C Programming/Intro exercise

< C Programming

The "Hello, World!" Program

Tradition dictates that we begin with a program that displays a "Hello, World!" greeting to the screen, followed by a new line, and then exits. Below is the C source code that does just that. Type this code into your preferred text editor/IDE and save it to a file named **hello.c**.

```c
#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

Source code analysis

Although this is a very simple program, a lot of hidden meaning is packed into the many symbols you see in the code. Though your compiler understands it, you can only guess at what the code, sprinkled with some familiar English words, might do. One of your first jobs as a new programmer will be to learn the many "words" and symbols of the C programming language, the language your compiler understands. Once you learn the meaning underlying the code, you will be able to "talk to" the compiler and give it your own orders and build any kind of program you are inventive and resourceful enough to create.

But note that knowing the meanings of arcane symbols is not all there is to programming. You can't master another language by reading a translation dictionary. To become fluent in another language, you have to practice conversing in that language. Learning a programming language is no different. You have to practice "talking" to the compiler with the source code you write. So be sure to type in the code example above and feel free to experiment and alter it with your curiosity as your guide.

OK, so let's dive in and look at the first line in our program:

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

Before understanding what this line does, you have to know that your machine already comes pre-installed with some C software code. The code is there to save you from the drudgery of writing code that performs basic, common tasks. This reusable code is referred to as a **library.** And so our first line in our example program signals to the compiler that we'd like to "check out" some code from the library and make use of it in our program. Here, we are borrowing code that will help us print text to the screen.

The way we tell the C compiler to include library code into our own code is by using what's called a preprocessor directive. One of the very first tasks your compiler will perform is to search through your

source code for preprocessor directives which modify your source code in some way. In our case, the `#include` preprocessor directive tells the compiler to copy source code from a library and insert it directly into the code where the preprocessor directive is found. Since our directive is at the very top of the file, the library code will be inserted at the top of the source file. (Note that this all happens in the computer's memory, so the original file on your disk never actually gets altered.)

But which library code should the compiler insert? The next bit in the line, the `<stdio.h>`, tells the compiler to copy and paste the C code from the **stdio.h** file into your code. The angle brackets surrounding the file name tell the compiler to look for the file in the standard library as opposed, to say, your own personal library of reusable code. Note that files with the **.h** extension are called **header files**. The stdio.h header file contains many **functions** related to input and output that are defined according to the C standard. Though this header file gives us access to many different functions, the only library function we are interested in from stdio.h is the `printf` function.

OK, but what, exactly, is a function? Let's take a look at the next line in our code so we can begin to get an idea:

```
3  int main(void)
```

Here we create a **function** named `main` that is the starting point for all C programs. All C programs require a function called "main" or they will not compile. Our function name is surrounded by two mysterious symbols, **int** and **(void)**. The "int" bit tells the compiler what kind of value our function will return while the "(void)" bit tells our compiler what kind of values we will "pass" into our function. We'll skip over what exactly this means for now as these values will be covered in more detail later in the book. The most important thing to understand right now is that together, these symbols **declare** our function to the compiler and tell it that it exists.

So what is a function? In computer science, the term “function” is used a bit more loosely than in mathematics, since functions often express imperative ideas (as in the case of C) - that is, *how-to* process, instead of declarations. For now, suffice it to say, functions define a set of computer statements that work together to carry out a specific task. In C, the statements associated with a function are placed between a set of curly braces, `{ }`, which mark the beginning and end of the statements. Together, the curly braces and the statements are called a **block.** Let's take a look at the first line in our function's block:

```
5      printf("Hello World!\n");
```

This line of code is the heart of our program, the one that outputs our greeting to the user’s **console** (also known as the *terminal* in the context of Unix-like operating systems), the text-based interface installed on your computer. This statement is a **function call** and has two main parts: the name of the library function used to print our greeting, **printf**, followed by the data that we will pass to the function, seen here between the pair of parentheses. The data we are passing to the function is the **string**, “Hello World!\n”. The "\n" part at the end of the string is a special kind of character called an **escape sequence.** The "\n" escape sequence generates the new line at the end of our text. Strings and escape sequences will be covered in more detail

later. We terminate the function call statement with a semicolon so the compiler knows that it should begin looking for a new statement which it finds on the next line:

```
6     return 0;
```

Here, we say that our `main` function returns an integer value using the `return` keyword. The integer value we are returning is "0". But what does this mean, exactly? In the specific context of the `main` function, the value we return is called the **exit status**, which we report back to the operating system to indicate whether our code ran without error. As our programs grow in complexity, we can use other integers as codes to indicate various types of errors. This style of providing exit status is a long standing convention[1]. We will go into much more detail on return values of functions later in the book.

So that's a lot to take in, even for such a short program. Don't worry if you don't understand all of it and don't worry about memorizing it. You do not learn programming by memorizing, you learn by repetition and by doing. Memorizing all the notes to Beethoven's 5th symphony does not make you a concert pianist, you must get on the keyboard and practice and play!

Next we will show you how to take the source code you typed in and turn it into an executable file with your compiler.

Compiling

Compiling is the process we used to describe translating the orders you gave to the compiler in your source code into the machine language that can be run by your operating system and microprocessor. In this way, your C compiler is a middle-man. You talk to the compiler in a language it understands, C source code, and the compiler translates the source into machine code to save you a lot of painstaking, tedious work writing assembly code.

If the compiler finds your source code confusing, it will throw an error along with a message to help you fix up your source code and clear up any confusion. You will then need to try to recompile the code and repeat the process until it compiles without error. Note that code that compiles without error doesn't mean it's free of bugs. It just means the compiler understands the instructions provided by your source code.

Unix-like

If you are using a Unix(-like) system, such as GNU/Linux, Mac OS X, or Solaris, it will probably have GCC installed, otherwise on Linux you can install it using the package manager of your distribution. Open the virtual console or a terminal emulator and enter the following (be certain your current working directory is the one containing your source code):

```
gcc hello.c
```

By default gcc will generate our executable binary with the name *a.out*. To run your new generated program type:

```
./a.out
```

You should see `Hello, World!` printed after the last prompt.

To see the exit status of the last program you ran, type on your shell command:

```
echo $?
```

This shows the value the `main` function has returned, which is 0 in the above example.

There are a lot of options you can use with the gcc compiler. For example, if you want the output to have a name other than a.out, you can use the -o option. The following shows a few examples:

`-o`
indicates that the next parameter is the name of the resulting program (or library). If this option is not specified, the compiled program will, for historic reasons, end up in a file called "a.out" or "a.exe" (for cygwin users).

`-Wall`
indicates that gcc should warn about many types of suspicious code that are likely to be incorrect.

You can use these options to create a program called "helloworld" instead of "a.out" by typing:

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

Now you can run it by typing:

```
./helloworld
```

All the options are well documented in the manual[2] for GCC.

On IDEs

If you are using an IDE you may have to select console project, and to compile you just select build from the menu or the toolbar. The executable will appear inside the project folder, but you should have a menu button so you can just run the executable from the IDE. The process is roughly the same on all IDEs.

References

1. https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html
2. https://gcc.gnu.org/onlinedocs/