

C Tips from the New School

21st Century C



Free Sampler

O'REILLY®

Ben Klemens

Want to read more?

You can [buy this book](#) at **oreilly.com**
in print and ebook format.

Buy 2 books, get the 3rd FREE!

Use discount code: OPC10

All orders over \$29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer,
including the iBookstore, the [Android Marketplace](#),
and [Amazon.com](#).



O'REILLY®

Spreading the knowledge of innovators

[oreilly.com](#)

21st Century C

by Ben Klemens

Copyright © 2013 Ben Klemens. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Nathan Jepson

Production Editor: Rachel Steely

Copyeditor: Linley Dolby

Proofreader: Teresa Horton

Indexer: Ellen Troutman

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrators: Robert Romano and Rebecca Demarest

November 2012: First Edition.

Revision History for the First Edition:

2012-10-12 First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449327149> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *21st Century C*, the image of a common spotted cuscus, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-32714-9

[LSI]

1350047263

Table of Contents

Preface	ix
----------------------	-----------

Part I. The Environment

1. Set Yourself Up for Easy Compilation	3
Use a Package Manager	4
Compiling C with Windows	6
POSIX for Windows	7
Compiling C with POSIX	8
Compiling C Without POSIX	8
Which Way to the Library?	10
A Few of My Favorite Flags	11
Paths	12
Runtime Linking	15
Using Makefiles	15
Setting Variables	16
The Rules	19
Using Libraries from Source	22
Using Libraries from Source (Even if Your Sysadmin Doesn't Want You To)	23
Compiling C Programs via Here Document	25
Include Header Files from the Command Line	25
The Unified Header	26
Here Documents	27
Compiling from stdin	28
2. Debug, Test, Document	31
Using a Debugger	31
GDB Variables	34
Print Your Structures	36
Using Valgrind to Check for Errors	39
Unit Testing	41

Using a Program as a Library	43
Coverage	44
Interweaving Documentation	45
Doxygen	46
Literate Code with CWEB	47
Error Checking	49
What Is the User's Involvement in the Error?	49
The Context in Which the User Is Working	51
How Should the Error Indication Be Returned?	52
3. Packaging Your Project	55
The Shell	56
Replacing Shell Commands with Their Outputs	56
Use the Shell's for Loops to Operate on a Set of Files	58
Test for Files	59
fc	62
Makefiles vs. Shell Scripts	64
Packaging Your Code with Autotools	66
An Autotools Demo	68
Describing the Makefile with makefile.am	71
The configure Script	75
4. Version Control	79
Changes via diff	80
Git's Objects	81
The Stash	84
Trees and Their Branches	85
Merging	86
The Rebase	88
Remote Repositories	89
5. Playing Nice with Others	91
The Process	91
Writing to Be Read by Nonnatives	91
The Wrapper Function	92
Smuggling Data Structures Across the Border	93
Linking	94
Python Host	95
Compiling and Linking	96
The Conditional Subdirectory for Automake	96
Distutils Bridging with Autotools	98

Part II. The Language

6. Your Pal the Pointer	103
Automatic, Static, and Manual Memory	103
Persistent State Variables	105
Pointers Without malloc	107
Structures Get Copied, Arrays Get Aliased	108
malloc and Memory-Twiddling	111
The Fault Is in Our Stars	112
All the Pointer Arithmetic You Need to Know	113
7. C Syntax You Can Ignore	117
Don't Bother Explicitly Returning from main	118
Let Declarations Flow	118
Set Array Size at Runtime	120
Cast Less	121
Enums and Strings	122
Labels, gotos, switches, and breaks	124
goto Considered	124
switch	125
Deprecate Float	128
8. Obstacles and Opportunity	133
Cultivate Robust and Flourishing Macros	133
Preprocessor Tricks	137
Linkage with static and extern	139
Declare Externally Linked Elements Only in Header Files	141
The const Keyword	143
Noun-Adjective Form	144
Tension	145
Depth	146
The char const ** Issue	147
9. Text	151
Making String Handling Less Painful with asprintf	151
Security	152
Constant Strings	153
Extending Strings with asprintf	154
A Pæan to strtok	156
Unicode	160
The Encoding for C Code	162
Unicode Libraries	163

The Sample Code	164
10. Better Structures	167
Compound Literals	168
Initialization via Compound Literals	169
Variadic Macros	169
Safely Terminated Lists	170
Foreach	171
Vectorize a Function	172
Designated Initializers	173
Initialize Arrays and Structs with Zeros	175
Typedefs Save the Day	176
A Style Note	178
Return Multiple Items from a Function	179
Reporting Errors	180
Flexible Function Inputs	182
Declare Your Function as printf-Style	183
Optional and Named Arguments	185
Polishing a Dull Function	187
The Void Pointer and the Structures It Points To	192
Functions with Generic Inputs	192
Generic Structures	196
11. Object-Oriented Programming in C	201
What You Don't Get (and Why You Won't Miss It)	202
Scope	202
Overloaded with Operator Overloading	205
Extending Structures and Dictionaries	209
Extending a Structure	210
Implementing a Dictionary	214
Base Your Code on Pointers to Objects	218
Functions in Your Structs	219
Count References	223
Example: A Substring Object	224
An Agent-Based Model of Group Formation	228
12. Libraries	235
GLib	235
POSIX	236
Using mmap for Gigantic Data Sets	236
Easy Threading with Pthreads	238
The GNU Scientific Library	246
SQLite	248

The Queries	249
libxml and cURL	250
Epilogue	cclv
Glossary	257
Bibliography	261
Index	263

Set Yourself Up for Easy Compilation

Look out honey 'cause I'm using technology.

—Iggy Pop, “Search and Destroy”

The C standard library is just not enough to get serious work done.

Instead, the C ecosystem has expanded outside of the standard, which means that knowing how to easily call functions from common but not-ISO-standard libraries is essential if you want to get past doing textbook exercises. Unfortunately, this is the point where most textbooks taper off and leave you to work it out for yourself, which is why you can find C detractors who will say self-dissonant things like *C is 40 years old, so you have to write every procedure from scratch in it*—they never worked out how to link to a library.

Here is the agenda for the chapter:

- Setting up the requisite tools. This is much easier than it was in the dark days when you had to hunt for every component. You can set up a full build system with all the frills in maybe 10 or 15 minutes (plus all the download time to load so much good stuff).
- How to compile a C program. Yes, you know how to do this, but we need a setup that has hooks for the libraries and their locations; just typing `cc myfile.c` doesn't cut it anymore. *Make* is just about the simplest system to facilitate compiling programs, so it provides a good model for discussion. I'll show you the smallest possible makefile that offers enough room to grow.
- Whatever system we use will be based on a small set of environment-like variables, so I'll discuss what they do and how to set them. Once we have all that compilation machinery in place, adding new libraries will be an easy question of adjusting the variables we've already set up.
- As a bonus, we can use everything up to this point to set up a still simpler system for compilation, which will let us cut and paste code onto the command prompt.

A special note to IDE users: you may not be a make user, but this section will nonetheless be relevant to you, because for every recipe that make executes when compiling code, your IDE has an analogous recipe. If you know what make is doing, you'll have an easy time tweaking your IDE.

Use a Package Manager

Oh man, if you are not using a package manager, you are missing out.

I bring up package managers for several reasons: first, some of you may not have the basics installed. For you, I put this section first in the book, because you need to get these tools, and fast. A good package manager will have you set up quite rapidly with a full POSIX subsystem, compilers for every language you've ever heard of, a half-decent array of games, the usual office productivity tools, a few hundred C libraries, et cetera.

Second, as C authors, the package manager is a key means by which we can get libraries for folding into our work.

Third, when you've been writing enough code, there will come a time when you want to distribute your code, making the jump from being somebody who downloads packages to being somebody producing a package. This book will take you halfway, showing you how to prepare your package for easy autoinstallation, so that when the administrator of a package repository decides to include your code in the repository, he or she will have no problem building the final package.

If you are a Linux user, you set up your computer with a package manager and have already seen how easy the software obtention process can be. For Windows users, I'll cover [Cygwin](#) in detail. Mac users have several options, such as [Fink](#) and [Macports](#). All the Mac options depend on Apple's Xcode package, typically available on the OS install CD (or directory of installable programs, depending on the vintage), or by registering as a developer with Apple.

What packages will you need? Here's a quick rundown of the usual suspects. Because every system has a different organization scheme, some of these may be bundled differently, installed by default in a base package, or oddly named. When in doubt about a package, install it, because we're past the days when installing too many things could somehow cause system instability or slowdown. However, you probably don't have the bandwidth (or maybe even the disk space) to install every package on offer, so some judgment will be required. If you find that you are missing something, you can always go back and get it later. Packages to definitely get:

- A compiler. Definitely install gcc; Clang may be available.
- gdb, a debugger.
- Valgrind, to test for C memory usage errors.
- gprof, a profiler.

- make, so you never have to call your compiler directly.
- pkg-config, for finding libraries.
- Doxygen, for documentation generation.
- A text editor. There are literally hundreds of text editors to choose from. Here are a few subjective recommendations:
 - Emacs and vim are the hardcore geek’s favorites. Emacs is very inclusive (the *E* is for *extensible*); vim is more minimalist and is very friendly to touch typists. If you expect to spend hundreds of hours staring at a text editor, it is worth taking the time to learn one of them.
 - Kate is friendly and attractive, and provides a good subset of the conveniences we expect as programmers, such as syntax highlighting.
 - As a last resort, try nano, which is aggressively simple, and is text-based, and therefore works even when your GUI doesn’t.
- If you are a fan of IDEs, get one—or several. Again, there are many to choose from; here are a few recommendations:
 - Anjuta: in the GNOME family. Friendly with Glade, the GNOME GUI builder.
 - KDevelop: in the KDE family.
 - Code::blocks: relatively simple, works on Windows.
 - Eclipse: the luxury car with lots of cupholders and extra knobs. Also cross-platform.

In later chapters, I’ll get to these more heavy-duty tools:

- Autotools: Autoconf, Automake, libtool
- Git
- Alternate shells, such as the Z shell.

And, of course, there are the C libraries that will save you the trouble of reinventing the wheel (or, to be more metaphorically accurate, reinventing the locomotive). You might want more, but here are the libraries that will be used over the course of this book:

- libcurl
- libGlib
- libGSL
- libSQLite3
- libXML2

There is no consensus on library package naming schemes, and you will have to work out how your package manager likes to dissect a single library into subparts. There is typically one package for users and a second for authors who will use the library in their own work, so be sure to select both the base package and the `-dev` or `-devel` packages. Some systems separate documentation into yet another package. Some require that you

download debugging symbols separately, in which case gdb should lead you through the steps the first time you run it on something lacking debugging symbols.

If you are using a POSIX system, then after you've installed the preceding items, you will have a complete development system and are ready to get coding. For Windows users, we'll take a brief detour to understand how the setup interacts with the main Windows system.

Compiling C with Windows

On most systems, C is the central, VIP language that all the other tools work to facilitate; on a Windows box, C is strangely ignored.

So I need to take a little time out to discuss how to set up a Windows box for writing code in C. If you aren't writing on a Windows box now, feel free to skip this segment and jump to [“Which Way to the Library?” on page 9](#).

This is not a rant about Microsoft; please do not read it as such. I am not going to speculate on Microsoft's motives or business strategies. However, if you want to get work done in C on a Windows box, you need to know the state of affairs (which is frankly inhospitable) and what you can do to get going.

POSIX for Windows

Because C and Unix coevolved, it's hard to talk about one and not the other. I think it's easier to start with POSIX. Also, those of you who are trying to compile code on a Windows box that you wrote elsewhere will find this to be the most natural route.

As far as I can tell, the world of things with filesystems divides into two (slightly overlapping) classes:

- POSIX-compliant systems
- The Windows family of operating systems

POSIX compliance doesn't mean that a system has to look and feel like a Unix box. For example, the typical Mac user has no idea that he or she is using a standard BSD system with an attractive frontend, but those in the know can go to the Accessories → Utilities folder, open the Terminal program, and run `ls`, `grep`, and `make` to their hearts' content.

Further, I doubt that many systems live up to 100% of the standard's requirements (like having a Fortran `77 compiler). For our purposes, we need a shell that can behave like the barebones POSIX shell, a handful of utilities (`sed`, `grep`, `make`, ...), a C99 compiler, and additions to the standard C library such as `fork` and `iconv`. These can be added as a side note to the main system. The package manager's underlying scripts, Autotools, and almost every other attempt at portable coding will rely on these tools to some

extent, so even if you don't want to stare at a command prompt all day, these tools will be handy to have for installations.

On server-class OSes and the full-featured editions of Windows 7, Microsoft offers what used to be called INTERIX and is now called the Subsystem for Unix-based Application (SUA), which provides the usual POSIX system calls, the Korn shell, and gcc. The subsystem is typically not provided by default but can be installed as an add-on component. But the SUA is not available for other current editions of Windows and will not be available for Windows 8, so we can't depend on Microsoft to provide a POSIX subsystem for its operating systems.

And so, Cygwin.

If you were to rebuild Cygwin from scratch, this would be your agenda:

1. Write a C library for Windows that provides all the POSIX functions. This will have to smooth over some Windows/POSIX incongruities, such as how Windows has distinct drives like C: while POSIX has one unified filesystem. In this case, alias C: as */cygdrive/c*, D: as */cygdrive/d*, and so on.
2. Now that you can compile POSIX-standard programs by linking to your library, do so: generate Windows versions of *ls*, *bash*, *grep*, *make*, *gcc*, *X*, *rxvt*, *libglib*, *perl*, *python*, and so on.
3. Once you have hundreds of programs and libraries built, set up a package manager that allows users to select the elements they want to install.

As a user of Cygwin, all you have to do is download the package manager from the setup link at [Cygwin's website](#) and pick packages. You will certainly want the preceding list, plus a decent terminal (try RXVT, or install the X subsystem and use the xterm), but you will see that virtually all of the luxuries familiar from a development system are there somewhere. Now you can get to compiling C code.

Compiling C with POSIX

Microsoft provides a C++ compiler, in the form of Visual Studio, which has an ANSI C compatibility mode. This is the only means of compiling C code currently provided by Microsoft. Many representatives from the company have made it clear that C99 support (let alone C11 support) is not forthcoming. Visual Studio is the only major compiler that is still stuck on C89, so we'll have to find alternative offerings elsewhere.

Of course, Cygwin provides gcc, and if you've followed along and installed Cygwin, then you've already got a full build environment.

If you are compiling under Cygwin, then your program will depend on its library of POSIX functions, *cygwin1.dll* (whether your code actually includes any POSIX calls or not). If you are running your program on a box with Cygwin installed, then you obviously have no problem. Users will be able to click on the executable and run it as expected, because the system should be able to find the Cygwin DLL. A program

compiled under Cygwin can run on boxes that don't have Cygwin installed if you distribute *cygwin1.dll* with your code.

On my machine, this is *(path to cygwin)/bin/cygwin1.dll*. The *cygwin1.dll* file has a GPL-like license (see “[The Legal Sidebar](#)” on page xvi), in the sense that if you distribute the DLL separately from Cygwin as a whole, then you are required to publish the source code for your program.¹ If this is a problem, then you'll have to find a way to recompile it without depending on *cygwin1.dll*, which means dropping any POSIX-specific functions from your code and using MinGW, as discussed later. You can use *cygcheck* to find out which DLLs your program depends on, and thus verify that your executable does or does not link to *cygwin1.dll*.

Compiling C Without POSIX

If your program doesn't need the POSIX functions (like `fork` or `popen`), then you can use MinGW (Minimalist GNU for Windows), which provides a standard C compiler and some basic associated tools. Msys is a companion to MinGW that provides other useful tools, such as a shell.

The lack of POSIX-style amenities is not the real problem with MinGW. Msys provides a POSIX shell, or leave the command prompt behind entirely and try [Code::blocks](#), an IDE that uses MinGW for compilation on Windows. Eclipse is a much more extensive IDE that can also be configured for MinGW, though that requires a bit more setup.

Or if you are more comfortable at a POSIX command prompt, then set up Cygwin anyway, get the packages providing the MinGW versions of gcc, and use those for compilation instead of the POSIX-linking default version of Cygwin gcc.

If you haven't already met Autotools, you'll meet it soon. The signature of a package built using Autotools is its three-command install: `./configure`; `make`; `make install`. Msys provides sufficient machinery for such packages to stand a good chance of working. Or if you have downloaded the packages to build from Cygwin's command prompt, then you can use the following to set up the package to use Cygwin's Mingw32 compiler for producing POSIX-free code:

```
./configure --host=ming32
```

Then run `make`; `make install` as usual.

Once you've compiled under MinGW, via either command-line compilation or Autotools, you've got a native Windows binary. Because MinGW knows nothing of *cygwin1.dll*, and your program makes no POSIX calls anyway, you've now got an executable program that is a bona fide Windows program, that nobody will know you compiled from a POSIX environment.

1. Cygwin is a project run by Red Hat, Inc., who will also allow users to purchase the right to not distribute their source code as per the GPL.

No, the real problem with MinGW is the paucity of precompiled libraries.² If you want to be free of *cygwin1.dll*, then you can't use the version of *libglib.dll* that ships with Cygwin. You'll need to recompile GLib from source to a native Windows DLL—but GLib depends on GNU's gettext for internationalization, so you'll have to build that library first. Modern code depends on modern libraries, so you may find yourself spending a lot of time setting up the sort of things that in other systems are a one-line call to the package manager. We're back to the sort of thing that makes people talk about how C is 40 years old, so you need to write everything from scratch.

So, there are the caveats. Microsoft has walked away from the conversation, leaving others to implement a post-grunge C compiler and environment. Cygwin does this and provides a full package manager with enough libraries to do some or all of your work, but it is associated with a POSIX style of writing and Cygwin's DLL. If that is a problem, you will need to do more work to build the environment and the libraries that you'll need to write decent code.

Which Way to the Library?

OK, so you have a compiler, a POSIX toolchain, and a package manager that will easily install a few hundred libraries. Now we can move on to the problem of using those in compiling our programs.

We have to start with the compiler command line, which will quickly become a mess, but we'll end with three (sometimes three and a half) relatively simple steps:

1. Set a variable listing the compiler flags.
2. Set a variable listing the libraries to link to. The half-step is that you sometimes have to set only one variable for linking while compiling, and sometimes have to set two for linking at compile time and runtime.
3. Set up a system that will use these variables to orchestrate the compilation.

To use a library, you have to tell the compiler that you will be importing functions from the library twice: once for the compilation and once for the linker. For a library in a standard location, the two declarations happen via an `#include` in the text of the program and a `-l` flag on the compiler line.

Example 1-1 presents a quick sample program that does some amusing math (for me, at least; if the statistical jargon is Greek to you, that's OK). The C99-standard *error function*, `erf(x)`, is closely related to the integral from zero to x of the Normal distribution with mean zero and standard deviation $\sqrt{2}$. Here, we use `erf` to verify an area

2. Although Msys, MinGW, and a few other elements are provided as packages, this handful of packages pales in comparison to the hundreds of packages provided by the typical package manager. Notably, precompiled libraries are not a one-click or one-command install. However, by the time you read this, my complaint may have been addressed, and there might be many more MinGW packages available.

popular among statisticians (the 95% confidence interval for a standard large- n hypothesis test). Let us name this file *erf.c*.

Example 1-1. A one-liner from the standard library. (erf.c)

```
#include <math.h> //erf, sqrt
#include <stdio.h> //printf

int main(){
    printf("The integral of a Normal(0, 1) distribution "
           "between -1.96 and 1.96 is: %g\n", erf(1.96*sqrt(1/2.)));
}
```

The `#include` lines should be familiar to you. The compiler will paste *math.h* and *stdio.h* into the code file here, and thus paste in declarations for `printf`, `erf`, and `sqrt`. The declaration in *math.h* doesn't say anything about what `erf` does, only that it takes in a `double` and returns a `double`. That's enough information for the compiler to check the consistency of our usage and produce an object file with a note telling the computer: once you get to this note, go find the `erf` function, and replace this note with `erf`'s return value.

It is the job of the linker to reconcile that note by actually finding `erf`, which is in a library somewhere on your hard drive.

The math functions found in *math.h* are split off into their own library, and you will have to tell the linker about it by adding an `-lm` flag. Here, the `-l` is the flag indicating that a library needs to be linked in, and the library in this case has a single-letter name, `m`. You get `printf` for free, because there is an implicit `-lc` asking the linker to link the standard `libc` assumed at the end of the linking command. Later, we'll see GLib 2.0 linked in via `-lglib-2.0`, the GNU Scientific Library get linked via `-lgsl`, and so on.

So if the file were named *erf.c*, then the full command line using the gcc compiler, including several additional flags to be discussed shortly, would look like this:

```
gcc erf.c -o erf -lm -g -Wall -O3 -std=gnu11
```

So we've told the compiler to include math functions via an `#include` in the program, and told the linker to link to the math library via the `-lm` on the command line.

The `-o` flag gives the output name; otherwise, we'd get the default executable name of `a.out`.

A Few of My Favorite Flags

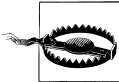
You'll see that I use a few compiler flags every time, and I recommend you do, too.

- `-g` adds symbols for debugging. Without it, your debugger won't be able to give you variable or function names. They don't slow down the program, and we don't care if the program is a kilobyte larger, so there's little reason to not use this. It works for gcc, Clang, and icc (Intel C Compiler).

- `-std=gnu11` is gcc-specific, and specifies that gcc should allow code conforming to the C11 and POSIX standards. Otherwise, gcc will count certain now-valid bits of syntax as invalid. As of this writing, some systems still predate C11, in which case, use `-std=gnu99`. gcc only; everybody else switched to C99 being the default a long time ago. The POSIX standard specifies that `c99` be present on your system, so the compiler-agnostic version of the above line would be:

```
c99 erf.c -o erf -lm -g -Wall -O3
```

In the following makefiles, I achieve this effect by setting the variable `CC=c99`.



On Macs, `c99` is a specially-hacked version of gcc, and is probably not what you want. If you have an undesirable version of `c99` or it is missing entirely, make your own. Put a file named `c99` in the directory at the head of your path with the text:

```
gcc --std=c99 $*
```

or just

```
clang $*
```

as you prefer. Make it executable via `chmod +x c99`.

- `-O3` indicates optimization level three, which tries every trick known to build faster code. If, when you run the debugger, you find that too many variables have been optimized out for you to follow what's going on, then change this to `-O0`. This will be a common tweak in the `CFLAGS` variable, later. This works for gcc, Clang, and icc.
- `-Wall` adds compiler warnings. This works for gcc, Clang, and icc. For icc, you might prefer `-w1`, which displays the compiler's warnings, but not its remarks.



Use your compiler warnings, always. You may be fastidious and know the C standard inside out, but you aren't more fastidious or knowledgeable than your compiler. Old C textbooks filled pages admonishing you to watch out for the difference between `=` and `==`, or to check that all variables are initialized before use. As a more modern textbook author, I have it easy, because I can summarize all those admonishments into one single tip: use your compiler warnings, always.

If your compiler advises a change, don't second-guess it or put off the fix. Do everything necessary to (1) understand why you got a warning and (2) fix your code so that it compiles with zero warnings and zero errors. Compiler messages are famously obtuse, so if you are having trouble with step (1), paste the warning message into your search engine to see how many thousands of others were confounded by this warning before you. You may want to add `-Werror` to your compiler flags so your compiler will treat warnings as errors.

Paths

I've got over 700,000 files on my hard drive, and one of them has the declarations for `sqrt` and `erf`, and another is the object file holding the compiled functions. (You can try `find / -type f | wc -l` to get a rough file count on any POSIX-standard system.) The compiler needs to know in which directories to look to find the correct header and object file, and the problem will only get more complex when we use libraries that are not part of the C standard.

In a typical setup, there are at least three places where libraries may be installed:

- The operating system vendor may define a standard directory or two where libraries are installed by the vendor.
- There may be a directory for the local sysadmin to install packages that shouldn't be overwritten on the next OS upgrade from the vendor. The sysadmin might have a specially hacked version of a library that should override the default version.
- Users typically don't have the rights to write to these locations, and so should be able to use libraries in their home directories.

The OS-standard location typically causes no problems, and the compiler should know to look in those places to find the standard C library, as well as anything installed alongside it. The POSIX standard refers to these directories as “the usual places.”

But for the other stuff, you have to tell the compiler where to look. This is going to get Byzantine: there is no standard way to find libraries in nonstandard locations, and it rates highly on the list of things that frustrate people about C. On the plus side, your compiler knows how to look in the usual locations, and library distributors tend to put things in the usual locations, so you might never need to specify a path manually. On another plus side, there are a few tools to help you with specifying paths. And on one last plus side, once you have located the nonstandard locations on your system, you can set them in a shell or makefile variable and never think about them again.

Let us say that you have a library named `Libuseful` installed on your computer, and you know that its various files were put in the `/usr/local/` directory, which is the location officially intended for your sysadmin's local libraries. You already put `#include <useful.h>` in your code; now you have to put this on the command line:

```
gcc -I/usr/local/include use_useful.c -o use_useful -L/usr/local/lib -luseful
```

- `-I` adds the given path to the include search path, which the compiler searches for header files you `#included` in your code.
- `-L` adds to the library search path.
- Order matters. If you have a file named *specific.o* that depends on the Libbroad library, and Libbroad depends on Libgeneral, then you will need:

```
gcc specific.o -lbroad -lgeneral
```

Any other ordering, such as `gcc -lbroad -lgeneral specific.o`, will probably fail. You can think of the linker looking at the first item, `specific.o`, and writing down a list of unresolved function, structure, and variable names. Then it goes to the next item, `-lbroad`, and searches for the items on its still-missing list, all the while potentially adding new unresolved items, then checking `-lgeneral` for those items still on the missing list. If there are names still unlocated by the end of the list (including that implicit `-lc` at the end), then the linker halts and gives what is left of its missing-items list to the user.

OK, back to the location problem: where is the library that you want to link to? If it was installed via the same package manager that you used to install the rest of your operating system, then it is most likely in the usual places, and you don't have to worry about it.

You may have a sense of where your own local libraries tend to be, such as `/usr/local` or `/sw` or `/opt`. You no doubt have on hand a means of searching the hard drive, such as a search tool on your desktop or the POSIX:

```
find /usr -name 'libuseful*'
```

to search `/usr` for files with names beginning with *libuseful*. When you find Libuseful's shared object file is in `/some/path/lib`, the headers are almost certainly in `/some/path/include`.

Everybody else finds hunting the hard drive for libraries to be annoying, too, and `pkg-config` addresses this by maintaining a repository of the flags and locations that packages self-report as being necessary for compilation. Type `pkg-config` on your command line; if you get an error about specifying package names, then great, you have `pkg-config` and can use it to do the research for you. For example, on my PC, typing these two commands on the command line:

```
pkg-config --libs gsl libxml-2.0
pkg-config --cflags gsl libxml-2.0
```

gives me these two lines of output:

```
-lgsl -lgslcblas -lm -lxml2
-I/usr/include/libxml2
```

These are exactly the flags I need to compile using GSL and LibXML2. The `-l` flags reveal that GNU Scientific Library depends on a Basic Linear Algebra Subprograms (BLAS) library, and the GSL's BLAS library depends on the standard math library. It seems that all the libraries are in the usual places, because there are no `-L` flags, but the `-I` flag indicates the special location for LibXML2's header files.

Back to the command line, the shell provides a trick in that when you surround a command by backticks, the command is replaced with its output. That is, when I type:

```
gcc `pkg-config --cflags --libs gsl libxml-2.0` -o specific specific.c
```

the compiler sees:

```
gcc -I/usr/include/libxml2 -lgs1 -lgs1cblas -lm -lxml2 -o specific specific.c
```

So `pkg-config` does a lot of the work for us, but it is not sufficiently standard that we can expect everybody has it or that every library is registered with it. If you don't have `pkg-config`, then you'll have to do this sort of research yourself, by reading the manual for your library or searching your disk as we saw previously.



There are often environment variables for paths, such as `CPATH` or `LIBRARY_PATH` or `C_INCLUDE_PATH`. You would set them in your `.bashrc` or other such user-specific list of environment variables. They are hopelessly nonstandard—`gcc` on Linux and `gcc` on the Mac use different variables, and any other compiler may use others still. I find that it's easier to set these paths on a per-project basis in the `makefile` or its equivalent, using `-I` and `-L` flags. If you prefer these path variables, check the end of your compiler's manpage for the list of relevant variables for your situation.

Even with `pkg-config`, the need for something that will assemble all this for us is increasingly apparent. Each element is easy enough to understand, but it is a long, mechanical list of tedious parts.

Runtime Linking

Static libraries are linked by the compiler by effectively copying the relevant contents of the library into the final executable. So the program itself works as a more-or-less standalone system. *Shared libraries* are linked to your program at run-time, meaning that we have the same problem with finding the library that we had at compile time all over again at runtime. What is worse, *users* of your program may have this problem.

If the library is in one of the usual locations, life is good and the system will have no problem finding the library at runtime. If your library is in a nonstandard path, then you need to find a way to modify the runtime search path for libraries. Options:

- If you packaged your program with Autotools, Libtool knows how to add the right flags, and you don't have to worry about it.
- The most likely reason for needing to modify this search path is if you are keeping libraries in your home directory because you don't have (or don't want to make use of) root access. If you are installing all of your libraries into *libpath*, then set the environment variable `LD_LIBRARY_PATH`. This is typically done in your shell's startup script (`.bashrc`, `.zshrc`, or whatever is appropriate), via:

```
export LD_LIBRARY_PATH=libpath:$LD_LIBRARY_PATH
```

There are those who warn against overuse of the `LD_LIBRARY_PATH` (what if somebody puts a malicious impostor library in the path, thus replacing the real library without your knowledge?), but if all your libraries are in one place, it is not unreasonable to add one directory under your ostensible control to the path.

- When compiling the program with gcc, Clang, or icc based on a library in *lib-path*, add:

```
LDADD=-Llibpath -Wl,-Rlibpath
```

to the subsequent makefile. The `-L` flag tells the compiler where to search for libraries to resolve symbols; the `-Wl` flag passes its flags through from gcc/Clang/icc to the linker, and the linker embeds the given `-R` into the runtime search path for libraries to link to. Unfortunately, `pkg-config` often doesn't know about runtime paths, so you may need to enter these things manually.

Using Makefiles

The *makefile* provides a resolution to all this endless tweaking. It is basically an organized set of variables and shell scripts. The POSIX-standard *make* program reads the makefile for instructions and variables, and then assembles the long and tedious command lines for us. After this segment, there will be little reason to call the compiler directly.

In “[Makefiles vs. Shell Scripts](#)” on page 62, I'll cover a few more details about the makefile; here, I'm going to show you the smallest practicable makefile that will compile a basic program that depends on a library. Here it is, all six lines of it:

```
P=program_name
OBJECTS=
CFLAGS = -g -Wall -O3
LDLIBS=
CC=c99

$(P): $(OBJECTS)
```

Usage:

- Once ever: Save this (with the name *makefile*) in the same directory as your *.c* files. If you are using GNU Make, you have the option of capitalizing the name to *Makefile* if you feel that doing so will help it to stand out from the other files. Set your program's name on the first line (use *progrname*, not *progrname.c*).
- Every time you need to recompile: Type `make`.



Your Turn: Here's the world-famous *hello.c* program, in two lines:

```
#include <stdio.h>
int main(){ printf("Hello, world.\n"); }
```

Save that and the preceding makefile to a directory, and try the previous steps to get the program compiled and running. Once that works, modify your makefile to compile *erf.c*.

Setting Variables

We'll get to the actual functioning of the makefile soon, but five out of six lines of this makefile are about setting variables (most of which are currently set to be blank), indicating that we should take a moment to consider environment variables in a little more detail.



Historically, there have been two main threads of shell grammar: one based primarily on the Bourne shell, and another based primarily on the C shell. The C shell has a slightly different syntax for variables, e.g., `set CFLAGS="-g -Wall -O3"` to set the value of `CFLAGS`. But the POSIX standard is written around the Bourne-type variable-setting syntax, so that is what I focus on through the rest of this book.

The shell and `make` use the `$` to indicate the value of a variable, but the shell uses `$var`, whereas `make` wants any variable names longer than one character in parens: `$(var)`. So, given the preceding makefile, `$(P): $(OBJECTS)` will be equivalent to

```
program_name:
```

There are several ways to get `make` to recognize a variable:

- Set the variable from the shell before calling `make`, and `export` the variable, meaning that when the shell spawns a child process, it has the variable in its list of environment variables. To set `CFLAGS` from a POSIX-standard command line:

```
export CFLAGS='-g -Wall -O3'
```

At home, I omit the first line in this makefile, `P=program_name`, and instead set it once per session via `export P=program_name`, which means I have to edit the makefile itself still less frequently.

- You can put these `export` commands in your shell's startup script, like `.bashrc` or `.zshrc`. This guarantees that every time you log in or start a new shell, the variable will be set and exported. If you are confident that your `CFLAGS` will be the same every time, you can set them here and never think about them again.
- You can export a variable for a single command by putting the assignment just before the command. The `env` command lists the environment variables it knows about, so when you run the following:

```
PANTS=kakhi env | grep PANTS
```

you should see the appropriate variable and its value. This is why the shell won't let you put spaces around the equals sign: the space is how it distinguishes between the assignment and the command.

Using this form sets and exports the given variables for one line only. After you try this on the command line, try running `env | grep PANTS` again to verify that `PANTS` is no longer an exported variable.

Feel free to specify as many variables as you'd like:

```
PANTS=kakhi PLANTS="figus fern" env | grep 'P.*NTS'
```

This trick is a part of the shell specification's *simple command* description, meaning that the assignment needs to come before an actual command. This will matter when we get to noncommand shell constructs. Writing:

```
VAR=val if [ -e afile ] ; then ./program_using_VAR ; fi
```

will fail with an obscure syntax error. The correct form is:

```
if [ -e afile ] ; then VAR=val ./program_using_VAR ; fi
```

- As in the earlier makefile, you can set the variable at the head of the makefile, with the lines like `CFLAGS=...`. In the makefile, you can have spaces around the equals sign without anything breaking.
- `make` will let you set variables on the command line, independent of the shell. Thus, these two lines are close to equivalent:

```
make CFLAGS="-g -Wall"    Set a makefile variable.
CFLAGS="-g -Wall" make    Set an environment variable that only make and its children see.
```

All of these means are equivalent, as far as your makefile is concerned, with the exception that child programs called by `make` will know new environment variables but won't know any makefile variables.

Environment Variables in C

In your C code, get environment variables with `getenv`. Because `getenv` is so easy to use, it's useful for quickly setting a variable on the C side, so you can try a few different values from the command prompt.

[Example 1-2](#) prints a message to the screen as often as the user desires. The message is set via the environment variable `msg` and the number of repetitions via `reps`. Notice how we set defaults of 10 and "Hello." should `getenv` return `NULL` (typically meaning that the environment variable is unset).

Example 1-2. Environment variables provide a quick way to tweak details of a program (getenv.c)

```
#include <stdlib.h> //getenv, atoi
#include <stdio.h> //printf

int main(){
    char *repstext=getenv("reps");
    int reps = repstext ? atoi(repstext) : 10;

    char *msg = getenv("msg");
    if (!msg) msg = "Hello.";

    for (int i=0; i< reps; i++)
        printf("%s\n", msg);
}
```


As previously, we can export a variable for just one line, which makes sending a variable to the program still more convenient. Usage:

```
reps=10 msg="Ha" ./getenv  
msg="Ha" ./getenv  
reps=20 msg=" " ./getenv
```

You might find this to be odd—the inputs to a program should come *after* the program name, darn it—but the oddness aside, you can see that it took little setup within the program itself, and we get to have named parameters on the command line almost for free.

When your program is a little further along, you can take the time to set up `getopt` to set input arguments the usual way.

`make` also offers several built-in variables. Here are the (POSIX-standard) ones that you will need to read the following rules:

`$@`

The full target filename. By *target*, I mean the file that needs to be built, such as a `.o` file being compiled from a `.c` file or a program made by linking `.o` files.

`$*`

The target file with the suffix cut off. So if the target is *prog.o*, `$*` is *prog*, and `$*.c` would become *prog.c*.

`$<`

The name of the file that caused this target to get triggered and made. If we are making *prog.o*, it is probably because *prog.c* has recently been modified, so `$<` is *prog.c*.

The Rules

Now, let us focus on the procedures the makefile will execute, and then get to how the variables influence that.

Setting the variables aside, segments of the makefile have the form:

```
target: dependencies  
      script
```

If the target gets called, via the command `make target`, then the dependencies are checked. If the target is a file, the dependencies are all files, and the target is newer than the dependencies, then the file is up-to-date and there's nothing to do. Otherwise, the processing of the target gets put on hold, the dependencies are run or generated, probably via another target, and when the dependency scripts are all finished, the target's script gets executed.

For example, before this was a book, it was a series of tips posted to a blog (at <http://modelingwithdata.org>). Every blog post had an HTML and PDF version, all generated

via LaTeX. I'm omitting a lot of details for the sake of a simple example (like the many options for `latex2html`), but here's the sort of makefile one could write for the process.



If you are copying any of these makefile snippets from a version on your screen or on paper to a file named *makefile*, don't forget that the white-space at the head of each line must be a tab, not spaces. Blame POSIX.

```
all: html doc publish

doc:
    pdflatex $(f).tex

html:
    latex -interaction batchmode $(f)
    latex2html $(f).tex

publish:
    scp $(f).pdf $(Blogserver)
```

I set `f` on the command line via a command like `export f=tip-make`. When I then type `make` on the command line, the first target, `all`, gets checked. That is, the command `make` by itself is equivalent to `make first_target`. That depends on `html`, `doc`, and `publish`, so those targets get called in sequence. If I know it's not yet ready to copy out to the world, then I can call `make html doc` and do only those steps.

In the simple makefile from earlier, we had only one target/dependency/script group. For example:

```
P=domath
OBJECTS=addition.o subtraction.o

$(P): $(OBJECTS)
```

This follows a sequence of dependencies and scripts similar to what my blogging makefile did, but the scripts are implicit. Here, `P=domath` is the program to be compiled, and it depends on the object files `addition.o` and `subtraction.o`. Because `addition.o` is not listed as a target, `make` uses an implicit rule, listed below, to compile from the `.c` to the `.o` file. Then it does the same for `subtraction.o` and `domath.o` (because GNU `make` implicitly assumes that `domath` depends on `domath.o` given the setup here). Once all the objects are built, we have no script to build the `$(P)` target, so GNU `make` fills in its default script for linking `.o` files into an executable.

POSIX-standard `make` has a specific recipe for compiling a `.o` object file from a `.c` source code file:

```
$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $*.c
```

The `$(CC)` variable represents your C compiler; The POSIX standard specifies a default of `CC=c99`, but current editions of GNU `make` set `CC=cc`, which is typically a link to `gcc`. In the minimal makefile at the head of this segment, `$(CC)` is explicitly set to `c99`, `$(CFLAGS)` is set to the list of flags earlier, and `$(LDFLAGS)` is unset and therefore replaced

with nothing. So if `make` determines that it needs to produce *your_program.o*, then this is the command that will be run, given that makefile:

```
c99 -g -Wall -O3 -o your_program.o your_program.c
```

When GNU `make` decides that you have an executable program to build from object files, it uses this recipe:

```
$(CC) $(LDFLAGS) first.o second.o $(LDLIBS)
```

Recall that order matters in the linker, so we will need two linker variables. In the previous example, we needed:

```
cc specific.o -lbroad -lgeneral
```

as the relevant part of the linking command. Comparing the correct compilation command to the recipe, we see that we need to set `LDLIBS=-lbroad -lgeneral`. If we had set `LDFLAGS=-lbroad -lgeneral`, then the recipe would produce `cc -lbroad -lgeneral specific.o`, which is likely to fail. Notice that `LDFLAGS` also appears in the recipe for compilation from `.c` to `.o` files.



If you'd like to see the full list of default rules and variables built in to your edition of `make`, try:

```
make -p > default_rules
```

So, that's the game: find the right variables and set them in the makefile. You still have to do the research as to what the correct flags are, but at least you can write them down in the makefile and never think about them again.

If you use an IDE, or CMAKE, or any of the other alternatives to POSIX-standard `make`, you're going to be playing the same find-the-variables game. I'm going to continue discussing the preceding minimal makefile, and you should have no problem finding the corresponding variables in your IDE.

- The `CFLAGS` variable is an ingrained custom, but the variable that you'll need to set for the linker changes from system to system. Even `LDLIBS` isn't POSIX-standard, but is what GNU `make` uses.
- The `CFLAGS` and `LDLIBS` variables are where we're going to hook all the compiler flags locating and identifying libraries. If you have `pkg-config`, put the backticked calls here. For example, the makefile on my system, where I use Apophenia and GLib for just about everything, looks like:

```
CFLAGS=`pkg-config --cflags apophenia glib-2.0` -g -Wall -std=gnu11 -O3  
LDLIBS=`pkg-config --libs apophenia glib-2.0`
```

Or, specify the `-I`, `-L`, and `-l` flags manually, like:

```
CFLAGS=-I/home/b/root/include -g -Wall -O3  
LDLIBS=-L/home/b/root/lib -lweirdlib
```

- After you add a library and its locations to the `LIBS` and `CFLAGS` lines and you know it works on your system, there is little reason to ever remove it. Do you really care that the final executable might be 10 kilobytes larger than if you customized a new makefile for every program? That means you can write one makefile summarizing where all the libraries are on your system and copy it from project to project without any rewriting.
- If you have a second (or more) C file, add `second.o third.o`, and so on to the `OBJECTS` line (no commas, just spaces between names) in the makefile at the head of this section. `make` will use that to determine which files to build and which recipes to run.
- If you have a program that is one `.c` file, you may not need a makefile at all. In a directory with no makefile and `erf.c` from earlier, try using your shell to:

```
export CFLAGS='-g -Wall -O3 -std=gnu11'
export LDLIBS='-lm'
make erf
```

and watch `make` use its knowledge of C compilation to do the rest.

What Are the Linker Flags for Building a Shared Library?

To tell you the truth, I have no idea. It's different across operating systems, both by type and by year, and even on one system the rules are often hairy.

Instead, *Libtool*, one of the tools introduced in [Chapter 3](#), knows every detail of every shared library generation procedure on every operating system. I recommend investing your time getting to know Autotools and thus solve the shared object compilation problem once and for all, rather than investing that time in learning the right compiler flags and linking procedure for every target system.

Using Libraries from Source

So far, the story has been about compiling your own code using `make`. Compiling code provided by others is often a different story.

Let's try a sample package. The GNU Scientific Library includes a host of numeric computation routines.

The GSL is packaged via *Autotools*, a set of tools that will prepare a library for use on any machine, by testing for every known quirk and implementing the appropriate workaround. Autotools is central to how code is distributed in the present day, and [“Packaging Your Code with Autotools” on page 64](#) will go into detail about how you can package your own programs and libraries with it. But for now, we can start off as users of the system and enjoy the ease of quickly installing useful libraries.

The GSL is often provided in precompiled form via package manager, but for the purposes of going through the steps of compilation, here's how to get the GSL as source code and set it up, assuming you have root privileges on your computer.

```
wget ftp://ftp.gnu.org/gnu/gsl/gsl-1.15.tar.gz ❶
tar xvzf gsl-*.gz ❷
cd gsl-1.15
./configure ❸
make
sudo make install ❹
```

- ❶ Download the zipped archive. Ask your package manager to install `wget` if you don't have it, or type this URL into your browser.
- ❷ Unzip the archive: `x=extract`, `v=verbose`, `z=unzip` via `gzip`, `f=filename`.
- ❸ Determine the quirks of your machine. If the `configure` step gives you an error about a missing element, then use your package manager to obtain it and run `configure` again.
- ❹ Install to the right location—if you have permissions.

If you are trying this at home, then you probably have root privileges, and this will work fine. If you are at work and using a shared server, the odds are low that you have superuser rights, so you won't be able to provide the password needed to do the last step in the script as superuser. In that case, hold your breath until the next section.

Did it install? [Example 1-3](#) provides a short program to try finding that 95% confidence interval using GSL functions; try it and see if you can get it linked and running:

Example 1-3. Redoing [Example 1-1](#) with the GSL (`gsl_erf.c`)

```
#include <gsl/gsl_cdf.h>
#include <stdio.h>

int main(){
    double bottom_tail = gsl_cdf_gaussian_P(-1.96, 1);
    printf("Area between [-1.96, 1.96]: %g\n", 1-2*bottom_tail);
}
```

To use the library you just installed, you'll need to modify the makefile of your library-using program to specify the libraries and their locations.

Depending on whether you have `pkg-config` on hand, you can do one of:

```
LDLIBS=`pkg-config --libs gsl`
#or
LDLIBS=-lgsl -lgslcblas -lm
```

If it didn't install in a standard location and `pkg-config` is not available, you will need to add paths to the heads of these definitions, such as `CFLAGS=-I/usr/local/include` and `LDLIBS=-L/usr/local/lib -Wl,-R/usr/local/lib`.

Using Libraries from Source (Even if Your Sysadmin Doesn't Want You To)

You may have noticed the caveats in the last section about how you have to have root privileges to install to the usual locations on a POSIX system. But you may not have root access if you are using a shared computer at work, or if you have an especially controlling significant other.

Then you have to go underground and make your own private root directory.

The first step is to simply create the directory:

```
mkdir ~/root
```

I already have a `~/tech` directory where I keep all my technical logistics, manuals, and code snippets, so I made a `~/tech/root` directory. The name doesn't matter, but I'll use `~/root` as the dummy directory here.



Your shell replaces the tilde with the full path to your home directory, saving you a lot of typing. The POSIX standard only requires that the shell do this at the beginning of a word or just after a colon (which you'd need for a path-type variable), but most shells expand mid-word tildes as well. Other programs, like `make`, may or may not recognize the tilde as your home directory.

The second step is to add the right part of your new root system to all the relevant paths. For programs, that's the `PATH` in your `.bashrc` (or equivalent):

```
PATH=~/.root/bin:$PATH
```

By putting the `bin` subdirectory of your new directory before the original `PATH`, it will be searched first, and your copy of any programs will be found first. Thus, you can substitute in your preferred version of any programs that are already in the standard shared directories of the system.

The Manual

I suppose there was once a time when the manual was actually a printed document, but in the present day, it exists in the form of the `man` command. For example, use `man strtok` to read about the `strtok` function, typically including what header to include, the input arguments, and basic notes about its usage. The manual pages tend to keep it simple, sometimes lack examples, and assume the reader already has a basic idea of how the function works. If you need a more basic tutorial, your favorite Internet search engine can probably offer several (and in the case of `strtok`, see the section [“A Pæan to strtok” on page 154](#)). The GNU C library manual, also easy to find online, is very readable and written for beginners.

- If you can't recall the name of what you need to look up, every manual page has a one-line summary, and `man -k searchterm` will search those summaries. Many systems also have the `apropos` command, which is similar to `man -k` but adds some features. For extra refinement, I often find myself piping the output of `apropos` through `grep`.
- The manual is divided into sections. Section 1 is command-line commands, and section 3 is library functions. If your system has a command-line program named `printf`, then `man printf` will show its documentation, and `man 3 printf` will show the documentation for the C library's `printf` command.
- For more on the usage of the `man` command (such as the full list of sections), try `man man`.
- Your text editor or IDE may have a means of calling up manpages quickly. For example, vi users can put the cursor on a word and use `K` to open that word's manpage.

For libraries you will fold into your C programs, note the new paths to search in the preceding makefile:

```
LDLIBS=-L/home/your_home/root/lib (plus the other flags, like -lgl -lm ...)
CFLAGS=-I/home/your_home/root/include (plus -g -Wall -O3 ...)
```

Now that you have a local root, you can use it for other systems as well, such as Java's `CLASSPATH`.

The last step is to install programs in your new root. If you have the source code and it uses Autotools, all you have to do is add `--prefix=$HOME/root` in the right place:

```
./configure --prefix=$HOME/root; make; make install
```

You didn't need `sudo` to do the install step, because everything is now in territory you control.

Because the programs and libraries are in your home directory and have no more permissions than you do, your sysadmin can't complain that they are an imposition on others. If your sysadmin complains anyway, then, as sad as it may be, it might be time to break up.

Compiling C Programs via Here Document

At this point, you have seen the pattern of compilation a few times:

1. Set a variable expressing compiler flags.
2. Set a variable expressing linker flags, including a `-l` flag for every library that you use.
3. Use `make` or your IDE's recipes to convert the variables into full compile and link commands.

The remainder of this chapter will do all this one last time, using an absolutely minimal setup: just the shell. If you are a kinetic learner who picked up scripting languages by cutting and pasting snippets of code into the interpreter, you'll be able to do the same with pasting C code onto your command prompt.

Include Header Files from the Command Line

The gcc and Clang have a convenient flag for including headers. For example:

```
gcc -include stdio.h
```

is equivalent to putting

```
#include <stdio.h>
```

at the head of your C file; similarly for `clang -include stdio.h`.

By adding that to our compiler invocation, we can finally write *hello.c* as the one line of code it should be:

```
int main(){ printf("Hello, world.\n"); }
```

which compiles fine via:

```
gcc -include stdio.h hello.c -o hi --std=gnu99 -Wall -g -O3
```

or shell commands like:

```
export CFLAGS='-g -Wall -include stdio.h'
export CC=c99
make hello
```

This tip about `-include` is compiler-specific and involves moving information from the code to the compilation instructions. If you think this is bad form, well, skip this tip.

The Unified Header

There was once a time when compilers took several seconds or minutes to compile even relatively simple programs, so there was human-noticeable benefit to reducing the work the compiler has to do. My current copies of *stdio.h* and *stdlib.h* are each about 1,000 lines long (try `wc -l /usr/include/stdlib.h`) and *time.h* another 400, meaning that this seven-line program:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    srand(time(NULL)); // Initialize RNG seed.
    printf("%i\n", rand()); // Make one draw.
}
```

is actually a ~2,400-line program.

Your compiler doesn't think 2,400 lines is a big deal anymore, and this compiles in under a second. So why are we spending time picking out just the right headers for a given program?

Once you have a unified header, even a line like `#include <allheads.h>` is extraneous if you are a gcc or Clang user, because you can instead add `-include allheads.h` to your CFLAGS and never think about which out-of-project headers to include again.



Your Turn: Write yourself a single header, let us call it *allheads.h*, and throw in every header you've ever used, so it'll look something like:

```
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <gsl/gsl_rng.h>
```

I can't tell you exactly what it'll look like, because I don't know exactly what you use day to day.

Now that you have this aggregate header, you can just throw one:

```
#include <allheads.h>
```

on top of every file you write, and you're done with thinking about headers. Sure, it will expand to perhaps 10,000 lines of extra code, much of it not relevant to the program at hand. But you won't notice, and unused declarations don't change the final executable.

Headers also serve the purpose of limiting scope, but this is generally more important for functions and structures you wrote than those in the libraries. The purpose of limiting scope is not to keep the namespace small so the computer won't overheat; it is to reduce cognitive load for you, the programmer. I'm guessing you're not even familiar with most of the functions to be found in the libraries you are using, and if you don't know about them, they can't possibly be taking up cognitive load. Other languages don't even make the distinction and heap on the keywords, like the R project, which has 752 words internally defined at startup.

Here Documents

Here documents are a feature of POSIX-standard shells that you can use for C, Python, Perl, or whatever else, and they will make this book much more useful and fun. Also, if you want to have a multilingual script, here documents are an easy way to do it. Do some parsing in Perl, do the math in C, then have Gnuplot produce the pretty pictures, and have it all in one text file.

Here's a Python example. Normally, you'd tell Python to run a script via:

```
python your_script.py
```

You can give the filename '-' to use stdin as the input file:

```
echo "print 'hi.'" | python '-'
```



We need '-' and not just - to indicate that this is plain text and not introducing a switch like the c in `python -c "print 'Hi'"`. Many programs follow the GNU custom that two dashes indicate that they should stop reading switches and read subsequent inputs plain. Thus:

```
echo "print 'hi.'" | python -- -
```

also works, but is the sort of thing that scares people.

You could, in theory, put some lengthy scripts on the command line via `echo`, but you'll quickly see that there are a lot of small, undesired parsings going on—you might need `"hi\"` instead of `"hi"`, for example.

Thus, the *here document*, which does no parsing at all. Try this:

```
python '-' <<"XXXX"
lines=2
print "\nThis script is %i lines long.\n" %(lines,)
XXXX
```

- Here documents are a standard shell feature, so they should work on any POSIX system.
- The "XXXX" is any string you'd like; "EOF" is also popular, and "-----" looks good as long as you get the dash count to match at top and bottom. When the shell sees your chosen string alone on a line, it will stop sending the script to the program's stdin. That's all the parsing that happens.
- There's also a variant that begins with <<-. This variant removes all tabs at the head of every line, so you can put a here document in an indented section of a shell script without breaking the flow of indentation. Of course, this would be disastrous for a Python here document.
- As another variant, there's a difference between <<"XXXX" and <<XXXX. In the second version, the shell parses certain elements, which means you can have the shell insert the value of `$shell_variables` for you. The shell relies heavily on the \$ for its variables and other expansions; the \$ is one of the few characters on a standard keyboard that has no special meaning to C. It's as if the people who wrote Unix designed it from the ground up to make it easy to write shell scripts that produce C code....

Compiling from stdin

OK, back to C: we can use here documents to compile C code pasted onto the command line via `gcc` or `Clang`, or have a few lines of C in a multilingual script.

We're not going to use the makefile, so we need a single compilation command. To make life less painful, let us alias it. Paste this onto your command line, or add it to your `.bashrc`, `.zshrc`, or wherever applicable:

```
go_libs="-lm"
go_flags="-g -Wall -include allheads.h -O3"
alias go_c="c99 -xc '-' $go_libs $go_flags"
```

where `allheads.h` is the aggregate header you'd put together earlier. Using the `-include` flag means one less thing to think about when writing the C code, and I've found that bash's history gets wonky when there are `#`s in the C code.

On the compilation line, you'll recognize the `'-'` to mean that instead of reading from a named file, use stdin. The `-xc` identifies this as C code, because `gcc` stands for GNU Compiler Collection, not GNU C Compiler, and with no input filename ending in `.c` to tip it off, we have to be clear that this is not Java, Fortran, Objective C, Ada, or C++ (and similarly for Clang, even though its name is meant to invoke *C language*).

Whatever you did to customize the `LDLIBS` and `CFLAGS` in your makefile, do here.

Now we're sailing, and can compile C code on the command line:

```
go_c << '---'
int main(){printf("Hello from the command line.\n");}
---
./a.out
```

We can use a here document to paste short C programs onto the command line, and write little test programs without hassle. Not only do you not need a makefile, you don't even need an input file.

Don't expect this sort of thing to be your primary mode of working. But cutting and pasting code snippets onto the command line can be fun, and being able to have a single step in C within a longer shell script is pretty fabulous.

O'Reilly Ebooks—Your bookshelf on your devices!



When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in five, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, Android .apk, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O'Reilly ebooks through the iBookstore, the [Android Marketplace](http://AndroidMarketplace.com), and Amazon.com.

O'REILLY®

Spreading the knowledge of innovators

oreilly.com