

# Session 1 - An Introduction To C

---

The course starts gently by introducing basic programming concepts in C such as variables, types, functions, and loops. If you're an experienced programmer then feel free to skim through these sections to get to the more C-specific parts!

I would personally recommend doing all the exercises regardless of your level, to ensure that you know the material. At times, my writing can be verbose so please do skim through the content, scan for bold/italicised text, and carefully read content when you cannot do the corresponding exercises.

## Contents

- What is C
- Hello World!
- Compiling & Running C Code
  - Compiling and Hello World Exercises
- Variable Declaration & Initialisation
- Int
- Format Specifiers (**%d**, **%s**, **%c**)
  - Int, Char, and Format Specifier Exercises
- Comments (and increment operator)
- More Integer Types (char, short, int, long, long long)
- underflow/overflow
- use of stdint library
- unsigned integers
- floating-point types
- Functions
- Basic Loops
- Arrays (& undefined behaviour)

## What is C

C is a general purpose programming language that originated as the language for UNIX ....

## Hello, World!

Let's start with the first program almost *everyone* writes.

```
#include <stdio.h>

main(){
    printf("hello world");
}
```

There are **3** main things to note already:

1. The **stdio** library is the **standard input/output** library which we'll use to print text to the console. Think of **libraries** as a collection of tools that other people have coded so that we don't have to; typically a library is dedicated to *one thing* and I'm sure you can guess what **stdio** is dedicated to.

We import it using the **include** *preprocessor directive*; don't worry, we'll learn more about what preprocessor directives are later! Just know that preprocessor directives start with a **#** (fun fact: also called an octothorpe), and that **#include <stdio.h>** tells C to copy all the code from stdio so that we get access to **printf** and everything else in that library.

2. All C programs start at the **main** function; not having a main will lead to an error!

```
undefined reference to 'WinMain'
```

*The error I receive on my machine*

It is similar to Java's main, which you may have already learned about in CS118, in that it is the first block of code run.

If you don't know what functions are yet, then just think of them as blocks of code. We will discuss them soon!

3. Statements in C, like using the **printf** function, end with a *semicolon*. This helps C read the program as semicolons tell it where one statement ends and another begins.

Okay so we've seen our first C program but how do we actually run it?

## Running C Code

If you're working on DCS machines, then you should have the GNU C compiler installed.

If not then you should check by running the following command:

```
gcc --version
```

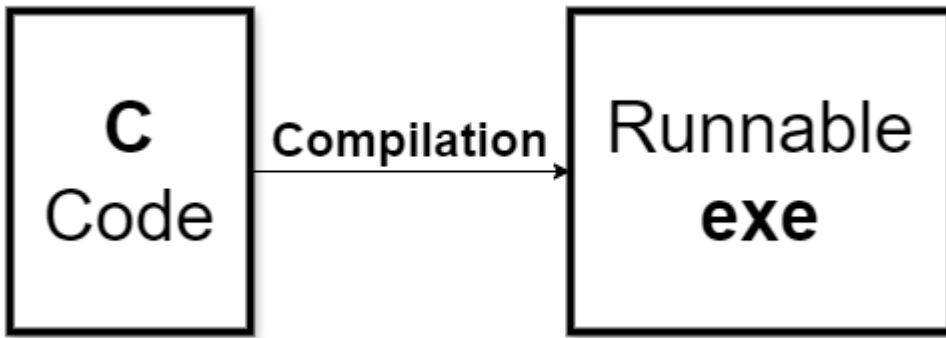
*gcc stands for GNU Compiler Collection*

If you do not have it installed, then follow the installation instructions from [here](#).

### What is a compiler?

A compiler turns **high-level** code (such as **C**) into **low-level** machine code that the computer can run.

The idea is that **C** code is relatively portable so you can write some **C** code, give that code to multiple different machines and they'll be able to run it using their respective compilers. In practice, since **C** often talks "directly" to Operating Systems, you will often have to write slightly different **C** code for different machines for complex programs.



- Compiling C code turns it into an exe
- You can run the exe with `./TheExe`

## Compiling and Executing C Code

Use `gcc` to compile C code. As an example:

1. create a new file directly under the main folder (named C\_Course-Introduction)
2. Call it `exampleToCompile.c`
3. Run the command `gcc exampleToCompile.c -o exampleProgram`

This should give you an exe named `exampleProgram.exe` that you can run using:

```
./exampleProgram
```

Note that the `-o` flag lets you name the exe and, by extension, place it in a specific folder. If you wrote `exes/exampleProgram` instead of `exampleProgram`, then it would place the newly created program in the `exes` folder (assuming this folder exists). This will make it easier to keep track of your files!

## Exercises for Hello World & Compiling

1. Compile one of the c files inside of `cFiles` (**hint:** use the [last paragraph of Compiling and Executing](#) to help you)
2. Try changing up the text outputted to the console. Note that `C` does not automatically insert *newlines* after each usage of `printf`!

```
#include <stdio.h>

main(){
    printf("hello ");
    printf("world");
}
```

So the above code outputs:

```
hello world
```

Key thing to note is that it's all on one line

If you want a newline, use `\n` which stands for the *newline* character.

```
#include <stdio.h>

main(){
    printf("hello\n");
    printf("world");
}
```

Have fun writing your own output!

## Variables

So we can interact with the console using `printf` but we need a way of handling data; at the very least, we need a way of storing input when we start reading input from the user.

### Enter variables!

Variables have:

- a **name**
- a **type** ( which tell you what kind of data is stored)
- a **value**, regardless of if you provide one!

Declaring with value

```
type variableName = initialValue;
```

Declaring without value

```
type variableName;
```

This is the difference between **declaration** and **initialisation**.

Both of these **declare** the variable (they tell **C** that memory must be reserved for the variable), but only the first **initialises** the variable with a value.

We will see soon that the variable that isn't initialised *does* have a value, but it might not be what you expect!

## Integer variables

First, we'll explore the integer (positive & negative whole number) types!

```
#include <stdio.h>
```

```
main(){
    int a = 5;
    int b, c;

    b = 1;
    c = -4;

    printf("%d\n", a + b + c);
}
```

*A program with a variety of int initialisations*

2

1. Note the different ways we've initialised the variables a,b,c- all valid!
2. Note the use of `+` as an operation for integers (and numbers in general). You also get `-` for subtraction, `*` for multiplication, `/` for division.
3. Notice that we're using `printf` differently. The previous use of `printf` was just passing it a piece of text; this current use of `printf` passes it a formatting piece of text (`%d\n`), and the values to slot into the format.

As it's useful to output values in meaningful ways for debugging complex programs and even outputs for simple command-line programs, we'll take a closer look at using `printf`.

## Format Specifiers (using printf)

The formatting piece of text includes **format specifiers** which begin with `%` and are replaced by the other values you pass `printf`; different values (or different ways of displaying values) require different format specifiers.

For example, `%d` is used to stand in for a regular old base 10 integer (**d** for **d**ecimal). The pieces of text that we've been passing are called **strings** and so use `%s`.

```
#include <stdio.h>

int main(){
    int hours = 11;
    int minutes = 23;
    int seconds = 59;

    printf("%s - %d : %d : %d \n", "The Time", hours, minutes, seconds);
}
```

*Example for time using both `%s` and `%d`*

What did I mean by “**different displays of values**”? To explain, I will introduce the **char** which is a *smaller int* data type, with a *tiny* range of possible values (-128 to +127), that is typically used to represent *characters* using **ASCII** rather than to represent integers.

For those who don't know, ASCII is a standard encoding of *integers* to commonly used *characters*. To show a few:

<b>Integer</b>	65	66	67	97	98	99	33	35
<b>Character</b>	A	B	C	a	b	c	!	#

Of course the **digits** must be encoded too; they're encoded by the numbers **48-57**.

<b>Integer</b>	48	49	50	51	52	53	54	55	56	57
<b>Character</b>	0	1	2	3	4	5	6	7	8	9

So a char can be displayed as a *number* using **%d**, but also as a *character* using **%c**.

```
char a = 101;
printf("%c : %d", a, a);
```

*Two different ways of displaying the same value: as the character the char represents, and the corresponding number itself*

```
e : 101
```

As a **char** can represent a fraction of the values an **int** can, an int can also be used to store and display a character since it can be converted into a char very easily. So the following code is also valid:

```
int b = 100;
printf("%c : %d", b, b);
```

*Displaying both representations using an int instead of a char*

```
d : 100
```

If you're concerned about not being able to display the **%** symbol, fret not! You can do this using **%%** which displays a single **%**.

## Exercises for using ints and chars

1. There are two groups in a room: 6 UWCS Exec and 20 C Programmers. 4 people in this room are both UWCS Exec and C Programmers. Finish the following C program in order to complete the total number of people in the room:

```
#include <stdio.h>

main(){
    int exec = 6;
    int cProgrammers = 20;
    int both = 4;

    int total;

    printf("The total number of people: %d", total);
}
```

2. Try to spell your name out using chars! If you're feeling ambitious, try doing this using a mixture of capital and lower case letters. Refer to [this if needed](#).
3. Timmy wants to display how charged an attack is in his game! Can you help him output the text:

```
Laser is charging, at 57% now!
```

His current code is:

```
#include <stdio.h>

main(){
    int charge = 57;

    printf("FILL ME IN");
}
```

## Break from ints: Comments

We'll be using **comments** to help *describe* and *explain* bits of code; they don't do anything to the execution of the program, but you should use comments in your own **C** programs to help the reader quickly understand what is going on (often you in the future)!

When I wrote this code,  
only god & I understood what it did.



Now... only god knows.

Use `//` for single-line comments. Let's see how they're used to showcase the *increment* operator (`++`), which increases a number by 1.

```
main(){
    int i = 5;
    i++;

    printf("%d\n", i); // output: 6

    i++;
    i++;

    printf("%d\n", i); // output: 8
}
```

#### *Use of single-line comments*

If you need to write something longer, then consider using **multi-line comments**.

To start a multi-line comment, use `/*`

To end a multi-line comment, use `*/`

```
#include <stdio.h>

/*
    The main function is the entry-point to the program.
    This program shows the increment operator on integers, and its effect.
    The main purpose of this program is to demonstrate the use of comments.

    (This is a multiline comment)
*/
main(){
    int i = 5;
    i++;
```



```

printf("%d\n", i); // output: 6

i++;
i++;

printf("%d\n", i); // output: 8
}

```

The full program with a multi-line comment describing main

Try to practice using **single-line** and **multi-line** comments in future exercises to get used to them!

## Which int to use?

There are several **ints** (char, short, int, long, long long) to choose from, and generally you should choose them based on *what they're used for* e.g. *chars* for representing *characters*, *shorts* for *small integers*, *longs* for *large integers*.

For most use cases, the size taken up by an integer is negligible so most people tend to default to an int in a place where a short would be perfectly fine. This only really matters in situations where space is very valuable such as small embedded systems or storing **a lot** of integers. In fact, it can be beneficial to use an int as your future uses may require a larger range and arithmetic can actually be faster (although usually negligibly) if your machine's native size is 32-bit or 64-bit.

Despite the idea of **C** being a portable language, the size (and range) of integer types are machine-dependent; therefore, the **C** standard only specifies the *minimum* number of bytes for each type.

For instance, the minimum number of bytes for an **int** is 2 however most modern machines use 4.

Type	(Minimum) number of bytes	(Minimum) range of values
char	1	-128 to +127
short (aka short int)	2	-32768 to +32767
int	2	-32768 to +32767
long (aka long int)	4	-2147483648 to +2147483647
long long (aka long long int)	8	$-2^{63}$ to $+2^{63} - 1$

Overflow/Underflow - What if the number goes above/below the range?

Going above the range (overflow) of an int type will wrap around to the minimum value.

Similarly, going below the range (underflow) of an int type will wrap around to the maximum value.

(image: Integer overflow and underflow) Note that the decrement operator (`—`) subtracts 1 from a number.

Overflow/Underflow - Type Casting

When we saw that integers can be displayed as characters, we were actually converting integers to characters first: this is a type cast.

What if I want only positive numbers?

The standard integers we use are signed integers. As in, they have a sign (positive or negative). If you want to use only the positive range, then you can use unsigned integers by adding "unsigned" in the type!

What if I want a guaranteed size?

If the size of the int is crucial, such as in small embedded systems, you can use the stdint library.

Type

Number of bytes

Range of values

int8\_t

1

$-2^7$  to  $2^7 - 1$

int16\_t

2

$-2^{15}$  to  $2^{15} - 1$

int32\_t

4

$-2^{31}$  to  $2^{31} - 1$

uint8\_t

1

0 to  $2^8 - 1$

uint16\_t

2

0 to  $2^{16} - 1$

uint32\_t

4

0 to  $2^{32} - 1$

Most modern implementations also provide int64\_t and uint64\_t.

Using decimals: floats

Functions

Good code is often reusable and easily adaptable code.

Arrays

Next Session...

We'll be covering arguably the most important and famous topic in C: pointers.