C: A Beginner's Guide

Joshua Weinstein

2018

Contents

0.1 Preface

This book is intended to serve as a tool to learn how to program in the C programming language. The book is targeted at those who have little to no experience in programming or computer science as a whole. Even if someone hasn't learned any prior programming languages, this book can serve as a first learning experience. This book discusses in depth the low level components of the C language, like memory, addresses, the stack, the heap, and more. The book introduces programming and C from the ground up, simplifying concepts often claimed as difficult or complex.

This books also introduces computer science concepts, like functions, types, data structures and algorithms, in the context of the C language. The purpose of this is to teach the fundamentals of programming from a simple, backbone starting point, and work up to more "magic" levels of abstraction, such as those seen in the Python language. This approach exposes students to a mechanical and complex understanding of programming. Such insight is exceptionally useful when optimizing and writing high performance software.

The book pays careful attention to the features of C which are often described as confusing, such as the *void* pointer, the functions *malloc* and *free*, and much more.

Chapter 1

Introduction

C is a low-level programming language designed for high portability and fast performance. C first originated in the late 1970's when Brian Kernighan and Dennis Ritchie wrote the book, *The C Programming Language*. C has tremendously influenced the world of programming, leading to the development of C++, an object oriented version of C, and the syntax of many other languages, like Java. C is still widely used today, in many applications such as databases, kernels, drivers, and much more. One of the most popular programming languages, Python, runs on an interpreter written in C.

This chapter will introduce the very fundamental ideas behind programming, and work as a primer for learning actual C concepts. Ideas such as addressing, memory, the stack, the heap, variables, printing, files will be discussed with friendly examples. If you have no or little experience in programming, it is highly recommended you start with this chapter. Otherwise, skip ahead to the next chapter.

1.1 Programs and Computing

In the most basic sense, a computer is a machine which can complete tasks by running instructions commonly known as "code". A computer cannot read or understand human languages, like English. A computer only understands instructions in a format known as *binary* code. Binary code is composed entirely of one's and zeros. 00000001 is an example of binary, which actually translates to the number, 1.

A programming language is a human readable language which is *compiled* into binary code. Such languages serve as a intermediate way of communication between a computer's mechanics and a human mind's mechanics. These languages allow humans to instruct computers to perform almost any task imaginable. A program is a file or series of files of code written in a particular programming language. Web browsers, mobile phone applications, websites, are all

programs. All of them are in some way compiled into binary code, which is then executed by a computer as instructions. Programs that need high performance are often written in C or C++, a tightly related language to C.

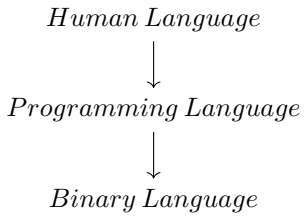


Figure 1.1: The layers of communication between humans and computers

The format in which humans convey information through language is very different than how computers convey information. Programming languages are not at all similar to learning a foreign language. The *grammar* of programming languages is far more restricted than that of a spoken language. When two people speak to one another, they are conveying information that is often *declarative* in nature, where information is stated in some format as true. "I cook food," is a declarative sentence, it states data or information, it does not detail commands or instructions to follow.

Imperative programming involves the communication and passing of information as commands and instructions. C is an imperative programming language, so is Java, and C++. Such languages focus on grouping commands into executable sets of instructions to compute larger more complex values. In many imperative languages, groupings of commands are called *functions*. In the most basic sense, functions apply commands on some specified set of values.

A function is a special set of commands written in code, with some specified input values and output values. Input values to functions are usually called parameters. Functions are used in programming languages to tell the computer to perform specific instructions onto some set of parameters. They generally take the form of:

$$function: name(input) \rightarrow output$$

The input can be on or more values, so can the output. Functions will be explored more in later chapters.

Definitions

1. **Declarative programming**: A style of programming where data is stated and declared.

2. **Imperative programming**: A style of programming that uses commands and instructions that compute data.

3. **Functions**: A group of commands executed on some specified set of values.

1.2 Memory

When a person performs day to day tasks, like driving to work, making meals, or writing letters, they use their memory of past events. Even though the brain is not a computer, it still records data from events we experience which we think about or remember at some later time. When a computer executes binary code or compiled code from a programming language, it needs to use memory. If humans had no way of remembering or recalling anything in the past, they could never solve any problem, mathematical or otherwise. Yet, a computer's memory is not truly chronological. Consider the following example:

Tom needs to buy groceries to make a soup. He knows he needs to buy tomatoes and corn. He makes a list, specifying one tomato and two cobs of corn. He goes tot he grocery store and checks his list, to buy the items he wanted.

In this example, *Tom* needs to buy some amount of vegetables, so he uses a list to check which items he should buy. If a computer were to purchase or order groceries, it would also need a list of what items to purchase. In the previous section, we discussed functions, an arrangement of instructions that are executed on some input values. Lets look at an abstract example.

$$function: BuyGroceries(list) \rightarrow PurchasedItems$$

Here, a function is taking in an input of a list, and outputting the items the list stated to be purchased. The output is related to what the list holds in terms of data. More specifically:

$$\begin{array}{ccc} List & \longrightarrow & PurchasedItems \\ | & & | \\ input & & output \end{array}$$

To keep track of the list, when a computer is executing commands, it needs to store that list. Secondly, it needs a place to output those purchased items to. To store such information, for a simple function as shown or many other tasks, a computer uses memory.

1.2.1 What is memory?

Memory is a physical component inside a computer that stores temporary data and information necessary for a computer to complete tasks. When a

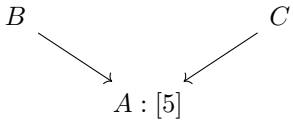
program receives an input, such as a mouse click on a button, or message of text, it needs to have a place to store that input, and any other values it needs to compute the desired output. To start with, we can think of memory as a finite number of boxes, each which can store a value. Such boxes allow us to reference that value in order to change it or create new values. Here is an example:

Example:

$$A : [0]\ B : [0]\ C : [0]$$

Let A, B, and C be boxes of memory. Initially, they all hold the value 0. A memory box must always hold some value. Next, let us inset or load numbers into boxes B and C.

Memory box B and C can be added together, with the result of the mathematical operation being stored in box A.



The final state of each memory box is :

$$A : [5]\ B : [2]\ C : [3]$$

In this example, at every step, the memory of what value A, B, or C holds is persisted. Those values are stored and changed as the commands continue to be executed. This is an example of how memory in registers also works, a topic that will be covered later on.

Types of memory

In a computer, there are a few different types of memory. They are used for different tasks, and different roles. Some are also slower than others.

1. **Registers**: This is the fastest, most highly accessible form of memory, it's commonly use in executing instructions.

2. **RAM**: Also known as simply memory or local memory, this is memory used to store temporary values during the time a program is running. In most programming languages assignment statements, such as $a = 3$ are using local memory.

3. **Disk**: This is where a computer stores files, the least accessible, yet longest lasting form of memory.

Memory is measured in units. The smallest, accessible unit of memory is called a byte. Computers and files often measure data in *kilobytes* or *megabytes*. These mean a thousand bytes and a million bytes respectively. The number system behind memory and data will be discussed in the next section.

1.3 Numbers

A computer and it's programs use numbers, just as humans encounter and use numbers in their daily lives. Phone numbers, street addresses, freeway exits, numbers exist all around us. Numbers are the core component of any data inside a computer. Computers can only understand sequences of numbers, they don't see text or images like living beings can. In a previous section, we discussed *binary* language in reference to communicating with a computer. Binary language is just a composed sequence of ones and zeros. This section will provide insight into the system and types of numbers used in programming, such as binary numbers.

1.3.1 Binary Numbers

Numbers are quantifiable, distinct values that exist in different sizes, composed of digits. The possible digits a number can have are determined by it's radix. The size of a number is determined by the *domain* of values it is assigned to represent. To better understand this, let's investigate the system of binary numbers.

Example: Binary Numbers

A binary number, B_r^l has a radix, r and a length, l. The radix for all binary numbers is 2, while the length is variable depending on the value represented. The Binary number for 2 is 10*. The beginning binary numbers are:*

$$0 \equiv 0$$
$$1 \equiv 1$$
$$2 \equiv 10$$
$$3 \equiv 11$$
$$4 \equiv 100$$
$$5 \equiv 101$$
$$6 \equiv 101$$

For every position in a binary number, there are only two possible values, 1 or 0. As the integer represented in the binary number becomes larger, so does the length of the binary number.

Every 1 or 0 in a binary number is called a *bit*. You may be familiar with the term 64-bit or 32-bit operating systems. This size refers to the size of the binary number that holds instructions for computers with such systems to execute. Binary numbers are usually divided by increasing powers of two, such as 8-bit, 16-bit, 32-bit, and 64-bit numbers. Each class of a binary number, can only hold a maximum number of values. Also called permutations, the number of permutations for a binary number is given by the formula:

$$B = 2^n$$

where n a finite length, and 2 is the radix of binary numbers. In the case of 8-bit numbers,

$$B_8 = 2^8 \equiv 256$$

8-bit numbers, also called *bytes*, can hold the number values between 0 and 255. From the 8-bit number 00000000 to 11111111, there exists 256 possible permutations. We will learn in later chapters that bytes the smallest addressable unit of memory. For now, we can think of a byte as the smallest number to represent a box of memory.

1.3.2 Integers

In contrast to binary numbers, integers are numbers you are likely largely familiar with from everyday life. 100, 500, 6,000,000 are all integers. An integer is any positive or negative whole number. Numbers with decimal portions are called *float* numbers and will be discussed later on.

Integers can be represented in many sizes, but in most programming languages, they are represented by 32-bit or 64-bit numbers. For a 32-bit number, the maximum value is:

$$N_{32} = 2^{32} \equiv 2,147,483,647$$

for a 64-bit number, the maximum value is

$$N_{64} = 2^{64} \equiv 9,223,372,036,854,775,807$$

Note: You may have noticed that 2 is also used above as a radix for integers. As mentioned previously, a computer can only interpret and understand binary language. Integers in the exact form humans read them are not binary. Thus, each integer in a programming language is mapped to a binary number.

Integers can also be *signed* or *unsigned*. Being signed or unsigned allows us to split the number of available values a sized number can represent. The most common use case is to represent both positive and negative numbers.

Definitions

1. **Binary Number**: A number of some length composed entirely of 1's and 0's.
2. **Integer**: A whole, positive or negative real number, such as 55.
3. **Radix**: The number of possible digits a number can be composed from.
4. **Permutation**: An ordering of elements, such as digits, where the order and presence matter, e.g 101 and 110.

1.4 Compiling a Program

The C programming language is a compiled programming language, meaning the textual representation of the language as a whole is converted into binary or *machine* language at once. The result of that conversion is called an executable. Executables are programs that can be run by a computer. In order to transform a C program into an executable a compiler needs to be used. A compiler is a program that takes an input of one or more C files, and returns or results in an output of an executable.

Nearly all computers come with a default C compiler. The most common and widely used is called *gcc*, a compiler that is available for windows, macintosh and linux operating systems. Alternatively, many websites offer online compilers that you can write and compile code on web page. That will only display a textual output of the program, but that's more than sufficient for the course of this book.

To compile a C program locally on a computer with a *UNIX* operating system, such as macintosh or linux, do the following. We will compile and run the following program as an example

```
#include <stdio.h>

int main(void)
{
      puts("Hello World!");
      return 0;
}
```

Steps:

1. Open your *Terminal* program.
2. Type the command emphvi program.c. A new view will open in your terminal.
3. Press I to enter insert mode, and paste the above program into the terminal.
4. Press the *esc* key, then type *:wq*, to write and quit the text program.
5. Type the following command *gcc program.c -o program*

6. Then finally, run the program by typing *./program.c*

You should get the following response:

```
Hello World!
```

This program is a basic, fundamental piece of code that tells the computer to *put* the text "Hello World!" into your terminal. Next, the specific portions and statements in this program will be discussed.

1.4.1 Structure of a Program

A C program before it's compiled into an executable, is a collection of one or more *.c* files. Amongst these files, is a function called *main*. The main function of a C program contains the entry point, the starting point for the instructions and commands the program will perform. You can think of a program as an entrance for the program to accept input.

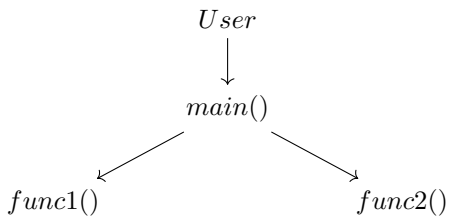


Figure 1.2: A diagram of a program with a main function.

Within the main function, you see the parameters specified as `void`, meaning the function accepts no parameters. In later chapters, you will learn `void` is a *type* that indicates an absence of a form or structure.

At the top of the program there is a statement that begins with `#include`. This statement pastes in the textual content of another file into the current one. The include statement allows multiple files to be used in a single C program. The pointed brackets, $< \cdots >$, hold the named of a file to be included. In this particular situation, the program includes `stdio.h`, a file that comes packed with a C compiler, which contains functionality for a program to print output, receive input, or write to files. This header file along with other standard files will be discussed later.

int to the left of the *main* function indicates the return type of the function. In C, all functions specify the type of output they return. In general, *return* is a keyword used in many programming languages to signify the output of a function. It does not mean that what is returned had to be originally passed into the function.

The `puts` function is a function from the standard file `stdio.h`, which takes a string as an input and outputs that string to the terminal. A "string" in C, or any programming language is a type of data that holds readable text. It is composed of 8-bit integers called *characters*, the represent readable symbols such as h or e. The mechanics of strings and characters will be discussed in another chapter.

Definitions

1. **Header File**: A file which allows multiple C files to be compiled into one executable.
2. **stdio.h**: A file packaged with a C compiler that provides functions to receive input and output, or write and read to files.
3. **main function**: The function that acts as an entry point to a C program.
4. **include**: A pre-processor command that pastes the text of another C file into the current C file.
5. **return type**: A signifier in C function syntax that specifies the size and kind of value being outputted from a function.

Chapter 2

Types

C is a language that uses types on values in a program. Types exist as a label across some raw from of data. In the previous chapter, we learned that integers in C or other languages exist in sizes, such as 8-bit integers. Types allow the compiler for a C program to recognize the size of the data contained in a value. Knowing the size is extremely useful because it allows a compiler to know the correct memory that needs to be used while executing commands on that value

This chapter will introduce the primitive types of C, along with modifiers that can act on types. This chapter will also discuss the concept of type safety, and when type safety can be disobeyed.

2.1 Declaration and Initialization

In C, all values have a type, signifying the format of data a particular value holds. Values must have their types *declared*. See the following code:

```
1 int a;
2 char b, c;
```

Here, a is declared as an `int` and b with c are declared both as a `char`. This implies any further accessing of these variables will be done according to their declared type. Attempting to re-declare an already declared value, for example, results in an error.

```
int a;
char a, c;


 error: redefinition of 'a' with a different type: 'int' vs '
     char'
        int a;
            ^
 note: previous definition is here
        char a, b;
             ^
```

Note: *Many C compilers, including gcc, give detailed error messages which indicate where the error in the code is. However, the above error is a compile time error, meaning the error is detected by reading the textual input of the C code. Errors that occur during the execution of the program, called runtime errors, are much harder to detect and rarely have error messages.*

Declaring a type does not assign it any value. Values such as integers shown above are given a default value of zero. Both declaring and initializing in the same statement is permitted. You can also reassign a declared variable, like *a*, to another value of the same type.

```
int a = 6;
a = 8;
```

Additionally, attempting to initializing an undeclared variable results in an error, as the C compiler needs to be aware of what kind of data is being held by that value.

2.2 Integers

Integers in the C language play an important role in storing whole, numerical values. Integer types are named according to the size of the data they can hold. `int` is the most widely used and basic integer type in C. Although this can differ depending on the compiler and type of computer a C program is compiled on, the `int` type is 32-bits long.

You can assign any integer to a variable name, like this

```
int x = 1;
```

The above code means anytime that `x` is accessed in the program, it will hold the value 1. As mentioned, the int type is a 32-bit integer. Yet, 32 bits are not needed to represent the number one. In fact, only 1 bit is needed to represent one and subsequently the number zero. However, a single bit is not an *addressable* size of memory, nor is it an efficient chunk of memory to deal with. Therefore, the smallest units of memory used in C are bytes, or 8-bit integers. Let's take a closer look in what is happening in the above statement:

$$x \longrightarrow [00000001, 00000000, 00000000, 00000000]$$

or if we simply want to view the bytes as decimal numbers,

$$x \longrightarrow [1, 0, 0, 0]$$

A memory chunk of 32 bits, or four *bytes* is being assigned to x. Since 1 is a number that can fit into a single byte, only the first byte of `x` is occupied. Despite the number being small, the other bytes of memory are grouped with this value to permit the number to grow and take up more space. Next, lets try and add a larger number to x.

```
int x = 1;
x += 300;
```

In C, the `+=` symbol is an operator that allows you to reassign variables and add to them at the same time. It translates to $x = x + 300$. However, x now holds a value greater than the maximum 8-bit integer of 255. The other three bytes associated with that integer now start holding data, as shown below:

$$x \longrightarrow [45, 1, 0, 0] \equiv 301$$

In this example, the 2nd byte of `x`, 1, essentially translates to, *this value is 255 plus 45*. The 2nd byte acts as a counter for the number of times 255 is counted in the larger, 32-bit integer. The largest possible 32-bit integer value would be read as $[255, 255, 255, 255]$. If you attempt to store a numeric value larger than what an integer type can hold, it will cause an *overflow* error. These types of errors are usually handled by the C compiler changing the type of the value in order to hold the large number, or stating the number is too large overall:

```
warning: implicit conversion from 'int' to 'char' changes value
    from 500 to -12 [-Wconstant-conversion]
        char ch = 500;
             ~~   ^~~
error: integer literal is too large to be represented in any
    integer type
        int i = 50000000000000000000;
                ^
```

A `char` is a *signed* 8-bit integer, meaning it is a number that can hold the values from -127 to 127. The number zero is counted as a possible value, which is why the positive limit is 127, not 128.

2.2.1 Integer Types

C has several different integer types, which differ by their size and usages. Here is a list of all the following integer types:

- ***char***: A 8-bit integer that holds values from -128 to 127. It also represents textual characters.
- ***unsigned char***: An unsigned version of *char*, holds values from 0 to 255.
- ***short***: A 16-bit signed integer.
- ***unsigned short***: A 16-bit unsigned integer.
- ***int***: A 32-bit signed integer.
- ***unsigned int***: A 32-bit unsigned integer.
- ***long***: A 64-bit signed integer.

- ***unsigned long***: A 64-bit unsigned integer.
- ***size_t***: A special type that always hold the largest size of integer for the given platform.

For any integer type, if you don't need negative values, or encountering a negative value would be considered an error, it's best to use `unsigned` types. Using unsigned types also allows the integers used to hold a higher maximum value.

The sizeof operator

In the C programming language, all types have a *known* size. When C code is compiled into an executable program, the compiler reads and checks for the size of a type it encounters, whether that type is builtin or defined by the code file. It's important to note the `sizeof` operator always evaluates to a number of bytes, not a number of items or other size unit. Although we have not covered this yet, if the operator is used on an array, it evaluates to the size of the entire array:

```
int a = 3;
int b[3];
printf("The size of a is \%ld\n", sizeof(a));
printf("The size of b is \%ld\n", sizeof(b));
```

Which results in

```
The size of a is 4
The size of b is 12
```