C

Many programs that run on linux systems are written in C, which is a language that has been around for almost 50 years, and is still widely used to this day. C is a language that allows for programmers to access low level system commands, yet still be human readable, and offers a fair amount of abstraction! C is compiled down to system level binaries.

Compilation

Before we do anything else, we need to know how to run our programs. C is not as simple as bash (where you just type the name of the program and it just runs). C is a compiled language. If you've used IDEs such as eclipse, c-lion, xcode, all of this compilation is hidden away from you! You press the "build" or "play" button, and the program just runs. However, what's going on behind the scenes is that a command is being run on some c code. So first, let's write a basic program, and it wouldn't be an introduction if our first program wasn't hello world

`helloWorld.c`

```c
#include <stdio.h>

int main(int argc, char** argv){
  printf("Hello World\n");

  return 0;
}
```

Just like c++ and java, c requires semicolons after every clause. We need to include `stdio.h` to use the printf function defined in there. Since our main function has an integer as a return type, we need to return something, so we're going to return 0. If you remember back to chapter 5, of mastering modern linux, all programs in linux return some exit status. These exit statuses range from 0 to 255. Later we will use predefined constants to illustrate our success and failure.

To compile the program, in your command line type:

```
gcc helloworld.c
```

This will produce an executable file `a.out`. In linux systems, it should be noted that compiled executables don't have the `.exe` file extension that windows executables do. To run an executable file in linux, you want to write:

```
./a.out
```

> Hello World

The `./` is just using the current directory, providing the shell with the full path name of the file (just like our bash scripts).

Suppose, however, that you didn't want your executable file to be named `a.out`. You could always rename it with the `mv` command, but that will become tedious. You can just as easily use the `gcc`'s `-o` option:

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

Now, to run your new file:

```
./hello
```

> Hello World

When using the gcc compiler, it breaks the compilation process down into 5 steps:

- Preprocessing
- Compilation
- Optimization
- Assembly
- Linking

Preprocessing

The c preprocessor works to include files, expand on definitions and macros, and work with any other preprocessor directives (i.e. the things that start with a `#`). This can be done manually with the `cpp` command:

```
cpp helloworld.c
```

The output of which is a ton. The #include preprocessor directive includes entire files. So far we've seen the general syntax of `#include <stdio.h>`. By using the syntax of `#include < fileName >`, we're signalling to the preprocessor that the file to be included isn't in our working directory, but is instead in one of the standard system direcotires. On the other hand, supposing that we wished to include a header file of our own making, we could use: `#include "filename"`, by wrapping it in quotation marks.

The preprocessor also allows for us to create **constants** and **macros** with `define`:

```c
#include <stdio.h>

#define SUCCESS 0
#define HI(x)((x)==0?(printf("hi\n")):(printf("goodbye\n")))

int main(int argc, char** argv){
  printf("Hello World\n");
  HI(1);
  return SUCCESS;
}
```

Now, we've defined in the preprocessor phase that there exists a **constant** `SUCCESS` that will let us return 0 (and is a bit more human readable). We've additionally created a **macro** called `HI`. Macro's work similarly to functions in that they take in an argument, and can react based off of that argument. The macro we defined is using a `ternary` which, depending on whether or not we pass in a 0, we either print `hi` or `goodbye`.

It is also possible to handle conditional inclusion of header files with:

```c
#if someCondition
// include or define some files here
#else
// include or define other stuff here
#endif
```

For the conditions, you can also attempt to see if there exists a predefined definition with `#ifdef`, and the converse with `#ifndef`. Our `SUCCESS` was kind of cool, but there already does exist something like that in the `stdlib.h` library. Let's write our code to include that:

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

#ifdef EXIT_SUCCESS
#define SUCCESS EXIT_SUCCESS
#else
#define SUCCESS -1
#endif

#define HI(x)((x)==0?(printf("hi\n")):(printf("goodbye\n")))

int main(int argc, char** argv){
  printf("Hello World\n");
  HI(1);
  printf("SUCCESS is %d", SUCCESS);
  return SUCCESS;
}
```

Above we checked to see if there was a definition for EXIT_SUCCESS, if there is, then we'll just define `SUCCESS` as that, but otherwise, we can defined `SUCCESS` as a `-1` (not that clever, but it'll get the point across).

NOTE: Notice also how we printed our number? The `%d` is acting as a placeholder for our value. We'll see much more of this as we move on, but this is one major difference from c++ that you may take a moment to get used to.

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

> Hello World
>
> goodbye
>
> SUCCESS is 0

You may be wondering what the point of the `#ifdef` conditions are. Consider the lifecycle of software. Developing software requires writing not only a lot of code, but also a lot of tests to ensure that your code is account for as many possibilities as we can fathom. You don't necessarily want include a number of header files required for running tests, or even printing out debug logs to the console, so you can instead wrap those logs as well as the included files inside of preprocessor #ifs that will never be included if a key is not defined.

Compilation and Optimization

Compilation is the phase which takes the output of the preprocessed files, parses through them and generates the compiled code. Depending on whether or not a particular flag was set for the compiler, the code could be optimized, but the optimization phase is entirely optional.

Assembly

This phase assembles the compiled code into an object (`.o`) file.

Linking

Linking is when the user supplied object files are then combined with system supplied object file from the imported libraries. Linking is done with the `ld` linker command.

Now that we've got the basics of *how* to compile a program, let's see how to write some `c`.