

Lab A

28 May 2009

Today's lab has two parts, both hopefully short; the first describes “makefiles”, a mechanism that lets you avoid retyping the compiler commands over and over again, and the second describes some more features of the C preprocessor.

1 Makefiles

The command `make` is designed as a high-level command which, given the graph of what depends on what in a particular project, executes all the commands necessary to make a final product. If one header file is changed, for instance, only those files that include it will be recompiled. The dependencies are specified in a file named `Makefile`, whose complete format specification is deeply arcane, but we'll see a few of the highlights.

The starting point for this lab is the end of the previous one; it assumes you have at least the files `sms.h`, `sms.c`, `smssend.c`, and `smsrecv.c` in place.

Targets

The first concept is the idea of a *target*: these are the files that `make` is creating, such as the actual executable file. Create a file named `Makefile` (note the capital M), and type the following lines:

```
smssend: smssend.c sms.c sms.h
        gcc -std=c99 -Wall -o smssend smssend.c sms.c
```

NB: the second of these lines starts with a single tab character. Do not use spaces! This is a notoriously easy mistake, and `make` is not very forgiving.

There are three pieces to this.

- The token before the colon, here `smssend`, is the name of the target. It should be the precise filename of a file that will be created by the command.

- The items after the colon on the first line are the dependencies. These are any files which, if modified, indicate that the target needs to be remade. Note, crucially, that this list *does* include the header file.
- The second line (and subsequent lines, if any, up to a blank line or another target) will then give the exact commands that need to be executed in order to create the target file. The header file is not included here because it wouldn't be included if you typed `cc` manually.

Once you have created this file, you should be able to type

```
make smssend
```

and the command will be executed. (If it says nothing needs to be done, `rm` the file `smssend` and then try it again.)

Add a target for `smsrecv` as well, then move on to the next section.

Variables

One nice feature of `make` is that you can extract repeated pieces of commands into separate variables, much as you would in a programming language. Here, we have the compiler flags, which are by convention usually stored in a variable `CFLAGS`. At the top of the file, add the line

```
CFLAGS = -std=c99 -Wall
```

and then modify the `smssend` command to be

```
gcc $(CFLAGS) -o smssend smssend.c sms.c
```

and similarly for `smsrecv`.

Intermediate files

So far we haven't been using the real power of `make`; anytime anything changes, we recompile all the source files that go into a particular executable. But in fact, if we make use of separate compilation, as seen in last week's lab, we only need to recompile the source files that change.

First, add a target for one of the `.o` files:

```
smssend.o: smssend.c sms.h
gcc -c smssend.c
```

Note that this target depends on both its corresponding source file and the header included therein, because if the header file changes, the source file will need to be recompiled. **C and make do not automatically know this.** You must explicitly specify this or your compiled files can get out of sync.

Once you have added this target and corresponding ones for `smsrecv.o` and `sms.o`, you can modify the executable targets:

```
smssend: smssend.o sms.o
gcc -o smssend smssend.o sms.o
```

This says: `smssend` depends on `smssend.o` and `sms.o`. Go check on them and see if they exist and are up to date, and if not, **make** them first. Then, if either of them got updated, this target needs to be updated as well.

Note that the command here removes the `CFLAGS`—this is because the files being passed to the compiler are all `.o` files, so `gcc` is only acting as a linker in this case; the compiler flags would be ignored anyway.

These are but the very basics of **make**; your path to true **make-fu** will be a long one. In the meantime, take heart in the knowledge that almost nobody writes a makefile from scratch, ever; they just copy old ones and modify them for the new project.

2 Preprocessor directives

The preprocessor, you might recall, is the first step of a C compilation; anything that begins with a pound sign (`#`) is a preprocessor directive. The main one you've seen so far is `#include`, which looks up a file and physically includes it in what is passed to the actual compiler.

Another important preprocessor directive is `#define`. This mechanism has been historically used for three purposes: defining constants (which is theoretically subsumed by the `const` keyword but still widely used); defining something that C calls macros (which aren't true macros, though that's rant for a different day, and which are theoretically subsumed by `inline`

functions, or will be if the standard compilers ever get around to correctly implementing `inline` functions); and controlling conditional compilation.

You can define a constant as follows:

```
#define PI 3.14159265
```

Everywhere you use `PI` as an identifier, the preprocessor will substitute `3.14159265`, and the compiler will use that value. This is a textual substitution only, so BE CAREFUL with what you use as the ‘value’ of a `#defined` constant: if you write

```
#define PI 3.14159265 // approximately
```

then the line

```
if (x < PI / 2)
```

gets rewritten as

```
if (x < 3.14159265 // approximately / 2)
```

and that won’t compile.

You can test whether a particular identifier has been `#defined` using `#ifdef` and `#ifndef`; one use of this is to only `#define` a constant if it doesn’t already exist (so as not to overwrite a more precise definition, for instance)

```
#ifndef PI
#define PI 3.14159265
#endif
```

You can also use `#else` to specify both branches.

The body of the `#ifdef` or `#ifndef` can contain regular C code, as well, which forms the basis of a useful trick.

Conditional compilation

First, to motivate why you’d ever want to do this, go to `smssend.c` and duplicate the line that `#includes sms.h`, then `make smssend`. This should give you some compiler errors having to do with redefining the `struct`.

To fix this problem, add three lines to `sms.h`. At the top, add

```
#ifndef _SMS_H_
#define _SMS_H_
```

and at the bottom, add

```
#endif
```

What does this do? The first time through, the funny-looking identifier `_SMS_H_` isn't defined, so the rest of the file is included—including a line that defined `_SMS_H_` (to be replaced with blank whitespace, but it's still a definition). The second time the preprocessor `#includes` this file, that identifier *is* defined, so the remainder of the file, up through the `#endif`, is *not* included.

I strongly suggest you put lines analogous to these around every C or C++ header file you ever write. It is an extraordinarily useful habit to develop, and saves you buckets of trouble later.

There's one last preprocessor trick to control compilation: commenting out big blocks of code. This can be hard to do with `/*` and `*/` because you can't nest them; instead, you can prefix a block of code with

```
#if 0
```

and follow it with

```
#endif
```

and it won't be seen by the compiler. In fact, most syntax-highlighting editors will treat this just the same as the other kinds of comments, and grey it or otherwise make it fade out.

3 And now...

Hopefully that didn't take the whole lab period. Work on your project! Ask questions!