Standard Library

In this lecture we will be looking at something known as the standard library.

Introduction

Built-into the standard C++ language, there are hundreds of useful functions that are provided as part of the C++ programming environment. These functions can provide all sorts of useful functionality, and can save the programmer many hours of work that would have been required to code the functions had they not been provided.

No significant program is written in just a bare programming language. First a set of supporting libraries are developed. These then form the basis for further work.

A Standard Library

In a general computer programming sense is defined as a set of standardized subroutines, macro definitions, global variables, class definitions, templates, and/or other commonly-used programming objects to extend some programming language beyond what the bare language makes available.

Minimal Program

The minimal C++ program is:

```
int main() { }
```

It defines a function called *main*, which takes no arguments and does nothing. Every C++ program must have a function named *main*(). The program starts by executing that function. The *int* value returned by *main*(), if any, is the program's return value to ''the system.''
If no value is returned, the system will receive a value indicating successful completion. A nonzero value from *main*() indicates failure.
Typically, a program produces some output. Here is a program we have seen before, it just writes out *Hello*, *world*!:

```
#include <iostream>
int main()
{
     std: :cout << "Hello, world!\n";
}
```

The line #*include* <*iostream*> instructs the compiler to *include* the declarations of the standard stream I/O facilities as found in *iostream*. Without these declarations, the expression -

```
std::cout << "Hello, world!\n"
```

- would make no sense. The operator << (‘‘put to’’) writes its second argument onto its first.
In this case, the string literal "*Hello*, *world*!*n*" is written onto the standard output stream *std*: :*cout*.
As we should know by now, a string literal is a sequence of characters surrounded by double quotes. In a string literal, the backslash character \ followed by another character denotes a single special character. In this case, *n* is the newline character, so that the characters written are *Hello*, *world*! followed by a newline.

The Standard Library Namespace
The standard library is defined in a namespace, something we will cover in a moment, called *std*.
That is why I you write *std*: :*cout* rather than plain *cout*.
When you write it this way you are being explicit about using the *standard cout*, rather than some other *cout*.

Every standard library facility is provided through some standard header similar to <*iostream*>.
For example:

```
#include<string>
#include<list>
```

This makes the standard *string* and *list* available. To use them, the *std*: : prefix can be used:

```
std::string s= "You can't be young forever but you can
           always be immature!";
std::list<std: :string> slogans;
```

When we add the line:

```
// make std names available without std:: prefix
using namespace std;
```

We no longer have to use the *std::* prefix.

Namespace

Namespaces in General
In many programming languages, a namespace is a context for identifiers.

In general, a namespace is an abstract zone which is or could be populated by names, or technical terms, or words. A namespace uniquely identifies a set of names so that there is no ambiguity when objects having different origins but the same names are mixed together. In a namespace, each name must be unique. The namespace is the context, and in the namespace each word can uniquely represent (map to) a real-world concept.

Each language is a namespace, whether it is a natural (ethnic) language, a constructed language, the technical terminology of a profession, a dialect, a sociolect, or an artificial language (e.g. a programming language).

Illustration
Within the limited namespace called "your family", you might be known as "Charlie". Within a larger namespace containing strangers as well, the name "Charlie" might not be unique, so you are instead "Charlie Brown, 17 Main Street". In (the namespace of) some other family, the name "Charlie" might refer to a different person than you.

Namespaces C++ Conclusion
The using directive allows you to bring the contents of a namespace into scope so that you can use it.

The notion of scope is the context in which the namespace is used, are we using the basic first name for all members, happens in we code the using namespace std; directive, or are we using full names. There is an in between where we can specify what functionality we want to refer to on a first name basis.

Effectively by including the `using namespace std;` directive we are making every name from the standard namespace global, which is why we define it at the beginning of our code.

Some people have commented that this is in poor taste to do this, but it is up to the programmer, we have used it so far for ease of use.
This is covered in more detail below, when we look at header files.

C++ Standard Library

The standard C++ library is a collection of functions, constants, classes and objects that extends the C++ language providing basic functionality to interact with the operating system and some standard classes, objects and algorithms that may be commonly needed. It is often divided in three main groups:

- C standard library
- Iostream Library
- STL (Standard Template Library)

The iostream library and STL library are generally defined together as standard C++ header files.
This description is a little crude as some files, such as complex (describes complex numbers) are not part of either.

The C++ Standard Library

- The standard library contains classes that are considered so useful they should be made a “standard” part of C++.

- The standard library includes at least 50 header files, those specified by the standard, although many compilers add one or two files to them. They typically include the 32 C++ header files plus 18 header files that define the facilities of the original C library.
- The components of the C++ standard library are contained within the *std::* namespace.

The original C header files have been added to the C++ standard, under the *std::* namespace. These lack the .h extension and should be used in preference to the old-style C headers, those with the .h extension.
We have seen in the code we have used so far, we have used the header files without the .h extension.
This is covered in more detail below, when we discuss the actual library files.

Built-into the standard C++ library are hundreds of useful functions, classes and types. These are provided, as standard, as part of every complete C++ implementation. These facilities can provide all sorts of useful functionality, and can save the programmer many hours of work that would have been required to code the functions had they not been provided.

In addition to the standard C++ library, most implementations offer ‘‘graphical user interface’’ systems, often referred to as GUIs or window systems, for interaction between a user and a program. Similarly, most application development environments provide ‘‘foundation libraries’’ that support corporate or industrial ‘‘standard’’ development and/or execution environments.

We won’t go in to these at all as the intent is to provide a self-contained description of C++ as defined by the standard and to keep the examples portable. Naturally, a programmer is encouraged to explore the more extensive facilities available on most systems, but that is left to your own exercises.

Instead we will look at the Standard Library as a whole and a document is supplied that reviews some of the most common and useful facilities; the majority of those are contained with in a few header files and are concerned with:

- Character handling
- Mathematics
- Random number generation
- String handling
- Date and time manipulation

When we covered functions I mentioned that before you decide to write your own function you should check to see whether there is a library one that can be used. This applies also to attempting to implement many of your own types, such as lists.

So how do we check if there is a built-in function we can use?
This is the most difficult part, especially with functions that aren't that common.
To find a function we generally need the aid of reference material, there are even whole books given over to these elements of functionality.
The most useful functions how-ever should be defined in the back of any C++ book; most books cover the more commonly used libraries such as math, time and string.
I will go through some common functions today and by searching the web it is often possible to find sites that contain function information.
Examples are:

- http://www.cplusplus.com/ref/
- http://www.halpernwightsoftware.com/stdlib-scratch/quickref.html

Compilers often have documentation; this is the case with Visual Studio. The documentation is in the form of a header file list.

So, we've managed to find a standard function that we can use what next?
Well then we need to find out:

- The name of the function.
- The header file (library) that contains the function.
- What the function expects from us.
- What we expect back from the function.
- Whether there is anything we need to be careful or aware of when calling / using the function.

Using Functions we Have Found

If you have searched through the header files and managed to locate a header file and function, or other type to use, you need to know a few details to use it.
Luckily most references come with a well documented procedure for use:

exit

<cstdlib>

```
void  exit ( int status );
```

Terminate calling process.

The process performs standard cleanup and then terminates. Before the cleanup is done any function registered by atexit is called. The cleanup consists on flushing all buffers and close any open files.

The status parameter is returned to the parent of the process (if any) or the operating system as if a return statement was specified in main function. Generally a return value of 0 or constant EXIT_SUCCESS indicates success

and any other value or constant EXIT_FAILURE is used to indicate an error or an abnormal program termination.

Parameters.

status

status value returned to parent process (if any) or operating system, generally:

status value		*description*
EXIT_SUCCESS	0	Normal termination
EXIT_FAILURE	1	Abnormal termination. Error in process.

Return Value.

(none)

Portability.

Defined in ANSI-C++

Example:

```
/* exit example */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  FILE * pFile;
  pFile = open ("myfile.txt","r");
  if (pFile==NULL)
  {
    printf ("Error opening file");
    exit (1);
  }
  else
  {
    /* file operations here */
  }
  return 0;

}
```

See also:

abort, atexit

These procedures tell us, as with any functions we define what return value/type, if any, it has and any parameters it accepts, and are these value or reference parameters.

In reference material there is generally a paragraph or two below the function that describes how we pass parameters, how to call the function, use the class and other related information.

Without this information, using the supporting functionality in the libraries can be very difficult.

The libraries that are available and including them

All C++ library entities are declared or defined in one or more standard headers. To make use of a library entity in a program, write an include directive that names the relevant standard header.

Examples include:

```
#include <iostream>      // I/O - cin, cout
#include <cstdlib>       // General purpose C library
#include <ctime>         // Access to Time
```

Commenting `#include` statements
It is good practice to put a comment at the end of each #include statement and list at least some of the functions you are using in the particular library.

For example:

```
#include <iostream>          // for cin, cout, etc
#include <cstdio>            // for getchar
#include <iomanip>           // for setw
```

This, as with most commenting, helps you and anyone else who may look at your source code know exactly why you included those header files.

Header File List
In ANSI-C++ the way to include header files from the standard library has changed.
This is particularly important information when looking at a trying to understand other people's code.

The standardisation of C++ also specifies how header file should be included. The following modifications from the way the C language specifies inclusion of header files has been implemented:

- Header file names no longer maintain the .h extension typical of the C language and of pre-standard C++ compilers, as in the case of iostream.h and iomanip.h. This extension h simply disappears and files previously known as iostream.h become iostream (without .h).

- Header files that came from the C language now have to be preceded by a c character in order to distinguish them from the new C++ exclusive header files that have the same name. For example stdio.h becomes cstdio.
- All classes and functions defined in standard libraries are under the *std::* namespace instead of being global. This does not apply to C macros that remain as C macros.
- For C++ headers, the old-style .h headers also don't support namespaces, but the extension-less headers do -- which is the current standard.

Here you have a list of the standard C++ header files, supplied by the Microsoft Developer Network (MSDN). This implementation also includes two additional headers, <hash_map> and <hash_set>, which are not required by the C++ Standard. These files include the STL.

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

The Standard C++ library works in conjunction with the 18 headers from the Standard C library, sometimes with small alterations. The headers come in two forms, new and traditional.

The new-form headers: Traditional-form headers:

ANSI-C++	**ANSI-C**
<cassert>	<assert.h>
<cctype>	<ctype.h>
<cerrno>	<errno.h>
<cfloat>	<float.h>
<ciso646>	<iso646.h>
<climits>	<limits.h>

<clocale>	<locale.h>
<cmath>	<math.h>
<csetjmp>	<setjmp.h>
<csignal>	<signal.h>
<cstdarg>	<stdarg.h>
<cstddef>	<stddef.h>
<cstdio>	<stdio.h>
<cstdlib>	<stdlib.h>
<cstring>	<string.h>
<ctime>	<time.h>
<cwchar>	<wchar.h>
<cwtype>	<wtype.h>

The Difference Between .h and no h
Standard C++ uses different filenames for the Standard C Library headers. This change was made to avoid filename conflicts, account for differences in linkage (extern "C" versus extern "C++"), and also to emphasize that there are subtle differences between the original Standard C Library and the Standard C Library-like subset of the Standard C++ Library.

If you put the .h onto the filename, then you implicitly get everything dumped into the global namespace (but that is not standard C++ behaviour, it's a compiler vendor convention.) The standard library really lives in the namespace called std, and for all uses you should prefix your types with *std::*
For example:

```
std::cout;
```

And so on. Or, if you know you'll be using the `std::cout` type frequently, you can make it a tad bit easier to code with a using declaration:

```
using std::cout;
```

Then it pulls just `cout` into the global namespace.

We have already seen the:

```
using namespace std;
```

Many people view this as the worst possible solution, as this puts everything into the global namespace.
Using this directive to put the whole namespace into the global namespace has been compared to dumping a bucket of Legos on the floor just to find one piece.

This is one reason why you are recommended to always include standard headers without the .h extension; they then are not automatically in the global scope.

Basically, including the standard library without the .h is implicitly calling "using namespace std" for you.
Another very good reason for including without the .h extension is that some files that I know of differ depending on .h or non-.h. The String header is the primary example of this.

The ISO standard is to include without a .h and so therefore that should be the primary header used.

The most commonly used header files are:

```
ctype           // Character handling
iostream        // Input/Output
fstream         // File handling
iomanip         // Input/Output formatting
cmath           // Maths functions – sin, cos etc
cstdlib         // Random Number Generation, String to int
cstdio          // Stream Input/Output
string          // String Handling
ctime           // Time/Date structures/Functions
```

Some Standard Library Functions
Here are some examples of functions provided in the standard C++ libraries.

cstdlib

```
void exit(int status)          // halts program, returns status
int  abs(int n)                // absolute value of n
long labs(long n)              // absolute value of n
int  rand(void)                // pseudo-random number 0..RAND_MAX
void srand(unsigned int seed)  // sets starting point of rand to seed
```

cmath

```
double fabs(double x)           // absolute value of x
double pow(double x, double y)  // x to power of y
double sqrt(double x)           // square root of x
double sin(double x)            // trignometric sine of x in radians
                                // similarly for cos and tan
double asin(double x)           // inverse sine, in radians, of xn
                                // similarly for acos and atan
double sinh(double x)           // hyperbolic sine of x
                                // similarly for cosh and tanh
```

string

```
strcat          // Append string
strchr          // Find character in string
strcmp          // Compare two strings
strcoll         // Compare two strings using locale settings
strcpy          // Copy string
strcspn         // Search string for occurrence of character set
strlen          // Return string length
strncmp         // Compare some characters of two strings
```

```
strncpy          // Copy characters from one string to another
```

cstdio

```
feof             // Check if End Of File has been reached.
ferror           // Check for errors.
fflush           // Flush a stream.
fgetc            // Get next character from a stream.
fgetpos          // Get position in a stream.
fgets            // Get string from a stream.
fopen            // Open a file.
fprintf          // Print formatted data to a stream.
fputc            // Write character to a stream.
fputchar         // Write character to stdout.
fputs            // Write string to a stream.
fread            // Read block of data from a stream.
freopen          // Reopen a file using a different file mode.
fscanf           // Read formatted data from a stream.
```

ctime

```
asctime          // Convert tm structure to string
clock            // Return number of clock ticks since process start
ctime            // Convert time_t value to string
difftime         // Return difference between two times
gmtime           // Convert time_t value to tm structure as UTC time
localtime        // Convert time_t value to tm struct as local time
mktime           // Convert tm structure to time_t value
time             // Get current time
```

The Standard Template Library

What is the Standard Template Library?

The Standard Template Library (STL) is a general-purpose C++ library of algorithms and data structures, originated by Alexander Stepanov and Meng Lee. The STL, based on a concept known as generic programming, is part of the standard ANSI C++ library. The STL is implemented by means of the C++ template mechanism, hence its name.

The STL is a new C++ library that provides a set of easily useable-able C++ container classes and generic algorithms (template functions).

- The container classes include vectors, lists, deques, sets, multisets, maps, multimaps, stacks, queues and priority queues.
- The generic algorithms include a broad range of fundamental algorithms for the most common kinds of data manipulations, such as searching, sorting, merging, copying, and transforming.

The STL, based on a concept known as **generic programming**, is part of the standard ANSI C++ library. The STL is implemented by means of the C++ **template** mechanism, hence its name. While some aspects of the library are very complex, it can often be applied in a very straightforward way, facilitating reuse of the sophisticated data structures and algorithms it contains.

***Note: The C++ Template mechanism is covered in Programming 2.*
On the website is an attached list that covers the parts of the C++ standard library and indicates which files are part of the STL.

Extensions to the Library

In addition to the standard C++ library, most implementations offer other library files.
There are additions are an extension to a part of the Standard Library, this includes extending the STL, but does not generally include the ANSI-C based library files.
Microsoft compilers provide the following extension: <hash_map> and <hash_set>

The <hash_map> and <hash_set> extensions are also provided as in many other implementations of the Standard Library.

These additions should be avoided if possible, in favour of what is contained as part of the standard C++ library. This helps keep code portable. Where no standard library exists for a procedure you wish to carry out, but an extension does you are encouraged to use this extension.

Non-Standard Libraries

So far we have looked at *standard libraries*.

These are called *standard* because they are (or should be) available with virtually every implementation of every C++ Compiler that adheres to the standard, regardless of which platform, machine, or operating system that that compiler may support.

Extensions to the standard library generally come packaged up with the compiler or standard library download and included in this, on via web support is generally documentation covering the use of these libraries.
The terms non-standard and extensions are sometimes used to refer to any library file that isn't part of the actual ANSI defined standard. I am using the terms to here to differentiate between two similar concepts. An example in the subtle difference would be the <hash_map> and <hash_set> extensions. These are provided by many implementations of the Standard Library.

By only using the *standard libraries*, you make your code more portable. Meaning you can be reasonably sure that you can take your C++ program and compile it on another platform, and it should work with little or no modifications.
By using *standard* built-in libraries/functions, you can be reasonably sure that your programs are portable across platforms.

However, there are also *non-standard* built-in functions / libraries that contain useful functionality. Two such *non-standard* libraries include:

- dos.h
- conio.h

These are hangovers from DOS / C days. If you use these libraries, then you may need to modify your program before you can compile it on another platform (compiler, hardware, or operating system), as there is absolutely no guarantee that other platforms support *non-standard* libraries such as these.

Non-standard libraries should be avoided where possible.

However, if you do ever decide to use these libraries, then it is absolutely essential that you:

- Document the use of these *non-standard* libraries
- Document precisely which functions you are using in these *non-standard* libraries in your source code.

For example:

```
// NON-Standard Libraries:
#include <dos.h>    // getdate
#include <conio.h>  // kbhit, clrscr
```

In short, if you ever use *non-standard* libraries, then make sure that this is made absolutely clear in your source code.

Standard Library Facilities

The facilities provided by the standard library can be classified like this:
[1] Basic run-time language support (e.g., for allocation and run-time type information);
[2] The C standard library (with very minor modifications to minimize violations of the type system);
[3] Strings and I/O streams (with support for international character sets and localization);
[4] A framework of containers (such as *vector*, *list*, and *map*) and algorithms using containers (such as general traversals, sorts, and merges);
[5] Support for numerical computation (complex numbers plus vectors with arithmetic operations, BLAS-like and generalized slices, and semantics designed to ease optimization);

The main criterion for including a class in the library was that it would somehow be used by almost every C++ programmer (both novices and experts), that it could be provided in a general form that did not add significant overhead compared to a simpler version of the same facility, and that simple uses should be easy to learn.
Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

Every algorithm works with every container without the use of conversions. This framework, conventionally called the STL [Stepanov,1994], is extensible in the sense that users can easily provide containers and algorithms in addition to the ones provided as part of the standard and have these work directly with the standard containers and algorithms.

Advice
[1] Don't reinvent the wheel; use libraries.
[2] Don't believe in magic; understand what your libraries do, how they do it, and at what cost they do it.
[3] When you have a choice, prefer the standard library to other libraries.
[4] Do not think that the standard library is ideal for everything.
[5] Remember to *#include* the headers for the facilities you use.
[6] Remember that standard library facilities are defined in namespace *std*;.
[7] Use *string* C++ string class rather than *char** C-style strings.

Documentation of Commonly used Library Procedures

On the website is an additional document that covers the basics of some of the commonly used library functions.
This should be looked at as it will help you understand C++ code you come across and will also help you add additional functionality to your own programs.

One that we are covering in this lecture is the generation of random numbers as this is important to the assessment.

Random Number Generation

The mathematical built-in functions relating to random number generation are all contained in the header file:

```
cstdlib
```

We include this by using the include directive as indicated below:

```
#include <cstdlib>
```

Random numbers are very useful for a wide range of areas, including graphics, scientific, and games programming. Random numbers are useful because they can allow your program to act differently or produce different results based on the value of a random number.

Some of the most commonly used random number functions include:

Function	Description / Return Value
void srand (unsigned x)	Prepares (seeds) the random number

	generator for use. This function should be called first, before any other random number functions.
int rand (void)	Generates a random number between 0 and RAND_MAX
int random (int num)	Generates a random number between 0 and num-1

rand()

The C++ standard library includes a pseudo random number generator for generating random numbers.
To generate a random number we use the rand() function. This will produce a result in the range 0 to RAND_MAX, where RAND_MAX is a constant defined by the implementation.

The following code generates a number in the range 0 to RAND_MAX, displaying it to the screen.

```
#include <iostream> // For cout.
#include <cstdlib>  // For rand.

int main()
{
     cout << rand() << '\n';
     return 0;
}
```

RAND_MAX

The value of RAND_MAX varies between compilers and can be as low as 32767, which would give a range from 0 to 32767 for rand(). To find out the value of RAND_MAX for your compiler run the following small piece of code:

```
#include <iostream> // For cout.
#include <cstdlib>  // For rand.

using namespace std;

int main()
{
     cout << "The value of RAND_MAX is " << RAND_MAX
          << endl;

     return 0;
}
```

Pseudo Random

Rand(), the pseudo random number generator produces a sequence of numbers that gives the appearance of being random, when in fact the sequence will eventually repeat and is predictable.

srand()
We can seed the generator with the srand() function. This will start the generator from a point in the sequence that is dependent on the value we pass as an argument. If we seed the generator once with a variable value, for instance the system time, before our first call of rand() we can generate numbers that are random enough for simple use (though not for serious statistical purposes).

In our earlier example the program would have generated the same number each time we ran it because the generator would have been seeded with the same default value each time. The following code will seed the generator with the system time then output a single random number, which should be different each time we run the program.

```
#include <ctime>    // For time
#include <iostream> // For cout.
#include <cstdlib>  // For rand, seed.


using namespace std;

int main()
{
     srand((unsigned)time(0));
     cout << rand() << endl;
}
```

Don't make the mistake of calling srand() every time you generate a random number; we only usually need to call srand() once, prior to the first call to rand().

Time and date built-in functions are contained in the header file:

```
ctime
```

We include this by using the include directive as indicated below:

```
#include <ctime>
```

Through this we can the use the time function as described above.

Generating a number in a specific range
If we want to produce numbers in a specific range, rather than between 0 and RAND_MAX, we can use the modulo operator.
Statistically it is not the most accurate way to generate a range but it's the simplest and adequate for most general purposes.

If we use rand()%n we generate a number from 0 to n-1.
By adding an offset to the result we can produce a range that is not zero based: 1 + rand()%n

```
#include <ctime>    // For time
#include <iostream> // For cout.
#include <cstdlib>  // For rand, seed.

using namespace std;

int main()
{
     srand((unsigned)time(0));
     // Generates random number between 1 and 26
     cout << 1 + rand()%26 << endl;
}
```

Assigning rand() values to integers
The rand() function returns an integer, therefore we can easily assign it's return value to an integer variable:

```
srand((unsigned)time(0));
int random_integer = rand();
```

Increased Randomisation
If you wish to use a slightly more random generation method, then you should use the following method:

```
  random_integer = 1 + int(26.0 * rand()/(RAND_MAX+1.0));
```

The first number (the one before int) specifies the lowest number to be generated and the next one (first number after the int), indicates the range of the number to be generated.

Real Life Example
This is a real life example of why for professional games a better random number generator is needed.

In mid-1999, hackers on the internet beat an On-Line Casino by guessing how their random numbers were generated, and using this information to, for example, determine which card will be next in an on-line game of Black Jack, what we call pontoon or 21.
In this case, the casino's program used the current time as the *seed* to the random number generator. All the hackers did was set their computer to the same time, and called the *seed* function. The result was that their program would generate the same random numbers in the same order as the casino's program. This meant, for example, that they knew what card would be next in an on-line game of Black Jack or Poker or any other game, making winning a virtual certainty.