Crash Course: Compiling and Linking Multiple C Files + Makefiles

---

DISCLAIMER: most of discussion below is just informational -- you ought to know a little about makefiles because they can make your life easier. But don't expect any exam questions on makefiles!

The very short version of how to use the UNIX make utility for this assignment is:

- download the file called "`makefile`" along with the other source files (`list.h, llist.c, ll_tst.c`)
- Modify `llist.c` as specified in the assignment.
- Modify (incrementally) `ll_tst.c` as your tester program.
- To recompile, just type:

```
make ll_tst
```

at the command prompt.

In this assignment, a list Abstract Data Type (ADT) is captured by two files: `list.h` and `llist.c`. These files operate in a way similar to the stack ADT example we have studied in class.

`list.h`	A header file which specifies a List Abstract Data Type (ADT). It makes a type `LIST` usable by programs that want to use the List ADT.
	
	Specifically, it contains these lines of code:
	
	`// hidden implementation of list_struct`
	`typedef struct list_struct LIST;`
	
	Notice that the actual fields of a "`list_struct`" are not given. This is called an "incompletely specified type" in C.
	
	The fields of a list_struct are "hidden" in the `llist.c` file described below.
	
	A "client" program that wants to use this list ADT only knows that there is a type called LIST, but not the details of the structure. The compiler is ok with this so long as the client program only declares variables that are LIST *pointers* and does not de-reference these pointers.
	It also gives the "prototypes" of the functions operating on lists.

`llist.c`	This file contains an actual *implementation* of the functions specified in the header file. It also specifies the detailed contents of the LIST data type (i.e., the contents of a C struct). This organization hides the actual implementation of the LIST type from "client" programs.

	Client programs can only operate with LIST *pointers*. They cannot de-reference these pointers. They can only use the available functions to perform operations on lists.

You will also notice a file called `ll_tst.c`. This is a "client" program of the list ADT. It is a simple driver program which uses the list functions to create and modify lists.

Manual Compilation

Let's review how to compile these files by hand.

First, we want to compile `llist.c`. Remember that list.c is not a "program" in that it does not contain a `main` function. Instead, you can think of it as a library of functions that might be useful to any number of programs. Since it is not a "program", we will compile it into an "object file" as follows:

```
gcc -c llist.c
```

This will produce an object file called `llist.o`.

Now, to compile our toy driver program (which calls functions implemented in `llist.c`) we do this:

```
gcc ll_tst.c llist.o -o ll_tst
```

This tells gcc to compile the program `ll_tst.c` and to "link" it with the object file `llist.o`; finally is says to save the executable in the file `ll_tst` (instead of `a.out` for example).

Makefiles

The compilation steps above can be tedious to remember. In addition, there are some dependencies between the steps -- for example, suppose you decide to add a new test case to `ll_tst.c` but you are not modifying `llist.c` (at least not at the moment). In this situation, `llist.o` is “up to date” and doesn’t need to be recompiled, so you can just perform the second step above.

On the other hand, if I modify llist.c, I will need to perform *both* steps -- even if I haven’t modified `ll_tst.c`

You can imagine that if your project had more than just a few files (and maybe multiple programmers), it would be hard to remember exactly what to do to bring everything up to date after a modification to some file -- plus, it would be a waste of brain space!

A `makefile` provides a systematic way to capture these kind dependencies and "do the right thing" based on the dependencies and time stamps on files.

You have been given such a `makefile` which works for this small 3-file project we have.

To use a `makefile`, we run the standard UNIX utility program `make` (seems like a good name!). So in the directory with the source files and the makefile, you can just type "make" from the command line. Below is the result of me running make (`"lillis$"` is just the prompt).

```
lillis$ make ll_tst
gcc  -Wall -c llist.c
gcc  -Wall ll_tst.c llist.o -o ll_tst
```

You can see that make is invoking `gcc` itself (I have added the `-Wall` flag to turn on all warnings) Here is what the makefile itself looks like:

```
# these two lines specify make "variables"
# makes it easy to change to a different compiler and
# change the flags.  Notice the variables are "expanded"
# with a $ sign.
CC=gcc
FLAGS=-Wall

ll_tst: llist.o ll_tst.c
	$(CC) $(FLAGS) ll_tst.c llist.o -o ll_tst

llist.o: list.h llist.c
	$(CC) $(FLAGS) -c llist.c

# the clean target lets us "start from scratch"
clean:
	rm -f *.o ll_tst
```

In a nutshell, there are two types of lines in the makefile. The line

```
ll_tst:  ll_tst.c llist.o
```

specifies a "target" called `ll_tst` which *depends* on files `ll_tst.c` and `llist.o`. The next line is an "action":

```
	gcc ll_tst.c llist.o -o ll_tst
```

This line specifies what to do to create the target when all of its dependencies are up to date. **IMPORTANT DETAIL**: the action statement must have a **tab** before it! Regular spaces won't work.

Let's take a look again at what happened when I ran make:

```
lillis$ make ll_tst
gcc  -Wall -c llist.c
gcc  -Wall ll_tst.c llist.o -o ll_tst
```

Here is what make did:

> It first found the specified target in the makefile and then examines the dependencies (`ll_tst.c` and `llist.o`). Since `llist.o` is also a target in the makefile, it applies the same rules to it to make sure it is up to date/bring it up to date -- i.e., it recursively "makes" target `llist.o`).
>
> It discovers that `llist.o` needs to be compiled (e.g., because the source file `llist.c` is newer than `llist.o`) and issues the `gcc` command.
>
> Now that `llist.o` is up to date, it returns to working on `ll_tst`: it issues the `gcc` command to create `ll_tst`.

Summary

Makefiles and the make program are useful in several ways:

- For our particular project, we don't have to manually issue all of the separate compilation commands. We figure them out once and put them in the makefile. When you have more than a few files this saves you a lot of trouble remembering how files relate to each other, etc.
- They also let us capture the dependencies between files and targets. As a result, when we make modifications to our code and recompile, only the necessary files are recompiled.
- We can distribute our code with a makefile and others can compile it with a single command.