Section 1: Lecture 1

Introduction to C++,
C++ Standard Library/library files,
Basics of a Typical C++ Environment,
Pre-processors Directives,
Illustrative Simple C++ Programs,
Header Files and Namespaces.

Introduction to C++

C++ is a general-purpose programming language ,That is a better than C, and supports the following features:

—Object
—Class
—data abstraction,
—supports encapsulation
—supports inheritance
—supports polymorphism
—Dynamics binding
—Message passing

C++ Standard Library/library files.

- In C++, the **C++ Standard Library** is a collection of classes and functions, which comes along with the compiler. A header file contains predefined functions and variables that you may wish to use in more than one program. Header file has common declarations and definitions (classes, structures, function prototypes)

  Example:-

- Iostream.h-used for input/output operation by the user.
- fstream.h, Math.h, String.h

Pre-processors Directives,

- Pre-processors Directives Occurs before a program is compiled.
- Pre-processors Directives Lines begin with **#** .
- Type-

  1)The **#include**

  2) **#define**

1)The #include

- Copy of a specified file included in place of the directive
- **#include <filename>**
  - Searches standard library for file
  - Use for standard library files
- **#include "filename"**
  - Searches current directory, then standard library
  - Use for user-defined files

2) `#define`

- Preprocessor directive used to create symbolic constants and macros
- Symbolic constants
  - When program compiled, all occurrences of symbolic constant replaced with replacement text
- Format

  `#define` *identifier replacement-text*
- Example:

```
#define PI 3.14159
```

- Everything to right of identifier replaces text

```
#define PI = 3.14159
```

  - Replaces "**`PI`**" with "**`= 3.14159`**"
- Cannot redefine symbolic constants once they have been created

#define

```
#define cube(x) (x*x*x)
------
volume=cube(side) will be equivalent to

Volume=(side*side*side);
```

Illustrative Simple C++ Programs.

Let's use the code below to find what makes up a very simple C++ program - one that simply prints "Hello World!" and stops. Adjust your browser so you can see the text window below the code. Then point your mouse at different statements in the program.

```
#include <iostream>
using namespace std;
int main()
{
cout << "Hello World!" << endl;
return 0;
}
```

Header files

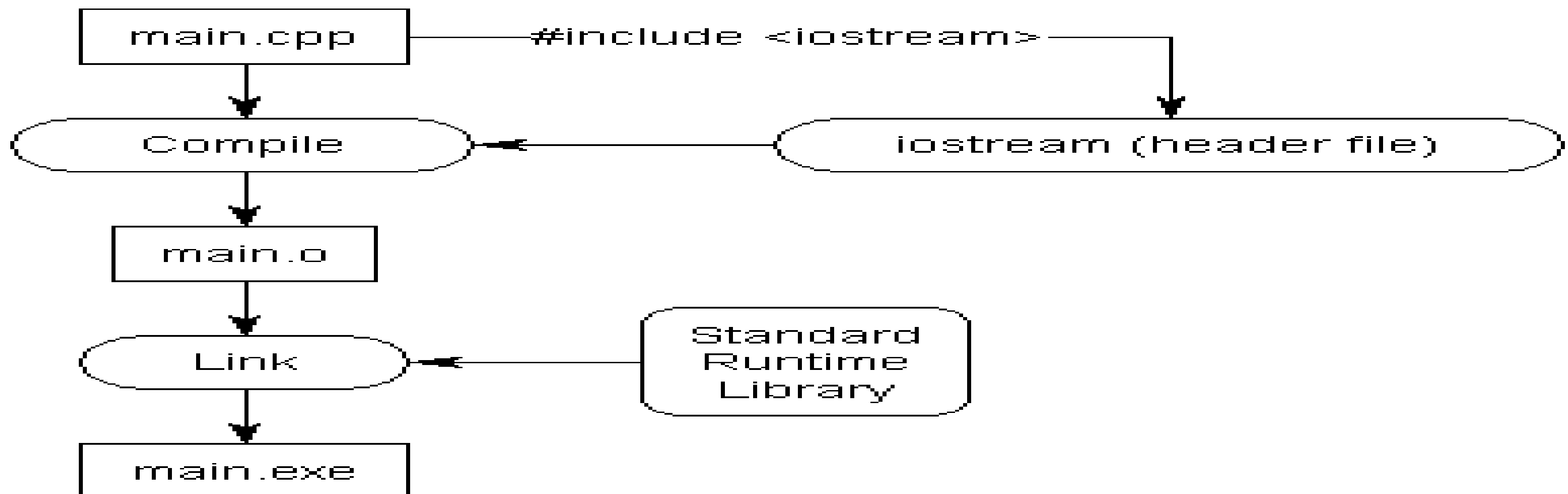
- Header files almost always have a .h extension. The purpose of a header file is to hold declarations for other files to use. When we use the line #include <iostream>, we are telling the compiler to locate and then read all the declarations from a header file named “iostream”.

  Consider the following program:

  #include <iostream>
- int main(){
- using namespace std;
- cout << "Hello, world!" << endl;
- return 0;}
- This program prints “Hello, world!” to the console using cout. However, our program never defines cout, so how does the compiler know what cout is? The answer is that cout has been declared in a header file called “iostream”. When we use the line #include <iostream>, we are telling the compiler to locate and then read all the declarations from a header file named “iostream”.

Cont..

- Keep in mind that header files typically only contain declarations. They do not define how something is implemented, and you already know that your program won't link if it can't find the implementation of something you use. So if cout is only *defined* in the "iostream" header file, where is it actually implemented? It is implemented in the runtime support library, which is automatically linked into your program during the link phase.



international C++ Standard, header files

- algorithm
- bitset
- complex
- deque
- exception
- fstream
- functional
- iomanip
- ios
- iosfwd
- iostream
- istream
- iterator
- limits
- List
- valarray
- vector

- locale
- map
- memory
- new
- numeric
- ostream
- queue
- set
- sstream
- stack
- stdexcept
- streambuf
- string
- strstream
- typeinfo
- utility

C++ Headers for C library facilities

- cctype
- cerrno
- cfloat
- ciso646
- climits
- clocale
- cmath
- csetjmp
- csignal
- cstdarg
- cstddef
- cstdio
- cstdlib
- cstring
- ctime

Incomplete list of non-standard header files

- assert
- conio
- math
- stdio
- stdlib
- time

Namespaces

- Namespaces allow to group entities like classes, objects and functions under a name. An identifier defined in a namespace is associated only with that namespace. This way the global scope can be divided in "sub-scopes", each one with its own name.

  The format of namespaces is:

  ```
  namespace identifier
  {
  entities
  }
  ```

  Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace.

Cont.. namespace

For example

```
1.      namespace  myNamespace
2.   {
3.   int  a, b;
4.   }
```

In this case, the variables a and b are normal variables declared within namespace called myNamespace.

In order to access these variables from outside themyNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
1 myNamespace::a
2 myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

- // namespaces
- #include <iostream> using namespace std; namespace first
- {
- int var = 5;
- }
- namespace second
- {
- double var = 3.1416;
- }
- int main ()
- { cout << first::var << endl;
- cout << second::var << endl; return 0;
- }

output

- 5
- 3.1416

using

The keyword using is used to introduce a name from a namespace into the current declarative region. *For example:*

```
// using
#include <iostream>
using namespace std;
namespace first
{
int x = 5; int y = 10;
}
namespace second
{
double x = 3.1416; double y =
2.7183;
}
```

```
int main ()
{
using first::x;
using second::y;
cout << x << endl;
cout << y << endl;
cout << first::y << endl;
cout << second::x << endl;
return 0;
}
```

Output

5

2.7183

10

3.1416

Using namespace have validity only in the same block in which they are stated . *For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:*

```
// using namespace example
 #include <iostream>
using namespace std;
   namespace first
     {
     int x = 5;
    }
    namespace second
   {
   double x = 3.1416;
   }S
```

```
       int main ()
{
       {
       using namespace first;
      cout << x << endl;
      }
      {
       using namespace second;
      cout << x << endl;
      }
return 0;
 }
```

output

5
3.1416

Namespace alias/ Namespace std

Namespace alias

- We can declare alternate names for existing namespaces according to the following format:

  namespace new_name = current_name;

Namespace std

- All the files in the C++ standard library declare all of its entities within the std namespace. That is why we have generally included the using namespace std;statement in all programs that used any entity defined in iostream.

Uses/application of namespace

- n large computer programs or documents it is not uncommon to have hundreds or thousands of identifiers. Namespaces (or a similar technique, see Emulating namespaces) provide a mechanism for hiding local identifiers. They provide a means of grouping logically related identifiers into corresponding namespaces, thereby making the system more modular.
- Data storage devices and many modern programming languages support namespaces. Storage devices use directories (or folders) as namespaces. This allows two files with the same name to be stored on the device so long as they are stored in different directories. In some programming languages (eg. C++, Python), the identifiers naming namespaces are themselves associated with an enclosing namespace. Thus, in these languages namespaces can nest, forming a namespace tree. At the root of this tree is the unnamed global namespace.