GNU Coding Standards

Richard Stallman
last updated 28 March 1994

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 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, 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 change, we need legal papers for it.

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.

I 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.

3 Change Logs

Keep a change log for each directory, describing the changes made to source files in that directory. 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 eliminate conceptual inconsistencies between different parts of a program; they can give you a history of how the conflicting concepts arose.

Use the Emacs command `M-x add-change` to start a new entry in the change log. 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.

Separate unrelated 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.

Here are some examples:

```
* 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 them; 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 the entry.

There's no need to describe the full purpose of the changes or how they work together. It is better to put such explanations in comments in the code. That's why just "New function" is enough; there is a comment with the function in the source to explain what it does.

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

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.

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. Just write in the entry for the function being called, "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. Write just, "Doc fix." There's no need to keep a change log 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.

4 Compatibility with Other Implementations

With certain 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`' 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. Additional programs with no counterpart in Unix may be useful, but our first priority is usually to duplicate what Unix already has.

5 Makefile Conventions

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

5.1 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.)

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 './' and '$(srcdir)/' is important when 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 current directory is not the source directory, because 'foo.man' and 'sedscript' are not in the current 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) $(CFLAGS) $< -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 -s $(srcdir)/sedscript $(srcdir)/foo.man > foo.1
```

5.2 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 echo egrep expr grep
ln mkdir mv pwd rm rmdir sed test touch
```

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.

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 lex
make makeinfo ranlib texi2dvi yacc
```

When you use `ranlib`, you should test whether it exists, and run it only if it exists, so that the distribution will work on systems that don't have `ranlib`.

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

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

5.3 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.

'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.

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 5.4 [Command Variables], page 9), and then run the `install-info` program if it is present. `install-info` is a script that edits the Info 'dir' file to add or update the menu entry for the given Info file; it will be part of the Texinfo package. Here is a sample rule to install an Info file:

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

'uninstall'

Delete all the installed files that the 'install' target would create (but not the noninstalled files such as 'make all' would create).

'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.

'realclean'
: Delete 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.

  One exception, however: 'make realclean' should not delete 'configure' even if 'configure' can be remade using a rule in the Makefile. More generally, 'make realclean' should not delete anything that needs to exist in order to run 'configure' and then begin to build the program.

'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.

'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. 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.

  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 "Making Releases" in *GNU Coding Standards*.

'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.

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; 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)
```

5.4 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 name CFLAGS is an exception to this rule, but we keep it because it is 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.

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.

5.5 Variables for Installation Directories

Installation directories should always be named by variables, so it is easy to install in a non-standard place. The standard names for these variables are:

'prefix' A prefix used in constructing the default values of the variables listed below. The default value of prefix should be '/usr/local' (at least for now).

'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).

  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.

'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'.

'libdir'
: The directory for installing executable files to be run by the program rather than by users. Object files and libraries of object code should also go in this directory. The idea is that this directory is used for files that pertain to a specific machine architecture, but need not be in the path for commands. The value of libdir should normally be '/usr/local/lib', but write it as '$(exec_prefix)/lib'.

'datadir'
: The directory for installing read-only data files which the programs refer to while they run. This directory is used for files which are independent of the type of machine being used. This should normally be '/usr/local/lib', but write it as '$(prefix)/lib'.

'statedir'
: The directory for installing data files which the programs modify while they run. These files should be independent of the type of machine being used, and it should be possible to share them among machines at a network installation. This should normally be '/usr/local/lib', but write it as '$(prefix)/lib'.

'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'.

  Most compilers other than GCC do not look for header files in '/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'.

  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.

'mandir' The directory for installing the man pages (if any) for this package. It should include the suffix for the proper section of the manual—usually '1' for a utility. It will normally be '/usr/local/man/man1', but you should write it as '$(prefix)/man/man1'.

'man1dir' The directory for installing section 1 man pages.

'man2dir' The directory for installing section 2 man pages.

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

'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.

'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'.

'srcdir' The directory for the sources being compiled. The value of this variable is normally inserted by the configure shell script.

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.
libdir = $(exec_prefix)/lib
# 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.

6 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, 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. So would '`sun3-bsd4.2`', since SunOS is basically BSD and no other BSD system is used on a Sun. 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 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 '`x`', '`x-toolkit`', '`gnu-as`' (or '`gas`'), '`gnu-ld`', '`gnu-libc`', and '`gdb`'.

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 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 compilation 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 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. So please write in C.

There are three exceptions for this rule:

- It is okay to use a special language if the same program contains an interpreter for that language.

  Thus, it is not a problem that GNU Emacs contains code written in Emacs Lisp, because it comes with a Lisp interpreter.
- 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 not of extremely widespread interest, then perhaps it's not important if the application is inconvenient to install.

8 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.

9 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 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 identifer 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 */
```

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

```
#ifndef foo
  ...
#else /* foo */
  ...
#endif /* foo */
```

10 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.

11 Naming Variables and Functions

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 System V. 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.

12 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.

Since most computer systems do not yet implement ANSI C, using the ANSI C features is effectively using a GNU extension, so the same considerations apply. (Except for ANSI features that we discourage, such as trigraphs—don't ever use them.)

13 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.
- 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. The problem 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 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`'; other 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 are off-limits 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 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`.) 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.

14 Program Behavior for All Programs

Avoid arbitrary limits on the length or number of *any* data structure, including filenames, 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`.

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.

By default, the GNU system will provide the signal handling functions of BSD and of POSIX. So GNU software should be written to use these.

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.

15 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 filename. 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.

16 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.

17 Portability As It Applies to GNU

Much of what is called "portability" in the Unix world refers to porting to different Unix versions. This is a secondary consideration for GNU software, because its primary purpose is to run on top of one and only one kernel, the GNU kernel, compiled with one and only one C compiler, the GNU C compiler. The amount and kinds of variation among GNU systems on different cpu's will be like the variation among Berkeley 4.3 systems on different cpu's.

All users today run GNU software on non-GNU systems. So supporting a variety of non-GNU systems is desirable; simply not paramount. The easiest way to achieve portability to a reasonable range of systems is to use Autoconf. It's unlikely that your program needs to know more information about the host machine than Autoconf can provide, simply because most of the programs that need such knowledge have already been written.

It is difficult to be sure exactly what facilities the GNU kernel will provide, since it isn't finished yet. Therefore, assume you can use anything in 4.3; just avoid using the format of semi-internal data bases (e.g., directories) when there is a higher-level alternative (`readdir`).

You can freely assume any reasonably standard facilities in the C language, libraries or kernel, because we will find it necessary to support these facilities in the full GNU system, whether or not we have already done so. The fact that there may exist kernels or C compilers that lack these facilities is irrelevant as long as the GNU kernel and C compiler support them.

It remains necessary to worry about differences among cpu types, such as the difference in byte ordering and alignment restrictions. It's unlikely that 16-bit machines will ever be supported by GNU, so there is no point in spending any time to consider the possibility that an int will be less than 32 bits.

You can assume that all pointers have the same format, regardless of the type they point to, and that this is really an integer. There are some weird machines where this isn't true, but they aren't important; don't waste time catering to them. Besides, eventually we will put function prototypes into all GNU programs, and that will probably make your program work even on weird machines.

Since some important machines (including the 68000) are big-endian, it is important not to assume that the address of an `int` object is also the address of its least-significant byte. Thus, don't make the following mistake:

```
int c;
...
```

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

You can assume that it is reasonable to use a meg of memory. Don't strain to reduce memory usage unless it can get to that level. If your program creates complicated data structures, just make them in core and give a fatal error if malloc returns zero.

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.

18 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.

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. The table appears below.

If you use names not already in the table, please send '`gnu@prep.ai.mit.edu`' a list of them, with their meanings, so we can update the table.

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`'). Even if you allow an output file name as an ordinary argument for compatibility, try to provide a suitable option as well. This will lead to more consistency among GNU utilities, so that there are fewer idiosyncracies for users to remember.

Programs should support an option '`--version`' which prints the program's version number on standard output and exits successfully, and an option '`--help`' which prints option usage information on standard output and exits successfully. These options should inhibit the normal function of the command; they should do nothing except print the requested information.

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

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

'`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`.

'arglength'
: '-l' in m4.

'ascii'
: '-a' in diff.

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

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

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

'batch'
: Used in GDB.

'baud'
: Used in GDB.

'before'
: '-b' in tac.

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

'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.

'compress'
: '-Z' in tar.

'concatenate'
: '-A' in tar.

'confirmation'
: '-w' in tar.

'context' Used in diff.

'copyright'
: '-C' in ptx and recode.

'core' Used in GDB.

'count' '-q' in who.

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

'create' Used in tar and cpio.

'cxref' '-x' in etags.

'date' '-d' in touch.

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

'define' '-D' in m4.

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

'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.

'diversions'
'-N' in m4.

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

'ed' '-e' in diff.

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

'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.

'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.

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

'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.

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

'forward-search'
: '-F' in etags.

'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.

'hashsize'
: '-H' in m4.

'header' '-h' in objdump and recode

'heading' '-H' in who.

'help' Used to ask for brief usage information.

'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.

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

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

'ignore-indentation'
: '-S' 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.

'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.

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

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

'link' '-l' in cpio.

'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@prep.ai.mit.edu.

'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.

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

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

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

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

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

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

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

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

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

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

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

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

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

'no-split'
: Used in Makeinfo.

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

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

'no-validate'
Used in Makeinfo.

'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.

'override'
'-o' in rm.

'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.

'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.

'question'
'-q' in Make.

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

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

'rcs' '-n' in diff.

'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.

'release' '-r' in uname.

'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. every program accepting '--silent' should accept '--quiet' as a synonym.

'size' '-s' in ls.

'sort' Used in ls.

'sparse' '-S' in tar.

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

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

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

'stop' '-S' in Make.

'strict' '-s' in recode.

'strip' '-s' in install.

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

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

'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.

'text' '-a' in diff.

'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; '-G' in m4 and ptx.

'tty' Used in GDB.

'typedefs'
: '-t' in etags.

'typedefs-and-c++'
: '-T' in etags.

'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, 'etags', 'mv', 'tar'.

'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 etags.

'volume' '-V' in tar.

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

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

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

'writable'
: '-T' in who.

'zeros' '-z' in gprof.

19 Documenting Programs

Please use Texinfo for documenting GNU programs. See the Texinfo manual, either the hardcopy or the version in the GNU Emacs Info subsystem (C-h i). See existing GNU Texinfo files (e.g., those under the 'man/' directory in the GNU Emacs distribution) for examples.

The title page of the manual should state the version of the program which the manual applies to. 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.

The manual should document all command-line arguments and all commands. It should give examples of their use. But don't organize the manual as a list of features. Instead, organize it by the concepts a user will have before reaching that point in the manual. Address the goals that a user will have in mind, and explain how to accomplish them. Don't use Unix man pages as a model for how to write GNU documentation; they are a bad example to follow.

The manual should have a node named '*program* `Invocation`' or '`Invoking` *program*', where *program* stands for the name of the program being described, as you would type it in the shell to run the 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.

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.

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.

It is ok to supply a man page for the program as well as a Texinfo manual if you wish to. But keep in mind that supporting a man page requires continual effort, each time the program is changed. Any time you spend on the man page is time taken away from more useful things you could contribute.

Thus, even if a user volunteers to donate a man page, you may find this gift costly to accept. Unless you have time on your hands, it may be better to refuse the man page unless the same volunteer agrees to take full responsibility for maintaining it—so that you can wash your hands of it entirely. If the volunteer ceases to do the job, then don't feel obliged to pick it up yourself; it may be better to withdraw the man page until another volunteer offers to carry on with it.

Alternatively, if you expect the discrepancies to be small enough that the man page remains useful, 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, and describing how to access the Texinfo documentation.

20 Making Releases

Package the distribution of Foo version 69.96 in a gzipped tar file named '`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 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-DOG. A name on MS-DOG consists of up to 8 characters, optionally followed by a period and up to three characters. MS-DOG 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`' 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