Introduction

This short tutorial will introduce you to the **UNIX** C programming environment. It's not exhaustive or super detailed, but it should get you started.

It's important to know your tools, libraries, and documentation. `gcc`, `gdb`, and `ld` are important tools for C compilation.

There are lots of library routines available, but luckily `libc` is linked with all C programs by default, so you just need to include the necessary header files.

Finally, understanding where to get library functions (like finding and reading `man` pages) is a skill worth mastering.

A Simple C Program

Let's start with a simple C program (hw.c). Unlike in Java, there is no connection between the file name and the file content, so use common sense when naming files.

The first line specifies an included file, in this case `stdio.h`, which "prototypes" many common input/output methods, including `printf()`. Using the `#include` directive tells the C preprocessor (`cpp`) to look for a certain file (e.g., `stdio.h`) and insert it straight into your code.

By default, `cpp` searches your current directory for the file.Type `pwd` in the terminal to show your current directory position.

The next section specifies the `main()` routine's signature, notably that it returns an integer (`int`) and that it will be called with two arguments.

1. `argc` is an integer representing the number of arguments on the command line
2. `argv` is an array of character pointers containing words from the command line, the last of which is null.

It then writes "hello, world" and advances the output stream to the next line, thanks to the backslash followed by a "n" after the `printf()` call. The program then returns a value to the shell that ran it.

In the `csh` and `tcsh` shells, this value is saved in the `status` variable to see if the program exited cleanly or with an error.

Compilation and Execution

Now we'll compile the program. On some platforms, you may be able to use a separate (native) compiler, cc, instead of gcc.

You simply type the following in the shell prompt:

```
gcc hw.c
```

`gcc` is a "compiler driver" that coordinates the various compilation procedures. There are usually 4–5 steps.

1. First, `gcc` runs `cpp`, the C preprocessor, to handle `#define` and `#include` directives. Because `cpp` is a source-to-source translator, the output is still source code (i.e., a C file).

2. Then the real compilation begins, generally with `cc1`. This converts source-level C code to machine-specific assembly code.

The assembler as will then be executed, producing object code (bits and things that machines can understand) The link-editor (or linker) ld will put it all together into a final executable program.

Fortunately, most people don't need to know how gcc works, they just use it with the right settings. Your compilation above produced an executable named a.out. To run the program, type in the terminal:

```
./a.out
```

To process the command-line inputs, the OS will set `argc` and `argv` appropriately when we start this program. `argc` will be 1, `argv[0]` will be the string "`./a.out`", and `argv[1]` will be null, signaling the array end.

Useful Flags

Before going on to C, let's review some useful `gcc` compilation settings.

```
gcc -o hw hw.c  # -o: to specify the executable name
gcc -Wall hw.c  # -Wall: gives much better warnings
gcc -g hw.c     # -g: to enable debugging with gdb
gcc -O hw.c     # -O: to turn on optimization
```

You can mix and match these flags (e.g., gcc -o hw -g -Wall hw.c). Use `-Wall`, which provides you a lot more warnings about possible problems. Don't ignore the warnings! Instead, correct them and let them go.

Linking with Libraries

Your program may need to use a library routine. Because the C library contains so many routines, finding the proper `#include` file is usually easy. The manual pages (or man pages) are the best way to do this.

Assume you want to use the `fork()` system call. At the shell prompt, type man fork to see a text description of `fork()`. A little code snippet at the top will tell you which files to `#include` in your application to make it compile. For `fork()`, you must `#include` the file `unistd.h` as follows:

```
#include <unistd.h>
```

Some library routines aren't in the C library, so you'll have to do some extra work. For example, the math library provides handy sine, cosine, and tangent routines. If you wish to use `tan()` in your code, read the `man` page first. The top of the Linux man page for `tan` reads:

```
#include <math.h>
...
Link with -lm.
```

The first line says — you need to `#include` the math library (`/usr/include/math.h`). The next line, however, tells you how to "link" your software with the math library. Many useful libraries reside in `/usr/lib`, including the math library.

Static & Dynamically Linked Libraries

Libraries are classified as either statically linked (.a) or dynamically linked (.b) (which end in .so).

- **Static-linked libraries** are combined directly into your executable; that is, the linker combines the low-level library code with your executable and the outcome is a large executable.
- **Dynamic linking** improves on this by just referencing a library in an executable; when your program is run, the operating system loader dynamically links in the library.
    - This method is preferred to the static method because it saves disk space (no need for large executables) and allows applications to share library code and static data in memory.

The math library has both static and dynamic versions, namely /usr/lib/libm.a and /usr/lib/libm.so. To link with the math library, you first have to indicate it to the link-editor by running gcc with the appropriate flags.

```
gcc -o hw hw.c -Wall -lm
```

The `-lm` flag instructs the linker to look for `libm.so` or `libm.a`. If you prefer a static library over a dynamic one, there is another flag you may use – see if you can find out what it is. Because dynamic libraries have a modest performance cost, people occasionally prefer the static version.

Finally, you can use the compiler flags `-I/foo/bar` and `-L/foo/bar` to check for headers and libraries in the directory `/foo/bar`, respectively. UNIX shorthand for the current directory is "`.`" (dot). The `-I` flag goes on the build line, while the `-L` flag goes on the link.

Separate Compilation

If a program becomes large enough, it may be split into different files, compiled individually, and linked together. For example, you want to compile `hw.c` and `helper.c` separately, then link them together.

```
# we are using -Wall for warnings, -O for optimization
> gcc -Wall -O -c hw.c
> gcc -Wall -O -c helper.c
> gcc -o hw hw.o helper.o -lm
```

The `-c` flag tells the compiler to produce object files, in this case, `hw.o` and `helper.o`. Each of these files is a machine-level version of the code contained in a source file.

To make an executable, you have to "link" the object files. We've done this on the last command above (`gcc -o hw hw.o helper.o`).

Because the input files are object files (`.o`), `gcc` skips the last step and uses the link-editor `ld` to link them together into a single executable. This line is often referred to as the "link line" because it is where link-specific instructions like `-lm` are specified.

Similarly, build-only flags like `-Wall` and `-O` need only be put on compile lines, not link lines.

You could specify all the C source files in a single line (`gcc -Wall -O -o hw hw.c helper.c`), but this needs the system to recompile each file, which takes a lot of time. Compiling each one separately saves time and increases productivity by just re-compiling files that have changed during editing. This process is best managed by another program, `make`.

Makefiles

Make is a powerful tool that automates most of the build process. Let's examine a simple example saved in a Makefile file.

Building with `make`
To build your program, type make on the command line.

```
make
```

This will default to looking for `Makefile` or `makefile` as input. Most of these notes are from the `gmake` info page; check the Documentation section to see how to find them. `gmake` and `make` are identical on Linux systems.

Makefiles use rules to decide what needs to be done. A rule's general form:

```
target: prerequisite1 prerequisite2 ...
command1
command2 ...
```

- **Target** - The name of a command-generated file, such as an executable or object file.
  - A target can also be an action name, like "clean" in our example.
- **Prerequisite** - This file is used to create the target. A target often requires multiple files.
  - To generate the executable `hw`, we need to build two object files first: `hw.o` and `helper.o`.
- **Command** - An action that make carries out. A rule may include several commands on separate lines.

Every command line **must** start with a tab character! If you merely put spaces, make will terminate with an unclear error message.

How it works

When `make` is run, it goes like this:

1. When it gets to the target `hw`, it finds it needs two prerequisites, `hw.o` and `helper.o`, to create it. `hw` relies on those two object files.

2. Then, `Make` will evaluate each target. Examining `hw.o` reveals it depends on `hw.c`.

3. If `hw.c` has been modified more recently than `hw.o`, make will realize that `hw.o` is out of date and should be generated again. It will execute the command, `gcc -O -Wall -c hw.c`, which generates `hw.o`.
    - `Make` will know which object files need to be re-generated based on their dependencies, and will only do the work required to rebuild the executable. If `hw.o` does not exist at all, it will be created.
4. Continuing on, helper.o may be regenerated or created using the same conditions.

`Make` is now ready to run the command to generate the final executable, and it does so: `gcc -o hw hw.o helper.o -lm`

Clean

To use the `clean` target, you have to ask for explicitly. In the terminal, type:

```
make clean
```

Because the clean target has no prerequisites, typing make `clean` will always execute the command(s). The `clean` target removes the object files and executable, which is useful if you want to recreate the program from scratch.

Why Makefiles?

`makefiles` may seem cumbersome but fortunately, there are many shortcuts that make it easier to use. For example, the `makefile` to the left does the same thing but is more user friendly.

This makefile can make your life a little easier.
* The `SRCS` variable at the top of the makefile, for example, allows you to easily add additional source files to your build.
* Change the `TARG` line to change the executable name, compiler, flags, and library specifications.
* Finally, figuring out a target's prerequisites isn't always easy, especially in large programs. Another program called `makedepend` helps with this.

Debugging

Even with a strong build environment and a correctly compiled program, your program may be buggy. Fortunately, gdb, the GNU debugger, can help.

Let's take the following buggy code, saved in `buggy.c`, and compiled into the executable `buggy`.

In this case, the main program dereferences the NULL variable `p`, causing a segmentation fault. This problem may be easy to find, but in a more complex application, it may not be.

Recompile your software with the `-g` flag on each compile line to prepare for debugging. This adds extra debugging information to your executable for debugging. Also, don't use optimization (`-O`). It may work, but it may also cause issues when debugging.

You can use the debugger after recompiling with `-g`. Run `gdb` as follows:

```
gdb buggy
```

This launches an interactive debugger session. Once inside, you may see something like:

```
GNU gdb ...
Copyright 2008 Free Software Foundation, Inc.
(gdb)
```

The first step is to run the program. Simply type `run` at the gdb command prompt. You might see something similar to:

```
Starting program: buggy

Program received signal SIGSEGV, Segmentation fault.
0x8048433 in main (argc=1, argv=0xbffff844) at buggy.c:19
19          printf("%d\n", p->x);
```

A "segmentation fault" was generated at the line where we tried to dereference `p`, according to `gdb`. This just means we accessed memory we shouldn't have.

If you weren't sure what was going on, you may look into some variables. `gdb` lets you to do this interactively during the debug session. Type `print p` in the `gdb` command line.

Using the `print` primitive shows us that `p` is a pointer to a Data `struct`, and that it is now set to `NULL` (or zero, or hex 0, which is shown as "`0x0`").