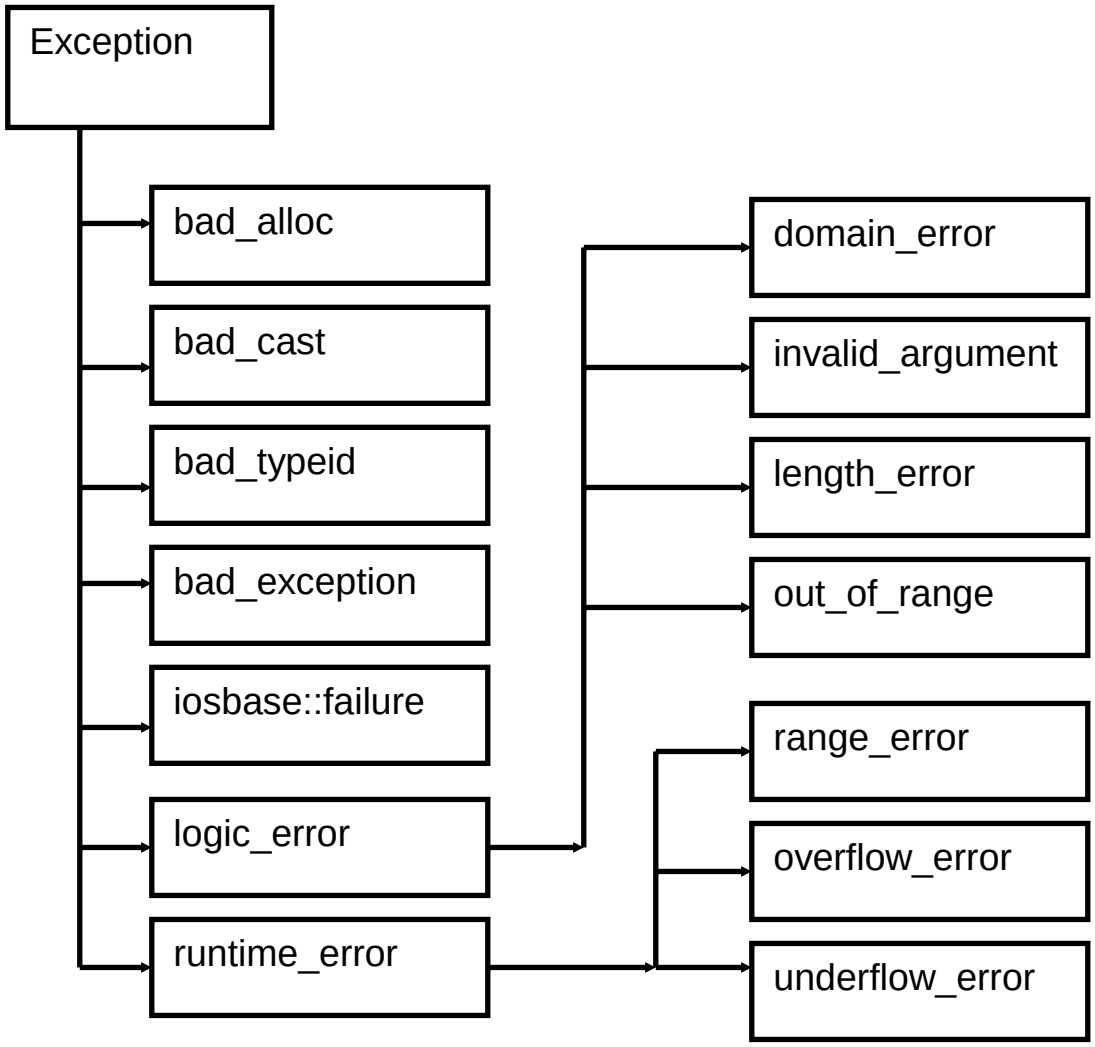
Error and Exception Handling

The C++ Standard Library is made up of files from various sources. Some support detailed exception handling, such as the string class. Other parts of the Library have functions and classes that rarely check for logic errors and throw exceptions only if runtime errors occur. When using the template classes and functions that comprise the STL, therefore, a program must add code to handle possible logical exceptions that may occur. To assist with this, here is a brief discussion of C++ Exception classes.

Standard Exception Classes

All C++ exceptions are derived from the base class, exception.
See diagram on the next page.

Standard C++ Exception Hierarchy



Notes:

Notes:

Exceptions for Language Support

In the chart above, several exceptions are thrown by C++ when core language features fail. They are the following exception classes:

1. An exception of class **bad_alloc** is thrown when the operator **new** fails.

2. An exception of class **bad_cast** is thrown when **dynamic_cast** is used and a datatype conversion fails during runtime.

3. An exception of class **bad_typeid** is thrown when the typeid operator is used with a null pointer or zero.

4. An exception of class **bad_exception** is thrown when an unexpected exception occurs and there is no code to handle it. This results in a call to unexpected(). Any function with an exception specifier list that happens to throw an unlisted exception will cause C++ to call unexpected(), and this calls terminate(), which ends the program quickly.

Exceptions for the Standard Library

These exceptions are derived from class logic_error. They are thought of as errors that a program could avoid by adding logic to handle them or better testing.

They are the following classes:

Notes:

1. An exception of class **invalid_argument** is thrown when any function is provided with an invalid argument.

2. An exception of class **length_error** is thrown when program statements attempt to do anything that exceeds a maximum allowable size such as appending too many characters to a string.

3. An exception of class **out_of_range** is thrown when an argument value is not in the expected range, such as when a bad index value is used for an array or collection class.

4. An exception of class **domain_error** is thrown to report domain errors.

5. An exception of class **ios_base::failure** is thrown when a stream changes state due to an error or reaching EOF.

Errors Outside a Program

Sometimes errors occur that a program cannot either anticipate or control. These are:

1. An exception of class **range_error** is thrown to report range errors in internal computations of C++.

2. Exception of classes **overflow_error** and **underflow_error** are used by C++ to report situations of arithmetic overflow or underflow.

Notes:

Header files for C++ Exception classes

<exception>	Contains exception and bad_exception.
<new>	Contains bad_alloc
<typeinfo>	Contains bad_typeid and bad_cast.
<ios>	Contains ios_base::failure.
<stdexcept>	Contains all other exceptions.

Memory Allocators

C++ uses special objects to allocate and deallocate memory. The STL template classes assume that a standard allocator object, Allocator, is available, although a program can optionally define a custom allocator class to be used with the STL containers. Apparently allocators were once needed to resolve the issues of near, far, and huge pointers in the past. However today, with most implementations providing pointers of the same size, allocators are useful to support objects that provide shared memory space, garbage collection algorithms, and object oriented databases.

Notes:

Unit Two
Generic Programming with Templates

Unit Topics

Notes:

2.1 Why Templates?

C++ is a strongly typed language, meaning that a program must declare all variables with their datatypes before they are used. This is great for ensuring that the correct amount of storage is used for each variable and that only the correct operations can be performed on its data. In addition data type checking for each variable prevents memory leaks and illegal operations that might causes exceptions in a running program.

However one drawback of this paradigm comes when a program needs to perform the same or a similar action on many types of variables. Without having classes or OO technology, a language such as C, used #define preprocessor directives to define variables and even small functions without datatypes. Another way to perform such actions in C was to use void pointers, since no datatype is explicitly involved. This 'gets around' the problem but doesn't solve it in a datatype safe manner – it just avoids the type checking that C or C++ provides.

Furthermore, neither of these options uses the power of modern OO thinking. Other techniques such as making all such functions or classes refer to Object or an Object reference, again don't solve the problem entirely; they are objected oriented, but still the problem remains that the data type checking part of C++ is bypassed.

To preserve the power of C++'s datatype checking, another way is to use many overloaded functions in a class, each having a different type of data as input. The problem with this comes later when the program needs to be

enhanced, maintained, fixed, etc. Then the job of keeping all of these functions in synch is a difficult one. Why not have ONE function that takes one parameter of any datatype? That is what a function template is.

Notes:

Notes:

2.2 Function Templates

Defining a function template

Function templates are special functions that can operate on different data types without separate code for each of them. For a similar operation on several kinds of data types, therefore, it isn't always as efficient to write several different versions by overloading a function; writing one template function will take care of many cases where overloaded functions might be written. C++ provides for both function templates and class templates. We will discuss function templates first.

The keyword **typename** was introduced to specify that the identifier following it is a type and need not be a class. Also, any identifier of a template is considered to be a value except if it is qualified by typename. Thus declaring a function to be a template can be done either way and they mean the same thing to C++:

```
template <class T> class xyz { };
template <typename T> class xyz { };
```

In general, we'll use "typename T" instead of "class T" to make it clear that any data type can be used, including built-in types.

Suppose an algorithm needs an add function but the programmer doesn't know right now if all calling programs will invoke this function using two integers, floats, chars etc.
So the programmer starts by creating two overloaded functions like this:

Notes:

```
int Add(int a, int b) { return a+b;}
float Add(float a, float b) { return a+b;}
```

Later the program might need to add two chars, two bytes or two doubles. Of course one can cast the chars to integers but the doubles pose a problem. Maybe a better plan would have used a generic datatype that could later be determined when the function is used in a running program.

Maybe later requirements could mean that 2 class objects will be passed to an Add function. How can a programmer think generally and create an addition function that could handle all types of data, even those not invented yet?

The template function concept solves this. Think generically about the type of data that the function could take. Then create the following template. Note that the template statement must be present right before the function it goes with, to make the function a 'template'. In addition, it is standard, but not required, to represent the generic datatype as 'T'.

A template function can create a family of functions. The Add template function represents a family of 'Add' functions that take two parameters of the same datatype, adds them together, and return the sum.

Let's try an example where a program can call this function with any type of data that supports the + operator, and it will work.

Notes:

Using a Function Template

For example, a program can call the Add function with two integers and it will return an integer. Call it with two chars, and it will return a char that represents their sum. To use it successfully with a user defined class, be sure the class overloads the + operator.

```
template <typename T>
T Add(T a, T b)  {
    return a+b;
}
```

Here we have created a template function with T as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type.

To use this function template use the following syntax for the function call:

function_name <type> (parameters);

For example, to call the Add() template function to sum two integer values:

```
int x = 5,y = 6, z;
z = Add <int> (x, y);
```

When the compiler encounters a call to a template function, it uses the template to automatically generate a function replacing each appearance of template parameter T by the type passed as the actual template parameter

(int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is a small program using explicit instantiation with integers and longs.

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

template <typename T>
T Add(T a, T b)   {
     return a+b;
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5,  n;
  k = Add<int> (i, j);
  n = Add<long> (l, m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Template Arguments and Argument Deduction

Notes:

Notes:

In this specific case where only one generic type T, is used as a parameter for Add, the compiler can automatically determine which data type to instantiate the function for without having to explicitly specify the datatype within angle brackets (like we have done before specifying <int> and <long>). This is called argument deduction.

Thus, in the sample program, the statements could have been written using implicit instantiation:

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

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k = Add (i, j);
  n = Add (l, m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Since both i and j are of type int in the first call to Add(), the compiler can automatically determine the template parameter is int and it can instantiate the function. The same is true when both parameters are of type long or any other built-in datatype.

Notes:

Because our template function includes only one template parameter (T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;
long l;
k = Add (i, l);    //invalid template function call
```

Function templates that accept more than one type parameter can also be defined, simply by specifying more template parameters between the angle brackets. For example:

```
template <typename T, typename U>
T GetMin (T a, U b) {
  return a < b ? a : b;
}
```

In this case, the function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed.

For example, we could call GetMin() with:

```
long l = 10,j = 20;
int l = 30;
```

Notes:

```
i = GetMin<long, int> (j,l);      // OR we can use  i = GetMin (j,l);
```

Template and Function call Parameters

The example above, where two different datatypes are used in the function template, illustrates the two types of parameter lists that template functions have. The first list, called the template parameters lies within the angle brackets on the template statement:

```
template <typename T, typename U>
```

The second parameter list for function templates lie within the parentheses of the function call itself:

```
T GetMin (T a, U b)
```

As seen in this example, usually all the template parameters are related to the function call parameters so that invoking the function allows C++ to deduce the datatypes of all the template parameters.

However, it is possible to add template parameters that aren't in the function call list. If that is done, these parameters must be placed before any parameters that are in both lists, and a program invoking the function must specify all the arguments up to the last argument datatype that cannot be determined from the function call parameters. For example in this template function the datatypes T and U can be determined when the function is called. But the return datatype, R, cannot. Any program using

Notes:

this function must provide the return datatype and optionally can provide all datatypes.

```
template <typename R, typename T, typename U>
R max(T a, U b) {
   return a<b ? b: a;
}
double d = max<double>(5.5D, 2);   <- must provide return datatype
int i = max<int, char, char>('a', 'b');   <- can provide all datatypes
```

Like other functions, template functions can be inline or external and even static. They can be combined in a program with regular functions of the same name. When this occurs, the regular function is invoked if it fits, otherwise the template is used.

Templates are processed twice by a C++ compiler:

1. The template code declaration *can be* checked for correct syntax during compilation.
2. When the template is instantiated during program link-time, the template code is checked to ensure that all calls are valid based on the type and number of arguments specified.

A consequence of this two phase processing is that some errors don't show up until a template is actually instantiated!

Function Template Overloading

Notes:

One can overload a function template either by a non-template function or by another function template.

If a program invokes an overloaded function template, the compiler will try to deduce its template arguments and check its explicitly declared template arguments. If successful, it will instantiate a function template specialization, then add this specialization to the set of candidate functions used in overload resolution.

The compiler proceeds with overload resolution, choosing the most appropriate function from the set of candidate functions. Non-template functions take precedence over template functions (which forms the basis for so-called “template specialization,” as we’ll discuss later. Here is an example:

```
#include <iostream>
 using namespace std;
template<typename T> void f(T x, T y) { cout << "Template" << endl; }
void f(int w, int z) { cout << "Non-template" << endl; }

int main() {
   f( 1 ,  2 );
   f('a', 'b');
   f( 1 , 'b');
}
```

The following is the output of the above example:

Notes:

Non-template
Template
Non-template

The function call f(1, 2) could match the argument types of both the template function and the non-template function. The non-template function is called because a non-template function takes precedence in overload resolution.

The function call f('a', 'b') can only match the argument types of the template function. The template function is called.

Argument deduction fails for the function call f(1, 'b'); the compiler does not generate any template function specialization and overload resolution does not take place. The non-template function resolves this function call after using the standard conversion from char to int for the function argument 'b'.

Here is an example using two function templates, one that has two parameters of a given datatype and the other that has three of a given datatype.

Notes:

```
// in a file called  min.h
template <typename T>
T min (T p1, T p2) {
   if (p1 < p2)
      return p1;
   else
      return p2;
}

template <typename T>
T min (T p1, T p2, T p3) {
   if (min (p1, p2) < p3)                  <- notice this calls the function
       return min (p1, p2) ;                 <- so does this line
   else
       return p3;
}
```

Here is the example program using these two function templates. Notice that the program must supply either two or three arguments when calling the function template or it doesn't work at all. If it were a regular function and it were invoked with an integer and a double, C++ could handle it by implicit casting. But template functions refuse to accept implicit casting. The program must explicitly cast arguments in order to get it to work.

```
#include "min.h"
#include <iostream>
using namespace std;
```

Notes:

```
int main() {
   int  i = min<>(9,6);
    cout <<"Minimum of integers: "<<  i << endl;

    double d= min(10.5,11.3);
    cout <<"Minimum of Doubles: "<< d << endl;

    //  Returns error since implicit cast not acceptable for templates
    d = min(10,11.3);
    cout  << "Minimum of Double: "<< d <<endl;

    // function template will accept an explicit cast though
    d= min<double>(10,11.3);
    cout  << "Minimum of Double: " << d <<endl;
    // 3 arguments work to call the overloaded function template

    i = min(3,6,4);
    cout <<   "Minimum int 3 Parameters: " << i  << endl;
}
```

Here is another example using a function template:

```
template <typename T>
T max (T a, T b) {
  T result     = (a>b)? a : b;                    //data member of type T
        // actual stuff the function does
```

Notes:

```
  return result;                    // return statement
}

#include <iostream>                 //more examples using functions
using namespace std;
int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  double q;

  k=max<int>(i, j);             //invoke template explicitly as int
  cout << k << endl;

  k=max(i, j);                      //invoke template with integers
  cout << k << endl;

  k=max<int>(5,9);                  // again invoke with literal integers
  cout << k << endl;

  n=max<long>(l, m);                //invoke explicitly with longs
  cout << n << endl;

  n=max(l, m);                      //invoke with longs not explicitly
  cout << n << endl;

  q=max(23.2,45.4);                 //invoke template with doubles
  cout << q << endl;
```

```
}
```

Notes:

Notes:

Function Template Basics

- C++ function templates define a family of functions for different datatypes.
- When a program invokes a function template the arguments passed to the function direct C++ to instantiate only the function for those datatypes from the template.
- A program can invoke a function template either by explicitly telling C++ the datatypes to use, or it can simply pass arguments to the function, leaving it up to C++ to deduce the right datatypes.
- Function templates can be overloaded, but this is best done carefully.
- When using function templates, be sure that the type of arguments used to invoke the function will work with the given set of function templates.
- Know that if an ordinary function has the same name as a function template, any call that fits this ordinary function, will use the ordinary function instead of the template, even if a template would also fit.
    - We'll discuss this later when we cover "template specialization"

Notes:

2.3 Class Templates

Why use a class template?

If a set of functions or classes have the same functionality for different data types, this becomes a good class template. The class can also hold a 'state' or data about itself. A class template can be implemented in C++ as either a class or a struct. Note that structs behave much like classes do, except that their data/methods are public by default while data/methods for a class are private by default.

Advantages:

- One C++ class template can handle different types and numbers of parameters since it has several functions that can have the same name. So the two `min` function templates above can go together in one class along with any other `add` functions needed.
- The C++ compiler generates code for only for datatypes used. If the template is instantiated for **int** type, compiler generates only an **int** version for the C++ class template.
- Templates reduce the effort of coding, testing, and documenting code for different data types.
- Testing and debugging efforts are consistent and efficient.

Notes:

How to create a class template

- The declaration of C++ class templates must begin with the keyword **template**.
- A parameter must be included inside angle brackets using either the keyword **class** or **typename**. Additional parameters can be included within the angle brackets, including non-typed parameters and other templates as parameters.
- Finally, just like any class, the class body declaration with its member data and member functions follows.

Here is a template class that stores two elements of any valid datatype.

```
template <typename T>
class aPair {
public:
   aPair (const T &f, const T &s): first (f), second (s) {  } // ← defined inline
   T max();                                    // ← defined outside class
private:
   T first; T second;
};
```

For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
aPair<int> myInts (115, 36);
```

Notes:

And this class can also create an object storing two doubles:

```
aPair<double> myDbl(3.0, 2.18);
```

Any template functions defined outside the class template body, must always use the full template definition statement, and generally must be in the same *logical* source file as the template class declaration. Here is the function implemented along with a small program using this class template.

```
template <typename T>
class aPair {
public:
    aPair (const T &f, const T &s): first (f), second (s) { }
   T max();
private :
    T first; T second;
};

template <typename T>
T aPair<T>::max ()  // <- notice the <T> after the class name
{  return first  >second ? first  : second;  }

int main () {
  aPair <int> myInts (34, 56);
  cout << myInts.max() << endl;
}
```

Notes:

Here is another example of a template class. This class has one template parameter, T. This datatype will be applied when its constructor and increase() function are called.

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

template <typename T>                    // class template
class aContainer {
public:
   aContainer (const T &arg): element (arg) {}
   T increase () {return ++element;}
private:
   T element;
};

int main () {
     aContainer<int> myint (7);
     cout << myint.increase() << endl;
     return 0;
}
```

Implementing template class member functions

Notes:

Templates declared inside classes are called member templates, and they can be either member **function** templates or member **class** templates (nested classes).

Member function templates are template functions that are members of any class or class template. Notice that member templates are not the same as template members. For example:

A **template member** is a member declared inside a **class template**.

```
template <typename T>
struct aClass  {
   T * p;                    // a template (data) member
   void f(T *) const;        // another template member (function)
};
```

On the other hand, a 'member function template' is a template with its template parameters declared inside **any** class.

Example of a member function template in a non-template class:

```
struct aClass {
        template <typename T>     // ← member function template
        void mf(T* t) ;
};

int main() {
```

Notes:

```
        int i;
        double d;
        aClass* x = new aClass();
        x->mf(&i);
        x->mf(&d);
}
```

Here's an example of declaring a member function template in a template class.

```
template<typename T> struct aClass {
        template<typename U>
        void mf(const U &u);
        // ...
 };
```

Member template functions cannot be virtual functions and cannot override virtual functions from a base class when they are declared with the same name as a base class virtual function.

Notes:

2.4 Template Specialization

There is an exception to every rule. Any generic code development will need a small case where it needs to do some hard coding or to avoid some amount generic code. That is where the C++ class template specialization comes in.

The idea of C++ class template specialization is similar to function template overloading. It fixes the template code for certain data types that need to be handled differently than most data. For example, string or char data is not handled identically to true numeric datatypes, so a specialization of an 'add' template for strings or char data may need to work differently than it would for adding integers or numbers. Once the template is specialized, all the member functions should be declared and defined for the specific data type.

When a program instantiates a template with a given set of template arguments the compiler generates a new definition based on those template arguments. To override this, instead specify the definition the compiler uses for a given set of template arguments. This is called *explicit specialization*. A program can explicitly specialize any of the following:

- Function or class template
- Member function of a class template
- Static data member of a class template
- Member class of a class template
- Member function template of a class template
- Member class template of a class template

Notes:

The **template<> prefix** indicates that the following template declaration takes no template parameters. The *declaration_name* is the name of a previously declared template. Note that one can forward-declare an explicit specialization so the *declaration_body* is optional, at least until the specialization is referenced.

Example of template specialization

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

template <typename T> class aContainer { // class template
public:
   aContainer (const T &arg) : element (arg) { }
   T increase () {return ++element;}
private:
   T element;
};

template <> class aContainer <char> { // class template specialization:
public:
   aContainer (char arg): element (arg) { }
   char uppercase ();  // ← note how we've added a totally new method!
private:
   char element;
};
```

Notes:

```
// member of class template specialization:
char aContainer<char>::uppercase()  {
  if ((element >= 'a') && (element <= 'z'))
    element += 'A' - 'a';
  return element;
}

int main () {
  aContainer<int> myint (7);
  aContainer<double> mydouble (10.5) ;
  aContainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mydouble.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

Partial Specialization

Class templates can also be partly specialized. Consider a class template like this:

```
template <typename T, typename U>
class aClass {
  // …
};
```

Notes:

This template takes arguments of 2 different datatypes, so it can be partially specialized in several ways.

```
#include <iostream>

template <typename T, typename U>
struct aClass
{
        aClass () { std::cout << "aClass <T, U>\n"; }
};

template <typename T>     //specialize so that both are same datatype
struct aClass <T, T> {
        aClass () { std::cout << "aClass<T, T>\n"; }
};

template <typename T>     //specialize so that second datatype is int
struct aClass <T, int > {
        aClass () { std::cout << "aClass<T, int>\n"; }
};

template <typename T, typename U>   // both are now pointers
struct aClass <T*, U*> {
  aClass () { std::cout << "aClass<T*, U*>\n"; }
};
```

Notes:

```
int main(void)
{
        aClass <int, double> a;        // prints “aClass<T, U>”
        aClass <double, double> b; // prints “aClass<T, T>”
        aClass <double, int> c;        // prints “aClass<T, int>”
        aClass <int *, double *> d;    // prints “aClass<T*, U*>”
        aClass <int, int> e; // prints??
        return 0;
}
```

We can fix this ambiguity by adding the following partial specialization:

```
template <>
struct aClass <int, int> {
  aClass () { std::cout << "aClass<int, int>\n"; }
};
```

Notes:

2.5 Default Template Parameters

Templates classes (but not template functions), can take default template parameters.

```
template <typename T = int>
class aStack {
  public:
    // …
  private:
    vector<T> s;
};
```

And then a program can use those defaults when instantiating the template:

```
// use default so aContainer is a vector of doubles.
aStack<double> aStackofdoubles;

//don't use default, now a vector of ints.
aStack< > aStackofints;
```

Notes:

2.6 Non-Type Template Parameters

There are three types of parameters that can be used with templates:

- Typed template parameters
- Non-type template parameters
- Template template parameters

As discussed in the previous sections, templates can have regular **typed** parameters, similar to those found in functions. Typed template parameters are preceded by the keyword **typename** or **class**.

Templates can also have regular parameters – these are called 'non type' template parameters.

- The syntax of a non-typed template parameter is the same as a declaration of one of the following types:
  - integral or enumeration
  - pointer to object or pointer to function
  - reference to object or reference to function
  - pointer to member.

- A program may qualify a non-type template parameter with **const** or **volatile**.

- Non-type template parameters have restrictions: they must be integral values, enumerations, or instance pointers with external linkage. They

Notes:

can't be string literals nor global pointers since both have internal linkage. Nor can they be floating point, typename or void type.

- Non-type template parameters are not lvalues – i.e., they are simply 'plain old literal data'

Non-type Class Template Parameters

For example, int N is a non-typed template parameter.

```
#include <iostream>  using namespace std;
template <typename T, int N>  // ← typed param T, non-type param N
class aClass {
public:
    void setmem (int x, T value);
    T getmem (int x);
private:
   T memblock [N];
};
int main () {
   aClass <int,5> myints;
   aClass <double,15> myfloats;
   myints.setmem (0,100);
   myfloats.setmem (3,3.1416);
   cout << myints.getmem(0) << endl;
   cout << myfloats.getmem(3) << endl;
   return 0;
}
```

Notes:

Non-Type Function Template Parameters

A program can also define non-type function template parameters.

```
template <typename T, int V>
T Add (const T & n) {
   return n + V;
}

#include <iostream>
using namespace std;

int main () {
  int I;
  I = Add<int,6> (10);
}
```

Notes:

Deducing type of template parameters

The compiler cannot deduce the value of a major array bound unless the bound refers to a reference or pointer type. Major array bounds are not part of function parameter types. The following code demonstrates this:

```
template<int i> void f(int a[10][i]) { }
template<int i> void g(int a[i]) { }
template<int i> void h(int (&a)[i]) { }

int main () {
   int b[10][20];
   int c[10];
   f(b);
   // g(c);
   h(c);
}
```

The compiler would not allow the call g(c); the compiler cannot deduce template argument i.

The compiler cannot deduce the value of a non-type template argument used in an expression in the template function's parameter list. The following example demonstrates this:

```
template<int i> class X { };
template<int i> void f(X<i - 1>) { }
```

Notes:

```
int main () {
  X<0> a;
  f<1>(a);
  // f(a);
}
```

To call function f() with object a, the function must accept an argument of type X<0>. However, the compiler cannot deduce that the template argument i must be equal to 1 in order for the function template argument type X<i - 1> to be equivalent to X<0>. Therefore the compiler would not allow the function call f(a).

To enable the compiler to deduce a non-type template argument, the type of the parameter must match exactly the type of value used in the function call. For example, the compiler will not allow the following:

```
template<int i> class A { };
template<short d> void f(A<d>) { }

int main() {
   A<1> a;
   f(a);
}
```

The compiler will not convert int to short when the example calls f(). However, deduced array bounds may be of any integral type.

Notes:

2.7 Template Template Parameters

A template can take a parameter that is itself the name of a template. These parameters have the name of template template parameters. Let's use a simple example. Start with a template for a stack that would accept a type of data to be stored and a container type to adapt into a stack.

The first parameter, T, is just the name of a datatype as usual. The second parameter, aContainer, is a 'template template' parameter. It's the name of a class template that has a single type name parameter, and we didn't specify a type of data contained in the original container.

The aStack template uses its 'type name' parameter to instantiate its 'template template' parameter. The resulting container type is used to implement the aStack object:

```
template <typename T, template <typename T> class aContainer = deque>
class aStack {
public:
   // …
private:
   aContainer<T> s;
};
```

Notes:

This approach allows coordination between the element and its container to be handled by the implementation of the aStack itself, rather than in all the various code that specializes the aStack class.

This single point of specialization reduces the possibility of mismatches between the element type and the container used to hold the elements.

So we can create an aStack object from a list or from a deque…as follows…and the 'aStack' created from the list will hold integers while the aStack created from the deque will hold strings. And all of this resulted from allowing the aStack template to have a parameter that is itself a template.

```
aStack<string> aDequeStackofStrings; // ← defaults to deque
aStack<int, list> aListStackofInts;
```

Notes:

2.8 Other Template Related Topics

Friends and Templates

There are four kinds of relationships between classes and their friends when templates are involved:

- One-to-many: A non-template function may be a friend to all template class instantiations.
- Many-to-one: All instantiations of a template function may be friends to a regular non-template class.
- One-to-one: A template function instantiated with one set of template arguments may be a friend to one template class instantiated with the same set of template arguments. This is also the relationship between a regular non-template class and a regular non-template friend function.
- Many-to-many: All instantiations of a template function may be a friend to all instantiations of the template class.

Here are some examples showing of these relationships:

```
class B{
  template<typename V> friend int j();
}
```

- Function j() has a many-to-one relationship with class B. All instantiations of j() are friends with class B.

Notes:

```
template<typename S> int g();

template<typename T> class A {
  friend int e();
  friend int f(T);
  friend int g<T>();
  template<typename U> friend int h();
};
```

- Function e() has a one-to-many relationship with class A. Function e() is a friend to all instantiations of class A.
- Function f() has a one-to-one relationship with class A. The function declares only one function because it is not a template but the function type depends on the template parameter. The compiler will give you a warning for this kind of declaration similar to the following: The friend function declaration "f" will cause an error when the enclosing template class is instantiated with arguments that declare a friend function that does not match an existing definition.
- Function g() has a one-to-one relationship with class A. Function g() is a function template. It must be declared before here or else the compiler will not recognize g<T> as a template name. For each instantiation of A there is one matching instantiation of g(). For example, g<int> is a friend of A<int>.
- Function h() has a many-to-many relationship with class A. Function h() is a function template. For all instantiations of A all instantiations of h() are friends.

These relationships also apply to friend classes.

Notes:

Nested Classes and Templates

A nested class is a class enclosed within the scope of another class. They are used when objects are needed by a class but no code outside the class needs to know about these objects. They help organize code and controlling access and dependencies. They can also be templates. For example:

```
template <typename T>
class aClass
{
public:
        template<typename U>        ← member function template
        void mf(const U &u) { }

private:
        template <typename U>              ← member class template
        class Nested { /* … */ };
};
```

Local classes

A local class is a class defined within the scope of a **function**, whether it's a class member function or a standalone function. But a local or unnamed class cannot be used as a template parameter.

Notes:

Macros and Templates

In many ways, templates work like preprocessor macros, replacing the templated variable with the given type. However, there are many differences between a macro and a template:

```
#define min(i, j) (((i) < (j)) ? (i) : (j))

template<typename T> T min (const T &i, const T &j)
{ return ((i < j) ? i : j) }
```

Here are some problems with the macro:

- The compiler does not verify that the macro parameters are of compatible types because macros don't do type checking.

- In the macro version, the i and j parameters are evaluated twice, e.g.,

  ```
  int a = 10, b = 20;
  int c = min (a++, b++);
  ```

- Because macros are expanded by the preprocessor, compiler error messages will refer to the expanded macro, rather than the macro definition itself. Also, the macro will show up in expanded form during debugging so programmers can't see its code during debugging, e.g., debuggers can't set breakpoints in macros.

Notes:

Templates and void Pointers

Many functions implemented with void pointers could be implemented with templates instead. The disadvantages of using void pointers are that the C++ compiler cannot distinguish between datatypes, so it cannot perform type checking or type-specific behavior such as using type-specific operators, operator overloading, or constructors and destructors.

With templates, programmers can create functions and classes that operate on *typed* data. While the template appears abstract because it is generic, when code invokes the template, the compiler creates a separate version of the function for each specified type and will instantiate only the template functions used. This enables the compiler to treat class and function templates as if they acted on specific types. Templates can also improve coding clarity, because programmers do not need to create special cases for classes.

With some understanding of templates, let's take a look at the specific templates offered in the C++ Standard Library, as the 'Standard Template Library'.

Notes:

2.9 Using Templates

Inclusion Model

When working with ordinary classes and functions, C++ programmers have learned to place the class and function declarations into header files. Then the function implementation and any global variables are defined in separate source files which #include the class header file. This allows the program to link together the class declarations and definitions plus any other program needing the class only needs to #include the class header files.

For template functions, however this method will not work. Why? Because when a program statement invokes the template function, C++ needs to be able to instantiate the function based on the template or blueprint of that function family. However, when the C++ compiler looks for a definition in the header file it cannot find one (it's in the implementation source file); and as the compiler processes the implementation source file it doesn't know what type of datatype function will be instantiated later in the program file.

Processing the program file, the compiler assumes that the template function will be defined elsewhere and leaves a reference for the linker to locate where that is. When the linker comes along to process all the compiled object files together into an exe file, it discovers the reference to 'find' the function definition and it cannot put the two files (header and source) together to find it.

Notes:

One way that programmers have solved this is to place all the template functions and their full implementation in the same file or to #include the source file at the end of the header file. This method has now increased the cost of using this template header file due to its increased size and the size of any other header files, such as <iostream> that are commonly needed in the function implementations.

Note that non-inline functions create a new copy of the function each time they are instantiated, rather than being expanded at the site of the call. Occasionally when creating large libraries of functions, this can lead to two copies of the function in different files.

All of this discussion applies to member template functions, and member functions and data members of class templates. Place declarations and definitions in the same file or #include the source file at the end of the header file.

Explicit Instantiation

Another way to prevent linker and compilation errors is to explicitly provide each template argument when invoking a template function. For example in a file called printDataType.h, create and define a small template function:

```
#include <iostream>
#include <typeinfo>
using namespace std;
```

Notes:

```
template <typename T>
void printDataType (const T& a) { cout << typeid(a).name() << endl; }
```

Here are examples of using this particular function template in a program plus other templates; all being explicitly instantiated.

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

template <typename T>
void printDataType (const T& a)  {  cout << typeid(a).name() << endl; }

template <typename T>
struct aClass
{
  aClass () { }
  T result (T t) { return t; }
  void add (const T &t) {}
};

// explicitly instantiate this function template for types int and double
template void printDataType<int>(const int &);
template void printDataType<double>(const double &);

// explicitly instantiate a template class' constructor for int data
```

```
template aClass<int>::aClass();

// explicitly instantiate an entire template class and all its functions
// later cannot explicitly instantiate particular functions of this class
// because ALL are instantiated by this statement
template struct aClass<double>;

// explicitly instantiate just some member functions
template aClass<string>::aClass();
template void aClass<string>::add(const string  &);
template string aClass<string>::result(string);
```

Notes:

Separation Model

Exported template functions can be used without their definition being visible; so a program can have the template declaration a header file and function implementations in a source file (not yet supported by Visual Studio or GNU G++).

The export keyword must precede the keyword template and cannot be combined with the keyword inline. Classes can use the keyword export but it means that their functions are exported unless inline.

So printDataType.h contains:

```
export
template <typename T>
```

Notes:

```
void printDataType (const T& a);
```

And in printDataType.cpp, the export keyword doesn't need to be present:

```
#include <iostream>
#include <typeinfo>
#include "printDataType.h"
   using namespace std;
template <typename T>
void printDataType (const T& a) {
     cout << typeid(a).name() << endl;
}
```

2.10 Components of the Standard Template Library

Notes:

The STL has five main types of components, all of which use templates heavily:

- **Containers** store collections of objects. Examples include `vector`, `list`, `deque`, `set`, `multiset`, `map`, `multimap`, `unordered_set`, `unordered_multiset`, `unordered_map`, and `unordered_multimap`.

- **Iterator objects** are for accessing the objects in a container (like an index, but are generalization of pointers).

- Generic **algorithms**, such as `count()`, `search()`, and `sort()`, for work with objects using iterators, but can also be used with non-STL C++ code.

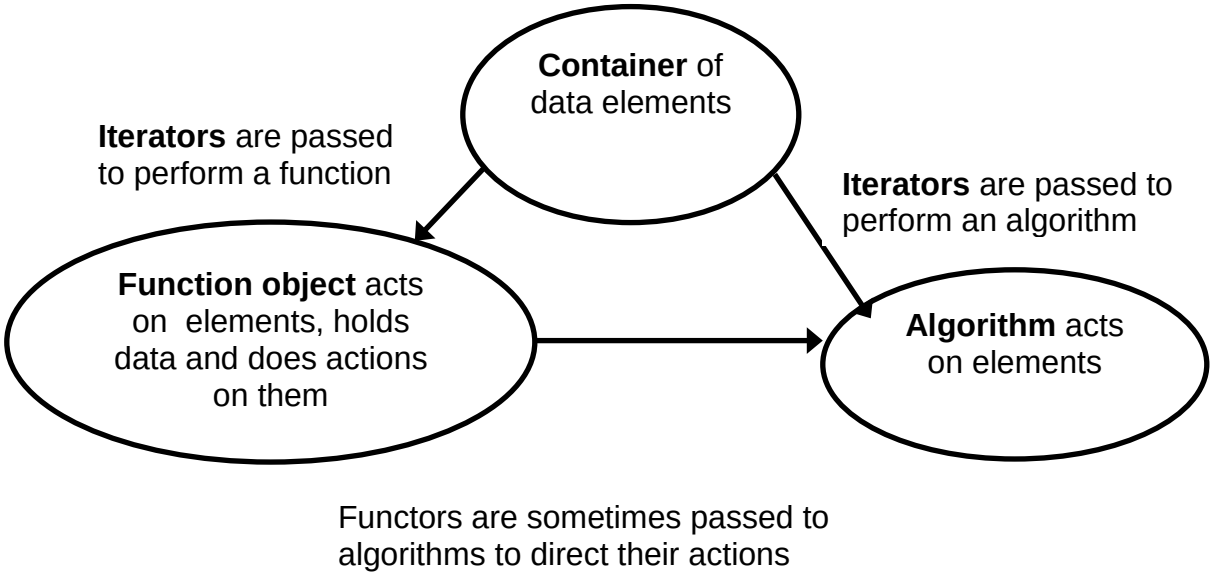
- **Functors** generalize functions and can manipulate the algorithms and put them together to create more complicated processes. They are an important part of generic programming because they allow abstraction not only over the types of objects, but also over the operations that are being performed.

- **Utilities**: The STL also includes some low-level **utilities** for allocating and de-allocating memory; these are called allocators.

Notes:

An STL data structure, called a container, only has a minimal set of operations for creating, copying, and destroying itself plus operations for adding and removing elements.

Container member functions don't evaluate their elements or change their order. Instead, algorithms and functors work on a container's elements using smart pointers called iterators, to access their values and modify the order of elements.



Unit Three
Sequential Containers

Unit topics:

Notes:

3.1 Containers

What are Containers?

The STL containers are classes whose objects can be used to build collections of data of same type, like built-in C++ arrays. But these collections also have data and functions because they are objects. The string, bitset, and valarray classes are very similar to these containers and will be discussed here as well.

Common characteristics

1. They copy the values of their elements when inserted rather than maintaining references to these elements. This is called 'value semantics'. Thus anything added to an STL container must be able to be copied; a user defined class must define a public copy constructor or to use the class objects with an STL container, pointers to the objects must be stored in the container.

2. All elements added to "classic" STL containers have an order; in sequential containers this is maintained by the order added to the container; in associative containers, a key value provides the ordering.

3. Operations on the elements of an STL container are not tested by the container for being correct or safe: a program using the STL must

provide its own exception handling code for any errors that might occur.

Notes:

Common Operations

1. **Initialization constructors**: Each container supports both default and copy constructors, plus a destructor. Containers can be initialized with a range of values as well.

2. **Size functions**: There are 3 functions related to a container's size. They are: size(), which returns the actual number of elements in the container, empty(), checks whether there are any elements at all in the container, and max_size(), which returns the maximum number of elements the container can contain.

3. **Comparison operators**: The equality and relational operators, ==, !=, <, <=, >, >= are defined for containers holding the same datatype as elements. Two containers are equal if all elements are equal and in the same order. The relational operators work by comparing the containers element by element, and if one container has fewer elements it is 'less than' the other. Or if a container's element value is less than the value of that element at the same index in the other container, then this contain is 'less than' the other one. If the comparison test runs through both containers and they are of equal s size with identical elements in the same order, they are equal and the relational operator returns false for whatever it is testing for.

Notes:

4. **Assignments and swap() function**: When one container is assigned to another one, all elements in the source container are copied by value into the destination container, and all old elements are removed. This takes time. A faster way to get this done if the source container will not be needed afterward, is to use the swap() function instead. This will swap the internal pointers to the container's elements, allocator and sorting criteria, and is very fast and exception-safe.

Container Categories

1. **Sequential containers**. These containers arrange the data they contain in a linear manner.

2. **Associative containers**. These containers maintain data in structures suitable for fast associative look-up. STL now supports both ordered and unordered associative containers.

3. **Adapters.** These are containers that provide different ways to access sequential and associative containers' data.

Notes:

Here they are in more detail:

Category	Containers	Characteristics
Sequential	vector	Linear and contiguous storage like an array that allows fast insertions and removals at the end only.
	list	Doubly linked list that allows fast insertions and removals anywhere
	forward_list	Single linked list that allows fast insertions and removals anywhere
	deque	Linear but non-contiguous storage that allows fast insertions and removals at both ends.
Associative (both ordered & unordered)	set	Defines where the elements' values are the keys and duplicates *are not* allowed. It has fast lookup using the key
	multiset	Defines where the elements' values are the keys and duplicates *are* allowed. It has fast lookup using the key
	map	Key-to-value mapping where a single key can only be mapped to one value
	multimap	Key-to-value mapping where a single key can be mapped to many values
Adapter	stack	First in, last out data structure
	queue	First in, first out data structure
	priority_queue	Queue that maintains items in a sorted order based on a priority value

Notes:

Non STL containers in the standard C++ library

Besides the powerful string class, there are also several other non-STL containers in the Standard C++ library. These can sometimes use the STL's iterators, algorithms, and functors.

The **bitset** container packs bits into integers and does not allow direct addressing of its members. The program can perform bitwise arithmetic on these elements and, as a result, they are often used as flags.

The **valarray** template class is a vector-like container and allows programs to have arithmetic statements referencing the container itself such as a = b + c where a, b, c are all valarray containers AND each element of these containers has the math performed on it.

The **complex** template in the C++ Standard Library supports complex numbers along with their specific mathematical definitions for addition, subtraction, multiplication and division.

The **string** container in the C++ Standard Library often acts much like a form of a container of char data, and can use the STL iterators, algorithms and functions.

Now that we know generally what the concept of a container is, how do we operate on their elements? We use a type of pointer that is an object of a class. These pointers, called iterators in the STL, can know things (have data members), and perform their own functions.

Notes:

3.2 Sequential Containers

Sequential containers are collections of data elements placed in some order, usually according to when the element was added to the container. The order of the elements has nothing to do with their value. The non-vector sequential containers are something like arrays, but don't have to be physically contiguous in storage.

The STL defines four types of sequential collections: vector, deque, list, and forward_list. And programs can also use the non-STL collections like string, bitset, and valarray in similar ways that they use STL sequential containers.

Sequential containers are better than simple C/C++ arrays because:

- They have a size() member function
- They provide a past end entry iterator
- They provide copy constructors
- They allow assignment
- Can be passed by value
- They grow and shrink dynamically, i.e., they know how to cleanup after themselves because they have destructors

Notes:

➢ 3.3 Vector

A **vector** (**#include <vector>**) is a collection of elements of type T, where T can be integer, double, char or any object. It is a model of a dynamic array that grows and shrinks at the end. A vector is a sequential container. As such, its elements are ordered following a strict linear sequence.

Vector containers are implemented as dynamic arrays; Just as regular arrays, vector containers have their elements stored in contiguous storage locations, which means that their elements can be accessed not only using iterators but also using offsets on regular pointers to elements.

But unlike builtin C/C++ arrays, storage in vectors is handled automatically, allowing it to be expanded and contracted as needed.

Vectors are efficient for:

- Accessing individual elements by their position index.
- Iterating over the elements in any order.
- Adding and removing elements from the end.

Compared to builtin C/C++ arrays, they provide almost the same performance for these tasks, plus they have the ability to be easily resized. They usually consume a bit more memory than arrays, however, because their capacity is handled automatically, to allow for extra storage space for future growth.

Notes:

Compared to the other STL sequential containers, vectors are generally the fastest for accessing elements and to add or remove elements from the end of the sequence.

Internally, vectors have a size, which represents the amount of elements currently contained in the vector. Vectors also have a capacity, which determines the amount of storage space allocated, and which can be either equal or greater than the actual size. The extra amount of storage allocated is not used, but is reserved for the vector to be used in the case it grows. This way, the vector does not have to reallocate storage on each occasion it grows, but only when this extra space is exhausted and a new element is inserted.

Reallocations are costly operations in terms of performance, since they generally involve copying all values used by the vector to be copied to a new location. Therefore, whenever large increases in size are planned for a vector, it is recommended to explicitly indicate a capacity for the vector using the member function vector::reserve().

Notes:

The vector implementation in the C++ Standard Template Library take two template parameters:

```
template < typename T, typename Allocator = allocator<T> > class vector;
```

1. **T**: Datatype of the elements that can be stored in the vector.

2. **Allocator**: This is the allocator object used to define the storage allocation model. By default, the Allocator class template from <memory> for type T is used, which defines the simplest memory allocation model and is value-independent.

Characteristics of vectors:

- Random access to elements
- Constant time insertion and removal of elements at end
- Linear time insertion of elements at beginning or middle because other elements have to be moved
- Number of elements can vary dynamically
- Simplest and most efficient type of STL container
- Member function reserve() can pre-allocate all memory needed
- Assignment, copy constructor, and destructor supported

One can use a vector<int> like one would use an ordinary C array, except that vector eliminates the chore of managing dynamic memory allocation.

Notes:

Efficiency Tips for vectors

- Provides rapid indexed access with the overloaded subscript operator, [ ], because they are stored in contiguous memory like a C or C++ raw array.

- Supports random-access iterators. All STL algorithms can operate on a **vector**. The iterators for a **vector** are normally implemented as pointers to elements of the **vector**.

- It is faster to insert many elements at once than one at a time.

- **Vector** is the type of sequenced template that should be used by default if you don't need to access both ends of the collection and you don't need to traverse it backwards.

Some useful vector functions:

vector()	creates empty vector
vector(size_type n)	creates a vector with n elements
vector(size_type n, const T& t)	creates a vector with n copies of t
vector(const vector&)	creates a vector from another vector
vector(InputIterator, InputIterator)	creates a vector from a range of values
bool empty() const	returns true of vector is empty
reference operator[](size_type n)	returns the nth element (doesn't check)
reference at(size_type n);	returns the nth element (does check)
reference front()	returns the first element

reference back()	returns the last element
void push_back(const T&)	adds a new element at the end
void pop_back()	removes the last element
void insert(iterator, value_type)	inserts value prior to iterator (inefficient)
iterator erase(iterator pos)	removes the element at position pos
iterator erase(iterator first, iterator last)	removes the range of elements
bool operator==(const vector&, const vector&)	test two vectors for equality
size_type capacity() const	number of elements memory allocated
void reserve (size_type)	increase number of elements allocated

Data member	**Type definition**
reference	Allocator::reference
const_reference	Allocator::const_reference
iterator	Random access iterator
const_iterator	Constant random access iterator
size_type	Unsigned integral type (usually same as size_t)
difference_type	Signed integral type (usually same as ptrdiff_t)
value_type	T
allocator_type	Allocator
pointer	Allocator::pointer
const_pointer	Allocator::const_pointer
reverse_iterator	Random access reverse iterator
const_reverse_iterator	Constant random access reverse iterator

Notes:

Notes:

Here is an example using a vector declared to hold 3 integers. One way to initialize a vector's values is to directly access them just like a C-style array would be accessed. Arithmetic can be performed on vector elements and the result assigned to another element:

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

int  main()
{
     vector<int> v(3);
     v[0] = 7;
     v[1] = v[0] + 3;
     v[2] = v[0] + v[1];              // v[0] == 7, v[1] == 10, v[2] == 17
    return 0;
}
```

Here is another example of using a vector, where the vector's methods push_back() and size(), are used along with an index.

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

int main()
{
       vector <int> myVector;
```

Notes:

```
        // append elements with values 0-6
        for (int i=0; i < 7; ++i)
                myVector.push_back(i);

        // print elements separated by a space
        for (int i=0; i<myVector.size() ; ++i)
                cout << myVector[i] << ' ';

        cout << endl;
}
```

What if we need a vector of doubles? It's just as easy to create and use:

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

int main() {
        vector<double> aVector;
        aVector.push_back(1.2);
        aVector.push_back(4.5);

        aVector[1] = aVector[0] + 5.0;
        aVector[0] = 2.7;          // now it has two elements: 2.7, 6.2
        return 0;
}
```

Notes:

Here is an example of vectors being created using the C-style array and char pointer:

```
#include <string.h>
#include <vector>
using namespace std;

int main( )  {
   int ar[10] = {  12, 45, 234, 64, 12, 35, 63, 23, 12, 55  };
   const char* str = "Hello World";
   vector<int> vec1(ar, ar+10);
  // In C++11 you can say
  // vector<int> vec1 ( {12, 45, 234, 64, 12, 35, 63, 23, 12, 55  });
   vector<char> vec2(str, str+strlen(str));
  // In C++11 you can say
  // vector<char> vec2 ({'H', 'e', 'l', 'l', 'o'});

   return 0;
}
```

Notes:

Here is an example showing how to create and use a vector that contains another vector inside it…a vector of vectors, or a 2-dimensional vector.

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

int main()  {
  vector< vector<int> > myVec2D;          // create 2D int vector
  vector<int> oneVec, twoVec;             // create two int vectors


  oneVec.push_back(1); oneVec.push_back(2); oneVec.push_back(3);
  twoVec.push_back(4); twoVec.push_back(5);  twoVec.push_back(6);

  myVec2D.push_back(oneVec); myVec2D.push_back(twoVec);

  cout << endl << "Using Iterator:" << endl;
  for(vector< vector<int> >::iterator iter2D= myVec2D.begin();
       iter2D!= myVec2D.end(); ++iter2D)
    for(vector<int>::iterator iter =iter2D->begin();
         iter!=iter2D->end(); ++iter)
            cout << *iter << endl;

  cout << endl << "Using subscript operators:" << endl;
  for (size_t i = 0; i < myVec2D.size (); ++i)
    for (size_t j = 0; j < myVec2D[i].size (); ++j)
            cout << myVec2D[i][j] << endl;
```

Notes:

```
   cout << endl << "Using range-based for loop:" << endl;
   // C++11 version
   for (auto inner_vec : myVec2D)
      for (auto i : inner_vec)
        cout << i << endl;

return 0;
}
```

Notes:

Here is another simple example of a vector used with an iterator. Notice that one can access the elements of the container directly by dereferencing the iterator.

```
#include <vector>
#include <iterator>
#include <iostream>
#include <numeric>
using namespace std;

int main() {
  vector<int> aVector;

  for( int i=0; i < 10; ++i)                    // put some values in vector
       aVector.push_back(i);

  int total = 0;
  for (vector<int>::iterator anIterator = aVector.begin();  // set iterator at start
       anIterator != aVector.end();
       ++anIterator)        // process vector  using iterator
    total += *anIterator;                    // add up values stored

  cout << "Total=" << total << endl; // display the total of values

  cout << "Total=" << std::accumulate(aVector.begin(), aVector.end(), 0)
        << endl;
```

```
  return 0;
}
```

Notes:

Notes:

Vector specialization: vector<bool>

The vector class template has a special template specialization for the bool data type. This specialization is provided to optimize for space allocation: In this template specialization, each element occupies only one bit.

The references to elements of a bool vector returned by the vector members are not references to bool objects, but a special member type which is a reference to a single bit, defined inside the vector<bool> class specialization as:

```
template <> class vector<bool>::reference {
  friend class vector;
  reference();                                  // no public constructor
public:
  ~reference();
  operator bool () const;                       // convert to bool
  reference& operator= ( const bool x );        // assign from bool
  reference& operator= ( const reference& x ); // assign from bit
  void flip();                                  // flip bit value.
}
```

Notes:

3.4 deque

A **deque** (**#include <deque>,** pronounced like deck), is a double-ended queue which grows and shrinks at both ends quickly. Insertions and deletions can be made in the middle but these are slower. They are another type of sequential container with their elements ordered following a strict linear sequence.

Deques may be implemented by specific libraries in different ways, but in all cases they allow for the individual elements to be accessed through random access iterators, with storage always handled automatically (expanding and contracting as needed).

Deque containers have the following properties:

- Individual elements can be accessed by their position index.
- Iteration over the elements can be performed in any order.
- Elements can be efficiently added and removed from any of its ends (either the beginning or the end of the sequence).

Therefore deques provide a similar functionality to vectors, but with efficient insertion and deletion of elements at the beginning and end of the sequence. However, unlike vectors, deques don't have all elements in contiguous storage locations, thus eliminating the possibility of traversing the deque container with simple pointer arithmetic.

Notes:

Both vectors and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors are very similar to a C-style array that grows by reallocating all of its elements in a unique block when its capacity is exhausted, the elements of a deques can be divided in several chunks of storage, with the class keeping all this information and providing uniform access to the elements.

Deques are a little more complex internally, but this generally allows them to grow more efficiently than vectors with their capacity managed automatically; this is especially noticeable in large sequences, because massive reallocations are avoided.

The deque implementation in the C++ Standard Template Library take two template parameters:

```
template <typename T, typename Allocator = allocator<T> > class deque;
```

1. **T**: Datatype of the elements to be stored.

2. **Allocator**: This is the allocator object used to define the storage allocation model. By default, the Allocator class template from <memory> for type T is used, which defines the simplest memory allocation model and is value-independent.

Notes:

All of the member function implemented in **vector** are also implemented in **deque** except for **capacity()** and **reserve()**. Also included are two new functions:

void push_front(const T&)	adds an element to the front
void pop_front()	removes the first element

Data member	**Type definition**
reference	Allocator::reference
const_reference	Allocator::const_reference
iterator	Random access iterator
const_iterator	Constant random access iterator
size_type	Unsigned integral type (usually same as size_t)
difference_type	Signed integral type (usually same as ptrdiff_t)
value_type	T
allocator_type	Allocator
pointer	Allocator::pointer
const_pointer	Allocator::const_pointer
reverse_iterator	Random access reverse iterator
const_reverse_iterator	Constant random access iterator

Notes:

Efficiency of deque vs. vector:

- When performing a large number of **push_back()** calls, use **vector::reserve()**.
- When performing many de-allocations, **deque** takes longer to reclaim memory than **vector** since it is allocated using multiple "chunks."
- When using **insert()** or **pop_front()** with a **deque** is more efficient than **vector**.
- For element access, **vector::at()** or vector[] is better than **deque's at()** or deque[] methods.

Here is a simple example showing how to create a deque, add elements to it using an index, and then display those elements using an index. Here we process the deque as though it were any container, using its push_back function, but without using its iterator.

```
#include <deque>
#include <iostream>
using namespace std;

int main( ) {
        deque <float> myDeck;

        // can also use the push_back( )
        for (size_t i=0; i < 7; i++)
```

Notes:

```
            myDeck.push_front(i * 1.1);
        for (size_t i=0; i < myDeck.size(); i++)
            cout <<  myDeck[i] << ' ' ;
        cout << endl;
        return 0;
}
```

Here is another deque example:

```
#include <iostream>
#include <deque>
using namespace std;

int main()  {
  deque<char> aDeck;

  for(size_t i = 0; i <5; i++)
     aDeck.push_back(i + 'A');

  cout << "Original sequence: ";
  for(size_t i = 0; i <aDeck.size(); i++)
    cout << aDeck[i] << " ";
  cout << endl;

  deque<char>::iterator It1 = aDeck.begin() + 2; // Note random access
  deque<char>::iterator It2 = aDeck.begin() + 3; // iterators here
  cout << "*It1: " << *It1 << ", ";
```

Notes:

```
  cout << "*It2: " << *It2 << endl;
  cout << endl;

  aDeck.insert(It1, 'X');

  cout << "Sequence after insert: ";
  for(size_t i = 0; i <aDeck.size(); i++)
    cout << aDeck[i] << " ";
  cout << endl;

  // These iterator dereferences may cause the program to crash since STL
  // does not implement “robust iterators”..
  cout << "*It1: " << *It1 << ", ";
  cout << "*It2: " << *It2 << endl;

  return 0;
}
```

Here is a third example, using the deque functions push_back, insert, begin, and end along with the **copy** algorithm to copy the members to the output stream.

```
#include <deque>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;
```

Notes:

```
int main() {
    deque<int> aDeck;
    aDeck.push_back(3);
    aDeck.push_front(1);
    aDeck.insert(aDeck.begin() + 1, 2);
    aDeck[2] = 0;
    copy(aDeck.begin(), aDeck.end(), ostream_iterator<int>(cout, " "));
    // Could call print(aDeck) here as well (as per method defined below).
    return 0;
}
```

Here is an example of a deque with its const iterator:

```
void print(const deque<string> &d) {
   cout << "The number of items in the deque:" << d.size() << endl;

   for (deque<string>::const_iterator  iter = d.begin(); iter != d.end(); ++iter)
       cout << *iter << " ";

   cout << endl << endl;
}
```

On the next page is a more complicated example using two deques of characters. In this example, after elements are added to both deques, their sizes are printed out. Then they are swapped using a deque member

Notes:

function and their sizes now printed out. Finally the swap STL algorithm is used to swap their elements.

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

int main()
{
  deque<char> myFirstDeck, myNextDeck;

  for(size_t i = 0; i <26; i++)
     myFirstDeck.push_back(i+'A');

  for(size_t i = 0; i <10; i++)
     myNextDeck.push_front(i+'0');

  cout << "Size of myFirstDeck and myNextDeck: ";
  cout << myFirstDeck.size() << " " << myNextDeck.size() << endl;
  cout << "myFirstDeck: ";

  for(size_t i = 0; i <myFirstDeck.size(); i++)
        cout << myFirstDeck[i];
  cout << endl << "myNextDeck: ";
```

```
   for(size_t i = 0; i <myNextDeck.size(); i++)
       cout << myNextDeck[i];
  cout << "\n\n";

  // swap deque elements using member function.
  myFirstDeck.swap(myNextDeck);

  cout << "Size of myFirstDeck and myNextDeck after first swap: ";
  cout << myFirstDeck.size() << " " << myNextDeck.size() << endl;

  cout << "myFirstDeck after first swap: ";
  for(size_t i = 0; i <myFirstDeck.size(); i++)
    cout << myFirstDeck[i];
  cout << endl;

  cout << "myNextDeck after first swap: ";

  for(size_t i = 0; i <myNextDeck.size(); i++)
    cout << myNextDeck[i];
  cout << "\n\n";

  swap(myFirstDeck, myNextDeck);

  return 0;
}
```

Notes:

Notes:

3.5 list

A **list** (**#include <list>**) is a sequential collection of elements of type T. List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept by the association to each element of a link to the element preceding it and a link to the element following it. This implementation provides the following advantages to list containers:

- Efficient insertion and removal of elements anywhere in the container.
- Efficient moving elements and block of elements within the container or even between different containers.
- Iterating over the elements in forward or reverse order.

Compared to vectors and deques, lists perform generally better in inserting, extracting and moving elements in any position within the container, and therefore also in algorithms that make intensive use of these features.

The main drawback of lists compared to these other sequential containers is that lists don't provide direct (i.e., random) access to the elements by their position. Thus, to access the sixth element in a list, a program must iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these.

Notes:

List also use extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements). As with deques, however, storage is handled automatically by the class, allowing lists to be expanded and contracted as needed. If you are concerned about the space overhead of the extra pointer needed to implement a doubly-linked list then consider using the STL forward_list.

Characteristics of lists

- Has all the functions that **vectors** have except **capacity()**, **reserve()**, **at()**, and the **[ ]** operator.
- Does not support random access iterators but does support bidirectional iterators, which allows both forward and backward transversal.
- Constant time insertion and removal of list elements anywhere (assuming an iterator points to the desired location).
- Iterators to deleted elements are invalid but inserting elements does not invalidate iterators

Notes:

The list implementation in the C++ Standard Template Library take two template parameters:

```
template <typename T, typename Allocator = allocator<T> > class list;
```

1. **T**: Datatype of the elements to be stored.

2. **Allocator**: This is the allocator object used to define the storage allocation model. By default, the Allocator class template from <memory> for type T is used, which defines the simplest memory allocation model and is value-independent.

Data Member	**Type Definition**
reference	Allocator::reference
const_reference	Allocator::const_reference
iterator	Bidirectional iterator
const_iterator	Constant bidirectional iterator
size_type	Unsigned integral type (usually same as size_t)
difference_type	Signed integral type (usually same as ptrdiff_t)
value_type	T
allocator_type	Allocator
pointer	Allocator::pointer
const_pointer	Allocator::const_pointer
reverse_iterator	Bidirectional reverse iterator
const_reverse_iterator	Constant bidirectional reverse iterator

Notes:

Useful functions of list

begin()	Returns iterator pointing to first element
end()	Returns iterator pointing _after_ last element
push_front(...)	Add element to front of list
pop_front(...)	Destroy element at front of list
push_back(...)	Add element to end of list
pop_back(...)	Destroy element at end of list
swap(,)	Swap two elements
erase(...)	Delete elements
insert(,)	Insert new element
size()	Number of elements in list
empty()	True if list is empty
sort() list; <algorithm>	Other STL sort algorithms expect random access iterators

Here is a simple example using a list:

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

int main() {
   list<char> aList;
   for(int i = 0; i <10; i++)
     aList.push_back('A'+i);
```

Notes:

```
  cout << "Size = " << aList.size() << endl;
  cout << "Contents: ";
  while (!aList.empty ()) {
      list<char>::iterator p = aList.begin();
      cout << *p;
      aList.pop_front();
  }
  return 0;
}
```

Another example using a list:

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

int main() {
  list<int> myList;

  myList.push_back(0);                  // Insert a new element at the end
  myList.push_front(0);                 // Insert a new element at the start
  myList.insert(++myList.begin(),2);    // Insert "2" after first element
  myList.push_back(5);
  myList.push_back(6);
```

```
    copy(myList.begin(), myList.end(),
            ostream_iterator<int>(cout, " "));
    cout << endl;   return 0;
}
```

Notes:

Notes:

3.6 string

The C++ strings library (**#include <string>**) provides the definitions of the basic_string class, which is a class template specifically designed to manipulate strings of characters of any character type. It also include two specific instantiations: string and wstring, which respectively use char and wchar_t as character types.

string functions

There are a few global functions that provide some additional functionality for strings to interact either with other string objects or with objects of other types, mainly through the overloading of operators:

operator+	Add (i.e., concatenate) strings.
swap	Swap contents of two strings.
comparison operators	String comparison operators.

Here are some string functions that work with input and output streams:

getline	Get line from istream.
operator<<	Insert string into ostream.
operator>>	Extract string from istream.

Notes:

The **string** class is not part of the STL; however, it often acts much like a form of a container of char data, and can use the STL iterators, algorithms and functions:

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

int main() {
   string myLetters="abcdefghijklmnopqrstuvwxyz";

   int i = 0;
   for(string::iterator myIndex=myLetters.begin();
         myIndex!=myLetters.end();
         ++myIndex)
         cout << ++i << " " << *myIndex << endl;
    return 0;
}
```

Notes:

Examples of **string** creation:

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

int main()   {
     // create a string by using a char string
     const char *ptr = "say goodnight, Gracie";

     // create a string initialized by char* string
     string Str1(ptr);

    string Str2("say goodnight, Gracie");  // or even using the constructor
    string Str3(Str2);                     // by another string

    // by a substring of another string - a 9 char substring starting with the
    //   4th character
     string Str4(Str3, 4, 9);

    string Str5 = Str3.substr(4, 9) ;    // or an explicit call to substr method
    return 0;
}
```

Notes:

To find where a substring starts, erase it, and insert another, use the following code:

```
string::size_type pos = Str2.find("Gracie",0);
if(pos != string::npos ) {   // check that found
   Str2.erase(pos, 6);
   Str2.insert(pos, "Irene");
   cout << Str2 << endl;
}
```

Appending **strings** can be done using the plus (+) operator:

```
Str3 = Str2+Str4;
```

To find the first instance of one of a number of characters, use the following code:

```
string match = "Ie";
pos = Str2.find_first_of(match,0);
cout << "Found character " << Str2[pos]
       << " at position " << pos << endl;
```

Notes:

More example of using **string** class functions:

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

main ()  {
   string a("abcd efg");
   string b("xyz ijk");
   string c;
   cout << a << " " << b << endl;
   cout << "String empty: "  << c.empty()    << endl;

   c = a + b;
   cout << c << endl;
   cout << "String length: "   << c.length()   << endl;
   cout << "String size: "     << c.size()     << endl;
   cout << "String capacity: " << c.capacity() << endl;
   cout << "String empty: "    << c.empty()    << endl;

   string d = c;
   cout << d << endl;
   cout << "First character: " << c[0] << endl;
```

Notes:

```
  string f("    Leading and trailing blanks      ");
  cout << "String f:" << f << endl;
  cout << "String length: " << f.length() << endl;

  cout << "String f:" << f.append("ZZZ") << endl;
  cout << "String length: " << f.length() << endl;
  string g("abc abc abd abc");
  cout << "String g: " << g << endl;

 // Replace 12,1,"xyz",3: abc abc abd xyzbc
  cout << "Replace 12,1,\"xyz\",3: "
       << g.replace(12,1,"xyz",3) << endl;
  cout << g.replace(0,3,"xyz",3) << endl;
  cout << g.replace(4,3,"xyz",3) << endl;
  cout << g.replace(4,3,"ijk",1) << endl;

  cout << "Find: " << g.find("abd",1) << endl;
  cout << (g.find("qrs",1) == string::npos) << endl;

 string h("abc abc abd abc");
 cout << "Find_first_not_of \"abc\",0: "
      << h.find_first_not_of("abc",0) << endl;
 cout << "Find_first_not_of \" \": "
       << h.find_first_not_of(" ") << endl;
 return 0;
}
```

Notes:

3.7 Bitset

The C++ Standard Library contains several classes that work similarly to the STL's predefined container classes.

A **bitset** provides a set of bits as a data structure. They can be manipulated by various binary operators such as logical AND, OR, etc. They are used to model sets of flags, bits or Boolean values. Once a bitset is constructed, the size of the container (bitset) cannot be changed.

Operations of bitset

!=	returns true if the two bitsets are not equal.
==	returns true if the two bitsets are equal.
&=	performs the AND operation on the two bitsets.
^=	performs the XOR operation on the two bitsets.
|=	performs the OR operation on the two bitsets.
~	reverses the bitset (same as calling flip())
<<=	shifts the bitset to the left
>>=	shifts the bitset to the right
[n]	returns a reference to the nth bit in the bitset.

Notes:

Useful functions of bitset

- any — true if any bits are set
- count — returns the number of set bits
- flip — reverses the bitset
- none — true if no bits are set
- reset — sets bits to zero
- set — sets bits
- size — number of bits that the bitset can hold
- test — returns the value of a given bit
- to_string — string representation of the bitset
- to_ulong — returns an integer representation of the bitset

Bitsets can either be constructed with no arguments or with an unsigned long number value that will be converted into binary and inserted into the bitset. When creating bitsets, the number given in the place of the template determines how long the bitset is.

Notes:

Here is an example using a bitset:

```
#include <iostream>
#include <bitset>
using namespace std;

int main()    {
      bitset<16> aFewBits(32);
      cout << "Bits:"  <<  aFewBits  << endl;
      aFewBits[0] = true;      aFewBits[2] = false;
      aFewBits[10] = true;   aFewBits[12] = true;
      cout << "Bits:" <<   aFewBits << endl;
      aFewBits <<= 2;   // rotate bits
      cout << "Bits rotate: " << aFewBits   << endl;
      aFewBits.flip();                                          // flip bits
      cout << "After flipping bits:    " << aFewBits  << endl;
      if(aFewBits.any())
            cout << "aFewBits has at least 1 bit set.\n";
      cout << "aFewBits has " << aFewBits.count()  << " bits set.\n";
      if(aFewBits[0] == 1)
                  cout << "bit 0 is on\n";
       if(aFewBits.test(1))
              cout << "bit 1 is on\n";
       // can add bits to integers
       cout << "Add 11 to bit 0: " << aFewBits[0] + 11 << endl;
       return 0;
}
```

Notes:

3.8 valarray

The **valarray** template class is a vector-like container that is optimized for efficient numeric computation. It doesn't provide iterators.

Although one can instantiate a **valarray** with nonnumeric types, because it mainly has mathematical functions that are intended to operate directly on the numeric data elements, this might not be the most efficient container for non-numeric data. Most of **valarray** functions and operators appear to operate on a **valarray** as a whole – but they actually do their work element-by-element.

Useful Functions

The **valarray** class provides a constructor that takes an array of the target type and the count of elements in the array to initialize the new **valarray**.

The **shift( )** member function shifts each **valarray** element one position to the left (or to the right, if its argument is negative) and fills in holes with the default value for the type (zero in this case). There is also a **cshift( )** member function that does a circular shift (or rotate). This is for bitwise arithmetic operations that often occur in electronics applications.

All mathematical operators and functions are overloaded to operate on **valarrays;** in other words, a valarray's elements can have all of these operations performed on them. Binary operators, such as addition,

Notes:

subtraction, multiplication, division and the modulus operation all require **valarray**s of the same type and size.

The **apply( )** member function of valarrays, like the STL **transform( )** algorithm, applies a function to each element, but the result is collected into a result **valarray**.

Useful Operators

The relational operators (equal, not equal, greater than, less than, etc), return suitably-sized instances of **valarray<bool>** that indicate the result of element-by-element comparisons.

Most operations return a new result array, but a few, such as **min( )**, **max( )**, and **sum( )**, return a single scalar value.

Here is an example:

```
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;

int main()  {
  valarray<int> aValuesArray(10);
  int i;
```

Notes:

```
for(i = 0; i <10; i++)
    aValuesArray[ i ] = i;

cout << "Original contents: ";
for(i = 0; i <10; i++)
  cout << aValuesArray[i] << " ";
cout << endl;

aValuesArray = aValuesArray.cshift(3);
cout << "Shifted contents: ";
for(i = 0; i <10; i++)
  cout << aValuesArray[i] << " ";
cout << endl;

valarray<bool> aValuesArray2 = aValuesArray < 5;
cout << "Those elements less than 5: ";
for(i = 0; i <10; i++)
  cout << aValuesArray2[i] << " ";
cout  << endl;

valarray<double> aValuesArray3(5);
for(i = 0; i <5; i++)
   aValuesArray3[i] = (double) i;

cout << "Original contents: ";
for(i = 0; i <5; i++)
  cout << aValuesArray3[i] << " ";
```

Notes:

```
  cout << endl;

  aValuesArray3 = sqrt(aValuesArray3);
  cout << "Square roots: ";
  for(i = 0; i <5; i++)
    cout << aValuesArray3[i] << " ";
  cout << endl;

  aValuesArray3 = aValuesArray3 + aValuesArray3;
  cout << "Double the square roots: ";
  for(i = 0; i <5; i++)
    cout << aValuesArray3[i] << " ";
  cout << endl;

  aValuesArray3 = aValuesArray3 - 10.0;
  cout << "After subtracting 10 from each element:\n";
  for(i = 0; i <5; i++)
    cout << aValuesArray3[i] << " ";
  cout << endl;
  return 0;
}
```

Notes:

Unit Four
Iterators

Unit topics:

Notes:

4.1 Iterators are smart pointers

What is an iterator?

An iterator in C++ is a concept that implements the iterator design pattern into a specific set of behaviors that work well with the C++ standard library. The standard library uses iterators to work with elements in a range in a consistent manner. Anything that implements this set of behaviors is called an iterator.

The iterator pattern defines a handful of simple requirements. An iterator should allow its consumers to:

- Move to the beginning of the range of elements.
- Advance to the next element.
- Return the value referred to.
- Query it to see if it is at the end of the range.

Iterators are defined by the operations that they must support. And because the underlying representation of an iterator is usually implementation independent, one can use a regular pointer, an integer, or a class object so long as it supports the operations *****, **++**, **=** and **==**.

Notes:

Their capabilities range from the random access iterator that has all of the power of a regular C++ pointer, to the input and output iterators that can only go forward in a collection either reading or writing. In between are the bi-directional, reverse, insertion, stream and forward iterators. Each is a class with its own member functions, overloaded operators and data.

Each predefined standard STL container comes with its own iterator, and that might be a random access iterator, a bidirectional iterator or another type. But one can always use an additional type of iterator if the predefined one doesn't work as needed. Or one can define a custom iterator based on the STL predefined classes.

Iterators are central to generic programming because they are an interface between containers and algorithms. Algorithms usually take iterators as arguments, so for an algorithm to work with a container, a container must provide a way to access its elements using iterators.

The following copy() template function is a commonly used STL algorithm that uses iterators.

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy (InputIterator first,
                     InputIterator last,
                     OutputIterator result) {
  for (; first != last; ++first, ++result)
    *result = *first; // *result is "written to", whereas *first is "read from"
  return result;
```

```
}
```

Notes:

Notes:

4.2 Iterators in the STL

The C++ standard library provides iterators for the standard containers (e.g., list, vector, and deque), and a few other non-container classes, such as string.

The definition of iterators makes them much more than simple C/C++ pointers although they perform that type of behavior. They also have a ‘state’, in other words they can hold data about themselves. Their base classes are found in the header <iterator>.

An iterator gives a program access to the contents of a container or other sequence of data, such as an I/O stream. One can think of an iterator as an abstraction of a pointer; the syntax for using iterators resembles that of pointers.

An iterator points to a single element in a container or sequence and can be advanced to the next element with the ++ (increment) operator. The unary * (dereference) operator returns the element that the iterator points to. Iterators, except for output iterators, can be compared, and two iterators are equal if they point to the same position in the same sequence, or if they both point to one position past the end of the same sequence.

There are also constant iterators defined for each container so that a program can safely traverse a container without modifying its elements. The STL algorithms that don’t modify container elements (search, find etc) require const iterators as arguments.

Notes:

Here is a simple example using a vector with its iterator:

```
vector<int> v;
// …
 for (vector<int>::iterator it = v.begin();
       it != v.end(); // <- end() points to the position AFTER last value
       ++it) {
   cout << *it << endl;
}
```

The example does the following, showing how iterators work:

- Obtain an iterator to the first element in a container by calling that container's begin() member function.
- Advance an iterator to the next element with the pre- or post-increment operator, as in ++it or it++ (it's good practice to use ++it since this is more efficient).
- Get the value it refers to with the pointer dereference operator *, as in *it.
- Finally, determine if an iterator is at the end of a range by comparing it to the iterator returned by the container's end member function, which returns an iterator that refers to one past the end of the elements. This is why the continuation test for the for loop in the example above is it != v.end().

Notes:

Here is another vector example, declared to take strings as elements:

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

int main()  {
    std::vector<std::string> aVector;
    // …
    for (std::vector<std::string>::iterator strItr = aVector.begin();
         strItr != aVector.end();
         ++strItr)
         std::cout << *strItr << ' ';
    return 0;
}
```

This example again shows a vector with its iterator; the double colon [::] indicates that the iterator is a trait of the vector.

To access an element's value using the iterator, we can dereference the iterator, as if it were a pointer:

```
cout << "string value of element: " << *strItr;
```

Similarly, to invoke an operation of the underlying string element through our iterator strItr, we can use the member selection arrow syntax:

Notes:

```
cout << "( " << strItr->size() << " ): " << *strItr << endl;
```

The iterator returned by a container's end() member function represents a logical element that's one past the last element in a container, not the physical memory location that's just beyond the last element.

One should never dereference it, because it is just a marker and holds nothing of value. The point of such a construct is to provide a logical end marker for a range, regardless of the context in which that range is used. Standard algorithms and all member functions that work with a range of iterators use this convention. This is why standard algorithms, such as sort in <algorithm> work like this to sort every element within the given range but not including the iterator used as the second argument to sort().

```
sort(v.begin(), v.end());
```

It is always best to use a const object if the program doesn't need to modify its elements. Thus, if a program is using a const container, the above code won't even compile. In that case, use a const_iterator, which works just like the iterator type in the example above, except that when it is dereferenced, it returns a const object. Here is an example that works with const objects:

```
void printConst(const vector<int>& v) {
  for (vector<int>::const_iterator it = v.begin();
       it != v.end(); ++it) {
    cout << *it << endl;
```

```
    }
}
```

Notes:

Notes:

STL Iterator Advantages

- The STL provides predefined iterators as a convenient abstraction for accessing many different types of containers
- Iterators for templated classes are generated inside the class scope with the syntax *class_name*<*parameters*>::iterator, so they can be directly used with their container.
- Iterators can be thought of as limited (possible stateful) pointers, and can be dereferenced to get the values of elements pointed to, passed to stand-alone functions and algorithms to operate indirectly on the container and more.
- An iterator can be a pointer or a class, and it can be derived from the STL's iterator class template. Any pointer can be treated as an iterator.

STL Iterator Disadvantages

- *Iterators do not provide bounds checking; it is possible to overstep the bounds of a container, resulting in segmentation faults*
- Different containers support different types of iterators by default; thus it is not always possible to change the underlying container type without making changes to the code
- Iterators can be invalidated if the underlying container (the container being iterated over) is changed significantly
- Iterators have the same advantages and power as pointers. But they also have the same risks and inconveniences. For example, using a pointer we can accidentally modify data that we are not supposed to.
- There are also definitions for a class **const_iterator** that provides basically the same functionality as a regular iterator except that modifying the data "pointed to" by the **const_iterator** is not allowed.
- One result of the possible restrictions on an iterator is that most algorithms have two iterators as their arguments, or an iterator and a number of elements count.
- It isn't a good idea to test a output iterator against NULL, because it can’t read the elements it points to.

Notes:

Notes:

- Testing for equality or inequality is safe except for output iterators, which is why the loops using output iterators often use iterator != x.end() as their termination test, to test the container's function itself rather than the iterator's value.

We can use the iterator traits template to know what we need to define when defining our own iterators or what we can check for when using the STL predefined iterators.

Here are the traits.

- **iterator_category** must be one of these 5 values: input_iterator_tag, output_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag, or random_access_iterator_tag.
- **value_type** is the base datatype of the iterator, and it can be void for output iterators since one cannot dereference them
- **difference_type** is the difference between two iterators.
- **pointer** is a pointer to the same datatype as the value_type is
- **reference** is a reference to the same datatype as the value_type

Notes:

Standard STL Iterator templates from **<iterator>** header file:

```
template <typename Category,
          class T,
          class Distance  = ptrdiff_t,
          class Pointer   = T*,
          class Reference = T&>
struct iterator
{
  typedef Category  iterator_category;
  typedef T         value_type;
  typedef Distance  difference_type;
  typedef Pointer   pointer;
  typedef Reference reference;
}
template <typename I>
struct iterator_traits
{
  typedef typename I::iterator_category iterator_category;
  typedef typename I::value_type        value_type;
  typedef typename I::difference_type   difference_type;
  typedef typename T::pointer           pointer;
  typedef typename I::reference         reference;
};

template <typename T> struct iterator_traits<T*>;
template <typename T> struct iterator_traits<const T*>;
```

Notes:

Generic algorithms often need to have access to these traits described on the previous page. Many STL algorithms take a range of iterators, and they might need to declare a temporary variable whose type is the iterators' **value_type**. The class iterator_traits is a mechanism that allows such declarations. In addition to traits, the base class iterators have a few simple functions of their own:

```
template<typename Iter, typename difference_type>
void advance(Iter& i, difference_type d);

template<typename Iter>
difference_type distance(Iter start, Iter finish);
```

Here is another code example of using a vector and an iterator:

```
#include <list>
#include <vector>
using namespace std;
...
list<int> iList;
list<int>::iterator iListIter;
vector<double> dVector;
vector<double>::iterator dVectorIter;
vector<double>::const_iterator dVectorConstIter;
```

Notes:

4.3 Input iterators

This is the simplest type of iterator, because it can only read forward in a sequence or collection of elements. It reads the elements only once and can return the elements when it finds them. One can dereference an Input Iterator to obtain the value it points to, but one can't assign a new value to the element using the iterator's functions. Input iterators can only do these operations:

- construct themselves **i(j);**
- assignment operator **i=j;**
- equality/inequality operator **i==j; i!=j;**
- dereference operator ***j; j->m;** // Can't write
  // but can read
- pre/post increment operator **++j; j++; *j++;**

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;
int main ()
{
  vector<int> v;
  for (istream_iterator<int> i (cin);
        i != istream_iterator<int> ();
        ++i)
     v.push_back (*i);
    // copy (istream_iterator<int> (cin),
    //          istream_iterator<int>(),
```

```
    //          back_inserter(v));
}
```

Input Iterators are used by non-modifying algorithms. For example, the STL **find, find_if,** and **count** algorithms, require no more than input iterators as their first two arguments. One can pass them more powerful iterators or pointers, but only the simplest input iterators are required.

The find and find_if algorithms also return only an input iterator pointing to the element that matches the value to be found, or if the value was not found, an iterator pointing beyond the last element of the container.

Notes:

Notes:

4.4 Output iterators

These are similar to input iterators, except that output iterators perform only write operations. Output iterators also process a container only in a single pass and do no error checking. So the code itself needs to check that a value to be written is valid and whether each write operation was successful. Output iterators can do the following:

- constructor **i(j);**
- assignment operator **i=j;**
- dereference operator ***j=t; *j++=t;** // Can write, // but can't read!
- pre/post increment operator **++j; j++;**

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
   vector<int> v;
   copy (istream_iterator<int> (cin), istream_iterator<int>(),
            back_inserter(v));

   for (vector<int>::iterator i = v.begin ();
         i != v.end ();
        ++i)
       cout << *i << endl;
   // copy (v.begin (), v.end (), ostream_iterator<int> (cout, "\n"));
}
```

Notes:

Many of the STL algorithms return an output iterator as their result. Some of these include:

```
template<typename InputIterator, typename OutputIterator>
  OutputIterator copy(InputIterator start,
                      InputIterator finish,
                      OutputIterator result);

template<typename InputIterator1,
         typename InputIterator2,
         typename OutputIterator,
         typename BinaryOperation>
  OutputIterator transform(InputIterator1 start1,
                           InputIterator1 finish1,
                                InputIterator2 start2,
                                OutputIterator result,
                                BinaryOperation binary_op);

template<typename InputIterator, typename OutputIterator,
          typename Predicate>
  OutputIterator remove_copy_if(InputIterator start,
                                 InputIterator finish,
                                OutputIterator result,
                                Predicate pred);
```

Notes:

4.5 Forward iterators

A forward iterator combines the features of an input iterator and an output iterator. It permits values to be both accessed and modified. Forward iterators support the following operations:

- constructor **i(j);**
- assignment operator **i=j;**
- equality/inequality operator **i==j; i!=j;**
- dereference operator **t = *j; *j=t; // Can assign & read**
- pre/post increment operator **++j; j++;**

One function that uses forward iterators is the replace() generic algorithm, which replaces occurrences of specific values with other values. This algorithm could be written as follows:

```
template <typename ForwardIterator, typename T>
void replace(ForwardIterator first,
             ForwardIterator last,
             const T& old_value,
             const T& new_value)
{
  while (first != last)  {
    if (*first == old_value)  // Can read from and write to *first iterator
       *first = new_value;
    ++first;
  }
}
```

Notes:

Ordinary pointers, like all iterators produced by containers in the C++ Standard Library, can be used as forward iterators.

Forward Iterators can be either:

- ***constant***, in which case one can access the object it points to, but one can't assign a new value to the element using this iterator
- ***mutable***, in which case one can both access the elements pointed to and also change them.

Notes:

4.6 Bidirectional iterators

Bidirectional Iterators work with forward and multi-pass algorithms. They allow traversing collections backward and forward. A bidirectional Iterator can be incremented to obtain the next element in a container, or decremented to obtain the previous element. Bidirectional iterators support the following operations:

- constructor **i(j);**
- assignment operator **i=j;**
- equality/inequality operator **i==j; i!=j;**
- dereference operator **t=*j; *j=t; // Can assign & read**
- pre/post increment/decrement operators **++j; j++; --i; i--;**

All standard STL containers (with the exception of forward_list) provide bidirectional iterators or better.

Input, output and forward Iterators only support forward motion. An iterator used to traverse a singly linked list, for example, need only be a forward iterator, but an iterator used to traverse a doubly linked list has to be a bidirectional iterator.

Notes:

A list's default iterator is bidirectional, which we can use to print the list's elements backwards.

```
#include <list>
#include <iostream>
 using namespace std;
int main()   {
    list<int> aList (1, 1);
    aList.push_back (2);
    list<int>::iterator first = aList.begin();
    list<int>::iterator last = aList.end();
    while (last != first) {
       --last;
       cout << *last << " "; // Could write this as cout << *--last << “”
    }
    return 0;
}
```

Notes:

Here is another example of a list, using its bidirectional iterator with the algorithm copy():

```
#include <list>
#include <iterator>
#include <string>
#include <iostream>
using namespace std;
int main()   {
      list<string> aList;
      aList.push_back("peach");
      aList.push_back("apple");
      aList.push_back("banana");
      // Make y the same size - creates empty strings for y's elements.
      list<string> y(aList.size());

     // This is the STL algorithm std::copy...
     copy(aList.begin(), aList.end(), y.begin());  // source is aList, target is  y

     // Print result forwards and backwards.
     copy (y.begin (), y.end (), ostream_iterator<string> (cout, " "));
     cout << endl;
     copy (y.rbegin (), y.rend (), ostream_iterator<string> (cout, " "));
     cout << endl;
     return 0;
}
```

Notes:

Bidirectional iterators can be used to reverse the values of elements in a container, placing the results into a new container:

```
template <typename BidirectionalIterator, typename OutputIterator>
OutputIterator
reverse_copy(BidirectionalIterator first,
             BidirectionalIterator last,
             OutputIterator result)
{
  while (first != last)
    *result++ = *--last;
  return result;
}
```

Notes:

4.7 Random access iterators

Random Access Iterators allow the operations of pointer arithmetic: addition of arbitrary offsets, subscripting, and subtraction of one iterator from another to find a distance. These are the most powerful iterators and are like regular pointers but they are also smart, e.g., they can hold state. In addition doing all that bidirectional iterators do, random access iterators can do the following as well:

- **operator+ (int)**
- **operator+= (int)**
- **operator- (int)**
- **operator-= (int)**
- **operator- (random access iterator)**
- **operator[] (int)**
- **operator < (random access iterator)**
- **operator > (random access iterator)**
- **operator >= (random access iterator)**
- **operator <= (random access iterator)**

Notes:

The vector class provides a random access iterator that can be used as follows:

```
#include <vector>
#include <iostream>
using namespace std;
int main ()   {
   vector<int> myVector;
   int total_even = 0;

   for( int i=0; i < 10; i++ )
       myVector.push_back(i);

    for (vector<int>::iterator  myVectorIter = myVector.begin();
          myVectorIter != myVector.end();
         myVectorIter += 2)
    {
         total_even += * myVectorIter;
    }

    cout << "Total even =" << total_even << endl;
    return 0;
}
```

Notes:

Here is another example using a vector with its random access iterator:

```
#include <vector>
#include <iostream>
  using namespace std;
int main ()    {
   vector<int> aVector (1, 1);  //created a vector with one element: 1
   aVector.push_back (2);
   aVector.push_back (3);
   aVector.push_back (4);                          // vector v: 1 2 3 4
   // now create 2 iterators, they are random access iterators
   vector<int>::iterator i = aVector.begin();
   vector<int>::iterator j = i + 2; cout << *j << " ";
    i += 3; cout << *i << " ";
    j = i - 1; cout << *j << " ";
    j -= 2;
    cout << *j << " ";
    cout << aVector[1] << endl;
    (j < i) ? cout << "j < i" : cout << "not (j < i)"; cout << endl;
    (j > i) ? cout << "j > i" : cout << "not (j > i)"; cout << endl;
    i = j;
    i <= j && j <= i ? cout << "i and j equal" : cout << "i and j not equal";
    cout << endl;
    return 0;
}
```

Notes:

Random access iterators can jump to any element in the container:

```
#include <vector>
#include <iterator>
#include <iostream>
#include <string>
  using namespace std;
int main ()  {
    const char *s[] = { "a", "b", "c", "d", "e", "f", "g", "h", "i", "j" };
    vector<string> v (s, s + 10);
   // C++11 enables this:
  // vector<string> v ({ "a", "b", "c", "d", "e", "f", "g", "h", "i", "j" });

    copy (v.begin (), v.end (), ostream_iterator<string> (cout, "\n"));

    vector<string>::iterator p = v.begin();  // initialized to start of vector
    p += 5;  // Now p refers to the 5th element
    p[1] = "z";    // Value at p[1] is changed but p refers to 5th

    // position still; position wasn"t changed
    p -= 5; // Back to start

    copy (p, v.end (), ostream_iterator<string> (cout, "\n"));
    return 0;
}
```

Notes:

4.8 STL Iterator Operations Summary

For most iterators:

*it;	Use dereference (*) op to get/set value
++it;	Points to next element. Value after update
it++;	Points to next element. Value before update
it2 = it2;	Assignment
it1 == it2;	Equality comparison
it1 != it2;	Inequality

Additional operators for bidirectional iterators:

--it;	Pre-decrement
it--;	Post-decrement. May be less efficient

Additional operators for random-access iterators:

It += i;	Increments it by i positions
It -= i;	Decrements it by i positions
it2 + I;	Increments it by i positions
it2 – I;	Decrements it by i positions
it[i];	Returns reference to ith element after it
it1 < it2;	Comparison
it1 <= it2;	Comparison
it1 > it2;	Comparison
It1 <= it2;	Comparison

Notes:

Unit Five
Associative Containers

Unit topics:

Notes:

5.1 What is an Ordered Associative Container?

An ordered associative container is a container that supports efficient insertion, removal, and lookup of elements (values) based on keys. It supports insertion and removal of elements, but differs from a sequential container because it doesn't let one insert an element at a specific position or remove an element from a given position based on the position itself – all access is done to the 'value' using a key. And once one adds the elements to the container, the key cannot change, although the element and its key can be removed. The STL provides four ordered associative containers.

For **sets** and **multisets** there are only a group of values stored, but they are stored in some kind of order keyed by the values. So if the values are numbers, the ordering is by numeric value. If they are letters, the collection is ordered by alphabetical order, OR the values can be classes and the programmer can define a function to base the key order upon.

For **maps** and **multimaps** there are actually two series of data that go together hand in hand. One is the values and the other is called the key. These are called pair associative containers because each value is paired with a key. The keys cannot be changed by iterators once they are assigned (you can, of course, erase a key).

Associative containers are implemented as binary trees. This means that each element has a parent element and can have up to two child elements. In addition, all ancestors to the left of an element have lesser values and all ancestors to the right of an element have greater values.

Notes:

Notes:

STL Predefined Ordered Associative Containers

set<key>	Supports unique keys only and provides fast key retrieval
multiset<key>	Supports duplicate keys, also fast key retrieval
map<key, value>	Supports unique keys only, and fast retrieval of type T data based on the key
multimap<key, value>	Supports duplicate keys, otherwise performs like map.

Of course one can also create a custom associative container and use it with the STL iterators, algorithms, and functions.

What are the applications of such containers? Think of a symbolic debugger. It can use an associative container that maps strings (variable names) to memory addresses (the variables' addresses). This way, when one evaluate or modify a variable by its name, the debugger knows where that variable is stored in memory.

Another example is a phone book, in which names serve as keys, and telephone numbers are their associated values.

Notes:

5.2 The pair Container

Pair

This template group is the basis for the map and set associative containers because it stores heterogeneous pairs of data together. Here are the templates:

```
template <typename T1, typename T2>
struct pair {
        T1 first;
        T2 second;
        pair();
        pair (const T1&, const T2&);
        ~pair();
};

// non-member operators and a non-member function
template <typename T1, typename T2>
bool operator== (const pair<T1, T2>&,  const pair T1, T2>&);

template <typename T1, typename T2>
bool operator< (const pair<T1, T2>&, const pair T1, T2>&);

template <typename T1, typename T2>
pair<T1,T2> make_pair (const T1&, const T2&);
```

Notes:

The concept of a **pair** is essential for understanding how associative containers work. A pair binds a key (known as the first element) with an associated value (known as the second element).

In the library <utility> and included in other standard libraries such as <map> there is a class called pair that implements the mathematical idea of a Cartesian point. A pair is used to construct maps and other associative data structures.

Purpose	**Notation**	**Meaning**
class name	pair<X, Y>	set of pairs (x,y) where x is in class X and y is in class Y.
get first	pair.first	the first item in pair
get second item	pair.second	the second item in pair
make a new pair	make_pair(x,y)	constructs instance of pair<X,Y> with first=x and second=y.

The Standard Library defines a pair template that behaves like a special tuple of two elements. This example creates two pairs:

```
#include <utility>                                    //definition of pair
#include <string>

pair <string, string> myTest("Chapter1", "Meet the STL");
pair <string, void *> aValue ("mynum", 0x410928a8);
```

Notes:

Here is a simple program that wraps a pair as a class and then instantiates an object of this simple class to make a point:

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

template<typename T1, typename T2>
class Point : private pair<T1, T2> {
public:
     Point(T1 x, T2 y): pair<T1, T2>(x, y), x(first), y(second) { }
     T1 &x;
     T2 &y;
};

int main() {
   Point<int, int> myPoint(1, 2);
   cout << "x = " << myPoint.x << "\ny = " << myPoint.y << endl;
}
```

Notes:

5.3 The set Container

A **set** is a collection of ordered data in a balanced binary tree structure.

STL template:

```
#include <set>
template <typename Key,
          typename Compare = less<Key>,
          typename Allocator = allocator <Key> >
class set { /* … */ };
```

Characteristics of set

- It orders the elements that are added to it.
- A set contains only one copy of any element (key) added to it.

Useful functions of set

insert()	Inserts elements anywhere based on key
erase()	Remove elements anywhere based on key
count()	Returns the number of elements with a certain key
find()	Returns an iterator to the first element with a certain value
lower_bound()	Returns an iterator to the earliest element that has a key that does not match the value passed
upper_bound()	Returns an iterator to the earliest element that has a key matching the value passed
key_comp()	Returns the stored functor that determines the order of the elements.

Notes:

Example of creating one set and using an iterator with it

```
#include <set>
#include <iostream>
#include <algorithm>
#include <functional>
#include <iterator>
  using namespace std;

int main()
{
  int A[3]= {1, 2, 3};
  set<int, greater<int> > setofNum(A, A+3);

  set<int, greater<int> >::iterator myIndex=  setofNum.find (9);

  if (myIndex == setofNum.end())
    cout << "9 not found\n";
  else
    cout << "9  found\n";
  setofNum.insert(9);

  cout << setofNum.size() << " elements in the set" << endl;
  myIndex =  find(setofNum.begin(), setofNum.end(), 9);
  if (myIndex == setofNum.end())
    cout << "9 not found in the set\n"  << endl;
  else
```

Notes:

```
    cout << "9 found it! \n" << endl;

  copy (setofNum.begin (), setofNum.end (),
          ostream_iterator<int> (cout, "\n"));

  return 0;
}
```

Notes:

Example: Showing how insert() works.

```
#include <iostream>
#include <iterator>
#include <set>
using namespace std;

int main () {
  set<int> myset;
  set<int>::iterator it;
  pair<set<int>::iterator,bool> ret;

  for (int i=1; i<=5; i++) myset.insert(i*10);    // set: 10 20 30 40 50
  ret = myset.insert(20);               // no new element inserted
  if (ret.second==false) it=ret.first;  // "it" now points to element 20

  myset.insert (it,25);                 // max efficiency inserting
  myset.insert (it,24);                 // max efficiency inserting
  myset.insert (it,26);                 // no max efficiency inserting

  int myints[]= {5,10,15};              // 10 already in set, not inserted
  myset.insert (myints,myints+3);

  cout << "myset contains:" << endl;
  copy (myset.begin (), myset.end (), ostream_iterator<int> (cout, "\n"));
  return 0;
}
```

Notes:

Example: Creating a few sets and using swap

```
#include  <set>
#include <list>
#include <iostream>
#include <iterator>
#include <algorithm>
using namespace std;

int main() {
   int a[] = {5, 4, 6, 7, 8, 2, 1, 3};
   set <int> firstSet (a, a + 8); // Defaults to less<int>.
   set <int> secondSet (a, a + 8);

   list <int> aList (secondSet.begin (), secondSet.end ());
   set <int, greater<int> > thirdSet (aList.begin(), aList.end());

   set <int, less<int> > fourthSet;
   fourthSet.swap (firstSet);
   copy (thirdSet.begin (), thirdSet.end (),
         ostream_iterator<int> (cout, " "));
   cout << endl;
   copy (fourthSet.begin (), fourthSet.end (),
         ostream_iterator<int> (cout, " "));

   return 0;
```

Notes:

```
}
```

There is only one way to add a new element to a set or multiset after initializing via the constructor; you insert a value using the **insert()** member function.

Insert operations for a **set** return a pair of values, where the first field contains the iterator, and the second field contains a boolean value that is true if the element was inserted, and false otherwise. In a set, an element will not be inserted if it matches an existing element.

Here is an example of inserting a value in a set

```
#include <set>
#include <iostream>
  using namespace std;
int main()   {
  set<int> firstSet;
   //insert an element with value of 55
   firstSet.insert (55);
   if (firstSet.insert(55).second)
        cout << "element was inserted" << endl;
   else
      cout << "element was not reinserted " << endl;
   return 0;
}
```

Notes:

To remove a value from a set, use erase(). The arguments can be either a specific value, an iterator that denotes a single value, or a pair of iterators that denote a range of values.

```
#include <set>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;
int main()   {
    int a[] = {5, 4, 6, 7, 8, 2, 1, 3};
    set <int> firstSet (a, a + 8);
    firstSet.erase(4);      // erase element whose value is 4
    set<int>::iterator five = firstSet.find(5);
    firstSet.erase(five); // erase element with value 5

    // erase all values in range 6 up to 8, define 2 iterators
    set<int>::iterator six = firstSet.find(6);
    set<int>::iterator eight = firstSet.find(8);
    firstSet.erase (six, eight);

    copy (firstSet.begin (), firstSet.end (), ostream_iterator<int> (cout, " "));
    return 0;
}
```

Iterators for sets

Notes:

The member functions **begin()** and **end()** produce iterators for both sets and multisets. The iterators produced by these functions are **constant** so that keys cannot be changed when traversing the set since this could perturb the ordering of the keys!!

Elements are generated by the iterators in sequence, ordered by the comparison operator provided when the set was declared. If there was no comparison function used when the set or multiset was declared, C++ uses the < operator if it is defined. This works for all primitive data type elements in a set, but if one use a set for holding a user defined class, one need to have a function that compares values passed to the set when it is declared.

There are also reverse iterators: member functions **rbegin()** and **rend()** produce iterators that yield the elements in reverse order.

5.4 The multiset Container

Multisets are just like regular sets except that they allow duplicate keys.

STL template:

```
#include <set>
template <typename Key,
          typename Compare = less<Key>,
          typename Allocator = allocator <T> >
class multiset { /* … */ };
```

Notes:

Characteristics of multiset

- Stores objects of type key in a sorted manner.
- Its value type, as well as its key type, is 'key.'
- Two or more elements may be identical.

Set and multiset are particularly well suited to the set-related algorithms: **set_union, set_intersection, set_difference**, and **set_symmetric_difference** because:

- Set algorithms require their arguments to be sorted ranges and the elements in set and multiset are sorted in ascending order.

- The output range of these algorithms is always sorted and inserting a sorted range into a set or multiset is a fast operation.

- Inserting a new element into a multiset or set does not invalidate iterators that point to existing elements; only iterators pointing to elements being erased are invalid.

Notes:

The following is an example of multiset using several STL algorithms. The copy algorithm copies each element from the specified range (given by its first two arguments), into a place pointed to by the third argument.

```
#include <set>
#include <iostream>
#include <iterator>
#include <algorithm>
  using namespace std;
int main()    {
  const int N = 10;
  int a[N] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};
  int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};

  multiset<int> A(a, a + N);
  multiset<int> B(b, b + N);
  multiset<int> C;
  cout << "Set A: ";
  copy(A.begin(), A.end(), ostream_iterator<int>(cout, " ")); cout << endl;

  cout << "Set B: ";
  copy(B.begin(), B.end(), ostream_iterator<int>(cout, " ")); cout << endl;

  cout << "Union: ";
  set_union(A.begin(), A.end(), B.begin(), B.end(),
                ostream_iterator<int>(cout, " "));
  cout << endl;
```

```
  cout << "Intersection: ";
  set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                   ostream_iterator<int>(cout, " "));
  cout << endl;
  set_difference(A.begin(), A.end(), B.begin(), B.end(),
                 inserter(C, C.end())); // essentially back_inserter(C)
  cout << "Set C (difference of A and B): ";
  copy(C.begin(), C.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  return 0;
}
```

Notes:

Notes:

5.5 The map Container

A **map** is a sorted associative container that associates key objects with value objects. It is called a pair associative container because its **value_type** is actually implemented as **pair<const Key, Value>**. For a **map**, keys must be unique. The keys don't have to be integers, so a **map** is more general than a sequential container such as **vector**, **deque**, or **list**.

STL template:

```
#include <map>
template <typename Key, typename Value,
          typename Compare = less<Key>,
          typename Allocator = allocator <pair<const Key, Value>> >
class map { /* … */ };
```

Characteristics of map

- Inserting a new element into a **map** does not invalidate iterators that point to existing elements.
- Erasing an element from a **map** also does not invalidate any iterators, except those iterators that actually point to the element that is being erased.

To use a **map** you must supply a key type *and* data type. You can also choose to provide a comparing function that does the job of the **less than** operator so that data can be placed in order by key:

```
map <key_type, data_type [, comparison_function]>
```

Notes:

Simple—but powerful—example code using a **map**:

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

struct print {
    void operator () (const map<string, int>::value_type&p) {
      cout << p.second << " " << p.first << endl;
    }
};

int main() {
  map<string, int> myMap;        // map of words and their frequencies
  string aWord;                   // input buffer for words.

  while (cin >> aWord)
    ++myMap[aWord];

  for_each (myMap.begin(), myMap.end(), print ());

  return 0;
}
```

Notes:

Here's another example, using both a map and a pair:

```
#include <utility>
#include <iostream>
#include <string>
#include <map>
using namespace std;
struct print { // Iterator value is a key-value pair.
    void operator () (const map<string, string>::value_type &p)
   { cout << p.first << "=" << p.second << endl; }
};

int main() {
   pair<string, int> pr1, pr2("heaven", 7);
   cout << pr2.first << "=" << pr2.second << endl;    // Prints heaven=7

   //-- Declare and initialize pair pointer.
   unique_ptr < pair<string, int> > prp (new pair<string, int>("yards", 9));
   cout << prp->first << "=" << prp->second << endl;      // Prints yards=9

   //-- Declare map and assign value to keys.
   map<string, string> engGerDict;
   engGerDict["shoe"] = "Schuh"; engGerDict["head"] = "Kopf";

   //-- Iterate over map in sorted order.
   for_each (engGerDict.begin(), engGerDict.end(), print ());
   return 0;
}
```

Notes:

5.6 The multimap Container

A **multimap** is a sorted associative container that associates key objects with value objects. It is called a pair associative container because its value type is actually implemented as **pair<const Key, Value>**.

For a **multimap** keys don't have to be unique. Nor do the keys have to be integers, so maps are multimaps are more general than a sequential container such as **vector**, **deque**, or **list**.

STL template:

```
#include <map>
template <typename Key, typename Value,
          typename Compare = less<Key>,
          typename Allocator = allocator <pair<const Key, Value> >
class multimap
```

Here is an example using a multimap:

```
#include <string>
#include <map>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <functional>
```

Notes:

```
using namespace std;

typedef multimap <string, string> names_type;

struct print {
  print(ostream& out) : os (out) {}

  void operator() (const names_type::value_type &p) {
    os << p.first << " belongs to the " << p.second << " family\n";
  }

  ostream& os;
};

// Print out a multimap
ostream& operator<<(ostream& out, const names_type &l) {
  for_each (l.begin (), l.end (), print (out));
  return out;
}

ostream &operator << (ostream &out,
                      const pair<names_type::iterator,
                      names_type::iterator> &p) {
  for_each (p.first, p.second, print (out));
  return out;
}
```

Notes:

```
int main(int argc, char* argv[]) {
  names_type names;  // create a multimap of names
  typedef names_type::value_type value_type;

  // Put the names in the multimap
  names.insert(value_type(string("Sue"), string("Smith")));
  names.insert(value_type(string("Jane"), string("Smith")));
  names.insert(value_type(string("Kay"), string("Smith")));
  names.insert(value_type(string("Kurt"), string("Jones")));
  names.insert(value_type(string("Sue"), string("Jones")));
  names.insert(value_type(string("John"), string("Jones")));
  names.insert(value_type(string("Sophie"), string("Mackay")));
  names.insert(value_type(string("Steve"), string("Mackay")));
  names.insert(value_type(string("Sue"), string("Mackay")));

  // print out the names
  cout << "All the names:" << endl << names << endl;

  // Find the people named Sue
  pair<names_type::iterator,names_type::iterator> p =
    names.equal_range ("Sue");

  // print them out
  cout << endl << names.count("Sue") << " People called Sue:"
         << endl << p << endl;
```

```
  return 0;
}
```

Notes:

Notes:

5.7 Unordered Associative Containers

Unordered associative containers refer to a group of class templates in recent versions of STL that implement variations of the hash table data structure. Being templates, they can be used to store arbitrary elements, such as integers or custom classes.

STL Predefined Unordered Associative Containers

unordered_set<key>	Supports unique keys only and provides fast key retrieval
unordered_multiset<key>	Supports duplicate keys, otherwise performs like unordered_set.
unordered_map<key, T>	Supports unique keys only, and fast retrieval of type T data based on the key
unordered_multimap<key, T>	Supports duplicate keys, otherwise performs like unordered_map.

The unordered associative containers are similar to the associative containers in C++ standard library but have different constraints. As their name implies, the elements in the unordered associative containers are not ordered. This is due to the use of hashing to store objects. The containers can still be iterated through like a regular associative containers.

The main difference between the ordered associative containers and the unordered associative containers is that the former keeps the keys sorted according to some total order. For example, in a map<string, int>, the

Notes:

elements are sorted according to the lexicographical order of the strings. An unordered associative container, on the other hand, divides the keys into a number of subsets, and the association of each key to its subset is done by a hash function.

Consequently, searching a key is confined to its subset rather than the entire key space. Searching an unordered associative container can therefore be faster than searching a sorted associative container under some circumstances; but unlike ordered associative containers, the performance is less predictable.

5.8 The unordered_map Container

An **unordered_map** is a unordered associative container that associates key objects with value objects. It is called a pair associative container because its **value_type** is actually implemented as **pair<const Key, Value>**. For an **unordered_map**, keys must be unique. The keys don't have to be integers, so an **unordered_map** is more general than a sequential container such as **vector**, **deque**, or **list**.

STL template:

```
#include <unordered_map>
template <typename Key, typename Value,
          class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          typename Allocator = allocator <pair<const Key, Value>> >
class unordered_map { /* … */ };
```

Notes:

Characteristics of unordered_map

- Inserting a new element into an **unordered_map** does not invalidate iterators that point to existing elements.
- Erasing an element from an **unordered_map** also does not invalidate any iterators, except those iterators that actually point to the element that is being erased.

To use an **unordered_map** you must supply a key type and data type. You can also choose to provide a unary function object that acts as a hash function for a key, which takes a single object of type key and returns a value of type std::size_t. You can also optionally provide a binary function object that implements an equivalent relation on values of type key, which takes two arguments of type key and returns a value of type bool:

```
map <key_type, data_type [, hash_function_object],
        [predicate_function_object]>
```

Notes:

Simple—but powerful—example code using an **unordered_map** and C++11 features.

```
// -std=c++0x
#include <iostream>
#include <initializer_list>
#include <unordered_map>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int main ()
{
  std::vector<std::string> word_list ({"now", "is", "the", "time",
        "for", "all", "good", "men", "to", "come", "to", "the",
        "aid", "of", "their", "party"});
  typedef std::unordered_map<std::string, int> WORD_MAP;
  WORD_MAP word_map; // Map of words and their frequencies.

  for (auto const &word_iter = word_list.begin ();
       word_iter != word_list.end ();
       ++word_iter)
    word_map[*word_iter]++;

  for (auto const &i : word_map)
```

Notes:

```
   std::cout << i.second << " " << i.first << std::endl;

 std::vector<std::pair <std::string, int>> v;

 for (auto const &i : word_map)
    v.push_back (i);

 std::sort (v.begin (), v.end ());

 std::cout << std::endl;

// Print out the sorted vector.  Note auto type deduction.
 for (auto const &i : v)
   std::cout << i.second << " " << i.first << std::endl;

 auto total = 0;

 for_each (word_map.begin(),
             word_map.end(),
            [&total] (const WORD_MAP::value_type &p)
            {
               total += p.second;
            });

 std::cout << "total number of words = " << total << std::endl;
 return 0;
}
```

Notes:

Unit Six
Adapted Iterators

Unit topics:

Notes:

6.1 STL Predefined Adaptors

- Iterator adaptors include **reverse** and **insert** iterators.
- Container adaptors include **stack**, **queue**, and **priority_queue**.
- Function adaptors include **negators** and **binders**.

6.2 What are Iterator Adaptors?

Iterator Adaptors are types of iterators that operate on more than just STL containers; they can also 'adapt' the standard containers' iterators to work differently if that is desired.

Iterator adaptors turn the standard iterators into things that can operate in reverse, in insertion mode, and with streams.

So we take the input, output, forward, bidirectional iterators and make them into the following adapted iterators:

- Insert iterators
- Reverse iterators
- Stream iterators
- Raw storage iterators ← discussed later.

Use adaptors to make a custom class act like an STL collection

Notes:

One can cause any existing class with sequence-like characteristics act like an STL collection simply by writing an iterator adaptor class. Wrap anything: a tokenizer, a parser, a database query, a sequence of frames in an animation, a stream of MIDI data, whatever is needed. The code for the STL-provided stream adaptors is concise and self-explanatory, and is a good place to start to develop one's own.

Notes:

6.3 The Inserter Iterator Adaptors

Insert iterators, also called **inserters**, change the assignment of a new value into an insertion of that value into a sequence of values, thus not overwriting other values. Note that these are output iterators since they write values, and they override the container's assignment operator.

The insert iterators allow insertion at the front, back or middle of the elements depending upon the container type.

There are several types of inserters and they also can change how algorithms work:

Type	**Function used**	**Container**
front_insert_iterator	push_front(value)	deque, list
back_insert_iterator	push_back(value)	vector, deque, list, string
insert_iterator	insert(value, position)	vectors, deques, lists, maps, and sets

Notes:

Why use inserters?

The STL algorithms that copy elements, such as **copy(), unique_copy(), copy_backwards(), remove_copy(),** & **replace_copy()** are passed an iterator that marks the position within a container to begin copying.

With each element copied, the value is assigned and the iterator is incremented. Each copy requires that we guarantee that the target container is of a sufficient size to hold the set of assigned elements. So we may need to expand the containers as we perform the algorithm unless we want to make the containers huge from the start…thus the inserters are useful. Start with an empty container, and use the inserter along with the algorithms to make the container grow only as needed.

back inserter

For example, a back_inserter() causes the container's push_back() operator to be invoked in place of the assignment operator. This is the preferred inserter for vectors. The argument passed to back_inserter is the container itself.

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

int main()    {
```

Notes:

```
        vector <int>aVect;
        back_insert_iterator< vector <int> >  n (aVect);
        int i;
        while( cin>> i )
                *n++ =i;
        // copy (istream_iterator<int>(cin), istream_iterator<int>(),
        //          back_inserter (aVect));
        copy (aVect.begin (), aVect.end (), ostream_iterator<int> (cout, "\n"));
        return 0;
}
```

Here is another example using a back inserter:

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

int main()
{
  vector<int> aVect1, aVect2;

 for(int i = 0; i <5; i++)
    aVect1.push_back(i);

  cout << "Original contents of aVect1:\n";
  copy (aVect1.begin (), aVect1.end (), ostream_iterator<int> (cout, "\n"));
```

Notes:

```
  back_insert_iterator<vector<int> > bck_i_itr(aVect1);
 *bck_i_itr++ = 100;        // insert rather than overwrite at end
 *bck_i_itr++ = 200;

 cout << "aVect1 after insertion:\n";
 copy (aVect1.begin (), aVect1.end (), ostream_iterator<int> (cout, "\n"));

 cout << "Size of aVect2 before copy: " << aVect2.size() << endl;

 copy(aVect1.begin(), aVect1.end(), back_inserter(aVect2));
 cout << "Size of aVect2 after copy: " << aVect2.size() << endl;

 cout << "Contents of aVect12 after insertion:\n";
 copy (aVect2.begin (), aVect2.end (), ostream_iterator<int> (cout, "\n"));
 return 0;
}
```

front inserter

A front_inserter() causes the container's push_front() operator to be invoked. This inserter can be used only with the list and deque containers, because one cannot add to the front of a vector:

```
#include <iostream>
#include <iterator>
#include <list>
```

Notes:

```
using namespace std;

int main()  {
  list<int> aList1, aList2;
  list<int>::iterator itr;
  int i;

  for(i = 0; i <5; i++)
    aList1.push_back(i);

  cout << "Original contents of aList:\n";
  copy (aList1.begin (), aList1.end (), ostream_iterator<int> (cout, "\n"));

  front_insert_iterator<list<int> > frnt_i_itr(aList1);
  // create a front_insert_iterator to aList

  *frnt_i_itr++ = 100;   // insert rather than overwrite at front
  *frnt_i_itr++ = 200;

  cout << "aList after insertion:\n";
  copy (aList1.begin (), aList1.end (), ostream_iterator<int> (cout, "\n"));

  cout << "Size of aList2 before copy: " << aList2.size() << endl;

  copy(aList1.begin(), aList1.end(), front_inserter(aList2));

  cout << "Size of aList2 after copy: " << aList2.size() << endl;
```

Notes:

```
  cout << "Contents of aList2 after insertion: ";
  copy (aList2.begin (), aList2.end (), ostream_iterator<int> (cout, "\n"));

  return 0;
}
```

inserter

An inserter() causes the container's insert() operation to be invoked. inserter() takes two arguments: the container and an iterator into the container indicating the position at which insertion should begin. Here is an example using an inserter with a vector:

One can construct an **insert_iterator** directly from a container and an iterator i. The values written to the insert iterator are inserted before i. inserters can be used in place of output iterators. Create one like this to insert elements at the back of a deque:

```
deque<int> aDeck;
insert_iterator<deque<int> > i (aDeck, aDeck.end() );
```

One can also declare back and front inserters like this using a deque:

```
deque<int> aDeck;
back_insert_iterator<deque<int> > backItr (aDeck);
front_insert_iterator<deque<int> > forwardItr (aDeck);
```

Notes:

Here is an example using an insert iterator for vector (note that this inefficient due to the copying overhead of inserting into a vector anywhere but at the end):

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

int main()    {
  vector<int> aVect;
  vector<int>::iterator itr;
  int i;

  for(i = 0; i <5; i++)
    aVect.push_back( i );

  cout << "Original contents of aVect:\n";
  copy (aVect.begin (), aVect.end (), ostream_iterator<int> (cout, "\n"));

  itr = aVect.begin();
  itr += 2; // point to element 2

  insert_iterator<vector<int> > i_itr(aVect, itr);
  *i_itr = 100;    // insert rather than overwrite
  *i_itr = 200;
```

```
  cout << "aVect after insertion:\n";
  copy (aVect.begin (), aVect.end (), ostream_iterator<int> (cout, "\n"));
  return 0;
}
```

Here are the inserter function templates:

```
template <typename Container>
back_insert_iterator<Container> back_inserter(Container& x) {
   return back_insert_iterator<Container>(x);
}

template <typename Container>
front_insert_iterator<Container> front_inserter(Container& x) {
   return front_insert_iterator<Container>(x);
}

template <typename Container, typename Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i) {
   return insert_iterator<Container>(x, Container::iterator(i));
}
```

Note how functions adapters (e.g., back_inserter()) are used to leverage C++'s implicit type inference feature for functions.

Notes:

Notes:

6.4 The Reverse Iterator Adaptor

Reverse iterators walk backwards through collections. There are two types: **reverse_iterator**, which only goes backward through the data, and **reverse_bidirectional_iterator,** which can go backward and forward.

Useful Operators

These iterators can be created using a default constructor or by a single argument constructor which initializes the new **reverse_iterator** with a **random_access_iterator**. Here are some of their operators:

* operator	returns a reference to the current item i.e. value pointed to.
++ operator	advances the iterator to the previous item (--current) and returns a reference that item.
++ operator (int)	advances the iterator to the integer previous item (--current) and returns a copy of the previous item.
-- operator	advances the iterator to the next item (++current) and returns a reference to that item.
-- operator (int)	(int) advances the iterator to the integer next item (++current) and returns a copy to the previous item.
== operator	This returns true only if the iterators x and y point to the same item.

Notes:

Here is an example:

```
#include <iostream>
#include <deque>
#include <iterator>
#include <algorithm>
     using namespace std;
int main() {
  deque<int> aDeck;

  for(int i = 0; i <10; i++)  aDeck.push_back(i);

  cout << "Contents printed backward:\n";

  copy (aDeck.rbegin (), aDeck.rend (), ostream_iterator<int> (cout, "\n"));

  cout << "Contents printed backwards with reverse iterator adapter:\n";

  reverse_iterator<deque<int>::iterator> rfirst(aDeck.end());
  reverse_iterator<deque<int>::iterator> rlast(aDeck.begin());

  copy (rfirst, rlast, ostream_iterator<int> (cout, "\n"));

  cout << "Contents printed backwards with reverse_copy:\n";
  reverse_copy (aDeck.begin (), aDeck.end (),
                ostream_iterator<int> (cout, "\n"));
  return 0;
}
```

Notes:

6.5 The Stream Iterator Adaptor

Instead of extracting and inserting data explicitly from the passed streams, we can use the **stream adaptors** to make the streams appear as STL containers, and the **copy** algorithm to manage the process of building the sorted collection. There are three stream adaptors: **istream**, **ostream** and **stream buffers**.

In addition, because STL algorithms are template functions specialized by the types of the iterators returned by the underlying containers it is perfectly reasonable to use the **copy** algorithm with a stream adaptor to move the contents of one container to another of a different type.

Here is a function using both istream and ostream adaptors:

```
#include <iostream>
#include <set>
#include <algorithm>
#include <string>
#include <iterator>

int main (void)    {
   list<string> myList;
   copy(istream_iterator<string>(cin), istream_iterator<string>(),
         inserter(myList, myList.begin())); // Essentially front_inserter()
   copy(myList.begin(), myList.end(), ostream_iterator<string>(cout, "\n"));
   return 0;
}
```

Notes:

Istream iterator

The class template **istream_iterator** reads elements from an input stream using operator >>(). A value of type T is retrieved and stored when the iterator is constructed and each time operator++() is called.

The iterator is equal to the end-of-stream iterator value if the end-of-file is reached. *The constructor with no arguments can be used to create an end-of-stream iterator.* The only valid use of this iterator is to compare to other iterators when checking for end of file. Do not attempt to dereference the end-of-stream iterator; it plays the same role as the past-the-end iterator of the end() function of containers.

Since an **istream_iterator** is an input iterator, one cannot assign to the value returned by dereferencing the iterator. This also means that **istream_iterators** can only be used for single pass algorithms.

```
#include <algorithm>          // for copy algorithm
#include <iostream>           // for cin, cout, endl
#include <iterator>           // for stream_iterators and inserter
#include <vector>             // for vector
#include <numeric>            // for accumulate algorithm
    using namespace std;

int main ()
{
  typedef vector<int> Vector;
```

Notes:

```
  typedef istream_iterator<Vector::value_type> is_iter;
  typedef ostream_iterator<Vector::value_type> os_iter;

  Vector v;
  Vector::value_type sum = 0;

  // default constructor to get ending iterator; get values from cin until EOF
  cout << "Enter a sequence of integers (eof to quit): ";
  copy (is_iter (cin), is_iter (), back_inserter (v));

  // Stream the whole vector and the sum to cout.
  copy (v.begin (), v.end () - 1, os_iter (std::cout, " + "));

  if (v.size () != 0)
    cout << v.back () << " = "  << accumulate (v.begin (), v.end (), sum)
         << endl;
  return 0;
}
```

Notes:

Ostream iterator

Another type of stream iterator, **ostream iterator**, writes values to a standard output stream, therefore allowing STL algorithms to write to output streams. Ostream iterators change the assignment of a new value to be an output operation using the **<<** operator.

When created, an ostream iterator must be provided with the output stream and optionally a character can be defined to separate the values being passed into the output stream

```
#include <iostream>
#include <iterator>
    using namespace std;
int main()   {
   ostream_iterator<int> oi(cout, " ");
   *oi++ = 6;
   *oi++ = 88;
   return 0;
}
```

Here is another example:

```
#include <iterator>
 #include <numeric>
 #include <deque>
 #include <iostream>
 using namespace std;
```

Notes:

```
int main () {
  int arr[4] = { 3,4,7,8 };
  int total=0;
  deque<int> d(arr+0, arr+4);
  // stream the whole deque and a sum to cout
  copy(d.begin(),d.end()-1,
          ostream_iterator<int>(cout," + "));
  cout << *(d.end()-1) << " = " <<
      accumulate(d.begin(),d.end(),total) << endl;
  return 0;
}
```

Stream buffer iterator (work with files)

A third type of stream iterator adaptor is **the stream buffer iterator**, and there are two types, **istreambuf_iterator** for reading and **ostreambuf_iterator** for writing streams. These template class objects can read or write individual characters from or to basic_streambuf objects.

The class template **istreambuf_iterator** reads successive characters from the stream buffer for which it was constructed. operator*() gives access to the current input character, if any, and operator++() advances to the next input character. If the end of stream is reached, the iterator becomes equal to the end of stream iterator value, which is constructed by the default

constructor, istreambuf_iterator(). An **istreambuf_iterator** object can be used only for one-pass-algorithms.

Notes:

```
#include <iostream>     // for cout, endl
#include <fstream>      // for ofstream, istreambuf_iterator
#include <stdio.h>      // for tmpnam () and remove ()
   using namespace std;
int main ( )  {
   const char *fname = tmpnam (0);    // temp filename
   if (!fname)
      return 1;
   ofstream out (fname, ios::out | ios::in | ios::trunc);

   // output the example sentence into the file
   out << "Here is a sample sentence for output.\n"
         "I hope that you like this sentence out there.";
   // go to the beginning of the file
   out.seekp (0);

   // construct an istreambuf_iterator pointing to the ofstream object
   // underlying streambuffer
   istreambuf_iterator<char> iter (out.rdbuf ());

   // construct an end of stream iterator
   const istreambuf_iterator<char> end;
   cout << endl;
   // output the content of the file
```

```
    while (iter != end)
            cout << *iter++;

    // Alternative more concise approach is:
    // copy (istreambuf_iterator<char> (out.rdbuf ()),
    //             istreambuf_iterator<char> (),
    //             ostream_iterator<char> (cout));

    cout << endl;
    remove (fname);     // remove the temp file
    return 0;
}
```

Notes:

Notes:

Unit Seven
Adapted Containers

Unit topics:

Notes:

7.1 What are Container Adaptors?

Container adaptors are classes that are based on other classes to implement a new functionality, often a more limited one. For example, **stack** restricts **vector** and **queue** restricts **deque**.

Member functions can be added or hidden or can be combined to achieve new functionality.

Adaptor containers change ordinary containers such as **vector**, **deque**, and **list** into **stack** and **queue**, by 'adapting' them to reflect a user's expectations (i.e., limiting what their underlying container can do).

An adaptor allows the standard algorithms to be used on a subset or to modify the data without having to copy the data elements into a new container.

Notes:

Notes:

7.2 The stack Container Adaptor

A **stack** is an ideal choice when one need to use a LIFO (Last In, First Out) data structure. For example, think about people entering the back seat of a car that has only one door: the last person to enter is the first to exit. It is implemented with a **deque** by default, but one can change that.

Here is a simple example using the STL stack class:

```
#include <iostream>
#include <stack>
   using namespace std;

int main()   {
  stack<char> aStack;
  aStack.push('A');
  aStack.push('B');
  aStack.push('C');
  aStack.push('D');

  while(!aStack.empty()) {
    cout << "Popping: ";
    cout << aStack.top() << endl;
    aStack.pop();
  }
  return 0;
}
```

Notes:

Here is its template:

```
template <typename T, typename Container = deque<T> >
class stack
{
public:
  explicit stack(const Container& c = Container());
  bool empty() const;
  size_type size() const;
  value_type& top();
  const value_type& top() const;
  void push(const value_type& t);
  void pop();

private :
   Container container_ ;
   //…
};
```

Notes:

7.3 The queue Container Adaptor

A **queue** or FIFO (First In, First Out), is characterized by having elements inserted into one end and removed from the other end, for example: a queue of people at a theater's box office. Again by default it is implemented from a **deque**, but that can be changed.

Here is an example program:

```
#include <iostream>
#include <queue>
#include <string>
   using namespace std;
int main()  {
  queue<string> aCue;
  cout << "Pushing one two three \n";
  aCue.push("one");
  aCue.push("two");
  aCue.push("three");

  while(!aCue.empty()) {
    cout << "Popping ";
    cout << aCue.front() << endl;
    aCue.pop();
  }
  return 0;
}
```

Notes:

Here is the queue template definition:

```
template <typename T, typename Container = deque<T> >
class queue
{
public:
  explicit queue(const Container& c = Container());
  bool empty() const;
  size_type size() const;
  value_type& front();
  const value_type& front() const;
  value_type& back();
  const value_type& back() const;
  void push(const value_type& t);
  void pop();

private:
  Container container_;
  // …
};
```

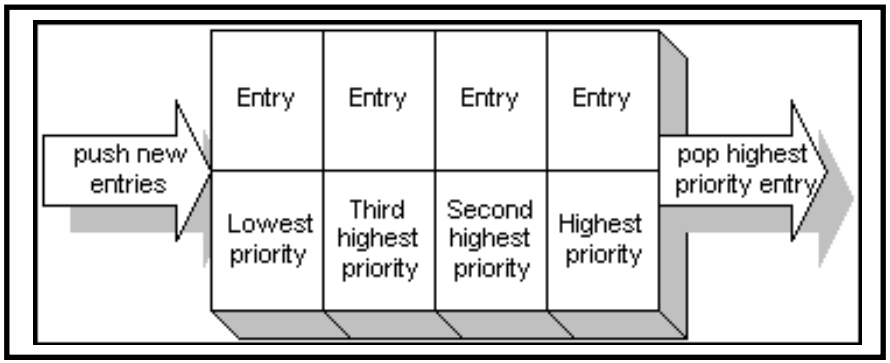
Notes:

7.4 The priorty_queue Container Adaptor

A **priority queue** assigns a priority to every element that it stores. New elements are added to the queue using the **push()** function, just as with a FIFO queue.

This queue also has a **pop()** function, which differs from the FIFO **pop()** in one key area. When you call **pop()** for the priority queue, you don't get the oldest element in the queue. Instead, one get the element with the highest priority.

The **priority queue** fits in well with certain types of tasks. For example, the scheduler in an operating system might use a priority queue to track processes running in the operating system:



Its template is found in the <queue> header file. It uses the **less** functor to compare its keys. Internally it is implemented from a **vector** that has been "heapified".

Notes:

```
template <typename T, typename Container = vector<T>,
          typename Compare = less<Container::value_type> >
class priority_queue
{
public:
  // Constructors
  explicit priority_queue(const Compare& comp = Compare(),
                          const Container& c = Container());
  template <typename InputIterator>
  priority_queue(InputIterator start, InputIterator finish,
                          const Compare& comp = Compare(),
                          const Container& c = Container());
  bool empty() const;
  size_type size() const;
  const value_type& top() const;  // const version only
  void push(const value_type& t);
  void pop();
};
```

On the next page is an example using a priority_queue that prints out city names and their distance from a given place. The output will be:

```
El Cajon     20
Poway        10
La Jolla      3
```

Notes:

```
#include <queue>                              // priority_queue
#include <string>
#include <iostream>
using namespace std;

struct Place {
    unsigned int dist;  string dest;
    friend ostream& operator<<(ostream &, const Place &) ;
    Place (const string dt, unsigned int ds) : dist(ds), dest(dt) {}
    // This method is needed to order the priority queue properly.
    bool operator< (const Place & right) const
     {   return dist < right.dist;  }
};
ostream & operator << (ostream& os, const Place & p)
 {   return os << p.dest << " " << p.dist; }

int main()
{
    priority_queue < Place > pque;
    pque.push(Place("Poway",   10));
    pque.push(Place("El Cajon",  20));
    pque.push(Place("La Jolla", 3));
    while (! pque.empty()) {              // remove top entry from queue
                cout << pque.top() << endl;
               pque.pop();
    }
  return 0;
}
```

Notes:

Unit Eight
Functors

Unit topics:

Notes:

8.1 What is a Function Pointer?

Functors work very much like function pointers, so it would be good to remember how C++/C function pointers work. Function pointers are pointers, i.e. variables, which point to the address of a function. A running program gets only a certain amount of space in the main memory. Both the executable compiled program code and the used variables reside in this memory. A function in the program code becomes nothing more than an address in memory.

When calling a function, say f(), at a certain point called label in a program, just put the call to the function f() at the point label in the source code. Then compile the program and every time the program execution reaches that point in the code, the function is called.

But what is efficient, if it is not known at compile time which function should be called? Perhaps different objects will be created in the program and this determines the proper function call?

Determining the function to call at runtime might be done using callback functions or selecting a functions out of a pool of possible functions using a switch statement.

Another way to achieve this functionality is to use the powerful concept of a function pointer.

Notes:

Here is an example first using a switch statement and then doing the same thing with a function pointer. This is a function designed to perform the four basic arithmetic operations as function calls.

```
float Plus (float a, float b) { return a+b; }
float Minus (float a, float b) { return a-b; }
float Multiply(float a, float b) { return a*b; }
float Divide (float a, float b) { return a/b; }

//  The char variable opCode  specifies which operation to execute
void Switch(float a, float b, char opCode)  {
    float result;
    switch(opCode)   {
    case '+' :
        result = Plus (a, b);
        break;
    case '-' :
        result = Minus (a, b);
        break;
    case '*' :
        result = Multiply (a, b);
        break;
   case '/' :
        result = Divide (a, b);
        break;
}
```

Notes:

Here is the same problem but using a function pointer named pt2Func, which points to a function which takes two floats and returns a float. The function pointer "specifies" which arithmetic operation is executed.

```
void SwitchFunctPtr(float a, float b, float (*pt2Func)(float, float))  {
      float result = pt2Func(a, b); // call using function pointer
      cout << "Switch replaced by function pointer: 2-5=";
      cout << result << endl;
}
```

And here is another function that calls both of these:

```
void testThem() {
     Switch(2.0, 5.0, /* '+' specifies function 'Plus' to be executed */ '+');
     SwitchFunctPtr (2.0, 5.0, /* pointer to function 'Minus' */ &Minus);
}
```

Important note: A function pointer always points to a function with a specific signature. Thus all functions to be used with that function pointer must have the same signature (parameters and return type) as the function pointer does.

In addition, there are actually two different types of function pointers. One type are pointers to ordinary C functions or to static C++ member functions. The second type are function pointers to non-static C++ member functions;

Notes:

these contain the hidden argument, the ‘this’ pointer. These two types are incompatible with each other; one cannot create a function pointer of one type and use it to point to the other type of function.

Declaring Function Pointers

Since a function pointer is a variable, it must be declared. This example defines some function pointers named pt2Function, pt2Member and pt2ConstMember. They all point to functions, which take one float and two char and return an int. In the C++ examples it is assumed that the functions pointed to are non-static member functions.

```
int (*pt2Function)(float, char, char) = 0;                  // C
int (aClass::*pt2Member)(float, char, char) = 0;            // C++
int (aClass::*pt2ConstMember)(float, char, char) const = 0; // C++
```

Function Call Conventions

Normally a programmers doesn’t have to think about a function’s calling convention: The compiler assumes cdecl as default if a program doesn’t specify another convention The calling convention tells the compiler things like how to pass the arguments or how to generate the name of a function. Examples of other calling conventions are stdcall, pascal, fastcall. The calling convention belongs to a function’s signature.

Notes:

Therefore functions and function pointers with different calling conventions are incompatible with each other. For Borland and Microsoft compilers one can specify a specific calling convention between the return type and the function's or function pointer's name. For the GNU GCC, use the attribute keyword. Write the function definition followed by the keyword attribute and then state the calling convention in double parentheses.

Examples specifying function all specifications:

```
void __cdecl f(float a, char b, char c);                  // Borland and Microsoft
void f(float a, char b, char c) __attribute__((cdecl));   // GNU GCC
```

Use the Address-of Operator when calling a function pointer

Note: Although one may omit the address-of operator on most compilers, always use the correct way to write portable code. Here are two standalone functions in C:

```
int aFun(float a, char b, char c) {
    return a+b+c;
}
int anotherFun(float a, char b, char c) const {
    return a-b+c;
}
pt2Function = aFun;             // short form
pt2Function = &anotherFun;  // correct assignment using address operator
```

Notes:

Here they are as member functions in a C++ class:

```
class aClass  {
public:
      int aFun(float a, char b, char c) {
            return a+b+c;
       }
       int anotherFun(float a, char b, char c) const {
           return a-b+c;
       }
      static int aStaticFun (float a, char b, char c) {
        return a+ b + c;
     }
};
```

Now here is some code that calls both member functions:

```
// correct assignment using address operator
pt2ConstMember = &aClass::anotherFun;

// note: <pt2Member> may also legally point to &anotherFun
pt2Member = &aClass::aFun;

pt2Function = &aClass::aStaticFun;
```

Notes:

Using the equality and inequality operators

The == and != can be used as with other pointers. This example verifies whether the pt2Function and pt2Member actually contain the address of the functions aFun and aClass::anotherFun.

Here is an example written in C

```
if (pt2Function != 0)  {                    // check if initialized
    if(pt2Function == &aFun)
          printf("Pointer points to aFun\n"); }
   else
          printf("Pointer not initialized!!\n");
```

Here is an example using C++

```
if (pt2ConstMember == &aClass::anotherFun)
          cout << "Pointer points to aClass::anotherFun" << endl;
```

In C call a function can be called using a function pointer by explicitly dereferencing it using the * operator. Another way to call it is to use the function pointer's name instead of the function's name.

In C++ the two operators .* and ->* are used together with an instance of a class to call one of its non-static member functions. If the call takes place

Notes:

within another member function, the program can use the ‘this’ pointer. Here are some examples.

In C the calls can be done either way:

```
int result1 = pt2Function (12, ’a’, ’b’);
int result2 = (*pt2Function) (12, ’a’, ’b’);
```

In C++, calls are done in the way:

```
aClass instance1;
int result3 = (instance1.*pt2Member)(12, ’a’, ’b’);
int result4 = (*this.*pt2Member)(12, ’a’, ’b’);       // ‘this’ pointer can be used

// instance2 is a pointer
aClass* instance2 = new aClass;
int result4 = (instance2->*pt2Member)(12, ’a’, ’b’);
delete instance2;
```

Notes:

Passing a Function Pointer as an Argument

A function pointer can be provided as a function's calling argument; this is how it works when using a callback function. The following code shows how to pass a pointer to a function, pt2Func, which returns an int and takes a float and two chars:

```
void PassPtr(int (*pt2Func)(float, char, char))  {
      int result = (*pt2Func)(12, 'a', 'b');     // call using function pointer
      cout << result << endl;
}

void TestPassPtr()   {
       PassPtr(&aFun);
}
```

Notes:

Returning a Function Pointer

A function pointer can be a function's return value. The following code shows two ways to return a pointer to a function that takes two floats as arguments and returns a float. To return a pointer to a member function, just change the definitions/declarations of the function pointers.

Here is one solution where a function takes a char and returns a pointer to a function that takes two floats and returns a float. The variable opCode specifies which function to return, and the functions pointed to are the Plus and Minus defined in the examples above.

```
float (*GetPtr1(const char opCode))(float, float) {
       if(opCode == '+')
              return &Plus;
       else
              return &Minus;
}
```

Here is another way to do it using a typedef. First define a pointer to a function which takes two floats and returns a float:

```
typedef float(*pt2Func)(float, float);
```

Notes:

Now use a function that takes a char and returns a function pointer which is defined with the typedef above. Again, opCode specifies which function to return:

```
pt2Func GetPtr2(const char opCode)
{
       if(opCode == '+')
              return &Plus;
       else
              return &Minus;
}

void TestFunctionPtr()
{
        // define a function pointer and initialize it to NULL
        float (*pt2Function)(float, float) = NULL;

        // get the function pointer from function 'GetPtr1'
        pt2Function=GetPtr1('+');
        cout << (*pt2Function)(2, 4) << endl; // call function using the pointer

        // get function pointer from function 'GetPtr2'
        pt2Function=GetPtr2('-');
        cout << (*pt2Function)(2, 4) << endl; // call function using the pointer
}
```

Notes:

8.2 What is a Functor?

A **functor**, or **function object**, is simply any object that can be called as if it is a function. The C++ Standard Template Library defines a few simple functors to do arithmetic and make relational and logical comparisons.

Create a functor

To create an object that behaves just like a function, one only need to provide a way to call this object by name using parentheses and (optionally) pass arguments. Just create a class that overloads the function call operator, which is simply a pair of parentheses: '**()**'.

Let's create the simplest functor we can:

```
class simple_function_object
{
public:
      int operator()(int i) { return i; }
 };
```

Now we can call this function just like any other function, only being a class it can hold data as well as several functions within itself.

Notes:

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

int main()   {
   // instantiate the functor
   simple_function_object aFunctor;

   // calls operator '()' of class 'function_object'
   cout << aFunctor(5) << endl;

   // also calls operator '()' of class 'function_object'
   cout << aFunctor.operator()(2) << endl;
   return 0;
}
```

Functors are the STL's improvement over traditional C function pointers. One could use regular function pointers with the correct argument signature, but the STL’s predefined functors offer some advantages, such as the ability to optimize the functors via inline method calls and/or the ability to maintain state within the functor.

Notes:

Functor classes

The following are functor base classes plus their derived classes, found in <functional> for the functors that take one or two arguments. Custom functor classes can inherit from these classes or be defined on their own as in the example above. Here are a few templates of the binary functors.

```
template <typename _Arg1, typename _Arg2, typename _Result>
struct binary_function {
  typedef _Arg1 first_argument_type;
  typedef _Arg2 second_argument_type;
  typedef _Result result_type;
};

template <typename T>
struct plus : public binary_function<T,T,T> {
  T operator()(const T& x, const T& y) const { return x + y; }
};

template <typename T>
struct minus : public binary_function<T,T,T> {
  T operator()(const T& x, const T& y) const { return x - y; }
};

template <typename T>
struct multiplies : public binary_function<T,T,T> {
```

Notes:

```
  T operator()(const T&  x, const T&  y) const { return  x *  y; }
};

template <typename T>
struct divides : public binary_function<T,T,T> {
  T operator()(const T&  x, const T&  y) const { return  x /  y; }
};
```

Here is the unary functor base class and its derived class:

```
template <typename _Arg, typename _Result>
struct unary_function {
  typedef _Arg argument_type;
  typedef _Result result_type;
};

template <typename T>
struct negate : public unary_function< T, T >
{
  T operator()(const T & x) const { return -x; }
};
```

Notes:

Here is an example of a user-defined functor that extends the unary_function template:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
   using namespace std;

class isEven : public unary_function<int, bool> {
public:
  bool operator()(int i)
  {
    return (result_type) !(i%2);
  }
};

int main()  {
  vector<int> aVect;
  int i;
  for(i = 1; i <= 20; i++)
     aVect.push_back(i);

  cout << "Sequence:";
  for(i = 0; i <aVect.size(); i++)
```

Notes:

```
    cout << aVect[ i ] << " ";
  cout << endl;

  i = count_if(aVect.begin(), aVect.end(), isEven());
  cout << i << " numbers are evenly divisible by 2.";
  return 0;
}
```

Notes:

STL Functor Classes

From those base classes, also found in <functional> are the following functor classes. These are used in the STL for working as parameters to both containers and algorithms.

```
// Arithmetic functors
template <typename T> struct plus;          // binary +
template <typename T> struct minus;         // binary -
template <typename T> struct multiplies;    // binary *
template <typename T> struct divides;       // binary /
template <typename T> struct modulus;       // binary %
template <typename T> struct negate;        // unary –

// Comparisons
template <typename T> struct equal_to;      // ==
template <typename T> struct not_equal_to;  // !=
template <typename T> struct less;          // <
template <typename T> struct greater;       // >
template <typename T> struct less_equal;    // <=
template <typename T> struct greater_equal; // >=

// Logical operations
template <typename T> struct logical_and;   // binary &&
template <typename T> struct logical_or;    // binary ||
template <typename T> struct logical_not;   // unary !
```

Notes:

8.3 Classifying Functors

Three of the most common reasons to use **functors** instead of using ordinary functions are:

1. To use an existing functor provided by the standard library instead of creating a new function.
2. To improve execution by using inline function calls.
3. To allow a functor to access or set state information that is held by an object.

How Functors differ from functions and algorithms

- Each **functor** does a single, specific operation
  - Often implemented as small classes or structs
  - Often has only one public member function: **operator()**

- **Functors** can have member variables
  - Arguments not stored may be supplied at point of call
  - Member variables can parameterize the operations such as the value k for a +k functor or arguments for an invocation on a remote object

Types of Functors

- The **comparison** and **predicate functors** return a boolean value indicating the result of a comparison such as:
  - one object greater than another, or

Notes:

- Telling an algorithm whether to perform a conditional action. For example to remove all objects with a particular attribute.

- The **numeric functors** perform operations like addition, subtraction, multiplication or division. These usually apply to numeric types, but some, like **+**, can be used with strings.

Classifications

STL **functors** are classified based on their capabilities in various ways. For example, they can be categorized by whether the functor's **operator( )** takes zero, one or two arguments. See the new few pages for other classifications.

#1. Functors classified based on number of arguments:

- **Generator** — Takes no arguments and returns a value of the desired type. (A **RandomNumberGenerator** is a special case of a generator because it takes no arguments but returns a specific thing: a number.) But generators can be defined to return any type of data; they just don't take any arguments as input.

- **UnaryFunction** — Takes a single argument of any type and returns a value which may be of a different type.

Notes:

- **BinaryFunction** Takes two arguments of any two types and returns a value of any type.

#2. Functors returning a boolean result classified

A special case of the unary and binary functions is the **predicate**, which simply means a function that returns a **boolean** result, which is used to make a **true**/**false** decision.

- **Predicate** This can also be called a **UnaryPredicate**. It takes a single argument of any type and returns a **boolean** result.

- **BinaryPredicate** This type of predicate takes two arguments of the same data type, and returns a **boolean** result. It can do anything inside the function.

Notes:

#3. Functors classified based on the operators supported:

There are sometimes qualifications on the functor types passed into algorithms. These qualifications are given as the template argument type identifier name for the algorithm. They limit the type of functor that can be used as the argument to that algorithm. Here are some of them.

- **LessThanComparable** A functor class that has a less-than operator, **<**.
- **Assignable** A functor class that has an assignment operator **=** for its own type. (All STL functors are assignable.)
- **EqualityComparable** A functor class that has an equivalence operator **==** .

For functors that are LessThanComparable and EqualityComparable, the STL provides templates in the <utility> header file so they can use the !=, the >=, and <= operators.

Notes:

#4. Functors classified based whether they can carry a ‘state’:

Having a state means having member variables that can take on values. For example a class named Rectangle can have a fillColor member variable. This could distinguish between different Rectangle object instances.

- **Stateless -** Stateless **functors** are the closest correspondent to a regular function. The **functor** doesn't have data members, or, if it does, they have no impact whatsoever on the function call operator.

- **State (constant) -** Invariable functors do have a state, but the function call operator is declared constant. This means that the operation will use the state, but won't change it.

- **State (variable) -** Variable functions objects not only have a state, but also can change this state with each operation.

Notes:

Functor efficiency considerations

- Passing parameters to Functors
  - Can be done by value or by reference
  - Same kinds of aliasing issues as with any other object

- Watch performance with many small functors
  - Watch out especially for creation and destruction overhead
  - May want to inline functors constructors and destructors
  - Put functors on stack instead of heap

- Functors are a powerful, general mechanism
  - Reduces programming complexity and increases reuse
  - Several new uses of generic programming
  - Could go farther with parameterization than just algorithms

Functors are often referred to as smart functions because they can :
- contain more than their main function,
- hold data members (have a state) and
- be passed to other functions.

Notes:

8.4 Arithmetic Functors

Arithmetic **functors** are (mostly) binary operations that return the sum, difference, product, or division of the first argument and the second. Binary means they take two arguments and unary means they take one argument.

Functor	Type	Result
negate<type>()	Unary	Negates supplied parameter (-param)
plus<type>()	Binary	Adds supplied parameters (param1 + param2)
minus<type>()	Binary	Subtracts supplied parameters (param1 – param2)
multiplies<type>()	Binary	Multiplies supplied parameters (param1 * param2)
divides<type>()	Binary	Divides supplied parameters (param1 / param2)
modulus<type>()	Binary	Remainder of parameters (param1 % param2)

Notes:

Example using the **arithmetic functors** with the transform algorithm:

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>
#include <iterator>
  using namespace std;
int main()  {
  vector<float> aVect;
  aVect.push_back(1); aVect.push_back(2); aVect.push_back(3);
  aVect.push_back(4); aVect.push_back(5);

  // Multiple all float in the vector by themselves (i.e., double themselves)
  transform(aVect.begin(),  aVect.end(),  aVect.begin(),
            aVect.begin (), multiplies<float>());

  copy (aVect.begin (), aVect.end (), ostream_iterator<float> (cout, "\n"));

  // Divide all floats in the vector by 3
  transform(aVect.begin(),aVect.end(), aVect.begin (),
            bind2nd(divides<float>(), 3));
  copy (aVect.begin (), aVect.end (), ostream_iterator<float> (cout, "\n"));

  return 0;
}
```

Notes:

8.5 Relational Functors

Sometimes is it convenient to compare two values and the STL relational functors have the standard six math functions =, != , >, >=, <, and <=.

Predicate	Type	Result
equal_to<type>()	Binary	Equality of parameters (param1 == param2)
not_equal_to<type>()	Binary	Inequality of parameters (param1 != param2)
less<type>()	Binary	Parameter 1 less than parameter 2 (param1 < param2)
greater<type>()	Binary	Parameter 1 greater than parameter 2 (param1 > param2)
less_equal<type>()	Binary	Parameter 1 less than or equal to parameter 2 (param1 <= param2)
greater_equal<type>()	Binary	Parameter 1 greater than or equal to parameter 2 (param1 >= param2)

Notes:

Here is an example to remove spaces in a string that uses the **equal_to** and **bind2nd** functors. It says perform **remove_if** when the equal to function finds a blank char in the string.

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
using namespace std;

int main() {
  string s="spaces in text";
  cout << s << endl;
  string::iterator new_end =  remove_if(s.begin(), s.end(),
                                        bind2nd(equal_to<char>(), ' '));
  cout << s << endl; // Note the “xt” aren’t removed at the end!
  s.erase(new_end, s.end());
  cout << s << endl;
  return 0;
}
```

Notes:

8.6 Logical Functors

The three STL **functors** for logical comparisons match the three types of logical comparisons allowed in C++:

Predicate	**Type**	**Result**
logical_and	Binary	logical conjunction x && y
logical_or	Binary	logical disjunction x || y
logical_not	Unary	logical negation ! x

The STL templates also allow one to create custom binary or unary functors that would work similarly to their in the algorithms:

```
template <typename Arg, typename Result>
struct unary_function {
   typedef Arg argument_type;
   typedef Result result_type;
};

template <typename Arg1, typename Arg2, typename Result>
struct binary_function {
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};
```

Notes:

Unit Nine
Function Adaptors

Unit topics:

Notes:

9.1 Function Adaptors

In general, adaptors can transform one predefined interface to another. The STL has predefined **functor** adaptors that will change their **functors** so that they can:

- Perform function composition and binding
- Allow fewer created functors
- Allow building functions as graphs (especially chains and trees) of other functions

These **functors** allow one to combine, transform or manipulate **functors** with each other, certain values or with special functions. They are divided into the following categories:

Type of adaptor	**STL adaptor**	**What it does**
binders	bind1st, bind2nd	adapt functors by 'binding' one of their arguments
negators	not1, not2	adapt functor by negating arguments
member functions	ptr_fun mem_fun mem_fun_ref	allow functors to be class member function adaptors

Notes:

9.2 Binder Adaptors

A **binder** can be used to transform a binary functor into an unary one by acting as a converter between the functor and an algorithm.

Binders always store both the binary functor as well as the argument internally. This argument then will being passed as one of the arguments of the functor every time it is being called.

Binder	**Result**
bind1st(Op, Arg)	Calls 'Op' with 'Arg' as its first parameter
bind2nd(Op, Arg)	Calls 'Op' with 'Arg' as its second parameter

How does a binder work? Suppose one create a class to add up two integer numbers. Call it **addEm** (which is similar to the standard STL **plus** functor):

```
class addEm {
public:
   int operator()(const int& i, const int &j) { return i + j; }
};
```

Notes:

Now it is a general class. But what if sometimes a programmer wants to use this class but fix the value being added so that it always adds 5 to whatever is being passed to addEm? One could write another class or one could use the STL **bind1st functor**:

```
bind1st(addEm(), 5)
```

The **addEm()** expression above is a **binary functor** that computes the sum of two integers, and **bind1st** invokes this functor by binding the first argument to 5. One can now use this bound addEm object in STL algorithms

Here is an example using the STL object and algorithms:

```
vector<double>  myVector;
…
int factor = 33;
transform(myVector.begin(), myVector.end(), myVector.begin(),
   bind2nd(multiplies<double>(), factor) );
```

The multiplies arithmetic functor is takes two arguments: the first must be a reference to a value and the second argument, factor, is a value to multiply the argument by.

For each element in the range the algorithm passes the currently processed value to the **functor**. However, we still need the additional argument 'factor'. The bind2nd adaptor simply causes the algorithm to

Notes:

always pass the factor as second argument. The results are stored in the **vector**.

Here is another example of using bind2nd to bind the second argument to a functor greater to the value 5.67 and then pass the result of this to the count_if algorithm as its third argument. This counts and returns how many values in a vector<float> have values greater than 5.67.

```
count_if(v.begin(),v.end(), bind2nd(greater<float>(), 5.67));
```

But how does this work? First consider **greater**, which is defined in the <functional> header file. It's a **binary functor** - something that can be used like a function and takes 2 arguments.

Then, bind2nd is an function adaptor, a kind of wrapper that in this case lets us use a binary function when a unary function is required. So in this example **count_if** passes an element to what it thinks is a unary function, which in turn passes this element plus a comparison value of 5.67 to a binary function whose return value is given back to count_if.

Notes:

9.3 Negator Adaptors

A **negator** can be used to store the opposite result of a functor.

Negater	**Result**
not1(Op)	Negates the result of unary 'Op'
not2(Op)	Negates result of binary 'Op'

Let's see how it works by creating our own template for simple boolean **functor** that will check if a value is an odd number and it will return true or false. This template takes only one argument, so it is **unary**:

```
template<typename T>
struct is_odd : unary_function<T, bool> {
   bool operator() (T number) const { return (number % 2 != 0); }
};
```

Let's use these templates with a STL algorithm named remove_copy_if and a STL functor, not1. This takes 4 arguments: start copying from here, stop here, and the true/false function to use to determine whether to copy the argument into the result output iterator.

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <functional>
```

Notes:

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

template<typename T>
struct is_odd : unary_function<T, bool> {
    bool operator() (T number) const { return (number % 2 != 0); }
};

int main() {
  vector<int>  vector1;
  vector1.push_back (1);
  vector1.push_back (2);
  vector1.push_back (3);
  vector1.push_back (4);

  vector<int> vector2;
  remove_copy_if (vector1.begin(), vector1.end(),
                        back_inserter (vector2), is_odd<int>());
  copy (vector2.begin(), vector2.end (), ostream_iterator<int> (cout, "\n"));

  vector<int> vector3;
  remove_copy_if (vector1.begin(), vector1.end(),
                        back_inserter (vector3), not1 (is_odd<int>()));
  copy (vector3.begin(), vector3.end (), ostream_iterator<int> (cout, "\n"));

 return 0;
```

Notes:

```
}
```

Here is a simple example showing a user defined functor being used with the not1 negator and the find_if algorithm:

```
#include <functional>
#include <vector>
#include <iterator>
#include <iostream>
#include <algorithm>
   using namespace std;

struct IntGreaterThanThree
    : public unary_function<int, bool>
  {
      bool operator() (int x) const { return x > 3; }
  };

int main() {
      vector<int> v;
      vector<int>::iterator itr;
      v.push_back(4); v.push_back(1); v.push_back(2);
      v.push_back(8); v.push_back(5);v.push_back(7);

      itr = find_if (v.begin(), v.end(), not1(IntGreaterThanThree()));
      // itr = find_if (v.begin(), v.end(), not1(bind2nd(greater<int> (), 3)));
```

Notes:

```
    copy (itr, v.end (), ostream_iterator<int> (cout, "\n"));

    return 0;
}
```

Notes:

9.4 Member Function Adaptors

A **member function adapter** can be used to allow class member functions as arguments to the STL predefined algorithms. There are 2 of them:

mem_fun(PtrToMember mf);
Converts a pointer to member to a functor whose first arg is a pointer to the object. Unary function if mf takes no arguments, binary function if mf takes an argument.

mem_fun_ref(PtrToMember mf);
Converts a pointer to member to a functor whose first arg is a reference to the object. Unary function if mf takes no arguments, binary function if mf takes one argument.

Example showing member function reference function adaptor, mem_fun_ref, plus examples using several STL algorithm functions:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
   using namespace std;

class WrapInt {
  int val;
```

Notes:

```
public:
  WrapInt(): val (0) { }
  WrapInt(int x): val (x) {}

  bool showval() {
     cout << val << " ";
     return true;
  }

  bool isPrime() {
    for(int i = 2; i <= (val/2); i++)
      if(!(val%i))
         return false;
    return true;
  }

  bool isEven() { return (bool) !(val % 2); }

  bool isOdd() { return (bool) (val %2); }
};

int main() {
  vector<WrapInt> aVect(10);
  vector<WrapInt>::iterator end_p;
  int i;
```

Notes:

```
for(i = 0; i <10; i++)
  aVect[ i ] = WrapInt(i+1);

cout << "Sequence contains: ";
for_each(aVect.begin(), aVect.end(), mem_fun_ref(&WrapInt::showval));
cout << endl;

// remove the primes
end_p = remove_if(aVect.begin(), aVect.end(),
                        mem_fun_ref(&WrapInt::isPrime));

cout << "Sequence after removing primes: ";
for_each(aVect.begin(),  end_p, mem_fun_ref(&WrapInt::showval));
cout << endl;
for(i = 0; i <10; i++)
  aVect[ i ] = WrapInt(i + 1);

end_p = remove_if(aVect.begin(), aVect.end(),
                        mem_fun_ref(&WrapInt::isEven));

cout << "Sequence after removing even values: ";
for_each(aVect.begin(), end_p, mem_fun_ref(&WrapInt::showval));
cout << endl;

for(i = 0; i < 10; i++)
  aVect[ i ] = WrapInt(i + 1);
```

Notes:

```
  end_p = remove_if(aVect.begin(), aVect.end(),
                        mem_fun_ref(&WrapInt::isOdd));

  cout << "Sequence after removing odd values: ";
  for_each(aVect.begin(), end_p, mem_fun_ref(&WrapInt::showval));

  return 0;
}
```

Notes:

9.5 Pointers to Functions

The **ptr_fun( )** adapters take a pointer to a function and turn it into a **functor**. They are not designed for a function that takes no arguments; they are for both unary functions and binary functions.

Here is an example of using a pointer to a class member function. Define ptrFunc as a pointer to a non class member function taking a single integer argument and returning a bool:

```
bool (*ptrFunc)( int );
```

The value of ptrFunc becomes an actual function address to the CPU:

```
bool myFunction ( int i ) {
   return i > 10 ;
}

int main ()
{
   bool (*ptrFunc)(int) ;
   ptrFunc = myFunction ;
   if ( (*ptrFunc)( 11 ))
      std::cout << "ptrFunc ( 11 ) true\n" ;
   return 0 ;
}
```

Notes:

Now what if this becomes a member function:

```
    bool (myClass::* ptrFunc)(int) ;
```

And then ptrFunc is now a pointer to a member function of class aClass:

```
#include <iostream>
  using namespace std;
class aClass
{
private :
   int x ;
public :
   aClass ( int y ) : x( y )
   {
      bool firstFunc ( int i )
      return i > x ;
   }
} ;
int main ()   {
  bool (aClass::*ptrFunc)(int) ;   //declare a member function pointer
  ptrFunc = &aClass::firstFunc ;
  aClass array ( 10 ) ;
  if ((array.*ptrFunc)( 11 ))
    cout <<  "the pointer member function is true" <<  endl;
}
```

Notes:

Example using the predefined STL object ptr_fun:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <functional>
#include <cstring>
   using namespace std;

int main()
{
  vector<char *> aVect;
  vector<char *>::iterator charIt;
  int i;

  aVect.push_back("One");
  aVect.push_back("Two");
  aVect.push_back("Three");
  aVect.push_back("Four");
  aVect.push_back("Five");

  cout << "Sequence contains:";
  copy (aVect.begin (), aVect.end (), ostream_iterator<char *> (cout, " "));
  cout << endl;
  cout << "Searching for Three.\n";
```

Notes:

```
  // use a pointer-to-function adaptor
  charIt = find_if(aVect.begin(), aVect.end(),
                        not1(bind2nd(ptr_fun(strcmp), "Three")));

  if(charIt != aVect.end()) {
    cout << "Found it.";
    cout << "Here is the rest of the story:";
    copy (charIt, aVect.end (), ostream_iterator<char *> (cout, " "));
    cout << endl;
  }
  return 0;
}
```

Notes:

9.6 Creating custom Functors

Whether for use with custom algorithms and containers or with STL's, sometimes one want to create custom **functors**. Here are some tips for making them work:

- **Functors need to provide an appropriate function call operator.** This is what makes them usable just like regular functions.
- **Functors must implement correct copy semantics.** Passing by value always introduces additional copies of the functors. Thus, **functors** with a state need to ensure that their state information gets mirrored correctly to additional copies of them. This requires a specialized copy constructor as well as a specialized assignment operator, so that the values pointed to get copied rather than just copying their references; this is what is known as making a "deep copy."
- **Functors should be small to avoid expensive copies.** Passing by value always comes at a performance penalty because instead of simply copying or assigning a small pointer, the complete object needs to be copied or assigned. This performance penalty will increase the bigger the object itself is.

Notes:

- **Functors should not contain any polymorphic elements (in other words, no virtual functions).** Using polymorphic functors opens the door to some problems. If a derived class **functor** is being passed by value into parameters of the corresponding base class type, a problem occurs: While copying the object, the additional parts of the derived class are removed.

  However, there is still a way to use polymorphic functors, but only by putting the polymorphic parts into a separate class. The remaining non-polymorphic functor then contains a pointer to this separate class. This is a well-known design pattern, usually referred to as the *Bridge* or “pimpl” pattern (which stands for “pointer implementation”).

When one create a new class one can define what "+" does for objects of that class. Other operators can be defined too. Objects with "()" defined are called function objects or functors. Here's a simple example:

```
class is_more_than_two {
public:
   bool operator() (int val) {return val>2;}
};
```

which we can use with count_if in this statement:

```
count_if(v.begin(),v.end(), is_more_than_two());
```

Unit Ten
Non-Mutating Algorithms

Unit topics:

Notes

10.1 Algorithms

There are several header files containing the STL algorithms:

#include <algorithm>
#include <numeric>
#include <functional> - for the functors

The Standard Template Library provides a number of simple, useful and general algorithms to perform the most common operations on groups and sequences of elements. These include traversals, searching, sorting and the insertion and removal of elements in containers.

These algorithms are implemented in a way that works well with the containers and iterators, but one can also use them alone with custom code with standard arrays and pointers.

No containers are passed to the algorithms, so programmers don't even need a container. Arguments to the algorithms are only data ranges, i.e., iterator values, plus functors (if needed for 'if' tests). Algorithms can work with any type of calling function that can pass the correct data to them.

Notes

Algorithm categories

There are various ways to categorize the STL algorithms. One way is to use four categories: non-mutating (also known as search, scan, compare, and count), mutating (also known as copy, move, swap, change, delete, generate and fill), sorting and sets, and numeric algorithms.

- Non-mutating — Operate using a range of iterators, but don't change the data elements found
- mutating — Operate using a range of iterators, but can change the order of the data elements
- sorting and sets — Sort or searches ranges of elements and act on sorted ranges by testing values
- numeric — Type of algorithms that produce numeric results

In addition to these main types, there are specific algorithms within each type that accept a test condition, known as a predicate. These are named ending with the _if suffix to remind us that they require an 'if' test's result (true or false), as an argument; these can be the result of functor calls.

Notes

Advantages of STL algorithms

- STL algorithms are decoupled from the particular containers they operate on and are instead parameterized by iterators.
- All containers with the same iterator type can use the same algorithms.
- Because algorithms are written to work on iterators rather than containers, the software development effort is drastically reduced. For example, instead of writing a search routine for each kind of container, one only write one for each iterator type and apply it any container.
- Since different containers can be accessed by the same iterators, just a few versions of the search routine must be implemented.

Notes

10.2 Non-mutating Algorithms

Algorithms that don't change the contents of what is found are called non-mutating. These do lookup and counting operations.

The non-mutating algorithms

- adjacent_find — Finds two items adjacent to each other
- count — Returns a count of elements matching a given value
- count_if — Returns a count of elements for which a predicate is true
- equal — Determines if two sets of elements are the same
- find — Finds a specified value in a given range
- find_end — Finds last sequence of elements in a range
- find_first_of — Finds first sequence of elements in a range
- find_if — Finds the first element for which a predicate is true
- for_each — Applies a function to a range of elements
- mismatch — Finds the first position where two ranges differ
- search — Searches for a range of elements
- search_n — Searches for n consecutive copies of an element within a range

Note that unexpected behavior can result if two sequences or strings of unequal size are compared. STL will not warn one or prevent this, so you need to have error checking in custom code.

Notes

How to use non-mutating algorithm functions

ForwardIter loc = **adjacent_find**(first, last[, bin_pred]);
advance(i, n);
difference_type n = **count**(first, last, value);
difference_type n = **count_if**(first, last, value, pred);
distance_type n = **distance**(first, last);
bool is_equal = **equal**(first1, last1, first2[, bin_pred]);
Function result = **for_each**(first, last, func);
InputIter loc = **find**(first, last, target);
InputIter loc = **find_if**(first, last, pred);
ForwardIter loc = **find_end**(first, last, subseq_first, subseq_last[, bin_pred]);
ForwardIter loc = **find_first_of**(first, last, targets_first, targets_last[, bin_pred]);
pair<InputIter1, InputIter2> differ = **mismatch**(first1, last1, first2[, bin_pred]);
ForwardIter loc = **search**(first, last, subseq_first, subseq_last[, bin_pred]);
ForwardIter loc = **search_n**(first, last, count, target[, bin_pred]);

10.3 Searching

- find: looks for a value in a range.
- find_if: looks for items in a range that satisfy a predicate.
- find_first_of: looks for items in first range that is also in the second range or uses a binary_predicate to find first matching item.
- find_end: looks backward for items in first range that are not also in the second range or uses a binary_predicate to find first non-matching item.
- adjacent_find: looks for first pair in range that are equal, or match under a binary_predicate.

For example, find a value without using any predefined algorithm:

```
list<int> myList;
int findIt = 10;

for (list<int>::iterator i = myList.begin(); i != myList.end(); ++i)
{
   if (*i == findIt)
      break;
   else
         ……
}
```

Suppose we want do a better job and make the find condition an iterator:

Notes

```
list<int>::iterator find (list<int>::iterator start,
                          list<int>::iterator end, int findIt)
{
  list<int>::iterator i;
  for (i = start; i != end; ++i)
  {
    if (*i == findIt)        break;
  }
  return i;      // return end if the value was not found
}
```

Now let's look at the STL templates for the find algorithms:

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

template <typename InputIterator, typename Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred)  {
   while (first != last && !pred(*first)) ++first;
   return first;
```

```
}

template <typename InputIterator, typename T, typename Size>
void count(InputIterator first, InputIterator last, const T& value,
          Size& n)
{
   while (first != last)
      if (*first++ == value) ++n;
}
```

These templates can be used with pointers, iterators or even with const iterators as the first two arguments (or any datatype that supports the unary * and **++** operators). The third parameter can be any value that supports comparison with the type ***i**.

But there's more; What if we want to find the first occurrence of a positive value in a sequence? Or the first prime number in the sequence? Or, working with a sequence of strings, what if we want to find the first string that contains no spaces?

The common factor in these situations is an algorithm similar to find, only we don't search for a particular value, but for an element that matches a predicate (a true/false test condition). This algorithm is part of the STL, and it is called **find_if**. It takes 3 values; the first two can be pointers (iterators or numbers) and a new third argument which must be a condition or boolean function (predicate).

Notes

Example using a function predicate with the find_if algorithm:

```
template <typename Iterator, typename Function>
void find_if (Iterator first, Iterator last, Function predicate)
{
   for (Iterator i = first; i != last; ++i)
   {
      if (predicate(*i))
         return i;
   }
   return i;
}
```

Note that the predicate parameter can be anything that supports the expression **predicate(x)** and returns a boolean value (or something convertible to a boolean value).

Here is the template for the find_linear algorithm, which uses a forward iterator:

```
template<typename ForwardIterator, typename T>
ForwardIterator find_linear (ForwardIterator first,
                             ForwardIterator last, T& value) {
   while (first != last) if (*first++ == value) return first;
   return last;
}
```

Notes

Here is some code using the find_if algorithm with a **functor** rather than simply using a pointer or iterator.

```
class is_negative
{
public:
   bool operator() (int value) const
        return value < 0;
};

…some code…

list<int> values;

//very compact test!
if (find_if (values.begin(), values.end(), is_negative()) != values.end())
// Could also use bind2nd (less<int>(), 0)
{
…. a negative number is found
}
```

In the example above, the compiler will instantiate a version of the **find_if** template with the first two parameters of type **list<int>::iterator** and the last parameter of type **is_negative**.

Notes

Notice the pair of brackets after the name **is_negative**. We are passing an **object** of class is_negative, which is instantiated on-the-fly to be passed to the function.

That same object "lives" only throughout the entire execution of the loop inside find_if. The expression **predicate(*i)** inside find_if simply calls the **member function operator()**, passing an int as parameter, which is each element of the list since that's what we get when we dereference the iterator.

What if we want to search the first value that it is less than 5 or less than a number specified by the user?

The above example can be extended to provide that extra flexibility since we can add data members to the functor and hold some data that we pass it when instantiating the object.

Notes

This example demonstrates extending the **functor** to add data members to hold values sent to the object, but it isn't a template function:

```
class is_greater_than
{
public:
   is_greater_than (int n)  : value(n)  {}
   bool operator() (int element) const
      return element > value;
private:
   const int value;
};

list<int> values;
// ... fill the list
if (find_if (values.begin(), values.end(),
           is_greater_than(5)) != values.end())
// Could also use bind2nd(greater<int>(), 5)
{
    //do something since the set contains an integer > 5
}
```

Finally, declare and then use a template function. Now this **is_greater_than** functor could take any type of numeric data or even a class if the **>** operator has been defined for it:

```
template <typename T>
class is_greater_than
{
public:
   is_greater_than (const T & n)   : value(n)   {}
   bool operator() (const T & element) const
    // Could say: return greater<T>(element, value);
     return element > value;
private:
   const T value;
};

list<int> values;
// ... fill the list
if (find_if (values.begin(), values.end(),
            is_greater_than<int> (5)) != values.end())
{
// … it contains a number greater than 5 so do something
 }
```

Notes

10.4 Counting

- count: scan range and count occurrence of a value.
- count_if: scan range and count times a predicate is true.

count
count is for counting occurrences of a value in a container. Here is its definition:

```
template <typename InputIterator, typename EqualityComparable>
iterator_traits<InputIterator>::difference_type
count(InputIterator first,
      InputIterator last,
      const EqualityComparable& value);
```

count counts the number of occurrences of the value between first and last and returns the result. Here is an example of using count:

```
int num = count(v.begin(), v.end(), 10);
cout << "Found " << num << " occurrences of 10." << endl;
```

Notes

count_if

count_if is like count and find_if. It counts every element in the range that satisfies the predicate. A predicate is a function object that tests some condition and returns true or false.

```
template <typename InputIterator, typename Predicate>
iterator_traits<InputIterator>::difference_type
count_if(InputIterator first,
         InputIterator last,
         Predicate pred);
```

This example counts the even numbers in the given range:

```
int numEvens = count_if(v.begin(), v.end(), evenPred ());
cout << "Found " << numEvens << " even numbers" << endl;
```

Notes

10.5 Max and Min

- max: returns larger of two items, possible using a *binary predicate*.
- max_element: finds largest item in a *range*, may use a *binary_predicate*.
- min: returns larger of two items, possible using a *binary_predicate*.
- min_element: finds largest item in a *range*, may use a *binary_predicate*.

max_element and min_element

Like find and find_if, they return an iterator pointing to the element found. If none is found, the iterator will point to the end of the range.

```
ForwardIterator
max_element(ForwardIterator first,
            ForwardIterator last)

ForwardIterator
 min_element(ForwardIterator first,
            ForwardIterator last)
```

Notice that max_element and min_element require Forward Iterators, not just Input Iterators. This is because they have to save the iterator of the largest or smallest element found, and Input Iterators don't support saving. Here is an example:

Notes

```
vector<int> stats;
vector<int>::iterator result;
...
cout << "Looking for max number" << endl;
result = max_element(stats.begin(), stats.end());

if (result != stats.end())
        cout << "Found " <<  (*result) << endl;
```

This version of max_element and min_element uses operator< to the biggest and smallest items. One can also pass a function object to max_element and min_element. The predicate should take two arguments, the current maximum and an element, and return true only when the element should replace the current maximum.

```
ForwardIterator
 max_element(ForwardIterator first,
                  ForwardIterator last,
                  Compare pred)

ForwardIterator
min_element(ForwardIterator first,
                 ForwardIterator last,
                 Compare pred)
```

Notes

Here's how to find the largest odd number in a collection:

```
class MaxOdd {
  public:
    bool operator() (int theMax, int x)
    { return ((x % 2) == 1) && (x > theMax); }
};
cout << "Looking for max odd" << endl;
result = max_element(stats.begin(), stats.end(), MaxOdd());

if (result != stats.end())
  cout << "Found " <<  (*result) << endl;
```

Notes

10.6 Comparing Ranges

- mismatch: search two parallel ranges and returns position of the first one that is unequal or doesn't satisfy a binary_predicate.
- search: look in first range for an occurrence of the second range, possibly using a binary_predicate.
- search_n: look in range for an occurrence of n items equal to a value, possibly using a binary_predicate.
- equal: test if a range equals, element another parallel range, possibly using a binary_predicate
- for_each: Apply a function to every item in a range.

Here is an example using for_each:

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

template<typename T> struct print : public unary_function<T, void>
{
  print(ostream& out) : os_(out), i_ (0) {}
  void operator() (const T &x) { os_ << x << ' '; ++i_; }
  int count () { return i_; }
  ostream& os_;
  int i_;
};
```

Notes

```
int main()  {
  int anArray[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(anArray) / sizeof(int);
  print<int> fun = for_each(anArray, anArray + N, print<int>(cout));
  cout << endl << fun.count() << " objects printed." << endl;
  return 0;
}
```

Notes:

Unit Eleven
Mutating Algorithms

Unit topics:

Notes:

11.1 Mutating Algorithms

Mutating algorithms sound as if they modify the actual elements in the container. What they actually do is to reorganize the elements based on some rule.

These algorithms work with iterators and since the iterators can be used with different types of containers, or even no containers, algorithms cannot remove or change the value of elements. In addition, since they only receive iterators, algorithms can't even figure out what container the elements are in!

For example, the **remove** and **remove_if** only reorganize the pointers to the elements, moving the "removed" elements to the end of the sequence and returning an iterator that indicates the first element that was "removed" (i.e., the first element that is not part of the resulting sequence).

Other code must remove the elements no longer wanted.

List of mutating algorithms

copy	copy_n	copy_backward	iter_swap
fill	fill_n	generate	generate_n
partition	random_shuffle	random_sample	random_sample_n
replace	replace_if	replace_copy	replace_copy_if
remove	remove_if	remove_copy	remove_copy_if
reverse	reverse_copy	rotate	rotate_copy
stable_partition	swap	swap_ranges	transform
unique	unique_copy		

OutputIter result_end = **copy**(first, last, result);
BidirectionalIter result_begin = **copy_backward**(first, last, result_end);
fill(first, last, value);
fill_n(first, count, value);
generate(first, last, generator);
generate_n(first, count, generator);
random_shuffle(first, last[, rand]);
replace (first, last, old_value, new_value);
replace_if(first, last, pred, new_value);
OutputIter result_end = **replace_copy**(first, last, result, oldval, newval);
OutputIter result_end = **replace_copy_if**(first, last, result, pred, newval);
ForwardIter new_last **remove**(first, last, value);
ForwardIter new_last **remove_if**(first, last, pred);
OutputIter result_end **remove_copy**(first, last, result, value);
OutputIter result_end **remove_copy_if**(first, last, result, pred);
reverse(first, last);
OutputIter result_end = **reverse_copy**(first, last, result);
rotate(first, middle, last);
OutputIter result_end = **rotate_copy**(first, middle, last, result);
swap(a, b);
ForwardIter2 new_last2 = **swap_ranges**(first1, last1, first2);
OutputIter result_end = **transform**(first, last, result, op);
OutputIter result_end = **transform**(first1, last2, first2, result, bin_op);
ForwardIter new_last = **unique**(first, last[, bin_pred]);
ForwardIter result_end = **unique_copy**(first, last[, bin_pred]);

Notes:

Notes:

11.2 Filling and Generating

- **fill**: change a range to all have the same given value.
- **fill_n**: change n items to all have the same given value.
- **generate**: change items in a range to be values produced by a function object.
- **generate_n**: change n items to be values produced by function object.
- **transform**: scans a range and for each use a function to generate a new object put in a second container, OR takes two intervals and applies a binary operation to items to generate a new container.

One example of a mutating algorithm is **transform** which requires four parameters: two iterators to specify the input sequence, one to specify the output sequence and another to specify the transforming operation. It returns a non-boolean value.

The following code is an example of the transform algorithm:

```
#include <iostream>
#include <algorithm>
#include <ctype.h>
#include <functional>
#include <string>

using namespace std;
```

Notes:

```
class to_lower
{
public:
  char operator() (char c) const       //returns char
  {
    return isupper (c) ? tolower(c) : c;
  }
};

string lower (const string &str)
{
    string lcase;
    transform (str.begin(), str.end(), back_inserter (lcase), to_lower());
    return lcase;
}

int main ()
{
  string s = "HELLO";
  cout << s << endl;
  s = lower (s);
  cout << s << endl;
  return 0;
}
```

Here is another example using transform, copy and generate:

Notes:

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
#include <iterator>
#include <cstdlib>
#include <ctime>
   using namespace std;

int main() {
    srand(  time(0) );
    vector<int> v;
    generate_n(  back_inserter(v), 10,  rand );
    vector<int> v2( 10, 20 );

    copy( v.begin(), v.end(), ostream_iterator<int>( cout," ") );
    cout <<  endl;
    transform( v.begin(), v.end(),  v2.begin(),  v.begin(),  modulus<int>() );
    // transform(v.begin(), v.end(), v.begin(), bind2nd(modulus<int>(), 20));
    copy( v.begin(), v.end(),  ostream_iterator<int>( cout," ") );
    cout <<  endl;
    sort( v.begin(), v.end(),  greater<int>() );
    copy( v.begin(), v.end(),  ostream_iterator<int>( cout," ") );
    cout <<  endl;
    return 0;
}
```

Notes:

First, generate_n is used to fill the vector with data. The vector is empty, but the magic of an back Insert Iterator makes it so that when generate_n writes to the output iterator it adds it to the end of the vector. One'll notice that in this case a function pointer to rand makes a perfectly acceptable Functor.

The transform line does the equivalent of v[i] = v[i] % v2[i]; for each v[i] in v. The first 3 parameters are the 2 input ranges -- the second input range is assumed to be at least as large as the first. The fourth parameter is the output iterator. In this case it's safe to use a normal iterator into the container since we know exactly how many elements will need to be written. The fifth parameter is the binary operation performed.

This example uses several STL algorithm functions to work with the elements of two vectors, each holding 10 integers. It also uses the plus() functor with the **transform** algorithm to add the elements.

```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>
#include <numeric>
#include <iterator>
  using namespace std;

int main() {
```

Notes:

```
  vector<int> V1(10), V2(10);

  fill(V1.begin(), V1.end(), 1);
  partial_sum(V1.begin(), V1.end(),V1.begin() );

  random_shuffle(V1.begin(), V1.end());
  fill(V2.begin(), V2.end(), 2);
  partial_sum(V2.begin(), V2.end(),V2.begin() );
  random_shuffle(V2.begin(), V2.end());

  copy(V1.begin(),V1.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  copy(V2.begin(),V2.end(), ostream_iterator<int>(cout, " "));
  transform(V1.begin(), V1.end(), V2.begin(), V2.begin(),plus<int>());
  cout << endl;  cout << endl;
  copy(V2.begin(),V2.end(), ostream_iterator<int>(cout, " "));
  cout <<endl;
  return 0;
}
```

Notes:

11.3 Manipulating Sequences

- swap: swaps values of two given variables.
- iter_swap: swaps two items in a container indicated by iterators.
- swap_ranges: interchanges value between two ranges.
- reverse: places the elements in the reverse order.
- reverse_copy: creates a backwards copy of a range.
- rotate: given a middle point in a range, reorganizes range so that middle comes first.
- rotate_copy: creates a rotated copy.

rotate

```
template <typename ForwardIterator>
  void rotate ( ForwardIterator first, ForwardIterator middle,
                ForwardIterator last ); <algorithm>
```

Rotate elements in range

Rotates the order of the elements in the range [first,last), in such a way that the element pointed by middle becomes the new first element.

```
template <typename ForwardIterator>
  void rotate ( ForwardIterator first, ForwardIterator middle,
                ForwardIterator last )
{
```

Notes:

```
  ForwardIterator next = middle;
  while (first!=next)
  {
    swap (*first++,*next++);
    if (next==last) next=middle;
    else if (first == middle) middle=next;
  }
}
```

Here is an example:

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>
using namespace std;

int main () {
  vector<int> myVect;
  vector<int>::iterator it;

  for (int i=1; i<10; ++i)
      myVect.push_back(i);

  cout << "myVect originally contains:\n";
  copy (myVect.begin (), myVect.end (), ostream_iterator<int> (cout, " "));
```

Notes:

```
  rotate(myVect.begin(),myVect.begin()+5,myVect.end());

    cout << "\nmyVect rotated contains:\n";
  copy (myVect.begin (), myVect.end (), ostream_iterator<int> (cout, " "));
  cout << endl;
  return 0;
}
```

reverse

```
template <typename BidirectionalIterator>
  void reverse ( BidirectionalIterator first, BidirectionalIterator
last);<algorithm>
```

Reverses the order of the elements in the range [first,last). Behaves like this:

```
template <typename BidirectionalIterator>
  void reverse ( BidirectionalIterator first, BidirectionalIterator last)
{
  while ((first!=last)&&(first!=--last))
    swap (*first++,*last);
}
```

Notes:

Here is an example program using it:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

int main () {
  vector<int> myVect;

  for (int i=1; i<10; ++i) myVect.push_back(i);

  reverse(myVect.begin(),myVect.end());
  cout << "myVect contains:";
  copy (myVect.begin (), myVect.end (), ostream_iterator<int> (cout, " "));
  cout << endl;

  return 0;
}
```

Notes:

11.4 Remove

- remove: deletes items in a range that equal a given value.
- remove_if: deletes items in a range if a predicate is true.
- remove_copy: makes a copy of items in a range but not those with a given value.
- remove_copy_if: makes a copy of items in a range but not those where a predicate is true.

remove

```
template < class ForwardIterator, typename T >
  ForwardIterator remove ( ForwardIterator first, ForwardIterator last,
                           const T& value ); <algorithm>
```

Removes from the range [first,last) the elements with a value equal to value and returns an iterator to the new end of the range, which now includes only the values not equal to value.

The behavior of this function template is equivalent to:

```
template < class ForwardIterator, typename T >
  ForwardIterator remove ( ForwardIterator first, ForwardIterator last, const
T& value )
{
  ForwardIterator result = first;
```

Notes:

```
  for ( ; first != last; ++first)
    if (!(*first == value)) *result++ = *first;
  return result;
}
```

This function does not alter the elements past the new end, which keep their old values and are still accessible. Here is an example:

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main ()
{
  int myints[] = {10,20,30,30,20,10,10,20};
  // bounds of range:
  int* pbegin = myints;
  int* pend = myints+sizeof(myints)/sizeof(int);

  cout << "original array contains:";
  copy (pbegin, pend, ostream_iterator<int> (cout, " "));
  cout << endl;

  int *nend = remove (pbegin, pend, 20);
```

Notes:

```
  cout << "range contains:";
  copy (pbegin, nend, ostream_iterator<int> (cout, " "));
  cout << endl;

  cout << "complete array contains:";
  copy (pbegin, pend, ostream_iterator<int> (cout, " "));
  cout << endl;

  return 0;
}
```

remove_if

```
template < class ForwardIterator, typename Predicate >
  ForwardIterator remove_if ( ForwardIterator first, ForwardIterator last,
                              Predicate pred ); <algorithm>
```

Removes from the range [first,last) the elements for which pred applied to its value is true, and returns an iterator to the new end of the range, which now includes only the values for which pred was false.

The behavior of this function template is equivalent to:

```
template < class ForwardIterator, typename Predicate >
  ForwardIterator remove_if ( ForwardIterator first, ForwardIterator last,
                              Predicate pred )
```

```
{
  ForwardIterator result = first;
  for ( ; first != last; ++first)
    if (!pred(*first)) *result++ = *first;
  return result;
}
```

Notes:

Notes:

Here is an example using **remove_if**:

```
#include <iostream>
#include <algorithm>
using namespace std;

bool IsOdd (int i) { return ((i%2)==1); }

struct IsEvenOdder {
  bool operator () (int i) { return ((i%2)==1); }
};

int main () {
  int myints[] = {1,2,3,4,5,6,7,8,9};

  // bounds of range:
  int* pbegin = myints;
  int* pend = myints+sizeof(myints)/sizeof(int);

  pend = remove_if (pbegin, pend, IsEvenOdder ());
  pend = remove_if (pbegin, pend, IsOdd);

  cout << "range contains:";
  copy (pbegin, pend, ostream_iterator<int> (cout, " "));
  cout << endl;
  return 0;
}
```

Notes:

remove_copy

```
template <typename InputIterator, typename OutputIterator, typename T>
  OutputIterator remove_copy ( InputIterator first, InputIterator last,
                               OutputIterator result, const T& value ); <algorithm>
```

Copies the values of the elements in the range [first,last) to the range positions beginning at result, except for the elements that compare equal to value, which are not copied.

The behavior of this function template is equivalent to:

```
template <typename InputIterator, typename OutputIterator, typename T>
  OutputIterator remove_copy ( InputIterator first, InputIterator last,
                               OutputIterator result, const T& value )
{
  for ( ; first != last; ++first)
    if (!(*first == value)) *result++ = *first;
  return result;
}
```

Notes:

Here is a program using **remove_copy**:

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

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  vector<int> myVect;
  vector<int>::iterator it;
  remove_copy (myints,myints+8, back_inserter (myVect), 20);

  cout << "myvector contains:";
  copy (myVect.begin(), myVect.end(), ostream_iterator<int> (cout, " "));
  cout << endl;
  return 0;
}
```

Notes:

Here is a program using remove_copy_if:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

struct twenty : unary_function<int, int>
{
        bool operator () (int x) const { return x == 20; }
};

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  vector<int> myVect;
  vector<int>::iterator it;
  remove_copy_if (myints,myints+8,
                          back_inserter (myVect), not1 (twenty ()));
 // remove_copy_if (myints,myints+8,
 //                          back_inserter (myVect),
 //                           not1 (bind2nd (equal_to<int> (), 20)));

  cout << "myvector contains:";
  copy (myVect.begin(), myVect.end(), ostream_iterator<int> (cout, " "));
  cout << endl;  return 0;
}
```

Notes:

11.5 Replace

- replace: scan a range and replace given old values by given new value.
- replace_if: scan a range and replace given old values by given new value IF a predicate is true.
- replace_copy: make a copy of a range but replace given old values by given new value.
- replace_copy_if: make a copy of a range but replace given old values by given new value IF a predicate is true.

```
template <typename ForwardIterator, typename T>
void replace(ForwardIterator first,
             ForwardIterator last,
             const T& old_value,
             const T& new_value)
{
  while (first != last)  {
    if (*first == old_value)  // Can read from and assign to *first iterator
       *first = new_value;
    ++first;
  }
```

```
}
```

Notes:

Notes:

11.6 Sort and Merge

Sorting algorithms are a special type of mutating algorithms which change the positions of data elements in a collection. Searching and comparison algorithms are special types of non-mutating algorithms that work best with sorted data.

In general, programming algorithms are often categorized by their efficiency, which means how long they take to run and how much storage or memory is used during their running. This is most important for those that sort or search large collections of data.

Several algorithms that perform searching or comparison operations require sorted input in order to work correctly: **binary_search, equal_range, inplace_merge, includes, lower_bound, merge, set_difference, set_intersection, set_symmetric_difference, set_union** and **upper_bound**. The **unique** and **unique_copy** algorithms work best with sorted input, but don't require it.

Therefore, it is important to choose the most efficient sort algorithm for the task on hand. We define efficient in terms of time using the concept of complexity.

Notes:

Algorithmic complexity is generally discussed in a form known as Big-O notation, where the *O* represents the complexity of the algorithm and a value *n* represents the size of the set the algorithm is run against. For example, ***O*(*n*)** means that an algorithm has a linear complexity.

For linear processes, it takes ten times longer to operate on a set of 10 items than it does on 1 item and ten times longer again to operate on 100 items than it does on a set of 10 items (10 * 10 = 100). So if reading a file of 10 lines takes 1 CPU millisecond then reading a file containing 100 lines would take 100 milliseconds for a linear algorithm. This is the complexity of a one loop algorithm.

But if the complexity is $\boldsymbol{O(n^2)}$ (quadratic complexity), then it takes 100 times longer to operate on a set of 100 items than it does on a set of 10 items. This type of algorithm has 2 nested loops with one inside another. The sorting algorithms that are $\boldsymbol{O(n^2)}$ in efficiency include the bubble, insertion, selection and shell sorts.

Finally, there are some sorting algorithms which have complexity of **O(n log n)**: the heap, merge, and quick sorts. These algorithms are fast, but that speed comes at the cost of complexity; these algorithms use recursion, advanced data structures and multiple arrays.

Notes:

The most important thing is to choose the sorting algorithm that is most appropriate for a custom program. Here is a list of the STL sorting, set, and heap algorithms:

adjacent_difference	binary_search
equal_range	includes
inplace_merge	is_heap
is_sorted	lexicographical_compare
lexicographical_compare_3way	lower_bound
make_heap	max
max_elementmerge	min
min_element	nth_element
partial_sort partial_sort_copy	pop_heap
power	push_heap
set_difference	set_intersection
set_union	set_symmetric_difference
stable_sort	sort
sort_heap	upper_bound

Notes:

How to call sorting related algorithm functions

OutputIter result_end = **adjacent_difference**(first, last, result[, bin_op]);
bool is_present = **binary_search**(first, last, value[, comp]);
pair<ForwardIter, ForwardIter> loc_range = **equal_range**(first, last, value[, comp]);
bool is_included = **includes**(first, last, subseq_first, subseq_last[, comp]);
inplace_merge(first, middle, last[, comp]);
bool is_inorder = **is_sorted**(first, last[, comp]);
bool is_less = **lexicographical_compare**(first1, last1, first2, last2[, bin_pred]);
ForwardIter loc = **lower_bound**(first, last, value[, comp]);
OutputIter result_end = **merge**(first1, last1, first2, last2, result[, comp]);
nth_element(first, nth, last[, comp]);
partial_sort(first, middle, last[, comp]);
RandomAccessIter result_end = **partial_sort_copy**(first, middle, last, result_first, result_last[, comp]);
OutputIter result_end = **partial_sum**(first, last, result[, bin_op]);
BidirectionalIter middle = **partition**(first, last, pred);
RandomAccessIter result_end = **random_sample**(first, last, out_first, out_last[, rand])
RandomAccessIter result_end = **random_sample_n**(first, last, out_first, count[, rand])
stable_sort(first, last[, comp]);
BidirectionalIter middle = **stable_partition**(first, last, pred);
sort(first, last[, comp]);
ForwardIter loc = **upper_bound**(first, last, value[, comp]);

Notes:

Sorting Concepts

The following is a discussion of several sorting concepts to enhance programmers' understanding when deciding which sorting algorithms to choose or ways to implement them.

Bubble Sort

Bubble sort is the oldest and easiest to implement, but with a complexity of $O(n^2)$ it is also the slowest sort. It works by comparing each item in the list with the item next to it and swapping them if required. It then repeats this process until it makes a pass all the way through the list without swapping any items because all items are in the correct order. This causes larger values to "bubble" to the end of the list while smaller values "sink" toward the beginning.

Heap Sort

Heap sort is the slowest of the $O(n \log n)$ sorting algorithms, but unlike the merge and quick sorts it doesn't require lots of recursion or multiple arrays. This makes it the most efficient choice for very large data sets containing millions of items. It begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array.

This is repeated until there are no items left in the heap and the sorted array is full. Simple implementations require two arrays - one to hold the heap and the other to hold the sorted elements, but more complex implementations use one array; when an item is removed from the heap, it frees up a space at the end of the array that the removed item can be placed.

Notes:

Insertion Sort

Insertion sort has a complexity of $O(n^2)$ and it is a more than twice as efficient as the bubble sort, but is inefficient for large sets of data. It inserts each item into its proper place in the final list. The simplest implementation uses two list structures - the source list and the list into which sorted items are inserted. But, to use storage efficiently, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Merge Sort

Merge sort has a complexity of $O(n \log n)$, and is only a little faster than the heap sort for larger datasets. But it needs at least twice the memory storage of the other sorts because it splits the list of data to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted and then merged back together to form the final sorted list. Simple implementations use three arrays - one for each half of the data set and one to store the sorted results.

Notes:

Quick Sort

Quick sort is an in-place, divide-and-conquer, recursive kind of sort which can have complexity of $O(n \log n)$. There are four steps to this sorting method: If there are one or less elements in the array to be sorted, return immediately. Then an element in the array is chosen to serve as a "pivot" point. The array is split into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot. Finally the sort recursively repeats the algorithm for both halves of the original array until everything is sorted.

Selection Sort

Selection sort has a complexity of $O(n^2)$. It works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. It is inefficient for large collections of data.

Shell Sort

Shell Sort was invented by Donald Shell. It is the most efficient of the $O(n^2)$ complexity sorting algorithms. This sort makes multiple passes through the data, each time sorting a number of equally sized subsets using the insertion sort. The size of the sets to be sorted gets larger with each pass through the data, until there is only one set left consisting of the entire list of data. It is efficient for medium sized lists of data, but not nearly as efficient as the merge, heap or quick sorts.

Notes:

IntroSort

Introsort is a new, hybrid sorting algorithm, created by David Musser, (one of the guys who created the STL). It acts almost like quick sort for most data and is just as fast, but smarter, than quick sort because it detects when partitioning the data is taking too long and switches its process to a heap sort algorithm at that point.

STL Sorting Algorithms

There are only a few of these sorting algorithms implemented in the STL:

merge uses the **merge sort** algorithm and combines two sorted ranges into a single sorted range of data. There are two versions: one uses the **<** operator for comparisons and the other requires a functor comparator. **inplace_merge** is another STL merge sort algorithm.

partial_sort and **partial_sort_copy** use the **heap sort** algorithm and sort the entire range of data. Both heap sort and introsort have the same complexity, but introsort is usually faster.

stable_sort uses the **merge sort** algorithm. If one are sorting a sequence of records that have several different key fields, then one may want to sort it by one field without completely destroying the ordering that one previously obtained from sorting it by a different field. For example, sorting by last name and then by first name.

sort currently uses the **introsort** implementation created by Musser, and sorts the entire range of data. This allows the efficiency of using quick sort until it becomes inefficient at which point it shifts to heap sort. Earlier versions of sort used the quick sort algorithm.

Below is an example of using **sort** on a vector. The sort algorithm orders the container's contents in ascending order, as defined by the "less than" (**<**) operator as applied to the vector elements.

Notes:

Notes:

```
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
     vector<int> aVector;
     aVector.push_back (10);
     aVector.push_back (3);
     aVector.push_back (7);              // vector contains 10,3,7 in order
     sort(aVector.begin(), aVector.end());   // vector now has 3,7,10
}
```

Here is another example using a vector and the sort() algorithm function.

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
using namespace std;

int main()  {
   int sum=0;
   vector<int> V;
   V.push_back(1); V.push_back(4);V.push_back(2);
   V.push_back(8); V.push_back(5);V.push_back(7);
```

Notes:

```
  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));

  sum = count_if (V.begin(), V.end(), bind2nd(greater<int>(),5));
  cout << endl << "There are " << sum << " number(s) greater than 5"
       << endl;

  // "remove" all the elements less than 4
  vector<int>::iterator new_end =
       remove_if(V.begin(), V.end(), bind2nd(less<int>(), 4));

 // remove_if doesn't actually remove elements. It moves the unwanted
 // elements to the end and returns an iterator pointing to the first
 // of these unwanted elements. It works this way because it's a generic
 // routine and it doesn't "know" whether the size of the underlying data
 // structure can be changed. However, v.erase() does know
 V.erase(new_end, V.end());

 copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
 cout << endl << "Elements less than 4 removed" << endl;

 sort(V.begin(), V.end());
 copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
 cout << endl << "Elements sorted" << endl;
 return 0;
}
```

Notes:

Example using Insertion Sort:

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

void sort( vector<int>::iterator begin, vector<int>::iterator end)
{
	vector<int>::iterator sorted;//lowest sorted item
	for (sorted = end-1; sorted != begin; sorted--)
	 {
	   vector<int>::iterator next = sorted-1;//put in place
	    int value = *next;
	    vector<int>::iterator i;
	   for (i = sorted; i != end; i++)
	    {
	      if (value <= *i)
	              break;
	        *next = *i;
	        next++;
	    }
	    *next=value;
	}
}
```

Notes:

```
int main()  {
   const int seed = static_cast<int>(time(0));
   srand(seed);//set random number differently each run
   const int Biggest = 100000;
   const int Size = 5000;
   const int Sample = 100;

   double total_time =0.0;

   for(int s = 0; s<Sample; s++)
   {
      vector <int> data;
      for(int i = 0; i< Size; i++)
            data.push_back(rand()%Biggest);
      time_t time1 = time(0);
      sort(data.begin(), data.end());
      time_t time2 = time(0);

      total_time+= difftime(time2, time1);
   }
   cout << "Insertion Sort. Size =" << Size << ", mean time =";
   cout << total_time /Sample;
   return 0;
}
```

The bubble sort algorithm is an example of a multi-pass one-directional algorithm. Here the template for this algorithm.

Notes:

```
template <typename BidirectionalIterator, typename Compare>
void bubble_sort (BidirectionalIterator first, BidirectionalIterator last,
                  Compare comp)
{
  BidirectionalIterator leftElement = first, rightElement = first;
  rightElement++;
  while (first != last)
  {
    while (rightElement != last) {
      if (comp(*rightElement, *leftElement))
        iter_swap (leftElement,   rightElement);
        rightElement++;
        leftElement++;
    }
    last--;
    leftElement = first, rightElement = first;
    rightElement++;
  }
}
```

Notes:

Unit Twelve
Other Algorithms

Unit topics:

Notes:

12.1 Set Algorithms

Calling set related algorithms

OutputIter result_end = **set_difference**(first1, last1, first2, last2, result[, comp]);
OutputIter result_end = **set_intersection**(first1, last1, first2, last2, result[, comp]);
OutputIter result_end = **set_symetric_difference**(first1, last1, first2, last2, result[, comp]);
OutputIter result_end = **set_union**(first1, last1, first2, last2, result[, comp]);

In addition, an output stream can be thought of as a container, thus the copy algorithm can be used with other containers to copy elements to it.

The line: copy(s1.begin(), s1.end(), ostream_iterator<int>(cout," ")); also puts spaces between the integers being printed.

Sometimes it is not known how many resulting elements will be produced by an operation, but it is necessary to add them all to a container. Use the iterator adaptors: inserter (which calls the container's insert() routine), front_inserter (which calls the container's push_front() routine - which vectors don't have) or back_inserter (which calls push_back()).

Notes:

Example:

```
#include <algorithm>
#include <set>
#include <iostream>
#include <iterator>
   using namespace std;

int main()  {
  int a1[10] = {1,2,3,4,5,6,7,8,9,10};
  int a2[6]  = {2,4,6,8,10,12};
  set<int> s1(a1, a1+10), s2(a2, a2+6), answer;
  cout << "s1=";
  copy(s1.begin(), s1.end(),
      ostream_iterator<int>(cout," "));
  cout << "\ns2=";
  copy(s2.begin(), s2.end(),
      ostream_iterator<int>(cout," "));
  set_difference(s1.begin(), s1.end(),
                 s2.begin(), s2.end(),
                 inserter(answer,answer.begin()));
  cout << "\nThe set-difference of s1 and s2 =";
  copy(answer.begin(), answer.end(),
      ostream_iterator<int>(cout," "));
  cout << endl;
  return 0;
}
```

Notes:

set_union

Union of two sorted ranges

Constructs a sorted range beginning in the location pointed by result with the set union of the two sorted ranges [first1,last1) and [first2,last2) as content.

The union of two sets is formed by the elements that are present in either one of the sets, or in both.

The comparison to check for equivalence of values, uses either operator< for the first version, or comp for the second, in order to test this; The value of an element, a, is equivalent to another one, b, when (!a<b && !b<a) or (!comp(a,b) && !comp(b,a)).

For the function to yield the expected result, the elements in the ranges shall be already ordered according to the same strict weak ordering criterion (operator< or comp).

The behavior of this function template is equivalent to:

```
template <typename InputIterator1, typename InputIterator2, typename
OutputIterator>
  OutputIterator set_union ( InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
```

Notes:

```
                    OutputIterator result )
{
  while (true)
  {
    if (*first1<*first2) *result++ = *first1++;
    else if (*first2<*first1) *result++ = *first2++;
    else { *result++ = *first1++; first2++; }

    if (first1==last1) return copy(first2,last2,result);
    if (first2==last2) return copy(first1,last1,result);
  }
}
```

Here is an example program:

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

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);
  vector<int>::iterator it;

  sort (first,first+5);
```

```
  sort (second,second+5);

  it=set_union (first, first+5, second, second+5,
                v.begin());

  cout << "union has " << int(it - v.begin())
       << " elements.\n";
  return 0;
}
```

Notes:

set_intersection

Intersection of two sorted ranges

Constructs a sorted range beginning in the location pointed by result with the set intersection of the two sorted ranges [first1,last1) and [first2,last2) as content.

The intersection of two sets is formed only by the elements that are present in both sets at the same time.

The comparison to check for equivalence of values, uses either operator< for the first version, or comp for the second, in order to test this; The value of an element, a, is equivalent to another one, b, when (!a<b && !b<a) or (!comp(a,b) && !comp(b,a)).

Notes:

For the function to yield the expected result, the elements in the ranges shall be already ordered according to the same strict weak ordering criterion (operator< or comp).

The behavior of this function template is equivalent to:

```
template <typename InputIterator1, typename InputIterator2, typename
OutputIterator>
  OutputIterator set_intersection ( InputIterator1 first1, InputIterator1 last1,
                                    InputIterator2 first2, InputIterator2 last2,
                                    OutputIterator result )
{
  while (first1!=last1 && first2!=last2)
  {
    if (*first1<*first2) ++first1;
    else if (*first2<*first1) ++first2;
    else { *result++ = *first1++; first2++; }
  }
  return result;
}
```

Here is an example program:

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

Notes:

```
int main () {
	int first[] = {5,10,15,20,25};
	int second[] = {50,40,30,20,10};
	vector<int> v;
	sort (first,first+5);
	sort (second,second+5);
	set_intersection (first, first+5, second, second+5, back_inserter (v));

	cout << "intersection has " << int(v.size ()) << " elements:\n";
	copy (v.begin(), v.end (), ostream_iterator<int> (cout, " "));
	cout << endl;
	return 0;
}
```

set_difference

Difference of two sorted ranges

Constructs a sorted range beginning in the location pointed by result with the set difference of range [first1,last1) with respect to [first2,last2) as content.

The difference of two sets is formed by the elements that are present in the first set, but not in the second one. Notice that this is a directional operation - for a symmetrical equivalent, see set_symmetric_difference.

Notes:

The comparison to check for equivalence of values, uses either operator< for the first version, or comp for the second, in order to test this; The value of an element, a, is equivalent to another one, b, when (!a<b && !b<a) or (!comp(a,b) && !comp(b,a)).

For the function to yield the expected result, the elements in the ranges shall be already ordered according to the same strict weak ordering criterion (operator< or comp).

The behavior of this function template is equivalent to:

```
template <typename InputIterator1, typename InputIterator2, typename
OutputIterator>
  OutputIterator set_difference ( InputIterator1 first1, InputIterator1 last1,
                                  InputIterator2 first2, InputIterator2 last2,
                                  OutputIterator result )
{
  while (first1!=last1 && first2!=last2)
  {
    if (*first1<*first2) *result++ = *first1++;
    else if (*first2<*first1) first2++;
    else { first1++; first2++; }

  }
  return copy(first1,last1,result);
}
```

Notes:

Here is an example:

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

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);
  vector<int>::iterator it;

  sort (first,first+5);
  sort (second,second+5);

  it=set_difference (first, first+5, second, second+5,
                     v.begin());
  cout << "difference has " << int(it - v.begin())
       << " elements.\n";
  return 0;
}
```

Notes:

12.2 Heap Algorithms

sort_heap uses the heap sort algorithm. It is one of several heap algorithms: push_heap, pop_heap, sort_heap and is_heap_sorted. The STL doesn't come with heap containers however there are some extensions that have them.

Example:

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

int main() {
   const int seed = static_cast<int>(time(0));
   srand(seed);
   const int Biggest = 100000;
   const int Size = 500000;
   const int Sample = 100;
   double total_time =0.0;

   for(int s = 0; s<Sample; s++)
   {
```

Notes:

```
    vector <int> data;
    for(int i = 0; i< Size; i++)
        data.push_back(rand()%Biggest);
     time_t time1 = time(0);
    make_heap(data.begin(), data.end());
    sort_/******/(data.begin(), data.end());
    time_t time2 = time(0);
    total_time+= difftime(time2, time1);
  }
  cout << "Heap Sort. Size =" << Size << ", mean time =";
  cout << total_time /Sample;
  return 0;
}
```

is_heap

```
bool is_heap( iterator start, iterator end );
bool is_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The is_heap() function returns true if the given range [start,end) is a heap. The StrictWeakerOrdering is a function object that defines 'less than' rule to be used for the sorting.

Notes:

pop_heap

```
void pop_heap( iterator start, iterator end );
void pop_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The pop_heap() function removes the larges element (defined as the element at the front of the heap) from the given heap.

sort_heap

```
void sort_heap( iterator start, iterator end );
void sort_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The sort_heap() function turns the heap defined by [start,end) into a sorted range.

Notes:

12.3 Numeric Algorithms

The numeric algorithms are math functions found in **<numeric> .**

accumulate
It calculates the sum of all the elements in a specified range. It doesn't change the input data and can also concatenate strings. It can also give the product of the data in the range.

adjacent_difference
It calculates the successive differences between each data element and the one before it, given an input range. It outputs the results to a destination range or performs any operation where the difference operation is replaced by another binary operation passed as an argument.

checked_adjacent_difference
It works the same way as adjacent_difference, but uses a checked iterator on the output iterator.

checked_partial_sum
It works the same way as partial_sum (see below), but uses a checked iterator on the output iterator.

Notes:

inner_product
It calculates the sum of the element-wise product of two ranges and adds it to a specified initial value or calculates the result of a generalized procedure where the sum and product binary operations are replaced by other specified binary operations.

partial_sum
It calculates a series of sums in an input range from the first element through the nth element and stores the result of each such sum in nth element of a destination range or it calculates the result of a generalized procedure where the sum operation is replaced by another specified binary operation.

unchecked_adjacent_difference
It is the same as adjacent_difference, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.

unchecked_partial_sum
It is the same as partial_sum, but allows the use of an unchecked iterator as output iterator when _SECURE_SCL=1 is defined.

Notes:

How call numeric algorithm functions

bool **next_permutation**(first, last[, comp]);
bool **prev_permutation**(first, last[, comp]);
T total = **accumulate**(first, last, init_total[, bin_op]);
T prod = **inner_product**(first1, last1, first2, init[, bin_op1, bin_op2]);

accumulate
The accumulate algorithm adds up a range of elements in a container.

```
T accumulate(InputIterator first,
             InputIterator last,
             T initial_value)
```

To add up all the numbers in our vector,

```
cout << "Sum of all the numbers is "
     << accumulate(stats.begin(), stats.end(), 0)
     << endl;
```

The third argument, 0 in this case, is the initial value for the sum.
The above call to accumulate uses operator+ but one can pass a function as a fourth argument to change this. That function should take two arguments. The first will be the "sum" so far, starting with the initial value given. Each element will be passed in turn to the second element.

Notes:

```
T accumulate(InputIterator first,
             InputIterator last,
             T initial_value,
             BinaryOperation op)
```

So, to add up just the odd numbers:

```
class SumOdd {
  public:
    int operator() (int sum, int y)
    { return ((y % 2) == 1) ? sum + y : sum; }
};
...
cout << "Sum of all the odd numbers is "
         << accumulate(stats.begin(), stats.end(), 0, SumOdd())
         << endl;
```

Unit Thirteen
Utilities

Unit topics:

13.1 Memory Allocators

Using and filling Memory

There are several memory allocators provided which allow a programmer to determine memory allocation strategies. All C++ standard library objects that allow for specification of an allocator have a default allocator that will be used if the program does not specify otherwise.

Custom allocators are needed when custom program code uses memory in unique ways. There are also other cases where the default allocator won't work well, such as when using standard containers with code that has its own replacements for the global operators **new** and **delete**. Thus several implementations of the STL have customized their allocators. Examples of what GNU and SGI provide are below.

GNU's libstdc++

- The C++ Standard encapsulates memory management characteristics for strings, container classes, and parts of iostreams in a template class called std::allocator.
- The C++ standard has these requirements:
  - When one add elements to a container, and the container must allocate more memory to hold them, the container makes the request via its Allocator template parameter. This includes adding characters to the string class.
  - The default Allocator of every container-of-T is std::allocator<T>.

- But std::allocator can allocate and de-allocate using implementation-specified strategies.
- So every call to an allocator object's allocate member function may not actually call the global operator **new**, and this is also true for calls to the de-allocate member function, **delete**.
- Starting with GCC 3.2, to globally disable memory caching within the library for the default allocator, set GLIBCPP_FORCE_NEW in the system's environment before running a program.
- The 3.4 code base continues to use this mechanism, only the environment variable is now GLIBCXX_FORCE_NEW.
- Starting with gcc-3.4, all extension allocators are standard style. Before this, SGI style was used

SGI's allocator

- The current SGI STL also supports the allocator interface in the C++ standard.
- But it is specific to the SGI STL; cannot be used in code that must be portable to other STL implementations
- The default allocator alloc maintains its own free lists for small objects
- The SGI allocator, alloc
  - obtains memory from malloc (not new)
  - does not free all memory - Memory allocated for blocks of small objects is not returned to malloc
  - leak detection tests should be run with malloc_alloc not alloc
  - The default allocator makes no special attempt to ensure that consecutively allocated objects are "close" to each other

13.2 The smart pointer: auto_ptr

auto_ptr acts like C++ pointer in many ways. But, when an auto_ptr object goes out of scope, it automatically deletes the memory that it is holding — something that ordinary C++ pointers fail to do.

In addition to memory leaks, the auto_ptr also prevents dangling references. How? When one assign one auto_ptr to another, the auto_ptr on the right hand side actually becomes the equivalent of a NULL pointer. The target of the assignment (on the left hand side) now points to the memory that the right hand side auto_ptr pointed to before.

An **auto_ptr** controls a dynamically allocated object and performs automatic cleanup when the object is no longer needed. Thus, if one don't use an **auto_ptr**, the program must be sure to free any memory that pointers use.

Example of using **auto_ptr**:

```
int main()
{
   T* regularPtr = new T;                  //create a pointer of any datatype
   auto_ptr<T> autoPtr( regularPtr ); // use auto_ptr to control it
   *autoPtr = 12;                          // same as "*regularPtr = 12;"

    autoPtr->MyFunct();                    // same as "regularPtr->MyFunct();"
```

Notes

```
    assert( regularPtr == autoPtr.get() );   // use get() to see the ptr value

   T* anotherPtr = autoPtr.release();  // use release() to remove auto_ptr
                                       // but save pointer value
   delete anotherPtr;                  // now we could use this pointer
                                       // when done we need to delete it
  }
```

If one create custom own data structures, it's also very useful to pass **auto_ptrs** to and from functions as function parameters. The **auto_ptrs** will return values because they 'own' or control the objects they point to and will correctly delete them when no longer needed. Thus no pointers are left dangling and memory is freed up efficiently.

13.3 The raw storage iterator

Raw storage iterators are used for efficiency when performing operations like copying existing container elements to regions of un-initialized memory, such as that obtained by the STL predefined algorithms **get_temporary_buffer** and **return_temporary_buffer**. It can also be used to iterate over un-initialized memory, initializing it with the results from any user defined algorithm. Here is the STL template:

```
#include <memory>
template <typename OutputIterator, typename T>
 class raw_storage_iterator : public output_iterator {
public:
   explicit raw_storage_iterator (OutputIterator);
   raw_storage_iterator<OutputIterator, t>& operator*();
   raw_storage_iterator<OutputIterator, T>&
     operator= (const T&);
   raw_storage_iterator<OutputIterator>& operator++();
   raw_storage_iterator<OutputIterator> operator++ (int);
};
```

13.4 Some Relational Operators

Inside the **<utility>** header, there are several useful general templates. Among them are necessary constants for multiple C++ environments, four relational operators and the pair template class.

```
template <typename T>
inline bool operator!=(const T& x, const T& y) {
  return !(x == y);
}

template <typename T>
inline bool operator>(const T& x, const T& y) {
  return y < x;
}

template <typename T>
inline bool operator<=(const T& x, const T& y) {
  return !( y < x);
}

template <typename T>
inline bool operator>=(const T& x, const T& y) {
  return !(x < y);
}
```

Notes

The four templates above require that the equality operator, **==**, be defined. Functors can use them to work on any datatype, even a class object.

Appendix
Resources

Unit topics:

Notes

Appendix A Optimization

Lacking in the STL:

- There is no **tree** collection, but one can use a set or multiset, which are implemented as trees.
- There are no **multidimensional arrays**, but a **vector** can hold other vectors as elements and there's also a Multi-Array class in Boost.
- There are no **graphs,** but one way to implement graphing is to have a **map** holding other **maps** or you can use the Boost Graph Library (BGL).
- There are no **sparse arrays**; this is a type of array where most of the entries are zero. The basic idea when storing sparse arrays is to only store the non-zero entries. This can save memory and processing time. The data can be compressed if the position and value of only non-zero data is stored. There's also a sparse matrix library in Boost.
- There were originally no **hash tables**, but they can be constructed as a **vector** (or **deque**) that holds lists (or even sets) as elements. C++11 has extended the STL to add hash tables in the form of unordered associative containers.

Notes

General Efficiency Tips:

- Wrap the STL so that functions can be specialized with custom error handling routines
- Use partial specialization (**void***) to avoid code bloat
- Evaluate different STL implementations to use the best one for one and use the latest version
- Use functors instead of callbacks
- Use pre-compiled headers and external header guards

Containers' Efficiency

- Pass containers by (const) reference, not by value!
- Use a custom container that doesn't call individual object destructors
- Use custom allocators
- Consider using a **vector** instead of a **list** for better cache coherency
- Remove and insert elements at the ends of container rather than in the middle

Notes:

- Treat non empty **vectors** as contiguous memory arrays
- Use **vector.reserve()** but be aware of **vector.capacity()** vs. **vector.size()**
- Use **std::vector<T>(c).swap(c)** to shrink vector to fit and free memory
- Use **map::insert** instead of **map::operator[]**
- **vector[i]** is faster than **vector::at(i)** because there is no error checking
- For small numbers of objects, **vector** in combination with **sort** can be faster than **map**
- Unordered associative containers (based on hashing) can sometimes be faster than ordered **map** containers
- forward_list singly-linked **list** containers can be faster and more space efficient than the standard list container (see Boost and C++11 for examples).
- Use **vector<bool>** or **bitset<N>** for compact bit list storage, but be aware of the limitations of **vector<bool>**

Notes:

Functors and Efficiency

- Use inline methods to avoid space used by empty objects
- Optimize contained object copy constructors and destructors
- Hand optimize string implementation so that string reference count is stored in string-allocated memory instead of in the string object

Iterator Efficiency

- Use pre-increment instead of post-increment when using STL iterators
- Use iterators instead of **[ ]** operator access for speed on vectors and deques
- Hoist **c.end()** out of **for()** loops when container end doesn't change
- Cache the iterator dereference in loops
- Understand the rules of iterator invalidation
- Pass iterators by value

Algorithm Usage

- Know what algorithms are available, both in STL and in Boost.
- Use **std::swap()** or **container.swap()** to swap objects and/or containers to simplify exception safety
- Use template specialization to optimize algorithms for common objects, e.g., **std::copy(char*, char*)**

Notes:

Notes:

Appendix B Extensions

ropes

A **rope** is a scalable string implementation: it is designed for efficient operations that involve the string as a whole. Operations such as assignment, concatenation, and substring take time that is nearly independent of the length of the string. Unlike C strings, **ropes** are a reasonable representation for very long strings such as edit buffers or mail messages. The rope class, along with crope (char rope) and wrope (wide char rope) are SGI extensions; they are not part of the C++ standard.

Advantages of rope

- Much faster concatenation and substring operations involving long strings; this time can be viewed as constant for most applications. It is reasonable to use a **rope** to represent a file inside a text editor.
- Potentially much better space usage. Minor modifications of a **rope** can share memory with the original. **Ropes** are allocated in small chunks, significantly reducing memory fragmentation problems introduced by large blocks.
- Assignment is a pointer assignment. Unlike "reference-counted copy-on-write" implementations, this remains largely true even if one of the copies is subsequently slightly modified. It is very inexpensive to checkpoint old versions of a string, *e.g.* in an edit history.
- One can view a function producing characters as a **rope**. Thus a piece of a **rope** may be a 100MByte file, which is read only when that section

Notes:

of the string is examined. Concatenating a string to the end of such a file does not involve reading the file.

Disadvantages of rope

- Single character replacements in a **rope** are expensive. A character update requires time roughly logarithmic in the length of the string. It is implemented as two substring operations followed by two concatenations.

- A **rope** can be examined a character at a time through a **const_iterator** in amortized constant time, as in **vector<char>**. However if the string is long this operation is slower for a **rope** by a significant constant factor even if little processing is done on each character. Non-const iterators involve additional checking and are hence a bit slower still.

- The size of their iterators is not small so, so copying them, though not tremendously expensive, is not a trivial operation.

Avoid post-incrementing iterators; use pre-increment whenever possible. Why? This is because post incrementing, when applied to a class type (and ropes are classes), always has to create a temporary object to effect the correct return value. This means that a copy constructor is invoked including memory allocation if the class contains pointer objects…this uses more memory and time than a pre-increment operator does.

Notes:

Heterogeneous Containers

Because STL was designed as a generic framework rather than an object-oriented one, it doesn't store objects of different types as elements in the same container – except for pairs. In general, a container that would allow this is called a **heterogeneous container**. Databases are often such containers. What if one want to create a media database and fill it with various types of music related files. One will use pointers in the container.

How does this work? To fill a heterogeneous container with pointers to different objects, simply pass the result of a **new** expression to the appropriate member function. For example:

```
void fill(vector < multimedia_file *> & v)
{
 v.push_back( new mp3_file ("hello world") ;
 v.push_back( new wav_file ("got_boring_mail") ;
);
```

The pointers inserted into v are of different types: one is of type mp3_file* and the second is of type wav_file*. One access the objects whose pointers are stored in the container as one would access an ordinary element. The only difference is that one use the -> notation.

Notes:

Smarter pointers
The STL offers several types of pointers that hold a ‘state’ , work with raw data, and clean up after themselves. But they don’t cover everything.

Boost’s library
The smart pointer library provides five smart pointer class templates. These templates are designed to complement the **std::auto_ptr** template.
They are examples of the "resource acquisition is initialization" idiom described in Bjarne Stroustrup's "The C++ Programming Language. This is what they offer:

scoped_ptr	Provides sole ownership of single objects & non-copyable.
scoped_array	Provides sole ownership of arrays & non-copyable
shared_ptr	The owned object can be shared with several pointers
shared_array	Array ownership shared with several pointers
weak_ptr	Non-owner pointer of an object shared_ptr owns
intrusive_ptr	Shared ownership of objects with reference count
unique_ptr	auto_ptr done right!

SourceForge’s ptr_vector
- Implemented as a wrapper for the **STL vector<T*>** that cuts one level of indirection for iterators and member functions.
- Lets one treat a **vector** of pointers as if it were a **vector** of values, iterators iterate over pointed-to objects, not pointers
- Iterators are **stable: ptr_vector** iterators remain valid when the **ptr_vector** container expands – unlike what happens in a **vector**

Notes:

- **ptr_vector** member functions **[ ]** operator, **at()**, **<front()** and **<back()** refer to the pointed to objects, not pointers

Appendix C Books

- Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Laboratories Technical Report 95-11(R.1), November 14, 1995. (Revised version of A. A. Stepanov and M. Lee: The Standard Template Library, Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.)

- Nicolai M. Josuttis: *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley. ISBN 0-201-37926-0.

- Scott Meyers: *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley. ISBN 0-201-74962-9.

- David R. Musser, Gillmer J. Derge, and Atul Saini: *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library (2nd Edition)* (Hardcover - Mar 27, 2001)

- David Vandevoorde and Nicolai M. Josuttis: *C++ Templates: The Complete Guide*. Addison-Wesley Professional 2002. ISBN 0-201-73484-2.

- Matthew H. Austern: *Generic Programming and the STL*, Reading, MA: Addison-Wesley, ISBN 0201309564

- Matthew Wilson: *Extending STL* – will be published by Addison-Wesley, in 2007.

Appendix D Websites

C++ Standard Template Library	http://cppreference.com/cppstl.html
More great STL websites	http://www.sgi.com/tech/stl, http://www.stlport.org/, http:///www.boost.org/.
Multiplatform C++ Standard Library	http://stlport.sourceforge.net/
C++ function templates	http://www.cplusplus.com/doc/tutorial/templates.html
C++ class templates	http://www.codersource.net/cpp_class_templates.html
Questions about C++ templates	http://www.comeaucomputing.com/techtalk/templates/#terms
C++ Standard Template Library	http://www.msoe.edu/eecs/ce/courseinfo/stl/
C++ STL tutorial	http://www.josuttis.com/libbook/idx.html
Excellent list of what is in STL	http://www.gotapi.com/index.html
Great place to learn about valarray	http://www.dinkumware.com/manuals/?manual=compleat&page=valarray.html
C++ tutorial	http://www.oopweb.com/CPP/Files/CPP.html
Proposed extensions to the STL	http://en.wikipedia.org/wiki/Technical_Report_1#Smart_Pointers
GNU allocator	http://gcc.gnu.org/onlinedocs/libstdc++/20_util/allocator.html
C++ links of all kinds	http://www-h.eng.cam.ac.uk/help/tpl/languages/C++.html
C++ tutorial	http://safari.oreilly.com/059600298X?tocview=true

Appendix E Exercises

Lab 1: Template for Fun (ctions)

In this exercise one will create a custom template container class, give it several template functions, and write a small program that instantiates the class, uses its functions and displays output.

1. Imagine one are going to need an array to hold some data of the same type, but one may need it for integers today, double or float data tomorrow and chars another day. So one decide to create a template class, whose datatype can be defined later; that is, when a program uses the class rather than defined now. Create several constructors, including one that takes only a datatype, a default constructor, and one that takes both a datatype and an initial size.

2. In addition, one want the collection to be able to grow as data is added to it and shrink if elements are removed. Plus one want it provide fast access and be memory efficient. Allow it to grow and shrink at both ends and be accessible at any element. So create template functions to add, remove and modify member data. Also write a function to tell how many elements the container object has.

3. Then write a program that creates 2 of these objects, adds at least 4 elements to each container, and performs at least 3 member functions on each of them. Finally print out (display) results along with sensible messages telling the user what was done.

Lab 2: Now a vector into space!

In this exercise one will create two vectors, of the same datatype, add data to each of them, and then swap them using the vector swap function. One will write a small program that instantiates the vectors, uses their functions and displays output.

Lab 3: Clear the deck!

Now we are going to use a predefined STL template sequential container class, deque. These are doubly-linked lists and are efficient at adding and removing elements at both ends of the collection, but are less efficient in adding or removing in other positions of the collection. We have also talked a little about iterators, which are defined with each STL class. Notice that the deque has a bidirectional iterator defined.

Create a program that instantiates a deque and an iterator. Use a deque member function to show us the deque is empty. Now using the iterator, add at least 10 values to the deque - one can add them to either end – use push_back() or push_front() member functions. Next, display the first and last values in the deque along with messages telling us what values these are (in other words, the first element etc). Now show us that the deque is no longer empty using a member function.

After this add 3 members at the third, seventh and ninth positions and display the entire deque with messages telling us where one added the new members. Access element 7 directly and display its value along with an appropriate message telling us what element it is.

Now try to access an out of range element, but handle it in a try/catch block since the STL does not do error checking. Finally, use a member function to clear out the deque of all elements and show that it is empty, and display a message telling us that “now the deck is empty.”

Lab 4: Set the Table

This exercise gives one experience using associative containers. The set or multiset containers allow one to access elements by key. Here the order one added the elements to the container does not determine their position like it does in sequential containers. So one are going to add some char data to a set and then print out how it is sorted.

So create a set called Letters to store a collection of chars, along with an iterator. Prove that the set is empty using a member function and displaying its results.

Now add at least 10 letters including M, but not including the letter S. Add them in non-alphabetic order so one can see that the container will 'order' the values for one. Display the set's contents using its iterator to walk through the set and show the set's size using the member function.

Next find and remove the letter M, showing us the resulting set afterward. Then insert the letter S using a member function and display the set again.

Now insert the values '1' – '5' as chars into the set. Display the set in a table form, each value on its own line along with a message after the char displayed…For example: A Apples are good.

Lab 5: A Stack of dirty dishes

This exercise will use a container adaptor: we want to adapt a deque into a stack.

Create a deque and fill it with 12 integers using the rule 2y + y where y is the loop index and y starts at 5. Check that one have values in the deque now by displaying its values in order.

Next use this deque to create a stack, using the stack constructor that takes an existing container and creates a stack from it. Display the stack using its member functions. What is its size? Is it empty? Add 5 more elements to the stack. Display its size.

Now take off 7 elements, displaying the stack after each one is removed. What is the size now? Can one access the element being removed from the pop() member function?

Lab 6: Nice Functions if one can Get Them

In this exercise one will first create a custom functor to see why this concept is useful, and then one will use it with a few predefined STL functors. One will need to define operator () for the class so it can act as a function.

Create a functor to test whether a value passed to it is within a range. This functor must be constructed with two values that are the range to be used later for testing. Then later when the functor is passed a value it checks whether that value is within its range and returns either true or false. Here is a sample function one can use for operator().

```
bool operator()(const T& t)
{   return ! (t < beginRange) && t < endRange;  }
```

Now create a program to use the custom functor. In this program, first create a vector and fill it up with 15 values by using some simple method such as a loop that does a calculation using the rand function or in some other way.

```
srand ( time(NULL) );
int getRandom = rand() % 10 + 1;
```

Then use the functor to check if the values fall within the range. Display output telling what the program is doing.

Lab 7: Find this if you can

In this exercise we will use several simple algorithms.

The find algorithm can be applied to many containers. It searches all the elements or a sub-range, looking for an element that is "equal to" a specified value; the equality operator (==) must be defined for the type of the container's elements.

The sort algorithm will put the elements of a container in ascending order using the < operator.

For this exercise, create a vector of ‘double’ data and fill it up with at least 10 members NOT in order. Display its contents and show it is not empty.

Then use sort to order it. Now display what the collection looks like in order.

Next, select a value, and use the find algorithm to find it in the vector.

Lab 8: Fun with auto_ptr

The <memory> header file contains a smart pointer class, auto_ptr. This is a useful class help stop memory leaks due to class destructors not being called or not cleaning up memory correctly. It works with only single values or objects; not with arrays.

To declare an auto_ptr object, one will give in a datatype being pointed to. Now one don't need to call delete; when the auto_ptr myintPtr goes out of scope, it deletes itself. And if one assign it to another auto_ptr, it returns to null instead of hanging around still pointing to that value or object it was pointing to. And one can dereference it just like any pointer:

```
auto_ptr<int> myIntPtr(new int);
*myIntPtr = 10;
```

One can also use a class with auto_ptr. Let's say I create a Circle class, which has data members x and y to represent its origin's coordinates:

```
auto_ptr<Circle> myCirclePtr(new Circle);
Circle -> origin.x = 2.0;
Circle -> origin.y = 3.0;
```

Create a simple class and give it at least 2 data members. Then create a program using the class and work with it via an auto_ptr. Create a class using an auto_ptr, give the class' data members values using the auto_ptr, and then display what is going on.