Introduction to C

EPCC HPC Summer School 2025

William Lucas

EPCC

Contributors

In this material:

- The slides are created using LaTeX *Beamer* available at `https://ctan.org/pkg/beamer`.

License - Creative Commons BY-NC-SA-4.0[1]

[1]You can find the full documentation about this license at: https://creativecommons.org/licenses/by-nc-sa/4.0/

Prerequisites

The content presented in this material assumes that:

- You know how to **open a terminal** on your machine, and are familiar with **essential shell commands** such as moving through file hierarchies.

Nota bene

Format

This set of slides is designed to be read as a **stand-alone document** for self-study purpose, and thus contains a lot of text and explanations. It is **less suitable** to be delivered as a live presentation.

Table of Contents

Structure

- The content presented in this topic is designed in small units that typically take **15 minutes of less**.
- Each unit is made of:
  - A few slides introducing a new technique or concept.
  - A short exercise illustrating this technique or concept.

Structure

- This approach using units makes sure you get to learn **and practice** every technique in turn.
- At the end, you will have one final exercise that will make use of everything you will have learned.
- Do not worry, they are guidelines and directions throughout the entire coding refresher.

Getting ready

- The first step will be to acquire the exercises.
- You can find them in the archive available on Learn, next to the slides.

Table of Contents

Declaration

In the C programming language:

- Variables cannot be used before being declared.
- Variables must be assigned a data type on declaration.
- A variable cannot change its data type.
- Like all statements in C, variable declarations must end with a semi-colon '`;`'.

Structure of a declaration:

```
type_of_variable name_of_variable;
```

Variable names

- Variable names are *case-sensitive*: lowercase and uppercase letters are not treated as identical.
  - Variables '`my_var`' and '`MY_VAR`' are distinct.
- The use of **digits** in variable names is allowed **except** for the first character.
  - Valid: `my_var_1`
  - Invalid: `1_my_var`
- Variable names can contain underscores '_', but **no other** special characters.
  - Valid: `my_var_123`
  - Invalid: `my-var-123`

Basic data types

`int`	A standard signed integer
`unsigned int`	A standard unsigned integer
`float`	A floating-point number
`double`	A double-precision floating-point number
`bool`[2]	A boolean

[2] Requires the `stdbool.h` library

Examples of declarations

```c
int a;
unsigned int b;
float c;
double d;
bool e;
```

Assignment

- Variable assignments use the operator ‘=’.
- Variables can be assigned a value upon declaration.
- Variable declarations and variable assignments, like all statements in C, must end with a semi-colon ‘`;`’.

Examples of declarations and assignments

```c
int a = -123; // Declaration + assignment
a = 456; // Assignment

float b = 0.456;
b = 0.789;

double c = 0.456;
c = 0.789;

bool d = true;
d = false;
```

Time to practise: 1.Variables

In this exercise, you will:

- Declare variables, using different types.
- Initialise variables.

Table of Contents

What is it?

- In C, like other programming languages, you have operators that allow you to have different interactions with your variables.
- The most basic ones are **arithmetic operators**, which allow you to perform arithmetic operations on your variables.

Arithmetic operators

- Addition

```c
a = b + c
```

- Subtraction

```c
a = b - c
```

- Multiplication

```c
a = b * c
```

- Division

```c
a = b / c
```

- Modulo

```c
a = b % c
```

Concatenated

- Sometimes, we want to store the result of an arithmetic operation in one of the variables part of the said operation. For example:

```
a = a * 20
b = b - c
```

- For convenience, in C, there are *compound assignment operators*, which embed the arithmetic operation in the assignment.

Semantics

Compound assignment operators are syntactic sugar, however, they bring no additional functionality.

Compound assignment operators

- Addition =+

```
a += b; // Equivalent to a = a + b;
```

- Subtraction −=

```
a -= b; // Equivalent to a = a - b;
```

- Product ∗=

```
a *= b; // Equivalent to a = a * b;
```

- Division /=

```
a /= b; // Equivalent to a = a / b;
```

- Modulo %=

```
a %= b; // Equivalent to a = a % b;
```

Compound assignment operators

Another typical situation, in loops in particular, is to increment or decrement a variable. For instance:

```
a = a + 1; // Incrementing
a = a - 1; // Decrementing
```

Each operation has its compound assignment operator:

- Increment ++

```
a++; // Equivalent to a = a + 1;
```

- Decrement --

```
a--; // Equivalent to a = a - 1;
```

Time to practise: 2.Operators

In this exercise, you will:

- Practice the assignment operator.
- Practice arithmetic operators.
- Practice compound assignment operators.

Table of Contents

Conditional execution

- Sometimes, a certain part of the code is meant to be executed only if a given condition is satisfied.
- To this end, we use conditional statements or branches.

How it works

- Create a conditional statement using the `if` keyword.
- Provide a condition.
- Provide a code block to execute if the condition is met.
- Provide ore or more alternative branch(es) using `else if` statements.
- Provide a `else` statement that will be executed if all other `if` and `else if` statements did not execute.
- A conditional statement can be made only of a single `if`, without any `else if` or `else` blocks.

Example

- An `if` statement alone.

```c
if(condition1)
{
}
```

- An `if` statement with an `else` statement.

```c
if(condition1)
{
}
else
{
}
```

Example

- An `if` statement with `else if` statements, and an `else` statement.

```c
if(condition1)
{
}
else if(condition2)
{
}
else if(condition3)
{
}
else
{
}
```

Comparison operators

- In addition to assignment and arithmetic operators, there are *comparison operators*: they evaluate an expression and **return a boolean** value.
- Comparison operators are typically found in **conditional statements** (i.e.: `if` and `else if`).

Comparison operators

- Equal to ==

```c
if(a == b)
```

- Not equal to !=

```c
if(a != b)
```

Caution

Note the double '=' for the equal to symbol. If you use only a single '=', it becomes an assignment and, without going into details, will still somehow evaluate to a boolean eventually but certainly not the value you were expecting.

Comparison operators

- Strictly lesser than <

```
if(a < b)
```

- Lesser than or equal to <=

```
if(a <= b)
```

- Strictly greater than >

```
if(a > b)
```

- Greater than or equal to >=

```
if(a >= b)
```

Logical operators

To combine your conditions, you will need *logical operators*.

- Logical `AND` `&&`

```c
if(a==2 && b<3) // true if both a==2 and b<3
```

- Logical `OR` `||`

```c
if(a==2 || b<3) // true if a==2, b<3 or both
```

- Logical `NOT` `!`

```c
if(!(a==2 && b<3)) // true if a!=2, b>=3 or both
```

Caution

Operators `&&` and `||` are made of **two symbols**. Without going into details, using a single character will still compile but do something totally different than what you expect.

Time to practise: 3.Branching

In this exercise, you will:

- Create branches with "`if`", "`else if`" and "`else`" blocks.
- Use comparison operators.
- Use logical operators.

Table of Contents

What is it?

- Loops are structure that allow to repeat the execution of a block for as long as a given condition holds.
- There are multiple loop structures in C, the two main ones are:
  - `while`
  - `for`

While loop

- The *while* loop takes a code block, which it will execute for as long as the condition given holds.

```
while(condition)
{
    // Code to repeat.
}
```

A classic scenario

- A classic situation is a loop that executes for a given number of iterations.
- It always comes down to the following structure:

```c
int max_iterations = 10;
int current_iteration = 0;
while(current_iteration < max_iterations)
{
    // Code to repeat.
    current_iteration++;
}
```

The `for` loop

- Another loop structure is available, as a syntactic convenience: the `for` loop.
- The `for` loop embeds "everything" in its structure, separated with semi-colon ';':
  - Part A The declaration and/or initialisation of the iterator.
  - Part B The condition that will indicate whether to continue the loop.
  - Part C The modification to apply to the iterator at the end of every iteration.

```c
for(<part A>; <part B>; <part C>)
{
    // Code to repeat.
}
```

The `for` loop

- You are free to declare the iterator before the loop, or inside it:
  - Declaring the iterator before the loop

```c
int i;
for(i = 0; i < some_value; i++)
{
    // Code to repeat.
}
```

  - Declaring the iterator in the loop

```c
for(int i = 0; i < some_value; i++)
{
    // Code to repeat.
}
```

Loops execution flow

- During the execution of a loop, there are two ways with which you can interfere with the execution flow:
  - `continue`
  - `break`

Skipping an iteration

`continue` Interrupts the current iteration and goes directly to the condition check phase for the next iteration.

```c
while(conditionA)
{
    if(conditionB)
    {
        continue;
    }
    // Section not executed if conditionB holds
}
```

Execution flow - Break loop

`break` Interrupts the current iteration and exits the loop. If there are nested loops, it breaks out only of the innermost loop from which it was called.

```c
while(conditionA)
{
    if(conditionB)
    {
        break;
    }
}
```

Time to practise: 4.Loops

In this exercise, you will:

- Create `for` and `while` loops.
- Use `continue` to skip iterations.
- Use `break` to interrupt the loop execution.

Table of Contents

Printing

- The function `printf` allows you to print text in C.

```c
printf(<formatted string>, <variables>);
```

- To inject a variable, the code depends on the variable type:

`%c`	A character.
`%d`	An integer.
`%f`	A float or a double.
`%p`	A pointer.
`\t`	A tabulation.
`\n`	A line break.

- You must put the variables in the order they are referred to in the format string.
- Requires `#include <stdio.h>` (stands for standard IO).

Example

- Printing just text

```c
printf("Some text, and no variables.\n");
```

- Printing a single variable value

```c
int a = 123;
printf("Value of variable 'a' is %d.\n", a);
```

- Printing multiple variables value

```c
int a = 123;
float b = 456.789;
printf("Value of variable 'a' is %d, and the value
    of variable 'b' is %f.\n", a, b);
```

Structure of a function

- The function `printf` works in an unusual way.[3]
- However, for the vast majority of functions, they rely on a classic structure.

```
return_type function_name(arg_1_type arg_1_name,
                          arg_2_type arg_2_name,
                          ...,
                          arg_n_type arg_n_name)
{
    // Code
}
```

[3] For curious students, see variadic functions.

Define your own - Arguments

- All arguments are **compulsory**; there are no optional arguments.
- Arguments cannot be named like in Python.
- Every argument must be accompanied with its type.
- The variables you receive are “passed by value”, in other words, they are copies. Any modification on them will not change the value of the original variable (you will see how to do this later on).
- Functions cannot be overloaded.

Define your own - About the return

- A function not returning anything is said to have a return type of `void`.
- A function either returns nothing or it returns one value. It cannot return multiple values.
- If a function returns something (i.e.: not `void`), then it must have a `return` keyword somewhere.

Do not forget the `return` keyword

... or “`missing return statement from non-void function`” will be the compiler’s way to remind it to you.

Examples of functions

```
int add_two_ints_together(int a, int b)
{
    return a + b;
}
```

```
void print_int_value(int a)
{
    printf("The int passed has value %d.\n", a);
}
```

Function declaration location

- **For now**, place your functions above the function `main`. If you put them below it, it will not work.
- **Later on**, you will learn how to properly structure a source code, and place your functions anywhere you want.

Time to practise: 5.Functions

In this exercise, you will:

- Create your own functions.

Table of Contents

What is it?

Definition

A *pointer* is a variable whose value is the **memory address** of another variable.

What is it?

- When a pointer points nowhere, we can set its value to `NULL`.
- Pointers have a type too: the type of the variable they point to.
- Pointers can be used to **pass variables by reference** to functions. In other words, making sure the function modifies the original instance of the variable, not a copy of it.

Pointer syntax

Pointers need some new syntax:

- Declare a pointer by placing a `*` symbol after the type it's going to point to.
- The reference operator `&` when placed before a variable name returns that variable's address in memory.
- The dereference operator `*` when placed before a pointer works the opposite way, by returning the value in the memory that is pointed to.

Remember that `*` has two different meanings when it comes to working with pointers.

Example

```c
int a = 123;

// Declare a pointer on an integer
int* pointer_on_a;

// Store the address in it
pointer_on_a = &a;

// Get the value inside the variable pointed.
int b = *pointer_on_a;
```

Example with functions

```c
void increment_value(int a)
{
    a++;
}

void increment_value_via_pointer(int* a)
{
    (*a)++;
}
```

```c
int a = 123;
increment_value(a);
// Variable 'a' is still 123.

increment_value_via_pointer(&a);
// Variable 'a' is now 124.
```

Time to practise: 6.Pointers

In this exercise, you will:

- Declare pointers.
- Get the address of a variable.
- Receive pointer arguments in functions.
- Dereference pointers.

Table of Contents

Arrays

In C, there are two ways to allocate arrays:

- Static allocation
- Dynamic allocation

Statically allocated arrays

- You know their size at **compile time**.
- They are allocated on the **stack**, and therefore are automatically destroyed when going out-of-scope.

Statically allocated arrays

- To declare an array of 10 integers

```c
int my_array[10];
```

- To access the 3rd element in this array

```c
my_array[2]
```

Shouldn't the 3rd element be `[3]`?

In C, indexes start at 0 so the first element is not `[1]` but `[0]`, and the last element is never `[size]` but `[size-1]`.

Dynamically allocated arrays

- Their size is not required until **runtime**.
- They are allocated on the **heap**, where the corresponding memory is not deallocated until explicitly freed by the developer.

Dynamically allocated arrays

To allocate an array, we use the function `malloc`.

```c
void* malloc(size_t size);
```

- It takes the number of **bytes** to allocate. It is encoded as a `size_t`, which is just a type that can hold “large” integers.
- It is not aware of a **data type**.
- It is not aware of a **number of dimensions**.

Tip: finding a type size

You can find the number of bytes that a data type occupies by calling `sizeof(my_type)`.

Dynamically allocated arrays

- The function `malloc` returns a pointer on the first element in the array allocated.
- If the allocation failed, it returns the value `NULL`.
- The function `malloc` does not know the type of data you want to put in that memory, so it returns a pointer on "anything", represented with `void*`.
  - It is good practice to cast it back to a pointer on your type.

Example: to allocate an array of 10 integers

```c
int* my_array = (int*) malloc(sizeof(int) * 10);
my_array[0] = 123;
my_array[1] = 456;
```

Dynamically allocated arrays

- The memory allocated is not deallocated until specified explicitly by the developer using the function `free`.
- The function `free` takes the pointer that was returned by a previous call to `malloc`.

Example: to free a dynamically allocated array of 10 integers

```
int* my_array = (int*) malloc(sizeof(int) * 10);
// Some code
free(my_array);
```

Memory leaks

A *memory leak* is when a dynamically allocated memory is not freed, thus wasting memory resources for the entire duration of your program.

Receiving arrays as function arguments

- As an array

```c
void my_function(int my_array[10])
{
    // Some code
    my_array[0] = 123;
    my_array[1] = 456;
    // ...
}

int main(int argc, char* argv[])
{
    int array[10];
    my_function(array);
}
```

Receiving arrays as function arguments

- As an pointer

```c
void my_function(int* my_array)
{
    // Some code
    my_array[0] = 123;
    my_array[1] = 456;
    // ...
}

int main(int argc, char* argv[])
{
    int* array = (int*) malloc(sizeof(int) * 10);
    my_function(array):
    free(array);
}
```

Statically allocated arrays - Multidimensional

- To declare a 2D array

```c
int my_array[row_count][column_count];
```

- To access the element on the 2nd row, 5th column

```c
my_array[1][4]
```

Multidimensional arrays

- The concept of multidimensional arrays is only in humans brain, because your computer's memory is just one big 1D array.
- When we allocate memory, whether is on the stack or on the heap, it is contiguous (i.e.: there is no hole in it).
- Therefore, consecutive elements are consecutive in memory.
- With multidimensional arrays, programming languages must decide how to lay out dimensions in memory. Consecutive elements in one dimension will be consecutive in memory, while consecutive elements in a different dimension will require jumps in memory.

Multidimensional arrays

- The C language is said to be “row-major order”, it means that elements in the same row are consecutive in memory.
- Regardless of the number of dimensions, this means that elements in the **rightmost dimension** are consecutive in memory.
- The last element of a row, and the first element of the following row are consecutive in memory too. Remember that, in memory, all rows are next to each other, in a big 1D array.

Example:

```c
int my_2D_array[20][10];
my_2D_array[0][0] = 123;
my_2D_array[0][1] = 456;
my_2D_array[0][2] = 789;
// etc...
```

Passing multidimensional arrays as function arguments

```c
void my_function(int my_array[20][10])
{
    my_2D_array[0][0] = 123;
    my_2D_array[0][1] = 456;
    my_2D_array[0][2] = 789;
    // etc...
}

int main(int argc, char* argv[])
{
    int array[20][10];
    my_function(array);
}
```

Time to practise: 7.Arrays

In this exercise, you will:

- Allocate arrays statically
- Allocate arrays dynamically
- Deallocate a dynamically allocated array
- Allocate a multidimensional array
- Pass arrays to functions

Table of Contents

Structures

In C, you can create "compound" datatypes using structure, they allow you to group together variables into a coherent unit. For example:

```c
struct person
{
    float weight;
    float height;
};
```

Structures accessors

To access members of a structure, you need to use the dot '`.`'.

```c
struct person
{
    float weight;
    float height;
};

struct person p;
p.weight = weight;
p.height = height;
```

Structure pointers

If you have a pointer on a structure, say in a function, you can *dereference* it, then access the members.

```c
float get_person_height(struct person* p)
{
    return (*p).height;
}
```

Structure pointers - Syntactic sugar

This `(*my_struct_pointer).member` notation is a bit cumbersome. So, when using a structure pointer, you can use the arrow '`->`' instead.

```c
float get_person_height(struct person* p)
{
    return p->height;
}
```

Only syntactic sugar

The "`x->y`" notation is only for convenience: it does nothing else than converting to "`(*x).y`".

Typedef - Technique

- The `struct` keyword in `struct person` can be redundant.
- You can "rename" the `struct person` in `person_t` using `typedef`.
- This is only a syntactic change, the structure itself is not changed.

Convention

When using `typedef`, you can no longer tell that the type is a structure, so we typically append `_t` to the name instead.

Typedef - Example

In the code below, from line 6, `struct person` and `person_t` can be used interchangeably.

```
 1  typedef struct person
 2  {
 3      float weight;
 4      float height;
 5  } person_t;
 6
 7  person_t record_person(float weight, float height)
 8  {
 9      person_t p;
10      p.weight = weight;
11      p.height = height;
12  }
```

Table of Contents

Decoupling function declaration and definition

- So far, we only defined functions, and had to put them above the function `main`.
- A function declaration provides the interface:
  - The **name** of the function.
  - The **return type** of the function.
  - The **list of arguments** to the function.
- A function declaration ends with a semi-colon ';'.

Example

- A function declaration

```c
void my_function();
```

- A function definition

```c
void my_function()
{
    // Some code
}
```

Location of function definitions

If a function declaration is present before the function `main`, the corresponding function definition can be:

- further down in the same file.
- in a different file (as you will learn in the next unit).

Example

```c
// Function declaration
int add_numbers(int a, int b);

// Main function
int main(int argc, char* argv[])
{
    int total = add_numbers(123, 456);
    printf("Total number is %d.\n", total);
}

// Function definition
int add_numbers(int a, int b)
{
    return a + b;
}
```

Time to practise: 8.FunctionDeclarations

In this exercise, you will:

- Write a function declaration
- Decouple function declaration and definition.

Table of Contents

Source and headers

In C, there are two types of files

Source	Contains functions definitions. Uses ".c" file extension.
Header	Contains functions declarations. Uses ".h" file extension.

Ideally, you want to split your code such that functions are grouped by theme / semantics.

Splitting the code

File my_functions.h

```c
void add_numbers(int a, int b);
```

File my_functions.c

```c
void add_numbers(int a, int b)
{
    return a + b;
}
```

File main.c

```c
int main(int argc, char* argv[])
{
    int total = add_numbers(123, 456);
    printf("Total number is %d.\n", total);
}
```

Including headers

- When your source code is scattered across multiple files, you may call a function that is declared, and defined, in a different file.
- To make sure the compiler knows where to look, you must include the corresponding header.
- To do so, you use the `#include` preprocessing directive.

Example of a multi-file structure

File `my_functions.c`

```c
#include "my_functions.h"
void add_numbers(int a, int b)
{
    return a + b;
}
```

File `main.c`

```c
#include "my_functions.h"
int main(int argc, char* argv[])
{
    int total = add_numbers(123, 456);
    printf("Total number is %d.\n", total);
}
```

Inclusion guard

- When you include a file, the include line is replaced with the content of the file included before the compilation starts.
- You must make sure that a header cannot be included multiple times, or issues of duplicate declarations can arise.
- To guarantee a single inclusion, you can use *preprocessor directives*.

Preprocessor directives

Definition

Preprocessor directive are instructions given to the preprocessor, which treats the source code before the compiler reads it.

Preprocessor directives have three main uses:

- include other files (c.f.: `#include`)
- exclude certain portions of code from compilation.
- automatically replace certain occurrences with literals.

Preprocessor directives as inclusion guards

```c
#ifndef SOME_NAME
    #define SOME_NAME
    void my_function();
#endif
```

- The first time this header is included, the symbol `SOME_NAME` does not exist.
- The `ifndef` directive, which stands for "**if n**ot **def**ined", checks that the symbol `SOME_NAME` does not exist. It is the case here indeed, so it executes the block associated.
- The `define` declares the symbol, and the function is declared in the line that follows.
- From now on, including this header will have no effect because the symbol `SOME_NAME` is now defined and the check done by the directive `ifndef` will systematically fail.

Preprocessor directives as occurrence replacement

- Defines can also assign a value to a symbol.
- Each occurrence of that symbol will be replaced with the value provided.
- This is useful to easily change the size of statically allocated arrays passed through functions without having to update all values one by one manually.

Example

- Example: every occurrence of “SIZE” will be replaced in the source code before it is read by the compiler for compilation.

```c
#define SIZE 100
void my_function(int my_array[SIZE])
{
    my_array[0] = 123;
    my_array[1] = 456;
    // etc...
}

int main(int argc, char* argv[])
{
    int array[SIZE];
    my_function(array);
]
```

Example

- What the compiler receives, once the preprocessor finished its part.

```c
void my_function(int my_array[100])
{
    my_array[0] = 123;
    my_array[1] = 456;
    // etc...
}

int main(int argc, char* argv[])
{
    int array[100];
    my_function(array);
]
```

Multi-file compilation

You now have headers with function declarations, source files with function definitions and your main file. However, you must still compile it all. To do so:

- You must list all sources files.
- You must **not** list header files.
- Assuming you have two source files: `main.c` and `my_functions.c`, and one header `my_functions.h`:

```
gcc -o main main.c my_functions.c
```

Time to practise: 9.MultiFileStructure

In this exercise, you will:

- Place a function declaration in a different file.
- Place a function definition in a different file.
- Cover all includes needed.
- Compile everything, and execute.

Table of Contents

Pagerank

- Now that you have acquired strong foundations of C programming, here is a little challenge.
- Developing a program known as **PageRank**.
- It was originally developed to sort webpages based on popularity, and became the backbone of the Google Search Engine.

Graphs

- Pagerank uses the graph data structure to represent the net.
- A graph is a series of vertices connected with edges, like cities linked with roads.
- In Pagerank, every vertex represents a webpage, and every edge represents a hyperlink from a given webpage to another webpage.
- The above repeats until a termination condition, in this case a maximum number of iterations, is reached.

Algorithm

- Every vertex begins with a value of $\frac{1}{n}$, with n equal to the number of vertices.
- At every iteration, each vertex splits its value equally across all of its neighbours, and adds it to the neighbour's existing value.
- Each neighbour multiplies its value by $(1 - \alpha)$, and then adds α.
- α is the *dump factor*: it is used to control how much of of the webpage popularity comes from the number of links pointing to it.

Time to practice: 10.Pagerank

You will find guidelines to get you to develop the Pagerank application.

- Take your time.
- Do it step by step.
- Do not hesitate to ask questions.

Table of Contents

Naming conventions

- In C, we tend to use exclusively lowercase letters, using underscores when needed.

```c
int here_is_an_int = 123;
```

- For constants, we tend to use exclusively uppercase letters, using underscores when needed.

```c
const int HERE_IS_A_CONST_INT = 123;
```

Indentation conventions

- Certain programming languages such as Python enforce a specific indentation.
- In C, you are free to indent it the way you want.
- Here are two of the most popular styles, pick whichever you prefer, but **stay consistent**.

```c
if(condition) {
    // Some code
}
```

```c
if(condition)
{
    // Some code
}
```

Table of Contents

Comments

- Single-line comments start with `//`.

```c
// Here is a single-line comment
```

- Multi-line comments are surrounded with `/*` and `*/`.

```c
/* Here is a
   multi-line comment */
```

- Documentation comments are surrounded with `/**` and `*/`[4].

```c
/** Here is a
    documentation comment **/
```

[4] or `**/` if you want to make it symmetrical, though this is pure aesthetic.

Documentation

- Learn Doxygen.
- Simple markup to insert in your comments.
- Run a command and it will automatically generate an entire website, and PDF if you want.
- Can make a good impression in interviews...

```c
/**
 * @brief This function calculates the quotient
 * between the two numbers provided.
 * @param[in] a The dividend.
 * @param[in] b The divisor.
 * @return The quotient of the division.
 * @pre \param b must not be equal to 0.
 **/
float divide(float a, float b);
```

Table of Contents

Summary

- **Congratulations**: you have learned the essentials of the C programming language and compilation.
- Feel free to refer back to this coding refresher at any point in your studies.
- Admittedly, there are also more advanced concepts in C. If you are curious, feel free to explore the web or ask your teachers to learn more.