Managing Modularity: Makefiles and Libraries

Norman Matloff

Department of Computer Science
University of California at Davis
(530) 752-1953
matloff@cs.ucdavis.edu

July 4, 2005

Contents

1 Technical Overview

1.1 Managing Size and Complexity

In commercial software engineering environments, a program is typically large and complex. In order to better manage that size and complexity, good software practice is, as you know, to use modular, top-down design, with a large number of function calls. There typically are dozens, even hundreds of functions.

During the debugging of such a program, we will make a change to the source code, then compile and run, then make another change to the source code, then compile and run again, and so on. However, this is wasteful: We usually will change only a small part of the program each time, but we must recompile the entire program, including the parts which were not changed and thus should not have to be recompiled.

With this problem in mind, we typically will split the source code into a large number of files, rather than just one file. We may, for example, have a separate file for each significant function, and maybe one more file for the small, miscellaneous functions. Each file will have a name whose suffix is .c. We will compile each of these files separately, yielding machine-language files with .o suffices. Finally, we will use the linker, ld (which is a separate program from the compiler, but is called by the compiler), to link all the .o files into one executable file named a.out or some other name which we choose ourselves. When we subsequently make a change to our program, if that change affects only one of the .c files, we only recompile that file, forming a revised .o file. We then link again.

This saves a large amount of time! A large program might take, say, 10 or 15 minutes to compile in its entirety, whereas with the approach described here we might only recompile a small portion of the program. The time saved-and far more important, the minimizing of the interruption to the programmer's train of thought during the debugging process-can increase productivity tremendously.

1.2 "Did I Remember to Recompile That File or Not?"

In the frenzy of the debugging process, it is easy to get confused. One might make a couple of changes to some of the source files, but then when recompiling them lose track of which files one has recompiled and which still remain to be recompiled. To simplify life, the Unix **make** program is used.[1]

The **make** program will automatically decide whether a file needs to be recompiled or not. The way it does this is quite simple: If the last date/time of modification for the .c file is later than that of the corresponding .o file, the latter is obviously out of date, and thus the former must be recompiled. Whenever the programmer makes a change (or several changes), he/she simply types `make', and the required recompiling and relinking will be done automatically.

Also, if we have a set of modules which will be used by many programs, we create a *library* out of them. Then in compiling any program which uses that set of modules, we simply include the library as if it were a .o file.

2 Example

Here I have taken our program WC.c and split it into a number of files, and also set up a file for **make** to use.

2.1 Overview of the Files

Here are the contents of my directory:

```
Defs.h          Main.c          ReadLine.c      WordCount.c
ExternVars.h    PrintLine.c     UpdateCounts.c
```

I have placed each function in its own file, e.g. I've put the function ReadLine() in the file ReadLine.c. Main.c contains main(), and also the declarations of the global variables. For each of the latter, I have put an **extern** entry in the file ExternVars.h; accordingly, in each of the .c files except Main.c, I have put in a line

```
#include "ExternVars.h"
```

I've also made a file Defs.h for any #define lines which I need. The file Makefile contains the instructions for **make**.

2.2 The Files' Contents

Here are the listings of the various files:

ExternVars.h

```
extern char Line[MaxLine];
extern int NChars,NWords,NLines,LineLength;
```

Defs.h

```
#define MaxLine 200
```

Main.c

```
/* introductory C program

   implements (a subset of) the Unix wc command  --  reports character,
   word and line counts; in this version, the "file" is read from the
   standard input, since we have not covered C file manipulation yet,
   so that we read a real file can be read by using the Unix `<'
   redirection feature */


#include "Defs.h"


extern int ReadLine(),WordCount();


char Line[MaxLine];  /* one line from the file */


int NChars = 0,  /* number of characters seen so far */
    NWords = 0,  /* number of words seen so far */
    NLines = 0,  /* number of lines seen so far */
    LineLength;  /* length of the current line */


main()

{  while (1)  {
      LineLength = ReadLine();
      if (LineLength == 0) break;
      UpdateCounts();
   }
   printf("%d %d %d\n",NChars,NWords,NLines);
}
```

PrintLine.c

```
#include "Defs.h"
#include "ExternVars.h"


PrintLine()  /* for debugging purposes only */

{  int I;

   for (I = 0; I < LineLength; I++) printf("%c",Line[I]);
   printf("\n");
}
```

ReadLine.c

```
#include "Defs.h"
#include "ExternVars.h"


int ReadLine()

/* reads one line of the file, returning also the number of characters
   read (including the end-of-line character); that number will be 0
   if the end of the file was reached */

{  char C;  int I;

   if (scanf("%c",&C) == -1) return 0;
   Line[0] = C;
   if (C == '\n') return 1;
   for (I = 1; ; I++) {
      scanf("%c",&C);
      Line[I] = C;
      if (C == '\n') return I+1;
   }
}
```

WordCount.c

```
#include "Defs.h"
#include "ExternVars.h"


int WordCount()

/* counts the number of words in the current line, which will be taken
   to be the number of blanks in the line, plus 1 (except in the case
   in which the line is empty, i.e. consists only of the end-of-line
   character); this definition is not completely general, and will be
   refined in another version of this function later on */

{  int I,NBlanks = 0;

   for (I = 0; I < LineLength; I++)
   if (Line[I] == ' ') NBlanks++;

   if (LineLength > 1) return NBlanks+1;
   else return 0;
}
```

UpdateCounts.c

```
#include "Defs.h"
#include "ExternVars.h"


extern int WordCount();


UpdateCounts()

{  NChars += LineLength;
```

```
   NWords += WordCount();
   NLines++;
}
```

Makefile

```
  3     WC: Main.o ReadLine.o WordCount.o UpdateCounts.o PrintLine.o
  4             cc -g -o WC Main.o ReadLine.o WordCount.o UpdateCounts.o \
                PrintLine.o
  5
  6     Main.o: Main.c Defs.h ExternVars.h
  7             cc -g -c Main.c
  8
  9     ReadLine.o: ReadLine.c Defs.h ExternVars.h
 10             cc -g -c ReadLine.c
 11
 12     WordCount.o: WordCount.c Defs.h ExternVars.h
 13             cc -g -c WordCount.c
 14
 15     UpdateCounts.o: UpdateCounts.c Defs.h ExternVars.h
 16             cc -g -c UpdateCounts.c
 17
 18     PrintLine.o: PrintLine.c Defs.h ExternVars.h
 19             cc -g -c PrintLine.c
```

(Note that I have added line numbers only to Makefile, but even then, keep in mind that that file does not actually contain these numbers.)

2.3 Use of Header Files

All the .c files above have a line

```
#include "Defs.h"
```

What an **include** statement does is tell the compiler to make a copy of the indicated file and compile it together with the file containing the **include**. For example, when the file PrintLine.c is being compiled, the compiler will see the **include**, and simply pretend that the entire contents of Defs.h had been in PrintLine.c.

All the .c files except for Main.c also have a line

```
#include "ExternVars.h"
```

Again, when the compiler is compiling, say, WordCount.c, it will simply act as if all the lines in ExternVars.h were in WordCount.c. This is a convenient way to get all the **extern** statements we need into WordCount.c, ReadLine.c and so on. It's just a convenience to avoid typing, but it also helps clear our minds-we don't have to remember to put in all the **extern** lines in each of these files.

Note that it wouldn't make sense to have the line

```
#include "ExternVars.h"
```

in Main.c, since the global variables are being declared there, thus making **extern** statements for these variables incorrect.

Note too that I have put in **extern**'s for the functions; e.g. I have the line

```
extern int ReadLine(),WordCount();
```

in Main.c. I don't really need this, since C assumes a function's return value type is **int** by default, but I would need it for non-**int** types, and thus it is a good habit to have such a statement even though the two functions concerned here are of **int** type.

The .h files should be for variable declarations, "define's" and other include's. They should not include any program statements (other than macros).

2.4 Details of Using `make'

Now let's take a look at the file Makefile. It consists of a series of **dependency/build** entries; Lines 3-4 comprise one such entry, Lines 6-7 form another, etc.[2] Each such entry tells first, what other files a given file depends on, and second, how to rebuild the given file if one of the ones it depends on changes.

For instance, look at Lines 3-4. Here Line 3 is a **dependency line**, while Line 4 is a **command line**. The file WC, which is our executable file for the program, is called a **target file**, meaning a file which we hope to build; Lines 3-4 provide information

(a) as to whether the current version of the target is outdated (Line 3), and
(b) if the target is outdated, how to rebuild it (Line 4).

Specifically, Line 3 says that WC depends on a number of .o files, i.e. Main.o, ReadLine.o, WordCount.o, UpdateCounts.o and PrintLine.o. So, if **make** is told to produce the file WC, it will check whether the current version of WC is at least as new as these .o files; if even one of these files is newer than WC, **make** will see that WC is out of date, and will rebuild WC, using the prescription in the command line, Line 4.

A couple of things about the latter should be noted. First, we are using the -o option of **cc**, which states that we don't want the default name of a.out for our executable file; we want the executable file to be named WC. Second, note that there are no .c files in this command at all, so there is no compiling to be done; why, then, are we using **cc**?

The answer to the last question is that **cc** automatically calls **ld**, the Unix linker command; this is always done, but you have not had a chance to be aware of it before now. In this case, we need all those .o files to be linked together into one executable file (WC), and calling **cc** will result in **ld** being called to do the linking.[3]

In general, a makefile consists of sets of lines of the form we see in Lines 3-4, i.e. in the form

```
target: dependency components
TAB shell command
TAB shell command
TAB shell command
...
```

(where TAB means the tab key, not the string `TAB'). Note that for instance Line 4 is a shell command.[4]

Let's see **make** in action:

```
 1 heather% ls
 2 Defs.h          Main.c          PrintLine.c     UpdateCounts.c  typescript
 3 ExternVars.h    Makefile        ReadLine.c      WordCount.c
 4 heather% make
 5 cc -g -c Main.c
 6 cc -g -c ReadLine.c
 7 cc -g -c WordCount.c
 8 cc -g -c UpdateCounts.c
 9 cc -g -c PrintLine.c
10 cc -g -o WC Main.o ReadLine.o WordCount.o UpdateCounts.o PrintLine.o
11 heather% ex WordCount.c
12 :1
13 
14     :i
15     #define ZERO 0
16     .
17     :wq
18     Wrote "WordCount.c"  26 lines, 582 characters
19     heather% make
20     cc -g -c WordCount.c
21     cc -g -o WC Main.o ReadLine.o WordCount.o UpdateCounts.o PrintLine.o
```

At first, we see that there are no .o files at all (Lines 2-3). When we type "make" (Line 4), **make** will read the file Makefile and try to build the first item it encounters (if it is not already up-to-date), which is WC, which depends on all the .o files (Line 3 of the Makefile listing). Since none of those files exist, **make** will have to generate them, and to do so, it looks further down in the file Makefile. For example, it sees on Line 6 of the Makefile that it can build Main.o by doing "cc -g -c Main.c" (the -c means compile only, i.e. just produce a .o file, and don't call **ld**), and it goes ahead and does so (Line 5 of the script file). After generating all the .o files, it then links them (Line 10 of the script file). Now all the .o files have been generated, as has WC.

Now let's update a file, say WordCount.c, and see what happens. Lines 11-18 of the script file show me making a change to this file, by adding a **#define** line (I used the **ex** editor so that it could be seen easily within the script file).

Now let's see how **make** will handle this change (Lines 19ff). What happened is that **make** found that WC depends on WordCount.o (Line 3 of Makefile), which in turn depends on WordCount.c (Line 12 of Makefile), and **make** noticed that the timestamp on WordCount.c was later than that of WordCount.o, i.e. the latter was out-of-date. So, **make** first recompiled WordCount.c, to produce an up-to-date WordCount.o, and then linked all the .o files together to produce an up-to-date file WC.

By the way, we can also specify individual targets with **make**. For exmaple, if for some reason we wanted to only build ReadLine.o, we could type

```
make ReadLine.o
```

Note the **cc** will also call **ld**, the linker. In Line 21, for example, the real work of linking all those .o files together into an executable file will be done by **ld**. For this reason, the **cc** command line might include some options which you don't find in the **man** page for **cc**; these are options for **ld**, and will be passed on to the latter by **cc**.

3 More Sophisticated Use of Make

The **make** program is extremely powerful, and is capable of much more than we see in this simple introduction. For example, one can define variables, as in the following version of Makefile:

```
CC = cc

wc: Main.o ReadLine.o WordCount.o PrintLine.o
        $(CC) -o wc Main.o ReadLine.o WordCount.o PrintLine.o

Main.o: Main.c Include.h
        $(CC) -g -c Main.c

ReadLine.o: ReadLine.c Include.h
        $(CC) -g -c ReadLine.c

WordCount.o: WordCount.c Include.h
        $(CC) -g -c WordCount.c

PrintLine.o: PrintLine.c Include.h
        $(CC) -g -c PrintLine.c
```

Here we have defined the variable CC (when referring to it, we must include the dollar sign and parentheses) to be cc, the usual C compiler. The advantage of doing this is that if we instead wanted to use the Gnu C compiler, gcc, all we would have to do is change that one line in Makefile to

```
CC = gcc
```

instead of having to change all the lines where cc had been used. For large, complex programs with very long makefiles, this is a big help.

Note that any shell command can be put in the target entries; this is also a very powerful feature.

To learn more, see the man page as a first step, and the GNU documentation (available with **make** at ftp sites).

4 Library Archive Files

4.1 The Basics

When we have a collection of functions which often use, it is convenient to collect their compiled versions into a **library archive** file, which has a suffix of, for example, .a. Say for example we have some functions in files x.c and y.c. After compiling them into .o files (using cc's -c option), we could then type

```
ar r z.a x.o y.o
```

This would create the file z.a containing x.o and y.o. (After running **ar**, we often will follow up with **ranlib**, which will add an index of the contents to the file.) If we then had a source file w.c which made calls to functions in x.c and y.c, we could simply type

```
cc w.c z.a
```

The compiler would compile w.c and then link the modules in z.a to it, resulting in an executable file a.out.

You see above how you can make your own library archives. There are also archives for the C library, for the X11 windows functions and so on, which are on every Unix system (though they may be in different directories). If for example you want to call the C library's square root function, sqrt(), you put "-lm" on your compiler command line, e.g.

```
cc r.c -lm
```

In doing so you are also linking in a library archive, libm.a, which contains all the C library math functions. The notation "-lsomething" means, "link in an archive named libsomething.a"

Where does the compiler find these library archives? First, the compiler knows to search in certain "official" directories, such as /usr/lib. But if you have your own archive elsewhere, you have various options available to you.

For example, you can link archives in explicitly, by placing the full path name on the compiler command line, e.g.

```
cc abc.c /u/v/libq.a
```

(Note again that here **cc** is not only compiling abc.c, but is also linking the archive file /u/v/libq.a.)

Or, if your archive is named libsomething.a, you can use the compiler's -L option, say

```
cc def.c -L/u/v -lq
```

This tells the compiler that it should add the directory /u/v to the list of directories it searches for libraries, including in this case libq.a.

Similarly, the compiler's -I option tells the compiler to add the given directory to the list of directories where the compiler searches for include-files specified by "elbows."

For instance, say the file ghi.c contains a line

```
#include <zzz.h>
```

Then

```
cc -I/m/n ghi.c
```

tells the compiler to look for this include-file (and others) in /m/n, in addition to its standard spot /usr/include.

Typical makefiles contain many -L and -I constructs.

4.2 Advanced Material

4.2.1 Dynamic Libraries

By default, libraries are *statically linked*. For example, in our command line above,

```
cc abc.c /u/v/libq.a
```

then the resulting executable file, a.out, will contain a copy of libq.a.

But if many programs use libq.a, this would be wasteful of disk space, since that would mean that many copies of libq.a would be stored on disk.

A better alternative would be *dynamic linking*. In this case, instead of a.out containing a copy of libq.a, there would be "note" in that file explaining that libq.a is needed, and that when a.out is loaded, the file libq.a should be loaded together with a.out. This way, the disk would have only one copy of libq.a, not many.

In Unix, the dynamically-linked library files are generally given .so ("shared object") suffixes in their names, e.g. libm.so.6 for the math library, where the `6' indicates a version number.[5]

Creation of dynamically-linkable libraries requires careful usage of the compiler, linker and the shell environment. It is not enough to use the -L and -l command-line options at compile time, because at run time the loader will not know which directories to look in to find the dynamic library specified by the "note."

One approach to solving this problem is to have the user set the environment variable LD_LIBRARY_PATH. In the C shell, for example, the directories /x/y/z can be added to this path via the command

```
setenv  LD_LIBRARY_PATH /x/y/z
```

This is the easy way to go, and I recommend it. However, you may not want to bother the user with this, and there are other reasons to avoid it.[6] Or you can add a line

```
/x/y/z
```

to your file **/etc/ld.so.conf**, which will take effect the next time you boot up.

The GNU version of C/C++ compiler, **gcc**, features a command-line option, -fPIC, specifying that the compiler create *position-independent code*, while allows it to exist anywhere in memory, rather than at a particular distance from a.out. The **ld** command-line option -rpath (again, this is typically passed to **ld** by **gcc**), results in a note in a.out specifying which directory to look for the library if the OS does not find it in the places the OS usually searches. The **ld** option -Bdynamic specifies that the executable file a.out load its libraries dynamically.

The original library load path is gotten at bootup from the file /etc/ld.so.conf

A lot of this can be automated if one uses a GNU program, **libtool**, when invoking **gcc**. The command line will have the form

```
libtool gcc <options>
```

So, here we are actually running **libtool**, but it in turn fires up **gcc**. The design of **libtool** is such that it will provide exactly the right options to **gcc**, e.g. -fPIC, in order to produce dynamically-linked executables.[7]

Sometimes you might download a binary package from the Web, e.g. free RPM packages for Linux, only to find that they tell you a library is missing. You can use the **ldd** command to get more detailed information, and then either download the missing libraries, or if you have them, put them in directories where the OS will see them.

4.2.2 How Can One Tell What Is in a .a File?

Say we have libabc.a. Then the two commands,

```
ar t libabc.a
nm -s libabc.a
```

will tell us which files libabc.a was created from, and which symbols - function and variable names - are in the file.

Footnotes:

[1]This became so popular that versions of **make** were later developed for PCs running Windows, etc., and **make** is now considered a standard software engineering tool, as indispensable as a keyboard.

[2]By the way, if you need to have a very long entry, spanning several lines, place a backslash at the end of each line, serving as a continuation character. Do not-**do NOT!**-put lines longer than 80 characters in your Makefile, as it may produce bizarre errors which you-and others whom you might consult-will waste a large amount of time trying to figure out.

[3]We could call **ld** ourselves, but this is rather complicated.

[4]Of course, all the special options of **cc** which we are using here, e.g. -o in Line 4, belong to **cc**, not to **make**; we could use these same options if we were invoking **cc** from the shell.

[5]Often you will see what appear to be two or more files with similar-looking names, e.g. /lib/libm-2.1.2.so and /lib/libm.so.6, but usually the files will be identical, one being a symbolic link of the others, set up using the Unix **ln** command.

[6]See "Why LD_LIBRARY_PATH is bad," http://www.visi.com/ barr/ldpath.html

[7]In intermediate steps, by the way, **libtool** creates files with suffixes of .lo intead of .o and .la instead of .a, so if you see these files when you compile packages made available on the Web, for example, you will know what they are.

File translated from TEX by TTH, version 3.30.
On 4 Jul 2005, 14:20.