Appendix O: Tools for C programming

Ben Klemens

April 4, 2009

This is the online appendix to the book Modeling with Data. It explains the tools that go into a productive C environment, and how to install them. The intended audience is people who are computer-literate with a Windows/Mac-type graphical system, but who are not familiar with the POSIX toolchain.

The problem is that you have choices. Because C is a standard and not a product, there are an endless number of tools for writing your code, systems that will compile that code, and libraries whose functions you can use. That means that no one can put together a definitive C development package—so you will need to select and gather tools yourself.

The package manager

The *package manager* offers a consistent means of downloading all the pieces of the puzzle from online sources. With a package manager, the installation process is short; without one, the process is nasty and brutish.

If you are using a POSIX system, you are probably already familiar with your package manager. Red Hat Package Manager [RPM] and Debian's Apt are the most popular, and both have many front-ends, with names like YaST and synaptic.

If you are using Windows, you have a few options. One is to get Mingw, which provides a compiler and some basic auxiliary tools. It has its own installation program and some package facilities. Another option (which I recommend) is Cygwin[1]. It is free, has a full-fledged package manager, and provides an entire POSIX subsystem, with the attendant compilers, editors, and so on. Along the same lines, the Portable Ubuntu[2] is a version of the Ubuntu Linux distribution that runs as a program under Windows. It is newer than Cygwn, and so has a predictable set of plusses and minuses: no-install setup, more processor intensive, more features, less tested.

Mac OS X users will be using the terminal (typically found in the accessories subfolder of the applications folder) to do most of the work below. The package manager of choice is Fink[3]. You may also need to download Xcode[4], Apple's development toolchain, to get Fink and compilation to work.

[1]`http://cygwin.com/`
[2]`http://portableubuntu.sourceforge.net/index.php?section=download`
[3]`http://fink.sourceforge.net/`
[4]`http://developer.apple.com/tools/download/`

Now that you have a package manager, which packages out of the thousands of available packages should you install? The first element you will need is a *compiler*. This book is centered around `gcc`, and all package managers offer it; some systems offer other C compilers that are as efficacious. You will also rely heavily on a debugger; the companion to gcc is `gdb`. You will need the `make` utility, which is discussed in full in Appendix A.

The next step is picking function libraries. This book relies heavily on four libraries (beyond the standard C library that comes with the C compiler): the GSL, the GSL BLAS, SQLite3, and Apophenia.

Not all package systems include all four libraries. Also, a library includes two components: the object files that one needs to run programs that use the library's functions, and the header files that describe those functions. Both are necessary for compiling new C programs as you will be doing. Also, some package systems have the annoying habit of separating the documentation for libraries into a separate package.

Thus, you will need to look for several packages to fully install a library. For example, the GSL may be divided into packages named `gsl`, `libgsl`, `libgsl-devel`, and `gsl-doc`. There is unfortunately no common custom or standard for naming, but you will almost certainly need at least one `-dev` or `-devel` package for each library (possibly including `libc6-devel`). When in doubt as to whether you need a package, install it. [However, don't try to install *all* the packages—you could be waiting a day for the download and your hard drive probably won't be able to hold them all.]

Some of the above libraries may be entirely missing from your package manager. In this case, see below about compiling the libraries from source code.

While you are at the package manager, why not stock up? Because there is a standardized and automated means of installing any program, you can easily pull down Gnuplot, unzip (to open compressed files), a chess computer, PDF viewers/generators, CD-to-MP3 converters (like grip or notlame), the $\TeX$/$\LaTeX$ document preparation systems, or a photo-editing system like the GIMP. Unlike certain operating systems of old, installing more packages onto a POSIX subsystem does not reduce the stability of the system: the additional programs and libraries just take up more space on the hard drive.

IDEs

Returning to setting up your environment for writing scripts, you broadly have the choice of two paradigms in which to work. The first is the integrated development environment (IDE). This is an all-in-one environment comparable to a multiwindow stats package, with one window for your program, one for compilation, one for output, et cetera. Popular choices include Dev-C++ or Eclipse, which are available via most package managers. Many other IDEs of varying quality are available for any graphically-oriented operating system.

The other option is via the command line. Since you are certainly using a system that supports multiple windows,[5] you are basically using your operating system as an

[5]Even if you are dialing in to a server's single-window terminal, you can either dial in twice, or use

IDE. In this paradigm, you can have one window that is dedicated to a text editor with your code, and another window or two for compilation, debugging, and output.

The command prompt

Even if you are devoted to your IDE, it is worth knowing your command prompt. You will need it to install libraries that are not avaialable via the package manager, you may need it to set environment variables, and you may find it helpful when your IDE seems to be acting strangely.

Linux/UNIX users will be using xterm, Eterm, Aterm, gnome-terminal, rxvt, or any of a number of other such options. Mac users, the terminal can be found in the Applications→Accessories folder.

Windows users, you will be doing work from the Cygwin prompt; if you did not ask Cygwin to put an icon on your desktop or menu bar, you can find a Cygwin folder among the other program folders in the Start menu. By default, the Cygwin prompt is just a script run from Windows's command box, which is not very pleasing. Cygwin can install an Xterm, which provides many æsthetic and practical benefits over the Windows command box. If you installed Gnuplot or any graphical games, then Cygwin also installed the X Window subsystem. Type `startx` at the Cygwin prompt to get an Xterm, from which you can run the various graphical programs, and enjoy a better command-prompt experience.

Users of graphical window systems tend to think of the desktop as the base for all their data; users of command-line systems have a home directory. So the first bit of orientation is finding out how to get to one from the other.

In Linux and MacOS, it's easy: the Desktop directory is directly inside the home directory.

Cygwin sets up its own filesystem. The default is that it is based at `c:\cygwin`, but you may have selected something else in the first step of the Cygwin setup. Within that directory, you will find a series of directories with short names, like usr, etc, lib, and home. These are the customary base directories for a POSIX filesystem, harking back to the days when letters were expensive. Inside the `home` directory, you will naturally find your home directory. In the other direction, what Windows calls its `c:` drive, Cygwin calls `/cygdrive/c`, so your home directory is likely in `/cygdrive/c/Documents\ and\ Settings/yourname`.

As with most shells, you can use tab completion to type long names. Try typing `/cygdrive/c/Doc` and then hit the <tab> key; the system should fill in the rest for you. Notice that the directory-dividing slashes in POSIX systems are standard forward slashes (over the ? key on standard US keyboards), not backslashes (sharing a key with the —).

Help

Over time, documentation has grown increasingly interactive and hyperlinked. The original UNIX systems included manual pages via the `man` command, and there are

`screen` to multiplex the window.

manual pages for most of the C functions in the standard libraries. Try `man printf` or `man atoi`, for example.

By the mid-90s, the TEXinfo format emerged. TEXinfo documents require a TEXinfo reader (EMACS can serve as one, or there is a standalone version), which can navigate among links and tables of contents in the documentation. TEXinfo documentation is a part of the GNU coding standards, so you are generally guaranteed that parts of the GNU toolchain, such as `gcc` and the version of `make` that you are probably using, will have TEXinfo documentation. You can read these documents by commands such as `info gcc` or `info make`. If you have trouble navigating in the `info` reader, then you can get help with `info info`.

The current norm for presenting complex documents is the Web page, so more current libraries like Apophenia have documentation formatted for your web browser. Because any one library has more functions than anyone could possibly memorize, consider your web browser to be an integral part of your code-writing environment.

You probably have have all the manuals for both the programs and the functions listed in this chapter on your hard drive now. But if you are missing documentation, you can surely find it online. Just enter the command you would have typed at the command line into your favorite search engine. A Web search for `info gsl` or `man printf` will turn up exactly the documentation that is missing from your system, formatted for your Web browser.

Text files

C programs are files of text, and all of your work will be manipulations of text, so it is in your long-run best interest to get a good text editor. At the very least, you will get error messages listing line numbers, so if your text editor can't tell you which is line 105, you will need to get a new one. Text editors written with programming in mind often color different syntax elements differently, which gives a quick visual indication that you spelled `double` as `doulbe` or forgot an endquote. Most offer an outline or folding mode, that shows functions only as headers so you can see the broad form of your program, but unfold functions as necessary to work on their internals.

The two most popular are EMACS and vi. EMACS is better for people who prefer to have everything under one roof—it is often billed as an IDE—while vi is better for the minimalists and touch typists. Both involve a learning curve, meaning that they will be difficult to use at first, will require reading the manual, and will in the long run save you hours over using simpler text editors such as those typically included with IDEs. Some implementation of both is available for all computer types, and you are encouraged to start learning one or the other now. If neither suits your fancy, there are literally hundreds of others to choose from.

Installing from source

You are guaranteed that the package manager will provide you with a compiler, a make facility, a debugger, and a choice of text editors. However, few package repositories provide all relevant libraries; many focus on consumer- or business-oriented libraries and so pass on numerical libraries.

In this case, you will need to download the source code and compile it yourself.

- Get the GSL from the GSL home page[6]. If you are near North Carolina, maybe try the ibiblio mirror[7].
- For SQLite3, go to the SQLite home page[8].
- Download Apophenia here.

Now that you have the source code, you need to unpack it and compile it. Fortunately, all of these libraries, as with most libraries, use the GNU's Autoconf system to handle almost all configuration issues for you, so you need to do minimal work. Here are the steps:

```
tar xvzf pkg.tgz  #change pkg.tgz to the appropriate name
cd package_dir    #same here.
./configure
make
sudo make install   #see below if you don't have root privileges.
```

If the system can't find `tar` or `make`, then go back to your package manager and install them. The last line runs `make install`, but as the administrator (aka *superuser*, aka *root*). On some systems, you would instead use `su -c "make install"`. Cygwin users can just run `make install`. There are further steps below if you need but do not have root privileges.

Info: Cygwin likes to give you many lines of technical information when compiling, typically about substituting one symbol for another, which look worrisome to many. Don't panic: these really are just informative, and any line beginning with `Info:` does not indicate an error.

LD_LIBRARY_PATH

During one or many of the installation steps above, you probably got a warning that looks like this:

```
----------------------------------------------------------------------
Libraries have been installed in:
   /usr/local/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
[...]
----------------------------------------------------------------------
```

[6]http://sources.redhat.com/gsl
[7]ftp://ftp.ibiblio.org/pub/mirrors/gnu/ftp/gnu/gsl/gsl-1.8.tar.gz
[8]http://www.sqlite.org

If you did, then that means you will need to tell the linker where to look for your newly installed libraries, by adding a line in your shell's configuration file to search for the libraries you'd just installed. Cutting and pasting the following to the command prompt should do the trick. You will only need to do it once. If your operating system's name ends in the letter X, try this:

```
echo "export LD_LIBRARY_PATH=/usr/local/lib:\$LD_LIBRARY_PATH" >> ~/.bashrc
source ~/.bashrc
```

If you are using Cygwin:

```
echo "export PATH=/usr/local/lib:\$PATH" >> ~/.bashrc
echo "export LIBRARY_PATH=/usr/local/lib:\$LIBRARY_PATH" >> ~/.bashrc
source ~/.bashrc
```

These commands add a line to your `.bashrc` file, which starts every time the shell (typically `bash`) starts.

However, environment variables frequently differ from system to system, so the above may not work for you. For more on environment variables and setting your library path, see Appendix A of the main textbook.

Access denied

You may be working on a system where you do not have access to the places to which libraries are typically installed. You will need to create a subdirectory in your home directory in which to install packages. The compilation from source will be the same as before, but with one addition: by adding the `--prefix` switch to the `./configure` command.[9] Here is a script to give you the idea.

```
export MY_LIBS = src   #choose a directory name to be created in your
home directory.
tar xvzf pkg.tgz       #change pkg.tgz to the appropriate name
cd package_dir         #same here.
mkdir $HOME/$MY_LIBS
./configure --prefix $HOME/$MY_LIBS
make
make install   #Now you don't have to be root.
echo "export LD_LIBRARY_PATH=$HOME/$MY_LIBS:\$LD_LIBRARY_PATH" >> ~/.bashrc
```

A few final tips

Here are some notes on how I configure my own system. Perhaps some of these little tricks will be useful to you as well.

[9] `configure` is typically very configurable. Try `./configure --help` for a list of options specific to the code you are compiling.

GDB

Adding definitions for your most-used processes to `.gdbinit` can make using `gdb` much more pleasant. For example, you will frequently be viewing vectors and matrices, so it's nice to have a quick way to do so. Add the following to the `.gdbinit` file in your home directory (it may be hidden):

```
define pv
    p apop_vector_show($arg0)
end

define pm
    p apop_matrix_show($arg0)
end

define pd
    p apop_data_show($arg0)
end

define pa
    p *($arg0)@$arg1
end

define mr #one more little convenience
    make
    run
end
```

Then `pv my_vector` or `pm my_matrix` will show the full contents of these items. For arrays, you will need to give a name and a size, like `pa items 5`.

Compiling

Once you have a makefile that works for your system, it will generally work with minimal modification for any program (especially if the code is in one file).

Thus, I have a single standard makefile, which I copy from directory to directory; it is very much like the one provided as sample code. The final program name is not hard-coded (as the sample makefile does), but is simply a variable name, `PROG`. I put a link (or a copy) of the makefile in the directory with today's project, set the `PROG` environment variable, and the system is now entirely set up for compilation.

```
ln -s ~/tech/makefile
export PROG=todaysproject
make run
```

[In fact, I've even aliased `alias e=export` in my `.bashrc` for still less typing: `e PROG=todaysproject`.]