Tip 75: Compile C for Windows

Ben Klemens

29 February 2012

level: Possibly very basic
purpose: Make up for Microsoft

On most systems, C is the central language that everything works around; on a Windows box C is strangely ignored.

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 Because C and UNIX co-evolved, 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 you wrote elsewhere on a Windows box will find this to be the most natural route.

For those of you who just got here, POSIX is a standardization of UNIX-like systems, and it includes a set of C functions (like `fork` and `popen`), the specs for the standard shell, and the sort of utilities that made you a Windows user, like `grep`, `sed`, `vi`, et cetera. Of course, any POSIX-compliant system provides a lot more than the basic standard, and you'll often see the POSIX set of features added on as a side-door to a system that you'd otherwise never think of as POSIXy.

As far as I can tell, the world divides into two classes:

- POSIX-compliant systems, and
- the Windows family of operating systems.

That first class includes everything from Apple, Tivos, the Nintendo Wii, Amazon's Kindle, and probably your telephone. [It's a fun game to think of systems that are not at some level POSIX. Hint: if it uses a Linux kernel, then somebody somewhere has a POSIX toolchain that targets the device, though they may be hiding it from you.] There had in the past existed POSIX add-ons for Windows operating systems, like Interix, but I'm not aware of anything current.

And so, Cygwin.

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

1. Write a library for Windows that provides all the POSIX function calls. This will have to smooth over some Windows/POSIX incongruities, like how Windows

has distinct drives like `C:` while POSIX has one unified filesystem. In this case, mount `C:` as `/cygdrive/c`, `D:` as `/cygdrive/d`, et cetera.

2. Now that you can compile POSIX-compliant programs by linking to your library, do so: generate Windows versions of `ls`, `bash`, `grep`, `make`, `gcc`, `X`, `rxvt`, `libglib`, `perl`, `python`, et cetera.

3. Write 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 web site[1] and pick packages. You will certainly want the compiler and 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.

C plus POSIX Microsoft provides a C++ compiler, in the form of Visual Studio. That compiler has an ANSI C compatibility mode, which is the only C compiler currently provided by Microsoft. Many representatives from the company[2] 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 in ANSI C mode, 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.

This would be a good time to remind you of the discussion of Autotools (Entry #110). Few of us have the time to sit around fretting about the little incompatibilities between POSIX via Cygwin and POSIX via everything else, and those few who do are contributors to Autotools. If you find compiling things under Windows to be more glitch-filled than you'd expected, the perhaps set up your code with Autotools and see if any of those glitches get auto-handled for you.

If you are compiling under Cygwin, then your program will depend on that library of POSIX function calls, `cygwin1.dll` (whether your code actually includes any POSIX call 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 is copyrighted material, so if you distribute it, then you have to conform to Cygwin's copyright conditions, which require that your program be distributed under an open source license. 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 below. You can use `cygcheck` to find out which DLLs your program depends upon, and thus verify that your executable does or does not link to `cygwin1.dll`.]

[1]`http://cygwin.com`

[2]`http://blogs.msdn.com/b/vcblog/archive/2007/11/05/iso-c-standard-update.aspx`

C minus 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 provides other useful tools, like a shell.

The lack of POSIX-style amenities is not the real problem with MinGW. If you want a shell, Msys provides one. Or leave the command prompt behind entirely and try Code::blocks[3], 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 and configuration.

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 GCC. [There is also a MinGW64 which has increasingly good odds of being appropriate for your box.]

From Msys, packages built via `./configure; make; make install` have a fighting chance of working. Or if you have downloaded the packages to build from Cygwin's command prompt, then you can use

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

to get Cygwin to use the Mingw32 compiler to produce POSIX-free code.

Once you've compiled under MinGW, you've got a native Windows binary. Since 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.

No, the real problem with MinGW is the lack of a package manager, which means that there is no direct way to get pre-compiled libraries. 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–and 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 libraries that you'll need to write decent code.

[3]`http://www.codeblocks.org/`