Home
Titanic
Free Software
C Course
C++ STL course
Contents
Introduction
Romans
Function Templates
Class Templates
More Romans
Vector
Deque
List
Allocators
Sequence Adapters
Stack
Queue
Priority Queue
Strings
Iterators
Iterator Adapters
Holy Grail
Map
Set
Algorithms
Algorithms
Functions
STL Links
Bibliography
VMS
Motif
Marian's Page
Links
E-Mail

Phil Ottewell's STL Tutorial

Version 1.2 © Phil Ottewell

Table of Contents

Warning for the humour-impaired: Any strange references that you can't understand are almost certainly a skit on Monty Python
These notes formed part of an internal course on the STL which I was asked to give to my colleagues a

1. Introduction

The C++ Standard Template Library, generally referred to as the *STL*, saves you, the p
to re-invent the wheel. This course is aimed at programmers who have reasonable fam
programming language, and know about classes, constructors and the like. Avoiding e
examples have been written to clearly demonstrate STL features. The sample program
Library (as distinct from Standard Template Library) features like `fstream` and `iostr`
of how to use these. A discussion of the Standard Library as a whole is beyond the sco
Stroustrup and others in the bibliography for more information.

What motivated people to write the STL ? Many people felt that C++ classes were ina
requiring containers for user defined types, and methods for common operations on th
might need self-expanding arrays, which can easily be searched, sorted, added to or re

March 2001

messing about with memory reallocation and management. Other Object-Oriented lan implement this sort of thing, and hence they were incorporated into C++.

Driving forces behind the STL include Alexander Stepanov and Meng Lee at Hewlett- California, Dave Musser at General Electric's Research Center in Schenectady, New Y and of course "Mr C++" himself, Bjarne Stroustrup at AT&T Bell Laboratories.

The example programs are known to work on Alpha/VAX VMS 6.2 onwards using D Windows NT 4.0 SP3 with Visual C++ 5.0 , and Windows 2000 with Visual C++ 6.0 `#pragmas` have been guarded with `#ifdef _VMS` or `#ifdef _WIN32`. To build under V command file. Just give a program name like `example_1_1` as its argument and it will extension `.CXX, .CPP, .C` , in that order. If you provide the extension then it uses th files get an `_ALPHA` suffix. Here is an example:

```
$ @MAKE EXAMPLE_1_1 ! On an Alpha
  DEV$DISK:[PHIL.WWW.STL]
CC/PREFIX=ALL EXAMPLE_1_1.C  -> EXAMPLE_1_1.OBJ_ALPHA
LINK EXAMPLE_1_1  -> EXAMPLE_1_1.EXE_ALPHA

$ @MAKE EXAMPLE_1_2 ! Now on a VAX
  DEV$DISK:[PHIL.WWW.STL]
CXX/ASSUME=(NOHEADER_TYPE_DEFAULT)/EXCEPTIONS/TEMPLATE_DEFINE=(LOCAL)
  EXAMPLE_1_2.CXX  -> EXAMPLE_1_2.OBJ
CXXLINK EXAMPLE_1_2  -> EXAMPLE_1_2.EXE
```

A slight buglet introduced in DEC C++ 5.6 for Alpha VMS means that you might get CXXLINK step.

```
%LINK-W-NUDFSYMS, 1 undefined symbol:
%LINK-I-UDFSYM,         WHAT__K9BAD_ALLOCXV
%LINK-W-USEUNDEF, undefined symbol WHAT__K9BAD_ALLOCXV referenced
        in psect __VTBL_9BAD_ALLOC offset %X00000004
        in module MEMORY file SYS$COMMON:[SYSLIB]LIBCXXSTD.OLB;1
```

The undefined symbol is harmless and never referenced, but you can obtain the officia ftp.service.digital.com. Download and run `cxxae01056.a-dcx_axpexe` to unpack it, t `SYSTEM` use `@SYS$UPDATE:VMSINSTAL` to install it.

Download individual sample programs using **Right Click** and "Save" on their links, o examples in a .zip file. For Windows NT or Windows 2000, the Developer Studio files distribution.

The first two programs are implementations of expanding, integer arrays which are so Example 1.1 is in ANSI C, and 1.2 is C++using the STL. I have tried to make the C pr possible by using `typedef`, but this is still not really adequate, as we will see.

Right Click & save `example_1_1.c`

```
/*
   Phil Ottewell's STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht

   Example 1.1                 © Phil Ottewell 1997 <phil@yrl.co.uk>

   Purpose:
            Simple vector and sort demonstration using ANSI C
*/
```

```
/* ANSI C Headers */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Typedef array type in case we need to change it */
typedef int array_type;

/* Function Prototypes */
int compare_values( const void *a, const void *b );
int *get_array_space( int num_items );

int main( int argc, char *argv[] )
{
    int i;
    int nitems = 0;
    array_type ival;
    array_type *v;

    fprintf(stdout,"Enter integers, <Return> after each, <Ctrl>Z to f

    while( EOF != fscanf( stdin, "%d", &ival) ) {
      v = get_array_space( nitems+1 );
      v[nitems] = ival;
      fprintf( stdout, "%6d: %d\n", nitems, v[nitems] );
      ++nitems;
    }

    if ( nitems ) {
      qsort( v, nitems, sizeof(array_type), compare_values );
      for ( i = 0; i < nitems; ++i )
        fprintf( stdout, "%d ", v[i] );
      fprintf( stdout, "\n" );
    }

    return( EXIT_SUCCESS );
}

/*---- Comparison func returns: -ve if a < b,  0 if a == b,  +ve if a
int compare_values( const void *a, const void *b )
{
    const array_type *first, *second;
/*  End of declarations ... */
    first = (array_type *)a;
    second = (array_type *)b;
    return( *first - *second );
}

/*---- Allocate space: n == 0 return pointer, n > 0 expand/realloc if
int *get_array_space( int n )
{
    const int extra_space = 2;
    array_type *new_space_ptr;
    static array_type *array_space_ptr;
    static int mxitm;
/*  End of declarations ... */

    if ( n > 0 ) {
      if ( n > mxitm ) {
        n += extra_space;
        if ( array_space_ptr ) {
          new_space_ptr = realloc(array_space_ptr,sizeof(array_type)*
          if ( new_space_ptr ) {
/*          Successfully expanded the space */
            array_space_ptr = new_space_ptr;
/*          Clear new storage space */
            memset( &array_space_ptr[mxitm], 0, sizeof(array_type)*(n
          } else {
/*          Couldn’t allocate the space */
            exit( EXIT_FAILURE );
          }
        } else {
          array_space_ptr = (array_type *)calloc( n, sizeof(array_typ
          if ( !array_space_ptr ) {
/*          Couldn’t allocate the space */
            exit( EXIT_FAILURE );
          }
        }
        mxitm = n;
      }
    }
    return( array_space_ptr );
}
```

In the this program (see Phil’s C Course for an introduction to C) I used a `typedef` fo store and sort, and a `get_array_space` function to allocate and/or expand the availabl basic error handling `get_array_space` is rather ungainly. It handles different types o `typedef`, but if I wanted more than one type of data storing, or even more than one bu would have to write a unique `get_array_space_`*`type`* function for each. The `compar` also have to be rewritten, though this is also the case in the C++ code, for user defined values.

Right Click & save `example_1_2.cxx`

```
// Phil Ottewell’s STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 1.2                  © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//          Simple vector and sort demonstration using the STL

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <algorithm>
#include <iostream>
#include <vector>

#ifdef _WIN32
using namespace std;
#endif

int main( int argc, char *argv[] )
{
    int ival, nitems = 0;
    vector<int> v;

    cout << "Enter integers, <Return> after each, <Ctrl>Z to finish:"

    while( cin >> ival, cin.good() ) {
      v.push_back( ival );
      cout.width(6);
      cout << nitems << ": " << v[nitems++] << endl;
    }
    if ( nitems ) {
      sort( v.begin(), v.end() );
      for (vector<int>::const_iterator viter=v.begin(); viter!=v.end(
        cout << *viter << " ";
      cout << endl;
    }

    return( EXIT_SUCCESS );
}
```

Contrast the C++ program, *Example 1.2*, with the previous code. Using the STL, we ir template class, which allows us to store any data type we like in what is essentially a c self-expanding, random access array.

2. Templates ite Domum

This is (incorrect) Latin for "Templates Go Home !" and represents the ambivalence tl programmers feel towards this language feature. I hope that you will be persuaded of i this section.

C++ supports a number of OOP (Object Oriented Programming) concepts. Broadly sp *encapsulation* through the member functions and `private` or `protected` data membe allowing classes to be derived from other classes and abstract base classes, and *polym* functions, function signatures and templates. Templates achieve polymorphism by allc or functions in a generic way, and let the compiler/linker generate an *instantiation* of t the actual types we require.

The STL is built, as its name suggests, on the C++ *template* feature. There are two typ *templates* and *class templates*. Both perform similar roles in that they allow functions in a generic form, enabling the function or class to be generated for any data type - use

At first sight this might not appear to be very different from macros in the C language. Example 1.1 we could have made it more flexible by using a macro to define the comp

```
#define COMPARE_VALUES( value_type ) \
value_type compare_values_##value_type( const void *a, const void *b
{const value_type *first, *second; \
 first = (value_type *)a; second = (value_type *)b; return( *first -

COMPARE_VALUES( float )  /* Generate function for floats, */
COMPARE_VALUES( double ) /* doubles and */
COMPARE_VALUES( int )    /* ints */
        .
/*    Pick comparison function */
      qsort( v, nitems, sizeof(array_type), compare_values_int );
```

The same method can be used for structure generation. There are a number of drawbac have to explicitly generate functions for all the types you want to use, and there is no t particular case, so you could easily pass `compare_values_float` when you meant to would have to be rigorous about your naming convention. In addition, some people wc not as transparent, since you can’t see what they expand into until compilation time.

Templates avoid these problems. Because they are built into the language, they are abl safety checking and deduce the types of their arguments automatically, generating the arguments are used. C++ allows you to overload operators like < for user-defined type

definition often suffices for built-in and user-defined classes. The following two sectic
use of template functions and classes.

Function Templates

Template functions have the following form:

```
template < template-argument-list >
function-definition
```

The *template-argument-list* is one or more type-names within the scope of the templat
functions the first argument is *always* a type, as in this code fragment.

```
template <class T>
T mymin( T v1, T v2)
{
  return( (v1 < v2) ? v1 : v2 );
}
```

You should be able to use the `typename` keyword in your function (or class) declaratic

```
// This may well give an error but is perfectly legal
template <typename T>
T mymin( T v1, T v2)
{
  return( (v1 < v2) ? v1 : v2 );
}
```

Stroustrup favours the `class T` format because it means fewer keystrokes. Personally
`typename T` form, but won't use it because it will give errors with some compilers.

Those of use who started programming using proper languages like Fortran are used t
selecting the correct function variant :-) Not many people using the Fortran `MAX` functi
`IMAX0,JMAX0,KMAX0` and so on. The compiler selects the specific function according t
Remember that the `class T` type doesn't *have* to be a class. It can be a built-in type li
compiler always tries to find a "real" function with matching arguments and return typ
function from the template, as in the following program.

Right Click & save `example_2_1.cxx`

```
// Phil Ottewell's STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 2.1                  © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//         Demonstrate simple function template

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <iostream>
#include <string>

#ifdef _WIN32
using namespace std;
#endif

// Template function
```

```
template <class T>
T mymin( T v1, T v2)
{
   return( (v1 < v2) ? v1 : v2 );
}

// "Real" function
double mymin( double v1, double v2)
{
// Here be found Porkies !!
   return( (v1 > v2) ? v1 : v2 );
//     ^
//     |
//     Wrong sign just to show which function is being called
}

int main( int argc, char *argv[] )
{
    string a("yo"), b("boys"), smin;
    int i = 123, j = 456, imin;
    double x = 3.1415926535898, y = 1.6180339887499, fmin;
//  End of declarations ...

    imin = mymin( i, j );
    cout << "Minimum of " << i << " and " << j << " is " << imin << e

    smin = mymin( a, b );
    cout << "Minimum of " << a << " and " << b << " is " << smin << e

    fmin = mymin( x, y );
    cout << "$ SET PORKY/ON" << endl;
    cout << "Wrong answer if \"real\" mymin called instead of templat
    cout << "Minimum of " << x << " and " << y << " is " << fmin << e

    return( EXIT_SUCCESS );
}
```

The "real" function signature matched the float case, and was used in preference to the `namespace std` line is necessary on Windows if we wish to avoid prefixing STL featu `std::cin`. It is not necessary with VMS and DEC C++ 5.6, though it may be with futu

Class Templates

Template classes have the following form:

```
template < template-argument-list >
class-definition
```

The *template-argument-list* is one or more type-name within the scope of the template

Right Click & save `example_2_2.cxx`

```
// Phil Ottewell’s STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 2.2                  © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//          Demonstrate simple class template

// ANSI C Headers
#include <stdlib.h>
```

```
// C++ STL Headers
#include <iostream>

template <class T, int size>
class MyVector
{
  public:
    MyVector() { obj_list = new T[ size ]; nused = 0; max_size = size
    ~MyVector() { delete [] obj_list; }

    void Append( const T &new_T ) { if ( nused < max_size )
                                      obj_list[nused++] = new_T; }

    T &operator[]( int ndx )
    {
       if ( ndx < 0 && ndx >= nused ) {
         throw("up"); // barf on error
       }
       return( obj_list[ndx] );
    }

  private:
    int max_size;
    int nused;
    T* obj_list;
};

#ifdef _WIN32
using namespace std;
#endif

int main( int argc, char *argv[] )
{
    int i;
    const int max_elements = 10;
    MyVector< int, max_elements > phils_list;
//  End of declarations ...

//  Populate the list
    for ( i = 0; i < max_elements; i++)
      phils_list.Append( i );

//  Print out the list
    for ( i = 0; i < max_elements; i++)
      cout << phils_list[i] << " ";

    cout << endl;

    return( EXIT_SUCCESS );
}
```

3. What has the STL ever done for us ?

"Well, yes, vectors, I mean obviously the vectors are good ..."
"Don't forget queues Reg, I mean, where would we be without properly organized queues and iterators
murmurs of agreement
"Yes, alright, apart from vectors and queues ..."
"Sorts Reg - I hated having to code up a new sort routine for every class." *Here, here, etc.*
"Right. So apart from vectors, queues and associated containers classes, iterators, various useful algorit
the STL ever done for us ?"
"Memory allocators Reg. We can allocate container memory using any scheme we like, and change it v
keeps things in order" *Reg loses his temper*

"Order ? Order ? Oh shut up!"

At the end of this section, the waffle above should start making sense to you, but is un
humorous as a result of your studies.

There are three types of *sequence containers* in the STL. These, as their name suggest
sequence. They are the `vector`, `deque` and `list`:

- vector<Type>
- deque<Type>
- list<Type>

To choose a container, decide what sort of operations you will most frequently perforr
the following table to help you.

Operation	Vector	Deque	List
Access 1st Element	Constant	Constant	Constant
Access last Element	Constant	Constant	Constant
Access "random" element	Constant	Constant	Linear
Add/Delete at Beginning	Linear	Constant	Constant
Add/Delete at End	Constant	Constant	Constant
Add/Delete at "random"	Linear	Linear	Constant

Time overhead of operations on sequence containers

Each container has attributes suited to particular applications. The subsections and coc
further clarify when and how to use each type of sequence container.

Throughout this tutorial, I have given the `#include` file needed to use a feature immec
heading. Note that some of the header names have changed since earlier versions of th
has been dropped. Older books may refer to, for example, `<algo.h>`, which you shoul
`<algorithm>` . If you include ANSI C headers, they *should* have the .h, e.g. `<stdlib.`
ANSI C headers, prefixed by the letter "c" and minus the .h are becoming more widel
implementations currently support them, e.g. `<cstdlib>`.

On OpenVMS systems a reference copy of the source code for the STL can be found i
`SYS$COMMON:[CXX$LIB.REFERENCE.CXXL$ANSI_DEF]` . So for <vector> look in there
For Windows, go into Visual Studio, click on the "binocular search" button on the too
"Index" tab, type `vector header file` (replace `vector` with your choice if header fi
`<Return>` , then click on the entry in the "Select topic to display" list at the bottom.

Vector

```
#include <vector>
```

We introduced the `vector` in Example 1.2, where we used it instead of an array. The v an array, and allows array-type syntax, e.g. my_vector[2] . A vector is able to acces (referred to as "random" access in the preceding table) with a constant time overhead, deletion at the *end* of a vector is "cheap". As with the string, **no bounds checking** i use operator [].

Insertions and deletions anywhere other than at the end of the vector incur overhead of elements in the vector, because all the following entries have to be shuffled along t entries, the storage being contiguous. Memory overhead of a vector is very low and array.

The table below shows some of the main vector functions.

Some Vector Access Functions	Purpose
begin()	Returns iterator pointing to first
end()	Returns iterator pointing _after_ l
push_back(...)	Add element to end of vector
pop_back(...)	Destroy element at end of vector
swap(,)	Swap two elements
insert(,)	Insert new element
size()	Number of elements in vector
capacity()	Element capacity before more memory
empty()	True if vector is empty
[]	Random access operator

The next example shows a vector in use.

Right Click & save example_3_1.cxx

```
// Phil Ottewell’s STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 3.1                  © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//          Demonstrate use of a vector

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <algorithm>
#include <iostream>
#include <vector>

#ifdef _WIN32
using namespace std;
#endif

int main( int argc, char *argv[] )
{
    int nitems = 0;
    int ival;
    vector<int> v;

    cout << "Enter integers, <Return> after each, <Ctrl>Z to finish:"

    while( cin >> ival, cin.good() ) {
      v.push_back( ival );
      cout.width(6);
      cout << nitems << ": " << v[nitems++] << endl;
    }
    if ( nitems ) {
      sort( v.begin(), v.end() );
      for (vector<int>::const_iterator viter=v.begin(); viter!=v.end(
        cout << *viter << " ";
      cout << endl;
    }

    return( EXIT_SUCCESS );
}
```

Note how the element sort takes `v.begin()` and `v.end()` as range arguments. This is and you will meet it again. The STL provides specialized variants of vectors: the `bits` former allows a degree of array-like addressing for individual bits, and the latter is int with real or integer quantities. To use them, include the `<bitset>` or `<valarray>` hea always supported in current STL implementations). Be careful if you `erase()` or `inse` middle of a `vector`. This can invalidate *all existing iterators*. To erase all elements in `clear()` member function.

Deque

`#include <deque>`

The double-ended queue, `deque` (pronounced "deck") has similar properties to a `vect` suggests you can efficiently insert or delete elements at *either end*.

The table shows some of the main `deque` functions.

Some Deque Access Functions	Purpose
begin()	Returns iterator pointing to first e
end()	Returns iterator pointing _after_ la
push_front(...)	Add element to front of deque
pop_front(...)	Destroy element at front of deque
push_back(...)	Add element to end of deque
pop_back(...)	Destroy element at end of deque
swap(,)	Swap two elements
insert(,)	Insert new element
size()	Number of elements in deque
capacity()	Element capacity before more memory
empty()	True if deque is empty
[]	Random access operator

A `deque`, like a `vector`, is not very good at inserting or deleting elements at random p random access to elements using the array-like [] syntax, though not as efficiently as a `vector` an `erase()` or `insert()` in the middle can invalidate *all existing iterators*.

The following program shows a `deque` representing a deck of cards. The queue is dou modify it to cheat and deal off the bottom :-)

Right Click & save `example_3_2.cxx`

```
// Phil Ottewell's STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 3.2                  © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//          Demonstrate deque sequence container with a dumb card gam
//          which is a bit like pontoon/blackjack/vingt-et-un
//          Note sneaky use of random_shuffle() sequence modifying ag

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <algorithm>
#include <deque>
#include <iostream>

#ifdef _WIN32
using namespace std;
#endif

class Card
{
  public:
    Card() { Card(1,1); }
    Card( int s, int c ) { suit = s; card = c; }
    friend ostream & operator<<( ostream &os, const Card &card );
    int value() { return( card ); }
  private:
    int suit, card;
};

ostream & operator<<( ostream &os, const Card &card )
{
   static const char *suitname[] = { "Hearts", "Clubs", "Diamonds", "
   static const char *cardname[] = { "Ace", "2", "3", "4", "5", "6",
                                     "8", "9", "10", "Jack", "Queen",
   return( os << cardname[card.card-1] << " of " << suitname[card.sui
}

class Deck
{
  public:
    Deck() { newpack(); };
    void newpack() {
      for ( int i = 0; i < 4; ++i ) {
        for ( int j = 1; j <= 13; ++j ) cards.push_back( Card( i, j )
      }
    }
//  shuffle() uses the STL sequence modifying algorithm, random_shuff
    void shuffle() { random_shuffle( cards.begin(), cards.end() ); }
    bool empty() const { return( cards.empty() ); }
    Card twist() { Card next = cards.front(); cards.pop_front(); retu
  private:
    deque< Card > cards;
};

int main( int argc, char *argv[] )
{
    Deck deck;
    Card card;
    int total, bank_total;
    char ch;
//  End of declarations ...

    while ( 1 ) {
      cout << "\n\n ---- New deck ----" << endl;
      total = bank_total = 0;
      deck.shuffle();
      ch = 'T';
      while ( 1 ) {
        if ( total > 0 && total != 21 ) {
          cout << "Twist or Stick ? ";
          cin >> ch;
          if ( !cin.good() ) cin.clear(); // Catch Ctrl-Z
          ch = toupper( ch );
        } else {
          if ( total == 21 ) ch = 'S'; // Stick at 21
        }

        if ( ch == 'Y' || ch == 'T' ) {
          card = deck.twist();
          total += card.value();
          cout << card << " makes a total of " << total << endl;
          if ( total > 21 ) {
            cout << "Bust !" << endl;
            break;
          }
        } else {
          cout << "You stuck at " << total << "\n"
               << "Bank tries to beat you" << endl;

          while ( bank_total < total ) {
            if ( !deck.empty() ) {
              card = deck.twist();
              bank_total += card.value();
              cout << card << " makes bank's total " << bank_total <<
              if ( bank_total > 21 ) {
                cout << "Bank has bust - You win !" << endl;
                break;
              } else if ( bank_total >= total ) {
                cout << "Bank has won !" << endl;
                break;
              }
            }
          }
          break;
        }
      }

      cout << "New game [Y/N] ? ";
      cin >> ch;
      if ( !cin.good() ) cin.clear(); // Catch Ctrl-Z
      ch = toupper( ch );

      if ( ch != 'Y' && ch != 'T' ) break;
      deck.newpack();
    }

    return( EXIT_SUCCESS );
}
```

The card game is a version of pontoon, the idea being to get as close to 21 as possible.
picture cards as 10. Try to modify the program to do smart addition and count aces as
store your "hand" and give alternative totals.

Notice the check on the state of the input stream after reading in the character respons
if you hit, say, `<Ctrl>Z`, the input stream will be in an error state and the next read wil
causing a loop if you don't clear `cin` to a good state.

List

```
#include <list>
```

Lists don’t provide [] random access like an array or `vector`, but are suited to applicat add or remove elements to or from the *middle*. They are implemented as double linkec support bidirectional iterators, and are the most memory-hungry standard container, ve In compensation, lists allow low-cost growth at either end or in the middle.

Here are some of the main `list` functions.

Some List Access Functions	Purpose
-------------------------	------
begin()	Returns iterator pointing to first el
end()	Returns iterator pointing _after_ las
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
capacity()	Element capacity before more memory n
empty()	True if list is empty
sort()	Specific function because <algorithm> sort routines expect random access it

Right Click & save `example_3_3.cxx`

```
// Phil Ottewell’s STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 3.3                  © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//         Demonstrate list container

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <algorithm>
#include <iostream>
#include <list>
#include <string>

#ifdef _WIN32
using namespace std;
# pragma warning(disable:4786) // We know basic_string generates long
#endif

int main( int argc, char *argv[] )
{
    string things[] = { "JAF", "ROB", "PHIL", "ELLIOTT", "ANDRZEJ" };
    const int N = sizeof(things)/sizeof(things[0]);
    list< string > yrl;
    list< string >::iterator iter;

    for ( int i = 0; i < N; ++i) yrl.push_back( things[i] );
    for ( iter = yrl.begin(); iter != yrl.end(); ++iter ) cout << *it

//  Find "ELLIOTT"
    cout << "\nNow look for ELLIOTT" << endl;
    iter = find( yrl.begin(), yrl.end(), "ELLIOTT" );
//  Mary should be ahead of Elliott
    if ( iter != yrl.end() ) {
      cout << "\nInsert MARY before ELLIOTT" << endl;
      yrl.insert( iter, "MARY" );
    } else {
      cout << "\nCouldn't find ELLIOTT" << endl;
    }
    for ( iter = yrl.begin(); iter != yrl.end(); ++iter ) cout << *it

    return( EXIT_SUCCESS );
}
```

The loop over elements starts at `yrl.begin()` and ends *just before* `yrl.end()`. The S return iterators pointing *just past the last element*, so loops should do a `!=` test and not most likely invalid, position. Take care not to reuse (e.g. ++) iterators after they have b they will be invalid. Other iterators, however, are still valid after `erase()` or `insert(`

Container Caveats

Be aware that copy constructors and copy assignment are used when elements are add the `vector` and `deque`) deleted from containers, respectively. To refresh your memori copy assignment member functions look like this example:

```
class MyClass {
public:
  .
  // Copy constructor
  MyClass( const MyClass &mc )
  {
    // Initialize new object by copying mc.
    // If you have *this = mc , you'll call the copy assignment funct
  }

  // Copy assignment
  MyClass & operator =( const MyClass &mcRHS )
  {
    // Avoid self-assignment
    if ( this != &mcRHS ) {
      // Be careful not to do *this = mcRHS or you'll loop
      .
    }
    return( *this );
  }
};
```

When you put an object in a container, the copy constructor will be called. If you eras destructors and copy assignments (if other elements need to be shuffled down) will be example, `RefCount.cxx` for a demonstration of this.

Another point to bear in mind is that, if you know in advance how many elements you container, you can `reserve()` space, avoiding the need for the STL to reallocate or m

```
  vector<MyClass> things;
  things.reserve( 30000 );
  for ( ... ) {
    things.push_back( nextThing );
  }
```

The above code fragment reserves enough space for 30000 objects up front, and produc
in the program.

Allocators

Allocators do exactly what it says on the can. They allocate raw memory, and return it
destroy objects. Allocators are very "low level" features in the STL, and are designed
allocation and deallocation. This allows for efficient storage by use of different schem
classes. The default allocator, `alloc`, is thread-safe and has good performance charact
is best to regard allocators as a "black box", partly because their implementation is stil
also because the defaults work well for most applications. Leave well alone !

4. Sequence Adapters

Sequence container adapters are used to change the "user interface" to other STL sequ
written containers if they satisfy the access function requirements. Why might you wa
wanted to implement a stack of items, you might at first decide to base your stack clas
let's call it `ListStack` - and define public member functions for `push(), pop(), em`
However, you might later decide that another container like a `vector` might be better
would then have to define a new stack class, with the same public interface, but based
`VectorStack`, so that other programmers could choose a `list` or a `vector` based queu
number of names for what is essentially the same thing start to mushroom. In addition
the programmer using his or her own underlying class as the container.

Container adapters neatly solve this by presenting the same public interface irrespectiv
container. Being templatized, they avoid name proliferation. Provided the container ty
operations required by the adapter class (see the individual sections below) you can us
underlying implementation. It is important to note that the adapters provide a restricte
underlying container, and you *cannot* use `iterators` with adapters.

Stack

```
#include <stack>
```

The `stack` implements a Last In First Out, or LIFO structure, which provide the publi
`pop(), empty()` and `top()`. Again, these are self explanatory - empty returns a `bool`
`stack` is empty. To support this functionality `stack` expects the underlying container
`pop_back(), empty() or size()` and `back()`

```
  Container Function    Stack Adapter Function
  ------------------    ----------------------
  back()                top()
  push_back()           push()
  pop_back()            pop()
  empty()               empty()
  size()                size()
```

You would be correct in surmising that you can use `vector`, `deque` or `list` as the und
you wanted a user written type as the container, then if provided the necessary public
"plug" it into a container adapter.

Example 4.1 demonstrates a stack implemented with a `vector` of pointers to char. No
container adapters differs from that shown in Saini and Musser or Nelson’s book, and
1996 Working Paper of the ANSIC++ Draft Standard.

Right Click & save `example_4_1.cxx`

```
// Phil Ottewell’s STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 4.1                 © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//          Demonstrate use of stack container adaptor

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <iostream>
#include <vector>
#include <stack>

#ifdef _WIN32
using namespace std;
#endif

int main( int argc, char *argv[] )
{
    stack< const char *, vector<const char *> > s;

//  Push on stack in reverse order
    s.push("order");
    s.push("correct"); // Oh no it isn’t !
    s.push("the");
    s.push("in");
    s.push("is");
    s.push("This");

//  Pop off stack which reverses the push() order
    while ( !s.empty() ) {
      cout  << s.top() << " "; s.pop(); /// Oh yes it is !
    }
    cout << endl;

    return( EXIT_SUCCESS );
}
```

Note how the `stack` declaration uses two arguments. The first is the type of object sto
container of the same type of object.

Queue

`#include <queue>`

A `queue` implements a First In First Out, or FIFO structure, which provides the public
`pop(), empty(), back()` and `front()` (`empty()` returns a `bool` value which is tru
To support these, `queue` expects the underlying container to have `push_back(), pop`
`size()` and `back()`

Container Function	Queue Adapter Function

```
  -----------------     --------------------
  front()               front()
  back()                back()
  push_back()           push()
  pop_front()           pop()
  empty()               empty()
  size()                size()
```

You can use `deque` or `list` as the underlying container type, or a user-written type. Y because `vector` doesn’t support `pop_front()`. You could write a `pop_front()` funct would be inefficient because removing the first element would require a potentially la the other elements, taking time **O**(N).

The following code shows how to use a `queue`.

Right Click & save `example_4_2.cxx`

```
// Phil Ottewell’s STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 4.2                 © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//         Demonstrate use of queue container adaptor

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <iostream>
#include <queue>

#ifdef _WIN32
using namespace std;
#endif

int main( int argc, char *argv[] )
{
    queue< const char * > s;

//  Push on stack in correct order
    s.push("First");
    s.push("come");
    s.push("first");
    s.push("served");
    s.push("- why");
    s.push("don’t");
    s.push("bars");
    s.push("do");
    s.push("this ?");


//  Pop off front of queue which preserves the order
    while ( !s.empty() ) {
      cout  << s.front() << " "; s.pop();
    }
    cout << endl;

    return( EXIT_SUCCESS );
}
```

Note how we haven’t given a second argument in the `queue` declaration, but used the header file.

Priority Queue

```
#include <queue>
```

A `priority_queue`, defined in the `<queue>` header, is similar to a `queue`, with the ad
ordering the objects according to a user-defined priority. The order of objects with equ
predictable, except of course, they will be grouped together. This might be required by
process scheduler, or batch queue manager. The underlying container has to support p
`pop_back(), empty(), front()`, plus a random access iterator and comparison fun
order.

Container Function	Priority Queue Adapter Function
front()	top()
push_back()	push()
pop_back()	pop()
empty()	empty()
size()	size()
[] random iterators	Required to support heap ordering operations

Hence a `vector` or a `deque` can be used as the underlying container, or a suitable user

The next sample program demonstrates a `priority_queue` implemented with a `vect`
that the syntax of using container adapters differs from that shown in Saini and Musse

Right Click & save `example_4_3.cxx`

```
// Phil Ottewell's STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 4.3                     © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//         Demonstrate use of priority_queue container adaptor
//         by using a task/priority structure
//

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <functional>
#include <iostream>
#include <queue>
#include <string>
#include <vector>

#ifdef _WIN32
using namespace std;
#endif

class TaskObject {
public:
    friend class PrioritizeTasks;
    friend ostream & operator<<( ostream &os, TaskObject &task);
    TaskObject( const char *pname = "", unsigned int prio = 4 )
    {
      process_name = pname;
      priority = prio;
    }
private:
    unsigned int priority;
    string process_name;
};

// Friend function for "printing" TaskObject to an output stream
ostream & operator<<( ostream &os, TaskObject &task )
{
    os << "Process: " << task.process_name << " Priority: " << task.p
    return ( os );
}

// Friend class with function object for comparison of TaskObjects
class PrioritizeTasks {
public :
    int operator()( const TaskObject &x, const TaskObject &y )
    {
      return x.priority < y.priority;
    }
};

int main( int argc, char *argv[] )
{
    int i;
    priority_queue<TaskObject, vector<TaskObject>, PrioritizeTasks> t

    TaskObject tasks[] = { "JAF", "ROB", "PHIL", "JOHN"
                          ,TaskObject("OPCOM",6)  , TaskObject("Swapp
                          ,TaskObject("NETACP",8) , TaskObject("REMAC

    for ( i = 0; i < sizeof(tasks)/sizeof(tasks[0]) ; i++ )
      task_queue.push( tasks[i] );

    while ( !task_queue.empty() ) {
        cout << task_queue.top() << endl; task_queue.pop();
    }
    cout << endl;

    return( EXIT_SUCCESS );
}
```

Example 4.3 program shows a user-defined comparison function object (discussed lat
the PrioritizeTasks class. This is used to determine the relative priority of tasks an
made a friend of the TaskObject class so that it can access the private data members.
off the priority_queue, they are in our notional execution order, highest priority firs

5. Strings

```
#include <string>
```

A member of the C++ standards committee was allegedly told that if strings didn’t ap
then there was going to be a lynching. There hasn’t been a lynching, and whilst we ca
I think there is general agreement that it is a good thing to have strings at last. Those o
programming with proper languages, like Fortran, have long criticized the rather ugly
manipulation - "What ? You have to call a function to add two strings ?" being a typic

The C++ string template class is built on the basic_string template. Providing much
the container classes like vector, it has built in routines for handling character set co

characters, like NT’s Unicode. The string class also provides a variety of specialized s
finding substrings. The characteristics of the character set stored in the string are desc
structure within the string, there being a different definition of this for each type of ch
you needn’t concern yourself too much with these details if you are using strings of A
like the `vector`, expand as you add to them, which is much more convenient than C-st
either have to know how big they will be before you use them, or `malloc` and `reallo`
string that can be accommodated is given by the `max_size()` access function.

Some String Access Functions	Purpose
find(...)	Find substring or character, start
find_first_of(...)	Find first occurrence of any charac given set, starting from start of s
find_last_of(...)	Find last occurrence of any charact given set, starting from start of s
find_not_first_of(...)	Find first occurrence of characters in given set, starting from start o
find_last_not_of(...)	Find last occurrence of characters given set, starting from start of s
rfind(...)	Find substring or character, start
size()	Number of elements in vector
[]	Random access to return a single ch - no bounds checking
at(...)	Random access to return a single ch - with bounds checking
+	Concatenate strings
swap(,)	Swap two strings
insert(,)	Insert a string at the specified po
replace(...)	Replace selected substring with ano

The `string` provides the highest level of `iterator` functionality, including [] random
relevant standard algorithms work with `string`. You can `sort`, `reverse`, `merge` and s
of some algorithms, like `swap()`, are provided for strings to take advantage of certain
The `operator []` allows you to access a single character in a string, but without any b
`at()` function if you want bounds checking. The `operator+` allows easy string concat
do things like

```
string firstname, lastname, name;
.
name = firstname + " " + lastname;
```

or

```
name = firstname;
name += " ";
name += lastname;
```

Easily understandable documentation on the string class is still a bit thin on the groun
compiled some sample code to illustrate the main facilities.

Right Click & save `example_5_1.cxx`

```
// Phil Ottewell’s STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 5.1                   © Phil Ottewell 1997 <phil@yrl.co.uk>
//
```

```
// Purpose:
//          Demonstrate use of standard string class

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <algorithm>
#include <iostream>
#include <string>

#ifdef _WIN32
using namespace std;
#endif

int main( int argc, char *argv[] )
{
    size_t ip;
    string needle = "needle";     // Initialize with C style string l
    string line("my string");     // Ditto
    string haystack(line,0,3);    // Initialize with another string s
                                  // at element 3, i.e. "string"
    string string3(line,0,2);     // Initialize with first 2 characte
                                  // line, i.e. "my"
    string s1;                    // INITIALIZING with single charact
    string s2;                    // = ’A’ or (’A’) or an integer NOT
                                  // These will currently have .lengt
    string dashes(80,’-’);        // You can initialize using a chara
                                  // this, and character ASSIGNMENT i

    char old_c_string[64];

//  Concatenation using + operator
    s1 = "Now is the Winter ";
    s2 = "of our discontent made Summer";
    cout << "s1 = \"" << s1 << "\"," << "s2 = \"" << s2 << "\"\n"
         << "s1 + s2 = \"" << s1 + s2 << "\"" << endl << dashes << en

//  Find a substring in a string
    haystack = "Where is that " + needle + ", eh ?";
    cout << "haystack = \"" << haystack << "\"" << endl;
    ip = haystack.find(needle);
//  Use substr function to get substring - use string::npos (the "too
//  character count) to get the rest of the string
    cout << "ip = haystack.find(needle) found \""
         << haystack.substr(ip,string::npos )
         << "\" at position ip = " << ip << endl << dashes << endl;

//  Demonstrate use of Algorithms with strings
    line = "Naomi, sex at noon taxes, I moan";
    cout << line << " [Algorithm: reverse(line.begin(),line.end())]"
    reverse( line.begin(), line.end() );
    cout << line << " [line.length() = " << line.length() << "]" << e
         << dashes << endl;

//  Passing a string to a function requiring a C style string
    line = "copyright";
    strncpy( old_c_string,   line.c_str()   , sizeof(old_c_string)-1

    old_c_string[sizeof(old_c_string)-1] = ’\0’;
    cout << "strncpy \"" << line << "\" to c string which now contain
         << old_c_string << "\"" << endl << dashes << endl;

//  Insert into a string
    s1 = "piggy middle";
    s2 = "in the ";
    cout << "s1 = \"" << s1 << "\", s2 = \"" << s2 << "\"" << endl;
    s1.insert(6,s2); // Insert s2 in s1
    cout << "s1.insert(6,s2) = " << s1 << endl << dashes << endl;

//  Erase
    cout << "[Use s1.erase(ip,4) to get rid of \"the \"]" << endl;
    ip = s1.find("the");
    if ( ip != string::npos ) s1.erase( ip, 4 ); // Note check on ::n
    cout << s1 << endl << dashes << endl;

//  Replace
    cout << "[Use s1.replace(ip,2,\"is not in the\") to replace "
         << "\"in\" with \"is not in the\"]" << endl;
    ip = s1.find("in");
//  Note inequality check on string::npos to see if search string was
    if ( ip != string::npos ) s1.replace( ip, 2, "is not in the" );
    cout << s1 << endl << dashes << endl;

    return( EXIT_SUCCESS );
}
```

The next program puts some of the string functions to use in a simple expression evalı arithmetic-style expressions. It also shows the `at()` function, which unlike `operator[` `out_of_range`exception for a bad index. Try calculating the rest energy of an electror

Right Click & save `example_5_2.cxx`

```
// Phil Ottewell’s STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 5.2                  © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//         Transform an arthmetic expression into reverse polish
//         notation, substitute symbols and evaluate.

// ANSI C Headers
#include <ctype.h>
#include <float.h>
#include <math.h>
#include <stdlib.h>

// C++ and STL Headers
#include <iostream>
#include <map>
#include <string>
#include <stack>

// Function prototypes
double perform_operation( char oper, double operand1, double operand2
int precedence( char oper );

#ifdef _WIN32
using namespace std;
#endif

#ifdef _WIN32
# pragma warning(disable:4786) // We know basic_string generates long
#endif

int main( int argc, char *argv[] )
{
    size_t ip;
    double value, operand[2];
    char nxc, cstring[64];
    string expression, item;
    stack< string > x, y;
    map< string, double > symbol_values;
//  End of declarations ...

//  Set a couple of built-in symbols
    symbol_values["pi"] = 3.1415926535898;
    symbol_values["c"] = 299792459.0;   // Speed of light, m/s
    symbol_values["e"] = 1.602e-19;     // Electron charge, Coulombs
    symbol_values["me"] = 9.10956e-31;  // Electron rest mass, kg
    symbol_values["mp"] = 1.672614e-27; // Proton rest mass, kg
    symbol_values["mn"] = 1.674920e-27; // Neutron rest mass, kg

    if ( argc < 2) {
      cout << "Enter expression: ";
      getline( cin, expression );
    } else {
      expression = *argv[1]; // Use expression from command line if g
    }

//  Junk everything except alphanumerics, brackets and operators
    ip = 0;
    while ( ip < expression.length() ) {
      nxc = expression.at(ip);
      if ( isspace(nxc) ||
           (!isalnum(nxc) && !precedence(nxc) && nxc != ’(’ && nxc !=
        expression.erase(ip,1);
      } else {
        ++ip;
      }
    }
    if ( !expression.length() ) {
      cout << "Bye" << endl;
      return( EXIT_SUCCESS );
    }

//  Add space as an end of expression marker and to allow final pass
    expression = expression + " ";

//  Process the expression

    while ( expression.length() ) {
      nxc = expression.at(0);
      if ( nxc == ’(’ ) {
        y.push( expression.substr(0,1) ); // Push ’(’ onto Operator s
        expression.erase(0,1);

      } else if ( nxc == ’)’ ) {
        while ( !y.empty() ) { // If right brack loop until left brac
          item = y.top(); y.pop();
          if ( item.at(0) == ’(’ ) {
            break;
          } else {
            x.push( item );
          }
        }
        expression.erase(0,1);

      } else if ( !precedence( nxc ) ) {
//      If not brackets or operator stick value or variable on stack
        ip = expression.find_first_of("^*/+-() ");
        if ( ip == expression.npos ) ip = expression.length();
        item = expression.substr(0,ip);
        x.push( item ); // Push value string onto stack
        expression.erase(0,ip);

      } else {
//      nxc is operator or space
        while ( 1 ) {
          if ( y.empty() ) {
            y.push( expression.substr(0,1) );
            break;
          }

          item = y.top(); y.pop();
          if ( item.at(0) == ’(’ || precedence(nxc) > precedence(item
            y.push( item );
            y.push( expression.substr(0,1) );
            break;
          } else {
            x.push( item );
          }
        }
        expression.erase(0,1);
      }
    }

//  Put stack into correct order and substitute symbols if any
    while ( !x.empty() ) {
      item =  x.top(); x.pop();
      nxc = item.at(0);
      if ( !precedence(nxc) && !isdigit(nxc) ) {
        value = symbol_values[item]; // Not oper or number, must be a
        sprintf( cstring, "%.*g", DBL_DIG, value );
        item = string(cstring);
      }
      cout << item  << endl;
      y.push( item );
    }
    cout << endl;

//  Now evaluate, using X stack to hold operands until we meet an ope
    while ( !y.empty() ) {
      item =  y.top(); y.pop();
      nxc = item.at(0);
      if ( nxc == ’ ’) break; // End marker
      if ( !precedence(nxc) ) {
        x.push( item ); // Must be number - throw it on X stack till
      } else {
        operand[0] = operand[1] = 0.0;
        operand[1] = atof( x.top().c_str() ); x.pop(); // Get values
        if ( !x.empty() ) {
          operand[0] = atof( x.top().c_str() ); x.pop();
        }
        value = perform_operation( nxc, operand[0], operand[1] );
        sprintf( cstring, "%.*g", DBL_DIG, value );
        item = string( cstring );
        x.push( item ); // Put result on X stack
      }
    }
    cout << x.top() << endl;

    return( EXIT_SUCCESS );
}

int precedence( char oper )
{
//  Returns an precedence of operator, or 0 if it isn’t a known opera
//  Known Operators: " ","+","-","*","/","^"
//                     |                   |
//                   Do nothing          Raise to power

    if ( oper == '^') return( 4 );
    if ( oper == '*'  ||  oper == '/') return( 3 );
    if ( oper == '+'  ||  oper == '-') return( 2 );
    if ( oper == ' ') return( 1 );
    return( 0 ); //    Not operator
}

double perform_operation( char oper, double operand1, double operand2
{
//  Return the result of performing the required operation on the ope

    if ( oper == '^') return( pow( operand1, operand2 ) );
    if ( oper == '*') return( operand1*operand2 );
    if ( oper == '/') return( operand1/operand2 );
    if ( oper == '+') return( operand1+operand2 );
    if ( oper == '-') return( operand1-operand2 );
    return( 0.0 ); // Invalid operator
}
```

The expression evaluator above introduces maps, discussed later. Here they are used t numeric value from the symbolic name stored in a string.

6. Iterators

```
#include <iterator> // Don't normally need to include this yourself
```

An iterator you will already be familiar with is a pointer into an array.

```
  char name[] = "Word";
  char ch, *p;

  p = name;   // or &name[0] if you like
  ch = p[3];  // Use [] for random access
  ch = *(p+3);// Equivalent to the above
  *p = 'C';   // Write "through" p into name
  while ( *p && *p++ != 'r' ); // Read name through p, look for lette
```

Looking at the above code sample shows how flexible and powerful iterators can be. T uses `p` in at least 5 different ways. We take it for granted that the compiler will generat for array elements, using the size of a single element.

The STL iterators you've already met are those returned by the `begin()` and `end()` cc that let you loop over container elements. For example:

```
  list<int> l;
  list<int>::iterator liter; // Iterator for looping over list elemen
  for ( liter = l.begin(); liter != l.end(); liter++ ) {
    *liter = 0;
  }
```

The end-of-loop condition is slightly different to normal. Usually the end condition we comparison, but as you can see from the table of iterator categories below, not all itera increment the iterator from `begin()` and stop just before it becomes equal to `end()`. It that, for virtually all STL purposes, **`end()` returns an iterator** "pointing" to an eleme

element, which it is not safe to dereference, but is safe to use in equality tests with an
type.

Iterators are a generalized abstraction of pointers, designed to allow programmers to a
types in a consistent way. To put it more simply, you can think of iterators as a "black
and algorithms. When you use a telephone to directly dial someone in another country
how the other phone system works. Provided it supports certain basic operations, like
reporting an engaged tone, hanging up after the call, then you can talk to the remote p
container class supports the minimum required iterator types for an algorithm, then th
with the container.

This is important because it means that you can use algorithms such as the `sort` and r
seen in earlier examples, without their authors having to know anything about the con
provided we support the type of iterator required by that algorithm. The `sort` algorith
needs to know how to move through the container elements, how to compare them, an
There are 5 categories of iterator:

- Random access iterators
- Bidirectional iterators
- Forward iterators
- Input iterators
- Output iterators

They are not all as powerful in terms of the operations they support - most don't allow
we've seen with the difference between `vector` and `list`. The following is a summar
most capable at the top, operations supported on the right.

```
         Iterator Type                   Operations Supported

               ^               +-----------------------------------
              / \               \          == != < > >= <=
             /   \               \         ++ -- + - += -=
            / []  \               \             *p=               /
           /Random \               \          ->     []          /
          / Access  \               \            =*p            /
         /-----------\               \-------------------------/
        /Bidirectional\               \       == != ++ --     /
       /               \               \      *p= -> =*p     /
      /-----------------\               \-------------------/
     /      Forward      \               \     == != ++    /
    /                     \               \   *p= -> =*p  /
   /-----------+-----------\               \-------+-----/
  /Input       |      Output\               \ == !=| ++ /
 /             |             \               \++ ->|*p=/
+--------------+--------------+               \ =*p|  /
                                               \   | /
                                                \  |/
                                                 \ /
                                                  v
```

The Iterator Hierarchy

The higher layers have all the functionality of the layers below, plus some extra. Only
the ability to add or subtract an integer to or from the iterator, like `*(p+3)`. If you writ
provide all the operations needed for its category, e.g. if it is a forward iterator it must
`*p=,  ->` and `=*p`. Remember that `++p` and `p++` *are different*. The former increments t

reference to itself, whereas the latter returns a *copy of itself* then increments.

Operators must retain their conventional meaning, and elements must have the conven a nutshell, this means that the copy operation must produce an object that, when testec original item, must match. Because only random iterators support integer add and subt output iterators provide a `distance()` function to find the "distance" between any twc value returned is

```
template<class C> typename iterator_traits<C>::difference_type
```

This is useful if, for example, you `find()` a value in a container, and want to know th you've found.

```
  map< key_type, data_type >::iterator im;
  map< key_type, data_type >::difference_type dDiff;
  im = my_map.find( key );
  dDiff = distance( my_map.begin(), im );
```

Of course, this operation might well be inefficient if the container doesn't support ranc in that case it will have to "walk through" the elements comparing the iterators.

Just as you can declare pointers to `const` objects, you can have iterators to `const` elen is used for this purpose, e.g.

```
  vector<my_type>::iterator i;       // Similar to   my_type *i
  vector<my_type>::const_iterator i; // Similar to   const my_type *i
```

The `iterator_traits` for a particular class is a collection of information, like the "ite which help the STL "decide" on the best algorithm to use when calculating distances. for random iterators, but if you only have forward iterators then it may be a case of slc list to find the distance. If you write a new class of container, then this is one of the thi of. As it happens, the `vector`, `list`, `deque`, `map` and set all provide at least Bidirectior write a new algorithm, you should not assume any capability better than that which yo the category of iterator you use in your algorithm, the wider the range of containers yc with.

Although the input and output iterators seem rather poor in capability, in fact they do able to read and write containers to or from files. This is demonstrated in the program Example 7.2.

Right Click & save `example_6_1.cxx`

```
// Phil Ottewell's STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 6.1                 © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//         Demonstrate input and output iterators from and to a file

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <algorithm>
#include <fstream>
```

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

#ifdef _WIN32
using namespace std;
#endif

int main( int argc, char *argv[] )
{
    int i, iarray[] = { 1,3,5,7,11,13,17,19 };
    fstream my_file("vector.dat",ios::out);// Add |ios::nocreate to a
                                           // creation if it doesn't 
    vector<int> v1, v2;

    for (i = 0;i<sizeof(iarray)/sizeof(iarray[0]); ++i) v1.push_back(

//  Write v1 to file
    copy(v1.begin(),v1.end(), ostream_iterator<int,char>(my_file," ")
    cout << "Wrote vector v1 to file vector.dat" << endl;

//  Close file
    my_file.close();

//  Open file for reading or writing
    my_file.open( "vector.dat", ios::in|ios::out );

//  Read v2 from file
    copy( istream_iterator<int,char>(my_file), // Start of my_file
          istream_iterator<int,char>(),        // Val. returned at eo
          inserter(v2,v2.begin()));

    cout << "Read vector v2 from file vector.dat" << endl;
    for ( vector<int>::const_iterator iv=v2.begin(); iv != v2.end();
      cout << *iv << " ";
    cout << endl;

    return( EXIT_SUCCESS );
}
```

The result of the possible restrictions on an iterator is that most algorithms have **two it** arguments, or (perhaps less safely) an iterator and a number of elements count. In part using iterators, you need to be aware that **it isn't a good idea to test an iterator agai** iterator is greater than another. Testing for equality or inequality is safe except for out the loops in the example code use `iterator != x.end()` as their termination test.

Iterator Adapters

Like the container adapters, `queue`, `priority_queue` and `stack`, iterators have adapte types:

- Reverse iterators
- Insert iterators
- Raw storage iterators

The reverse iterator reverses the behaviour of the `++` and `--` operators, so you can writ

```
  vector<int> v;
  vector<int>::reverse_iterator ir;
    .
  for ( ir = v.rbegin(); ir != v.rend(); ++ir ) {
//  Whatever, going from end to start of vector
    x = *ir;
  }
```

Standard containers all provide `rbegin()` and `rend()` functions to support this kind o

The insertions iterators will, depending on the type of container, allow insertion at the the elements, using `front_insert_iterator`, `back_insert_iterator` or `insert_i` might just as well use `container.push_back()` and so forth, their main use is as the like `front_inserter()`, `back_inserter` and `inserter`, which modify how a particu work.

Raw storage iterators are used for efficiency when performing operations like copying elements to regions of uninitialized memory, such as that obtained by the STL functio `get_temporary_buffer` and `return_temporary_buffer`. Look in the `<algorithm>` iterator use.

7. We are searching for the Associative Container

"We've already got one !"
Mumbles of "Ask them what it looks like"
"Well what does it look like ?"
"It's a verra naice !"

There are four types of *associative container* in the STL. Associative containers are us we want to be able to retrieve using a key. We could use a map as a simple token/valu might be a character string, and the value might be an integer.

Associative containers store items in key order, based on a user-supplied comparison variants allow duplicate keys. Lookup is **O**(logN), N being the number of items stored containers are:

- map<Key, Type, Compare>
- multimap<Key, Type, Compare>
- set<Key, Compare>
- multiset<Key, Compare>

All four associative containers store the keys in sorted order to facilitate fast traversal. *Compare* function can simply be a suitable STL function object, e.g. `map<string, i` you are storing *pointers* to objects rather than the objects themselves, then you will ne comparison function object *even for built-in types*. The `multi` variant of the container one entry with the same key, whereas `map` and `set` can only have one entry with a part

In Stroustrup's book he shows how to make a `hash_map` variant of `map`. When workin data sets this can perform lookups in **O**(1) time, compared to **O**(logN) performance fr hashing can exhibit pathological behaviour if many keys hash to the same value, and i required, that can be a slow operation.

Map and Multimap

```
#include <map>
```

A map is used to store key-value pairs, with the values retrieved using the key. The mu keys, whereas maps insist on unique keys. Items in a map are, when they are dereferen for example, returned as a **pair**, which is a class defined in the <utility> header. The **first** and **second** which are the key and the data respectively. The pair is used throug function needs to return two values.

Some Map Access Functions	Purpose
begin()	Returns iterator pointing to first ele
end()	Returns iterator pointing _after_ last
swap(,)	Swap two elements
insert(,)	Insert a new element
size()	Number of elements in map
max_size()	Maximum possible number of elements in
empty()	True if map is empty
[]	"Subscript search" access operator

In the sample program below, which uses first and second, a list of tokens and valu

```
pi = 3.1415926535898
c = 299792459.0
```

are read in from the file tokens.dat, then you are prompted to enter a token name for w value is displayed. Because map supports the [] subscript operator, you can access the key as the subscript.

Right Click & save example_7_1.cxx

```
// Phil Ottewell’s STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 7.1                  © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//          Use map to implement a simple token database

// ANSI C Headers
#include <stdlib.h>

// C++ and STL Headers
#include <fstream>
#include <iostream>
#include <map>
#include <string>

#ifdef _WIN32
using namespace std;
#endif

#ifdef _WIN32
# pragma warning(disable:4786) // We know basic_string generates long
#endif

int main( int argc, char *argv[] )
{
    size_t ip, lnum;
    fstream fs;
    string filename, line, token, value;
    map< string, double > token_data;
//  End of declarations ...

    if ( argc > 1) {
      filename = *argv[0];
    } else {
      filename = "tokens.dat";
    }

//  Open the file for reading
    fs.open( filename.c_str(), ios::in );

//  Read each line and parse it
    lnum = 0;
    while ( fs.good() ) {
      getline( fs, line );
      if ( fs.good() ) {
//      Parse out the tokens and values
        ++lnum;
        ip = line.find_first_of("=");
        if ( ip == line.npos ) {
          cerr << "Invalid Line " << lnum << ": " << line << endl;
          continue;
        }
        token = line.substr(0,ip);
        ip = token.find(" ");
        if ( ip != token.npos ) token = token.substr(0,ip);
        value = line.substr(ip+1);
        ip = value.find_first_of("0123456789.+-");
        if ( ip != value.npos ) {
//        Store token and value
          value = value.substr(ip);
          token_data[token] = atof( value.c_str() );
        } else {
          cerr << "Bad value at line " << lnum << ": " << value << en
        }
      }

//    Junk everything except alphanumerics, brackets and operators
/*
      ip = 0;
      while ( ip < line.length() ) {
        nxc = line.at(ip);
        if ( isspace(nxc) ||
             (!isalnum(nxc) && !precedence(nxc) && nxc != ’(’ && nxc
          line.erase(ip,1);
        } else {
          ++ip;
        }
      }
*/

    }

    if ( !lnum ) {
      cerr << "Invalid or empty file: " << filename << endl;
    } else {
      for ( map< string, double >::iterator im = token_data.begin();
            im != token_data.end(); ++im )
        cout << "\"" << im->first << "\"  = " << im->second << endl;

      cout << "Enter token name: ";
      getline( cin, token );

//    Use the find function so we can spot a "miss"
      im = token_data.find( token );
      if ( im != token_data.end() ) {
        cout << " Found \"" << im->first << "\"  = " << im->second <<
      } else {
        cout << "token_data.find(...) shows no token matches \"" << t
        cout << "Lookup using token_data[\"" << token << "\"] would h
             << token_data[token] << endl;
      }
    }

    return( EXIT_SUCCESS );
}
```

In Example 5.2 we used the following lookup method with a `map`

```
  value = symbol_values[item];
```

This is fine where we know that `item` is definitely in `symbol_values[]`, but generally `find(...)` function and test against `end()`, which is the value returned if the key doe

```
  map< key_type, data_type >::iterator i
  i = my_map.find( key );
  if ( i != my_map.end() ) {
    // Got it
  }
```

Several variants of an `insert()` function exist for the `map`. The single argument versi test whether the item was already in the `map` by returning a `pair< iterator, bool >` `.second bool` value will be true and the iterator will "point" at the inserted item. On false and the iterator will point at the duplicate key that caused the insertion to fail.

The `map` can only store one value against each key. Because each key can only appear second instance of the same key, then that will supercede the existing one. Edit the `to` Example 7.1 and convince yourself that this is the case. In situations where this restric `multimap` should be used.

Set and Multiset

```
#include <set>
```

The `set` stores unique keys only, i.e. the key *is* the value. Here are some of the `set` ac

Some Set Access Functions	Purpose
begin()	Returns iterator pointing to first ele
end()	Returns iterator pointing _after_ last
swap(,)	Swap two elements
insert(,)	Insert a new element
size()	Number of elements in set
max_size()	Maximum possible number of elements in
empty()	True if set is empty

Like `map`, `set` supports the `insert()` function. Entries are kept in order, but you can p comparison function to determine that order. Useful algorithms operating on a `set` are `set_union()`, `set_intersection()`, `set_difference()` and `set_symmetric_diff` supports bidirectional iterators, *all set iterators are const_iterators*, even if you declar `set<MyType>::iterator`, so watch out for that.

Right Click & save `example_7_2.cxx`

```
// Phil Ottewell's STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
```

```
//
// Example 7.2                  © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//          Demonstrate the use of a set.

// ANSI C Headers
#include <stdlib.h>

// C++ STL Headers
#include <algorithm>
#include <iostream>
#include <set>

#ifdef _WIN32
using namespace std;
# pragma warning(disable:4786) // We know basic_string generates long
#endif

struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  { return strcmp(s1, s2) < 0; }
};

int main( int argc, char *argv[] )
{
  const char* a[] = { "Gray", "Pete", "Oggy", "Philip", "JAF", "Simon
                      "Elliott", "Roy", "David", "Tony", "Nigel" };
  const char* b[] = { "Sandy", "John", "Andrzej", "Rob", "Phil", "Hap
                      "Elliott", "Roy", "David", "Tony", "Nigel" };

  set<const char*, ltstr> A(a, a + sizeof(a)/sizeof(a[0]) );
  set<const char*, ltstr> B(b, b + sizeof(b)/sizeof(b[0]) );
  set<const char*, ltstr> C;

  cout << "Set A: ";
  copy(A.begin(), A.end(), ostream_iterator<const char*, char>(cout,
  cout << endl;
  cout << "Set B: ";
  copy(B.begin(), B.end(), ostream_iterator<const char*, char>(cout,
  cout << endl;

  cout << "Union: ";
  set_union(A.begin(), A.end(), B.begin(), B.end(),
            ostream_iterator<const char*, char>(cout, " "), ltstr() )
  cout << endl;

  cout << "Intersection: ";
  set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                   ostream_iterator<const char*, char>(cout, " "), lt
  cout << endl;

  set_difference(A.begin(), A.end(), B.begin(), B.end(),
                 inserter(C, C.begin()), ltstr() );
  cout << "Set C (difference of A and B, i.e. in A but not B): ";
  copy(C.begin(), C.end(), ostream_iterator<const char*, char>(cout,
  cout << endl;

  set_symmetric_difference( A.begin(), A.end(), B.begin(), B.end(),
                            inserter(C, C.begin()), ltstr() );
  cout << "Set C (symmetric difference of A and B, i.e. in A OR B but
  copy(C.begin(), C.end(), ostream_iterator<const char*, char>(cout,
  cout << endl;
  return( EXIT_SUCCESS );
}
```

This example also shows the use of an output iterator which in this case is directing th could just as well be a file. The `set` can only store unique keys, hence if you try and i the same key a failure will result. The single argument version of `insert( const va` `pair( it, true or false)` with the same meaning as for `map`. Remember that you means, so the situation may arise where two "identical" set elements have different da where this restriction is not acceptable, the `multiset` should be used.

8. Algorithms and Functions

We've already met and used several of the STL algorithms and functions in the examp being one of them. In the STL, *algorithms* are all template functions, parameterized b example, `sort(...)` might be implemented like this:

```
template <class RandomAccessIter>
inline void sort (RandomAccessIter first, RandomAccessIter last)
{
    if (!(first == last)) {
//     Do the sort
    }
}
```

Because algorithms only depend on the iterator type they need no knowledge of the co on. This allows you to write your own container and, if it satisfies the iterator requiren will work with a container type "unknown" when they were written. Because the algo the compiler should be able to generate inline code to do the operation just as efficien coded" the routine.

Many algorithms need a little help from the programmer to determine, for example, w container is greater, less than or equal to another. This is where *function objects* come objects are used similarly to function pointers in C. We have seen one example of fun used by Example 1.1. In *OSF/Motif* programs we often need to supply a "callback fun executed when a particular event is seen by a "Widget", e.g.someone clicking on a but do this:

```
typedef void (*CallbackProcedure)( Context c, Event e, Userdata u);
struct {
  CallbackProcedure callback;
  Userdata udat;
} CallbackRecord;
      .
/*  My function to be called when button is pressed */
void ButtonPressedCB( Context c, Event e, Userdata u);
      .
  CallbackRecord MyCallbackList[] = { {ButtonPressedCB, MyData}, {NUL
      .
  SetButtonCallback( quit_button, MyCallbackList );
```

Problems with this approach are the lack of type safety, the overhead associated with i lack of inline optimization, and problems with interpreting "user data" which tends to pointer. STL function objects avoid these problems because they are templatized, so p object is fully defined at the point of use, the compiler can generate the code inline at data can be kept in the function object, and maintains type safety. You can, of course, algorithms if you wish, but in the following sections it should become apparent that fu generally a better idea.

Algorithms

```
#include <algorithm>
```

We all know that an algorithm is abstract logical, arithmetical or computational proce applied, ensures the solution of a problem. But what is an STL algorithm ? STL algori functions parameterized by the iterator types they require. In the iterators section I like box" that allowed algorithms to act on any container type which supported the correct diagram below.

```
+------------------+                      +------------------+
|                  +---------------------+                   |
|    Algorithms    |  <-- Iterators -->  |    Containers     |
|    Algorithms    |  <-- Iterators -->  |    Containers     |
|                  +---------------------+                   |
+------------------+                      +------------------+
```

The "abstraction layer" provided by iterators decouples algorithms from containers, an capability of the STL. Not all containers support the same level of iterators, so there a of the same algorithm to allow it to work across a range of containers without sacrifici containers with more capable iterators. The appropriate version of the algorithm is aut compiler using the iterator tag mechanism mentioned earlier. It does this by using the template function within a "jacket definition" of the algorithm, and a rather convolute don’t really need to worry about (see Mark Nelson’s book pages 338-346 if you really details).

Something you should worry about is whether the containers you are using support the algorithm. The best way to determine this is to use a reference book, or look at the <al For example, the `min_element` algorithm will have a definition similar to this:

```
template <class ForwardIterator>
ForwardIterator min_element (ForwardIterator first, ForwardIterator l
```

Hence it requires a container that supports at least forward iterators. It can lead to stran algorithm with a container that doesn’t provide the necessary iterators, because the co generate code from the various templates and get confused before collapsing with an e result in a lot of

```
%CXX-E-PARMTYPLIST, Ill-formed parameter type list.
%CXX-E-BADTEMPINST, Previous error was detected during the instantiat
%CXX-E-OVERLDFAIL, In this statement, the argument list .. matches no
```

errors. In Windows you tend to get lots of

```
.. <‘template-parameter-1’,‘template-parameter-2’, ..
.. could not deduce template argument for  ..
.. does not define this operator or a conversion to a type
     acceptable to the predefined operator
```

Try compiling the example code and see what errors you get.

Right Click & save example_8_1.cxx

```
// Phil Ottewell's STL Course - http://www.yrl.co.uk/~phil/stl/stl.ht
//
// Example 8.1                    © Phil Ottewell 1997 <phil@yrl.co.uk>
//
// Purpose:
//         "It's the wrong iterators Gromit, and they're going berse
//                     (Aplogies to Nick Park and Aardmann Animatio
#include <set>
#include <algorithm>

#ifdef _WIN32
using namespace std;
# pragma warning(disable:4786) // We know basic_string generates long
#endif

int main( int argc, char * argv[]) {return 0;}// Compile and note err
void techno_trousers( set<int> &x_nasa )
{
  sort(x_nasa.begin(),x_nasa.end()); // To make this work comment thi
//  min_element(x_nasa.begin(),x_nasa.end());// uncomment this and tr
}
```

There are 60 different algorithms in 8 main categories in the STL. See Stroustrup page
all the functions.

- **Nonmodifying Sequence Operations** - these extract information, find, position
  elements but **don't change** them, e.g. find() .
- **Modifying Sequence Operations** - these are miscellaneous functions that **do ch**
  on, e.g. swap(), transform(), fill(), for_each() .
- **Sorted Sequences** - sorting and bound checking functions, e.g. sort(), lower_
- **Set Algorithms** - create sorted unions, intersections and so on, e.g. set_union(
- **Heap Operations** - e.g. make_heap(), push_heap(), sort_heap() .
- **Minimum and Maximum** - e.g. min(), max(), min_element(), max_element(
- **Permutations** - e.g. next_permutation(), prev_permutation() .
- **Numeric** - include <numeric< for general numerical algorithms, e.g. partial_s

Some of the algorithms, like unique() (which tries to eliminate adjacent duplicates) c
simply eliminate or replace elements because they have no knowledge of what the ele
actually do is shuffle the unwanted elements to the end of the sequence and return an i
the "good" elements, and it is then up to you to erase() the others if you want to. To
algorithms have an _copy suffix version, which produces a new sequence as its output
required elements.

Algorithms whose names end with the _if suffix, only perform their operation on obj
criteria. To ascertain whether the necessary conditions, known as *predicates*, have bee
function object returning a bool value. There are two types of predicate: *Predicate* an
Predicates dereference a single item to test, whereas BinaryPredicates dereference two
compare for instance.

```
template
void count_if( InputIterator first, InputIterator last, Predicate pre
```

This will return the number of objects in the range `first` to just before `last` that matc
object `pred` , which takes one argument - a reference to the data type you are checkin
requires a BinaryPredicate.

```
template
ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator
                               BinaryPredicate binary_pred);
```

This will look in the range `first` to just before `last` for two adjacent objects that "ma
found, then it returns `last`. Because a match is determined by the BinaryPredicate fun
`binary_pred` , which takes two arguments (references to the appropriate data types),
conditions you like. In fact, there are two versions of `adjacent_find` : one just requir
uses the == operator to determine equality, and the one above which gives you more c

With the information above, you should now be able to look at an algorithm in the hea
manual, and determine what sort of function object, if any, you need to provide, and w
container must support if the algorithm is to be used on it.

Functions and Function Objects

```
#include <functional>
```

Function objects are the STL's replacement for traditional C function pointers, and if
algorithms, they are written as they would be if function objects were function pointer
function pointers (or plain, old functions) with the correct argument signature if you w
objects offer several advantages, as we will see.

We have already seen a function object used in Example 4.3 to compare two task obje
`qsort` style comparison function in many respects. The function object provides type
copy constructors to be used if necessary (rather than just doing a binary copy), and d
objects be contiguous in memory - it can use the appropriate iterators to walk through
objects can be used to "tuck away" data that would otherwise have to be global, or pas
pointer. The usual template features like inline optimization and automatic code gener
types also apply.

"Why a 'function object'?", you might ask. What would be wrong with using a functi

```
template <class T>
bool is_less_than( const T &x, const T &y ) { return x < y; };
    .
    sort( first, last, is_less_than<MyObjects>() );
```

Unfortunately this is not legal C++. You can't instantiate a template function in this w
The correct thing to do is to declare a template class with `operator()` in it.

```
template <class T>
class is_less_than {                            // A function object
  bool operator()( const T &x, const T &y ) { return x < y;}
};
    .
  sort( first, last, is_less_than<MyObjects>() );
```

This is legal because we have instantiated the function object for `MyObjects`. There ar function objects within the STL. The *comparison* and *predicate* function objects whic return a `bool` value indicating the result of a comparison, e.g. one object greater than a algorithm whether to perform a conditional action, e.g. remove all objects with a parti *numeric* function objects perform operations like addition, subtraction, multiplication apply to numeric types, but some, like +, can be used with strings. Several function ob STL, such as `plus`, `minus`, `multiplies`, `divides`, `modulus`, `negate`, `equal_to`, `not_` so on. See the <functional> header file for a complete list. If the data type defines the use the pre-defined template function objects like this:

```
  some_algorithm( first, last, greater<MyObjects>() );
```

so you don’t always need to create your own function object from scratch. Try and use versions if it is available. This saves effort, reduces the chances of error and improves the flexibility of STL function objects, *adapters* are provided which allow us to comp function objects from the standard ones. If we wanted to find values greater than 1997 we would use a *binder* to take advantage of the `greater()` function, which takes two each value with 1997.

```
  iter = find_if( v.begin(), v.end(), bind2nd(greater<int>(),1997) );
```

Other adapters exist to allow negation of predicates, calling of member functions, or u pointers with binders. This topic is covered in some detail by Stroustrup in pages 518

9. STL Related Web Pages

Here are a few of the URL’s I’ve collected relating to the STL and C++ draft standard http://www.altavista.digital.com/ and search for *STL Tutorial*, or the **Yahoo!** http://wv Template Library section for more links.

- Bjarne Stroustrup’s Homepage A man who needs no introduction, he other useful C++ and STL sites
  http://www.research.att.com/~bs/homepage.html
- Mumit’s STL Newbie Guide Mumit Khan’s informative STL introdu examples
  http://abel.hive.no/C++/STL/stlnew.html
- Standard Template Library Dave Musser’s Web Page. Highly recom
  http://www.cs.rpi.edu/~musser/stl.html
- The ISO/ANSI C++ Draft Jason Merrill’s HTML of the 02-Dec-199
  http://www.cygnus.com/misc/wp/index.html
- C++ Standard Template LibraryAnother great tutorial, by Mark Sebe
  http://www.objectplace.com/te
- December 1996 Working Paper of the ANSI C++ Draft Standard
  http://www.cygnus.com/misc/wp/dec96pub/
- The Standard Template Library Silicon Graphics STL Reference Ma nonstandard features)

http://www.sgi.com/Technology/STL/stl_index_cat.html

- Ready-made Components for use with the STL Collected by Boris F

  http://www.metabyte.com/~fbp/stl/components.html
- Sites of interest to C++ users by Robert Davies, this is packed full of URLs for both the beginner and more advanced C++ programmer

  http://webnz.com/robert/cpp_site.html#Learn

10. Bibliography

This is an **approved Amazon.com Associates site**, and if you cl will take you straight to the Amazon Books order page for that b a safe option for Internet purchases: The Netscape Secure Comm encrypts any information you type in. Click here to read their pol security.

- *The C++ Programming Language (Third Edition)* by Bjarne Stroust Addison-Wesley, ISBN 0-201-88954-4

  300 more pages than 2nd edition, much of the new material concerns the STL
- *STL Tutorial and Reference Guide C++ Programming with the Stan Library* by David R. Musser and Atul Saini, Pub. Addison-Wesley, I

  Fairly useful in conjunction with Nelson
- *C++ Programmer's Guide to the Standard Template Library* by Ma Books Worldwide, ISBN 1-56884-314-3

  Plenty of examples and more readable than most of the other books
- *The Annotated C++ Reference Manual* (known as the ARM) by Mar Bjarne Stroustrup, Pub. Addison-Wesley, ISBN 0-201-51459-1

  Explains templates - a "must have" book for anyone doing C++ programming
- *Data Structures and Algorithms in C++* by Adam Drozdek, Pub. PW Company, ISBN 0-534-94974-6

  Not about the STL, but useful for understanding the implementation
- *Standard Template Library : A Definitive Approach to C++ Progra* P. J. Plauger, Alexander A. Stepanov, Meng Lee, Pub. Prentice Hall,

Back to Top

STL Tutorial Resources at Rensselaer. Standard Template Library Programmer's Guide. Phil Ottewell's STL Tutorial. Java. java.sun.com. Prof. Zaychik's notes. PDF: lec 1, lec 2, lec 3 PPT: lec 1, lec 2, lec 3. Perl. C++ Standard Library, The: A Tutorial and Reference. Nicolai Josuttis. 4.6 out of 5 stars 159.Â Encompassing a set of C++ generic data structures and algorithms, STL provides reusable, interchangeable components adaptable to many different uses without sacrificing efficiency. Written by authors who have been instrumental in the creation and practical application of STL, STL Tutorial and Reference Guide, Second Edition includes a tutorial, a thorough description of each element of the library, numerous sample applications, and a comprehensive reference. You will find in-depth explanations of iterators, generic algorithms, containers, function objects, and much more. Phil Ottewell's STL Tutorial - Free download as Word Doc (.doc), PDF File (.pdf), Text File (.txt) or read online for free.Â Table of Contents 1. Introduction 2. Templates ite Domum o Function Templates o Class Templates 3. What has the STL ever done for us ? o Vector o Deque o List o Allocators 4. Sequence Adapters o Stack o Queue o Priority Queue 5. Strings 6. Iterators o Iterator Adapters 7. We are searching for the Associative Container o Map and Multimap o Set. The C++ standard library : a tutorial and reference / Nicolai M. Josuttis.â€"2nd ed. The GNU C Programming Tutorial - C programming language. 290 PagesÂ·2012Â·1.37 MBÂ·36,982 Downloads. Coding Standards, "Using another language is like using a The GNU C Programming Tutoria Using the STL: The C++ Standard Template Library. 594 PagesÂ·2000Â·4.32 MBÂ·1,003 DownloadsÂ·New!Â to a function copies the reference of an argument into the Download C++ Tutorial - Tutorials The C++ Standard Library: A Tutorial and Reference. 1,190 PagesÂ·2012Â·6.57 MBÂ·907 DownloadsÂ·New! , and variadic templates. The C++ Standard Library: A Tutorial and Reference C# Tutorial - Tutorials for Swing, Objective C, Android. 339 PagesÂ·2015Â·2.52 MBÂ·19,078 Downloads.