

Lab 9

21 May 2009

In today's lab, we'll return to working with C language features; in particular, the notion of separate compilation. You've already seen separate compilation to in Java, where individual `.java` files are each compiled into their own `.class` files, but there are a few more nuances in C.

As a starting point, we'll use a slightly modified version of my code from last week's lab. Copy `smssend.c` and `smsrecv.c` into your directory (don't overwrite your own, though!). These two files each stand alone currently, but they contain a certain amount of duplicated code. You'll fix that.

1 Basic `.h` file usage

The simplest way to distribute the load is with header files. Create a file `sms.h` and put the `struct` definition there; then remove the `struct` definitions from the other two files and replace them with

```
#include "sms.h"
```

The role of a `#include` is to act precisely as if the full contents of the referenced file were literally included at that spot in this file. Angle brackets are used for system header files, double quotes for local header files.

Confirm that both files compile and work with the `#included` file before proceeding.

2 Multiple `.c` files

Much like in Java, you can separate the code itself into multiple files, functions and all. Unlike Java, the C compiler will never look at any file other than the one(s) you refer to on the command line. If you want to compile a whole executable program, you need to `cc` *all* of the relevant `.c` files. If you try to `cc` a single file that is not a fully self-contained program, it will complain.

There is, however, a way to compile the files separately. By default, `cc` is actually running three different programs: the preprocessor (which handles

the `#includes`, among other things), the compiler proper, and the linker. The compiler proper is what does the actual translation to machine code, but it can't create an executable—that's the linker's job. But if you halt the process at the compiler step, you can compile an individual file that is only *part* of a complete program. This is done with the `-c` option to `cc`, creating a `.o` file corresponding to the `.c` file. Later on, if you call `cc` with several `.o` files, it will see that they are already compiled and just pass them along to the linker to create the executable.

Separate out the `read_msg` and `print_msg` functions into their own file `sms.c`, and put their prototypes into `sms.h`. Note that `sms.c` should itself also include `sms.h`. Verify that

```
cc -c sms.c
```

successfully compiles that file, and then that

```
cc sms.o smssend.c -o smssend
```

successfully compiles `smssend.c` and links it to the already-compiled `sms.o`. Do similarly for `smsrecv`.

3 Functions and `.h`

The `failif` function is present in both files is not really specific to SMS, but it is duplicated, so we might want to put it into the shared portion of the code. You might be wondering why we don't just put shared pieces right in the header; it has to do with the way compilation proceeds.

First, a demonstration. Move the definition of `failif` into the header file `sms.h`. Compile the individual files separately (with `cc -c`, and everything should be fine. Then link them, and you should see a problem. (Go ahead, do it.)

The issue here is that *both* `.o` files have a compiled unit named `failif`, and so the linker doesn't know which one is the 'right' one (or that they happen to be functionally identical). The fix is to make sure that function definitions *always* go only in the `.c` files. Header files can include function prototypes, structure definitions, and further `#include`ing of system headers. These user-defined headers can then be freely `#included` within multiple `.c` files.