How To Write A Basic Makefile

Matthew Monsour

One of the most important tools at a programmer's disposal is a build system. These tools help simplify a programmer's workflow, making it easy to test changes to large complex programs with the press of a single button, or use of a single command. Despite this, many programmers can't even set up Make, one of the most popular build systems, to work with their projects. Using this guide, one will be able to set up a very basic Makefile to start with, which can then be customized to meet their own needs.

Prerequisites

- Some variant of Make must be installed. The guide assumes the use of GNU Make, although other variants will still work.
- A text editor must be available. In the guide, GNU nano is used, although any capable text editor will work.
- A basic shell must be installed, as well. A unix-like shell and environment is preferred, but any shell that can run the installed variant of Make is sufficient.
- A working C compiler must be installed. While Makefiles for other languages can be written, this guide will focus on writing one for a project written in the C programming language. (More advanced users can then expand this Makefile to be used along with other languages or projects)

1 A Sample Project

For the sake of simplicity, this guide will focus on writing a Makefile that works with a given sample project. The following two C files can be saved to the computer for later use. By the end of the guide, a Makefile that can build the C files will be constructed. **Make sure the two files are saved to the same directory the Makefile will be created in.**

```
// main.c
#include <stdio.h>

void say_hello(char*);

int main()
{
        char name[25];
        int n;
        printf("What is your name?\n");
        scanf("%24s", name);
        say_hello(name);

        return 0;
}
```

```
// hello.c
#include <stdio.h>

void say_hello(char *name)
{
        printf("Hello, %s!\n", name);
        printf("How are you today?\n");
}
```

2 Creating The Makefile

1. In the same directory as the two C files, create a new file named "Makefile". Open this file in any text editor.

   - To use GNU nano, enter `nano Makefile` into the shell

2. At the beginning of the file, add the following variables. These will be used by the Makefile to determine what to build, and how to build it. They contain information such as which compiler to use, what intermediate files are needed, and what extra flags to send the compiler. Add only one per line, and **do not add any extra spaces!**

   - `CC=gcc`
   - `SRCS=main.c hello.c`
   - `CFLAGS=-Wall`
   - `OBJS=$(subst .c,.o,$(SRCS))`

   Note: gcc can be substituted for a different C compiler, although it is recommended to be used in this guide.

   After this step, the Makefile should look similar to this:

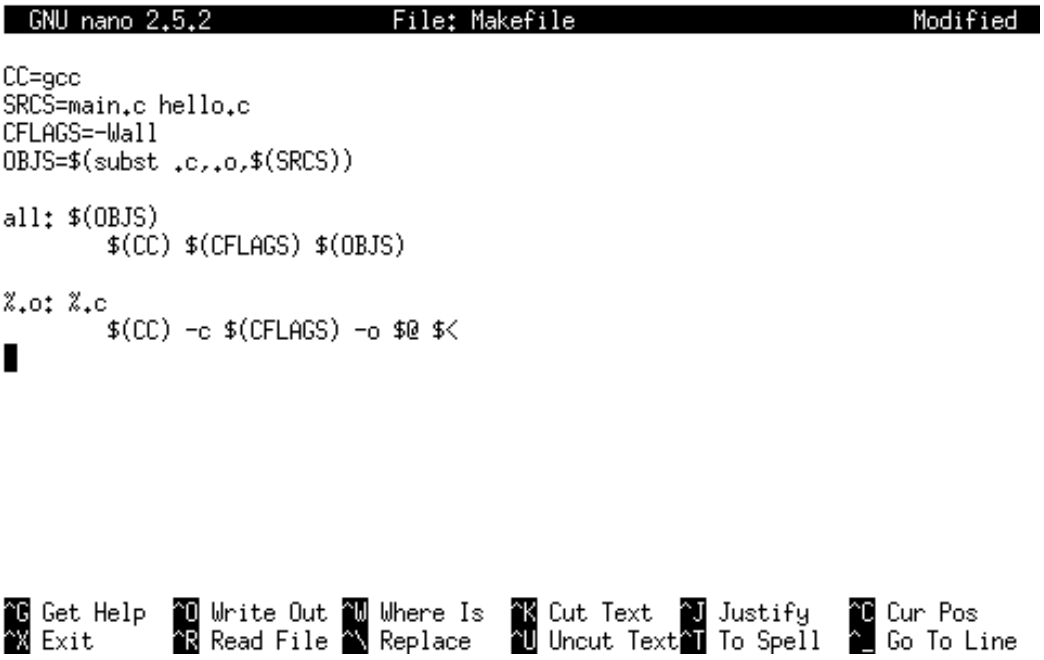
3. Next, add the folowing lines to the file. These lines create a rule for building the final program from the intermediate object files. Be sure to include the tab character in the second line.

   - `all:  $(OBJS)`
   - `	$(CC) $(CFLAGS) $(OBJS)`

4. Add some more lines to the Makefile, after the previous ones. These lines will tell Make how to create the object files from the given source files. Again, remember to include the tab character.

   - `%.o:  %.c`
   - `	$(CC) -c $(CFLAGS) -o $@ $<`

   At this point, the text file should look like this:



5. Finally, add just a few more lines to the end of the Makefile. These provide a way of cleaning up extra files, in order to perform a completely fresh build of the project.

   - `clean:`
   - `	rm -rf $(OBJS)`
   - `	rm -rf a.out`

The final Makefile should look similar to this:

```
  GNU nano 2.5.2                File: Makefile                          Modified

CC=gcc
SRCS=main.c hello.c
CFLAGS=-Wall
OBJS=$(subst .c,.o,$(SRCS))

all: $(OBJS)
        $(CC) $(CFLAGS) $(OBJS)

%.o: %.c
        $(CC) -c $(CFLAGS) -o $@ $<

clean:
        rm -rf $(OBJS)
        rm -rf a.out
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

3 Using the Makefile

1. Enter the shell, and navigate to the directory with the Makefile and source files.

```
sh-4.3$ ls
hello.c  main.c  Makefile
sh-4.3$
```

2. To build the project, run the command `make`

   This tells Make to use the `all` rule to build our project. In the previous section, all the information necessary for Make to build using this rule were inserted into the Makefile.

```
sh-4.3$ ls
hello.c  main.c  Makefile
sh-4.3$ make
gcc -c -Wall -o main.o main.c
gcc -c -Wall -o hello.o hello.c
main.c: In function 'main':
main.c:8:6: warning: unused variable 'n' [-Wunused-variable]
  int n;
      ^
gcc -Wall main.o hello.o
sh-4.3$
```

   Note the warning being displayed. This warning is not enabled by default in GCC, which indicates that the Makefile has enabled it. If the provided code was used, and the warning is displayed upon running Make, the Makefile was set up correctly!

3. To clean up the extra files, run the command `make clean`

   In the following picture, one can see the extra files created by GCC and the Makefile. This includes the executable program `a.out`. `make clean` will use the clean rule in the Makefile to delete these extra files, leaving us just with the original source files.

```
sh-4.3$ ls
a.out  hello.c  hello.o  main.c  main.o  Makefile
sh-4.3$ make clean
rm -rf main.o hello.o
rm -rf a.out
sh-4.3$ ls
hello.c  main.c  Makefile
sh-4.3$
```

Conclusions

Using this guide one should have successfully created and used a simple Makefile. While this is just a minimal example of how to use Make, it remains a good starting point. This guide can be combined with other online resoures to create an even more complex and useful Makefile, which can be used for a wide variety of projects.