# GNOME University
# Introduction to C
# Chapter 1

Christian Hergert
christian@hergert.me

October 2012

Welcome to GNOME University! This chapter will get you started down the path of programming in C by introducing the process of writing source code in C, compiling, and executing the resulting program. If you don't know what any of that means, that's okay, we will learn!

## 1 Source Code

To write programs using the C programming language, we begin with an empty text file. In that file, we write a sequence of characters and symbols to construct the program as we like. After which, we use a tool called a compiler to convert that source code into machine code. Machine code is a representation of your program in a format that your computers CPU can execute. As you might imagine, different types of CPU's have different representations of your program. Compilers allow us to convert a single piece of code into machine code for multiple types of CPU's. Sometimes developers will refer to a "toolchain", which is a name for the collection of tools you need to build software for your machine.

# 2 Hello, World

Let's start by starting up our text editor of choice. If you don't yet have a favorite text editor, try `gedit`. It is available as part of your GNOME desktop installation.

Copy the following listing into your text editor. Be careful and methodical as you go. When copying, notice the characters and keywords used. You will become familiar with them. When I learn a new language, I like to guess and reason about what the language does before I actually learn it.

**Do not copy the line numbers** to the left of the margin. They are provided for readability within the chapter text.

Listing 1: hello.c

```c
#include <stdio.h>

int
main (int   argc,
      char *argv[])
{
    printf ("Hello, World!\n");
    return 0;
}
```

Save the source code as a new file named `hello.c`. The next step in our process is to translate the program into code that the computer knows how to execute. This is called compiling. We will use the program `gcc` to perform this task.

Open a terminal and navigate to the directory where you saved `hello.c`. Compile the source code into a program using the following command. The options `-Wall -Werror` tells the compiler to be very strict about what it compiles. This helps catch bugs early. The option `-o hello` tells the compiler to place the compiled program in a file named `hello`.

```
gcc -Wall -Werror -o hello hello.c
```

If everything worked, the `gcc` command will exit silently and a new executable file named `hello` will have been placed in your current directory. If instead there was an error, carefully check that each character in `hello.c` matches the C code above and try again.

Once you have successfully compiled `hello.c`, we can execute the program from the terminal by prefixing `./` to the file-name to tell the terminal to execute a file within the current directory.

```
./hello
Hello, World!
```

There you have it, your very first C program!

## 3 Anatomy

Lets now analyze the anatomy of this simple C program. The very first line, `#include <stdio.h>` tells the compiler that we would like to use the ***stdio*** library. A **library** is a set of useful, reusable routines. In this case, the ***stdio*** library contains routines to interface with the standard input and output of your terminal[1]. This allows your program to output text to the terminal and input text from it.

The line following `#include` is simply an empty line. Empty lines do not affect the program. In some languages, this does matter. However, C is not one of those languages. You may use empty lines liberally to make your code more readable.

Next, we have `main (int argc, char *argv[]))`. This tells the compiler that we have a function named `main`. The `main` function is the beginning of the program. Every program has one. Every program gets two arguments, `argc` and `argv`. Arguments are declared inside of the pair of parenthesis `()`. We will go into what these parameters mean in a later chapter. Do not worry if this seems mysterious, we will cover it in detail in a later chapter.

After the line defining our function `main`, we have a `{`. This denotes the beginning of the body of the function. There is a corresponding `}` at the end of the function. Inside of these curly braces is the crux of our program.

`printf ("Hello, World!\n")` is a function call to the `printf` function (part

[1]Run `man stdio` in your terminal for more information on the stdio library.

of the ***stdio*** library). You can tell it is a function call because the function name is followed by a parenthesized argument list. In this case, our function call has a single argument, `"Hello, World!\n"`. Information can be shared between functions in a couple ways. This way, through the argument list, is the most common.

Our `"Hello, World!\n"` argument is what we call a ”string”. A string is a sequence of characters contained within two quotation marks (`"`). At the end of this string you see `\n`. This means that the string should contain a line break at the end of it.

We denote the end of a `statement` in C with a semicolon (`;`). What exactly a `statement` is will be vague for now, but it will become clear as we continue.

## 4 Variables

Writing a Hello World program is always fun in a new language, but it wont exactly win us any awards. Lets try to print out some interesting numbers using our program as a base.

Lets create a new text file with the following and save it as `answer.c`.

Listing 2: answer.c

```c
#include <stdio.h>

int
main (int   argc,
      char *argv[])
{
    int a = 42;
    printf ("Answer is %d, what's the question?\n", a);
    return 0;
}
```

Once again, let's use `gcc` to compile the program. You will get very used to this in no time!

```
gcc -Wall -Werror -o answer answer.c
```

And again, let's run it.

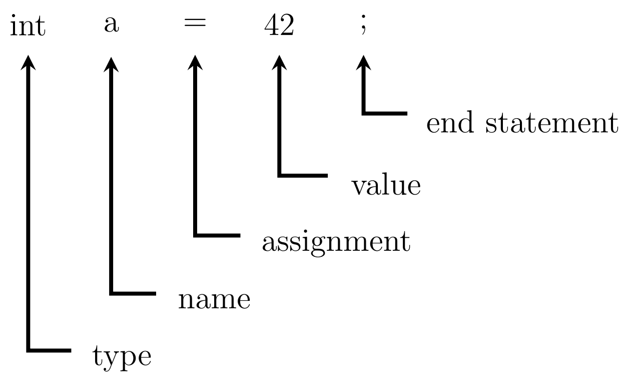

Figure 1: Variable assignment

```
./answer
Answer is 42, what's the question?
```

Since we have covered the anatomy of the `main` function earlier in this chapter, let's skip to the three lines inside the body of the function. Remember that these are the lines inside of the curly braces `{` and `}`.

In Figure 1 we have the dissection of the first line. This line declares a new variable named `a` of the type `int`. Variables are always defined in the format `type name` optionally followed by `= value` and then `;`.

`int` is short for `integer`. You might remember that an integer is an "whole" number, such as 1, 2, or -20. There are no decimal points in an integer. So if you tried to store the number `12.5` in an integer, it would simply be 12. The second half of this statement, `= 42` initialized the variable `a` to the value `42`.

The second line should look familiar. It again is calling the `printf` function. However, this time there are two arguments. We will cover this in more detail in a later chapter, but there are a few important things to take from this example. First, notice the comma `,` used to separate the two arguments provided to `printf`. In C, arguments are separated by commas. Also, note the `%d` inside of the first parameter. This is a magic key that `printf` will replace with the value of the second parameter.

## 5 Types

A type specifies what a variable can store. As you read earlier, `int` can store whole numbers like -1, 0, or 1. There are other types that the C language knows about too. For example, the type `double` knows how to store decimal numbers. Look at the example below assigning the decimal number `123.45` to the variable named `d`. Both `int` and `double` are examples of a `type`. We will learn about other types as we progress through the chapters.

```
double d = 123.45;
```