Useful Things in the C Programming Environment

Remzi Arpaci-Dusseau

Introduction

This is a very brief document to familiarize you with the basics of the C programming environment on UNIX-based systems. It is not comprehensive or detailed, but should just give you enough to get you going.

A couple of general points of advice about programming: if you want to become an expert programmer, you need to master more than just the syntax of a language. Specifically, you should **know your tools**, **know your libraries**, and **know your documentation**. The tools that are relevant to C and C++ compilation are `gcc`, `gdb`, and maybe `ld`. There are tons of library routines that are also available to you, but fortunately a lot of functionality is included in `libc`, which is linked with all C and C++ programs by default – all you need to do is include the right header files. Finally, knowing how to find the library routines you need (*e.g.*, learning to find and read man pages) is a skill worth acquiring. We'll talk about each of these in more detail later on.

Like (almost) everything worth doing in life, becoming an expert in these domains takes time, but the result of spending the time up-front will definitely be well worth the effort.

A Simple C Program

We'll start with a simple C program, perhaps saved in the file "hw.c". Unlike Java, there is no connection between the file name and the contents of the file; because of that, just use your common sense in naming files.

```
/* header file -- specifies headers for routines */
/* note that C comments are enclosed within a slash
   and a star, and may wrap over lines */
// if you use gcc, two slashes will work too
#include <stdio.h>

/* main returns an integer */
int
main(int argc, char *argv[])
{
    /* printf is our output function; by default, writes to standard out */
    /* printf returns an integer, but since we ignore it here */
    printf(``hello, world\n'');

    /* return 0 to indicate all went well */
    return(0);
}
```

The first line specifies a file to include, in this case `stdio.h`, which "prototypes" many of the commonly used input/output routines; the one we are interested in is `printf`. When you use the `#include` directive, you are telling the C preprocessor to find a particular file and to insert it directly into your code at the spot of the `#include`. By default, `cpp` will look in `/usr/include/` to try to find the file.

The next part specifies the signature of the `main` routine, namely that it returns an integer (`int`), and will be called with two arguments, an integer `argc`, which is a count of the number of arguments on the command line, and an array of pointers to characters, each of which contain a word from the command line, and the last of which is null. There will be more on pointers and arrays below.

The program then simply prints the string "hello, world" and advances the output stream to the next line, courtesy of the backslash followed by an "n" at the end of the call to `printf`. Afterwards, the program completes by returning a value, which is passed back to the shell that executed the program. A script or the user at the terminal could check this value (in many shells, it is stored in the `status` variable), to see whether the program exited cleanly or with an error.

Compilation and Execution

We now learn how to compile the program. Note that we will use `gcc` as our example, though on some platforms you may be able to use the native compiler, `cc`.

At the shell prompt, you just type:

```
prompt> gcc hw.c
```

`gcc` is not really the compiler, but rather the program called a "compiler driver"; thus it coordinates the many steps of the compilation. Usually there are four to five steps. First, `gcc` will execute `cpp`, the C pre-processor, to process certain directives (such as `#define` and `#include`). `cpp` is just a source-to-source translator, so its end-product is still just source code. Then the real compilation will begin, usually a command called `cc1`. This will transform source-level C code into low-level assembly code, specific to the host machine. The assembler `as` will then be executed, generating object code (bits and things that machines can really understand), and finally the link-editor (or linker) `ld` will put it all together into a final executable program. Fortunately(!), for most purposes, you can blithely be unaware of how `gcc` works, and just use it with the proper flags.

The result of your compilation above is an executable, named `a.out`. To then run the program, we simply type:

```
prompt> ./a.out
```

When we run this program, `argc` is equal to `1`, `argv[0]` is the string `./a.out`, and `argv[1]` is null, indicating the end of the array.

Useful Flags

Before moving on to the C language, we'll first point out some useful compilation flags for `gcc`.

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

Of course, you may combine these flags as you see fit (*e.g.*, `gcc -o hw -g -Wall hw.c`). Of these flags, you should always use `-Wall`, which gives you lots of extra warnings about possible mistakes. Don't ignore the warnings! Instead, fix them and thus make them go away.

Linking with Libraries

Sometimes, you may want to use a library routine in your program. Because so many routines are available in the C library (which is automatically linked with every program), all you usually have to do is find the right `#include` file. The best way to do that is via the `man` pages.

For example, let's say you want to use the `fork` system call.[1] By typing `man fork` at the prompt, you will get back a "manual" page describing how `fork` works. At the very top will be a short code snippet, and that will tell you which files you need to `#include` in your program in order to get it to compile. In the case of `fork`, you need to `#include` both `sys/types.h` and `unistd.h`.

However, some library routines do not reside in the C library, and therefore you will have to do a little more work. For example, the math library has many useful routines, such as sines, cosines, tangents, and the like. If you want to include the routine `tan` in our code, you should again first check the man page. At the top of the Solaris man page for `tan`, you will see the following two lines:

```
cc [ flag ... ] file ... -lm [ library ... ]
#include <math.h>
```

The second line you already should understand – you need to `#include` the math library, which is found in the standard place (`/usr/include/math.h`). However, what the first line is telling you is how to "link" your program with the math library. A number of useful libraries exist and can be linked with; many of those reside in `/usr/lib`. In this case, that is where the math library resides.

There are two types of libraries: statically-linked libraries (which end in `.a`), and dynamically-linked ones (which end in `.so`). Statically-linked libraries are combined directly into your executable; that is, the low-level code for the library is inserted into your executable by the linker, and results in a much larger binary object. Dynamic linking improves on this by just including the reference to a library in your program executable; when the program is run, the system dynamically links in the library. This method is preferred over the static approach because it saves disk space (no unnecessarily large executables are made) and allows applications to share library code and static data when in memory. In the case of the math library, both static and dynamic versions are available; that static library is called `/usr/lib/libm.a` and `/usr/lib/libm.so`. As you might have figured out, the standard place to find libraries in UNIX systems is `/usr/lib/`.

In any case, to link with the math library, you need to specify the library to the link-editor; this can be achieved by invoking `gcc` with the right flags.

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

The `-lXXX` flags tells the linker to look for `libXXX.so` or `libXXX.a`, probably in that order. If for some reason you insist on the static library over the dynamic one, there is another flag you can add – see you if you can find out what it is.

One final note: if you want the compiler to search for headers in a different path than the usual places, or want it to link with libraries that you specify, you can use the `-I/search/for/headers/here/` and `-L/look/for/libs/here` flags, respectively. The `-I` flag should go on the compile line, and the `-L` flag on the link line.

[1] Note that `fork` is a system call, and not just a library routine. However, the C library provides C wrappers for all the system calls, each of which simply trap into the operating system

Separate Compilation

Once a program starts to get large enough, you may want to split it into separate files, compiling each separately, and then link them together. For example, say you have two files, `hw.c` and `helper.c`, and you wish to compile them individually, and then link them together.

```
# note that we are using -Wall for warnings and -O for optimization
prompt> gcc -Wall -O -c hw.c
prompt> gcc -Wall -O -c helper.c
prompt> gcc -o hw hw.o helper.o -lm
```

The `-c` flag tells the compiler just to produce an object file – in this case, files called `hw.o` and `helper.o`. These files are not executables, but just machine-level representations of the code within each source file. To combine the object files into an executable, you have to "link" them together; this is accomplished with the third line (`gcc -o hw hw.o helper.o`). In this case, `gcc` sees that the input files specified are not source files (`.c`), but instead are object files (`.o`), and therefore skips right to the last step and invoked the link-editor `ld` to link them together into a single executable. Because of its function, this line is often called the "link line", and would be where you specify link-specific commands such as `-lm`. Analagously, flags such as `-Wall` and `-O` are only needed in the compile phase, and therefore need not be included on the link line.

Of course, you could just specify all the C source files on a single line to `gcc` (`gcc -Wall -O -o hw hw.c helper.c`), but this requires the system to recompile every source code file, a time-consuming process. By compiling each individually, you can save time by only recompiling those files that have changed during your editing, and thus increase your productivity. This process is best managed by another program, `make`, which we now describe.

Makefiles

The program `make` lets you automate much of your build process, and is thus a crucially important tool for any serious program (and programmer). Let's take a look at a simple example, saved in a file called `Makefile`.

```
hw: hw.o helper.o
    gcc -o hw hw.o helper.o -lm

hw.o: hw.c
    gcc -O -Wall -c hw.c

helper.o: helper.c
    gcc -O -Wall -c helper.c

clean:
    rm -f hw.o helper.o hw
```

To build your program, now all you have to do is type:

```
prompt> make
```

This will (by default) look for `Makefile` or `makefile`, and use that as its input (you can specify a different makefile with a flag; read the man pages to find out which). The gnu version of `make`, `gmake`, is more fully featured than traditional make, so we will focus upon it for the rest of this discussion (though we will use the two terms interchangeably). Most of these notes are based on the `gmake` info page; to see how to find those pages, see the Documentation section below.

Makefiles are based on rules, which are used to decide what needs to happen. The general form of a rule:

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

A "target" is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as 'clean' in our example.

A "prerequisite" is a file that is used as input to create the target. A target often depends on several files. For example, to build the executable `hw`, we need two object files to be built first: `hw.o` and `helper.o`.

Finally, a "command" is an action that `make` carries out. A rule may have more than one command, each on its own line. **Important:** You have to put a single tab character at the beginning of every command line! If you just put spaces, `make` will print out some obscure error message and exit.

Usually a command is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies commands for the target need not have prerequisites. For example, the rule containing the delete command associated with the target 'clean' does not have prerequisites.

Going back to our example, when make is executed, it roughly works like this: First, it comes to the target `hw`, and it realizes that that to build it, it must have two prerequisites, `hw.o` and `helper.o`. Thus, `hw` depends on those two object files. Make then will examine each of those targets. In examing `hw.o`, it will see that it depends on `hw.c`. Here is the key: if `hw.c` has been modified more recently than `hw.o` has been created, `make` will know that `hw.o` is out of date and should be generated anew; in that case, it will execute the command line, `gcc -O -Wall -c hw.c`, which generates `hw.o`. Thus, if you are compiling a large program, `make` will know which object files need to be re-generated based on their dependencies, and will only do the necessary amount of work to recreate the executable. Also note that `hw.o` will be created in the case that it does not exist at all.

Continuing along, `helper.o` may also be regenerated or created, based on the same criteria as defined above. When both of the object files have been created, make is now ready to execute the command to create the final executable, and goes back and does so: `gcc -o hw hw.o helper.o -lm`.

Up until now, we've been ignoring the `clean` target in the makefile. To use it, you have to ask for it explictly. Type

```
prompt> make clean
```

This will execute the command on the command line. Because there are no prerequisites for the `clean` target, typing `make clean` will always result in the command being executed. In this case, the `clean` target is used to remove the object files and executable. It is handy if you want to rebuild the entire program from scratch.

Now you might be thinking, "well, this seems ok, but these makefiles sure are cumbersome!" And you'd be right – if they always had to be written like this. Fortunately, there are a lot of short cuts that make `make` even easier to use. For example, this makefiles has the same functionality but is a little nicer to use:

```
# specify all source files here
SRCS = hw.c helper.c

# specify target here (name of executable)
TARG = hw

# specify compiler, compile flags, and needed libs
CC   = gcc
OPTS = -Wall -O
LIBS = -lm

# this translates .c files in src list to .o's
OBJS = $(SRCS:.c=.o)

# all is not really needed, but is used to generate the target
all: $(TARG)

# this generates the target executable
$(TARG): $(OBJS)
	$(CC) -o $(TARG) $(OBJS) $(LIBS)

# this is a generic rule for .o files
%.o: %.c
	$(CC) $(OPTS) -c $< -o $@

# and finally, a clean line
clean:
	rm -f $(OBJS) $(TARG)
```

Though we won't go into the details of make syntax, as you can see, this makefile can make your life somewhat easier. For example, it allows you to easily add new source files into your build, simply by adding them to the `SRCS` variable at the top of the makefile. You can also easily change the name of the executable by changing the `TARG` line, and the compiler, flags, and library specifications are all easily modified.

One final word about `make`: figuring out a target's prerequisites is not always trivial, especially in large and complex programs. Not surprisingly, there is another tool that helps with this, called `makedepend`. Read about `makedepend` on your own and see if you can incorporate it into a makefile.

Debugging

Finally, after you have a good build environment, and a correctly compiled program, you may find that your program is buggy. One way to fix the problem(s) is to think really hard – this method is sometimes successful, but often not. The problem is a lack of *information*; you just don't know exactly what is going on within the program, and therefore cannot figure out why it is not behaving as expected. Fortunately, there is some help: `gdb`, the GNU debugger.

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

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

```
struct Data {
    int x;
};

int
main(int argc, char *argv[])
{
    struct Data *p = NULL;
    printf("%d\n", p->x);
}
```

In this example, the main program dereferences the variable `p` before when it is NULL, which will lead to a segmentation fault. Of course, this problem should be easy to fix by inspection, but in a more complex program, finding such a problem is not always easy.

To prepare yourself for a debugging session, recompile your program and make sure to pass the `-g` flag to each compile line. This includes extra debugging information in your executable that will be useful during your debugging session. Also, don't turn on optimization (`-O`); though this may work, it may also lead to confusion during debugging.

After re-compiling with `-g`, you are ready to use the debugger. Fire up `gdb` at the command prompt as follows:

```
prompt> gdb buggy
```

This puts you inside an interactive session with the debugger. Note that you can also use the debugger to examine "core" files that were produced during bad runs, or to attach to an already-running program; read the documentation to learn more about this.

Once inside, you may see something like this:

```
prompt> gdb buggy
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
[stuff deleted]
(gdb)
```

The first thing you might want to do is to go ahead and run the program. To do this, simply type "run" at gdb command prompt. In this case, this is what you might see:

```
(gdb) run
Starting program: buggy

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

As you can see from the example, in this case, `gdb` immediately pinpoints where the problem occurred; a "segmentation fault" was generated at the line where we tried to dereference `p`. This just means that we accessed some memory that we weren't supposed to access. At this point, the astute programmer can examine the code, and say "aha! it must be that `p` does not point to anything valid!", and then go ahead and fix the problem.

However, if you didn't know what was going on, you might want to examine some variable. `gdb` allows you to do this interactively during the debug session.

```
(gdb) print p
$1 = (Data *) 0x0
```

By using the print primitive, we can examine `p`, and see both that it is a pointer to a struct of type Data, and that it is currently set to NULL (0x0).

Finally, you can also set breakpoints within your program to have the debugger stop the program at a certain routine. After doing this, it is often useful to step through the execution (one line at a time), and see what is happening.

```
(gdb) break main
Breakpoint 1 at 0x8048426: file buggy.cc, line 17.
(gdb) run
Starting program: /homes/hacker/buggy

Breakpoint 1, main (argc=1, argv=0xbffff844) at buggy.cc:17
17          struct Data *p = NULL;
(gdb) next
19          printf("%d\n", p->x);
(gdb)

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

In the example above, a breakpoint is set at the main() routine; thus, when we run the program, the debugger almost immediately stops execution at main. At that point in the example, a "next" command is issued, which executes the next source-level command. Both "next" and "step" are useful ways to advance through a program – read about them in the documentation for more details.[2]

This discussion really does not do `gdb` justice; it is a rich and flexible debugging tool, with many more features than can be described in the limited space here. Read more about it on your own and become an expert in your copious spare time.

Documentation

To learn a lot more about all of these things, you have to do two things: the first is to use these tools, and the second is to read more about them on your own. One way to find out more about `gcc`, `gmake`, and `gdb` is to read their man pages; type `man gcc`, `man gmake`, or `man gdb` at your command prompt. You can also use `man -k` to search the man pages for keywords, though that doesn't always work as well as it might.

One tricky thing about man pages: typing "man XXX" may not result in the thing you want, if there is more than one thing called XXX. For example, if you are looking for the kill() system call man page, and if you just type "man kill" at the prompt, you will get the wrong man page(!), because there is a command-line

[2] In particular, you can use the interactive "help" command while debugging with `gdb`.

program called kill. Man pages are divided into ”sections”, and by default, man will return the man page in the lowest section that it finds, which in this case is section 1. Note that you can tell which man page you got by looking at the top of the page: if you see ”kill(2)”, you know you are in the right man page. Type ”man man” to learn more about what is stored in each of the different sections of the man pages. Also note that ”man -a kill” can be used to cycle through all of the different man pages named ”kill”.

Also quite useful are the “info pages”, which provide much more detailed documentation on many GNU tools. You can access the info pages by running the program `info`, or via `emacs`, the preferred editor of hackers, by executing `Meta-x info`. A program like `gcc` has hundreds of flags, and some of them are surprisingly useful to know about. `gmake` has many more features that will improve your build environment. And `gdb` is quite a sophisticated debugger. Read the man and info pages, try out features that you hadn’t tried before, and become a power user of your programming tools.

Suggested Readings

Other than the man and info pages, there are a number of useful books out there. Note that a lot of this information is available for free on-line; however, sometimes having something in book form seems to make it easier to learn. Also, always look for O’Reilly books on topics you are interested in; I have found them to be uniformly of high quality.

- *The C Programming Language*, by Brian Kernighan and Dennis Ritchie. This is **the** C book to have.
- *Managing Projects with make*, by Andrew Oram and Steve Talbott. A reasonable and short book on make.
- *Debugging with GDB: The GNU Source-Level Debugger*, by Richard M. Stallman, Roland H. Pesch. A little book on using GDB.
- *Advanced Programming in the UNIX Environment*, by W. Richard Stevens. Stevens writes excellent books, and this is a must for Unix hackers. He also has an excellent set of books on TCP/IP and Sockets programming.