
Portable Makefiles

What do we mean by a portable *makefile*? As an extreme example, we want a *makefile* that runs without change on any system that GNU *make* runs on. But this is virtually impossible due to the enormous variety in operating systems. A more reasonable interpretation is a *makefile* that is easy to change for each new platform it is run on. An important added constraint is that the port to the new system does not break support for the previous platforms.

We can achieve this level of portability for *makefiles* using the same techniques as traditional programming: encapsulation and abstraction. By using variables and user-defined functions we can encapsulate applications and algorithms. By defining variables for command-line arguments and parameters, we can abstract out elements that vary from platform to platform from elements that are constant.

You then have to determine what tools each platform offers to get your job done, and what to use from each platform. The extreme in portability is to use only those tools and features that exist on all platforms of interest. This is typically called the “least common denominator” approach and obviously can leave you with very primitive functionality to work with.

Another version of the least common denominator approach is to choose a powerful set of tools and make sure to bring it with you to every platform, thus guaranteeing that the commands you invoke in the *makefile* work exactly the same everywhere. This can be hard to pull off, both administratively and in terms of getting your organization to cooperate with your fiddling with their systems. But it can be successful, and I’ll show one example of that later with the Cygwin package for Windows. As you’ll see, standardizing on tools does not solve every problem; there are always operating system differences to deal with.

Finally, you can accept differences between systems and work around them by careful choices of macros and functions. I’ll show this approach in this chapter, too.

So, by judicious use of variables and user-defined functions, and by minimizing the use of exotic features and relying on standard tools, we can increase the portability of

our *makefiles*. As noted previously, there is no such thing as perfect portability, so it is our job to balance effort versus portability. But before we explore specific techniques, let's review some of the issues of portable *makefiles*.

Portability Issues

Portability problems can be difficult to characterize since they span the entire spectrum from a total paradigm shift (such as traditional Mac OS versus System V Unix) to almost trivial bug fixes (such as a fix to a bug in the error exit status of a program). Nevertheless, here are some common portability problems that every *makefile* must deal with sooner or later:

Program names

It is quite common for various platforms to use different names for the same or similar programs. The most common is the name of the C or C++ compiler (e.g., `cc`, `xlc`). It is also common for GNU versions of programs to be installed on a non-GNU system with the `g` prefix (e.g., `gmake`, `gawk`).

Paths

The location of programs and files often varies between platforms. For instance, on Solaris systems the X directories are stored under `/usr/X` while on many other systems the path is `/usr/X11R6`. In addition, the distinction between `/bin`, `/usr/bin`, `/sbin`, and `/usr/sbin` is often rather fuzzy as you move from one system to another.

Options

The command-line options to programs vary, particularly when an alternate implementation is used. Furthermore, if a platform is missing a utility or comes with a broken version, you may need to replace the utility with another that uses different command-line options.

Shell features

By default, `make` executes command scripts with `/bin/sh`, but `sh` implementations vary widely in their features. In particular, pre-POSIX shells are missing many features and will not accept the same syntax as a modern shell.

The Open Group has a very useful white paper on the differences between the System V shell and the POSIX shell. It can be found at <http://www.unix-systems.org/whitepapers/shdiffs.html>. For those who want more details, the specification of the POSIX shell's command language can be found at http://www.opengroup.org/onlinepubs/007904975/utilities/xcu_chap02.html.

Program behavior

Portable *makefiles* must contend with programs that simply behave differently. This is very common as different vendors fix or insert bugs and add features. There are also upgrades to utilities that may or may not have made it into a vendor's release. For instance, in 1987 the `awk` program underwent a major revision.

Nearly 20 years later, some systems still do not install this upgraded version as the standard `awk`.

Operating system

Finally, there are the portability problems associated with a completely different operating system such as Windows versus Unix or Linux versus VMS.

Cygwin

Although there is a native Win32 port of `make`, this is a small part of the Windows portability problem, because the shell this native port uses is `cmd.exe` (or `command.exe`). This, along with the absence of most of the Unix tools, makes cross-platform portability a daunting task. Fortunately, the Cygwin project (<http://www.cygwin.com>) has built a Linux-compatible library for Windows to which many programs* have been ported. For Windows developers who want Linux compatibility or access to GNU tools, I don't believe there is a better tool to be found.

I have used Cygwin for over 10 years on a variety of projects from a combined C++/Lisp CAD application to a pure Java workflow management system. The Cygwin tool set includes compilers and interpreters for many programming languages. However, Cygwin can be used profitably even when the applications themselves are implemented using non-Cygwin compilers and interpreters. The Cygwin tool set can be used solely as an aid to coordinating the development and build process. In other words, it is not necessary to write a “Cygwin” application or use Cygwin language tools to reap the benefits of the Cygwin environment.

Nevertheless, Linux is not Windows (thank goodness!) and there are issues involved when applying Cygwin tools to native Windows applications. Almost all of these issues revolve around the line endings used in files and the form of paths passed between Cygwin and Windows.

Line Termination

Windows filesystems use a two-character sequence carriage return followed by line feed (or CRLF) to terminate each line of a text file. POSIX systems use a single character, a line feed (LF or *newline*). Occasionally this difference can cause the unwary some confusion as programs report syntax errors or seek to the wrong location in a data file. However, the Cygwin library does a very good job of working through these issues. When Cygwin is installed (or alternatively when the `mount` command is used), you can choose whether Cygwin should translate files with CRLF endings. If a DOS file format is selected, Cygwin will translate CRLF to LF when reading and the reverse when writing text files so that Unix-based programs can properly handle

* My Cygwin `/bin` directory currently contains 1343 executables.

DOS text files. If you plan to use native language tools such as Visual C++ or Sun's Java SDK, choose the DOS file format. If you are going to use Cygwin compilers, choose Unix. (Your choice can be changed at any time.)

In addition, Cygwin comes with tools to translate files explicitly. The utilities `dos2unix` and `unix2dos` transform the line endings of a file, if necessary.

Filesystem

Cygwin provides a POSIX view of the Windows filesystem. The root directory of a POSIX filesystem is `/`, which maps to the directory in which Cygwin is installed. Windows drives are accessible through the pseudo-directory `/cygdrive/letter`. So, if Cygwin is installed in `C:\usr\cygwin` (my preferred location), the directory mappings shown in Table 7-1 would hold.

Table 7-1. Default Cygwin directory mapping

Native Windows path	Cygwin path	Alternate Cygwin path
<code>c:\usr\cygwin</code>	<code>/</code>	<code>/cygdrive/c/usr/cygwin</code>
<code>c:\Program Files</code>	<code>/cygdrive/c/Program Files</code>	
<code>c:\usr\cygwin\bin</code>	<code>/bin</code>	<code>/cygdrive/c/usr/cygwin/bin</code>

This can be a little confusing at first, but doesn't pose any problems to tools. Cygwin also includes a `mount` command that allows users to access files and directories more conveniently. One option to `mount`, `--change-cygdrive-prefix`, allows you to change the prefix. I find that changing the prefix to simply `/` is particularly useful because drive letters can be accessed more naturally:

```
$ mount --change-cygdrive-prefix /
$ ls /c
AUTOEXEC.BAT      Home           Program Files   hp
BOOT.INI          I386          RECYCLER        ntldr
CD                IO.SYS        System Volume Information pagefile.sys
CONFIG.SYS        MSDOS.SYS     Temp            tmp
C_DILLA          NTDETECT.COM WINDOWS         usr
Documents and Settings PERSIST      WUTemp          work
```

Once this change is made, our previous directory mapping would change to those shown in Table 7-2.

Table 7-2. Modified Cygwin directory mapping

Native Windows path	Cygwin path	Alternate Cygwin path
<code>c:\usr\cygwin</code>	<code>/</code>	<code>/c/usr/cygwin</code>
<code>c:\Program Files</code>	<code>/c/Program Files</code>	
<code>c:\usr\cygwin\bin</code>	<code>/bin</code>	<code>/c/usr/cygwin/bin</code>

If you need to pass a filename to a Windows program, such as the Visual C++ compiler, you can usually just pass the relative path to the file using POSIX-style forward slashes. The Win32 API does not distinguish between forward and backward slashes. Unfortunately, some utilities that perform their own command-line argument parsing treat all forward slashes as command options. One such utility is the DOS `print` command; another is the `net` command.

If absolute paths are used, the drive letter syntax is always a problem. Although Windows programs are usually happy with forward slashes, they are completely unable to fathom the `/c` syntax. The drive letter must always be transformed back into `c:`. To accomplish this and the forward/backslash conversion, Cygwin provides the `cygpath` utility to translate between POSIX paths and Windows paths.

```
$ cygpath --windows /c/work/src/lib/foo.c
c:\work\src\lib\foo.c
$ cygpath --mixed /c/work/src/lib/foo.c
c:/work/src/lib/foo.c
$ cygpath --mixed --path "/c/work/src:/c/work/include"
c:/work/src;c:/work/include
```

The `--windows` option translates the POSIX path given on the command line into a Windows path (or vice versa with the proper argument). I prefer to use the `--mixed` option that produces a Windows path, but with forward slashes instead of backslashes (when the Windows utility accepts it). This plays much better with the Cygwin shell because the backslash is the escape character. The `cygpath` utility has many options, some of which provide portable access to important Windows paths:

```
$ cygpath --desktop
/c/Documents and Settings/Owner/Desktop
$ cygpath --homeroot
/c/Documents and Settings
$ cygpath --smprograms
/c/Documents and Settings/Owner/Start Menu/Programs
$ cygpath --sysdir
/c/WINDOWS/SYSTEM32
$ cygpath --windir
/c/WINDOWS
```

If you're using `cygpath` in a mixed Windows/Unix environment, you'll want to wrap these calls in a portable function:

```
ifdef COMSPEC
  cygpath-mixed      = $(shell cygpath -m "$1")
  cygpath-unix      = $(shell cygpath -u "$1")
  drive-letter-to-slash = /$(subst :,,$1)
else
  cygpath-mixed      = $1
  cygpath-unix      = $1
  drive-letter-to-slash = $1
endif
```

If all you need to do is map the *c:* drive letter syntax to the POSIX form, the drive-letter-to-slash function is faster than running the `cygpath` program.

Finally, Cygwin cannot hide all the quirks of Windows. Filenames that are invalid in Windows are also invalid in Cygwin. Thus, names such as *aux.h*, *com1*, and *prn* cannot be used in a POSIX path, even with an extension.

Program Conflicts

Several Windows programs have the same names as Unix programs. Of course, the Windows programs do not accept the same command-line arguments or behave in compatible ways with the Unix programs. If you accidentally invoke the Windows versions, the usual result is serious confusion. The most troublesome ones seem to be `find`, `sort`, `ftp`, and `telnet`. For maximum portability, you should be sure to provide full paths to these programs when porting between Unix, Windows, and Cygwin.

If your commitment to Cygwin is strong and you do not need to build using native Windows support tools, you can safely place the Cygwin `/bin` directory at the front of your Windows path. This will guarantee access to Cygwin tools over Windows versions.

If your *makefile* is working with Java tools, be aware that Cygwin includes the GNU `jar` program that is incompatible with the standard Sun *jar* file format. Therefore, the Java `jdk bin` directory should be placed before the Cygwin `/bin` directory in your `Path` variable to avoid using Cygwin's `jar` program.

Managing Programs and Files

The most common way to manage programs is to use a variable for program names or paths that are likely to change. The variables can be defined in a simple block, as we have seen:

```
MV ?= mv -f
RM ?= rm -f
```

or in a conditional block:

```
ifdef COMSPEC
  MV ?= move
  RM ?= del
else
  MV ?= mv -f
  RM ?= rm -f
endif
```

If a simple block is used, the values can be changed by resetting them on the command line, by editing the *makefile*, or (in this case because we used conditional assignment, `?=`) by setting an environment variable. As mentioned previously, one

way to test for a Windows platform is to check for the COMSPEC variable, which is used by all Windows operating systems. Sometimes only a path needs to change:

```
ifdef COMSPEC
    OUTPUT_ROOT := d:
    GCC_HOME     := c:/gnu/usr/bin
else
    OUTPUT_ROOT := $(HOME)
    GCC_HOME     := /usr/bin
endif

OUTPUT_DIR := $(OUTPUT_ROOT)/work/binaries
CC := $(GCC_HOME)/gcc
```

This style results in a *makefile* in which most programs are invoked via make variables. Until you get used to it, this can make the *makefile* a little harder to read. However, variables are often more convenient to use in the *makefile* anyway, because they can be considerably shorter than the literal program name, particularly when full paths are used.

The same technique can be used to manage different command options. For instance, the built-in compilation rules include a variable, TARGET_ARCH, that can be used to set platform-specific flags:

```
ifeq "$(MACHINE)" "hpux-hppa"
    TARGET_ARCH := -mdisable-fpregs
endif
```

When defining your own program variables, you may need to use a similar approach:

```
MV := mv $(MV_FLAGS)

ifeq "$(MACHINE)" "solaris-sparc"
    MV_FLAGS := -f
endif
```

If you are porting to many platforms, chaining the `ifdef` sections can become ugly and difficult to maintain. Instead of using `ifdef`, place each set of platform-specific variables in its own file whose name contains a platform indicator. For instance, if you designate a platform by its `uname` parameters, you can select the appropriate make include file like this:

```
MACHINE := $(shell uname -smo | sed 's/ /-/g')
include $(MACHINE)-defines.mk
```

Filenames with spaces present a particularly irritating problem for `make`. The assumption that whitespace separates tokens during parsing is fundamental to `make`. Many built-in functions such as `word`, `filter`, `wildcard`, and others assume their arguments are space-separated words. Nevertheless, here are some tricks that may help in small

ways. The first trick, noted in the section “Supporting Multiple Binary Trees” in Chapter 8, is how to replace spaces with another character using `subst`:

```
space = $(empty) $(empty)

# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,,$1)
```

The `space-to-question` function replaces all spaces with the globbing wildcard question mark. Now, we can implement `wildcard` and `file-exists` functions that can handle spaces:

```
# $(call wildcard-spaces,file-name)
wildcard-spaces = $(wildcard $(call space-to-question,$1))

# $(call file-exists
file-exists = $(strip                                     \
                $(if $1,,$(warning $1 has no value))      \
                $(call wildcard-spaces,$1))
```

The `wildcard-spaces` function uses `space-to-question` to allow the *makefile* to perform a wildcard operation on a pattern including spaces. We can use our `wildcard-spaces` function to implement `file-exists`. Of course, the use of the question mark may also cause `wildcard-spaces` to return files that do not correctly match the original wildcard pattern (e.g., “my document.doc” and “my-document.doc”), but this is the best we can do.

The `space-to-question` function can also be used to transform filenames with spaces in targets and prerequisites, since those allow globbing patterns to be used.

```
space := $(empty) $(empty)

# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,,$1)

# $(call question-to-space,file-name)
question-to-space = $(subst ?,$(space),,$1)

$(call space-to-question,foo bar): $(call space-to-question,bar baz)
    touch "$(call question-to-space,$@)"
```

Assuming the file “*bar baz*” exists, the first time this *makefile* is executed the prerequisite is found because the globbing pattern is evaluated. But the target globbing pattern fails because the target does not yet exist, so `$(?)` has the value `foo?bar`. The command script then uses `question-to-space` to transform `$(?)` back to the file with spaces that we really want. The next time the *makefile* is run, the target is found because the globbing pattern finds the target with spaces. A bit ugly, but I have found these tricks useful in real *makefiles*.

Source Tree Layout

Another aspect of portability is the ability to allow developers freedom to manage their development environment as they deem necessary. There will be problems if the build system requires the developers to always place their source, binaries, libraries, and support tools under the same directory or on the same Windows disk drive, for instance. Eventually, some developer low on disk space will be faced with the problem of having to partition these various files.

Instead, it makes sense to implement the *makefile* using variables to reference these collections of files and set reasonable defaults. In addition, each support library and tool can be referenced through a variable to allow developers to customize file locations as they find necessary. For the most likely customization variables, use the conditional assignment operator to allow developers a simple way of overriding the *makefile* with environment variables.

In addition, the ability to easily support multiple copies of the source and binary tree is a boon to developers. Even if they don't have to support different platforms or compilation options, developers often find themselves working with several copies of the source, either for debugging purposes or because they work on several projects in parallel. Two ways to support this have already been discussed: use a "top-level" environment variable to identify the root of the source and binary trees, or use the directory of the *makefile* and a fixed relative path to find the binary tree. Either of these allows developers the flexibility of supporting more than one tree.

Working with Nonportable Tools

As noted previously, one alternative to writing *makefiles* to the least common denominator is to adopt some standard tools. Of course, the goal is to make sure the standard tools are at least as portable as the application you are building. The obvious choice for portable tools are programs from the GNU project, but portable tools come from a wide variety of sources. Perl and Python are two other tools that come to mind.

In the absence of portable tools, encapsulating nonportable tools in make functions can sometimes do just as well. For instance, to support a variety of compilers for Enterprise JavaBeans (each of which has a slightly different invocation syntax), we can write a basic function to compile an EJB jar and parameterize it to allow one to plug in different compilers.

```
EJB_TMP_JAR = $(TMPDIR)/temp.jar

# $(call compile-generic-bean, bean-type, jar-name,
#                               bean-files-wildcard, manifest-name-opt )
define compile-generic-bean
$(RM) $(dir $(META_INF))
$(MKDIR) $(META_INF)
```

```

$(if $(filter %.xml %.xmi, $3), \
  cp $(filter %.xml %.xmi, $3) $(META_INF))
$(call compile-$1-bean-hook,$2)
cd $(OUTPUT_DIR) && \
$(JAR) -cfo $(EJB_TMP_JAR) \
      $(call jar-file-arg,$(META_INF)) \
      $(call bean-classes,$3)
$(call $1-compile-command,$2)
$(call create-manifest,$(if $4,$4,$2),)
endif

```

The first argument to this general EJB compilation function is the type of bean compiler we are using, such as Weblogic, Websphere, etc. The remaining arguments are the jar name, the files forming the content of the jar (including configuration files), and an optional manifest file. The template function first creates a clean temporary area by deleting any old temporary directory and recreating it. Next, the function copies in the *xml* or *xmi* files present in the prerequisites into the `$(META_INF)` directory. At this point, we may need to perform custom operations to clean up the *META-INF* files or prepare the *.class* files. To support these operations, we include a hook function, `compile-$1-bean-hook`, that the user can define, if necessary. For instance, if the Websphere compiler required an extra control file, say an *xsl* file, we would write this hook:

```

# $(call compile-websphere-bean-hook, file-list)
define compile-websphere-bean-hook
  cp $(filter %.xsl, $1) $(META_INF)
endif

```

By simply defining this function, we make sure the call in `compile-generic-bean` will be expanded appropriately. If we do not choose to write a hook function, the call in `compile-generic-bean` expands to nothing.

Next, our generic function creates the jar. The helper function `jar-file-arg` decomposes a normal file path into a `-C` option and a relative path:

```

# $(call jar-file-arg, file-name)
define jar-file-arg
  -C "$(patsubst %/,%, $(dir $1))" $(notdir $1)
endif

```

The helper function `bean-classes` extracts the appropriate class files from a source file list (the jar file only needs the interface and home classes):

```

# $(call bean-classes, bean-files-list)
define bean-classes
  $(subst $(SOURCE_DIR)/,, \
    $(filter %Interface.class %Home.class, \
      $(subst .java,.class,$1)))
endif

```

Then the generic function invokes the compiler of choice with `$(call $1-compile-command,$2)`:

```
define weblogic-compile-command
  cd $(TMPDIR) && \
    $(JVM) weblogic.ejbrc -compiler $(EJB_JAVAC) $(EJB_TMP_JAR) $1
endef
```

Finally, our generic function adds the manifest.

Having defined `compile-generic-bean`, we wrap it in a compiler-specific function for each environment we want to support.

```
# $(call compile-weblogic-bean, jar-name,
#                               bean-files-wildcard, manifest-name-opt )
define compile-weblogic-bean
  $(call compile-generic-bean,weblogic,$1,$2,$3)
endef
```

A Standard Shell

It is worth reiterating here that one of the irksome incompatibilities one finds in moving from system to system is the capabilities of `/bin/sh`, the default shell used by `make`. If you find yourself tweaking the command scripts in your *makefile*, you should consider standardizing your shell. Of course, this is not reasonable for the typical open source project where the *makefile* is executed in uncontrolled environments. However, in a controlled setting, with a fixed set of specially configured machines, this is quite reasonable.

In addition to avoiding shell incompatibilities, many shells provide features that can avoid the use of numerous small utilities. For example, the `bash` shell includes enhanced shell variable expansion, such as `%` and `##`, that can help avoid the use of shell utilities, such as `sed` and `expr`.

Automake

The focus of this chapter has been on using GNU `make` and supporting tools effectively to achieve a portable build system. There are times, however, when even these modest requirements are beyond reach. If you cannot use the enhanced features of GNU `make` and are forced to rely on a least-common-denominator set of features, you should consider using the `automake` tool, <http://www.gnu.org/software/automake/automake.html>.

The `automake` tool accepts a stylized *makefile* as input and generates a portable old-style *makefile* as output. `automake` is built around a set of `m4` macros that allow a very terse notation in the input file (called *makefile.am*). Typically, `automake` is used in conjunction with `autoconf`, a portability support package for C/C++ programs, but `autoconf` is not required.

While automake is a good solution for build systems that require maximum portability, the *makefiles* it generates cannot use any of the advanced features of GNU make with the exception of appending assignment, +=, for which it has special support. Furthermore, the input to automake bears little resemblance to normal *makefile* input. Thus, using automake (without autoconf) isn't terribly different from using the least-common-denominator approach.