How to Write Compile Scripts

Compiling and Linking With Unix Libraries

Norman Ramsey

Tufts University

nr@cs.tufts.edu

The world's software is organized as libraries

C files that work together to solve problems are often grouped into a *library*. Hanson's book *C Interfaces and Implementations* describes a library that is mostly data structures, with some string processing and arithmetic thrown in. The portable image (netpbm) formats come with a library. There is a library for reading, writing, and analyzing JPEG images. The Gnome graphical toolkit GTK is a (very large) library, as is the QT toolkit used by KDE.

Well-designed libraries are structured along these lines:

- There are *header files* (`.h` files), which provide the *interfaces* to the different components of the library.
- There are *implementations*. These implementations originate as a collection of source files in C, C++, or some other language. Each implementation is then compiled into a *relocatable object file* (`.o` file), and finally, the object files are collected into one or more *archive files* (`.a` file). Confusingly, the archive file is often called a "library."

  It is common also to compile implementations to *position-independent object code*, and from which one can build *dynamically linked libraries* (`.so` files on Unix, `.DLL`s on Windows).

When solving a new problem, it is often best to organize part of the solution as a library. Write your own interfaces and implementations, and link your main programs against your own relocatable object code!

Code that depends on a library is *client* code

When one implementation uses another, for example when `brightness` uses `Pnmrdr`, we say that the `brightness` implementation is a *client* of the `Pnmrdr` interface. In C, this client relationship is established by using the `#include` directive of the C preprocessor:

```
#include "pnmrdr.h"
```

It is crucial that *if an interface changes, all of its clients must be recompiled.* Failure to recompile can lead to core dumps. Managing timely recompilation is tedious, which is why Stu Feldman invented the *Make* program. Make examines a "Makefile" and automatically recompiles an implementation if its source code has changed or if any interface of which it is a client has changed.

A Makefile records two kinds of information:

- *Dependencies* say which implementation is a client of which interfaces. Each dependency is listed against a compiled version of an implementation; for example,

  ```
  brightness.o: /comp/40/include/pnmrdr.h
  ```

- *Recipes* that say how to build a file from sources. Here's an example recipe for building an object file from a C source. This recipe is a single command, which could be written on one line, but to make it fit the narrow column I've broken it up, using a backslash to continue a long line:

  ```
  brightness.o: brightness.c
          gcc -o brightness.o \
            -I/usr/include/cii40 \
            -O -g -c -Wall -Werror \
            -Wextra -Wfatal-errors \
            -std=c99 -pedantic \
            brightness.c
  ```

  Hidden in all this verbiage is the telltale `-c` option, which reveals that this command is compiling and not linking.

Programs are linked against object code and libraries

If all we ever did was build libraries, we'd never be able to run any programs. Eventually we have to *link* together object files *and* libraries to make an *executable binary* (traditionally called `a.out` on Unix and a `.EXE` file on Windows). Confusingly, it's not practical for us to call the linker directly; instead we have `gcc` call the linker on our behalf. You can tell the difference because if `gcc` is just compiling, there is a `-c` option; if it is linking, there isn't.[1]

When we compile, we have to manage the dependencies of clients on their interfaces. In both C and C++, these

[1] The distinction between compilation and linking is subject to some important programming conventions. When compiling, `-I` options are common, and `.c` files should appear on the command line; `-L` and `-l` should not be used. When linking, `-L` and `-l` options may appear, and `.o` files should appear on the command line; `-I` should not be used. No command line should ever mix `.c` and `.o` files.

dependencies are hard to manage. The main problem is that C and C++ leave it up to the programmer to keep track of dependencies, and worse, the dependencies show up in multiple places—there is no single point of truth.

- Dependencies of implementations on interfaces are manifest in source files as `#include` directives.
- Dependencies of implementations on interfaces must also be explicit in the Makefile. If the Makefile gets out of sync with the source code, havoc can ensue. For this reason, large projects *generate dependencies automatically*. In COMP 40 we will run screaming from this problem; in fact, we will avoid Make entirely.
- Dependencies *also* have to be made manifest on the *linker command line*. If you're linking a program, you have to mention not only object file containing the `main` function, but also all the other object files on which it depends, and the object files on which *they* depend, and so on, until you get everything you need.[2] In C and C++, this process is almost never automated. In COMP 40, we can't avoid this problem—to construct useful linker lines for your compile scripts, you'll have to think.

Another problem is that without your help, the C compiler can't always find the things on which your code depends.

- When you `#include` a `.h` file, sometimes the C compiler can find it without help, but other times the compiler must be told explicitly where to look for the `.h` file. The compiler looks in `/usr/include` without being told. But in the example recipe above, the `.h` files for the CII40 library have been installed in the directory `/usr/include/cii40`, and the compiler must be told, via the command-line option `-I/usr/include/cii40`, to look for them there. Omitting the `-I` option can result in *errors that are difficult to diagnose*. You have to look carefully for an error message saying that an `#include` has failed, because all too often this error message is followed by a spray of irrelevant, spurious error messages that have nothing to do with the real problem.[3]

  A more insidious problem is that you forget an `#include` when one is needed. Some symptoms of this problem are

  - The compiler complains of an "implicit declaration" of a function. Look up the function in section 3 of the Unix manual (e.g., `man 3 exit`) and the man page will tell you what `#include`s you need.
  - The compiler complains about something the declaration of a variable of abstract type. Maybe you forgot to `#include` the header containing the `typedef`?

- The linker never knows what libraries you might need—it has no way to tell what `.h` files go with what `.a` or `.so` files. Instead you have to name each library you want using the `-l` command-line option. For a programmer getting started with a new library, it can be hard to know what `-l` options are needed. For example, most of the C standard library does not require a `-l` option, because `-lc` is implicit in a standard linking command. But if you use functions from the header file `math.h`, you need to add `-lm` to your linker command line. This requirement routinely trips up beginning C programmers.

  How can you diagnose a missing `-l` option? If you are missing a library, the linker will probably complain about an "undefined reference." For example, if the linker complains of an undefined reference to `Pnmrdr_new`, you will have to know that this function is declared in header file `pnmrdr.h`, and you should guess that you are probably missing the linker option `-lpnmrdr`.[4]

  Another stumbling block for Unix programmers is that the Unix linker is picky about the *order* in which the `-l` options are given. If one library is a client of another, the client must be listed first. For example, because `Pnmrdr` depends on CII40, the `-lpnmrdr` option must come before `-lcii40`. If you get the order wrong, you will get a similar error message (undefined reference), but it will be harder to diagnose the problem.
- Occasionally the library you want is stored in a nonstandard place. In that case, you also need a `-L` option to tell the linker where to look for your libraries. (The standard place is generally `/usr/lib` on 32-bit systems and `/usr/lib64` on 64-bit systems; the option `-lm` tells the linker to look for `/usr/lib64/libm.so` or `/usr/lib64/libm.a`. If you tell it `-L/comp/40/lib64` it will also look for `/comp/40/lib64/libm.so` or `/comp/40/lib64/libm.a`. Our class-specific libraries will be available in `/comp/40/lib64`, and the corresponding interfaces will be in `/comp/40/include`.)

The burden on you is that compiling and linking successfully may require many, many command-line options. For example, on my machine at home, CII40 is installed in a nonstandard directory under `/usr/local`. (I followed the instructions at the CII web site.) As a result, to link my `brightness` program requires an unholy incantation:

```
gcc -O -g -std=c99 -pedantic \
  -Wall -Wextra -Werror -Wfatal-errors \
  -o brightness brightness.o \
  -L/comp/40/lib64 -lpnmrdr \
  -L/usr/local/cii-3.0 -lcii40
```

[2] The technical term for "everything" is the *transitive closure*.

[3] The people who write `gcc` don't care about their users, so they don't bother fixing the spurious error messages, even though techniques for doing so have been published for at least ten years. There is a better compiler called `clang`, but unfortunately it is not compatible with Red Hat Enterprise Linux 5.

[4] If you want to avoid guesswork, you have to find file `libpnmrdr.a` and us the `nm` command to find out if it provides the missing function. See the man page for `nm`.

The critical difference here, which is easy to miss, is that there is no `-c` option, which tells `gcc` it is supposed to link everything together and make an executable binary. The command does not mention any `.c` files, so there is no need for a `-I` option, but the `-L` options are needed to make `-lpnmrdr` and `-lcii40` work.

Easing the pain by using compile scripts

A good way to ease the pain is to forget about dependencies and Make; just compile every source program every time something changes.[5] Create a shell script called `compile` that compiles every C file in sight, using the right options. Your script should also create executable binaries from the `.o` files that result. Figure 1 (previous page) shows a rather elaborate script that creates the executable binaries needed for the first assignment. Some things to notice:

- I'm lucky to be using software supported by `pkg-config`: `pkg-config --cflags cii40` produces the compiler options needed to find the CII40 `.h` files when compiling with the `-c` option. And `pkg-config --libs cii40` produces the `-L` and `-l` options needed to link against the CII40 package.
- I've put frequently used options in shell variables. Common flags are in `FLAGS`; options for compiling are in `CFLAGS`, and options for linking are in `LFLAGS`. If you depend on new packages you may need to add new `-I` options to `CFLAGS`, and you will almost certainly need to add new `-l` options to `LIBS`.
- The script compiles every `.c` file in the directory. If you have code that doesn't compile, you can comment it out using `#if 0` and `#endif`.
- There is one `case` construct for each executable binary that the script knows how to link. The default is to link all binaries; but you can also suppress linking with the `-nolink` option,

If shell scripting is new to you, there is one trick which may help you understand what's going on:

- If at the top of the file, you insert the command

  ```
  set -x
  ```

  The shell will print every command before it is executed. This trick can be handy if you need to figure out where things are going wrong.

Advanced courses: Easing the pain with makefiles

For larger projects, compile scripts can be tedious. At some point you'll want a way to avoid recompiling every source file every time any character changes.

- Learn to write Makefiles. Don't use GNU Make; it is a monument to complexity and very difficult to understand. Like Stu Feldman, the creator of the original Make, I recommend the version called `mk` (pronounced "muck"), which was created for Version 7 Unix by Andrew Hume. This version is much simpler and easier to learn; it is the version I recommend you use in future courses. As of Fall 2011, it can be found at `http://swtch.com/plan9port/unix`.
- If you're lucky enough to be using software supported by `pkg-config`, then `pkg-config--cflags` will produce the compiler options needed to find the `.h` files when compiling with the `-c` option. And `pkg-config--libs` will produce the `-L` and `-l` options needed to link against the package.
- You can put frequently used options in a Makefile variable. `CFLAGS` is traditional for compiling, and often both `CFLAGS` and `LDFLAGS` are used for linking.
- You can generate accurate dependencies by using the `-MM` option of `gcc`, as in

  ```
  gcc -I/usr/include/cii40 -MM brightness.c
  ```

  The `-MM` option can be annoying because it produces *all* the dependencies, even dependencies on system `.h` files that are never supposed to change. It is useful primarily when generating Makefiles automatically.

[5] We follow Ken Thompson's advice "when in doubt, use brute force."

```
#!/bin/sh

########### the initial part will be the same for all assignments
set -e    # halt on first error

link=all  # link all binaries by default
linked=no # track whether we linked

case $1 in
  -nolink) link=none ; shift ;;  # don't link
  -link)   [ -n "$2" ] || { echo "You need to say *what* to link" >&2; exit 1; }
           link="$2" ; shift ; shift ;;  # link only one binary
esac

# use 'gcc' as the C compiler (at home, you could try 'clang')
CC=gcc

# the next two lines enable you to compile and link against CII40
CIIFLAGS=`pkg-config --cflags cii40`
CIILIBS=`pkg-config --libs cii40`

# the next three lines enable you to compile and link against course software
CFLAGS="-I. -I/comp/40/include $CIIFLAGS"
LIBS="$CIILIBS -lm"    # might add more libraries for some projects
LFLAGS="-L/comp/40/lib64"

# these flags max out warnings and debug info
FLAGS="-g -O -Wall -Wextra -Werror -Wfatal-errors -std=c99 -pedantic"

rm -f *.o  # make sure no object files are left hanging around

case $# in
  0) set *.c ;; # if no args are given, compile all .c files
esac

# compile each argument to a .o file
for cfile
do
  $CC $FLAGS $CFLAGS -c $cfile
done

########### the middle part is different for each assignment
# link together .o files + libraries to make executable binaries
# using one case statement per executable binary
case $link in
  all|brightness) $CC $FLAGS $LFLAGS -o brightness brightness.o -lpnmrdr $LIBS
                  linked=yes ;;
esac

case $link in
  all|fgroups)    $CC $FLAGS $LFLAGS -o fgroups    fgroups.o             $LIBS
                  linked=yes ;;
esac

########### the final part is the same for each assignment
# error if asked to link something we didn't recognize
if [ $linked = no ]; then
  case $link in  # if the -link option makes no sense, complain
    none) ;; # OK, do nothing
    *) echo "`basename $0`: don't know how to link $link" 1>&2 ; exit 1 ;;
  esac
fi
```

Figure 1. A compile script for the first COMP 40 assignment