Make demystified

1 Introduction

The *make* utility can greatly simplify the task of compiling a project with many source files. You may think of a *makefile* as a recipe that specifies what to do with your source files. The makefile may also contain a set of parameters and/or options that make will pass on to the preprocessor, compiler, linker, etc. The make utility is very general, and while is not restricted to dealing with C source code in any way, this short tutorial will focus on makefiles for C projects.

A makefile typically contains a number of *variables* and any number of so-called *targets*. We will start by looking at variables and return to targets later.

2 Variables

The default variable name for the compiler is `CC`. Preprocessor flags (i.e., options or parameters that will be passed to the preprocessor) when working with C code are specified using the `CPPFLAGS` variable. Similarly, the variable `CFLAGS` is for compiler flags, the variable `LDFLAGS` is for linker flags, and the variable `LDLIBS` is for library flags (e.g., a set of external libraries to be used when linking your program). Here is a quick overview of some of the things that these variables can be used for:

- `CC`: this variable specifies the C compiler (e.g., `CC=gcc` or `CC=clang`)
- `CPPFLAGS`: the preprocessor flags may include macro definitions (e.g., the flag `-DN=5` is equivalent to adding `#define N 5` to your source file(s)) and paths to some header files (e.g., the flag `-Iinclude` instructs the preprocessor to search for header files in the subdirectory `include`).
- `CFLAGS`: the compiler flags may be used to enable warnings (e.g., the flag `-Wall` enables a set of warnings), specify a specific language standard (e.g., the flag `-std=c99` instructs the compiler to use the C99 standard), and enable compiler optimization (e.g., the flag `-O2` enables “level 2” code optimization).
- `LDFLAGS`: the linker flags may include a path to an external library that is otherwise not found by the linker (e.g., the flag `-Llib` instructs the linker to look for libraries in a subdirectory `lib`).
- `LDLIBS`: this variable should include a flag for each external library that the linker needs to complete the linking step. For example, if you use a function defined in the `math.h` header file, you should link against the “math library” by adding the flag `-lm`.

Before we construct a basic makefile, let us take a look at how we can use the `make` program without a so-called makefile. We will use the “Hello World!” program as an example:

```c
#include <stdio.h>
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Assuming that the source file is named `hello.c`, you can compile this program with make using the command `make hello`. You can also specify/override the standard variables mentioned above when invoking make. For example, to enable warnings and to specify the compiler and C language standard, we can invoke make as follows:

```
$ make CC=gcc CFLAGS="-Wall -std=c99" hello
```

An alternative to specifying variables on the command line when calling make, we can construct a basic makefile with our preferred default values:

```makefile
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c99
LDFLAGS=
LDLIBS=
```

The default name for a makefile is `Makefile` without an extension, and if you use this name, you do not have the specify the name of the makefile when you invoke make. If you want your makefile to have a different name or if you need more than one makefile in a directory, you can let `make` know which makefile to use by invoking make as `make -f MyMakefile` (i.e., assuming your makefile is named `MyMakefile`).

So far our makefile only serves to enable warnings (`-Wall`) and to specify the C compiler (`gcc`) and the language standard (`-std=c99`). However, this makefile is sufficient for *simple* projects such as a project that consists of a *single* source file that includes a `main()` function but no external dependencies (except for standard header files).

3 Projects with multiple source files

Now let us consider a simple program that consists of three source files, say, `greetings.c`, `hello.c`, and `goodbye.c` along with two header files `hello.h` and `goodbye.h`:

```c
/* greetings.c */
#include "hello.h"
#include "goodbye.h"
int main(void) {
    hello();
    goodbye();
    return 0;
}
```

```c
/* hello.c */
#include <stdio.h>
void hello(void) {
    printf("Hello!\n");
}
```

```c
/* goodbye.c */
#include <stdio.h>
void goodbye(void) {
    printf("Goodbye!\n");
}
```

```c
/* hello.h */
#ifndef HELLO_H
#define HELLO_H
void hello(void);
#endif
```

```c
/* goodbye.h */
#ifndef GOODBYE_H
#define GOODBYE_H
void goodbye(void);
#endif
```

Each source files with a `.c` extension is a separate *compilation unit* which means that it can be compiled separately into an object file. Our basic makefile can be used for this purpose (in fact, you do not even need a makefile to do this):

```
$ make greetings.o
$ make hello.o
$ make goodbye.o
```

However, if you try to create a `greetings` executable using `make greetings`, the linker will generate an error message that says something about *undefined symbols*. The problem is that `make` only passes the object code from the greetings compilation unit to the linker, and as a result, the linker is unable to find all the pieces needed to construct the executable. You can link manually by specifying all the necessary components:

```
$ gcc greetings.o hello.o goodbye.o -o greetings
```

4 Targets

An alternative to the manual approach described in the previous section is to add some information to your makefile. As mentioned in the beginning, a makefile may also include a number of so-called *targets*. A target may have some dependencies which can be any number of source files and/or other targets. A target is specified as `target: dependencies`. For example, we may specify that the greetings executable depends on `hello.o` and `goodbye.o` by augmenting our basic makefile as follows:

```
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c99
LDFLAGS=
LDLIBS=
greetings: hello.o goodbye.o
```

The last line specifies that the `greetings` target depends on `hello.o` and `goodbye.o`. We can now build the `greetings` executable simply by executing `make greetings` or simply `make` (if `make` is invoked without a target, it will pick the first target that does not start with a `.`).

A target may also include a rule that specifies *how* to create a target from its dependencies. The previous makefile is roughly equivalent to the following, more explicit makefile:

```
hello.o: hello.c
    gcc -c -Wall -std=c99 -o hello.o
goodbye.o: goodbye.c
    gcc -c -Wall -std=c99 -o goodbye.o
greetings.o: greetings.c
    gcc -c -Wall -std=c99 -o greetings.o
greetings: greetings.o hello.o goodbye.o
    gcc greetings.o hello.o goodbye.o -o greetings
```

These rules explicitly specify how to create the greetings executable from the object files and how to create the object files from the source files. However, it is much easier to rely on make's *builtin* or *implicit* rules, especially if you are dealing with many files or want to experiment with different compiler options such as code optimization. Note that the make program will complain if the rules are not correctly indented: make requires a *tab* for indentation and does not accept *spaces* for indentation.

5 Phony targets

A makefile may also have so-called *phony* targets. These are targets that do not result in the creation of a file that has the target's name. The `greetings` target is not a phony target, because it results in a file named `greetings` when the make program is done processing the target. It is customary to add a phony target with the name *clean* which may be used to "clean up", i.e., by deleting object files and/or targets. A `clean` target may look like this:

```
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c99
LDFLAGS=
LDLIBS=
greetings: hello.o goodbye.o

.PHONY: clean
clean:
    rm -f greetings *.o
```

The command `make clean` will then execute the shell command `rm -f *.o greetings` which deletes/removes the `greetings` file and all files in the current directory with the `.o` extension. The `rm -f` command is Linux/Unix specific, so a more portable approach is to make use of make's `RM` variable which specifies a command that may be used for deleting files (just like `CC` specifies the C compiler). Moreover, to avoid an error message if the `greetings` executable does not exist, we may prepend a `-` to the `clean` rule. With these modifications, the makefile now looks like this:

```
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c99
LDFLAGS=
LDLIBS=
greetings: hello.o goodbye.o

.PHONY: clean
clean:
    -$(RM) greetings *.o
```

Another common phony target name is *all*. This is often used as shorthand for several targets. For example, suppose we write two test programs `test_hello.c` and `test_goodbye.c`:

```c
/* test_hello.c */
#include "hello.h"
int main(void) {
    hello();
    return 0;
}
```

```c
/* test_goodbye.c */
#include "goodbye.h"
int main(void) {
    goodbye();
    return 0;
}
```

Now let us add these targets to our makefile along with a phony *all* target:

```
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c99
LDFLAGS=
LDLIBS=
.PHONY: all
all: greetings test_hello test_goodbye

greetings: hello.o goodbye.o
test_hello: hello.o
test_goodbye: goodbye.o

.PHONY: clean
clean:
    -$(RM) greetings test_hello test_goodbye *.o
```

The phony `all` target has three dependencies: `greetings` and the two new targets `test_hello` and `test_goodbye`. Since the `all` target is the first target in the makefile, simply executing `make` will be equivalent to `make all` which causes `make` to process the three targets `greetings`, `test_hello`, and `test_goodbye` that are listed as dependencies. Finally, notice that the new targets have been added to the `clean` rule so that `make clean` will also delete the new targets if they exist.