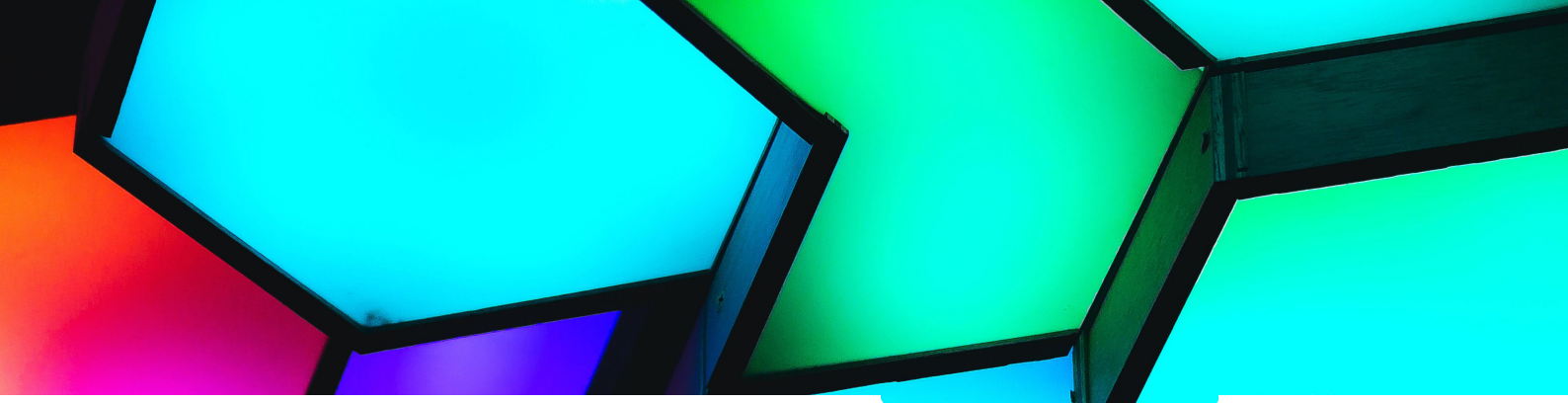

WARWICK
THE UNIVERSITY OF WARWICK

C++
Introduction for other language programmers

Research software engineering
September 2023

Contents

Introduction	3
Entry Points	4
First code	5
Printing	5
Includes	7
Functions	7
Conditionals	8
Variables	9
Scope	10
For loops	11
While loops	13
Compiling	14
Multiple files	16
Structs and classes	22

Introduction

C++ grew out of an attempt to combine C, which had become a very popular language for sys tems programming because it produced code almost as fast as hand written assembly code, with Simula, a language designed for performing simulations. In particular Simula worked using object oriented design - an approach to programming that seeks to deliberately separate a piece of software into discrete objects that interact with each other. Bjarne Stroustrup, the inventor of C++, found that object oriented design was a helpful way of designing and writing software, but that Simula itself was too slow, so he designed a language derived from C that kept the object oriented design approach but retained the performance of C. This original language was called "C with classes" (class being a common term in object oriented design for the definition of an object), but a few years after it was invented it was renamed to "C++", the new name being a pun since in both C and C++ the code "variable++" means "add one to the variable", so C++ is one bigger than C.

With this in mind the behaviour and design of C++ makes more sense. C++ was intended to be "C but with more", and although the languages have drifted away from each other they are still nearly entirely interoperable with generally only a few changes needed to C code to make it valid C++ code. Despite this C and C++ have very different approaches to how to write a "good" program and in this course we will be trying to teach you to write good C++ programs. If you are a C programmer, there will be things that we do that are new to you, but there will also be things that we don't do that you would expect. Doing things the C way will probably work. It probably isn't good C++.

In this course we're going to assume that you know an imperative compiled language. For some languages (C or Java) most of the basic syntax will be very familiar and you can probably skip some of this document, for others (Fortran, Rust, Julia) a lot of the syntax will be quite different to what you are familiar with, but there should be no new concepts. If any of the concepts in this document are new to you then you might be best learning the concepts in the language that you are most familiar with and then coming back here. If you are starting from knowing an interpreted language like Python, Matlab or R then many concepts here will be familiar to you as well, but you might find C++ an unforgiving starting point for learning how to use compiled languages.

"The Angry Penguin", used under creative commons licence from Swantje Hess and Jannis Pohlmann.

Entry Points

All normal computer programs work by having an entry point - a part of the code that is run automatically when the program starts. In C and C++ it is a function having a specific name "main". main is also used in other languages such as Rust. There are two (technically 3, but two are different ways of saying the same thing) valid versions of the main function, but we will only use the simplest

```
int main(){

}
```

This is the simplest form of main and is a C++ function returning an integer value (typename int in C++) and taking no parameters. The curly braces { } indicate the start and the end of the function. This code is actually the simplest possible valid C++ program. It does nothing, but it is valid and will compile and run.
This version is similar but the main function now takes two arguments. The first one is the number of command line parameters that the program was started with and the second one is an array of strings containing the command line parameters. We're not going to use command line arguments in this course and we're also not going to work with these old C-style strings, but this form is common enough that you have to see it.

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

}
```

You can only have one main function in your entire code (or how would the operating system know which one to start?) and your code will fail to compile if it finds two copies of main.
The final point to note is that both of these main functions are valid as they are, and that makes them subtly different to every other function in C++. They are functions that are defined to return an integer value that specifies whether or not your program completed successfully (0 is ran OK, any other value is a custom value that you can pick up from outside the program and use to identify the error). Any other function that can return a value must return a value in C++, but main is special and returns "0" (exited normally) unless you manually tell it not to.

For every other kind of function in C++ failing to return a value is **undefined behaviour**. Undefined behaviour is a **very bad thing** and you should not write code that invokes undefined behaviour. It is an indication of how bad undefined behaviour is that we are warning you about it before we actually tell you about writing any code. We can't list all of the things that can go wrong if your code invokes undefined behaviour because technically what can happen is completely undefined. Avoid it!

Entry points

Any programming language needs to have an entry point to know where the program starts.

In Fortran it is the PROGRAM block which is unique and different to every other code construct in the language. In Julia and Python it is simply that any code that isn't part of any other function is run when the program is started. In Java you can create different classes each having a **main** method and select which to use to start the program when you run it

First code

```
#include <iostream>

int mymax(int i1, int i2){
  if (i1>i2) return i1;
  return i2;
}

int main(){
  std::cout << "Maximum of 1 and 10 is " << mymax(1,10) << "\n";
  std::cout << "Maximum of 1 and -1 is " << mymax(1,-1) << "\n\n";

  int mytestval = 5;
  for (int value1 = 1; value1 < 10; value1++){
    std::cout << "Maximum of " << value1 << " and " << mytestval << "
is "
  << mymax(value1, mytestval) << "\n";
  }
  return 0;
}
```

This is the first code that comes in the introduction to the first lectured section of this course and it shows all of the key elements of computer programming - variable declarations, loops, conditionals and functions.

Here we recap all of these parts of C++, and possibly add some formal information that you might not know. Read on for more…

Printing

Generally one wants to get output from a code in some way and the easiest way of starting is printing to the screen. In C++ there are two ways of doing this - the C legacy "printf" statement and the stream IO system. We're going to teach stream IO which is a bit simpler and more common in real C++ code.

```
std::cout << "Maximum of 1 and 10 is " << mymax(1,10) << "\n";
```

Streams

Streams can be connected to lots of different things and used in the same way. File streams are very common

There are two elements to stream IO. The first is a stream object. This describes where the output should go. "std::cout" is the stream corresponding to normal output to the screen, but you can also create stream operators for outputting to files, to strings or just to chunks of memory. We're almost entirely going to be using "std::cout" here. The second element is the stream operator "<<" which outputs what is on the right of it to

the stream on the left of it and is applied from left to right through a list of objects to be output.

Technically the stream operator takes what is on the right of the operator, writes it to the specified stream object and then returns the stream object, so because the line is parsed from left to right it is equivalent to

```
std::cout << "Maximum of 1 and 10 is ";
std::cout << mymax(1,10);
std::cout << "\n";
```

This also shows up another feature of C++ that it has in common with C and is different to many other languages. In C++ lines continue until a ";" (semi-colon) character is encountered. There can be as many human readable line breaks in the source code as you want but as far as the C++ compiler is concerned they are all the same line until you reach the ;. C++ allows very long lines (at least 65,000 characters in all modern versions of C++) Semi-colons are also used in a few other places in C++, as we'll see soon, but mostly they are used for line endings.

Character vs String

In C++ all string literals are enclosed in double quotes, and unlike many languages you cannot use single quotes as an alternative. Single quotes in C++ are **character** literals and it is important to note that C++ considers a single character and a string of characters to be different things (you can have a string that is only a single character long, but that is a 1 character string, not a character).

You can also see **string literals** in use in this code, specifically the string "Maximum of 1 and 10 is ". Characters that can't be typed into strings directly are **escaped** by using the **** character followed by a character determining what special character to create. So **\t** is a tab character and **\n** is a newline character. As you can see we have included a newline character explicitly at the end of each **std::cout** statement because unlike many languages C++ doesn't explicitly add a newline character after printing, so you have to add one manually You can use escaping to include some characters that can be typed but would cause other problems, so if you want a double quote in your string literal then put in **\"** to avoid the quote closing the string literal and if you want to put in a backslash then that also has to be escaped as **** to let the compiler know that you want an actual backslash, not an escaped character.

Lines across OSes

What ends a line is not entirely common between operating systems. There are two special characters that may be involved : Line Feed (\n in C++) and Carriage Return(\r in C++), the names coming from early systems that printed their output to paper.

Unix like OSes like Linux have always used a single line feed character. Early Apple systems used just a single carriage return character and Windows originally used both, in what was often called CRLF. Nowdays, all operating systems can work with just line feed

mymax is another function that we will describe later that determines the larger of the two numbers passed to it. All functions in C++ are called by specifying their name, opening round brackets, entering a comma separated list of values or variables for the parameters to the function and then closing the round brackets. There are no other calling methods in C++ (no keyword calling, no calling a function with an array of values rather than a parameter list etc.). Unlike in languages like Fortran there is no fundamental distinction between how you call functions that return values and functions that do not, and you can freely call a function that returns a value without storing the return value to a variable and that return value is silently discarded.

Includes

```
#include <iostream>
```

One final note here is that like C but unlike most other languages most built in functions and constants of C++ are not always available, but must be imported by an include directive. The **std::cout** stream object is only available if you import the header **iostream**. Before using most features of C++ you will need to find out which header is needed for that feature and include it. Built in header files in C++ usually have no file extension, but using older C header files in C++ is common and they have the file extension ".h". Some developers also use ".hpp" for C++ header files. Whether there is a file extension or not, **#include** is doing the same job - opening the specified file and pasting in the content of it at the location of the **#include** directive.

Functions

```
int mymax(int i1, int i2){
  if (i1>i2) return i1;
  return
}
```

As we have already said, mymax is a function that we have written and functions need to have implementations defined. The structure of a function definition is almost always the same. First you have the return type of the function, which can be any C++ type (although C style arrays cannot be returned directly). There is a special return type of **void** for a function that does not return values. After the return type is the name of the function. After the function name is the list of parameters to the function enclosed in round brackets. These are pairs of variable types and names with each pair separated by commas, type followed by name. If you have a function with no parameters then just leave these brackets empty - unlike in C putting **void** into the brackets to signify that no parameters are wanted is not necessary (although it is permitted). You can have as many parameters to a function as you want but keep in mind that generally users will have to type in all of the parameters so you will want to keep the number of parameters as small as possible to let the function do its job. C++ is a **pass by value** language, which means that the parameters to a function are copied when the function is called and changing a parameter inside the function will not change the value that is passed as a parameter when the function is called. This is different to **pass by reference** languages like Fortran where the function is given a **reference** to the variable that is passed when the function is called and can (with certain restrictions) change the value of the parameters. To pass a variable by reference as a parameter to a function in C++ simply put an **&** after the parameter type and before the name. So **void func(int i)** becomes **void func(int &i)**.

No changes are needed when calling the function, but there are more restrictions on what can be passed by reference. In particular, you cannot pass literals like **"Hello world"** or **14.7** by reference because the memory that stores literals cannot be

Pass by reference

Most compiled languages nowadays can pass both by reference and by value. Avoiding copying large objects is so useful that mechanisms to avoid it are always present.

Other languages can almost all interoperate with C so must be able to pass by value to do that

changed. As well as being able to modify the value of a function parameter if it passed by reference you also can improve performance when passing large data sets since a copy of the data does not have to be made if it is passed by reference.

All functions in C++ that do not have a **void** return type (i.e. all functions that return a value at all) except **main** must have a **return** statement in all normal exit paths from the function. That return statement must be followed by a value of a type that matches the specified return type in the declaration. By "normal exit paths" we mean that if your code uses an **if** statement to perform code conditionally then a return statement must be encountered in every possible branch of the conditions. If this is not followed then you are into the world of undefined behaviour and unpredictable things can occur (including the compiler deciding that the branch without a return in it cannot happen and removing the test for it!). It is permitted to exit the code using the **exit** function or raise an **exception** (a built in method of error handling in C++ that we are not going to cover) rather than returning a value, but these are the only times that you do not have to return a value. You cannot (directly) return different types from a single function.

Literals

Literal value are, obviously, stored in the computer's memory or they couldn't be used, so it isn't obvious why you can't have a reference to them. The key to understanding is that while they do exist in memory they are normally in an area of memory that cannot be altered. This means that passing them by reference is dangerous because changing the value would crash the program. Flagging the parameter to the function as **const** to indicate that the value cannot be changed allows literals to be passed by reference. Compare this to **INTENT(IN)** and **INTENT(INOUT)** in Fortran

In C++ before a function can be called in a given line of code the compiler must have encountered the **declaration** of the function, that is to say the return type, name and parameters. The easiest way of doing that is to fully implement functions that you want in the order the you want them, but you can easily imagine a situation with a doubly recursive function with fnA calling fnB which calls fnA again etc. which would make this impossible. The solution to this is an explicit **function declaration**. Function declarations look exactly the same as the first part of the function **definition** containing the return type, name and comma separated parameter list in brackets, but then just end with a semi-colon. There are no curly brackets and no code defining what the function does. This declaration alone is sufficient to allow you to make use of a function without the compiler complaining, although you obviously do need a definition somewhere in your code. These declarations also allow you to use functions that are defined in a different file to your current one, and generally every .cpp file other than the one containing the "main" entrypoint has an associated **header file** (.h or .hpp) that contains the declarations of the functions in the .cpp file so that they can be #include-ed in files that want to use those functions. We will come back to header files later.

Conditionals

```
if (i1>i2) return i1;
```

myfunc contains a very simple example of a C++ **if** or conditional statement. The structure of the statement is very simple. **if** followed by open brackets, then a statement that evaluates to either false or true, then close brackets and then the statement or statement block to execute if the value is non zero. By statement block, we again mean a set of code lines defined in a set of { } curly braces and by statement we mean a single line of code without curly braces. If you use the curly braces then you can follow the closing } with an **else** statement and then another statement or statement block that is executed if the value in the **if** statement evaluates to true,

so the example in the **myfunc** function could be rewritten as

```
if (i1>i2)
  {return i1;}
else
  {return i2;}
```

Since you can follow **else** with a statement, you will often see code like

```
if (i1>i2)
  {return i1;}
else if (i1==0)
  {return i2;}
```

so that the **else** statement is only evaluated on another condition. This is not a formal **elseif** or **elif** statement such as is present in some languages, but is simply an **else** followed by a new **if** statement. That second **if** statement can itself be followed by an **else** that will trigger if that **if** evaluates to false, and you can continue chaining as many **if**/**else** conditions as you want like this, although there is also a **switch** construct for dealing with many related conditions on a single variable that can be faster.

It is worth remembering that C++'s **if** will automatically convert integer and pointer types into Boolean conditions on their equality to zero. This is required to be compatible with C (which uses integers equal to 0 for true and non zero for false), so a statement **if(value)** should always be read as **if(value==0)**. There is an explicit Boolean type **bool** in C++ that is the return type of the comparators like greater than (>), less than (<), equals (==) and not equals (!=) and can also be used as a variable declaration to hold logical conditions.

Variables

Initialization

C++ has very complex rules for initializing variables. If you just declare a variable then it is **default initialized** which, for normal variables, means contains whatever happened to be in the memory beforehand. Otherwise, if you put **{ }** or **( )** with nothing in them after the name of the variable then it is **value initialized**, which for normal variables means **zero initialized**. You can also manually specify initial values by **copy initialization** using =, or by **direct initialization** using () or {}

```
int mytestval = 5;
```

Before variables can be used in C++ they must be declared, that is a given variable name must be given a type. There is no concept of implicit variables like Fortran (which is convenient since it means that you don't have to turn it off). There is a concept related to the one in Python of a variable being given a type based on the value assigned to it, but there is an important difference - in C++ a variable name can only be given a type **once** (you can have the same variable name in different **scopes** but we'll come to that later) and it can never change it's type.

Variable definitions in C++ are mostly simple. Unlike Fortran and early versions of C, you can define variables wherever you want in a function, although thought must be given to clarity so it can be helpful to define your variables in specific places within your code. The declaration starts with a type declaration and then a

comma separated list of variables to be given that type. You may optionally specify an initial value for the variable with an = sign after the name of the variable and the value to give it. You will sometimes also see **constructor initialisation** or **uniform initialisation** which involve following the name of the variable with **(*value*)** or **{*value*}** respectively, but assignment by equality is more common and can be done both to intialise a variable and to assign it a value later. All of the forms of initialisation do the same basic job (at least for simple variables) - they assign an initial value to the variable. Assigning a value like this occurs each time a function is called and variables in a function do not retain their value between invocations of the function unless you explicitly flag them to with the **static** keyword before the type definition. Variables can be assigned a value at any time after they are declared. Assignment is always by using an equals sign to assign a value. This often annoys mathematicians, but is common in almost all programming languages with Pascal and R being notable exceptions.

Modifiers

There are several keywords that can be used before a typename like this and all of them affect all of the variables defined on that line. The most common are :

static means that a variable has the same lifetime as the whole program, so static variables defined in functions retain their values between calls and static variable in classes are the same for all instances of the class

const means that you want the compiler to guarantee that you can't change the value of a variable after it is initialized

```
auto mytestval = 5;
```

The option to define the type of a variable based on what it is initialised with is very simple - simply replace the typename with **auto** and the variable will be given a type based on its initial value. In this case you **must** have an initial value assigned on the declaration. There is no way of declaring a variable as **auto** type but assigning it a value later. **auto** is mostly not used for simple variables like this but can be a very convenient when you are setting the type of a variable based on the return from a function. **REMEMBER** a variable can still only have one type! You cannot change the type of a variable once it has been given a type, even if it has been given a type by **auto** inferring the type from assignment.

Scope

Shadowing

When you enter a new set of { } you can choose to **shadow** a variable that was defined outside that set of { }. You can create a new variable with the same name as an existing variable (potentially with a different type) and the compiler will act as if the original, **shadowed** variable no longer exists until the new **shadowing** variable goes out of scope. This doesn't affect the original variable, but may affect your hair while debugging - you might have less afterwards

Variables have **scope**, that is to say a part of the code in which they exist and can be used. Scope is in general quite a complex idea, but mostly it can be boiled down into a few simple rules

1. Variables exist within the { } set in which they are defined and do not exist outside those { }
2. Function parameters are scoped to within the function that they are parameters to

There is a related concept to scope of **lifetime** which is for how long a variable refers to a valid piece of the computer's memory, but the rules are fairly simple at least for normal variables. All non **static** variables have a lifetime that is the same as their scope - that is if they are out of scope they do not exist in the computer's memory. Pointer and reference variables (which we will cover in the course) decouple lifetime and scope because they

are types of variable that refer to other memory locations. We will discuss their lifetimes when we introduce them.

When working out lifetimes for variables one does have to be careful to remember that there are some times when variables are copied (such as when they are passed to a function) and the lifetime of the copy might be different to the lifetime of the original variable.

For loops

```
for (int value1 = 1; value1 < 10; value1++){

}
```

For loops in C are by far the most common type of loops. They are more complex than for loops in most other compiler languages, but also more powerful.

While the syntax is more powerful than in many languages you can easily produce a normal **for** loop like those in most languages, which is as shown below

In a for loop, you have three sections which are separated by semi-colons, and then the code to be run repeatedly in curly braces. In the above example, you first create the loop variable, here **value1** and you initialise it, here to the value 1. Then, in the next section you set up the condition for when the loop should continue running, here the loop should continue running for so long as the value is less than 10. Many other languages have loops that run until and including when a given value is reached, in which case this should be **value1<=10**. Finally, you set how the loop variables should be incremented on each iteration, here that **value1** should be incremented by 1 on each iteration. In most languages you **do** have control on how the loop variable should increment on each iteration, where it is usually called the **stride.** Where C and C++ are unusual are in needing to specify the increment even when you are just adding 1. This type of loop is by far the most common way of using for loops in C++, but it is much more powerful than this, and it is worth examining the for loop in more detail.

You start with the **for** keyword, open bracket, three sections which are separated by semi-colons, a closing bracket and then a line of code or code block in { } to execute repeatedly. The semicolons in the for loop are doing a different job to the semicolons that end lines and are are one of the few other uses of **;** in C++. Each of the three sections is run at very specific times as the loop is run, and each section may be empty if there is no code to run at that section.

```
for (int value1 = 1; value1 < 10; value1++){

}
```

The first section (now highlighted in orange) runs before the loop first starts and is generally used to initialise values used within the loop. In C++ and versions of C newer then C99 you can both initialise and define a loop variable in this section as is demonstrated in this example. You don't have to do this, and can use this section to initialise a variable defined outside the loop,

although the loop variable does have to be defined somewhere.

Unlike many languages, you are not **required** to specify an initial condition for the loop variable here, and if the loop variable has to be given a complex initialisation it is common for the loop variable to be defined and initialised outside the loop, in which case this section is just empty and the **for** statement opens with a **;**.

```
for (int value1 = 1; value1 < 10; value1++){

}
```

The second section (also now highlighted in orange) runs **before** every iteration of the loop. It must be an expression that returns a value and that value is treated as if it was a parameter to an **if** block. If the expression evaluates to **true** then the loop ceases evaluation and control transfers to after the code within the **for** statement. If the expression evaluates to a non-zero value then the loop iteration executes. If this section is empty the the condition is always assumed to be non-zero and the loop will continue until it is exited by another means. As such, the general use of this section is the termination condition for the loop.

```
for (int value1 = 1; value1 < 10; value1++){

}
```

The third and final section (highlighted in orange) runs **after** every iteration of the loop. It always runs unconditionally after each iteration of the loop and is generally used to say how to update the loop variables. This is similar to, but more powerful than the **stride** that many languages have in their loop constructs. With a stride one can say what should be added to the loop variable after each iteration, but here in C++ we can modify the loop variable however we want. If we want the loop index to increment by 1 (as shown here) we can use the **++** increment operator to simply increase the loop index by 1 (value++ is equivalent to value = value +1). If we want the loop index to increase by 2 then we can use the equivalent operator **value+=2**. If we want more complex things then we can just write them, so having **value *= 2** would be completely valid (as you might guess, that multiplies value by 2 each time the loop finished).

We have examined in detail here a for loop that is of conventional type, but from this breakdown you can see how the extra power comes. You don't have to have a normal loop variable, or indeed any particular requirements to be satisfied at all. You can call a function in the second section to check termination, so a loop that stops with 50/50 probability could be as simple as

```
for (;random_unit()<0.5;){

}
```

if **random_unit()** returns a random value between 0 and 1. The net effect of this is to make C++ for loops a bit more complex than in many languages, but much more powerful.

There is a different form of **for** loop in C++ which is equivalent to **for element in list:** in Python and loops over the elements of a container, giving you access to each element. This form of **for** typically looks like

```
for (auto element:list){
  std::cout << el << "\n";
}
```

We won't cover this form in detail here, but it is useful to know that it exists so that you aren't confused if you see it in the wild.

Loop controls

break - immediately exit the loop regardless of any other conditions and jump to the code after the loop body

continue - immediately begin the next cycle of the loop. In a for loop the third section will execute and the second section will be tested to see if the loop should then trigger. Calling **continue** on the last cycle of a loop will cause the loop to immediately terminate

Unlike in many languages you can modify the values of loop variables within the loop in C++, but it is generally a bad idea to both have the loop variable updated in the third section of the loop construct and change the variable in the loop body. It can lead to confusion

Be careful not to lose readability when designing loops! You can easily confuse yourself.

While loops

While loops are the other type of loops in C++ and come in two related forms. The first has while at the start of the loop

```
int i=0;
while (i<10){
 i++;
}
```

This type of while loop is very simple. You have **while** followed by a condition in brackets. The condition is evaluated before the start of each iteration and as long as that condition evaluates to non-zero the iteration will proceed. Here we are effectively reproducing a **for** loop but moving the increment operator inside the loop. This type of while loop will only execute at all if the condition is non-zero when the loop is first encountered. This type of loop can be reproduced in a **for** loop that looks like **for(;i<10;)** with the first and third section empty.

The second type of while loop looks like

```
int i=11;
do {
 i++;
} while (i<10);
```

and starts with **do**, then the loop body in { } and then finally a while statement much as before. In this case, the while statement is evaluated at the end of the loop iteration and is used to determine if the next iteration of the loop should continue. So in this example, even though i is given the initial value of 11, already greater than the test value of 10, the loop will run once because the test is only performed at the end of the loop. Note that unlike all of the other types of loops there is a **;** after the while statement here because the construct doesn't end with the close of the curly braces.

Compiling

Once your code is written you have to compile it into an executable so that it can be run. C++ is a very popular language with a formally defined ISO standard so there are many C++ compilers that you can chose from. If you are running under Linux then you will usually be using the free, open source "g++" compiler which compiles from the command line. You might have to install this from your package manager, where it will usually be either in a package called g++ or gcc.

If you are on a Mac then you want to start by installing XCode. XCode is an **integrated development environment** and can be used as a graphical tool, but also installs that g++ command line compiler (technically on Mac the default compiler is a different tool called **clang++** but typing **g++** will run **clang++** on a Mac. You can install actual **g++** through various package managers, but mostly **clang++** is as good as **g++**, so we don't particularly recommend that people do).

Native Windows Compilers

The most popular native Windows C++ compiler is Microsoft Visual Studio where the "community edition" is free to students (https://visualstudio.microsoft.com/students/), open source developers (might apply to some University projects) and individual developers (might apply to you personally, but as employed by the University you are not an individual developer). We are not going to cover how to use Visual Studio in this course.

On Windows 10 or 11 you can install the Windows Subsystem for Linux (https://learn.microsoft.com/en-us/windows/wsl/setup/environment) that basically gives you a working Linux installation within Windows. You can then install g++ inside that Linux environment and use it as you would with Linux (https://learn.microsoft.com/en-us/windows/wsl/setup/environment).

The final result of installing these things should be that there should be a command line somewhere where you can type **"g++"** and get a response that looks something like

```
g++: fatal error: no input files
compilation terminated.
```

or, on a Mac

```
clang: error: no input files
```

Both of these errors are saying that the compiler has been found and run successfully, but we haven't told it what files to compile. If you get an error about "**command not found**" then you have not successfully installed g++ so you will want to check about doing that. The problem with not having specified a file to compile is easily fixed - simply put in the name of your cpp file immediate after **g++**. The exact parameters to g++ and to clang++ are slightly different, but most of the common parameters are very similar, so for this document, we will treat them as interchangeable. So, if your C++ program is in the file "main.cpp", simply type

```
g++ main.cpp -o program
```

The first two bits are fairly obvious. **g++** is the compiler that we want to run, main.cpp is the C++ source file that we want to compile. The **"-o"** flag is fairly simple as well and simply specifies what filename the executable program that the compiler creates will have, in this case **"program"**. If you don't specify an output filename like this the compiler will generate a program called **"a.out"** (for historical reasons). That is an unhelpful name and shouldn't be generally used. It is the most common program observed running on many supercomputers which indicates how often people forget to use **-o** but it is always unhelpful!

If your program compiles successfully then there will be no output from the compiler and it will just return you straight to the command prompt. If the compiler outputs things that are flagged as "warnings" then that is not a good sign because it might mean that there is an error in your code but not one bad enough to stop the code from compiling. If the compiler outputs that there are errors then your code will not have compiled. The specified executable may exist anyway if you have previously compiled your code successfully with the same name because the compiler doesn't delete the existing file, it simply overwrites it with the new one. Watch out for this because you can get very confused if the code fails to compile and you run an old version of the executable file.

If you get errors or warnings from a compilation then there are no fixed rules on how to fix it. An error definitely means that you have written code that is invalid (compiler bugs are extremely rare, so it is almost never worth considering that a failure to compile is due to a compiler error.) One of the problems with C++ is that it can generate a large number of errors, and some of them might not be errors directly but are "follow-on" errors due to an earlier problem. For example, if you miss a semi colon on the end of a line then not only is that line invalid, but it will make the line below invalid. Missing close curly braces can make errors appear for many lines after the line that is really causing the problem. As a general rule, you should look through the list of errors and warnings from the top and read downwards. Mostly as you fix errors near the top other errors lower down will tend to fix themselves since they are these follow-on errors. Do not try and fix errors from the bottom up, even though it is tempting because those errors are the ones that you see first when the compiler finishes printing errors! One useful trick is to get the compiler to output the errors to a text file so that you can read them all at your leisure. The

command on Linux, MacOS and WSL is

```
g++ main.cpp -o program >& error.txt
```

When you run this command you will see no output to the terminal but the file **error.txt** will contain the errors from the compilation.

Once your program is compiled you can generally just run it by typing

```
./program
```

The **"./"** in front of the name of the executable being needed since for security reasons Linux and similar OSes don't generally just run programs in the directory that you are in by default, so specifying "./" means "find this program in my current directory".

There are methods of writing C++ code and having it be executed as soon as it is written rather than compiling it and then running the resultant code (see for example https://www.pranav.ai/cplusplus-for-jupyter) but generally there is little point writing a compiled language like that and it is very uncommon in the real world.

Multiple files

It is generally good form to not hold all of your code in a single file, but to split it into multiple files, where each file contains code that is logically connected together. By logically here, we mean in terms of making sense to a person trying to read your code, so you might combine functions that deal with different parts of your algorithm, split off code dealing with reading and writing files from code that works with the data or similar approaches. In object oriented code, it is very common for each object's code to be defined in its own file since the idea of object oriented design is the objects are already independent pieces of logically connected code. Some people go so far as to have each function in its own file, but this was mainly to make it easy to find a function in the days before modern editors so this is much less common than it used to be and not particularly recommended.

One Definition Rule

In C++ there is one very important rule that you have to know about which is called the **one definition rule**. The actual rule is quite long because it has to be precise, but the idea is fairly simple - things like function definitions can only exist **once** in the entire program, and certain other things (notably those class definitions that we mentioned as part of object oriented design) can only appear once when compiling a given file. It is easy to see examples of why the rule exists - if there are two definitions for the same function then how can the compiler know which one to choose?

In C++ functions can only be declared once because of the **one definition rule**. You can have as many **declarations** as you like, but only one definition. The solution is to separate the **definitions** of your functions from the **declarations** of them. This is done by splitting your code in *source files* (.cpp extensions) containing the **definitions** and header files (.h or .hpp extensions) containing the **declarations**. The declaration has to be available anywhere you use the function, so you **#include** the declaring header file anywhere you want to use it , but the definition is effectively compiled all by itself and **linked** to the places you call it by the

compiler. So what does this look like? Fortunately, it isn't too horrible

```
#include "function.h"

double my_function(double d){return d * 2.0;}
```

function.cpp

This is an example of a source file defining a function called **my_function**. It looks exactly like the files that we have already written but without the special **main** function that tells the program where to start. That is correct because there can only be one **main** function, so only one source code file in a C++ program has main in it (it is often called either **main.cpp** or **programname.cpp** where **programname** is the name you have chosen for your program) It looks a little bit different since we haven't split up the definition of the function over multiple lines in the source code, but remember that in C++ that the compiler really only cares about { } pairs and semi colons so that is only a change in how the code looks. You will notice that there is a **#include** line that is similar but different to the line that we already saw **#include<iostream>**. The difference is between using < > to indicate which file to include and " ". There are some subtleties to exactly what this difference is but basically you should use < > when you are including headers built into the language and " " when you are including your own header files. The difference is basically whether the compiler should search for files in your custom header file locations or the default header file locations. Using " " with a built in header file will work but will take the compiler slightly longer to compile your code. Using < > with your own header files **won't** work because the compiler will only look in the default locations for header files. You might wonder why we are including the header file matching this cpp file in the cpp file. This isn't essential for functions, but things like defining classes and structs is usually done in the header file and one needs the declaration of a class when one is writing functions that act on them, so one generally includes a header file in the cpp file that goes with that header file as well as other files that want to use functions and structures declared in the header file.

Next, we want to go to the header file. It isn't very complex, but has a couple of new elements

```
#ifndef FUNCTION_H
#define FUNCTION_H
  std::string my_function(double d);
#endif
```

function.h

The first thing to note is the **declaration** of the function **my_function** this has to match the return type, name and parameters of the function **definition** in the function.cpp file, but should not have the curly brackets or the implementation of the function. It should end after closing the round brackets for the function parameters and be followed with a semi colon. You can have as many of these function **declarations** as you want - feel free to try it! Just copy and paste the definition of my_function as many times as you want and the compiler won't complain at all. But you can only have a single **definition** of the function. Try to copy the definition in the cpp file and the compiler will complain.

A couple of brief notes before we go on

1. You will sometimes see that the declarations of functions don't have the names for the parameters in the round brackets after the function name, just the type names. This is perfectly OK. All that the compiler needs is to know the types of the parameters to the function to allow it to generate code to call that function. The names of the parameters are only needed in the definition of the function so that you can actually use the parameters in the function by name (if you want to scare yourself, look up variadic functions in C - you can access parameters passed to a function without them having names!)
2. You will sometimes see functions defined in header files as well as declared there. This is also fine. Because header files are pasted into files where they are used this is effectively the same as declaring and defining the function on the same line and it can be a popular approach to creating what are called **header only libraries** which are a popular way of shipping certain types of library code. This works but you often run into the **one definition rule**. If you include a header file with function definitions in it into two different cpp files which are both compiled as part of the same program then the compiler has no way of knowing that the apparent two definitions of the function are actually the same function in two places. There is a special keyword **inline** that you can use to tell the compiler that you absolutely guarantee to it that every copy of a function that it encounters is actually just another copy of the same function, but it has to be used with care because you are effectively telling the compiler to turn off some safety checks because you guarantee that they aren't needed! Similarly, for some special types of functions (which we will come to in the actual course) this **inline** behaviour is assumed and we'll discuss why in the course. If you put in the inline keyword but actually define two different versions of a function then there is no guarantee about which version of the function will be called at any given place and bad things are almost guaranteed to happen.

The final new element is the three lines that begin with **#.** These lines will look vaguely familiar because the **#include** directive that we have already met looks similar, and indeed the **#** at the start does mean that they are processed by the same part of the compiler chain - a thing called the **C preprocessor** (it is worth noting here that while the C preprocessor was originally from the C language, many languages including C++ make use if it nowadays and all still call it the C preprocessor).

The C preprocessor runs before the compiler sees the source code and is responsible for things like pasting in the content of header files in response to **#include** directives, but is is actually much more powerful. The preprocessor it its own language and doesn't know anything about C++ source code. It reads its own special code and has its own variables. The final result of the preprocessor effectively a new text file that is handed to the compiler. In this case those lines are called **include guards**. The three lines can be read as follows

1. **#ifndef FUNCTION_H** - If the preprocessor variable **FUNCTION_H** has **not** been given a value then pass the code between this and the matching **#endif** command to the compiler to compile. Note that this is if**n**def i.e. if not defined. There is also **#ifdef** which passes the code through if a preprocessor variable is defined. There are quite a few commands in these preprocessor conditionals. For example you will sometimes see **#if !defined(FUNCTION_H)** as an alternative to **#ifndef FUNCTION_H**
2. **#define FUNCTION_H** - Give the preprocessor variable **FUNCTION_H** a value. In this case, it is given an arbitrary value, but not any particular value. **#define FUNCTION_H 1** would give it the value of 1
3. **#endif** - End if matching **#ifndef FUNCTION_H**

The net result of this set of three commands is ***that only the first time that the compiler #includes a given header file when compiling a given cpp file is the contents of the header file pasted in***. It is important to note both of these conditions. Including a header file in two different cpp files will cause the header to be included in both of the cpp files, but including it twice in a single cpp file will not cause the header to be included twice. Why would you include a header file twice? Well, sometimes by accident but more often only **indirectly**. That is by including a header file that itself includes a header file. So, why would a header file include a header file? For various reasons, but mostly because you want to use a class or struct that is defined in a header file when defining your functions. For an example, imagine a different version of function.h where my_function returns not a double precision number but a string. C++ has various ways of representing strings, including the old C style **char*** character array system, but many C++ programmers use a string class called **std::string**. std::string is a class i.e. a definition of an object, so you create instances of it as you would other variables. Unlike simple variables, but like many other things in C++ std::string is defined in a header file, specifically one called **string**.

```
#ifndef DIFFERENT_FUNCTION_H
#define DIFFERENT_FUNCTION_H
#include <string>
  std::string my_function(double d);
#endif
```

different_function.h

Above we see what a header file defining a function that returns an std::string would look like. The only substantial change is that the return type is changed to **std::string** and that we are now including the **string** built in header. Because header files are literally just pasted in place by the preprocessor we **have** to include the header file before we can use std::string otherwise the compiler has no idea what an std::string is! So the rule generally is that you include a header file whenever you want to use the types or functions defined in that header file. But what about if you both want to use std::string directly and make use of my_function? In which case by including both files you will have two instances of **#include<string>**, one directly and one when the contents of **different_function.h** are pasted in. Since the one definition rule only allows a single definition of the std::string class (for each file that is being compiled) this would cause a problem. The **string** header has include guards much like the ones that we just saw in ensure that the second time that **#include<string>** is encountered a suitable preprocessor variable has been defined and the content of the file is not included a second time. Before leaving include guards there are two final elements to remember

1. Preprocessor variables defined by **#define** statements are only defined while processing each individual cpp file. If you are compiling multiple files then a value given by **#define** only applies to files that encounter that **#define** statement. Pretty much all compilers let you specify a flag on the compile line to define a preprocessor variable to have a given value and that will apply to all files that are compiled with a compiler line containing that flag. There is no standard way to ensure that a file is only included once across all cpp files that make up a project.

2. The name of the preprocessor variables used in the include guard are arbitrary, but they must be unique for each header file in the project. The standard header files like **string** or **iostream** have unique enough include guard variables that you do not really have to worry about colliding with them accidentally (i.e. **string** in the gcc compiler uses the variable **_GLIBCXX_STRING** for its include guard), but you need them to be unique across your

code. The approach that I used above is the one I prefer. Use the name of the header file appended with _H. That is not a rule, and there are cases where it will not be sufficient (imagine where you have a lot of subdirectories of your source code, all with a header file called "defines.h" for example), but so long as you can make the variable unique anything is fine. Variable names must be made up of only alphabetic characters a-z in either upper or lower case, numbers 0-9 and the underscore character. Some C preprocessors will permit other characters but you shouldn't use any others if you want your code to work on all compilers.

We said that you can define functions as many times as you want, but have to only declare them once. Since you put function definitions in headers, why do you need include guards? We have said that it is because you can't have multiple definitions of classes, but why not? Well, first lets go back to functions

```
#ifndef FUNCTIONS_HEADER_H
#define FUNCTIONS_HEADER_H
#include <string>
  double my_function(double d);
  std::string my_function(std::string s);
#endif
```

functions.h

The above looks like it should be an example of inconsistency that should cause problems, but it actually isn't. In C all functions have to have unique names and this maps onto a low level feature of how most operating systems work requiring that all functions within the program must have unique **symbol names**. In C++ this requirement is relaxed **a lot**. In C++ you can have multiple different functions with the same name so long as the functions have different parameters. This is called **function overloading** and is one of the very powerful features of C++. When you call the function the compiler will use the parameters that you actually call the function with to determine which version of the function to call, so in this case calling **my_function(1.0)** will call the double version and **my_function("Hello world!")** will call the string version. There are quite a few restrictions on exactly what can work, especially when you don't call a function with **exactly** the type that is required (so for example, if you have **my_function(int,long)** and **my_function(long,int)** and try to call it with two **int**s it won't be able to decide because neither is a better fit, even though they are valid overloads and if you called them with an int and a long it would select one function or the other based on the order.) but the main thing to remember is that the compiler can **only** select between overloaded functions by the number and type of the parameters to the function. In particular, it cannot select based on the return type of the function.

So the compiler has very few cases of truly inconsistent function declarations, mainly functions having the same name, same parameters but different return types, and when the compiler encounters those two functions it will pretty much always fail to compile your code unless you try to trick it in some way. This makes sense, all that the

Name Mangling

If you are wondering how C++ gets around the OS requirement that symbols be unique it is by a process called **name mangling** which combines the name that you give a function with the types of the parameters to the function (and a few other things) to produce the name of the symbol. The symbols in the final file are always unique. Other languages use the same trick - this is why subprograms in Fortran can have the same name in different modules.

compiler needs to do to call a function is to know what parameters to pack up to call it, what to do with the return value and where in memory to jump to find the machine code instructions for the function (that is what the unique symbol names are used for!), so as long as it doesn't encounter any inconsistent declarations it just keeps on ticking along until it has finished compilation. So long as you only have one cpp file defining the **definition** of the function then you won't have any problems (if you do have two **definitions** then the compiler might get to the very last stage of putting all of the bits of the code together, called the **linking** step, and then find that it has **duplicate symbols** which will cause it to fail to compile your code. If you get a **duplicate symbol** or **duplicate function** error in any language look for multiple definitions of the same function).

This approach could in theory work for declarations of **classes** and **structs** but is not allowed to do so. Classes and structs will be described in the next section but they basically combine several pieces of data together into a single data structure and are a very common feature of C++. Unlike with a function, which version of a class or struct is to be used cannot be determined by parameters, since there are no parameters - they act just like variables. This means that multiple structs defined with the same name would either be trivially identical (i.e. redefinition of a structure storing the same data in the same order) or would be an invalid redefinition. While in theory C++ could allow the trivial redefinition it would require more work from the compiler to check that it is a trivial redefinition so it is not permitted. This explains what the include guards are mostly there to do - prevent a given struct or class being defined multiple times due to the header file being included multiple times.

```
#include <iostream>
#include "function.h"

int main(){
  std::cout << my_function(1.234) << "\n";
}
```

main.cpp

Finally after all of this we have *almost* all of the elements to actually compile a C++ code with multiple files in it - we have defined a function in a second cpp file and declared it in a header file, now we just need to define the code that implements **main** so that the program can actually run, and then show how to compile the code. Well, main is very simple. All that is needed in main is to include the "function.h" header, create a main function the same as before and then call the function by name. There is no difference in how I call the function because it is defined in another file.

Compiling codes consisting of multiple files is one of those problems that is as complicated as you want to make it. The simplest way is to simply put all of the cpp files on a single line call to your compiler

```
g++ main.cpp function.cpp -o program
```

You don't need to put header files in the compile line because they are included in the cpp files by the preprocessor. Unlike in some languages, you don't have to put the .cpp files in the compile line in any particular order. Any order will do.

```
g++ -c function.cpp
```

Sometimes you want to compile the source files one by one. There are various reasons for this, but probably the most common one is to allow you to only recompile files that have changed since the last time the code was compiled. There are tools, such as **make** that help with this, but the core of all of them is the idea of compiling your code file by file. You can compile any given file to an intermediate file called an **object file**. An object file contains your source code transformed into machine code that the CPU can run, but without it being linked together into a runnable program. This means that you can convert each source code file into an object file individually without needing to know about other source files. The command is very similar to the normal compile command

Simply add the "-c" flag to your compile line and remove the "-o" flag. You don't need the "-o" flag because the compiler will chose a sensible output name when compiling object files - the name of the cpp file, but with the file extension change from **.cpp** to **.o**. Once you have compiled all of your cpp files to o files, you can then simply compile them together into a single executable

```
g++ main.o function.o -o program
```

Once again, the object files don't need to be put in any particular order when performing this final stage of compilation.

```
g++ main.cpp function.o -o program
```

It is also possible to combine directly compiling cpp files with object files, so the following is also a fairly common idiom.

Combining the final cpp file containing **main** and one (or more) .o files like this is also fairly common.

Structs and classes

The idea of classes and structs is a mix of the simple, the slightly more complex and the quite sophisticated. Here we're just going to talk about the first one, but we'll mention all three

1. Classes and structs bundle together different variables into a single object. This is useful when you have something that is described by several values that make no sense except when you have all of them. A good example would be a particle moving in space. It has x, y and z position values and x, y and z velocity values (it may also have acceleration etc. but we'll stop at position and velocity). Generally you will always want to think of a particle having all of those properties so you'll want to put them together. That is the simplest level of what a **struct** or **class** does. Once you have created your particle class (which defines the specification for what a struct or class looks like), you can create as many **instances** of that class as you like to actually hold data, in pretty much the same way that you create integers or doubles. Doing this basically creates your own type which you can use wherever you would use a normal type declaration.

2. You can also attach functions (typically called **methods**) to a class or struct. Unlike normal functions, you don't just call the function, you call the function attached to a specific instance of a class or struct and it is called knowing which instance of the class or struct it has been called on. We'll cover this properly in the actual course

3. You can create **inheritance chains**. This is an important element of object oriented programming. You create new classes that inherit behaviour from already defined classes. The reasons behind this inheritance are often quite complicated, but often it is to create specialisations. So you would implement a **road_vehicle** class and then create **bus**, **car**, and **bicycle** classes that are specialisations inheriting from **road_vehicle**. You could then write code that would take a general **road_vehicle** class but be happy being handed a bus, car or bicycle. We're not going to be covering object oriented programming really in this course, but it is worth knowing that this is a thing in case you see it.

```
#ifndef STRUCT_H
#define STRUCT_H
  struct particle{
    double x, y, z;
    double vx, vy, vz;
  };
#endif
```

struct.h

The above code for **struct.h** shows a header file that defines a struct. You would create an instance of it by just typing **particle P;** in a function and that would create a particle struct called **P**, just as typing **int I;** would create an integer called **I**. This is shown below

```
#include <iostream>
#include "struct.h"

int main(){
  particle p;
  p.x=14.0;
  p.vx=17.4;

  std::cout << p.x << "\n";
}
```

struct.cpp

You generally want to define a struct in a header file because this definition is needed whenever this struct is used, so you include this header file whenever you need to use it. But what would happen if you have two definitions of it?

```
#ifndef STRUCT_H
#define STRUCT_H
  struct particle{
    double x, y, z;
    double vx, vy, vz;
  };
  struct particle{
    float x, y, z;
    float vx, vy, vz;
  };

#endif
```

2struct.cpp

This is an example of two structs with the same name but only trivial changes. The question now is "how could the compiler tell which of these you mean?", and the answer is that it can't. With a function, the parameters that you pass to a function can, at least sometimes, uniquely identify which version of a function you want, but with a struct there is no information that is given that would allow you to select which one you want. As it happens C++ does actually have a mechanism called **templating** that would allow you to have two structs that differ only in type like this and we cover this briefly in the course, but without templating, there is nothing in the way that you declare an instance of particle that would tell you which one you want. This is why the ***definition*** of a class or struct has to be unique - there is no way to choose which one of multiple definitions you want. As we already mentioned, not allowing identical redefinitions of a structure is an arbitrary choice in for the language, but it is no loss since the redefinition would be redundant. The include guards ensure that you only get a struct or class in a header file declared once no matter how many times you include the header file.

Strictly it is possible to define a struct or class with the same name but different definitions in different "**translation units**", but it is generally unhelpful to have two different structs or classes hanging around in a single program, so don't do it!

We've been saying "class or struct" all through this section, but then just showed a struct, so the immediate question is : is there a difference? The answer is "not really". **struct** is a keyword that C++ inherited from its background as a derivative of C and it is used in much the same was as it is in C (in fact it can be used in exactly the same way as it is in C, but there is also the slightly easier syntax shown above for declaring instances of structs in C++ which we're going to be using in this course). **class** is a new keyword in C++ and while it has other meanings in the language, it can be used directly in places of **struct** in all cases in C++. There is one and only one difference between them. When you define a struct or class in C++ you can flag some of the member variables as being either public or private. Private variables can only be used by methods of the class (those functions that I said could be connected to classes and structs). Public variables can be accessed by any code, as you can see us doing in **struct.cpp** above using the dot (.) operator to access the internal elements of the struct. This is then the only difference between struct and class in C++ - structs are default public, classes are default private. You can alter either of them to have both public and private members but for the kind of data storage that we show above, struct is more common than class because of this default public variable behaviour. We will cover this in more detail in the actual course