How to structure a multi-file C program: Part 1

Grab your favorite beverage, editor, and compiler, crank up some tunes, and start structuring a C program composed of multiple files.

29 Jul 2019 | Erik O'Shaughnessy | 115 | 2 comments

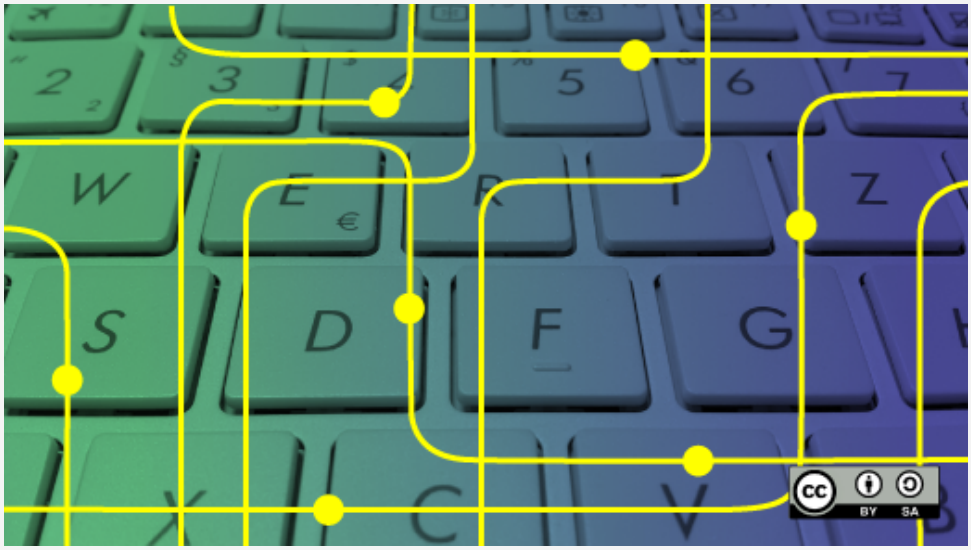


Image by : *Opensource.com*

It has often been said that the art of computer programming is part managing complexity and part naming things. I contend that this is largely true with the addition of "and sometimes it requires drawing boxes."

In this article, I'll name some things and manage some complexity while writing a small C program that is loosely based on the program structure I discussed in "How to write a good C main function"—but different. This one will do something. Grab your favorite beverage, editor, and compiler, crank up some tunes, and let's write a mildly interesting C program together.

Philosophy of a good Unix program

The first thing to know about this C program is that it's a Unix command-line tool. This means that it runs on (or can be ported to) operating systems that provide a Unix C runtime environment. When Unix was invented at Bell Labs, it was imbued from the beginning with a design philosophy. In my own words: *programs do one thing, do it well, and act on files*. While it makes sense to do one thing and do it well, the part about "acting on files" seems a little out of place.

It turns out that the Unix abstraction of a "file" is very powerful. A Unix file is a stream of bytes that ends with an end-of-file (EOF) marker. That's it. Any other structure in a file is imposed by the application and not the operating system. The operating system provides system calls that allow a program to perform a set of standard operations on files: open, read, write, seek, and close (there are others, but those are the biggies). Standardizing access to files allows different programs to share a common abstraction and work together even when different people implement them in different programming languages.

Having a shared file interface makes it possible to build programs that are *composable*. The output of one program can be the input of another program. The Unix family of operating systems provides three files by default whenever a program is executed: standard in (**stdin**), standard out (**stdout**), and standard error (**stderr**). Two of these files are opened in write-only mode: **stdout** and **stderr**, while **stdin** is opened read-only. We see this in action whenever we use file redirection in a command shell like Bash:

```
$ ls | grep foo | sed -e 's/bar/baz/g' > ack
```

This construction can be described briefly as: the output of **ls** is written to stdout, which is redirected to the stdin of **grep**, whose stdout is redirected to **sed**, whose stdout is redirected to write to a file called **ack** in the current directory.

We want our program to play well in this ecosystem of equally flexible and awesome programs, so let's write a program that reads and writes files.

MeowMeow: A stream encoder/decoder concept

When I was a dewy-eyed kid studying computer science in the <mumbles>s, there were a plethora of encoding schemes. Some of them were for compressing files, some were for

packaging files together, and others had no purpose but to be excruciatingly silly. An example of the last is the MooMoo encoding scheme.

To give our program a purpose, I'll update this concept for the 2000s and implement a concept called MeowMeow encoding (since the internet loves cats). The basic idea here is to take files and encode each nibble (half of a byte) with the text "meow." A lower-case letter indicates a zero, and an upper-case indicates a one. Yes, it will balloon the size of a file since we are trading 4 bits for 32 bits. Yes, it's pointless. But imagine the surprise on someone's face when this happens:

```
$ cat /home/your_sibling/.super_secret_journal_of_my_innermost_thoughts
MeOWmeOWmeowMEoW...
```

This is going to be awesome.

Implementation, finally

The full source for this can be found on GitHub, but I'll talk through my thought process while writing it. The object is to illustrate how to structure a C program composed of multiple files.

Having already established that I want to write a program that encodes and decodes files in MeowMeow format, I fired up a shell and issued the following commands:

```
$ mkdir meowmeow
$ cd meowmeow
$ git init
$ touch Makefile     # recipes for compiling the program
$ touch main.c       # handles command-line options
$ touch main.h       # "global" constants and definitions
$ touch mmencode.c   # implements encoding a MeowMeow file
$ touch mmencode.h   # describes the encoding API
$ touch mmdecode.c   # implements decoding a MeowMeow file
$ touch mmdecode.h   # describes the decoding API
$ touch table.h      # defines encoding lookup table values
$ touch .gitignore   # names in this file are ignored by git
$ git add .
$ git commit -m "initial commit of empty files"
```

In short, I created a directory full of empty files and committed them to git.

Even though the files are empty, you can infer the purpose of each from its name. Just in case you can't, I annotated each **touch** with a brief description.

Usually, a program starts as a single, simple **main.c** file, with only two or three functions that solve the problem. And then the programmer rashly shows that program to a friend or her boss, and suddenly the number of functions in the file balloons to support all the new "features" and "requirements" that pop up. The first rule of "Program Club" is don't talk about "Program Club." The second rule is to minimize the number of functions in one file.

To be honest, the C compiler does not care one little bit if every function in your program is in one file. But we don't write programs for computers or compilers; we write them for other people (who are sometimes us). I know that is probably a surprise, but it's true. A program embodies a set of algorithms that solve a problem with a computer, and it's important that people understand it when the parameters of the problem change in unanticipated ways. People will have to modify the program, and they will curse your name if you have all 2,049 functions in one file.

So we good and true programmers break functions out, grouping similar functions into separate files. Here I've got files **main.c**, **mmencode.c**, and **mmdecode.c**. For small programs like this, it may seem like overkill. But small programs rarely stay small, so planning for expansion is a "Good Idea."

But what about those **.h** files? I'll explain them in general terms later, but in brief, those are called *header* files, and they can contain C language type definitions and C preprocessor directives. Header files should *not* have any functions in them. You can think of headers as a definition of the application programming interface (API) offered by the **.c** flavored file that is used by other **.c** files.

But what the heck is a Makefile?

I know all you cool kids are using the "Ultra CodeShredder 3000" integrated development environment to write the next blockbuster app, and building your project consists of mashing on Ctrl-Meta-Shift-Alt-Super-B. But back in my day (and also today), lots of useful work got done by C programs built with Makefiles. A Makefile is a text file that contains recipes for working with files, and programmers use it to automate building their program binaries from source (and other stuff too!).

Take, for instance, this little gem:

```
00 # Makefile
01 TARGET= my_sweet_program
02 $(TARGET): main.c
03    cc -o my_sweet_program main.c
```

Text after an octothorpe/pound/hash is a comment, like in line 00.

Line 01 is a variable assignment where the variable **TARGET** takes on the string value **my_sweet_program**. By convention, OK, my preference, all Makefile variables are capitalized and use underscores to separate words.

Line 02 consists of the name of the file that the recipe creates and the files it depends on. In this case, the target is **my_sweet_program**, and the dependency is **main.c**.

The final line, 03, is indented with a tab and not four spaces. This is the command that will be executed to create the target. In this case, we call **cc** the C compiler frontend to compile and link **my_sweet_program**.

Using a Makefile is simple:

```
$ make
cc -o my_sweet_program main.c
$ ls
Makefile  main.c  my_sweet_program
```

The Makefile that will build our MeowMeow encoder/decoder is considerably more sophisticated than this example, but the basic structure is the same. I'll break it down Barney-style in another article.

Form follows function

My idea here is to write a program that reads a file, transforms it, and writes the transformed data to another file. The following fabricated command-line interaction is how I imagine using the program:

```
$ meow < clear.txt > clear.meow
$ unmeow < clear.meow > meow.tx
$ diff clear.txt meow.tx
$
```

We need to write code to handle command-line parsing and managing the input and output streams. We need a function to encode a stream and write it to another stream. And finally, we need a function to decode a stream and write it to another stream. Wait a second, I've only been talking about writing one program, but in the example above, I invoke two commands: **meow** and **unmeow**? I know you are probably thinking that this is getting complex as heck.

Minor sidetrack: argv[0] and the ln command

If you recall, the signature of a C main function is:

```
int main(int argc, char *argv[])
```

where **argc** is the number of command-line arguments, and **argv** is a list of character pointers (strings). The value of **argv[0]** is the path of the file containing the program being executed. Many Unix utility programs with complementary functions (e.g., compress and uncompress) look like two programs, but in fact, they are one program with two names in the filesystem. The two-name trick is accomplished by creating a filesystem "link" using the **ln** command.

An example from **/usr/bin** on my laptop is:

```
   $ ls -li /usr/bin/git*
3376 -rwxr-xr-x. 113 root root     1.5M Aug 30  2018 /usr/bin/git
3376 -rwxr-xr-x. 113 root root     1.5M Aug 30  2018 /usr/bin/git-receive-pack
...
```

Here **git** and **git-receive-pack** are the same file with different names. We can tell it's the same file because they have the same inode number (the first column). An inode is a feature of the Unix filesystem and is super outside the scope of this article.

Good and/or lazy programmers can use this feature of the Unix filesystem to write less code but double the number of programs they deliver. First, we write a program that changes its behavior based on the value of **argv[0]**, then we make sure to create links with the names that cause the behavior.

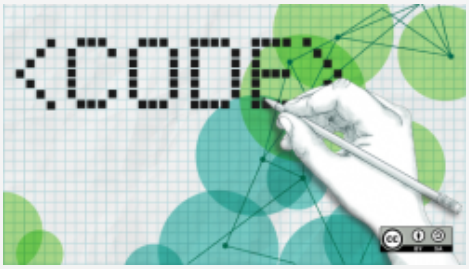
In our Makefile, the **unmeow** link is created using this recipe:

```
# Makefile
...
$(DECODER): $(ENCODER)
        $(LN) -f $< $@
        ...
```

I tend to parameterize everything in my Makefiles, rarely using a "bare" string. I group all the definitions at the top of the Makefile, which makes it easy to find and change them. This makes a big difference when you are trying to port software to a new platform and you need to change all your rules to use **xcc** instead of **cc**.

The recipe should appear relatively straightforward except for the two built-in variables **$@** and **$<**. The first is a shortcut for the target of the recipe; in this case, **$(DECODER)**. (I remember this because the at-sign looks like a target to me.) The second, **$<** is the rule dependency; in this case, it resolves to **$(ENCODER)**.

Things are getting complex for sure, but it's managed.



How to write a good C main function

Learn how to structure a C file and write a C main function that handles command line arguments like a champ.

Erik O'Shaughnessy

What is open source programming?

Open source is more than just chucking some code up on GitHub. Learn what it is—and what it's not.

Jim Salter

Topics : Programming

About the author

Erik O'Shaughnessy - Erik O'Shaughnessy is an opinionated but friendly UNIX system programmer living the good life in Texas. Over the last twenty years (or more!) he has worked for IBM, Sun Microsystems, Oracle, and most recently Intel doing computer system performance related work. He is; a mechanical keyboard aficionado, a gamer, a father, a husband, voracious reader, student of Okinawian karate, and seriously grouchy in the morning before coffee.

• More about me

Recommended reading

Solve a real-world problem using Java

Use Python to solve a charity's business problem

How this open source test framework evolves with .NET

Managing a non-profit organization's supply chain with Groovy

A practical guide to learning awk

Optimize runtime performance with C++'s move semantics

2 Comments

Diagnostic Solutions on 29 Jul 2019 1

Nice article

Châu on 29 Jul 2019 1

Very good article and I wait for next parts. If you have time please write article about your adventures/work in Sun Microsystems.

Get the highlights in your inbox every week.

Find us:

For more discussion on open source and the role of the CIO in the enterprise, join us at The EnterprisersProject.com.