(The Very) Basics of C Programming

Nuno Miguel Cardanha Paulino (nmcp@fe.up.pt)
Professor Auxiliar Convidado, DEI, FEUP

February 2020

This document is a condensed overview on basics of C programming, and on some concepts about compilation in general. The point of this document is to introduce you to some concepts that might not be familiar if your prior programming experience was not C/C++ or a compiled language.

This document is a work in progress! So check back for future versions.

Contents

1 C Compilation Flow

Unlike interpreted languages (e.g., Python and MATLAB) C/C++ programs are compiled into executable files. Compilation is the process of transforming one or more source code files into a single output file which can be executed on a specific target environment (e.g. Windows, Ubuntu, etc). Depending on the environment, compilation details vary (e.g. different compilers and libraries), but the general process remains the same. The following figure illustrates this:

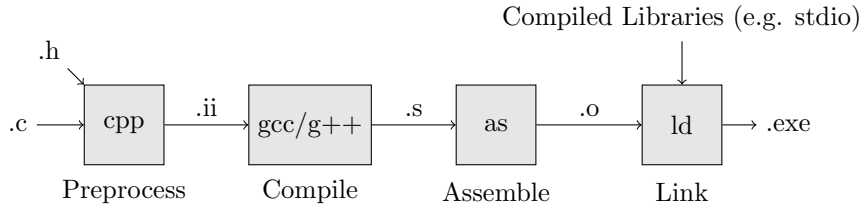

Figure 1: C/C++ Compilation Flow

In C there are two types of source files: *.c* files, which are most of the source code, and *.h* files, which are called headers. Source files and header files are joined together by the preprocessor, *compiled* into several *.s* files, which are then *assembled* into *.o* (object) files, and then *linked* into the final executable.

The *.s* files are are assembly files, which contain assembly instructions. The assembly code is specific to the processor (or processor family) you are coding for, so at this stage, the code is no longer independent of processor details, i.e.,

it is not high-level source code. The assembler transforms these instructions into binary code, which is what the processor reads from memory when executing. The final stage is to *link* all the object files together. Linking is the process of combining all the information, such as variable and function names, and their location in memory. A call to *gcc/g++*, from the command line or from an IDE like Visual Studio, looks similar to this:

```
gcc main.c file1.c file2.c -o myexecutable
```

This single call performs **all** of the steps shown in Figure 1. The compiler receives a list of source files, and the *-o* option which specifies the name of the output executable. Other options, called *flags*, can be passed to the compiler. See Section 10.

2 About Headers

The headers are included by the source files with the directive:

```
#include<headerfile.h>
```

Where *headerfile* is the name of the file. Directives which start with the cardinal symbol (*#*) are processed by the C Pre-processor (*cpp*), as shown in Figure 1. Another example of this are *macros*, explained in Section 3.

An included file is basically copied, by *cpp*, into the file that is including it. This means that you can (and must) define things such as constants, macros, and function prototypes in a **single** file, and include that information into another, or multiple, *.c* files. But why must a source file necessarily include header files? So that *symbols* in other *compilation units* are made visible to that file.

A *symbol* is any variable name, or function name. A *compilation unit* is any file that the compilation flow transforms from a *.c* into a *.o* file. But why are contents of files not "visible" to each other? Don't you pass all the files to the compiler? Yes, but files are compiled one at a time, so each file needs to know the name and type of other symbols, in other files, that it needs to use.

2.1 Basic Header Usage

Header files are (very) typically included at the top of source files. Although its possible to include a *.c* file, that's not good practice (remember, the "only" thing *include* does is copy-paste). If you want functions defined in *file1.c* to be used in *file2.c*, then you should create a *file1.h*, and include that file in *file2.c*. Specifically, if you want a function in *file1.c* to be visible to others, then *file1.h* should contain only its **prototype**. See Section 8 for more.

2.2 Header Guards

Header guards are a technique that make use of the preprocessor to prevent an include file from being included multiple times into the same *.c* file. This happens, for example, when you include *file1.h* and *file2.h* into *file3.c*, and *file1.h*

itself also includes *file2.h*. Header guards prevent symbols from being defined multiple times in the same compilation unit. They are written like this:

```
#ifndef _MYINCFILE_
#define _MYINCFILE_

// All your header content goes here
    // (function prototypes,
    // extern variables, typedefs, etc)

extern int aVariable;
    // I am a global variable declared somewhere else

int myfunction(int a);
    // I am a function prototype
    // I have a head but no body
#endif
```

The *#ifndef*, *#define*, and *#endif* are preprocessor directives. During the first phase shown in Figure 1, the preprocessor evaluates if a macro called *_MYINCFILE_* exists. If not, the content in the *#ifndef* is processed, i.e., its copied by *cpp* into the *.c* file including the header, and the *_MYINCFILE_* macro is defined. If this *if* is evaluated again, the content inside will not be copied by *cpp* again. See Section 3 for more information on macros.

You might find code that uses this single line at the beginning of the file:

```
#pragma once
```

The effect is the same as the header guards, but *#pragma once* is not supported by all compilers.

2.3 Where does the compiler look for header files?

It varies. The compiler will try to find library headers (like *stdio.h* and *math.h*) in directories that are different between operating systems. But since it is already configured to look for libraries in the proper place, you only need to worry about the location of your own headers. If you have all *.c* files and *.h* files in the same directory, the preprocessor will be able to find them, otherwise, you must specify the search directory for headers when calling the compiler:

```
gcc myfile.c -I./inc1/ -I./inc2/
```

The *-I* flag tells the compiler to look for headers in *inc1*, and *inc2*, which are subfolders inside the folder (i.e. working directory) that you have called *gcc* from. See Section 10 for other compilation flags.

---

3 Macros

3.1 Using Macros to Define Constants

As previously mentioned, macros are preprocessor directives. A macro is essentially the definition of a symbolic name for value or expression. For example:

```
#define VALUE 20

int a = VALUE;
int b = a + VALUE:
int c = function1(b, VALUE);
```

When the preprocessor replaces a macro name with its content, this is called **expansion**. Defining constants with macros allows you to keep the code cleaner and to more easily interpret what your expressions are calculating. Also, if you need to change *VALUE* across the entire code (which could be hundreds of files and thousands of lines), you only have to change the macro definition. You can even change macro values when you call the compiler, for instance:

```
gcc main.c –DVALUE=15
```

For this compilation, *VALUE* will be 15. You can use this to generate many executables with different parameters without changing the source code. One useful macro (which you will likely use) is defined in *math.h*, and defines the value of π to many decimal places, with the name *M_PI*.

3.2 Using Macros to Define Expressions

Another use for macros are expressions. As you may have concluded, the preprocessor basically replaces every occurrence of the macro name with what you have defined. This means you can define the macro as anything. For example:

```
#define BIT(a) (1 << a)

int a = BIT(2);
    // the third bit of "a" will be 1,
    // the others will be 0
```

The macro shown above accepts one argument, and before compilation, it is replaced with the expression using that argument. This macro is just a convenient and readable way of setting a single bit of a variable to 1. Macros are usually named with all capital letters, to distinguish them from function calls. Also note that the contents of the macro are enclosed in parenthesis, which is to prevent the following type of error:

```
#define BIT(a) 1 << a
    // I forgot the parenthesis!

int a = BIT(2) * 2;
    // This macro will expand like this:
    // int a = 1 << a * 2;
        // this means "a" will be multiplied BEFORE
        // the shift, due to operator precedence
```

4 Basics of Variables and Data Types

4.1 Declaring Variables

Programming is all about manipulating data. Variables are basically names given to positions in memory where you data is stored. In *C*, variables must have a type (more on types in Section 4.2), and must have unique names. Variables can be declared anywhere in *C*: inside the *main* function, any other function, or outside functions (i.e., global variables):

```
// file starts here
#include<stdio.h>
    // include whatever you need

int a = 4;
    // A global variable

int funcA(int var1) {
    int result = var1 * a;
        // A local variable
    return result;
}

int main() {
    int result = funcA(8);
}
// file ends here
```

Notice on how the *integer* variable in *main* has the same name as the variable declared inside *funcA*. Doesn't this violate the rule of unique names? No, because the variables are declared in different **scopes**. Names must be unique only inside the same scope, and each function (both *main* and *funcA*) define their own scope. However, declaring a variable called *a* (of any type), inside either function would **not** be allowed, since *a* is declared on the **global** scope.

To be precise, it's not the function that defines a scope, it's the curly brackets ({}). You can define scopes by enclosing any block of code inside curly brackets. For example:

```
// file starts here
#include<stdio.h>
    // include whatever you need

int main() {

    // one scope
    {
        int a = 2;
        // (...) more code here...
    }

    // another scope
    {
        int a = 4;
        // (...) more code here...
    }
```

```
}
// file ends here
```

Its also possible to define scopes inside scopes, and inner scopes will have access too all names of their parents, but not vice-versa.

4.2 Data Types of Variables

The (basic) types of variables defined in C include:

- char (8 bit)
- short (16 bit)
- int (at least 16 bit)
- long (at least 32 bit)
- long long (at least 64 bit)
- float (a 32 bit representation of real numbers)
- double (a 64 bit representation of real numbers)

As you have noticed, some types have an undefined number of bits. This is because what the compiler does with these types depends on the specific processor you are compiling the code for. Why is this important? Because the number of bits you use to store a value determines how large (or how small) the represented value can be. For 8 bits, you can represent values between 0–255, or between -127–128, if you are using a signed value. Variables are *signed* by default, if you want an *unsigned* variable, you declare it with that modifier:

```
unsigned char a = 0; // I go from 0 to 255
```

The maximum representable value for an unsigned variable is computed with $2^n - 1$, where n is the number of bits. The number of bits of a datatype is also called its *bitwidth*. To make sure that you have the exact size you want, its typical to include *stdint.h*. This library defines variable types (with *typedef*, see Section 5.3), which ensure a specific bit length, and if the variable is signed or unsigned. These types include:

- uint8_t (unsigned *int* with 8 bit)
- uint16_t (unsigned *int* with 16 bit)
- int32_t (signed *int* with 32 bit)
- etc.

Note that *stdint.h* only defines *integer* types, i.e., no real (*float*/*double*) numbers.

4.3 Implicit and Explicit Type Casting

To understand type casting, and data types in general, you have to understand that variables are only names to positions in memory that contain a certain number of bits. So what happens when you assign a variable to another variable? It depends on their types. For example:

```
char a = 12;
char b = a; // "a" is the same type as "b"
            // the value of "a" is simply
            // copied to the location of "b"
```

This is the most straightforward example. The two other cases are: the assignment of a variable to another with a smaller bitwidth, and vice versa. The behaviour also depends on whether the variables are signed or not:

```
// 32bit Signed to 8bit signed
int a = 1024;
char b = a;
    // An int is 4 bytes, and a char is 1 byte
    // During the assignment, only 1 byte is copied
    // Since the value a is holding exceeds 1 byte,
    // the actual value of "a" is lost

// 8bit Signed to 32bit signed (source is positive)
char c = 56;
int d = c;
    // This is the opposite situation,
    // and "c" is correctly copied to "d"

// 8bit Signed to 32bit signed (source is negative)
char e = -12;
int f = e;
    // "e" is one byte, with the value 0xF4
    // (in binary: 11110100)
    // Since both variables are signed,
    // the compiler performs "sign extension",
    // so instead of copying a single byte to "f":
    // 0x000000F4 (this would be a positive number),
    // the most significant bit of "e" are extended
    // to all bytes of "f": 0xFFFFFFF4

// 8bit Unsigned to 32bit signed
unsigned char g = 24;
int h = g; // This works

// 8bit Signed to 8bit unsigned
unsigned char i = -24;
int j = i;
    // i = 0xE8 (in hexadecimal)
    // The one byte of "i" is copied
    // but sign extension is not performed
    // This means that "j" will hold a value
    // equal to 232 (0x000000E8)
```

All these conversions (type casts) are **implicit**, meaning that the compiler determines what conversion to perform, if it can. If you want, you can call

gcc/g++ with the *-Wconversion* flag, and warnings will be printed for all conversions that may fail.

The other type of cast is the explicit cast:

```
int a = 1024;
unsigned int b = (unsigned int) a;
```

The type specified between parenthesis defines the target type. In this example, it is not required (or very useful) since the compiler knows the source and destination types. But explicit casts are useful for intermediate calculations:

```
char a = 32;
char b = 64;
char c = (char) ( ((int) a * (int) b) >> 8 );
    // 32 * 64 = 1388, which exceeds a single byte!
    // 1388 = 0x056c
    // By shifting 0x056c to the right by 8 bits
    // we want to save the most significant byte,
    // 0x05, into "c"
```

Without casting *a* and *b* to *int*, the intermediate result of their calculation would be a single byte. In other words, the result **overflows**, and the second byte is lost. Since we want to save that byte to *c*, by shifting right by 8 bits (and discarding $0x6c$), the result of *c* would instead be zero (0x00). By telling the compiler to use intermediate, unnamed, *int* variables to store the calculations, we get the behaviour we want.

Explicit casts are also necessary when using dynamic memory allocation, and *void* pointers. See Sections 6.1 and 6.3. Finally, you may ask why we simply don't use the *double* type for every variable, since it its 64 bit wide can can represent real numbers. The reasons are the amount of memory required, and computing efficiency. Typically, calculations performed on *integers* are faster than *float/double*.

4.4 Typed vs. Untyped Languages

Unlike Python, PhP, or MATLAB, the C/C++ language (as well as many other compiled languages) are strongly typed. This means that explicit types are given to variables when they are declared, and that a variable cannot change type. In other words, the following is **not** valid C/C++:

```
int aVariable = 2;
aVariable = "a string";
```

The identifier *aVariable* is used to refer to a specific location in memory where a *integer* datum is stored. The name *aVariable* refers only to this, and cannot be reused for other purposes. Specifically, C is statically typed, meaning that *you* define the type of a variable before compilation, and that it does not change. Although the most recent revisions of C support automatic variables, they are still statically typed. Example of an automatic variable declaration:

```
auto variableX = 4.5;
// The compiler can determine that this is a "float"
```

4.5 Variables in Header Files?

Never. See Section 2.1. In headers you should define macros, function prototypes, and make variable **names** visible by using the *extern* modifier (see Section 9). Two things can happen if you define variables in headers, depending on whether or not your headers have *header guards* (which they should):

- If you don't: If you define a variable in a header file, and then include that header in multiple *.c* files, then the same variable name (i.e., *symbol*) will be defined multiple times. It is easy to generate these errors if you do this, especially if you include several headers, and those headers also include each other.
- If you do: The header guards will prevent the headers from being included multiple times into the same compilation unit (*.c* file), which seems to solve the problem. But during linking, each *.o* will have effectively **different** variables with the **same name**, and linking will fail due to duplicate symbols.

---

5 Arrays and Structs

In the last section we saw the basics of variables. This section covers advanced topics like (static) arrays of variables, *structs*, and how we can attribute custom names (i.e., aliases) to data types.

5.1 (Static) Arrays

Arrays are a way to define a sequence of variables of the same type, under a single name. You may declare arrays of any variable type, by specifying that type, the array name, and its size:

```
int a[20];
    // Twenty ints
```

Each value in the array is commonly called an *element*. The array in the example is a **static array**, meaning that is has a fixed size, which cannot be changed after declaration. Notice how the array is declared, but unlike variable declarations we saw before, its values are not **initialized**. There are multiple ways to initialize array values:

```
int a[5] = {0}; // everything is zero
int b[5] = {1,2,3,4,5}; // each element has the specified value
int c[5] = {[2]=4}; // second element is "4", others are zero
```

To access specific elements of arrays, we use the index operator ([]), like this:

```
int a[5]; // Five elements
a[0] = 2; // which go from index "0"
a[4] = 6; // to index "4"
```

```
int i = 2;
a[i] = 4; // Using integer variables (int, short, char)
          // as indexes, is also allowed
```

There is no maximum size to a static array, in terms of what the language itself allows, but in practice, you will have limited memory. Trying to use very large static arrays is the kind of "error" (or bad coding practice) that will only manifest during compilation (it may fail), or execution (performance will be poor). It is also possible to declare arrays of multiple dimensions, but this document will not cover that. Instead, see Section 6.3 for information of arrays of variable size.

5.2 Structs

The *C* language allows you to define custom data structures. *Classes* are more sophisticated than *structs*, and are only supported in *C++*, which this document does not discuss. The main difference between the two is that *structs* can only contain *members*, and *Classes* can contain *members* and *methods*. A *member* is any variable of any data type, a *method* is a reference to function of any kind. This is an example *C struct*:

```
struct aName {
    int a, b;
    char byte1;
    int data[20];
};
```

This **defines** a *struct* of type *struct aName*, which contains all the variables declared within: three *ints*, one *char*, and 20 more *ints* in an array. Understand that *structs* are basically variables of a custom type. So we can declare a variable of this type, and access its *members* (or *elements*):

```
struct aName s1; // Declare a variable of type "struct aName"
s1.a = 20;       // Members of structs can be accessed with "."
s1.data[2] = 4;
```

We can even do this:

```
struct anotherTypeName {
    struct aName a1;
    int avalue;
};
```

That is, a *struct* definition may contain a declaration of another type of *struct*. However, it **can not** contain a declaration of a *struct* variable of its own type (although it can contain a pointer to a *struct* of its own type, see Section 6.1). Likewise, we **can not** do this:

```
struct aName a1; // Declare one struct
struct aName a2; // another one;

a1.a = 2;        // Put some values in struct "a1"
a2 = a1;         // This WON'T work
```

The compiler does not know how to **copy** structures, which is what the assignment operation (=) is. In order to copy structures, we must manually copy each *member* of the source structure to the destination structure:

```
struct aName a1; // Declare one struct
a1.a = 2;
a1.b = 4;
a1.byte1 = 1;
for(int i = 0; i < 20; i++)
    a1.data[i] = i;

struct aName a2; // another one;
a2.a = a1.a;
a2.b = a1.b;
a2.byte1 = a1.byte1;
for(int i = 0; i < 20; i++)
    a2.data[i] = a1.data[i];
```

Code becomes much better structured if you implement functionalities like this in functions, see Section 8. Finally, you can also declare arrays of *structs*:

```
struct aName a[20]; // Twenty structs
a[0].a = 2;
```

5.3 Defining Types

The *typedef* keyword allows you to give new names to data types. This is how the types explained in section 4.2, e.g., *uint8_t*, are defined. Usually, *typedef* is used to give *structs* shorter names. For example:

```
// Define and typedef the struct
typedef struct aName {
    int a, b;
    char byte1;
    int data[20];
}; aName_t;

// Declare the struct
aName_t a1;
```

---

6 Where do Variables go? What is Memory?

When you declare variables, their data is placed in memory. From the point of view of the *C* programmer (i.e., you), what happens exactly at the level of the RAM and operating system doesn't matter. The memory in a *C* program (and virtually all other languages) can be modeled as table like the example in Table 1.

Addresses	Content (32 bit)
0x00	
0x04 (a)	2
0x08 (b)	4
(...)	
0x20 (c[0])	1
0x24 (c[1])	2
0x28 (c[2])	3
0x2c (c[3])	5
0x30 (c[4])	8
(...)	
0xff	

Table 1: A simplified look at the memory

On the left-hand side are the memory addresses, which we use to lookup specific contents in memory, on the right-hand size. The number of rows of the table is the memory size, and the range of addresses you can use to look up all the rows is called the **memory space**. Notice how the addresses increment by 0x04 from row to row. This is because this table represents the data in memory as pieces of 32 bit each, which corresponds to 4 bytes. It is useful to think this way, since most data types you will use in *C* are 4 bytes wide.

It is the compiler than chooses where to place data in memory when it is declared, so the memory positions shown are just examples, but this code would place those elements into memory:

```
int a = 2;
int b = 4;
int c[5] = {1,2,3,5,8};
```

Notice that elements of arrays are placed into memory **sequentially**, and its because of that we can **iterate** over arrays using loops (see Section 7). Think of declaring a variable as basically giving a specific memory address a name. We can then use that name to read/write to that position. You can figure out the address of a variable with the **dereferencing** operator &:

```
int x = 2;  // Assume "x" is in memory position 0x08;
int y = (int) &x; // "y" is now equal to 0x08;
```

6.1 What are Pointers? Referencing and Dereferencing

Pointers are extremely important in *C*. Most of the difficulties you will have with *C* programming will most likely come from not understanding pointers and memory. Pointers are closely connected to the concepts of memory addresses, **referencing**, and **dereferencing**. The previous section closed with a dereferencing example where the address of a variable is placed into another variable. You'll notice how we had to cast the result of $\&x$ to *int*. This is

because the dereferencing operator returns **addresses**, and addresses can only be held by special data types, called **pointers**:

```
int x = 2;  // Assume "x" is in memory position 0x08;
int *y = &x; // "y" is now equal to 0x08;
```

By prefixing the variable name with an asterisk, that variable becomes a pointer. It should only be used to hold addresses of variables which have the same type. In terms of memory, pointers are nothing more than variables whose **value is a memory address**. That address can be, but not necessarily, the address of another variable. The contents of the memory for the example above would look something like what is shown in Table 2.

Addresses	Content (32 bit)
0x00	
(...)	
0x20 (a)	4
0x24 (b)	0x20
(...)	
0xff	

Table 2: A pointer variable holding the address of another variable

In Table 1, we saw an array declared in memory. When you declare a static array, the name of the array is actually a pointer type variable. You may have noticed that to access the elements of the array we used the [] operator. This is a **referencing** operation. Referencing is when you take a pointer, and use its value (an address), to read/write the value at that memory address. For example:

```
int a[5];
int *b = a; // This is valid

// doing this
int x = a[0];

// or this, is the same
int y = b[0];
```

Notice how the & was not applied to *a* when initializing *b*. This is because *a* is of type *int **, i.e., *int* pointer. These examples have used the square bracket to perform referencing, but the actual operator is the asterisk symbol, *:

```
int a = 5;  // Assume "a" is at address 0x20
int *b = &a; // Save the address

// assign the value at that address to another variable
int c = *b;
    // * means "point to this address"
```

Note how the asterisk means different things when declaring the pointer, and when initializing *c*. In the first case, it indicates that the variable is a pointer,

in the second case, the asterisk is an operator. In other words, this means that when you access elements of an array, this is happening:

```
int a[5];

// this
int b = a[1];

// is the same as this
int c = *(a + 1);
    // we take the base address,
    // add one position, and reference
```

When we take the name of the array, which is in fact a pointer to the start of the array, and add or subtract values to it, we are performing what is called **pointer arithmetic**.

6.2 Pointer Arithmetic

Check back later!

6.3 Dynamic Memory Allocation

Check back later!

---

7 Loops

Check back later!

---

8 Functions

Functions in *C* are blocks of reusable code, useful for containing functionalities that you want to use repeatedly, and to help you create more readable and organized code, since it helps you identify the purpose of a specific block of code. This is a (very simple) function definition:

```
// A function is declared, and defined
int aFunctionForAddition(int a, int b) {
    return a + b;
}

// The function is called
int result = aFunctionForAddition(20, 10);
// result will be "30"
```

A function is defined by: the data type of the variable it **returns**, in this case an *int*, by its name (which in *C* must be unique), and by a list of **arguments** separated by commas. There is no limit to the number of arguments, and they may all be of different data types.

The above example **declares** and **defines** the function at the same type, because it states the prototype of the function (i.e., the head), and then defines the code inside the function (i.e., the body). Like we saw in Section 2.2, its possible (and correct) to declare only the function prototype in header files, but you can also do it in *.c* files. Why is this useful? Because the compiler reads the file line by line. If you try to call a function in line 10 of your code, which is only declared and defined in line 20, that will produce and error. This could be resolved by "swaping" the position of the functions, but this solution has two problems: 1) for large code, you constantly have to keep track of where functions are called, and 2) it does not solve the problem of two functions calling each other. Instead, we can use **forward declaration**:

```
// A function is declared
int funcA(int a, int b);
    // Now the compiler know's I exist,
    // and what I look like

// Another function uses funcA
int funcB(int a) {
    return funcA(a, a);
}

// funcA is defined
int funcA(int a, int b) {
    return a + b;
}
```

Combine this with what you know about headers (Section 2.1), and you learn that when you have a header *file1.h* where you have declared all the prototypes for all the functions in *file1.c*, and then include *file1.h* in *file1.c* (at the top), what you have done is forward declare all the functions.

More elaborate things can be done with functions, especially if you resort to pointers, see Section 8.2.

8.1 Recursive Function Calls

Functions may call themselves, which is another way to implement loops. A typical recursion example in *C* is sum all numbers up to a specified number *"n"*:

```
// Forward declare!
int calcSum(int n);

// Recursive function
int calcSum(int n) {
    if (n == 0)
        return n;
    else
        return n + calcSum(n - 1);
}
```

In this example, the function calls itself with a modified input argument, and stops when that argument is 0. The result will be a sum which equals too

$n + (n-1) + (n-2) + (...) + 0$. There must always be a condition to terminate the recursion, otherwise the function will call itself "infinitely", and eventually the program will crash, due to **stack overflow**.

8.2 Passing Arguments by Value, Reference, or Address

Check back later!

---

9 Some Modifiers

Check back later!

---

10 Compilation Flags

Check back later!

---

11 Common Compilation Errors

Check back later!

---

12 Indentation

Check back later!