Day 1: Getting started with C++

Section 1.1: Hello World

This program prints Hello World! to the standard output stream:

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

Let's examine each part of this code in detail:

#include <iostream> is a **preprocessor directive** that includes the content of the standard C++ header file iostream.
iostream is a **standard library header file** that contains definitions of the standard input and output streams. These definitions are included in the std namespace, explained below.

The standard input/output (I/O) streams provide ways for programs to get input from and output to an external system -- usually the terminal.

int main() { ... } defines a new function named main. By convention, the main function is called upon execution of the program. There must be only one main function in a C++ program, and it must always return a number of the int type.
Here, the int is what is called the function's return type. The value returned by the main function is an **exit code.**

By convention, a program exit code of 0 or EXIT_SUCCESS is interpreted as success by a system that executes the program. Any other return code is associated with an error.
If no return statement is present, the main function (and thus, the program itself) returns 0 by default. In this example, we don't need to explicitly write return 0;.
All other functions, except those that return the void type, must explicitly return a value according to their return type, or else must not return at all.

```
#include <iostream>
```

```
int main() {
    std::cout << "Hello World!" << std::endl;
}
```

---

std::cout << "Hello World!" << std::endl; prints "Hello World!" to the standard output stream:
std is a namespace, and :: is the **scope resolution operator** that allows look-ups for objects by name
within a namespace.
There are many namespaces. Here, we use :: to show we want to use cout from the std namespace.
For more information refer to Scope Resolution Operator - Microsoft Documentation.
std::cout is the **standard output stream** object, defined in iostream, and it prints to the standard
output (stdout).

<< is, *in this context*, the **stream insertion operator**, so called because it *inserts* an object into the
stream **object.**

The standard library defines the << operator to perform data insertion for certain data types into output streams. stream << content inserts content into the stream and returns the same, but updated stream. This allows stream insertions to be chained: std::cout << "Foo" << " Bar"; prints "FooBar" to the console.

"Hello World!" is a **character string literal**, or a "text literal." The stream insertion operator for character string literals is defined in file iostream.
std::endl is a special **I/O stream manipulator** object, also defined in file iostream. Inserting a manipulator into a stream changes the state of the stream.

The stream manipulator std::endl does two things: first it inserts the end-of-line character and then it flushes the stream buffer to force the text to show up on the console. This ensures that the data inserted into the stream actually appear on your console. (Stream data is usually stored in a buffer and then "flushed" in batches unless you force a flush immediately.)

An alternate method that avoids the flush is:

```
std::cout << "Hello World!\n";
```
where \n is the **character escape sequence** for the newline character. The semicolon (;) notifies the compiler that a statement has ended. All C++ statements and class definitions require an ending/terminating semicolon.

Comments :

A comment is a way to put arbitrary text inside source code without having the C++ compiler interpret it with any functional meaning. Comments are used to give insight into the design or method of a program.
There are two types of comments in C++:
Single-Line Comments
The double forward-slash sequence // will mark all text until a newline as a comment:

```
int main() {
   // This is a single-line comment.
   int a;  // this also is a single-line comment
   int i;  // this is another single-line comment
}
```

C-Style/Block Comments
The sequence /* is used to declare the start of the comment block and the sequence */ is used to declare the end of comment. All text between the start and end sequences is interpreted as a comment, even if the text is otherwise valid C++ syntax. These are sometimes called "C-style" comments, as this comment syntax is inherited from C++'s predecessor language, C:
In any block comment, you can write anything you want. When the compiler encounters the symbol */, it terminates the block comment:
```
int main() {
   /*
    *  This is a block comment.
    */
int a; }
int main() {
   /* A block comment with the symbol /*
      Note that the compiler is not affected by the second /*
      however, once the end-block-comment symbol is reached,
      the comment ends.
```

*/
int a; }

The above example is valid C++ (and C) code. However, having additional /
* inside a block comment might result in a warning on some compilers.
Block comments can also start and end *within* a single line. For example:
void SomeFunction(*/* argument 1 */* int a, */* argument 2 */* int b);

Importance of Comments

As with all programming languages, comments provide several benefits:
Explicit documentation of code to make it easier to read/maintain
Explanation of the purpose and functionality of code
Details on the history or reasoning behind the code
Placement of copyright/licenses, project notes, special thanks, contributor credits, etc. directly in the source code.
However, comments also have their downsides:
They must be maintained to reflect any changes in the code
Excessive comments tend to make the code *less* readable
The need for comments can be reduced by writing clear, self-documenting code. A simple example is the use of explanatory names for variables, functions, and types. Factoring out logically related tasks into discrete functions goes hand-in-hand with this.

Comment markers used to disable code
During development, comments can also be used to quickly disable portions of code without deleting it. This is often useful for testing or debugging purposes, but is not good style for anything other than temporary edits. This is often referred to as “commenting out”.
Similarly, keeping old versions of a piece of code in a comment for reference purposes is frowned upon, as it clutters files while offering little value compared to exploring the code's history via a versioning system.

The standard C++ compilation process :

Executable C++ program code is usually produced by a compiler.
A **compiler** is a program that translates code from a programming language into another form which is (more)
directly executable for a computer. Using a compiler to translate code is called **compilation.**
C++ inherits the form of its compilation process from its "parent" language, C. Below is a list showing the four major
steps of compilation in C++:

1. The C++ preprocessor copies the contents of any included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.

2. The expanded source code file produced by the C++ preprocessor is compiled into assembly language appropriate for the platform.

3. The assembler code generated by the compiler is assembled into appropriate object code for the platform.

4. The object code file generated by the assembler is linked together with the object code files for any library
functions used to produce an executable file.
Note: some compiled code is linked together, but not to create a final program. Usually, this "linked" code can also be packaged into a format that can be used by other programs. This "bundle of packaged, usable code" is what C++ programmers refer to as a **library.**

Many C++ compilers may also merge or un-merge certain parts of the compilation process for ease or for additional analysis. Many C++ programmers will use different tools, but all of the tools will generally follow this generalized process when they are involved in the production of a program.
The link below extends this discussion and provides a nice graphic to help.
[1]:
http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Function :

A function is a unit of code that represents a sequence of statements. Functions can accept arguments or values and return a single value (or not).
To use a function, a **function call** is
used on argument values and the use of the function call itself is replaced with its return value. Every function has a **type signature** -- the types of its arguments and the type of its return type. Functions are inspired by the concepts of the procedure and the mathematical function.
Note: C++ functions are essentially procedures and do not follow the exact definition or rules of mathematical functions.
Functions are often meant to perform a specific task. and can be called from other parts of a program. A function must be declared and defined before it is called elsewhere in a program.

Note: popular function definitions may be hidden in other included files (often for convenience and reuse across many files). This is a common use of header files.

Function Declaration:

A function declaration is declares the existence of a function with its name and type signature to the compiler.
The syntax is as the following:
int add2(int i); // The function is of the type (int) -> (int)
In the example above, the int add2(int i) function declares the following to the compiler:
The **return type** is int.
The **name** of the function is add2.
The **number of arguments** to the function is 1:
The first argument is of the type int.
The first argument will be referred to in the function's contents by the name i.

The argument name is optional; the declaration for the function could also be the following:
int add2(int); // Omitting the function arguments' name is also permitted.

Per the **one-definition rule**, a function with a certain type signature can only be declared or defined once in an entire C++ code base visible to the C++ compiler. In other words, functions with a specific type signature cannot be re-defined -- they must only be defined once. Thus, the following is not valid C++:
If a function returns nothing, its return type is written as void. If it takes no parameters, the parameter list should be empty.

Function Call:

A function can be called after it has been declared. For example, the following program calls add2 with the value of 2 within the function of main:
int add2(int i); // The compiler will note that add2 is a function (int) -> int
int add2(int j); // As add2 already has a definition of (int) -> int, the compiler
                // will regard this as an error.
void do_something(); // The function takes no parameters, and does not return anything.
                  // Note that it can still affect variables it has access to.

```
#include <iostream>
int add2(int i);    // Declaration of add2
// Note: add2 is still missing a DEFINITION.
// Even though it doesn't appear directly in code,
// add2's definition may be LINKED in from another object file.
int main() {
std::cout << add2(2) << "\n"; // add2(2) will be evaluated at this point, //
and the result is printed.
return 0; }
```

Function Definition :

A *function definition is similar to a declaration, except it also contains the code that is executed when the function is called within its body.**

Function Overloading :

You can create multiple functions with the same name but different parameters.

```
int add2(int i)
{
// Data that is passed into (int i) will be referred to by the name i
// while in the function's curly brackets or "scope."
// Definition of a variable j as the value of i+2.
// Returning or, in essence, substitution of j for a function call to
// add2.
}
int j = i + 2;
return j;
int add2(int i)
{
int j = i + 2;
return j; }
int add2(int i, int j)
{
// Code contained in this definition will be evaluated
// when add2() is called with one parameter.
// However, when add2() is called with two parameters, the
// code from the initial declaration will be overloaded,
// and the code in this declaration will be evaluated
// instead.
}
int k = i + j + 2 ;
```

return k;

Both functions are called by the same name add2, but the actual function that is called depends directly on the amount and type of the parameters in the call. In most cases, the C++ compiler can compute which function to call. In some cases, the type must be explicitly stated.

Default Parameters:

Default values for function parameters can only be specified in function declarations.
In this example, multiply() can be called with one or two parameters. If only one parameter is given, b will have default value of 7. Default arguments must be placed in the latter arguments of the function. For example:

Special Function Calls - Operators:

There exist special function calls in C++ which have different syntax than name_of_function(value1, value2, value3). The most common example is that of operators.
Certain special character sequences that will be reduced to function calls by the compiler, such as !, +, -, *, %, and << and many more. These special characters are normally associated with non-programming usage or are used for

```
int multiply(int a, int b = 7); // b has default value of 7.
int multiply(int a, int b)
{
}
return a * b;              // If multiply() is called with one parameter, the
                           // value will be multiplied by the default, 7.
int multiply(int a = 10, int b = 20); // This is legal
int multiply(int a = 10, int b);      // This is illegal since int a is in the former
```

aesthetics (e.g. the + character is commonly recognized as the addition symbol both within C++ programming as well as in elementary math).

C++ handles these character sequences with a special syntax; but, in essence, each occurrence of an operator is reduced to a function call. For example, the following C++ expression:
3+3
is equivalent to the following function call:
operator+(3, 3)
All operator function names start with operator.

While in C++'s immediate predecessor, C, operator function names cannot be assigned different meanings by providing additional definitions with different type signatures, in C++, this is valid. "Hiding" additional function definitions under one unique function name is referred to as operator overloading in C++, and is a relatively common, but not universal, convention in C++.

Visibility of function prototypes and declarations :

In C++, code must be declared or defined before usage. For example, the following produces a compile time error:

```
int main() {
  foo(2); // error: foo is called, but has not yet been declared
}
void foo(int x) // this later definition is not known in main
{
}
```

There are two ways to resolve this: putting either the definition or declaration of foo() before its usage in main(). Here is one example:

However it is also possible to "forward-declare" the function by putting only a "prototype" declaration before its usage and then defining the function body later:

```
void foo(int x) {}  //Declare the foo function and body first
int main() {
  foo(2); // OK: foo is completely defined beforehand, so it can be called
here.
}
void foo(int);  // Prototype declaration of foo, seen by main
                // Must specify return type, name, and argument list types
int main() {
  foo(2); // OK: foo is known, called even though its body is not yet defined
}
void foo(int x) //Must match the prototype
{
   // Define body of foo here
}
```

The prototype must specify the return type (void), the name of the function (foo), and the argument list variable types (int), but the names of the arguments are NOT required.

One common way to integrate this into the organization of source files is to make a header file containing all of the prototype declarations:
and then provide the full definition elsewhere:

and then, once compiled, link the corresponding object file foo.o into the compiled object file where it is used in the linking phase, main.o:

An “unresolved external symbol” error occurs when the function *prototype* and *call* exist, but the function *body* is not defined. These can be trickier to resolve as the compiler won't report the error until the final linking stage, and it doesn't know which line to jump to in the code to show the error.

Preprocessor :

The preprocessor is an important part of the compiler.
It edits the source code, cutting some bits out, changing others, and adding other things.
In source files, we can include preprocessor directives. These directives tells the preprocessor to perform specific actions. A directive starts with a
# on a new line. Example:

```
 #define ZERO 0
```

The first preprocessor directive you will meet is probably the

```
 #include <something>
```

directive. What it does is takes all of something and inserts it in your file where the directive was. The hello world program starts with the line

```
 #include <iostream>
```

This line adds the functions and objects that let you use the standard input and output.
The C language, which also uses the preprocessor, does not have as many header files as the C++ language, but in C++ you can use all the C header files.

```
// foo.h
void foo(int); // prototype declaration
// foo.cpp --> foo.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
void foo(int x) { } // foo's body definition
// main.cpp --> main.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
```

```
int main() { foo(2); } // foo is valid to call because its prototype declaration was beforehand. // the prototype and body definitions of foo are linked through the object files
```

#define something something_else directive.

This tells the preprocessor that as it goes along the file, it should replace every occurrence of something with something_else. It can also make things similar to functions, but that probably counts as advanced C++.

The something_else is not needed, but if you define something as nothing, then outside preprocessor directives, all occurrences of something will vanish.

This actually is useful, because of the #if,#else and #ifdef directives. The format for these would be the following:

```
#if something==true
//code
#else
//more code
#endif
#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

These directives insert the code that is in the true bit, and deletes the false bits. this can be used to have bits of code that are only included on certain operating systems, without having to rewrite the whole code.