How to structure a multi-file C program: Part 2

Dive deeper into the structure of a C program composed of multiple files in the second part of this article.

31 Jul 2019 | Erik O'Shaughnessy | 96

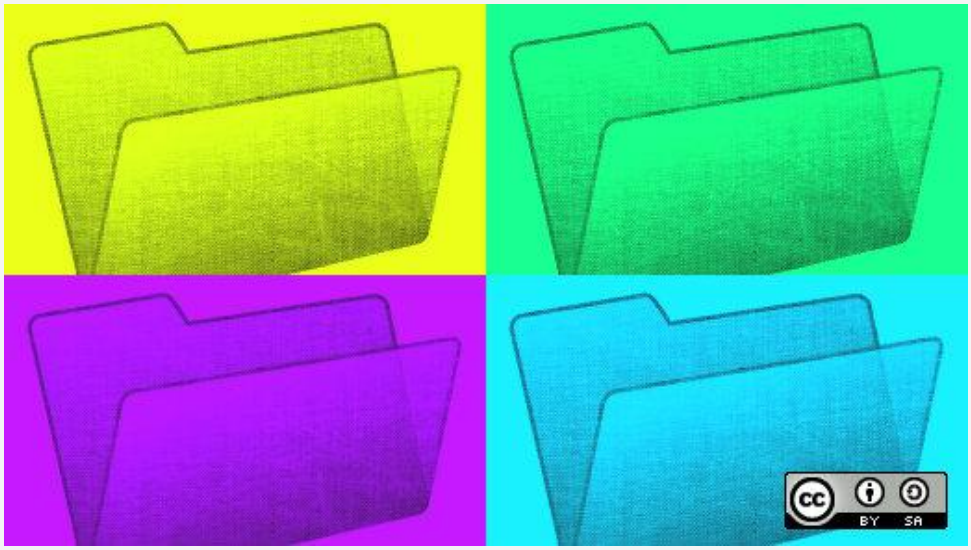

Image credits : *Open Clip Art Library (public domain). Modified by Jen Wike Huger.*

In Part 1, I laid out the structure for a multi-file C program called MeowMeow that implements a toy codec. I also talked about the Unix philosophy of program design, laying out a number of empty files to start with a good structure from the very beginning. Lastly, I touched on what a Makefile is and what it can do for you. This article picks up where the other one left off and now I'll get to the actual implementation of our silly (but instructional) MeowMeow codec.

The structure of the **main.c** file for **meow**/**unmeow** should be familiar to anyone who's read my article "How to write a good C main function." It has the following general outline:

```
/* main.c - MeowMeow, a stream encoder/decoder */

/* 00 system includes */
/* 01 project includes */
/* 02 externs */
/* 03 defines */
/* 04 typedefs */
/* 05 globals (but don't)*/
/* 06 ancillary function prototypes if any */

int main(int argc, char *argv[])
{
  /* 07 variable declarations */
  /* 08 check argv[0] to see how the program was invoked */
  /* 09 process the command line options from the user */
  /* 10 do the needful */
}

/* 11 ancillary functions if any */
```

Including project header files

The second section, **/* 01 project includes /***, reads like this from the source:

```
/* main.c - MeowMeow, a stream encoder/decoder */
...
/* 01 project includes */
#include "main.h"
#include "mmecode.h"
#include "mmdecode.h"
```

The **#include** directive is a C preprocessor command that causes the contents of the named file to be "included" at this point in the file. If the programmer uses double-quotes around the name of the header file, the compiler will look for that file in the current directory. If the file is enclosed in <>, it will look for the file in a set of predefined directories.

The file **main.h** contains the definitions and typedefs used in **main.c**. I like to collect these things here in case I want to use those definitions elsewhere in my program.

The files **mmencode.h** and **mmdecode.h** are nearly identical, so I'll break down **mmencode.h**.

```
/* mmencode.h - MeowMeow, a stream encoder/decoder */

#ifndef _MMENCODE_H
#define _MMENCODE_H

#include <stdio.h>

int mm_encode(FILE *src, FILE *dst);

#endif /* _MMENCODE_H */
```

The **#ifdef, #define, #endif** construction is collectively known as a "guard." This keeps the C compiler from including this file more than once per file. The compiler will complain if it finds multiple definitions/prototypes/declarations, so the guard is a *must-have* for header files.

Inside the guard, there are only two things: an **#include** directive and a function prototype declaration. I include **stdio.h** here to bring in the definition of **FILE** that is used in the function prototype. The function prototype can be included by other C files to establish that function in the file's namespace. You can think of each file as a separate *namespace*, which means variables and functions in one file are not usable by functions or variables in another file.

Writing header files is complex, and it is tough to manage in larger projects. Use guards.

MeowMeow encoding, finally

The meat and potatoes of this program—encoding and decoding bytes into/out of **MeowMeow** strings—is actually the easy part of this project. All of our activities until now have been putting the scaffolding in place to support calling this function: parsing the command line, determining which operation to use, and opening the files that we'll operate on. Here is the encoding loop:

```
/* mmencode.c - MeowMeow, a stream encoder/decoder */
...
    while (!feof(src)) {

      if (!fgets(buf, sizeof(buf), src))
        break;

      for(i=0; i<strlen(buf); i++) {
        lo = (buf[i] & 0x000f);
```

```
        hi = (buf[i] & 0x00f0) >> 4;
        fputs(tbl[hi], dst);
        fputs(tbl[lo], dst);
      }
          }
```

In plain English, this loop reads in a chunk of the file while there are chunks left to read (**feof(3)** and **fgets(3)**). Then it splits each byte in the chunk into **hi** and **lo** nibbles. Remember, a nibble is half of a byte, or 4 bits. The real magic here is realizing that 4 bits can encode 16 values. I use **hi** and **lo** as indices into a 16-string lookup table, **tbl**, that contains the **MeowMeow** strings that encode each nibble. Those strings are written to the destination **FILE** stream using **fputs(3)**, then we move on to the next byte in the buffer.

The table is initialized with a macro defined in **table.h** for no particular reason except to demonstrate including another project local header file, and I like initialization macros. We will go further into why a future article.

MeowMeow decoding

Alright, I'll admit it took me a couple of runs at this before I got it working. The decode loop is similar: read a buffer full of **MeowMeow** strings and reverse the encoding from strings to bytes.

```
/* mmdecode.c - MeowMeow, a stream decoder/decoder */
...
int mm_decode(FILE *src, FILE *dst)
{
  if (!src || !dst) {
    errno = EINVAL;
    return -1;
  }
  return stupid_decode(src, dst);
}
```

Not what you were expecting?

Here, I'm exposing the function **stupid_decode()** via the externally visible **mm_decode()** function. When I say "externally," I mean outside this file. Since **stupid_decode()** isn't in the header file, it isn't available to be called in other files.

Sometimes we do this when we want to publish a solid public interface, but we aren't quite done noodling around with functions to solve a problem. In my case, I've written an I/O-intensive function that reads 8 bytes at a time from the source stream to decode 1 byte to write to the destination stream. A better implementation would work on a buffer bigger than 8 bytes at a time. A *much* better implementation would also buffer the output bytes to reduce the number of single-byte writes to the destination stream.

```
/* mmdecode.c - MeowMeow, a stream decoder/decoder */
...
int stupid_decode(FILE *src, FILE *dst)
{
  char           buf[9];
  decoded_byte_t byte;
  int            i;

  while (!feof(src)) {
    if (!fgets(buf, sizeof(buf), src))
      break;
    byte.field.f0 = isupper(buf[0]);
    byte.field.f1 = isupper(buf[1]);
    byte.field.f2 = isupper(buf[2]);
    byte.field.f3 = isupper(buf[3]);
    byte.field.f4 = isupper(buf[4]);
    byte.field.f5 = isupper(buf[5]);
    byte.field.f6 = isupper(buf[6]);
    byte.field.f7 = isupper(buf[7]);

    fputc(byte.value, dst);
  }
  return 0;
}
```

Instead of using the bit-shifting technique I used in the encoder, I elected to create a custom data structure called **decoded_byte_t**.

```
/* mmdecode.c - MeowMeow, a stream decoder/decoder */
...

typedef struct {
  unsigned char f7:1;
  unsigned char f6:1;
  unsigned char f5:1;
  unsigned char f4:1;
  unsigned char f3:1;
  unsigned char f2:1;
  unsigned char f1:1;
```

```
  unsigned char f0:1;
} fields_t;

typedef union {
  fields_t      field;
  unsigned char value;
} decoded_byte_t;
```

It's a little complex when viewed all at once, but hang tight. The **decoded_byte_t** is defined as a **union** of a **fields_t** and an **unsigned char**. The named members of a union can be thought of as aliases for the same region of memory. In this case, **value** and **field** refer to the same 8-bit region of memory. Setting **field.f0** to 1 would also set the least significant bit in **value**.

While **unsigned char** shouldn't be a mystery, the **typedef** for **fields_t** might look a little unfamiliar. Modern C compilers allow programmers to specify "bit fields" in a **struct**. The field type needs to be an unsigned integral type, and the member identifier is followed by a colon and an integer that specifies the length of the bit field.

This data structure makes it simple to access each bit in the byte by field name and then access the assembled value via the **value** field of the union. We depend on the compiler to generate the correct bit-shifting instructions to access the fields, which can save you a lot of heartburn when you are debugging.

Lastly, **stupid_decode()** is *stupid* because it only reads 8 bytes at a time from the source **FILE** stream. Usually, we try to minimize the number of reads and writes to improve performance and reduce our cost of system calls. Remember that reading or writing a bigger chunk less often is much better than reading/writing a lot of smaller chunks more frequently.

The wrap-up

Writing a multi-file program in C requires a little more planning on behalf of the programmer than just a single **main.c**. But just a little effort up front can save a lot of time and headache when you refactor as you add functionality.

To recap, I like to have a lot of files with a few short functions in them. I like to expose a small subset of the functions in those files via header files. I like to keep my constants in header files, both numeric and string constants. I *love* Makefiles and use them instead of

Bash scripts to automate all sorts of things. I like my **main()** function to handle command-line argument parsing and act as a scaffold for the primary functionality of the program.

I know I've only touched the surface of what's going on in this simple program, and I'm excited to learn what things were helpful to you and which topics need better explanations. Share your thoughts in the comments to let me know.

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

A practical guide to learning awk

Optimize runtime performance with C++'s move semantics

Subscribe to our weekly newsletter

Enter your email address...

Select your country or region

Subscribe

Privacy Statement

Get the highlights in your inbox every week.

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

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a Creative Commons license but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

Copyright ©2020 Red Hat, Inc.