GNU Coding Standards

Richard Stallman
last updated August 26, 1998

1 About the GNU Coding Standards

The GNU Coding Standards were written by Richard Stallman and other GNU Project volunteers. Their purpose is to make the GNU system clean, consistent, and easy to install. This document can also be read as a guide to writing portable, robust and reliable programs. It focuses on programs written in C, but many of the rules and principles are useful even if you write in another programming language. The rules often state reasons for writing in a certain way.

Corrections or suggestions for this document should be sent to `gnu@gnu.org`. If you make a suggestion, please include a suggested new wording for it; our time is limited. We prefer a context diff to the '`standards.texi`' or '`make-stds.texi`' files, but if you don't have those files, please mail your suggestion anyway.

This release of the GNU Coding Standards was last updated August 26, 1998.

2 Keeping Free Software Free

This chapter discusses how you can make sure that GNU software remains unencumbered.

2.1 Referring to Proprietary Programs

Don't in any circumstances refer to Unix source code for or during your work on GNU! (Or to any other proprietary programs.)

If you have a vague recollection of the internals of a Unix program, this does not absolutely mean you can't write an imitation of it, but do try to organize the imitation internally along different lines, because this is likely to make the details of the Unix version irrelevant and dissimilar to your results.

For example, Unix utilities were generally optimized to minimize memory use; if you go for speed instead, your program will be very different. You could keep the entire input file in core and scan it there instead of using stdio. Use a smarter algorithm discovered more recently than the Unix program. Eliminate use of temporary files. Do it in one pass instead of two (we did this in the assembler).

Or, on the contrary, emphasize simplicity instead of speed. For some applications, the speed of today's computers makes simpler algorithms adequate.

Or go for generality. For example, Unix programs often have static tables or fixed-size strings, which make for arbitrary limits; use dynamic allocation instead. Make sure your program handles NULs and other funny characters in the input files. Add a programming language for extensibility and write part of the program in that language.

Or turn some parts of the program into independently usable libraries. Or use a simple garbage collector instead of tracking precisely when to free memory, or use a new GNU facility such as obstacks.

2.2 Accepting Contributions

If someone else sends you a piece of code to add to the program you are working on, we need legal papers to use it—the same sort of legal papers we will need to get from you. *Each* significant contributor to a program must sign some sort of legal papers in order for us to have clear title to the program. The main author alone is not enough.

So, before adding in any contributions from other people, please tell us, so we can arrange to get the papers. Then wait until we tell you that we have received the signed papers, before you actually use the contribution.

This applies both before you release the program and afterward. If you receive diffs to fix a bug, and they make significant changes, we need legal papers for that change.

This also applies to comments and documentation files. For copyright law, comments and code are just text. Copyright applies to all kinds of text, so we need legal papers for all kinds.

You don't need papers for changes of a few lines here or there, since they are not significant for copyright purposes. Also, you don't need papers if all you get from the suggestion is some ideas, not actual code which you use. For example, if you write a different solution to the problem, you don't need to get papers.

We know this is frustrating; it's frustrating for us as well. But if you don't wait, you are going out on a limb—for example, what if the contributor's employer won't sign a disclaimer? You might have to take that code out again!

The very worst thing is if you forget to tell us about the other contributor. We could be very embarrassed in court some day as a result.

We have more detailed advice for maintainers of programs; if you have reached the stage of actually maintaining a program for GNU (whether released or not), please ask us for a copy.

3 General Program Design

This chapter discusses some of the issues you should take into account when designing your program.

3.1 Compatibility with Other Implementations

With occasional exceptions, utility programs and libraries for GNU should be upward compatible with those in Berkeley Unix, and upward compatible with ANSI C if ANSI C specifies their behavior, and upward compatible with POSIX if POSIX specifies their behavior.

When these standards conflict, it is useful to offer compatibility modes for each of them.

ANSI C and POSIX prohibit many kinds of extensions. Feel free to make the extensions anyway, and include a '`--ansi`', '`--posix`', or '`--compatible`' option to turn them off. However, if the extension has a significant chance of breaking any real programs or scripts, then it is not really upward compatible. Try to redesign its interface.

Many GNU programs suppress extensions that conflict with POSIX if the environment variable `POSIXLY_CORRECT` is defined (even if it is defined with a null value). Please make your program recognize this variable if appropriate.

When a feature is used only by users (not by programs or command files), and it is done poorly in Unix, feel free to replace it completely with something totally different and better. (For example, `vi` is replaced with Emacs.) But it is nice to offer a compatible feature as well. (There is a free `vi` clone, so we offer it.)

Additional useful features not in Berkeley Unix are welcome.

3.2 Using Non-standard Features

Many GNU facilities that already exist support a number of convenient extensions over the comparable Unix facilities. Whether to use these extensions in implementing your program is a difficult question.

On the one hand, using the extensions can make a cleaner program. On the other hand, people will not be able to build the program unless the other GNU tools are available. This might cause the program to work on fewer kinds of machines.

With some extensions, it might be easy to provide both alternatives. For example, you can define functions with a "keyword" `INLINE` and define that as a macro to expand into either `inline` or nothing, depending on the compiler.

In general, perhaps it is best not to use the extensions if you can straightforwardly do without them, but to use the extensions if they are a big improvement.

An exception to this rule are the large, established programs (such as Emacs) which run on a great variety of systems. Such programs would be broken by use of GNU extensions.

Another exception is for programs that are used as part of compilation: anything that must be compiled with other compilers in order to bootstrap the GNU compilation facilities. If these require the GNU compiler, then no one can compile them without having them installed already. That would be no good.

3.3 ANSI C and pre-ANSI C

Do not ever use the "trigraph" feature of ANSI C.

ANSI C is widespread enough now that it is ok to write new programs that use ANSI C features (and therefore will not work in non-ANSI compilers). And if a program is already written in ANSI C, there's no need to convert it to support non-ANSI compilers.

However, it is easy to support non-ANSI compilers in most programs, so you might still consider doing so when you write a program. Instead of writing function definitions in ANSI prototype form,

```
int
foo (int x, int y)
...
```

write the definition in pre-ANSI style like this,

```
int
foo (x, y)
     int x, y;
...
```

and use a separate declaration to specify the argument prototype:

```
int foo (int, int);
```

You need such a declaration anyway, in a header file, to get the benefit of ANSI C prototypes in all the files where the function is called. And once you have it, you lose nothing by writing the function definition in the pre-ANSI style.

If you don't know non-ANSI C, there's no need to learn it; just write in ANSI C.

3.4 Using Languages Other Than C

Using a language other than C is like using a non-standard feature: it will cause trouble for users. Even if GCC supports the other language, users may find it inconvenient to have to install the compiler for that other language in order to build your program. For example, if you write your program in C++, people will have to install the C++ compiler in order to compile your program. Thus, it is better if you write in C.

But there are three situations when there is no disadvantage in using some other language:

- It is okay to use another language if your program contains an interpreter for that language.

  For example, if your program links with GUILE, it is ok to write part of the program in Scheme or another language supported by GUILE.
- It is okay to use another language in a tool specifically intended for use with that language.

  This is okay because the only people who want to build the tool will be those who have installed the other language anyway.
- If an application is of interest to a narrow community, then perhaps it's not important if the application is inconvenient to install.

C has one other advantage over C++ and other compiled languages: more people know C, so more people will find it easy to read and modify the program if it is written in C.

4 Program Behavior for All Programs

This chapter describes how to write robust software. It also describes general standards for error messages, the command line interface, and how libraries should behave.

4.1 Writing Robust Programs

Avoid arbitrary limits on the length or number of *any* data structure, including file names, lines, files, and symbols, by allocating all data structures dynamically. In most Unix utilities, "long lines are silently truncated". This is not acceptable in a GNU utility.

Utilities reading files should not drop NUL characters, or any other nonprinting characters *including those with codes above 0177*. The only sensible exceptions would be utilities specifically intended for interface to certain types of printers that can't handle those characters.

Check every system call for an error return, unless you know you wish to ignore errors. Include the system error text (from `perror` or equivalent) in *every* error message resulting from a failing system call, as well as the name of the file if any and the name of the utility. Just "cannot open foo.c" or "stat failed" is not sufficient.

Check every call to `malloc` or `realloc` to see if it returned zero. Check `realloc` even if you are making the block smaller; in a system that rounds block sizes to a power of 2, `realloc` may get a different block if you ask for less space.

In Unix, `realloc` can destroy the storage block if it returns zero. GNU `realloc` does not have this bug: if it fails, the original block is unchanged. Feel free to assume the bug is fixed. If you wish to run your program on Unix, and wish to avoid lossage in this case, you can use the GNU `malloc`.

You must expect `free` to alter the contents of the block that was freed. Anything you want to fetch from the block, you must fetch before calling `free`.

If `malloc` fails in a noninteractive program, make that a fatal error. In an interactive program (one that reads commands from the user), it is better to abort the command and return to the command reader loop. This allows the user to kill other processes to free up virtual memory, and then try the command again.

Use `getopt_long` to decode arguments, unless the argument syntax makes this unreasonable.

When static storage is to be written in during program execution, use explicit C code to initialize it. Reserve C initialized declarations for data that will not be changed.

Try to avoid low-level interfaces to obscure Unix data structures (such as file directories, utmp, or the layout of kernel memory), since these are less likely to work compatibly. If you need to find all the files in a directory, use `readdir` or some other high-level interface. These will be supported compatibly by GNU.

The preferred signal handling facilities are the BSD variant of `signal`, and the POSIX `sigaction` function; the alternative USG `signal` interface is an inferior design.

Nowadays, using the POSIX signal functions may be the easiest way to make a program portable. If you use `signal`, then on GNU/Linux systems running GNU libc version 1, you should include '`bsd/signal.h`' instead of '`signal.h`', so as to get BSD behavior. It is up to you whether to support systems where `signal` has only the USG behavior, or give up on them.

In error checks that detect "impossible" conditions, just abort. There is usually no point in printing any message. These checks indicate the existence of bugs. Whoever wants to fix the bugs will have to read the source code and run a debugger. So explain the problem with comments in the source. The relevant data will be in variables, which are easy to examine with the debugger, so there is no point moving them elsewhere.

Do not use a count of errors as the exit status for a program. *That does not work*, because exit status values are limited to 8 bits (0 through 255). A single run of the program might have 256 errors; if you try to return 256 as the exit status, the parent process will see 0 as the status, and it will appear that the program succeeded.

If you make temporary files, check the `TMPDIR` environment variable; if that variable is defined, use the specified directory instead of '`/tmp`'.

4.2 Library Behavior

Try to make library functions reentrant. If they need to do dynamic storage allocation, at least try to avoid any nonreentrancy aside from that of `malloc` itself.

Here are certain name conventions for libraries, to avoid name conflicts.

Choose a name prefix for the library, more than two characters long. All external function and variable names should start with this prefix. In addition, there should only be one of these in any given library member. This usually means putting each one in a separate source file.

An exception can be made when two external symbols are always used together, so that no reasonable program could use one without the other; then they can both go in the same file.

External symbols that are not documented entry points for the user should have names beginning with '`_`'. They should also contain the chosen name prefix for the library, to prevent collisions with other libraries. These can go in the same files with user entry points if you like.

Static functions and variables can be used as you like and need not fit any naming convention.

4.3 Formatting Error Messages

Error messages from compilers should look like this:

source-file-name:*lineno*: *message*

Error messages from other noninteractive programs should look like this:

program:*source-file-name*:*lineno*: *message*

when there is an appropriate source file, or like this:

program: *message*

when there is no relevant source file.

In an interactive program (one that is reading commands from a terminal), it is better not to include the program name in an error message. The place to indicate which program is running is in the prompt or with the screen layout. (When the same program runs with input from a source other than a terminal, it is not interactive and would do best to print error messages using the noninteractive style.)

The string *message* should not begin with a capital letter when it follows a program name and/or file name. Also, it should not end with a period.

Error messages from interactive programs, and other messages such as usage messages, should start with a capital letter. But they should not end with a period.

4.4 Standards for Command Line Interfaces

Please don't make the behavior of a utility depend on the name used to invoke it. It is useful sometimes to make a link to a utility with a different name, and that should not change what it does.

Instead, use a run time option or a compilation switch or both to select among the alternate behaviors.

Likewise, please don't make the behavior of the program depend on the type of output device it is used with. Device independence is an important principle of the system's design; do not compromise it merely to save someone from typing an option now and then. (Variation in error message syntax when using a terminal is ok, because that is a side issue that people do not depend on.)

If you think one behavior is most useful when the output is to a terminal, and another is most useful when the output is a file or a pipe, then it is usually best to make the default behavior the one that is useful with output to a terminal, and have an option for the other behavior.

Compatibility requires certain programs to depend on the type of output device. It would be disastrous if `ls` or `sh` did not do so in the way all users expect. In some of these cases, we supplement the program with a preferred alternate version that does not depend on the output device type. For example, we provide a `dir` program much like `ls` except that its default output format is always multi-column format.

It is a good idea to follow the POSIX guidelines for the command-line options of a program. The easiest way to do this is to use `getopt` to parse them. Note that the GNU version of `getopt` will normally permit options anywhere among the arguments unless the special argument '`--`' is used. This is not what POSIX specifies; it is a GNU extension.

Please define long-named options that are equivalent to the single-letter Unix-style options. We hope to make GNU more user friendly this way. This is easy to do with the GNU function `getopt_long`.

One of the advantages of long-named options is that they can be consistent from program to program. For example, users should be able to expect the "verbose" option of any GNU program which has one, to be spelled precisely '`--verbose`'. To achieve this uniformity, look at the table of common long-option names when you choose the option names for your program (see Section 4.5 [Option Table], page 9).

It is usually a good idea for file names given as ordinary arguments to be input files only; any output files would be specified using options (preferably '`-o`' or '`--output`'). Even if you allow an output file name as an ordinary argument for compatibility, try to provide an option as another way to specify it. This will lead to more consistency among GNU utilities, and fewer idiosyncracies for users to remember.

All programs should support two standard options: '`--version`' and '`--help`'.

`--version`

This option should direct the program to information about its name, version, origin and legal status, all on standard output, and then exit successfully. Other options and arguments should be ignored once this is seen, and the program should not perform its normal function.

The first line is meant to be easy for a program to parse; the version number proper starts after the last space. In addition, it contains the canonical name for this program, in this format:

```
GNU Emacs 19.30
```

The program's name should be a constant string; *don't* compute it from `argv[0]`. The idea is to state the standard or canonical name for the program, not its file name. There are other ways to find out the precise file name where a command is found in `PATH`.

If the program is a subsidiary part of a larger package, mention the package name in parentheses, like this:

```
emacsserver (GNU Emacs) 19.30
```

If the package has a version number which is different from this program's version number, you can mention the package version number just before the close-parenthesis.

If you **need** to mention the version numbers of libraries which are distributed separately from the package which contains this program, you can do so by printing an additional line of version info for each library you want to mention. Use the same format for these lines as for the first line.

Please do not mention all of the libraries that the program uses "just for completeness"—that would produce a lot of unhelpful clutter. Please mention library version numbers only if you find in practice that they are very important to you in debugging.

The following line, after the version number line or lines, should be a copyright notice. If more than one copyright notice is called for, put each on a separate line.

Next should follow a brief statement that the program is free software, and that users are free to copy and change it on certain conditions. If the program is covered by the GNU GPL, say so here. Also mention that there is no warranty, to the extent permitted by law.

It is ok to finish the output with a list of the major authors of the program, as a way of giving credit.

Here's an example of output that follows these rules:

```
GNU Emacs 19.34.5
Copyright (C) 1996 Free Software Foundation, Inc.
GNU Emacs comes with NO WARRANTY,
to the extent permitted by law.
You may redistribute copies of GNU Emacs
under the terms of the GNU General Public License.
For more information about these matters,
see the files named COPYING.
```

You should adapt this to your program, of course, filling in the proper year, copyright holder, name of program, and the references to distribution terms, and changing the rest of the wording as necessary.

This copyright notice only needs to mention the most recent year in which changes were made—there's no need to list the years for previous versions' changes. You don't have to mention the name of the program in these notices, if that is inconvenient, since it appeared in the first line.

`--help` This option should output brief documentation for how to invoke the program, on standard output, then exit successfully. Other options and arguments should be ignored once this is seen, and the program should not perform its normal function.

Near the end of the '--help' option's output there should be a line that says where to mail bug reports. It should have this format:

```
Report bugs to mailing-address.
```

4.5 Table of Long Options

Here is a table of long options used by GNU programs. It is surely incomplete, but we aim to list all the options that a new program might want to be compatible with. If you use names not already in the table, please send gnu@gnu.org a list of them, with their meanings, so we can update the table.

'after-date'
: '-N' in tar.

'all'
: '-a' in du, ls, nm, stty, uname, and unexpand.

'all-text'
: '-a' in diff.

'almost-all'
: '-A' in ls.

'append'
: '-a' in etags, tee, time; '-r' in tar.

'archive'
: '-a' in cp.

'archive-name'
: '-n' in shar.

'arglength'
: '-l' in m4.

'ascii'
: '-a' in diff.

'assign'
: '-v' in gawk.

'assume-new'
: '-W' in Make.

'assume-old'
: '-o' in Make.

'auto-check'
: '-a' in recode.

'auto-pager'
: '-a' in wdiff.

'auto-reference'
: '-A' in ptx.

'avoid-wraps'
: '-n' in wdiff.

'background'
: For server programs, run in the background.

'backward-search'
: '-B' in ctags.

'basename'
: '-f' in shar.

'batch' Used in GDB.

'baud' Used in GDB.

'before' '-b' in tac.

'binary' '-b' in cpio and diff.

'bits-per-code'
: '-b' in shar.

'block-size'
: Used in cpio and tar.

'blocks' '-b' in head and tail.

'break-file'
: '-b' in ptx.

'brief' Used in various programs to make output shorter.

'bytes' '-c' in head, split, and tail.

'c++' '-C' in etags.

'catenate'
: '-A' in tar.

'cd' Used in various programs to specify the directory to use.

'changes' '-c' in chgrp and chown.

'classify'
: '-F' in ls.

'colons' '-c' in recode.

'command' '-c' in su; '-x' in GDB.

'compare' '-d' in tar.

'compat' Used in gawk.

'compress'
: '-Z' in tar and shar.

'concatenate'
: '-A' in tar.

'confirmation'
: '-w' in tar.

'context' Used in diff.

'copyleft'
: '-W copyleft' in gawk.

'copyright'
: '-C' in ptx, recode, and wdiff; '-W copyright' in gawk.

'core' Used in GDB.

'count' '-q' in who.

'count-links'
: '-l' in du.

'create' Used in tar and cpio.

'cut-mark'
: '-c' in shar.

'cxref' '-x' in ctags.

'date' '-d' in touch.

'debug' '-d' in Make and m4; '-t' in Bison.

'define' '-D' in m4.

'defines' '-d' in Bison and ctags.

'delete' '-D' in tar.

'dereference'
: '-L' in chgrp, chown, cpio, du, ls, and tar.

'dereference-args'
: '-D' in du.

'diacritics'
: '-d' in recode.

'dictionary-order'
: '-d' in look.

'diff' '-d' in tar.

'digits' '-n' in csplit.

'directory'
: Specify the directory to use, in various programs. In ls, it means to show directories themselves rather than their contents. In rm and ln, it means to not treat links to directories specially.

'discard-all'
: '-x' in strip.

'discard-locals'
: '-X' in strip.

'dry-run' '-n' in Make.

'ed' '-e' in diff.

'elide-empty-files'
'-z' in csplit.

'end-delete'
'-x' in wdiff.

'end-insert'
'-z' in wdiff.

'entire-new-file'
'-N' in diff.

'environment-overrides'
'-e' in Make.

'eof' '-e' in xargs.

'epoch' Used in GDB.

'error-limit'
Used in makeinfo.

'error-output'
'-o' in m4.

'escape' '-b' in ls.

'exclude-from'
'-X' in tar.

'exec' Used in GDB.

'exit' '-x' in xargs.

'exit-0' '-e' in unshar.

'expand-tabs'
'-t' in diff.

'expression'
'-e' in sed.

'extern-only'
'-g' in nm.

'extract' '-i' in cpio; '-x' in tar.

'faces' '-f' in finger.

'fast' '-f' in su.

'fatal-warnings'
'-E' in m4.

'file' '-f' in info, gawk, Make, mt, and tar; '-n' in sed; '-r' in touch.

'field-separator'
'-F' in gawk.

'file-prefix'
: '-b' in Bison.

'file-type'
: '-F' in ls.

'files-from'
: '-T' in tar.

'fill-column'
: Used in makeinfo.

'flag-truncation'
: '-F' in ptx.

'fixed-output-files'
: '-y' in Bison.

'follow'
: '-f' in tail.

'footnote-style'
: Used in makeinfo.

'force'
: '-f' in cp, ln, mv, and rm.

'force-prefix'
: '-F' in shar.

'foreground'
: For server programs, run in the foreground; in other words, don't do anything special to run the server in the background.

'format'
: Used in ls, time, and ptx.

'freeze-state'
: '-F' in m4.

'fullname'
: Used in GDB.

'gap-size'
: '-g' in ptx.

'get'
: '-x' in tar.

'graphic'
: '-i' in ul.

'graphics'
: '-g' in recode.

'group'
: '-g' in install.

'gzip'
: '-z' in tar and shar.

'hashsize'
: '-H' in m4.

'header'
: '-h' in objdump and recode

'heading' '-H' in who.

'help' Used to ask for brief usage information.

'here-delimiter'
'-d' in shar.

'hide-control-chars'
'-q' in ls.

'idle' '-u' in who.

'ifdef' '-D' in diff.

'ignore' '-I' in ls; '-x' in recode.

'ignore-all-space'
'-w' in diff.

'ignore-backups'
'-B' in ls.

'ignore-blank-lines'
'-B' in diff.

'ignore-case'
'-f' in look and ptx; '-i' in diff and wdiff.

'ignore-errors'
'-i' in Make.

'ignore-file'
'-i' in ptx.

'ignore-indentation'
'-I' in etags.

'ignore-init-file'
'-f' in Oleo.

'ignore-interrupts'
'-i' in tee.

'ignore-matching-lines'
'-I' in diff.

'ignore-space-change'
'-b' in diff.

'ignore-zeros'
'-i' in tar.

'include' '-i' in etags; '-I' in m4.

'include-dir'
'-I' in Make.

'incremental'
'-G' in tar.

'info' '-i', '-l', and '-m' in Finger.

'initial' '-i' in expand.

'initial-tab'
'-T' in diff.

'inode' '-i' in ls.

'interactive'
'-i' in cp, ln, mv, rm; '-e' in m4; '-p' in xargs; '-w' in tar.

'intermix-type'
'-p' in shar.

'jobs' '-j' in Make.

'just-print'
'-n' in Make.

'keep-going'
'-k' in Make.

'keep-files'
'-k' in csplit.

'kilobytes'
'-k' in du and ls.

'language'
'-l' in etags.

'less-mode'
'-l' in wdiff.

'level-for-gzip'
'-g' in shar.

'line-bytes'
'-C' in split.

'lines' Used in split, head, and tail.

'link' '-l' in cpio.

'lint'
'lint-old'
Used in gawk.

'list' '-t' in cpio; '-l' in recode.

'list' '-t' in tar.

'literal' '-N' in ls.

'load-average'
'-l' in Make.

'login' Used in su.

'machine' No listing of which programs already use this; someone should check to see if any actually do, and tell gnu@gnu.org.

'macro-name'
'-M' in ptx.

'mail' '-m' in hello and uname.

'make-directories'
'-d' in cpio.

'makefile'
'-f' in Make.

'mapped' Used in GDB.

'max-args'
'-n' in xargs.

'max-chars'
'-n' in xargs.

'max-lines'
'-l' in xargs.

'max-load'
'-l' in Make.

'max-procs'
'-P' in xargs.

'mesg' '-T' in who.

'message' '-T' in who.

'minimal' '-d' in diff.

'mixed-uuencode'
'-M' in shar.

'mode' '-m' in install, mkdir, and mkfifo.

'modification-time'
'-m' in tar.

'multi-volume'
'-M' in tar.

'name-prefix'
'-a' in Bison.

'nesting-limit'
'-L' in m4.

'net-headers'
'-a' in shar.

'new-file'
'-W' in Make.

'no-builtin-rules'
: '-r' in Make.

'no-character-count'
: '-w' in shar.

'no-check-existing'
: '-x' in shar.

'no-common'
: '-3' in wdiff.

'no-create'
: '-c' in touch.

'no-defines'
: '-D' in etags.

'no-deleted'
: '-1' in wdiff.

'no-dereference'
: '-d' in cp.

'no-inserted'
: '-2' in wdiff.

'no-keep-going'
: '-S' in Make.

'no-lines'
: '-l' in Bison.

'no-piping'
: '-P' in shar.

'no-prof' '-e' in gprof.

'no-regex'
: '-R' in etags.

'no-sort' '-p' in nm.

'no-split'
: Used in makeinfo.

'no-static'
: '-a' in gprof.

'no-time' '-E' in gprof.

'no-timestamp'
: '-m' in shar.

'no-validate'
: Used in makeinfo.

'no-wait' Used in emacsclient.

'no-warn' Used in various programs to inhibit warnings.

'node' '-n' in info.

'nodename'
'-n' in uname.

'nonmatching'
'-f' in cpio.

'nstuff' '-n' in objdump.

'null' '-0' in xargs.

'number' '-n' in cat.

'number-nonblank'
'-b' in cat.

'numeric-sort'
'-n' in nm.

'numeric-uid-gid'
'-n' in cpio and ls.

'nx' Used in GDB.

'old-archive'
'-o' in tar.

'old-file'
'-o' in Make.

'one-file-system'
'-l' in tar, cp, and du.

'only-file'
'-o' in ptx.

'only-prof'
'-f' in gprof.

'only-time'
'-F' in gprof.

'output' In various programs, specify the output file name.

'output-prefix'
'-o' in shar.

'override'
'-o' in rm.

'overwrite'
'-c' in unshar.

'owner' '-o' in install.

'paginate'
'-l' in diff.

'paragraph-indent'
Used in makeinfo.

'parents' '-p' in mkdir and rmdir.

'pass-all'
'-p' in ul.

'pass-through'
'-p' in cpio.

'port' '-P' in finger.

'portability'
'-c' in cpio and tar.

'posix' Used in gawk.

'prefix-builtins'
'-P' in m4.

'prefix' '-f' in csplit.

'preserve'
Used in tar and cp.

'preserve-environment'
'-p' in su.

'preserve-modification-time'
'-m' in cpio.

'preserve-order'
'-s' in tar.

'preserve-permissions'
'-p' in tar.

'print' '-l' in diff.

'print-chars'
'-L' in cmp.

'print-data-base'
'-p' in Make.

'print-directory'
'-w' in Make.

'print-file-name'
'-o' in nm.

'print-symdefs'
'-s' in nm.

'printer' '-p' in wdiff.

'prompt' '-p' in ed.

'query-user'
: '-X' in shar.

'question'
: '-q' in Make.

'quiet'
: Used in many programs to inhibit the usual output. **Note:** every program accepting '--quiet' should accept '--silent' as a synonym.

'quiet-unshar'
: '-Q' in shar

'quote-name'
: '-Q' in ls.

'rcs'
: '-n' in diff.

're-interval'
: Used in gawk.

'read-full-blocks'
: '-B' in tar.

'readnow'
: Used in GDB.

'recon'
: '-n' in Make.

'record-number'
: '-R' in tar.

'recursive'
: Used in chgrp, chown, cp, ls, diff, and rm.

'reference-limit'
: Used in makeinfo.

'references'
: '-r' in ptx.

'regex'
: '-r' in tac and etags.

'release'
: '-r' in uname.

'reload-state'
: '-R' in m4.

'relocation'
: '-r' in objdump.

'rename'
: '-r' in cpio.

'replace'
: '-i' in xargs.

'report-identical-files'
: '-s' in diff.

'reset-access-time'
: '-a' in cpio.

'reverse' '-r' in ls and nm.

'reversed-ed'
'-f' in diff.

'right-side-defs'
'-R' in ptx.

'same-order'
'-s' in tar.

'same-permissions'
'-p' in tar.

'save' '-g' in stty.

'se' Used in GDB.

'sentence-regexp'
'-S' in ptx.

'separate-dirs'
'-S' in du.

'separator'
'-s' in tac.

'sequence'
Used by recode to chose files or pipes for sequencing passes.

'shell' '-s' in su.

'show-all'
'-A' in cat.

'show-c-function'
'-p' in diff.

'show-ends'
'-E' in cat.

'show-function-line'
'-F' in diff.

'show-tabs'
'-T' in cat.

'silent' Used in many programs to inhibit the usual output. **Note:** every program accepting '--silent' should accept '--quiet' as a synonym.

'size' '-s' in ls.

'socket' Specify a file descriptor for a network server to use for its socket, instead of opening and binding a new socket. This provides a way to run, in a nonpriveledged process, a server that normally needs a reserved port number.

'sort' Used in ls.

'source' '-W source' in gawk.

'sparse' '-S' in tar.

'speed-large-files'
'-H' in diff.

'split-at'
'-E' in unshar.

'split-size-limit'
'-L' in shar.

'squeeze-blank'
'-s' in cat.

'start-delete'
'-w' in wdiff.

'start-insert'
'-y' in wdiff.

'starting-file'
Used in tar and diff to specify which file within a directory to start processing with.

'statistics'
'-s' in wdiff.

'stdin-file-list'
'-S' in shar.

'stop' '-S' in Make.

'strict' '-s' in recode.

'strip' '-s' in install.

'strip-all'
'-s' in strip.

'strip-debug'
'-S' in strip.

'submitter'
'-s' in shar.

'suffix' '-S' in cp, ln, mv.

'suffix-format'
'-b' in csplit.

'sum' '-s' in gprof.

'summarize'
'-s' in du.

'symbolic'
'-s' in ln.

'symbols' Used in GDB and objdump.

'synclines'
'-s' in m4.

'sysname' '-s' in uname.

'tabs' '-t' in expand and unexpand.

'tabsize' '-T' in ls.

'terminal'
'-T' in tput and ul. '-t' in wdiff.

'text' '-a' in diff.

'text-files'
'-T' in shar.

'time' Used in ls and touch.

'to-stdout'
'-O' in tar.

'total' '-c' in du.

'touch' '-t' in Make, ranlib, and recode.

'trace' '-t' in m4.

'traditional'
'-t' in hello; '-W traditional' in gawk; '-G' in ed, m4, and ptx.

'tty' Used in GDB.

'typedefs'
'-t' in ctags.

'typedefs-and-c++'
'-T' in ctags.

'typeset-mode'
'-t' in ptx.

'uncompress'
'-z' in tar.

'unconditional'
'-u' in cpio.

'undefine'
'-U' in m4.

'undefined-only'
'-u' in nm.

'update' '-u' in cp, ctags, mv, tar.

'usage' Used in gawk; same as '--help'.

'uuencode'
'-B' in shar.

'vanilla-operation'
: '-V' in shar.

'verbose'
: Print more information about progress. Many programs support this.

'verify'
: '-W' in tar.

'version'
: Print the version number.

'version-control'
: '-V' in cp, ln, mv.

'vgrind'
: '-v' in ctags.

'volume'
: '-V' in tar.

'what-if'
: '-W' in Make.

'whole-size-limit'
: '-l' in shar.

'width'
: '-w' in ls and ptx.

'word-regexp'
: '-W' in ptx.

'writable'
: '-T' in who.

'zeros'
: '-z' in gprof.

4.6 Memory Usage

If it typically uses just a few meg of memory, don't bother making any effort to reduce memory usage. For example, if it is impractical for other reasons to operate on files more than a few meg long, it is reasonable to read entire input files into core to operate on them.

However, for programs such as `cat` or `tail`, that can usefully operate on very large files, it is important to avoid using a technique that would artificially limit the size of files it can handle. If a program works by lines and could be applied to arbitrary user-supplied input files, it should keep only a line in memory, because this is not very hard and users will want to be able to operate on input files that are bigger than will fit in core all at once.

If your program creates complicated data structures, just make them in core and give a fatal error if `malloc` returns zero.

5 Making The Best Use of C

This chapter provides advice on how best to use the C language when writing GNU software.

5.1 Formatting Your Source Code

It is important to put the open-brace that starts the body of a C function in column zero, and avoid putting any other open-brace or open-parenthesis or open-bracket in column zero. Several tools look for open-braces in column zero to find the beginnings of C functions. These tools will not work on code not formatted that way.

It is also important for function definitions to start the name of the function in column zero. This helps people to search for function definitions, and may also help certain tools recognize them. Thus, the proper format is this:

```
static char *
concat (s1, s2)        /* Name starts in column zero here */
     char *s1, *s2;
{                     /* Open brace in column zero here */
  ...
}
```

or, if you want to use ANSI C, format the definition like this:

```
static char *
concat (char *s1, char *s2)
{
  ...
}
```

In ANSI C, if the arguments don't fit nicely on one line, split it like this:

```
int
lots_of_args (int an_integer, long a_long, short a_short,
              double a_double, float a_float)
...
```

For the body of the function, we prefer code formatted like this:

```
if (x < foo (y, z))
  haha = bar[4] + 5;
else
  {
    while (z)
      {
        haha += foo (z, z);
        z--;
      }
    return ++x + bar ();
  }
```

We find it easier to read a program when it has spaces before the open-parentheses and after the commas. Especially after the commas.

When you split an expression into multiple lines, split it before an operator, not after one. Here is the right way:

```
if (foo_this_is_long && bar > win (x, y, z)
    && remaining_condition)
```

Try to avoid having two operators of different precedence at the same level of indentation. For example, don't write this:

```
mode = (inmode[j] == VOIDmode
        || GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])
        ? outmode[j] : inmode[j]);
```

Instead, use extra parentheses so that the indentation shows the nesting:

```
mode = ((inmode[j] == VOIDmode
         || (GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])))
        ? outmode[j] : inmode[j]);
```

Insert extra parentheses so that Emacs will indent the code properly. For example, the following indentation looks nice if you do it by hand, but Emacs would mess it up:

```
v = rup->ru_utime.tv_sec*1000 + rup->ru_utime.tv_usec/1000
    + rup->ru_stime.tv_sec*1000 + rup->ru_stime.tv_usec/1000;
```

But adding a set of parentheses solves the problem:

```
v = (rup->ru_utime.tv_sec*1000 + rup->ru_utime.tv_usec/1000
     + rup->ru_stime.tv_sec*1000 + rup->ru_stime.tv_usec/1000);
```

Format do-while statements like this:

```
do
  {
    a = foo (a);
  }
while (a > 0);
```

Please use formfeed characters (control-L) to divide the program into pages at logical places (but not within a function). It does not matter just how long the pages are, since they do not have to fit on a printed page. The formfeeds should appear alone on lines by themselves.

5.2 Commenting Your Work

Every program should start with a comment saying briefly what it is for. Example: '`fmt - filter for simple filling of text`'.

Please write the comments in a GNU program in English, because English is the one language that nearly all programmers in all countries can read. If you do not write English well, please write comments in English as well as you can, then ask other people to help rewrite them. If you can't write comments in English, please find someone to work with you and translate your comments into English.

Please put a comment on each function saying what the function does, what sorts of arguments it gets, and what the possible values of arguments mean and are used for. It is not necessary to duplicate in words the meaning of the C argument declarations, if a C type is being used in its customary fashion. If there is anything nonstandard about its use (such as an argument of type `char *` which is really the address of the second character of a string, not the first), or any possible values that would not work the way one would expect (such as, that strings containing newlines are not guaranteed to work), be sure to say so.

Also explain the significance of the return value, if there is one.

Please put two spaces after the end of a sentence in your comments, so that the Emacs sentence commands will work. Also, please write complete sentences and capitalize the first word. If a lower-case identifier comes at the beginning of a sentence, don't capitalize it!

Changing the spelling makes it a different identifier. If you don't like starting a sentence with a lower case letter, write the sentence differently (e.g., "The identifier lower-case is ...").

The comment on a function is much clearer if you use the argument names to speak about the argument values. The variable name itself should be lower case, but write it in upper case when you are speaking about the value rather than the variable itself. Thus, "the inode number NODE_NUM" rather than "an inode".

There is usually no purpose in restating the name of the function in the comment before it, because the reader can see that for himself. There might be an exception when the comment is so long that the function itself would be off the bottom of the screen.

There should be a comment on each static variable as well, like this:

```
/* Nonzero means truncate lines in the display;
   zero means continue them.  */
int truncate_lines;
```

Every '`#endif`' should have a comment, except in the case of short conditionals (just a few lines) that are not nested. The comment should state the condition of the conditional that is ending, *including its sense*. '`#else`' should have a comment describing the condition *and sense* of the code that follows. For example:

```
#ifdef foo
  ...
#else /* not foo */
  ...
#endif /* not foo */
#ifdef foo
  ...
#endif /* foo */
```

but, by contrast, write the comments this way for a '`#ifndef`':

```
#ifndef foo
  ...
#else /* foo */
  ...
#endif /* foo */
#ifndef foo
  ...
#endif /* not foo */
```

5.3 Clean Use of C Constructs

Please explicitly declare all arguments to functions. Don't omit them just because they are `int`s.

Declarations of external functions and functions to appear later in the source file should all go in one place near the beginning of the file (somewhere before the first function definition in the file), or else should go in a header file. Don't put `extern` declarations inside functions.

It used to be common practice to use the same local variables (with names like `tem`) over and over for different values within one function. Instead of doing this, it is better declare a separate local variable for each distinct purpose, and give it a name which is meaningful. This not only makes programs easier to understand, it also facilitates optimization by good compilers. You can also move the declaration of each local variable into the smallest scope that includes all its uses. This makes the program even cleaner.

Don't use local variables or parameters that shadow global identifiers.

Don't declare multiple variables in one declaration that spans lines. Start a new declaration on each line, instead. For example, instead of this:

```
int    foo,
       bar;
```

write either this:

```
int foo, bar;
```

or this:

```
int foo;
int bar;
```

(If they are global variables, each should have a comment preceding it anyway.)

When you have an `if-else` statement nested in another `if` statement, always put braces around the `if-else`. Thus, never write like this:

```
if (foo)
  if (bar)
    win ();
  else
    lose ();
```

always like this:

```
if (foo)
  {
    if (bar)
      win ();
    else
      lose ();
  }
```

If you have an `if` statement nested inside of an `else` statement, either write `else if` on one line, like this,

```
if (foo)
  ...
else if (bar)
  ...
```

with its `then`-part indented like the preceding `then`-part, or write the nested `if` within braces like this:

```
if (foo)
  ...
else
  {
    if (bar)
      ...
  }
```

Don't declare both a structure tag and variables or typedefs in the same declaration. Instead, declare the structure tag separately and then use it to declare the variables or typedefs.

Try to avoid assignments inside `if`-conditions. For example, don't write this:

```
if ((foo = (char *) malloc (sizeof *foo)) == 0)
  fatal ("virtual memory exhausted");
```

instead, write this:

```
foo = (char *) malloc (sizeof *foo);
if (foo == 0)
  fatal ("virtual memory exhausted");
```

Don't make the program ugly to placate `lint`. Please don't insert any casts to `void`. Zero without a cast is perfectly fine as a null pointer constant, except when calling a varargs function.

5.4 Naming Variables and Functions

The names of global variables and functions in a program serve as comments of a sort. So don't choose terse names—instead, look for names that give useful information about the meaning of the variable or function. In a GNU program, names should be English, like other comments.

Local variable names can be shorter, because they are used only within one context, where (presumably) comments explain their purpose.

Please use underscores to separate words in a name, so that the Emacs word commands can be useful within them. Stick to lower case; reserve upper case for macros and `enum` constants, and for name-prefixes that follow a uniform convention.

For example, you should use names like `ignore_space_change_flag`; don't use names like `iCantReadThis`.

Variables that indicate whether command-line options have been specified should be named after the meaning of the option, not after the option-letter. A comment should state both the exact meaning of the option and its letter. For example,

```
/* Ignore changes in horizontal whitespace (-b).  */
int ignore_space_change_flag;
```

When you want to define names with constant integer values, use `enum` rather than '`#define`'. GDB knows about enumeration constants.

Use file names of 14 characters or less, to avoid creating gratuitous problems on older System V systems. You can use the program `doschk` to test for this. `doschk` also tests for potential name conflicts if the files were loaded onto an MS-DOS file system—something you may or may not care about.

5.5 Portability between System Types

In the Unix world, "portability" refers to porting to different Unix versions. For a GNU program, this kind of portability is desirable, but not paramount.

The primary purpose of GNU software is to run on top of the GNU kernel, compiled with the GNU C compiler, on various types of CPU. The amount and kinds of variation among

GNU systems on different CPUs will be comparable to the variation among Linux-based GNU systems or among BSD systems today. So the kinds of portability that are absolutely necessary are quite limited.

But many users do run GNU software on non-GNU Unix or Unix-like systems. So supporting a variety of Unix-like systems is desirable, although not paramount.

The easiest way to achieve portability to most Unix-like systems is to use Autoconf. It's unlikely that your program needs to know more information about the host platform than Autoconf can provide, simply because most of the programs that need such knowledge have already been written.

Avoid using the format of semi-internal data bases (e.g., directories) when there is a higher-level alternative (`readdir`).

As for systems that are not like Unix, such as MSDOS, Windows, the Macintosh, VMS, and MVS, supporting them is usually so much work that it is better if you don't.

The planned GNU kernel is not finished yet, but you can tell which facilities it will provide by looking at the GNU C Library Manual. The GNU kernel is based on Mach, so the features of Mach will also be available. However, if you use Mach features, you'll probably have trouble debugging your program today.

5.6 Portability between CPUs

Even GNU systems will differ because of differences among CPU types—for example, difference in byte ordering and alignment requirements. It is absolutely essential to handle these differences. However, don't make any effort to cater to the possibility that an `int` will be less than 32 bits. We don't support 16-bit machines in GNU.

Don't assume that the address of an `int` object is also the address of its least-significant byte. This is false on big-endian machines. Thus, don't make the following mistake:

```
int c;
...
while ((c = getchar()) != EOF)
  write(file_descriptor, &c, 1);
```

When calling functions, you need not worry about the difference between pointers of various types, or between pointers and integers. On most machines, there's no difference anyway. As for the few machines where there is a difference, all of them support ANSI C, so you can use prototypes (conditionalized to be active only in ANSI C) to make the code work on those systems.

In certain cases, it is ok to pass integer and pointer arguments indiscriminately to the same function, and use no prototype on any system. For example, many GNU programs have error-reporting functions that pass their arguments along to `printf` and friends:

```
error (s, a1, a2, a3)
     char *s;
     int a1, a2, a3;
{
  fprintf (stderr, "error: ");
  fprintf (stderr, s, a1, a2, a3);
}
```

In practice, this works on all machines, and it is much simpler than any "correct" alternative. Be sure *not* to use a prototype for such functions.

However, avoid casting pointers to integers unless you really need to. These assumptions really reduce portability, and in most programs they are easy to avoid. In the cases where casting pointers to integers is essential—such as, a Lisp interpreter which stores type information as well as an address in one word—it is ok to do so, but you'll have to make explicit provisions to handle different word sizes.

5.7 Calling System Functions

C implementations differ substantially. ANSI C reduces but does not eliminate the incompatibilities; meanwhile, many users wish to compile GNU software with pre-ANSI compilers. This chapter gives recommendations for how to use the more or less standard C library functions to avoid unnecessary loss of portability.

- Don't use the value of `sprintf`. It returns the number of characters written on some systems, but not on all systems.
- `main` should be declared to return type `int`. It should terminate either by calling `exit` or by returning the integer status code; make sure it cannot ever return an undefined value.
- Don't declare system functions explicitly.

  Almost any declaration for a system function is wrong on some system. To minimize conflicts, leave it to the system header files to declare system functions. If the headers don't declare a function, let it remain undeclared.

  While it may seem unclean to use a function without declaring it, in practice this works fine for most system library functions on the systems where this really happens; thus, the disadvantage is only theoretical. By contrast, actual declarations have frequently caused actual conflicts.
- If you must declare a system function, don't specify the argument types. Use an old-style declaration, not an ANSI prototype. The more you specify about the function, the more likely a conflict.
- In particular, don't unconditionally declare `malloc` or `realloc`.

  Most GNU programs use those functions just once, in functions conventionally named `xmalloc` and `xrealloc`. These functions call `malloc` and `realloc`, respectively, and check the results.

  Because `xmalloc` and `xrealloc` are defined in your program, you can declare them in other files without any risk of type conflict.

  On most systems, `int` is the same length as a pointer; thus, the calls to `malloc` and `realloc` work fine. For the few exceptional systems (mostly 64-bit machines), you can use **conditionalized** declarations of `malloc` and `realloc`—or put these declarations in configuration files specific to those systems.
- The string functions require special treatment. Some Unix systems have a header file '`string.h`'; others have '`strings.h`'. Neither file name is portable. There are two things you can do: use Autoconf to figure out which file to include, or don't include either file.

- If you don't include either strings file, you can't get declarations for the string functions from the header file in the usual way.

  That causes less of a problem than you might think. The newer ANSI string functions should be avoided anyway because many systems still don't support them. The string functions you can use are these:

  ```
  strcpy   strncpy   strcat   strncat
  strlen   strcmp    strncmp
  strchr   strrchr
  ```

  The copy and concatenate functions work fine without a declaration as long as you don't use their values. Using their values without a declaration fails on systems where the width of a pointer differs from the width of `int`, and perhaps in other cases. It is trivial to avoid using their values, so do that.

  The compare functions and `strlen` work fine without a declaration on most systems, possibly all the ones that GNU software runs on. You may find it necessary to declare them **conditionally** on a few systems.

  The search functions must be declared to return `char *`. Luckily, there is no variation in the data type they return. But there is variation in their names. Some systems give these functions the names `index` and `rindex`; other systems use the names `strchr` and `strrchr`. Some systems support both pairs of names, but neither pair works on all systems.

  You should pick a single pair of names and use it throughout your program. (Nowadays, it is better to choose `strchr` and `strrchr` for new programs, since those are the standard ANSI names.) Declare both of those names as functions returning `char *`. On systems which don't support those names, define them as macros in terms of the other pair. For example, here is what to put at the beginning of your file (or in a header) if you want to use the names `strchr` and `strrchr` throughout:

  ```
  #ifndef HAVE_STRCHR
  #define strchr index
  #endif
  #ifndef HAVE_STRRCHR
  #define strrchr rindex
  #endif

  char *strchr ();
  char *strrchr ();
  ```

Here we assume that `HAVE_STRCHR` and `HAVE_STRRCHR` are macros defined in systems where the corresponding functions exist. One way to get them properly defined is to use Autoconf.

5.8 Internationalization

GNU has a library called GNU gettext that makes it easy to translate the messages in a program into various languages. You should use this library in every program. Use English for the messages as they appear in the program, and let gettext provide the way to translate them into other languages.

Using GNU gettext involves putting a call to the `gettext` macro around each string that might need translation—like this:

```
printf (gettext ("Processing file `%s'..."));
```

This permits GNU gettext to replace the string `"Processing file `%s'..."` with a translated version.

Once a program uses gettext, please make a point of writing calls to `gettext` when you add new strings that call for translation.

Using GNU gettext in a package involves specifying a *text domain name* for the package. The text domain name is used to separate the translations for this package from the translations for other packages. Normally, the text domain name should be the same as the name of the package—for example, '`fileutils`' for the GNU file utilities.

To enable gettext to work well, avoid writing code that makes assumptions about the structure of words or sentences. When you want the precise text of a sentence to vary depending on the data, use two or more alternative string constants each containing a complete sentences, rather than inserting conditionalized words or phrases into a single sentence framework.

Here is an example of what not to do:

```
printf ("%d file%s processed", nfiles,
        nfiles != 1 ? "s" : "");
```

The problem with that example is that it assumes that plurals are made by adding 's'. If you apply gettext to the format string, like this,

```
printf (gettext ("%d file%s processed"), nfiles,
        nfiles != 1 ? "s" : "");
```

the message can use different words, but it will still be forced to use 's' for the plural. Here is a better way:

```
printf ((nfiles != 1 ? "%d files processed"
         : "%d file processed"),
        nfiles);
```

This way, you can apply gettext to each of the two strings independently:

```
printf ((nfiles != 1 ? gettext ("%d files processed")
         : gettext ("%d file processed")),
        nfiles);
```

This can be any method of forming the plural of the word for "file", and also handles languages that require agreement in the word for "processed".

A similar problem appears at the level of sentence structure with this code:

```
printf ("#  Implicit rule search has%s been done.\n",
        f->tried_implicit ? "" : " not");
```

Adding `gettext` calls to this code cannot give correct results for all languages, because negation in some languages requires adding words at more than one place in the sentence. By contrast, adding `gettext` calls does the job straightfowardly if the code starts out like this:

```
printf (f->tried_implicit
        ? "#  Implicit rule search has been done.\n",
        : "#  Implicit rule search has not been done.\n");
```

5.9 Mmap

Don't assume that `mmap` either works on all files or fails for all files. It may work on some files and fail on others.

The proper way to use `mmap` is to try it on the specific file for which you want to use it—and if `mmap` doesn't work, fall back on doing the job in another way using `read` and `write`.

The reason this precaution is needed is that the GNU kernel (the HURD) provides a user-extensible file system, in which there can be many different kinds of "ordinary files." Many of them support `mmap`, but some do not. It is important to make programs handle all these kinds of files.

6 Documenting Programs

6.1 GNU Manuals

The preferred way to document part of the GNU system is to write a manual in the Texinfo formatting language. See the Texinfo manual, either the hardcopy, or the on-line version available through `info` or the Emacs Info subsystem (*C-h i*).

Programmers often find it most natural to structure the documentation following the structure of the implementation, which they know. But this structure is not necessarily good for explaining how to use the program; it may be irrelevant and confusing for a user.

At every level, from the sentences in a paragraph to the grouping of topics into separate manuals, the right way to structure documentation is according to the concepts and questions that a user will have in mind when reading it. Sometimes this structure of ideas matches the structure of the implementation of the software being documented—but often they are different. Often the most important part of learning to write good documentation is learning to notice when you are structuring the documentation like the implementation, and think about better alternatives.

For example, each program in the GNU system probably ought to be documented in one manual; but this does not mean each program should have its own manual. That would be following the structure of the implementation, rather than the structure that helps the user understand.

Instead, each manual should cover a coherent *topic*. For example, instead of a manual for `diff` and a manual for `diff3`, we have one manual for "comparison of files" which covers both of those programs, as well as `cmp`. By documenting these programs together, we can make the whole subject clearer.

The manual which discusses a program should document all of the program's command-line options and all of its commands. It should give examples of their use. But don't organize the manual as a list of features. Instead, organize it logically, by subtopics. Address the questions that a user will ask when thinking about the job that the program does.

In general, a GNU manual should serve both as tutorial and reference. It should be set up for convenient access to each topic through Info, and for reading straight through (appendixes aside). A GNU manual should give a good introduction to a beginner reading through from the start, and should also provide all the details that hackers want.

That is not as hard as it first sounds. Arrange each chapter as a logical breakdown of its topic, but order the sections, and write their text, so that reading the chapter straight through makes sense. Do likewise when structuring the book into chapters, and when structuring a section into paragraphs. The watchword is, *at each point, address the most fundamental and important issue raised by the preceding text.*

If necessary, add extra chapters at the beginning of the manual which are purely tutorial and cover the basics of the subject. These provide the framework for a beginner to understand the rest of the manual. The Bison manual provides a good example of how to do this.

Don't use Unix man pages as a model for how to write GNU documentation; most of them are terse, badly structured, and give inadequate explanation of the underlying concepts. (There are, of course exceptions.) Also Unix man pages use a particular format which is different from what we use in GNU manuals.

Please do not use the term "pathname" that is used in Unix documentation; use "file name" (two words) instead. We use the term "path" only for search paths, which are lists of file names.

Please do not use the term "illegal" to refer to erroneous input to a computer program. Please use "invalid" for this, and reserve the term "illegal" for violations of law.

6.2 Manual Structure Details

The title page of the manual should state the version of the programs or packages documented in the manual. The Top node of the manual should also contain this information. If the manual is changing more frequently than or independent of the program, also state a version number for the manual in both of these places.

Each program documented in the manual should should have a node named '*program* `Invocation`' or '`Invoking` *program*'. This node (together with its subnodes, if any) should describe the program's command line arguments and how to run it (the sort of information people would look in a man page for). Start with an '`@example`' containing a template for all the options and arguments that the program uses.

Alternatively, put a menu item in some menu whose item name fits one of the above patterns. This identifies the node which that item points to as the node for this purpose, regardless of the node's actual name.

There will be automatic features for specifying a program name and quickly reading just this part of its manual.

If one manual describes several programs, it should have such a node for each program described.

6.3 The NEWS File

In addition to its manual, the package should have a file named '`NEWS`' which contains a list of user-visible changes worth mentioning. In each new release, add items to the front of the file and identify the version they pertain to. Don't discard old items; leave them in the file after the newer items. This way, a user upgrading from any previous version can see what is new.

If the 'NEWS' file gets very long, move some of the older items into a file named 'ONEWS' and put a note at the end referring the user to that file.

6.4 Change Logs

Keep a change log to describe all the changes made to program source files. The purpose of this is so that people investigating bugs in the future will know about the changes that might have introduced the bug. Often a new bug can be found by looking at what was recently changed. More importantly, change logs can help you eliminate conceptual inconsistencies between different parts of a program, by giving you a history of how the conflicting concepts arose and who they came from.

6.4.1 Change Log Concepts

You can think of the change log as a conceptual "undo list" which explains how earlier versions were different from the current version. People can see the current version; they don't need the change log to tell them what is in it. What they want from a change log is a clear explanation of how the earlier version differed.

The change log file is normally called 'ChangeLog' and covers an entire directory. Each directory can have its own change log, or a directory can use the change log of its parent directory–it's up to you.

Another alternative is to record change log information with a version control system such as RCS or CVS. This can be converted automatically to a 'ChangeLog' file.

There's no need to describe the full purpose of the changes or how they work together. If you think that a change calls for explanation, you're probably right. Please do explain it—but please put the explanation in comments in the code, where people will see it whenever they see the code. For example, "New function" is enough for the change log when you add a function, because there should be a comment before the function definition to explain what it does.

However, sometimes it is useful to write one line to describe the overall purpose of a batch of changes.

The easiest way to add an entry to 'ChangeLog' is with the Emacs command *M-x add-change-log-entry*. An entry should have an asterisk, the name of the changed file, and then in parentheses the name of the changed functions, variables or whatever, followed by a colon. Then describe the changes you made to that function or variable.

6.4.2 Style of Change Logs

Here are some examples of change log entries:

```
* register.el (insert-register): Return nil.
(jump-to-register): Likewise.

* sort.el (sort-subr): Return nil.
```

```
* tex-mode.el (tex-bibtex-file, tex-file, tex-region):
Restart the tex shell if process is gone or stopped.
(tex-shell-running): New function.

* expr.c (store_one_arg): Round size up for move_block_to_reg.
(expand_call): Round up when emitting USE insns.
* stmt.c (assign_parms): Round size up for move_block_from_reg.
```

It's important to name the changed function or variable in full. Don't abbreviate function or variable names, and don't combine them. Subsequent maintainers will often search for a function name to find all the change log entries that pertain to it; if you abbreviate the name, they won't find it when they search.

For example, some people are tempted to abbreviate groups of function names by writing '`* register.el ({insert,jump-to}-register)`'; this is not a good idea, since searching for `jump-to-register` or `insert-register` would not find that entry.

Separate unrelated change log entries with blank lines. When two entries represent parts of the same change, so that they work together, then don't put blank lines between them. Then you can omit the file name and the asterisk when successive entries are in the same file.

6.4.3 Simple Changes

Certain simple kinds of changes don't need much detail in the change log.

When you change the calling sequence of a function in a simple fashion, and you change all the callers of the function, there is no need to make individual entries for all the callers that you changed. Just write in the entry for the function being called, "All callers changed."

```
* keyboard.c (Fcommand_execute): New arg SPECIAL.
All callers changed.
```

When you change just comments or doc strings, it is enough to write an entry for the file, without mentioning the functions. Just "Doc fixes" is enough for the change log.

There's no need to make change log entries for documentation files. This is because documentation is not susceptible to bugs that are hard to fix. Documentation does not consist of parts that must interact in a precisely engineered fashion. To correct an error, you need not know the history of the erroneous passage; it is enough to compare what the documentation says with the way the program actually works.

6.4.4 Conditional Changes

C programs often contain compile-time `#if` conditionals. Many changes are conditional; sometimes you add a new definition which is entirely contained in a conditional. It is very useful to indicate in the change log the conditions for which the change applies.

Our convention for indicating conditional changes is to use square brackets around the name of the condition.

Here is a simple example, describing a change which is conditional but does not have a function or entity name associated with it:

```
* xterm.c [SOLARIS2]: Include string.h.
```

Here is an entry describing a new definition which is entirely conditional. This new definition for the macro `FRAME_WINDOW_P` is used only when `HAVE_X_WINDOWS` is defined:

```
* frame.h [HAVE_X_WINDOWS] (FRAME_WINDOW_P): Macro defined.
```

Here is an entry for a change within the function `init_display`, whose definition as a whole is unconditional, but the changes themselves are contained in a '`#ifdef HAVE_LIBNCURSES`' conditional:

```
* dispnew.c (init_display) [HAVE_LIBNCURSES]: If X, call tgetent.
```

Here is an entry for a change that takes affect only when a certain macro is *not* defined:

```
(gethostname) [!HAVE_SOCKETS]: Replace with winsock version.
```

6.5 Man Pages

In the GNU project, man pages are secondary. It is not necessary or expected for every GNU program to have a man page, but some of them do. It's your choice whether to include a man page in your program.

When you make this decision, consider that supporting a man page requires continual effort each time the program is changed. The time you spend on the man page is time taken away from more useful work.

For a simple program which changes little, updating the man page may be a small job. Then there is little reason not to include a man page, if you have one.

For a large program that changes a great deal, updating a man page may be a substantial burden. If a user offers to donate a man page, you may find this gift costly to accept. It may be better to refuse the man page unless the same person agrees to take full responsibility for maintaining it—so that you can wash your hands of it entirely. If this volunteer later ceases to do the job, then don't feel obliged to pick it up yourself; it may be better to withdraw the man page from the distribution until someone else agrees to update it.

When a program changes only a little, you may feel that the discrepancies are small enough that the man page remains useful without updating. If so, put a prominent note near the beginning of the man page explaining that you don't maintain it and that the Texinfo manual is more authoritative. The note should say how to access the Texinfo documentation.

6.6 Reading other Manuals

There may be non-free books or documentation files that describe the program you are documenting.

It is ok to use these documents for reference, just as the author of a new algebra textbook can read other books on algebra. A large portion of any non-fiction book consists of facts, in this case facts about how a certain program works, and these facts are necessarily the same for everyone who writes about the subject. But be careful not to copy your outline structure, wording, tables or examples from preexisting non-free documentation. Copying from free documentation may be ok; please check with the FSF about the individual case.

7 The Release Process

Making a release is more than just bundling up your source files in a tar file and putting it up for FTP. You should set up your software so that it can be configured to run on a variety of systems. Your Makefile should conform to the GNU standards described below, and your directory layout should also conform to the standards discussed below. Doing so makes it easy to include your package into the larger framework of all GNU software.

7.1 How Configuration Should Work

Each GNU distribution should come with a shell script named `configure`. This script is given arguments which describe the kind of machine and system you want to compile the program for.

The `configure` script must record the configuration options so that they affect compilation.

One way to do this is to make a link from a standard name such as '`config.h`' to the proper configuration file for the chosen system. If you use this technique, the distribution should *not* contain a file named '`config.h`'. This is so that people won't be able to build the program without configuring it first.

Another thing that `configure` can do is to edit the Makefile. If you do this, the distribution should *not* contain a file named '`Makefile`'. Instead, it should include a file '`Makefile.in`' which contains the input used for editing. Once again, this is so that people won't be able to build the program without configuring it first.

If `configure` does write the '`Makefile`', then '`Makefile`' should have a target named '`Makefile`' which causes `configure` to be rerun, setting up the same configuration that was set up last time. The files that `configure` reads should be listed as dependencies of '`Makefile`'.

All the files which are output from the `configure` script should have comments at the beginning explaining that they were generated automatically using `configure`. This is so that users won't think of trying to edit them by hand.

The `configure` script should write a file named '`config.status`' which describes which configuration options were specified when the program was last configured. This file should be a shell script which, if run, will recreate the same configuration.

The `configure` script should accept an option of the form '`--srcdir=`*dirname*' to specify the directory where sources are found (if it is not the current directory). This makes it possible to build the program in a separate directory, so that the actual source directory is not modified.

If the user does not specify '`--srcdir`', then `configure` should check both '`.`' and '`..`' to see if it can find the sources. If it finds the sources in one of these places, it should use them from there. Otherwise, it should report that it cannot find the sources, and should exit with nonzero status.

Usually the easy way to support '`--srcdir`' is by editing a definition of `VPATH` into the Makefile. Some rules may need to refer explicitly to the specified source directory. To make this possible, `configure` can add to the Makefile a variable named `srcdir` whose value is precisely the specified directory.

The `configure` script should also take an argument which specifies the type of system to build the program for. This argument should look like this:

cpu-company-system

For example, a Sun 3 might be '`m68k-sun-sunos4.1`'.

The `configure` script needs to be able to decode all plausible alternatives for how to describe a machine. Thus, '`sun3-sunos4.1`' would be a valid alias. For many programs, '`vax-dec-ultrix`' would be an alias for '`vax-dec-bsd`', simply because the differences between Ultrix and BSD are rarely noticeable, but a few programs might need to distinguish them.

There is a shell script called '`config.sub`' that you can use as a subroutine to validate system types and canonicalize aliases.

Other options are permitted to specify in more detail the software or hardware present on the machine, and include or exclude optional parts of the package:

'`--enable-`*feature*[=*parameter*]'

Configure the package to build and install an optional user-level facility called *feature*. This allows users to choose which optional features to include. Giving an optional *parameter* of '`no`' should omit *feature*, if it is built by default.

No '`--enable`' option should **ever** cause one feature to replace another. No '`--enable`' option should ever substitute one useful behavior for another useful behavior. The only proper use for '`--enable`' is for questions of whether to build part of the program or exclude it.

'`--with-`*package*'

The package *package* will be installed, so configure this package to work with *package*.

Possible values of *package* include '`gnu-as`' (or '`gas`'), '`gnu-ld`', '`gnu-libc`', '`gdb`', '`x`', and '`x-toolkit`'.

Do not use a '`--with`' option to specify the file name to use to find certain files. That is outside the scope of what '`--with`' options are for.

'`--nfp`' The target machine has no floating point processor.

'`--gas`' The target machine assembler is GAS, the GNU assembler. This is obsolete; users should use '`--with-gnu-as`' instead.

'`--x`' The target machine has the X Window System installed. This is obsolete; users should use '`--with-x`' instead.

All `configure` scripts should accept all of these "detail" options, whether or not they make any difference to the particular package at hand. In particular, they should accept any option that starts with '`--with-`' or '`--enable-`'. This is so users will be able to configure an entire GNU source tree at once with a single set of options.

You will note that the categories '`--with-`' and '`--enable-`' are narrow: they **do not** provide a place for any sort of option you might think of. That is deliberate. We want to limit the possible configuration options in GNU software. We do not want GNU programs to have idiosyncratic configuration options.

Packages that perform part of the compilation process may support cross-compilation. In such a case, the host and target machines for the program may be different. The `configure` script should normally treat the specified type of system as both the host and the target, thus producing a program which works for the same type of machine that it runs on.

The way to build a cross-compiler, cross-assembler, or what have you, is to specify the option '`--host=`*hosttype*' when running `configure`. This specifies the host system without changing the type of target system. The syntax for *hosttype* is the same as described above.

Bootstrapping a cross-compiler requires compiling it on a machine other than the host it will run on. Compilation packages accept a configuration option '`--build=`*hosttype*' for specifying the configuration on which you will compile them, in case that is different from the host.

Programs for which cross-operation is not meaningful need not accept the '`--host`' option, because configuring an entire operating system for cross-operation is not a meaningful thing.

Some programs have ways of configuring themselves automatically. If your program is set up to do this, your `configure` script can simply ignore most of its arguments.

7.2 Makefile Conventions

This section describes conventions for writing the Makefiles for GNU programs.

7.2.1 General Conventions for Makefiles

Every Makefile should contain this line:

```
SHELL = /bin/sh
```

to avoid trouble on systems where the `SHELL` variable might be inherited from the environment. (This is never a problem with GNU `make`.)

Different `make` programs have incompatible suffix lists and implicit rules, and this sometimes creates confusion or misbehavior. So it is a good idea to set the suffix list explicitly using only the suffixes you need in the particular Makefile, like this:

```
.SUFFIXES:
.SUFFIXES: .c .o
```

The first line clears out the suffix list, the second introduces all suffixes which may be subject to implicit rules in this Makefile.

Don't assume that '`.`' is in the path for command execution. When you need to run programs that are a part of your package during the make, please make sure that it uses '`./`' if the program is built as part of the make or '`$(srcdir)/`' if the file is an unchanging part of the source code. Without one of these prefixes, the current search path is used.

The distinction between '`./`' (the *build directory*) and '`$(srcdir)/`' (the *source directory*) is important because users can build in a separate directory using the '`--srcdir`' option to '`configure`'. A rule of the form:

```
foo.1 : foo.man sedscript
        sed -e sedscript foo.man > foo.1
```

will fail when the build directory is not the source directory, because '`foo.man`' and '`sedscript`' are in the the source directory.

When using GNU `make`, relying on '`VPATH`' to find the source file will work in the case where there is a single dependency file, since the `make` automatic variable '`$<`' will represent the source file wherever it is. (Many versions of `make` set '`$<`' only in implicit rules.) A Makefile target like

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c bar.c -o foo.o
```

should instead be written as

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c $< -o $@
```

in order to allow '`VPATH`' to work correctly. When the target has multiple dependencies, using an explicit '`$(srcdir)`' is the easiest way to make the rule work well. For example, the target above for '`foo.1`' is best written as:

```
foo.1 : foo.man sedscript
        sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

GNU distributions usually contain some files which are not source files—for example, Info files, and the output from Autoconf, Automake, Bison or Flex. Since these files normally appear in the source directory, they should always appear in the source directory, not in the build directory. So Makefile rules to update them should put the updated files in the source directory.

However, if a file does not appear in the distribution, then the Makefile should not put it in the source directory, because building a program in ordinary circumstances should not modify the source directory in any way.

Try to make the build and installation targets, at least (and all their subtargets) work correctly with a parallel `make`.

7.2.2 Utilities in Makefiles

Write the Makefile commands (and any shell scripts, such as `configure`) to run in `sh`, not in `csh`. Don't use any special features of `ksh` or `bash`.

The `configure` script and the Makefile rules for building and installation should not use any utilities directly except these:

```
cat cmp cp diff echo egrep expr false grep install-info
ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true
```

The compression program `gzip` can be used in the `dist` rule.

Stick to the generally supported options for these programs. For example, don't use '`mkdir -p`', convenient as it may be, because most systems don't support it.

It is a good idea to avoid creating symbolic links in makefiles, since a few systems don't support them.

The Makefile rules for building and installation can also use compilers and related programs, but should do so via `make` variables so that the user can substitute alternatives. Here are some of the programs we mean:

```
ar bison cc flex install ld ldconfig lex
make makeinfo ranlib texi2dvi yacc
```

Use the following `make` variables to run those programs:

```
$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LDCONFIG) $(LEX)
$(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)
```

When you use `ranlib` or `ldconfig`, you should make sure nothing bad happens if the system does not have the program in question. Arrange to ignore an error from that command, and print a message before the command to tell the user that failure of this command does not mean a problem. (The Autoconf '`AC_PROG_RANLIB`' macro can help with this.)

If you use symbolic links, you should implement a fallback for systems that don't have symbolic links.

Additional utilities that can be used via Make variables are:

```
chgrp chmod chown mknod
```

It is ok to use other utilities in Makefile portions (or scripts) intended only for particular systems where you know those utilities exist.

7.2.3 Variables for Specifying Commands

Makefiles should provide variables for overriding certain commands, options, and so on.

In particular, you should run most utility programs via variables. Thus, if you use Bison, have a variable named `BISON` whose default value is set with '`BISON = bison`', and refer to it with `$(BISON)` whenever you need to use Bison.

File management utilities such as `ln`, `rm`, `mv`, and so on, need not be referred to through variables in this way, since users don't need to replace them with other programs.

Each program-name variable should come with an options variable that is used to supply options to the program. Append '`FLAGS`' to the program-name variable name to get the options variable name—for example, `BISONFLAGS`. (The names `CFLAGS` for the C compiler, `YFLAGS` for yacc, and `LFLAGS` for lex, are exceptions to this rule, but we keep them because they are standard.) Use `CPPFLAGS` in any compilation command that runs the preprocessor, and use `LDFLAGS` in any compilation command that does linking as well as in any direct use of `ld`.

If there are C compiler options that *must* be used for proper compilation of certain files, do not include them in `CFLAGS`. Users expect to be able to specify `CFLAGS` freely themselves. Instead, arrange to pass the necessary options to the C compiler independently of `CFLAGS`, by writing them explicitly in the compilation commands or by defining an implicit rule, like this:

```
CFLAGS = -g
ALL_CFLAGS = -I. $(CFLAGS)
.c.o:
        $(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $<
```

Do include the '`-g`' option in `CFLAGS`, because that is not *required* for proper compilation. You can consider it a default that is only recommended. If the package is set up so that it is compiled with GCC by default, then you might as well include '`-O`' in the default value of `CFLAGS` as well.

Put `CFLAGS` last in the compilation command, after other variables containing compiler options, so the user can use `CFLAGS` to override the others.

CFLAGS should be used in every invocation of the C compiler, both those which do compilation and those which do linking.

Every Makefile should define the variable INSTALL, which is the basic command for installing a file into the system.

Every Makefile should also define the variables INSTALL_PROGRAM and INSTALL_DATA. (The default for each of these should be $(INSTALL).) Then it should use those variables as the commands for actual installation, for executables and nonexecutables respectively. Use these variables as follows:

```
$(INSTALL_PROGRAM) foo $(bindir)/foo
$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a
```

Always use a file name, not a directory name, as the second argument of the installation commands. Use a separate command for each file to be installed.

7.2.4 Variables for Installation Directories

Installation directories should always be named by variables, so it is easy to install in a nonstandard place. The standard names for these variables are described below. They are based on a standard filesystem layout; variants of it are used in SVR4, 4.4BSD, Linux, Ultrix v4, and other modern operating systems.

These two variables set the root for the installation. All the other installation directories should be subdirectories of one of these two, and nothing should be directly installed into these two directories.

'prefix' A prefix used in constructing the default values of the variables listed below. The default value of prefix should be '/usr/local'. When building the complete GNU system, the prefix will be empty and '/usr' will be a symbolic link to '/'. (If you are using Autoconf, write it as '@prefix@'.)

'exec_prefix'
A prefix used in constructing the default values of some of the variables listed below. The default value of exec_prefix should be $(prefix). (If you are using Autoconf, write it as '@exec_prefix@'.)

Generally, $(exec_prefix) is used for directories that contain machine-specific files (such as executables and subroutine libraries), while $(prefix) is used directly for other directories.

Executable programs are installed in one of the following directories.

'bindir' The directory for installing executable programs that users can run. This should normally be '/usr/local/bin', but write it as '$(exec_prefix)/bin'. (If you are using Autoconf, write it as '@bindir@'.)

'sbindir' The directory for installing executable programs that can be run from the shell, but are only generally useful to system administrators. This should normally be '/usr/local/sbin', but write it as '$(exec_prefix)/sbin'. (If you are using Autoconf, write it as '@sbindir@'.)

'libexecdir'
The directory for installing executable programs to be run by other programs rather than by users. This directory should normally be '/usr/local/libexec',

but write it as '$(exec_prefix)/libexec'. (If you are using Autoconf, write it as '@libexecdir@'.)

Data files used by the program during its execution are divided into categories in two ways.

- Some files are normally modified by programs; others are never normally modified (though users may edit some of these).
- Some files are architecture-independent and can be shared by all machines at a site; some are architecture-dependent and can be shared only by machines of the same kind and operating system; others may never be shared between two machines.

This makes for six different possibilities. However, we want to discourage the use of architecture-dependent files, aside from object files and libraries. It is much cleaner to make other data files architecture-independent, and it is generally not hard.

Therefore, here are the variables Makefiles should use to specify directories:

'datadir'
: The directory for installing read-only architecture independent data files. This should normally be '/usr/local/share', but write it as '$(prefix)/share'. (If you are using Autoconf, write it as '@datadir@'.) As a special exception, see '$(infodir)' and '$(includedir)' below.

'sysconfdir'
: The directory for installing read-only data files that pertain to a single machine–that is to say, files for configuring a host. Mailer and network configuration files, '/etc/passwd', and so forth belong here. All the files in this directory should be ordinary ASCII text files. This directory should normally be '/usr/local/etc', but write it as '$(prefix)/etc'. (If you are using Autoconf, write it as '@sysconfdir@'.)

  Do not install executables here in this directory (they probably belong in '$(libexecdir)' or '$(sbindir)'). Also do not install files that are modified in the normal course of their use (programs whose purpose is to change the configuration of the system excluded). Those probably belong in '$(localstatedir)'.

'sharedstatedir'
: The directory for installing architecture-independent data files which the programs modify while they run. This should normally be '/usr/local/com', but write it as '$(prefix)/com'. (If you are using Autoconf, write it as '@sharedstatedir@'.)

'localstatedir'
: The directory for installing data files which the programs modify while they run, and that pertain to one specific machine. Users should never need to modify files in this directory to configure the package's operation; put such configuration information in separate files that go in '$(datadir)' or '$(sysconfdir)'. '$(localstatedir)' should normally be '/usr/local/var', but write it as '$(prefix)/var'. (If you are using Autoconf, write it as '@localstatedir@'.)

'libdir'
: The directory for object files and libraries of object code. Do not install executables here, they probably ought to go in '$(libexecdir)' instead. The value of libdir should normally be '/usr/local/lib', but write it as '$(exec_prefix)/lib'. (If you are using Autoconf, write it as '@libdir@'.)

'infodir' The directory for installing the Info files for this package. By default, it should be '/usr/local/info', but it should be written as '$(prefix)/info'. (If you are using Autoconf, write it as '@infodir@'.)

'lispdir' The directory for installing any Emacs Lisp files in this package. By default, it should be '/usr/local/share/emacs/site-lisp', but it should be written as '$(prefix)/share/emacs/site-lisp'.

If you are using Autoconf, write the default as '@lispdir@'. In order to make '@lispdir@' work, you need the following lines in your 'configure.in' file:

```
lispdir='${datadir}/emacs/site-lisp'
AC_SUBST(lispdir)
```

'includedir'

The directory for installing header files to be included by user programs with the C '#include' preprocessor directive. This should normally be '/usr/local/include', but write it as '$(prefix)/include'. (If you are using Autoconf, write it as '@includedir@'.)

Most compilers other than GCC do not look for header files in directory '/usr/local/include'. So installing the header files this way is only useful with GCC. Sometimes this is not a problem because some libraries are only really intended to work with GCC. But some libraries are intended to work with other compilers. They should install their header files in two places, one specified by `includedir` and one specified by `oldincludedir`.

'oldincludedir'

The directory for installing '#include' header files for use with compilers other than GCC. This should normally be '/usr/include'. (If you are using Autoconf, you can write it as '@oldincludedir@'.)

The Makefile commands should check whether the value of `oldincludedir` is empty. If it is, they should not try to use it; they should cancel the second installation of the header files.

A package should not replace an existing header in this directory unless the header came from the same package. Thus, if your Foo package provides a header file 'foo.h', then it should install the header file in the `oldincludedir` directory if either (1) there is no 'foo.h' there or (2) the 'foo.h' that exists came from the Foo package.

To tell whether 'foo.h' came from the Foo package, put a magic string in the file—part of a comment—and `grep` for that string.

Unix-style man pages are installed in one of the following:

'mandir' The top-level directory for installing the man pages (if any) for this package. It will normally be '/usr/local/man', but you should write it as '$(prefix)/man'. (If you are using Autoconf, write it as '@mandir@'.)

'man1dir' The directory for installing section 1 man pages. Write it as '$(mandir)/man1'.

'man2dir' The directory for installing section 2 man pages. Write it as '$(mandir)/man2'

'...'

Don't make the primary documentation for any GNU software be a man page. Write a manual in Texinfo instead. Man pages are just for the sake of people running GNU software on Unix, which is a secondary application only.

'manext' The file name extension for the installed man page. This should contain a period followed by the appropriate digit; it should normally be '.1'.

'man1ext' The file name extension for installed section 1 man pages.

'man2ext' The file name extension for installed section 2 man pages.

'...' Use these names instead of 'manext' if the package needs to install man pages in more than one section of the manual.

And finally, you should set the following variable:

'srcdir' The directory for the sources being compiled. The value of this variable is normally inserted by the `configure` shell script. (If you are using Autconf, use 'srcdir = @srcdir@'.)

For example:

```
# Common prefix for installation directories.
# NOTE: This directory must exist when you start the install.
prefix = /usr/local
exec_prefix = $(prefix)
# Where to put the executable for the command 'gcc'.
bindir = $(exec_prefix)/bin
# Where to put the directories used by the compiler.
libexecdir = $(exec_prefix)/libexec
# Where to put the Info files.
infodir = $(prefix)/info
```

If your program installs a large number of files into one of the standard user-specified directories, it might be useful to group them into a subdirectory particular to that program. If you do this, you should write the `install` rule to create these subdirectories.

Do not expect the user to include the subdirectory name in the value of any of the variables listed above. The idea of having a uniform set of variable names for installation directories is to enable the user to specify the exact same values for several different GNU packages. In order for this to be useful, all the packages must be designed so that they will work sensibly when the user does so.

7.2.5 Standard Targets for Users

All GNU programs should have the following targets in their Makefiles:

'all' Compile the entire program. This should be the default target. This target need not rebuild any documentation files; Info files should normally be included in the distribution, and DVI files should be made only when explicitly asked for.

By default, the Make rules should compile and link with '-g', so that executable programs have debugging symbols. Users who don't mind being helpless can strip the executables later if they wish.

'install' Compile the program and copy the executables, libraries, and so on to the file names where they should reside for actual use. If there is a simple test to verify that a program is properly installed, this target should run that test.

Do not strip executables when installing them. Devil-may-care users can use the install-strip target to do that.

If possible, write the install target rule so that it does not modify anything in the directory where the program was built, provided 'make all' has just been done. This is convenient for building the program under one user name and installing it under another.

The commands should create all the directories in which files are to be installed, if they don't already exist. This includes the directories specified as the values of the variables prefix and exec_prefix, as well as all subdirectories that are needed. One way to do this is by means of an installdirs target as described below.

Use '-' before any command for installing a man page, so that make will ignore any errors. This is in case there are systems that don't have the Unix man page documentation system installed.

The way to install Info files is to copy them into '$(infodir)' with $(INSTALL_DATA) (see Section 7.2.3 [Command Variables], page 43), and then run the install-info program if it is present. install-info is a program that edits the Info 'dir' file to add or update the menu entry for the given Info file; it is part of the Texinfo package. Here is a sample rule to install an Info file:

```
$(infodir)/foo.info: foo.info
        $(POST_INSTALL)
# There may be a newer info file in . than in srcdir.
        -if test -f foo.info; then d=.; \
         else d=$(srcdir); fi; \
        $(INSTALL_DATA) $$d/foo.info $@; \
# Run install-info only if it exists.
# Use 'if' instead of just prepending '-' to the
# line so we notice real errors from install-info.
# We use '$(SHELL) -c' because some shells do not
# fail gracefully when there is an unknown command.
        if $(SHELL) -c 'install-info --version' \
           >/dev/null 2>&1; then \
          install-info --dir-file=$(infodir)/dir \
                       $(infodir)/foo.info; \
        else true; fi
```

When writing the install target, you must classify all the commands into three categories: normal ones, *pre-installation* commands and *post-installation* commands. See Section 7.2.6 [Install Command Categories], page 51.

'uninstall'

Delete all the installed files—the copies that the 'install' target creates.

This rule should not modify the directories where compilation is done, only the directories where files are installed.

The uninstallation commands are divided into three categories, just like the installation commands. See Section 7.2.6 [Install Command Categories], page 51.

'install-strip'
: Like `install`, but strip the executable files while installing them. In many cases, the definition of this target can be very simple:

```
install-strip:
        $(MAKE) INSTALL_PROGRAM='$(INSTALL_PROGRAM) -s' \
                install
```

Normally we do not recommend stripping an executable unless you are sure the program has no bugs. However, it can be reasonable to install a stripped executable for actual execution while saving the unstripped executable elsewhere in case there is a bug.

'clean'
: Delete all files from the current directory that are normally created by building the program. Don't delete the files that record the configuration. Also preserve files that could be made by building, but normally aren't because the distribution comes with them.

Delete '.dvi' files here if they are not part of the distribution.

'distclean'
: Delete all files from the current directory that are created by configuring or building the program. If you have unpacked the source and built the program without creating any other files, 'make distclean' should leave only the files that were in the distribution.

'mostlyclean'
: Like 'clean', but may refrain from deleting a few files that people normally don't want to recompile. For example, the 'mostlyclean' target for GCC does not delete 'libgcc.a', because recompiling it is rarely necessary and takes a lot of time.

'maintainer-clean'
: Delete almost everything from the current directory that can be reconstructed with this Makefile. This typically includes everything deleted by `distclean`, plus more: C source files produced by Bison, tags tables, Info files, and so on.

The reason we say "almost everything" is that running the command 'make maintainer-clean' should not delete 'configure' even if 'configure' can be remade using a rule in the Makefile. More generally, 'make maintainer-clean' should not delete anything that needs to exist in order to run 'configure' and then begin to build the program. This is the only exception; `maintainer-clean` should delete everything else that can be rebuilt.

The 'maintainer-clean' target is intended to be used by a maintainer of the package, not by ordinary users. You may need special tools to reconstruct some of the files that 'make maintainer-clean' deletes. Since these files are normally included in the distribution, we don't take care to make them easy to reconstruct. If you find you need to unpack the full distribution again, don't blame us.

To help make users aware of this, the commands for the special `maintainer-clean` target should start with these two:

```
@echo 'This command is intended for maintainers to use; it'
@echo 'deletes files that may need special tools to rebuild.'
```

'TAGS' Update a tags table for this program.

'info' Generate any Info files needed. The best way to write the rules is as follows:

```
info: foo.info

foo.info: foo.texi chap1.texi chap2.texi
        $(MAKEINFO) $(srcdir)/foo.texi
```

You must define the variable `MAKEINFO` in the Makefile. It should run the `makeinfo` program, which is part of the Texinfo distribution.

Normally a GNU distribution comes with Info files, and that means the Info files are present in the source directory. Therefore, the Make rule for an info file should update it in the source directory. When users build the package, ordinarily Make will not update the Info files because they will already be up to date.

'dvi' Generate DVI files for all Texinfo documentation. For example:

```
dvi: foo.dvi

foo.dvi: foo.texi chap1.texi chap2.texi
        $(TEXI2DVI) $(srcdir)/foo.texi
```

You must define the variable `TEXI2DVI` in the Makefile. It should run the program `texi2dvi`, which is part of the Texinfo distribution.[1] Alternatively, write just the dependencies, and allow GNU `make` to provide the command.

'dist' Create a distribution tar file for this program. The tar file should be set up so that the file names in the tar file start with a subdirectory name which is the name of the package it is a distribution for. This name can include the version number.

For example, the distribution tar file of GCC version 1.40 unpacks into a subdirectory named 'gcc-1.40'.

The easiest way to do this is to create a subdirectory appropriately named, use `ln` or `cp` to install the proper files in it, and then `tar` that subdirectory.

Compress the tar file file with `gzip`. For example, the actual distribution file for GCC version 1.40 is called 'gcc-1.40.tar.gz'.

The `dist` target should explicitly depend on all non-source files that are in the distribution, to make sure they are up to date in the distribution. See Section 7.3 [Making Releases], page 53.

'check' Perform self-tests (if any). The user must build the program before running the tests, but need not install the program; you should write the self-tests so that they work when the program is built but not installed.

---

[1] `texi2dvi` uses TeX to do the real work of formatting. TeX is not distributed with Texinfo.

The following targets are suggested as conventional names, for programs in which they are useful.

`installcheck`
: Perform installation tests (if any). The user must build and install the program before running the tests. You should not assume that '`$(bindir)`' is in the search path.

`installdirs`
: It's useful to add a target named '`installdirs`' to create the directories where files are installed, and their parent directories. There is a script called '`mkinstalldirs`' which is convenient for this; you can find it in the Texinfo package. You can use a rule like this:

```
# Make sure all installation directories (e.g. $(bindir))
# actually exist by making them if necessary.
installdirs: mkinstalldirs
        $(srcdir)/mkinstalldirs $(bindir) $(datadir) \
                                $(libdir) $(infodir) \
                                $(mandir)
```

This rule should not modify the directories where compilation is done. It should do nothing but create installation directories.

7.2.6 Install Command Categories

When writing the `install` target, you must classify all the commands into three categories: normal ones, *pre-installation* commands and *post-installation* commands.

Normal commands move files into their proper places, and set their modes. They may not alter any files except the ones that come entirely from the package they belong to.

Pre-installation and post-installation commands may alter other files; in particular, they can edit global configuration files or data bases.

Pre-installation commands are typically executed before the normal commands, and post-installation commands are typically run after the normal commands.

The most common use for a post-installation command is to run `install-info`. This cannot be done with a normal command, since it alters a file (the Info directory) which does not come entirely and solely from the package being installed. It is a post-installation command because it needs to be done after the normal command which installs the package's Info files.

Most programs don't need any pre-installation commands, but we have the feature just in case it is needed.

To classify the commands in the `install` rule into these three categories, insert *category lines* among them. A category line specifies the category for the commands that follow.

A category line consists of a tab and a reference to a special Make variable, plus an optional comment at the end. There are three variables you can use, one for each category; the variable name specifies the category. Category lines are no-ops in ordinary execution because these three Make variables are normally undefined (and you *should not* define them in the makefile).

Here are the three possible category lines, each with a comment that explains what it means:

```
$(PRE_INSTALL)     # Pre-install commands follow.
$(POST_INSTALL)    # Post-install commands follow.
$(NORMAL_INSTALL)  # Normal commands follow.
```

If you don't use a category line at the beginning of the `install` rule, all the commands are classified as normal until the first category line. If you don't use any category lines, all the commands are classified as normal.

These are the category lines for `uninstall`:

```
$(PRE_UNINSTALL)     # Pre-uninstall commands follow.
$(POST_UNINSTALL)    # Post-uninstall commands follow.
$(NORMAL_UNINSTALL)  # Normal commands follow.
```

Typically, a pre-uninstall command would be used for deleting entries from the Info directory.

If the `install` or `uninstall` target has any dependencies which act as subroutines of installation, then you should start *each* dependency's commands with a category line, and start the main target's commands with a category line also. This way, you can ensure that each command is placed in the right category regardless of which of the dependencies actually run.

Pre-installation and post-installation commands should not run any programs except for these:

```
[ basename bash cat chgrp chmod chown cmp cp dd diff echo
egrep expand expr false fgrep find getopt grep gunzip gzip
hostname install install-info kill ldconfig ln ls md5sum
mkdir mkfifo mknod mv printenv pwd rm rmdir sed sort tee
test touch true uname xargs yes
```

The reason for distinguishing the commands in this way is for the sake of making binary packages. Typically a binary package contains all the executables and other files that need to be installed, and has its own method of installing them—so it does not need to run the normal installation commands. But installing the binary package does need to execute the pre-installation and post-installation commands.

Programs to build binary packages work by extracting the pre-installation and post-installation commands. Here is one way of extracting the pre-installation commands:

```
make -n install -o all \
      PRE_INSTALL=pre-install \
      POST_INSTALL=post-install \
      NORMAL_INSTALL=normal-install \
  | gawk -f pre-install.awk
```

where the file '`pre-install.awk`' could contain this:

```
$0 ~ /^\t[ \t]*(normal_install|post_install)[ \t]*$/ {on = 0}
on {print $0}
$0 ~ /^\t[ \t]*pre_install[ \t]*$/ {on = 1}
```

The resulting file of pre-installation commands is executed as a shell script as part of installing the binary package.

7.3 Making Releases

Package the distribution of `Foo version 69.96` up in a gzipped tar file with the name '`foo-69.96.tar.gz`'. It should unpack into a subdirectory named '`foo-69.96`'.

Building and installing the program should never modify any of the files contained in the distribution. This means that all the files that form part of the program in any way must be classified into *source files* and *non-source files*. Source files are written by humans and never changed automatically; non-source files are produced from source files by programs under the control of the Makefile.

Naturally, all the source files must be in the distribution. It is okay to include non-source files in the distribution, provided they are up-to-date and machine-independent, so that building the distribution normally will never modify them. We commonly include non-source files produced by Bison, `lex`, TeX, and `makeinfo`; this helps avoid unnecessary dependencies between our distributions, so that users can install whichever packages they want to install.

Non-source files that might actually be modified by building and installing the program should **never** be included in the distribution. So if you do distribute non-source files, always make sure they are up to date when you make a new distribution.

Make sure that the directory into which the distribution unpacks (as well as any subdirectories) are all world-writable (octal mode 777). This is so that old versions of `tar` which preserve the ownership and permissions of the files from the tar archive will be able to extract all the files even if the user is unprivileged.

Make sure that all the files in the distribution are world-readable.

Make sure that no file name in the distribution is more than 14 characters long. Likewise, no file created by building the program should have a name longer than 14 characters. The reason for this is that some systems adhere to a foolish interpretation of the POSIX standard, and refuse to open a longer name, rather than truncating as they did in the past.

Don't include any symbolic links in the distribution itself. If the tar file contains symbolic links, then people cannot even unpack it on systems that don't support symbolic links. Also, don't use multiple names for one file in different directories, because certain file systems cannot handle this and that prevents unpacking the distribution.

Try to make sure that all the file names will be unique on MS-DOS. A name on MS-DOS consists of up to 8 characters, optionally followed by a period and up to three characters. MS-DOS will truncate extra characters both before and after the period. Thus, '`foobarhacker.c`' and '`foobarhacker.o`' are not ambiguous; they are truncated to '`foobarha.c`' and '`foobarha.o`', which are distinct.

Include in your distribution a copy of the '`texinfo.tex`' you used to test print any '`*.texinfo`' or '`*.texi`' files.

Likewise, if your program uses small GNU software packages like regex, getopt, obstack, or termcap, include them in the distribution file. Leaving them out would make the distribution file a little smaller at the expense of possible inconvenience to a user who doesn't know what other files to get.

Table of Contents