CSCI851 Spring-2021 Advanced Programming Lec 02

C++ Foundations I:

Introduction to Programming

Java vs C++

Outline

- What is programming?
- Programming paradigms.
- Java vs C++.
- Machine language and the JVM.
- Compilers and linkers.

What is **programming**?

- Writing **instructions** for a **computer** with the purpose of getting the computer to perform required tasks.
  - I prefer to think of programming as encompassing a lot of the design too, rather than just being the writing the code part.

- The instructions are written in a programming **language** which has a specified **syntax**.

- In particular there is syntax associated with:
  - **Input** and **output**
  - **Variables and data types**
  - **Control structures**
  - etc.

Fundamental concepts: I/O, variables, control structures…

- Input is getting data from outside …
- Output is sending data to outside …
- For a given value of ***Outside***: Outside a program, outside a function, outside the computer …
- Variables are places within the program where we store data.
- Control structures relate to where we use the results of tests to determine where we go next.

Programming paradigms

Emphasis on how the program operates

Emphasis on what the program should achieve

Declarative programming

Result focused.

Imperative programming — *Data types, Instructions*

Structured programming — *Structured program theorem*

Procedural programming — *Modularity, scope*

Object Based & Object Oriented programming — *Objects, classes + …*

Generic programming — *Templates, parameterised types*

CSCI851

Imperative:

Statements change the program state and describe how things happen.

- Don’t get too carried away with the separation though.

- You can write procedural programs that call black-box functions or modules.
  - So you know what you want to be done but don’t want to know the details → which is heading towards being declarative, but only at a certain level.

C++ vs Java: Paradigms

- Java is object oriented.
  - It's difficult to avoid objects in Java, everything has to be in a class … but some of the data types are not objects so it's not a pure object oriented language.
  - Java does have some generic programming.

- C++ can be object oriented too but …
  - You don't have to have classes → procedural programming.
  - Even when objects are around you don't need to use all the object oriented programming features, so programming can be object-based.
  - C++ can also be generic, providing extra abstraction.

- Java is used for implementing client-server web based applications, among other things.
- It's often described as being platform independent, …
- … but it's perhaps better to think of it as being a dedicated platform that you are writing for, the Java Virtual Machine (JVM).
- The Java Virtual Machine is a platform dependent application that runs on hardware/OS.
  - In many cases, but not all, the JVM is written in C or C++.

- C++ is mostly used for desktop applications and systems programming.
  - Systems programming produces software that serves the system, vs application programming which serves the user.
  - "The aim of C++ is to help classical systems programming tasks." by Bjarne Stroustrup
  - It's also used for embedded systems, resource constrained systems, and large systems.
- The evolution of C++ standards has driven it towards platform independence, but it has operating system dependent functionality.

Machine language and the JVM

- Computers can understand only a machine language, which is a sequence of binary instructions (0's and 1's).
- Generally, programs are not written in machine language, although it is possible.
- There are "simpler" languages to program in:
  - Assembly language.
  - High-level languages, such as C or C++.
- While C++ program runs as executable native machine code; so directly on the hardware/OS, a Java program runs in a **Java Virtual Machine (JVM)**.

- Java is good for standardisation, safety, and web programming, but is often slow and less powerful which is relevant for top end game coding and building large complex applications.

- The JVM manages hardware/OS resources.

- C++ allows more direct control of the hardware resources, including using memory pointers.

  - This means you have to be more careful to avoid resource mishandling.
  - We will look at programming defensively from quite early on.

C++ vs Java: OO differences

- There are naming differences:
  - Java has fields and methods, C++ has data members and member functions.
- And different functionality:
  - C++ supports multiple inheritance.
  - C++ supports operator overloading, in addition to function overloading.
  - C++ supports C-functionality, including structs and unions.
- We are going to move away from object oriented programming initially.

Warning: Hello World …

```
alert('Hello, world!');
```

Javascript

```
public class HelloWorld {
     public static void main(String[] args) {
          System.out.println("Hello, world!");
     }
}
```

Java

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

int main()
{
     cout << "Hello World!" << endl;
     return 0;
}
```

C++

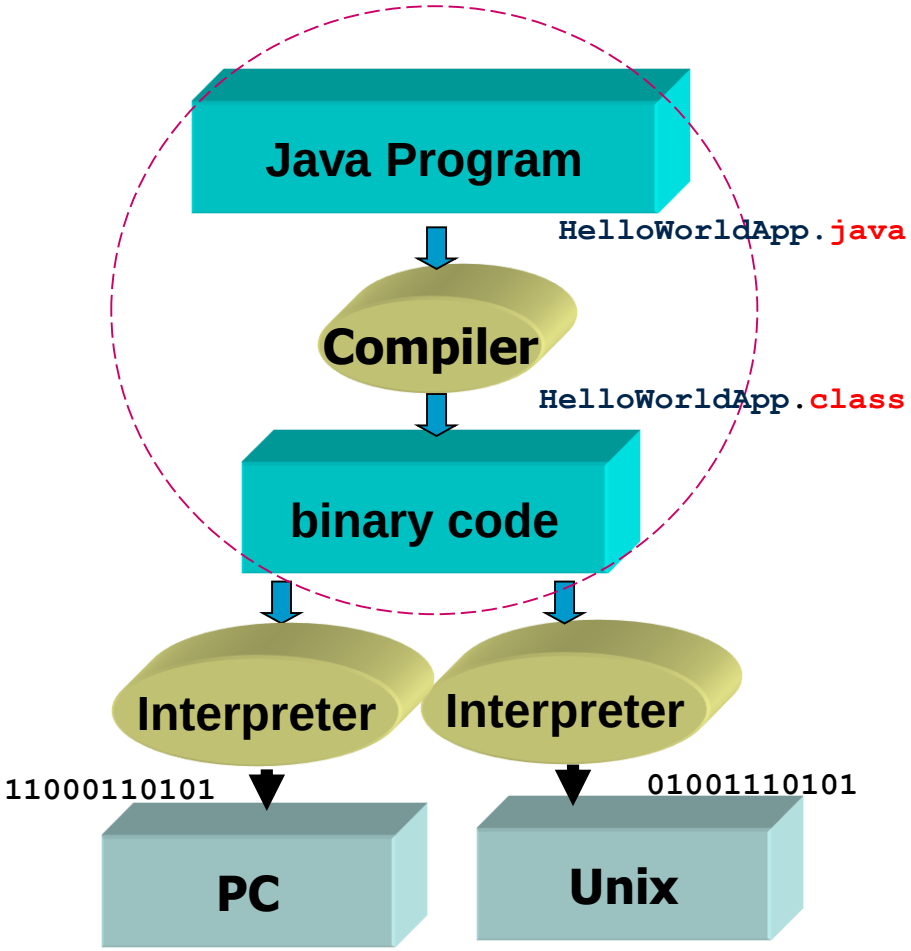
Both have a lot of syntax ☹

Compilers and linkers

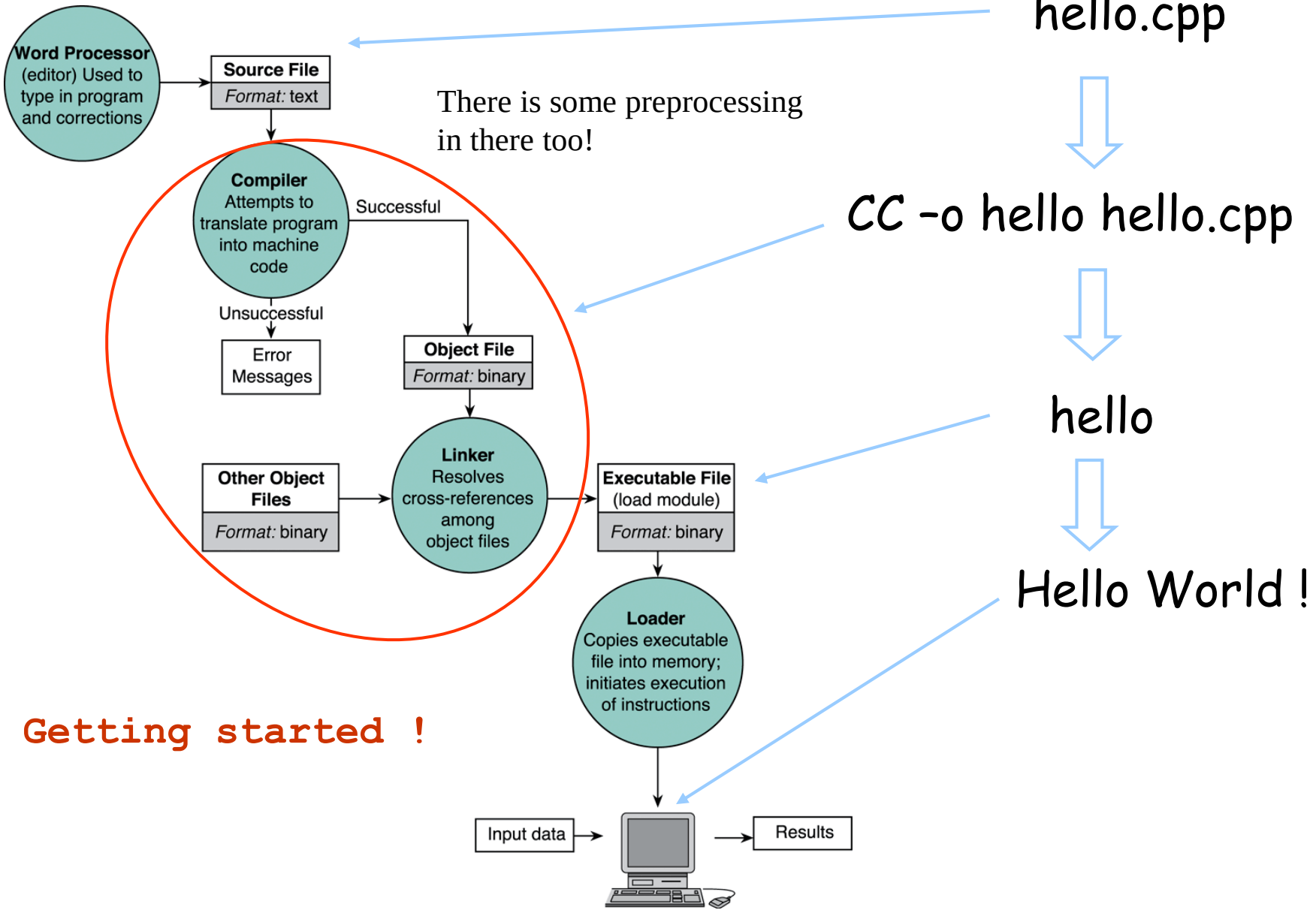
- Before the computer can run any program it needs to be converted from the programming language into machine instructions.
- This is the job of
  - A compiler, and/or
  - An interpreter …
- … in combination with a linker.

Java

- Roughly …
- The compiler take a .java file and produces a .class file, so bytecode.
- The JVM interprets that bytecode to turn it into native machine code.
- There are compilers that produce native machine code for some specified platform.



C++



Getting started !

What's next?

- Basics of C++ syntax and some preliminary procedural programming.
- Enough that you can do some decent work for the lab exercises.

C++ Foundations II:
Getting started and …
… Procedural Programming

Outline

- Procedural programming.
- Introducing `main()` …
- … explaining **Hello World!**
- Compilation.
- Comments.
- Primitive types, variables and memory.
- Functions.
- Multiple files.

Procedural programming

- In procedural programming we typically focus on a fairly specific aim, or end result.
  - The aim is often fairly static and if there need to be changes they are often localised.
  - We don’t have/need the abstraction that is common within object oriented programming.
- Code is written in a step-by-step way.
  - Typically small, and should be fairly easy to follow.

- With the code being written for a specific purpose it's possible to produce high performance code.
  - But the code probably cannot be applied to any other problem.

- With something like Object Oriented Programming (OOP), you can manage a larger code base, that makes extensive use of re-use and allows for the code to be readily developed in independent teams.

- With procedural programming you tend to get a better understanding of the step by step operations, whereas OOP tends to high the details.

- In procedural programming, procedures, also called routines, subroutines or functions, are declared or defined independent of the main program construct.
  - Main is something special we will get to soon.
- The program is a list of procedure calls, effectively a list of operations that change the state.

Step 1: Call procedure B

Step 2: Call procedure A

Step 3: Call procedure C

Step 4: Call procedure B

Step 5: Call procedure C

Step 6: Call procedure A

Step 7: Call procedure A

- It is possible to write programs without any procedures or “function calls”.

- Such code might be referred to as unstructured or sequential.

- While this may be appropriate for simple tasks there are advantages in using procedures:
  - Code can be re-used within a program.
  - Code defined to carry out a specific function can be easily transferred to another program.
  - Program flow can be more readily tracked.
    - This makes is easier to read and helps with bug fixing.

- Individual functions contain exact rules regarding the input and the output.
  - For example; exactly two integers as input, and one integer as output.

- This exactness is one of the major disadvantages with procedural languages.
  - There is a need to keep track of all the detail.

- Another major disadvantage is that *similar but not identical* pieces of code must be rewritten.
  - For example: The procedure for calculating the area of a rectangle would often be different from the procedure to calculate the area of a triangle; even though both are polygons.

Introducing `main()` …

- In Java, every method or function needs to be in a class.
- In a lot of languages, including C++, you can have stand alone functions.
- In Java applications, the `main()` function is a static public member of a Runnable class.
  - If you don't know what this means, don't stress…
- In contrast, C++ programs **must** have a stand alone `main()` function, that should return an integer, specifically an `int`, to the operating system.
  - With a return value of 0 typically meaning a normal termination, and some other value usually being associated with an error.

… explaining **Hello World!**

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

int main()
{
      cout << "Hello World!" << endl;
      return 0;
}
```

- We have our stand alone `main()`, a special case of:

```
return_type function(arguments){ }
```

Types: return_type, arguments?

- Data can be specified to be of a particular type, that's the technical term.
  - Type provides a context to data, it tells us how the string of bits we are dealing with should be treated.

- The **return_type** tells us what the output looks like, the **arguments** tell us what the input looks like.

Type and context…

- So far we have come across `int`, for integer.
- It's reasonable that we can perform arithmetic operations (+,-,…) on integers, and therefore should be able to on `int`'s.
- But if we have something like stores letters, characters → `char`, it's not so clear those operations should make sense.

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

int main()
{
      cout << "Hello World!" << endl;
      return 0;
}
```

- `#include` is like `import` in Java.
- It brings in a header, `iostream` here.
  - Effectively a collection of code written elsewhere.
- The header `iostream` is the part of the standard library for C++ associated with Input/Output.
- Bring in your own files using something like `#include “My_file.h”`
- The .h suffix conventionally indicates header.
- More on libraries and pre-processing later …

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

int main()
{
     cout << "Hello World!" << endl;
      return 0;
}
```

- Having the library iostream allows us to access input and output streams types, `istream` and `ostream`, representing input and output streams respectively.
  - Input: Use standard in: object `cin` of type `istream`.
  - Output: Use standard out: object `cout` of type `ostream`.

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

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

- So, what about the rest of the line?
- `<<` is the output or insertion operator.
  - It pushes the value right operand, in this case the literal `"Hello World!"`, to the buffer for the stream specified by the left operand, `cout`, …
  - … where it may sit until flushed to the stream by ...
- `endl`, a manipulator.
  - Manipulators modify a stream.

Literal or variable ☺

- Literals have explicit fixed values.

```
cout << "Whatever!" << endl;
```

- Variables do not.

```
cout << variable << endl;
```

- It's possible variables are fixed but the value isn't explicit here.

Output chaining …

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

- Generally, since `<<` is left-associative,

  `return X <<(ostream X, values)`

- … the left referenced `ostream` is returned and the next values in the chain added to it.
- Consider `cout << b << c << endl;`

  `(cout << b) << c << end;`

- Stream (buffer) contains b: `(cout << c) << endl;`
- Stream (buffer) contains b, c `cout << endl;`
- The manipulator `endl` writes the stream out from the memory buffer to the output stream.

- That just leaves …

```
using namespace std;
```

- This lets us avoid a longer version of …

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

- Specifically …

```
std::cout << "Hello World!" << std::endl;
```

- Namespaces are organisational tools for encapsulation.
  - We can use `X::Y` to refer to `Y` in namespace `X`.
  - They can help avoid name clashes.
- The standard namespace (`std`) contains a lot of common C++ functionality.

Not `using namespace std;`

- It's good practice not to use …

```
using namespace std;
```

- … because doing so risks collisions between names in the standard library and names you have used.
  - It's convenient for simple examples though so a fair few of our examples in the notes *will* use it…
- It's not unusual to be see use of `using` with something like the following:

```
using std::cout;
using std::cin;
using std::endl;
```

More `main()` …

- The form of `main()` didn't have arguments…

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

- … but it can have them...

```
int main( int argc, char *argv[]){ }
```

- The first parameter (`argc`) is the number of command-line arguments, *including the name of the executable itself*. c is for count.
- The second parameter is an array of C-style strings, so a sequence of characters, terminated with a special null character.

- Lets have a look and see how this an example of this runs on Banshee.

S2b_1.cpp

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

int main(int argc, char* argv[])
{
      int x = 0;
      for ( x=0; x < argc; x++ )
      {
        cout << "arg " << x + 1 << " " << argv[x] << endl;
      }
      return 0;
}
```

- This form of `main()` get parameters (input) from the command line when we run a program.
- The first line is the function header:

```
int main(int argc, char* argv[])
```

`atoi` and friends …

- The function `atoi` is often used to convert arguments to integers.
- You should be exploring this and the related functions in the Week Three lab.

Input … `cin` and >>

- So we should look at an example of `cin`, standard in, and >>, the input or extraction operator.

S2b_2.cpp

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

int main() {
        string name;
        int age;
        cout << "Enter a first name and age: ";
        cin >> name >> age;
        cout << name << " is " << age << " years old." << endl;
        return 0;
}
```

- Unlike in Java, the name of this program doesn't need to be tied to a class name.

Compilation and running: on Banshee

- We saw some examples of code and how to compile them.

```
$ CC S2b_2.cpp -o S2b_2
$ g++ S2b_2.cpp -o S2b_2
$ ./S2b_2
```

- Generally, with just one source file for now,

```
$ CC code.cpp -o output_name
$ g++ code.cpp -o output_name
```

But wait … how did these work

```
cin >> name >> age;
cout << name << " is " << age << " years old." << endl;
```

- … when we have different arguments for the operators >> and <<?
  - Operators are symbols determining particular functionality for an expression.
- These work because the insertion and extraction operators are overloaded to work with the primitive types.
  - Remember type and context earlier…
- An overloaded operator has different functionalities, depending on the arguments given to it.

- So the following all work …

```
int k = 2;
double d = 4/5;
char c = 'x';
cin >> k;              // read an int
cin >> d;              // read a double
cin >> c;              // read a char
cout << k << endl; // write an int
cout << d << endl; // write a double
cout << c << endl; // write a char
```

Commenting in C++

- It's good practice to include comments on what you want parts of your code to do …
- There are two types of comments:
  - line comments: //
  - block comments: /* */

```
// This is a one line comment

float price;   //retail price

/* This is a comment that covers
   a block over two lines        */

/**********************************
 * This is another block comment *
 **********************************/
```

Comments and variable names

- Comments should provide clarity to someone reading your code, not get in the way.
- `int intB;  //building number`
- `float price;  //retail price`

Or …

- `int building_number;`
- `Int buildingNumber;`
- `float retail_price;`
- `Float retailPrice;`

Primitive types in C++

- The types `int`, `double` and `char`, are all primitive or basic or built-in types in C++.

Type	Meaning	Minimum size
bool	Boolean	NA
char	Character	8 bits
wchar_t	Wide character	16 bits
char16_t	Unicode character	16 bits
char32_t	Unicode character	32 bits
short	Short integer	16 bits
int	Integer	16 bits
long	Long integer	32 bits
long long	Long integer	64 bits
float	Single-precision floating-point	6 significant digits
double	Double-precision floating-point	10 significant digits
long double	Extended-prevision floating-point	10 significant digits

New to C++11

- Actual sizes are compiler dependent.

Signed or unsigned

- Other than `bool`, `char`, `wchar_t`, `char16_t`, and `char32_t`, these types can have a qualifier `signed` or `unsigned`.

- Variables that are declared as `unsigned` only represent values greater than or equal to zero.

```
unsigned int cats_present = 3;
unsigned int cats_present = -1;
```

Stroustrop said …

- The Essence of C++, Columbia University 2014:

- “ If you understand `int` and `vector`, you understand C++.
  - The rest is “details” (1,300+ pages of details). ”

Variable declaration and assignment

- We have seen this a few times already.

```
variable_type   variable_name = variable_value;
double                      d = 4/5;
unsigned int     cats_present = 3;
```

- We can chain the operator = too, but = is right associative …

```
int x, y, z;
x=y=z=5;
```

- So `z` is set to `5`, then `y` to `z`, then `x` to `y`.

Linked to memory

- The primitive types map directly on to memory entities like bytes and words, entities that most processors are designed to work with.
- This allows C++ to efficiently use the hardware, without there being an abstraction in between.
- Memory is effectively seen as a sequence of bytes, each typed object is given a location in memory, and values are placed in such objects.

- We refer to and access the locations using pointers.
- We will leave pointers for now, and return to them when we look at arrays and dynamic memory.
- Pointers play a critical role in C++.

Functions …

- Procedural suggests procedures, or functions, should be used …

S2b_3.cpp

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

void print() {
      cout << "Hello world!" << endl;
}

int main() {
      print();
      return 0;
}
```

- Notice `print()` isn't part of a class.

- The function `print()` needs to be declared prior to reaching `main()`, otherwise the compiler won't recognise it.

- The structure of `main()` often tells a story about what our code does, and sometimes it's clearer if definitions are out of the way.
  - Declarations need to be prior to `main()`, definitions don't.

- So, it's not unusual to separate the declaration and the definition of functions.
  - When we do this we have a forward declaration, using the function header followed by a semi-colon `;`.

S2b_4.cpp

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

void print();     // function prototype

int main() {
  print();        // function call
  return 0;
}

// function definition
void print() {
  cout << "Hello world!" << endl;
}
```

Multiple files …

- Code is often, usually and preferably, spread across multiple files.
- It is often helpful to put all of the functions into other files.
- This is particularly useful if there are a lot of functions, and/or classes, and some of them are shared between programs.

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

void print() {
	cout << "Hello world!" << endl;
}

int main() {
	print();
	return 0;
}
```

print.h

```
void print();
```

Hello.cpp

```
#include "print.h"

int main() {
	print();
	return 0;
}
```

print.cpp

```
#include <iostream>

void print() {
    std::cout << "Hello world!" << endl;
}
```

```
$ CC Hello.cpp print.cpp -o Run
```

- Typically all data structure declarations, and function prototypes that you want to access in other files should be declared in header (`.h`) files, such as `print.h`.

- The definitions of those declarations go in the implementation file, such as `print.cpp`.

- Don't put `using namespace std;` in the included file.
  - You cannot turn it off so it applies to the rest of the main file. ☹

- Do include external functionality, like `iostream`, that you need in a file to be included, in the to be included file.
- You wouldn't necessarily need to include that external functionality in main but you shouldn't really assume you can get it through the header file of someone else.
- The file that contains `main()` is sometimes referred to as the driver file.
  - It doesn't typically have a paired header, main.h or similar, whereas usually the other code you write will come in file pairs, `myFunctions.h`, `myFunctions.cpp`.

C++ libraries …

- Sometimes we want to use functions, or classes, that other people have written.
- In particular, we often want to use the standard library.
  - Each C or C++ standard library has a corresponding header file, like `iostream`.
- These header files contain:
  - Function prototypes.
  - Definitions of various data types.
  - Definitions of constants.
  - Declarations of objects.

- For example, the math library, `cmath`, contains a square root function.

S2b_5.cpp

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

int main() {
        double a;
        cout << "Enter a number: ";
        cin >> a;
        cout << "Square root " << a << " = " << sqrt(a) << endl;
        return 0;
}
```

- More on the library can be found at http://www.cplusplus.com/reference/cmath/