GNU textutils

A set of text utilities
for version 2.0, 31 July 1999

David MacKenzie et al.

Copyright © 1994, 95, 96 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Foundation.

1 Introduction

This manual is incomplete: No attempt is made to explain basic concepts in a way suitable for novices. Thus, if you are interested, please get involved in improving this manual. The entire GNU community will benefit.

The GNU text utilities are mostly compatible with the POSIX.2 standard.

Please report bugs to `bug-textutils@gnu.org`. Remember to include the version number, machine architecture, input files, and any other information needed to reproduce the bug: your input, what you expected, what you got, and why it is wrong. Diffs are welcome, but please include a description of the problem as well, since this is sometimes difficult to infer. See section "Bugs" in *GNU CC*.

This manual was originally derived from the Unix man pages in the distribution, which were written by David MacKenzie and updated by Jim Meyering. What you are reading now is the authoritative documentation for these utilities; the man pages are no longer being maintained. The original `fmt` man page was written by Ross Paterson. François Pinard did the initial conversion to Texinfo format. Karl Berry did the indexing, some reorganization, and editing of the results. Richard Stallman contributed his usual invaluable insights to the overall process.

2 Common options

Certain options are available in all these programs. Rather than writing identical descriptions for each of the programs, they are described here. (In fact, every GNU program accepts (or should accept) these options.)

A few of these programs take arbitrary strings as arguments. In those cases, '`--help`' and '`--version`' are taken as these options only if there is one and exactly one command line argument.

'`--help`' Print a usage message listing all available options, then exit successfully.

'`--version`'
Print the version number, then exit successfully.

3 Output of entire files

These commands read and write entire files, possibly transforming them in some way.

3.1 `cat`: Concatenate and write files

`cat` copies each *file* ('`-`' means standard input), or standard input if none are given, to standard output. Synopsis:

```
cat [option] [file]...
```

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-A`'
'`--show-all`'
: Equivalent to '`-vET`'.

'`-B`'
'`--binary`'
: On MS-DOS and MS-Windows only, read and write the files in binary mode. By default, `cat` on MS-DOS/MS-Windows uses binary mode only when standard output is redirected to a file or a pipe; this option overrides that. Binary file I/O is used so that the files retain their format (Unix text as opposed to DOS text and binary), because `cat` is frequently used as a file-copying program. Some options (see below) cause `cat` read and write files in text mode because then the original file contents aren't important (e.g., when lines are numbered by `cat`, or when line endings should be marked). This is so these options work as DOS/Windows users would expect; for example, DOS-style text files have their lines end with the CR-LF pair of characters which won't be processed as an empty line by '`-b`' unless the file is read in text mode.

'`-b`'
'`--number-nonblank`'
: Number all nonblank output lines, starting with 1. On MS-DOS and MS-Windows, this option causes `cat` to read and write files in text mode.

'`-e`'
: Equivalent to '`-vE`'.

'`-E`'
'`--show-ends`'
: Display a '`$`' after the end of each line. On MS-DOS and MS-Windows, this option causes `cat` to read and write files in text mode.

'`-n`'
'`--number`'
: Number all output lines, starting with 1. On MS-DOS and MS-Windows, this option causes `cat` to read and write files in text mode.

'`-s`'
'`--squeeze-blank`'
: Replace multiple adjacent blank lines with a single blank line. On MS-DOS and MS-Windows, this option causes `cat` to read and write files in text mode.

'`-t`' Equivalent to '`-vT`'.

'`-T`'
'`--show-tabs`'
Display TAB characters as '`^I`'.

'`-u`' Ignored; for Unix compatibility.

'`-v`'
'`--show-nonprinting`'
Display control characters except for LFD and TAB using '`^`' notation and precede characters that have the high bit set with '`M-`'. On MS-DOS and MS-Windows, this option causes `cat` to read files and standard input in DOS binary mode, so the CR characters at the end of each line are also visible.

3.2 `tac`: Concatenate and write files in reverse

`tac` copies each *file* ('`-`' means standard input), or standard input if none are given, to standard output, reversing the records (lines by default) in each separately. Synopsis:

`tac` [*option*]... [*file*]...

Records are separated by instances of a string (newline by default). By default, this separator string is attached to the end of the record that it follows in the file.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-b`'
'`--before`'
The separator is attached to the beginning of the record that it precedes in the file.

'`-r`'
'`--regex`' Treat the separator string as a regular expression. Users of `tac` on MS-DOS/MS-Windows should note that, since `tac` reads files in binary mode, each line of a text file might end with a CR/LF pair instead of the Unix-style LF.

'`-s` *separator*'
'`--separator=`*separator*'
Use *separator* as the record separator, instead of newline.

3.3 `nl`: Number lines and write files

`nl` writes each *file* ('`-`' means standard input), or standard input if none are given, to standard output, with line numbers added to some or all of the lines. Synopsis:

`nl` [*option*]... [*file*]...

`nl` decomposes its input into (logical) pages; by default, the line number is reset to 1 at the top of each logical page. `nl` treats all of the input files as a single document; it does not reset line numbers or logical pages between files.

A logical page consists of three sections: header, body, and footer. Any of the sections can be empty. Each can be numbered in a different style from the others.

The beginnings of the sections of logical pages are indicated in the input file by a line containing exactly one of these delimiter strings:

'`\:\:\:`' start of header;

'`\:\:`' start of body;

'`\:`' start of footer.

The two characters from which these strings are made can be changed from '`\`' and '`:`' via options (see below), but the pattern and length of each string cannot be changed.

A section delimiter is replaced by an empty line on output. Any text that comes before the first section delimiter string in the input file is considered to be part of a body section, so `nl` treats a file that contains no section delimiters as a single body section.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-b` *style*'
'`--body-numbering=`*style*'
: Select the numbering style for lines in the body section of each logical page. When a line is not numbered, the current line number is not incremented, but the line number separator character is still prepended to the line. The styles are:

  '`a`' number all lines,

  '`t`' number only nonempty lines (default for body),

  '`n`' do not number lines (default for header and footer),

  '`p`*regexp*' number only lines that contain a match for *regexp*.

'`-d` *cd*'
'`--section-delimiter=`*cd*'
: Set the section delimiter characters to *cd*; default is '`\:`'. If only *c* is given, the second remains '`:`'. (Remember to protect '`\`' or other metacharacters from shell expansion with quotes or extra backslashes.)

'`-f` *style*'
'`--footer-numbering=`*style*'
: Analogous to '`--body-numbering`'.

'`-h` *style*'
'`--header-numbering=`*style*'
: Analogous to '`--body-numbering`'.

'`-i` *number*'
'`--page-increment=`*number*'
: Increment line numbers by *number* (default 1).

'`-l` *number*'
'`--join-blank-lines=`*number*'
: Consider *number* (default 1) consecutive empty lines to be one logical line for numbering, and only number the last one. Where fewer than *number* consecutive empty lines occur, do not number them. An empty line is one that contains no characters, not even spaces or tabs.

'`-n` *format*'
'`--number-format=`*format*'
: Select the line numbering format (default is `rn`):

  '`ln`' left justified, no leading zeros;

  '`rn`' right justified, no leading zeros;

  '`rz`' right justified, leading zeros.

'`-p`'
'`--no-renumber`'
: Do not reset the line number at the start of a logical page.

'`-s` *string*'
'`--number-separator=`*string*'
: Separate the line number from the text line in the output with *string* (default is the TAB character).

'`-v` *number*'
'`--starting-line-number=`*number*'
: Set the initial line number on each logical page to *number* (default 1).

'`-w` *number*'
'`--number-width=`*number*'
: Use *number* characters for line numbers (default 6).

3.4 `od`: Write files in octal or other formats

`od` writes an unambiguous representation of each *file* ('`-`' means standard input), or standard input if none are given. Synopsis:

```
od [option]... [file]...
od -C [file] [[+]offset [[+]label]]
```

Each line of output consists of the offset in the input, followed by groups of data from the file. By default, `od` prints the offset in octal, and each group of file data is two bytes of input printed as a single octal number.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-A` *radix*'
'`--address-radix=`*radix*'
: Select the base in which file offsets are printed. *radix* can be one of the following:

  '`d`' decimal;

  '`o`' octal;

'`x`' hexadecimal;

'`n`' none (do not print offsets).

The default is octal.

'`-j` *bytes*'
'`--skip-bytes=`*bytes*'
Skip *bytes* input bytes before formatting and writing. If *bytes* begins with '`0x`' or '`0X`', it is interpreted in hexadecimal; otherwise, if it begins with '`0`', in octal; otherwise, in decimal. Appending '`b`' multiplies *bytes* by 512, '`k`' by 1024, and '`m`' by 1048576.

'`-N` *bytes*'
'`--read-bytes=`*bytes*'
Output at most *bytes* bytes of the input. Prefixes and suffixes on `bytes` are interpreted as for the '`-j`' option.

'`-s [`*n*`]`'
'`--strings[=`*n*`]`'
Instead of the normal output, output only *string constants*: at least *n* (3 by default) consecutive ASCII graphic characters, followed by a null (zero) byte.

'`-t` *type*'
'`--format=`*type*'
Select the format in which to output the file data. *type* is a string of one or more of the below type indicator characters. If you include more than one type indicator character in a single *type* string, or use this option more than once, `od` writes one copy of each output line using each of the data types that you specified, in the order that you specified.

Adding a trailing "z" to any type specification appends a display of the ASCII character representation of the printable characters to the output line generated by the type specification.

'`a`' named character,

'`c`' ASCII character or backslash escape,

'`d`' signed decimal,

'`f`' floating point,

'`o`' octal,

'`u`' unsigned decimal,

'`x`' hexadecimal.

The type `a` outputs things like '`sp`' for space, '`nl`' for newline, and '`nul`' for a null (zero) byte. Type `c` outputs ' ', '`\n`', and `\0`, respectively.

Except for types '`a`' and '`c`', you can specify the number of bytes to use in interpreting each number in the given data type by following the type indicator character with a decimal integer. Alternately, you can specify the size of one of the C compiler's built-in data types by following the type indicator character with one of the following characters. For integers ('`d`', '`o`', '`u`', '`x`'):

'C' char,

'S' short,

'I' int,

'L' long.

For floating point (f):

F float,

D double,

L long double.

'-v'
'--output-duplicates'
Output consecutive lines that are identical. By default, when two or more consecutive output lines would be identical, od outputs only the first line, and puts just an asterisk on the following line to indicate the elision.

'-w[n]'
'--width[=n]'
Dump n input bytes per output line. This must be a multiple of the least common multiple of the sizes associated with the specified output types. If n is omitted, the default is 32. If this option is not given at all, the default is 16.

The next several options map the old, pre-POSIX format specification options to the corresponding POSIX format specs. GNU od accepts any combination of old- and new-style options. Format specification options accumulate.

'-a' Output as named characters. Equivalent to '-ta'.

'-b' Output as octal bytes. Equivalent to '-toC'.

'-c' Output as ASCII characters or backslash escapes. Equivalent to '-tc'.

'-d' Output as unsigned decimal shorts. Equivalent to '-tu2'.

'-f' Output as floats. Equivalent to '-tfF'.

'-h' Output as hexadecimal shorts. Equivalent to '-tx2'.

'-i' Output as decimal shorts. Equivalent to '-td2'.

'-l' Output as decimal longs. Equivalent to '-td4'.

'-o' Output as octal shorts. Equivalent to '-to2'.

'-x' Output as hexadecimal shorts. Equivalent to '-tx2'.

'-C'
'--traditional'
Recognize the pre-POSIX non-option arguments that traditional od accepted. The following syntax:

`od --traditional` [*file*] [[`+`]*offset*[`.`][`b`] [[`+`]*label*[`.`][`b`]]]

can be used to specify at most one file and optional arguments specifying an offset and a pseudo-start address, *label*. By default, *offset* is interpreted as an octal number specifying how many input bytes to skip before formatting and writing. The optional trailing decimal point forces the interpretation of *offset* as a decimal number. If no decimal is specified and the offset begins with '`0x`' or '`0X`' it is interpreted as a hexadecimal number. If there is a trailing '`b`', the number of bytes skipped will be *offset* multiplied by 512. The *label* argument is interpreted just like *offset*, but it specifies an initial pseudo-address. The pseudo-addresses are displayed in parentheses following any normal address.

4 Formatting file contents

These commands reformat the contents of files.

4.1 `fmt`: Reformat paragraph text

`fmt` fills and joins lines to produce output lines of (at most) a given number of characters (75 by default). Synopsis:

```
fmt [option]... [file]...
```

`fmt` reads from the specified *file* arguments (or standard input if none are given), and writes to standard output.

By default, blank lines, spaces between words, and indentation are preserved in the output; successive input lines with different indentation are not joined; tabs are expanded on input and introduced on output.

`fmt` prefers breaking lines at the end of a sentence, and tries to avoid line breaks after the first word of a sentence or before the last word of a sentence. A *sentence break* is defined as either the end of a paragraph or a word ending in any of '`.?!`', followed by two spaces or end of line, ignoring any intervening parentheses or quotes. Like TEX, `fmt` reads entire "paragraphs" before choosing line breaks; the algorithm is a variant of that in "Breaking Paragraphs Into Lines" (Donald E. Knuth and Michael F. Plass, *Software—Practice and Experience*, 11 (1981), 1119–1184).

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-c`'
'`--crown-margin`'
: *Crown margin* mode: preserve the indentation of the first two lines within a paragraph, and align the left margin of each subsequent line with that of the second line.

'`-t`'
'`--tagged-paragraph`'
: *Tagged paragraph* mode: like crown margin mode, except that if indentation of the first line of a paragraph is the same as the indentation of the second, the first line is treated as a one-line paragraph.

'`-s`'
'`--split-only`'
: Split lines only. Do not join short lines to form longer ones. This prevents sample lines of code, and other such "formatted" text from being unduly combined.

'`-u`'
'`--uniform-spacing`'
: Uniform spacing. Reduce spacing between words to one space, and spacing between sentences to two spaces.

'-*width*'
'-w *width*'
'--width=*width*'
: Fill output lines up to *width* characters (default 75). `fmt` initially tries to make lines about 7% shorter than this, to give it room to balance line lengths.

'-p *prefix*'
'--prefix=*prefix*'
: Only lines beginning with *prefix* (possibly preceded by whitespace) are subject to formatting. The prefix and any preceding whitespace are stripped for the formatting and then re-attached to each formatted output line. One use is to format certain kinds of program comments, while leaving the code unchanged.

4.2 pr: Paginate or columnate files for printing

`pr` writes each *file* ('-' means standard input), or standard input if none are given, to standard output, paginating and optionally outputting in multicolumn format; optionally merges all *file*s, printing all in parallel, one per column. Synopsis:

```
pr [option]... [file]...
```

By default, a 5-line header is printed at each page: two blank lines; a line with the date, the filename, and the page count; and two more blank lines. A footer of five blank lines is also printed. With the '-F' option, a 3-line header is printed: the leading two blank lines are omitted; no footer is used. The default *page_length* in both cases is 66 lines. The default number of text lines changes from 56 (without '-F') to 63 (with '-F'). The text line of the header takes up the full *page_width* in the form '`yyyy-mm-dd HH:MM string Page nnnn`'. String is a centered header string.

Form feeds in the input cause page breaks in the output. Multiple form feeds produce empty pages.

Columns are of equal width, separated by an optional string (default is '`space`'). For multicolumn output, lines will always be truncated to *page_width* (default 72), unless you use the '-J' option. For single column output no line truncation occurs by default. Use '-W' option to truncate lines in that case.

Including version 1.22i:

Some small *letter options* ('-s', '-w') has been redefined with the object of a better *posix* compliance. The output of some further cases has been adapted to other *unix*es. A violation of downward compatibility has to be accepted.

Some *new capital letter* options ('-J', '-S', '-W') has been introduced to turn off unexpected interferences of small letter options. The '-N' option and the second argument *last_page* of '+FIRST_PAGE' offer more flexibility. The detailed handling of form feeds set in the input files requires '-T' option.

Capital letter options dominate small letter ones.

Some of the option-arguments (compare '-s', '-S', '-e', '-i', '-n') cannot be specified as separate arguments from the preceding option letter (already stated in the *posix* specification).

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'+*first_page*[:*last_page*]'
'--pages=*first_page*[:*last_page*]'
: Begin printing with page *first_page* and stop with *last_page*. Missing ':*last_page*' implies end of file. While estimating the number of skipped pages each form feed in the input file results in a new page. Page counting with and without '+*first_page*' is identical. By default, counting starts with the first page of input file (not first page printed). Line numbering may be altered by '-N' option.

'-*column*'
'--columns=*column*'
: With each single *file*, produce *column* columns of output (default is 1) and print columns down, unless '-a' is used. The column width is automatically decreased as *column* increases; unless you use the '-W/-w' option to increase *page_width* as well. This option might well cause some lines to be truncated. The number of lines in the columns on each page are balanced. The options '-e' and '-i' are on for multiple text-column output. Together with '-J' option column alignment and line truncation is turned off. Lines of full length are joined in a free field format and '-S' option may set field separators. '-*column*' may not be used with '-m' option.

'-a'
'--across'
: With each single *file*, print columns across rather than down. The '-*column*' option must be given with *column* greater than one. If a line is too long to fit in a column, it is truncated.

'-c'
'--show-control-chars'
: Print control characters using hat notation (e.g., '^G'); print other unprintable characters in octal backslash notation. By default, unprintable characters are not changed.

'-d'
'--double-space'
: Double space the output.

'-e[*in-tabchar*[*in-tabwidth*]]'
'--expand-tabs[=*in-tabchar*[*in-tabwidth*]]'
: Expand *tab*s to spaces on input. Optional argument *in-tabchar* is the input tab character (default is the TAB character). Second optional argument *in-tabwidth* is the input tab character's width (default is 8).

'-f'
'-F'
'--form-feed'
: Use a form feed instead of newlines to separate output pages. The default page length of 66 lines is not altered. But the number of lines of text per page changes from default 56 to 63 lines.

'`-h` *HEADER*'
'`--header=`*HEADER*'
: Replace the filename in the header with the centered string *header*. Left-hand-side truncation (marked by a '`*`') may occur if the total header line '`yyyy-mm-dd HH:MM HEADER Page nnnn`' becomes larger than *page_width*. '`-h ""`' prints a blank line header. Don't use '`-h""`'. A space between the '`-h`' option and the argument is always indispensable.

'`-i[`*out-tabchar*`[`*out-tabwidth*`]]`'
'`--output-tabs[=`*out-tabchar*`[`*out-tabwidth*`]]`'
: Replace spaces with *tab*s on output. Optional argument *out-tabchar* is the output tab character (default is the TAB character). Second optional argument *out-tabwidth* is the output tab character's width (default is 8).

'`-J`'
'`--join-lines`'
: Merge lines of full length. Used together with the column options '`-`*column*', '`-a -`*column*' or '`-m`'. Turns off '`-W/-w`' line truncation; no column alignment used; may be used with '`-S[`*string*`]`'. '`-J`' has been introduced (together with '`-W`' and '`-S`') to disentangle the old (*posix* compliant) options '`-w`' and '`-s`' along with the three column options.

'`-l` *page_length*'
'`--length=`*page_length*'
: Set the page length to *page_length* (default 66) lines, including the lines of the header [and the footer]. If *page_length* is less than or equal 10 (and `<=` 3 with '`-F`'), the header and footer are omitted, and all form feeds set in input files are eliminated, as if the '`-T`' option had been given.

'`-m`'
'`--merge`'
: Merge and print all *file*s in parallel, one in each column. If a line is too long to fit in a column, it is truncated, unless '`-J`' option is used. '`-S[`*string*`]`' may be used. Empty pages in some *file*s (form feeds set) produce empty columns, still marked by *string*. The result is a continuous line numbering and column marking throughout the whole merged file. Completely empty merged pages show no separators or line numbers. The default header becomes '`yyyy-mm-dd HH:MM <blanks> Page nnnn`'; may be used with '`-h` *header*' to fill up the middle blank part.

'`-n[`*number-separator*`[`*digits*`]]`'
'`--number-lines[=`*number-separator*`[`*digits*`]]`'
: Provide *digits* digit line numbering (default for *digits* is 5). With multicolumn output the number occupies the first *digits* column positions of each text column or only each line of '`-m`' output. With single column output the number precedes each line just as '`-m`' does. Default counting of the line numbers starts with 1st line of the input file (not the 1st line printed, compare the '`--page`' option and '`-N`' option). Optional argument *number-separator* is the character appended to the line number to separate it from the text followed. The default separator is the TAB character. In a strict sense a TAB is always printed with single column output only. The *TAB*-width varies with the *TAB*-position, e.g. with

the left *margin* specified by '`-o`' option. With multicolumn output priority is given to '`equal width of output columns`' (a *posix* specification). The *TAB*-width is fixed to the value of the 1st column and does not change with different values of left *margin*. That means a fixed number of spaces is always printed in the place of the *number-separator tab*. The tabification depends upon the output position.

'`-N` *line_number*'
'`--first-line-number=`*line_number*'
: Start line counting with the number *line_number* at first line of first page printed (in most cases not the first line of the input file).

'`-o` *margin*'
'`--indent=`*margin*'
: Indent each line with a margin *margin* spaces wide (default is zero). The total page width is the size of the margin plus the *page_width* set with the '`-W/-w`' option. A limited overflow may occur with numbered single column output (compare '`-n`' option).

'`-r`'
'`--no-file-warnings`'
: Do not print a warning message when an argument *file* cannot be opened. (The exit status will still be nonzero, however.)

'`-s[`*char*`]`'
'`--separator[=`*char*`]`'
: Separate columns by a single character *char*. Default for *char* is the TAB character without '`-w`' and '`no character`' with '`-w`'. Without '`-s`' default separator '`space`' is set. '`-s[char]`' turns off line truncation of all three column options ('`-COLUMN`'|'`-a -COLUMN`'|'`-m`') except '`-w`' is set. That is a *posix* compliant formulation.

'`-S[`*string*`]`'
'`--sep-string[=`*string*`]`'
: Use *string* to separate output columns. The '`-S`' option doesn't affect the '`-W/-w`' option, unlike the '`-s`' option which does. It does not affect line truncation or column alignment. Without '`-S`', and with '`-J`', `pr` uses the default output separator, TAB. Without '`-S`' or '`-J`', `pr` uses a '`space`' (same as '`-S" "`'). Using '`-S`' with no *string* is equivalent to '`-S""`'. Note that for some of `pr`'s options the single-letter option character must be followed immediately by any corresponding argument; there may not be any intervening white space. '`-S/-s`' is one of them. Don't use '`-S "STRING"`'. POSIX requires this.

'`-t`'
'`--omit-header`'
: Do not print the usual header [and footer] on each page, and do not fill out the bottom of pages (with blank lines or a form feed). No page structure is produced, but form feeds set in the input files are retained. The predefined pagination is not changed. '`-t`' or '`-T`' may be useful together with other options; e.g.: '`-t -e4`', expand TAB characters in the input file to 4 spaces but don't make any other changes. Use of '`-t`' overrides '`-h`'.

'`-T`'
'`--omit-pagination`'
: Do not print header [and footer]. In addition eliminate all form feeds set in the input files.

'`-v`'
'`--show-nonprinting`'
: Print unprintable characters in octal backslash notation.

'`-w` *page_width*'
'`--width=`*page_width*'
: Set page width to *page_width* characters for multiple text-column output only (default for *page_width* is 72). '`-s[CHAR]`' turns off the default page width and any line truncation and column alignment. Lines of full length are merged, regardless of the column options set. No *page_width* setting is possible with single column output. A *posix* compliant formulation.

'`-W` *page_width*'
'`--page_width=`*page_width*'
: Set the page width to *page_width* characters. That's valid with and without a column option. Text lines are truncated, unless '`-J`' is used. Together with one of the three column options ('`-`*column*', '`-a -`*column*' or '`-m`') column alignment is always used. The separator options '`-S`' or '`-s`' don't affect the '`-W`' option. Default is 72 characters. Without '`-W` *page_width*' and without any of the column options NO line truncation is used (defined to keep downward compatibility and to meet most frequent tasks). That's equivalent to '`-W 72 -J`'. With and without '`-W` *page_width*' the header line is always truncated to avoid line overflow.

4.3 `fold`: Wrap input lines to fit in specified width

`fold` writes each *file* ('`-`' means standard input), or standard input if none are given, to standard output, breaking long lines. Synopsis:

```
fold [option]... [file]...
```

By default, `fold` breaks lines wider than 80 columns. The output is split into as many lines as necessary.

`fold` counts screen columns by default; thus, a tab may count more than one column, backspace decreases the column count, and carriage return sets the column to zero.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-b`'
'`--bytes`'
: Count bytes rather than columns, so that tabs, backspaces, and carriage returns are each counted as taking up one column, just like other characters.

'`-s`'
'`--spaces`'
: Break at word boundaries: the line is broken after the last blank before the maximum line length. If the line contains no such blanks, the line is broken at the maximum line length as usual.

‘`-w` *width*’
‘`--width=`*width*’
: Use a maximum line length of *width* columns instead of 80.

5 Output of parts of files

These commands output pieces of the input.

5.1 `head`: Output the first part of files

`head` prints the first part (10 lines by default) of each *file*; it reads from standard input if no files are given or when given a *file* of '`-`'. Synopses:

```
head [option]... [file]...
head -number [option]... [file]...
```

If more than one *file* is specified, `head` prints a one-line header consisting of

```
==> file name <==
```

before the output for each *file*.

`head` accepts two option formats: the new one, in which numbers are arguments to the options ('`-q -n 1`'), and the old one, in which the number precedes any option letters ('`-1q`').

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'*-countoptions*'
: This option is only recognized if it is specified first. *count* is a decimal number optionally followed by a size letter ('`b`', '`k`', '`m`') as in `-c`, or '`l`' to mean count by lines, or other option letters ('`cqv`').

'`-c` *bytes*'
'`--bytes=`*bytes*'
: Print the first *bytes* bytes, instead of initial lines. Appending '`b`' multiplies *bytes* by 512, '`k`' by 1024, and '`m`' by 1048576.

'`-n` *n*'
'`--lines=`*n*'
: Output the first *n* lines.

'`-q`'
'`--quiet`'
'`--silent`'
: Never print file name headers.

'`-v`'
'`--verbose`'
: Always print file name headers.

5.2 `tail`: Output the last part of files

`tail` prints the last part (10 lines by default) of each *file*; it reads from standard input if no files are given or when given a *file* of '`-`'. Synopses:

```
tail [option]... [file]...
tail -number [option]... [file]...
tail +number [option]... [file]...
```

If more than one *file* is specified, `tail` prints a one-line header consisting of

```
==> file name <==
```

before the output for each *file*.

GNU `tail` can output any amount of data (some other versions of `tail` cannot). It also has no '`-r`' option (print in reverse), since reversing a file is really a different job from printing the end of a file; BSD `tail` (which is the one with `-r`) can only reverse files that are at most as large as its buffer, which is typically 32k. A more reliable and versatile way to reverse files is the GNU `tac` command.

`tail` accepts two option formats: the new one, in which numbers are arguments to the options ('`-n 1`'), and the old one, in which the number precedes any option letters ('`-1`' or '`+1`').

If any option-argument is a number *n* starting with a '`+`', `tail` begins printing with the *n*th item from the start of each file, instead of from the end.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'*-count*'
'*+count*'
: This option is only recognized if it is specified first. *count* is a decimal number optionally followed by a size letter ('`b`', '`k`', '`m`') as in `-c`, or '`l`' to mean count by lines, or other option letters ('`cfqv`').

'`-c` *bytes*'
'`--bytes=`*bytes*'
: Output the last *bytes* bytes, instead of final lines. Appending '`b`' multiplies *bytes* by 512, '`k`' by 1024, and '`m`' by 1048576.

'`-f`'
'`--follow[=`*how*`]`'
: Loop forever trying to read more characters at the end of the file, presumably because the file is growing. This option is ignored when reading from a pipe. If more than one file is given, `tail` prints a header whenever it gets output from a different file, to indicate which file that output is from.

  There are two ways to specify how you'd like to track files with this option, but that difference is noticeable only when a followed file is removed or renamed. If you'd like to continue to track the end of a growing file even after it has been unlinked, use '`--follow=descriptor`'. This is the default behavior, but it is not useful if you're tracking a log file that may be rotated (removed or renamed, then reopened). In that case, use '`--follow=name`' to track the named file by reopening it periodically to see if it has been removed and recreated by some other program.

  No matter which method you use, if the tracked file is determined to have shrunk, `tail` prints a message saying the file has been truncated and resumes tracking the end of the file from the newly-determined endpoint.

When a file is removed, `tail`'s behavior depends on whether it is following the name or the descriptor. When following by name, tail can detect that a file has been removed and gives a message to that effect, and if '`--retry`' has been specified it will continue checking periodically to see if the file reappears. When following a descriptor, tail does not detect that the file has been unlinked or renamed and issues no message; even though the file may no longer be accessible via its original name, it may still be growing.

The option values '`descriptor`' and '`name`' may be specified only with the long form of the option, not with '`-f`'.

'`--retry`' This option is meaningful only when following by name. Without this option, when tail encounters a file that doesn't exist or is otherwise inaccessible, it reports that fact and never checks it again.

'`--sleep-interval=`*n*'
Change the number of seconds to wait between iterations (the default is 1). During one iteration, every specified file is checked to see if it has changed size.

'`--pid=`*pid*'
When following by name or by descriptor, you may specify the process ID, *pid*, of the sole writer of all *file* arguments. Then, shortly after that process terminates, tail will also terminate. This will work properly only if the writer and the tailing process are running on the same machine. For example, to save the output of a build in a file and to watch the file grow, if you invoke `make` and `tail` like this then the tail process will stop when your build completes. Without this option, you would have had to kill the `tail -f` process yourself.

```
$ make >& makerr & tail --pid=$! -f makerr
```

If you specify a *pid* that is not in use or that does not correspond to the process that is writing to the tailed files, then `tail` may terminate long before any *file*s stop growing or it may not terminate until long after the real writer has terminated.

'`--max-consecutive-size-changes=`*n*'
This option is meaningful only when following by name. Use it to control how long `tail` follows the descriptor of a file that continues growing at a rapid pace even after it is deleted or renamed. After detecting *n* consecutive size changes for a file, `open/fstat` the file to determine if that file name is still associated with the same device/inode-number pair as before. See the output of `tail --help` for the default value.

'`--max-unchanged-stats=`*n*'
When tailing a file by name, if there have been this many consecutive iterations for which the size has remained the same, then `open/fstat` the file to determine if that file name is still associated with the same device/inode-number pair as before. When following a log file that is rotated this is approximately the number of seconds between when tail prints the last pre-rotation lines and when it prints the lines that have accumulated in the new log file. See the

output of `tail --help` for the default value. This option is meaningful only when following by name.

'`-n` *n*'
'`--lines=`*n*'
Output the last *n* lines.

'`-q`'
'`-quiet`'
'`--silent`'
Never print file name headers.

'`-v`'
'`--verbose`'
Always print file name headers.

5.3 `split`: Split a file into fixed-size pieces

`split` creates output files containing consecutive sections of *input* (standard input if none is given or *input* is '`-`'). Synopsis:

```
split [option] [input [prefix]]
```

By default, `split` puts 1000 lines of *input* (or whatever is left over for the last section), into each output file.

The output files' names consist of *prefix* ('`x`' by default) followed by a group of letters '`aa`', '`ab`', and so on, such that concatenating the output files in sorted order by file name produces the original input file. (If more than 676 output files are required, `split` uses '`zaa`', '`zab`', etc.)

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'*-lines*'
'`-l` *lines*'
'`--lines=`*lines*'
Put *lines* lines of *input* into each output file.

'`-b` *bytes*'
'`--bytes=`*bytes*'
Put the first *bytes* bytes of *input* into each output file. Appending '`b`' multiplies *bytes* by 512, '`k`' by 1024, and '`m`' by 1048576.

'`-C` *bytes*'
'`--line-bytes=`*bytes*'
Put into each output file as many complete lines of *input* as possible without exceeding *bytes* bytes. For lines longer than *bytes* bytes, put *bytes* bytes into each output file until less than *bytes* bytes of the line are left, then continue normally. *bytes* has the same format as for the '`--bytes`' option.

'`--verbose`'
Write a diagnostic to standard error just before each output file is opened.

5.4 `csplit`: Split a file into context-determined pieces

`csplit` creates zero or more output files containing sections of *input* (standard input if *input* is '`-`'). Synopsis:

```
csplit [option]... input pattern...
```

The contents of the output files are determined by the *pattern* arguments, as detailed below. An error occurs if a *pattern* argument refers to a nonexistent line of the input file (e.g., if no remaining line matches a given regular expression). After every *pattern* has been matched, any remaining input is copied into one last output file.

By default, `csplit` prints the number of bytes written to each output file after it has been created.

The types of pattern arguments are:

'*n*'
: Create an output file containing the input up to but not including line *n* (a positive integer). If followed by a repeat count, also create an output file containing the next *line* lines of the input file once for each repeat.

'`/`*regexp*`/[`*offset*`]`'
: Create an output file containing the current line up to (but not including) the next line of the input file that contains a match for *regexp*. The optional *offset* is a '`+`' or '`-`' followed by a positive integer. If it is given, the input up to the matching line plus or minus *offset* is put into the output file, and the line after that begins the next section of input.

'`%`*regexp*`%[`*offset*`]`'
: Like the previous type, except that it does not create an output file, so that section of the input file is effectively ignored.

'`{`*repeat-count*`}`'
: Repeat the previous pattern *repeat-count* additional times. *repeat-count* can either be a positive integer or an asterisk, meaning repeat as many times as necessary until the input is exhausted.

The output files' names consist of a prefix ('`xx`' by default) followed by a suffix. By default, the suffix is an ascending sequence of two-digit decimal numbers from '`00`' and up to '`99`'. In any case, concatenating the output files in sorted order by filename produces the original input file.

By default, if `csplit` encounters an error or receives a hangup, interrupt, quit, or terminate signal, it removes any output files that it has created so far before it exits.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-f` *prefix*'
'`--prefix=`*prefix*'
: Use *prefix* as the output file name prefix.

'`-b` *suffix*'
'`--suffix=`*suffix*'
: Use *suffix* as the output file name suffix. When this option is specified, the suffix string must include exactly one `printf(3)`-style conversion specification,

possibly including format specification flags, a field width, a precision specifications, or all of these kinds of modifiers. The format letter must convert a binary integer argument to readable form; thus, only '`d`', '`i`', '`u`', '`o`', '`x`', and '`X`' conversions are allowed. The entire *suffix* is given (with the current output file number) to `sprintf(3)` to form the file name suffixes for each of the individual output files in turn. If this option is used, the '`--digits`' option is ignored.

'`-n` *digits*'
'`--digits=`*digits*'
Use output file names containing numbers that are *digits* digits long instead of the default 2.

'`-k`'
'`--keep-files`'
Do not remove output files when errors are encountered.

'`-z`'
'`--elide-empty-files`'
Suppress the generation of zero-length output files. (In cases where the section delimiters of the input file are supposed to mark the first lines of each of the sections, the first output file will generally be a zero-length file unless you use this option.) The output file sequence numbers always run consecutively starting from 0, even when this option is specified.

'`-s`'
'`-q`'
'`--silent`'
'`--quiet`' Do not print counts of output file sizes.

6 Summarizing files

These commands generate just a few numbers representing entire contents of files.

6.1 `wc`: Print byte, word, and line counts

`wc` counts the number of bytes, whitespace-separated words, and newlines in each given *file*, or standard input if none are given or for a *file* of '`-`'. Synopsis:

```
wc [option]... [file]...
```

`wc` prints one line of counts for each file, and if the file was given as an argument, it prints the file name following the counts. If more than one *file* is given, `wc` prints a final line containing the cumulative counts, with the file name '`total`'. The counts are printed in this order: newlines, words, bytes. By default, each count is output right-justified in a 7-byte field with one space between fields so that the numbers and file names line up nicely in columns. However, POSIX requires that there be exactly one space separating columns. You can make `wc` use the POSIX-mandated output format by setting the `POSIXLY_CORRECT` environment variable.

By default, `wc` prints all three counts. Options can specify that only certain counts be printed. Options do not undo others previously given, so

```
wc --bytes --words
```

prints both the byte counts and the word counts.

With the `--max-line-length` option, `wc` prints the length of the longest line per file, and if there is more than one file it prints the maximum (not the sum) of those lengths.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-c`'
'`--bytes`'
'`--chars`' Print only the byte counts.

'`-w`'
'`--words`' Print only the word counts.

'`-l`'
'`--lines`' Print only the newline counts.

'`-L`'
'`--max-line-length`'
Print only the maximum line lengths.

6.2 `sum`: Print checksum and block counts

`sum` computes a 16-bit checksum for each given *file*, or standard input if none are given or for a *file* of '`-`'. Synopsis:

```
sum [option]... [file]...
```

`sum` prints the checksum for each *file* followed by the number of blocks in the file (rounded up). If more than one *file* is given, file names are also printed (by default). (With the

'`--sysv`' option, corresponding file name are printed when there is at least one file argument.)

By default, GNU `sum` computes checksums using an algorithm compatible with BSD `sum` and prints file sizes in units of 1024-byte blocks.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-r`' Use the default (BSD compatible) algorithm. This option is included for compatibility with the System V `sum`. Unless '`-s`' was also given, it has no effect.

'`-s`'

'`--sysv`' Compute checksums using an algorithm compatible with System V `sum`'s default, and print file sizes in units of 512-byte blocks.

`sum` is provided for compatibility; the `cksum` program (see next section) is preferable in new applications.

6.3 `cksum`: Print CRC checksum and byte counts

`cksum` computes a cyclic redundancy check (CRC) checksum for each given *file*, or standard input if none are given or for a *file* of '`-`'. Synopsis:

```
cksum [option]... [file]...
```

`cksum` prints the CRC checksum for each file along with the number of bytes in the file, and the filename unless no arguments were given.

`cksum` is typically used to ensure that files transferred by unreliable means (e.g., netnews) have not been corrupted, by comparing the `cksum` output for the received files with the `cksum` output for the original files (typically given in the distribution).

The CRC algorithm is specified by the POSIX.2 standard. It is not compatible with the BSD or System V `sum` algorithms (see the previous section); it is more robust.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

6.4 `md5sum`: Print or check message-digests

`md5sum` computes a 128-bit checksum (or *fingerprint* or *message-digest*) for each specified *file*. If a *file* is specified as '`-`' or if no files are given `md5sum` computes the checksum for the standard input. `md5sum` can also determine whether a file and checksum are consistent. Synopses:

```
md5sum [option]... [file]...
md5sum [option]... --check [file]
```

For each *file*, '`md5sum`' outputs the MD5 checksum, a flag indicating a binary or text input file, and the filename. If *file* is omitted or specified as '`-`', standard input is read.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-b`'
'`--binary`'
: Treat all input files as binary. This option has no effect on Unix systems, since they don't distinguish between binary and text files. This option is useful on systems that have different internal and external character representations. On MS-DOS and MS-Windows, this is the default.

'`-c`'
'`--check`'
: Read filenames and checksum information from the single *file* (or from stdin if no *file* was specified) and report whether each named file and the corresponding checksum data are consistent. The input to this mode of `md5sum` is usually the output of a prior, checksum-generating run of '`md5sum`'. Each valid line of input consists of an MD5 checksum, a binary/text flag, and then a filename. Binary files are marked with '`*`', text with ' '. For each such line, `md5sum` reads the named file and computes its MD5 checksum. Then, if the computed message digest does not match the one on the line with the filename, the file is noted as having failed the test. Otherwise, the file passes the test. By default, for each valid line, one line is written to standard output indicating whether the named file passed the test. After all checks have been performed, if there were any failures, a warning is issued to standard error. Use the '`--status`' option to inhibit that output. If any listed file cannot be opened or read, if any valid line has an MD5 checksum inconsistent with the associated file, or if no valid line is found, `md5sum` exits with nonzero status. Otherwise, it exits successfully.

'`--status`'
: This option is useful only when verifying checksums. When verifying checksums, don't generate the default one-line-per-file diagnostic and don't output the warning summarizing any failures. Failures to open or read a file still evoke individual diagnostics to standard error. If all listed files are readable and are consistent with the associated MD5 checksums, exit successfully. Otherwise exit with a status code indicating there was a failure.

'`-t`'
'`--text`'
: Treat all input files as text files. This is the reverse of '`--binary`'.

'`-w`'
'`--warn`'
: When verifying checksums, warn about improperly formatted MD5 checksum lines. This option is useful only if all but a few lines in the checked input are valid.

7 Operating on sorted files

These commands work with (or produce) sorted files.

7.1 `sort`: Sort text files

`sort` sorts, merges, or compares all the lines from the given files, or standard input if none are given or for a *file* of '`-`'. By default, `sort` writes the results to standard output. Synopsis:

```
sort [option]... [file]...
```

`sort` has three modes of operation: sort (the default), merge, and check for sortedness. The following options change the operation mode:

'`-c`' Check whether the given files are already sorted: if they are not all sorted, print an error message and exit with a status of 1. Otherwise, exit successfully.

'`-m`' Merge the given files by sorting them as a group. Each input file must always be individually sorted. It always works to sort instead of merge; merging is provided because it is faster, in the case where it works.

A pair of lines is compared as follows: if any key fields have been specified, `sort` compares each pair of fields, in the order specified on the command line, according to the associated ordering options, until a difference is found or no fields are left. Unless otherwise specified, all comparisons use the character collating sequence specified by the `LC_COLLATE` locale.

If any of the global options '`Mbdfinr`' are given but no key fields are specified, `sort` compares the entire lines according to the global options.

Finally, as a last resort when all keys compare equal (or if no ordering options were specified at all), `sort` compares the entire lines. The last resort comparison honors the '`-r`' global option. The '`-s`' (stable) option disables this last-resort comparison so that lines in which all fields compare equal are left in their original relative order. If no fields or global options are specified, '`-s`' has no effect.

GNU `sort` (as specified for all GNU utilities) has no limits on input line length or restrictions on bytes allowed within lines. In addition, if the final byte of an input file is not a newline, GNU `sort` silently supplies one. A line's trailing newline is part of the line for comparison purposes; for example, with no options in an ASCII locale, a line starting with a tab sorts before an empty line because tab precedes newline in the ASCII collating sequence.

Upon any error, `sort` exits with a status of '`2`'.

If the environment variable `TMPDIR` is set, `sort` uses its value as the directory for temporary files instead of '`/tmp`'. The '`-T` *tempdir*' option in turn overrides the environment variable.

The following options affect the ordering of output lines. They may be specified globally or as part of a specific key field. If no key fields are specified, global options apply to comparison of entire lines; otherwise the global options are inherited by key fields that do not specify any special options of their own. The '`-b`', '`-d`', '`-f`' and '`-i`' options classify characters according to the `LC_CTYPE` locale.

'`-b`' Ignore leading blanks when finding sort keys in each line.

'`-d`' Sort in *phone directory* order: ignore all characters except letters, digits and blanks when sorting.

'`-f`' Fold lowercase characters into the equivalent uppercase characters when sorting so that, for example, '`b`' and '`B`' sort as equal.

'`-g`' Sort numerically, using the standard C function `strtod` to convert a prefix of each line to a double-precision floating point number. This allows floating point numbers to be specified in scientific notation, like `1.0e-34` and `10e100`. Do not report overflow, underflow, or conversion errors. Use the following collating sequence:

- Lines that do not start with numbers (all considered to be equal).
- NaNs ("Not a Number" values, in IEEE floating point arithmetic) in a consistent but machine-dependent order.
- Minus infinity.
- Finite numbers in ascending numeric order (with −0 and +0 equal).
- Plus infinity.

Use this option only if there is no alternative; it is much slower than '`-n`' and it can lose information when converting to floating point.

'`-i`' Ignore unprintable characters.

'`-M`' An initial string, consisting of any amount of whitespace, followed by a month name abbreviation, is folded to UPPER case and compared in the order '`JAN`' < '`FEB`' < ... < '`DEC`'. Invalid names compare low to valid names. The `LC_TIME` locale determines the month spellings.

'`-n`' Sort numerically: the number begins each line; specifically, it consists of optional whitespace, an optional '`-`' sign, and zero or more digits possibly separated by thousands separators, optionally followed by a radix character and zero or more digits. The `LC_NUMERIC` locale specifies the radix character and thousands separator.

`sort -n` uses what might be considered an unconventional method to compare strings representing floating point numbers. Rather than first converting each string to the C `double` type and then comparing those values, sort aligns the radix characters in the two strings and compares the strings a character at a time. One benefit of using this approach is its speed. In practice this is much more efficient than performing the two corresponding string-to-double (or even string-to-integer) conversions and then comparing doubles. In addition, there is no corresponding loss of precision. Converting each string to `double` before comparison would limit precision to about 16 digits on most systems.

Neither a leading '`+`' nor exponential notation is recognized. To compare such strings numerically, use the '`-g`' option.

'`-r`' Reverse the result of comparison, so that lines with greater key values appear earlier in the output instead of later.

Other options are:

'`-o` *output-file*'
: Write output to *output-file* instead of standard output. If *output-file* is one of the input files, `sort` copies it to a temporary file before sorting and writing the output to *output-file*.

'`-t` *separator*'
: Use character *separator* as the field separator when finding the sort keys in each line. By default, fields are separated by the empty string between a non-whitespace character and a whitespace character. That is, given the input line '` foo bar`', `sort` breaks it into fields '` foo`' and '` bar`'. The field separator is not considered to be part of either the field preceding or the field following.

'`-u`'
: For the default case or the '`-m`' option, only output the first of a sequence of lines that compare equal. For the '`-c`' option, check that no pair of consecutive lines compares equal.

'`-k` *pos1*`[,`*pos2*`]`'
: The recommended, POSIX, option for specifying a sort field. The field consists of the part of the line between *pos1* and *pos2* (or the end of the line, if *pos2* is omitted), *inclusive*. Fields and character positions are numbered starting with 1. So to sort on the second field, you'd use '`-k 2,2`' See below for more examples.

'`-z`'
: Treat the input as a set of lines, each terminated by a zero byte (ASCII NUL (Null) character) instead of an ASCII LF (Line Feed). This option can be useful in conjunction with '`perl -0`' or '`find -print0`' and '`xargs -0`' which do the same in order to reliably handle arbitrary pathnames (even those which contain Line Feed characters.)

'`+`*pos1*`[-`*pos2*`]`'
: The obsolete, traditional option for specifying a sort field. The field consists of the line between *pos1* and up to but *not including pos2* (or the end of the line if *pos2* is omitted). Fields and character positions are numbered starting with 0. See below.

In addition, when GNU `sort` is invoked with exactly one argument, options '`--help`' and '`--version`' are recognized. See Chapter 2 [Common options], page 2.

Historical (BSD and System V) implementations of `sort` have differed in their interpretation of some options, particularly '`-b`', '`-f`', and '`-n`'. GNU sort follows the POSIX behavior, which is usually (but not always!) like the System V behavior. According to POSIX, '`-n`' no longer implies '`-b`'. For consistency, '`-M`' has been changed in the same way. This may affect the meaning of character positions in field specifications in obscure cases. The only fix is to add an explicit '`-b`'.

A position in a sort field specified with the '`-k`' or '`+`' option has the form '*f*.*c*', where *f* is the number of the field to use and *c* is the number of the first character from the beginning of the field (for '`+`*pos*') or from the end of the previous field (for '`-`*pos*'). If the '`.`*c*' is omitted, it is taken to be the first character in the field. If the '`-b`' option was specified, the '`.`*c*' part of a field specification is counted from the first nonblank character of the field (for '`+`*pos*') or from the first nonblank character following the previous field (for '`-`*pos*').

A sort key option may also have any of the option letters '`Mbdfinr`' appended to it, in which case the global ordering options are not used for that particular field. The '`-b`' option may be independently attached to either or both of the '+*pos*' and '-*pos*' parts of a field specification, and if it is inherited from the global options it will be attached to both. Keys may span multiple fields.

Here are some examples to illustrate various combinations of options. In them, the POSIX '`-k`' option is used to specify sort keys rather than the obsolete '+*pos1*-*pos2*' syntax.

- Sort in descending (reverse) numeric order.

  ```
  sort -nr
  ```

  Sort alphabetically, omitting the first and second fields. This uses a single key composed of the characters beginning at the start of field three and extending to the end of each line.

  ```
  sort -k3
  ```

- Sort numerically on the second field and resolve ties by sorting alphabetically on the third and fourth characters of field five. Use '`:`' as the field delimiter.

  ```
  sort -t : -k 2,2n -k 5.3,5.4
  ```

  Note that if you had written '`-k 2`' instead of '`-k 2,2`' '`sort`' would have used all characters beginning in the second field and extending to the end of the line as the primary *numeric* key. For the large majority of applications, treating keys spanning more than one field as numeric will not do what you expect.

  Also note that the '`n`' modifier was applied to the field-end specifier for the first key. It would have been equivalent to specify '`-k 2n,2`' or '`-k 2n,2n`'. All modifiers except '`b`' apply to the associated *field*, regardless of whether the modifier character is attached to the field-start and/or the field-end part of the key specifier.

- Sort the password file on the fifth field and ignore any leading white space. Sort lines with equal values in field five on the numeric user ID in field three.

  ```
  sort -t : -k 5b,5 -k 3,3n /etc/passwd
  ```

  An alternative is to use the global numeric modifier '`-n`'.

  ```
  sort -t : -n -k 5b,5 -k 3,3 /etc/passwd
  ```

- Generate a tags file in case insensitive sorted order.

  ```
  find src -type f -print0 | sort -t / -z -f | xargs -0 etags --append
  ```

  The use of '`-print0`', '`-z`', and '`-0`' in this case mean that pathnames that contain Line Feed characters will not get broken up by the sort operation.

  Finally, to ignore both leading and trailing white space, you could have applied the '`b`' modifier to the field-end specifier for the first key,

  ```
  sort -t : -n -k 5b,5b -k 3,3 /etc/passwd
  ```

  or by using the global '`-b`' modifier instead of '`-n`' and an explicit '`n`' with the second key specifier.

  ```
  sort -t : -b -k 5,5 -k 3,3n /etc/passwd
  ```

7.2 `uniq`: Uniquify files

`uniq` writes the unique lines in the given '`input`', or standard input if nothing is given or for an *input* name of '`-`'. Synopsis:

`uniq` [*option*]... [*input* [*output*]]

By default, `uniq` prints the unique lines in a sorted file, i.e., discards all but one of identical successive lines. Optionally, it can instead show only lines that appear exactly once, or lines that appear more than once.

The input must be sorted. If your input is not sorted, perhaps you want to use `sort -u`.

If no *output* file is specified, `uniq` writes to standard output.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-`*n*'
'`-f` *n*'
'`--skip-fields=`*n*'
: Skip *n* fields on each line before checking for uniqueness. Fields are sequences of non-space non-tab characters that are separated from each other by at least one spaces or tabs.

'`+`*n*'
'`-s` *n*'
'`--skip-chars=`*n*'
: Skip *n* characters before checking for uniqueness. If you use both the field and character skipping options, fields are skipped over first.

'`-c`'
'`--count`'
: Print the number of times each line occurred along with the line.

'`-i`'
'`--ignore-case`'
: Ignore differences in case when comparing lines.

'`-d`'
'`--repeated`'
: Print only duplicate lines.

'`-D`'
'`--all-repeated`'
: Print all duplicate lines and only duplicate lines. This option is useful mainly in conjunction with other options e.g., to ignore case or to compare only selected fields. This is a GNU extension.

'`-u`'
'`--unique`'
: Print only unique lines.

'`-w` *n*'
'`--check-chars=`*n*'
: Compare *n* characters on each line (after skipping any specified fields and characters). By default the entire rest of the lines are compared.

7.3 comm: Compare two sorted files line by line

`comm` writes to standard output lines that are common, and lines that are unique, to two input files; a file name of '`-`' means standard input. Synopsis:

```
comm [option]... file1 file2
```

Before `comm` can be used, the input files must be sorted using the collating sequence specified by the `LC_COLLATE` locale, with trailing newlines significant. If an input file ends in a non-newline character, a newline is silently appended. The `sort` command with no options always outputs a file that is suitable input to `comm`.

With no options, `comm` produces three column output. Column one contains lines unique to *file1*, column two contains lines unique to *file2*, and column three contains lines common to both files. Columns are separated by a single TAB character.

The options '`-1`', '`-2`', and '`-3`' suppress printing of the corresponding columns. Also see Chapter 2 [Common options], page 2.

Unlike some other comparison utilities, `comm` has an exit status that does not depend on the result of the comparison. Upon normal completion `comm` produces an exit code of zero. If there is an error it exits with nonzero status.

7.4 tsort: Topological sort

`tsort` performs a topological sort on the given *file*, or standard input if no input file is given or for a *file* of '`-`'. Synopsis:

```
tsort [option] [file]
```

`tsort` reads its input as pairs of strings, separated by blanks, indicating a partial ordering. The output is a total ordering that corresponds to the given partial ordering.

For example

```
tsort <<EOF
a b c
d
e f
b c d e
EOF
```

will produce the output

```
a
b
c
d
e
f
```

`tsort` will detect cycles in the input and writes the first cycle encountered to standard error.

Note that for a given partial ordering, generally there is no unique total ordering.

The only options are '`--help`' and '`--version`'. See Chapter 2 [Common options], page 2.

7.5 `ptx`: Produce permuted indexes

`ptx` reads a text file and essentially produces a permuted index, with each keyword in its context. The calling sketch is either one of:

```
ptx [option ...] [file ...]
ptx -G [option ...] [input [output]]
```

The '`-G`' (or its equivalent: '`--traditional`') option disables all GNU extensions and revert to traditional mode, thus introducing some limitations, and changes several of the program's default option values. When '`-G`' is not specified, GNU extensions are always enabled. GNU extensions to `ptx` are documented wherever appropriate in this document. For the full list, see See Section 7.5.5 [Compatibility in ptx], page 37.

Individual options are explained in incoming sections.

When GNU extensions are enabled, there may be zero, one or several *file* after the options. If there is no *file*, the program reads the standard input. If there is one or several *file*, they give the name of input files which are all read in turn, as if all the input files were concatenated. However, there is a full contextual break between each file and, when automatic referencing is requested, file names and line numbers refer to individual text input files. In all cases, the program produces the permuted index onto the standard output.

When GNU extensions are *not* enabled, that is, when the program operates in traditional mode, there may be zero, one or two parameters besides the options. If there is no parameters, the program reads the standard input and produces the permuted index onto the standard output. If there is only one parameter, it names the text *input* to be read instead of the standard input. If two parameters are given, they give respectively the name of the *input* file to read and the name of the *output* file to produce. *Be very careful* to note that, in this case, the contents of file given by the second parameter is destroyed. This behaviour is dictated only by System V `ptx` compatibility, because GNU Standards discourage output parameters not introduced by an option.

Note that for *any* file named as the value of an option or as an input text file, a single dash `-` may be used, in which case standard input is assumed. However, it would not make sense to use this convention more than once per program invocation.

7.5.1 General options

'`-C`'
'`--copyright`'
: Prints a short note about the Copyright and copying conditions, then exit without further processing.

'`-G`'
'`--traditional`'
: As already explained, this option disables all GNU extensions to `ptx` and switch to traditional mode.

'`--help`' Prints a short help on standard output, then exit without further processing.

'`--version`'
: Prints the program verison on standard output, then exit without further processing.

7.5.2 Charset selection

As it is setup now, the program assumes that the input file is coded using 8-bit ISO 8859-1 code, also known as Latin-1 character set, *unless* if it is compiled for MS-DOS, in which case it uses the character set of the IBM-PC. (GNU `ptx` is not known to work on smaller MS-DOS machines anymore.) Compared to 7-bit ASCII, the set of characters which are letters is then different, this fact alters the behaviour of regular expression matching. Thus, the default regular expression for a keyword allows foreign or diacriticized letters. Keyword sorting, however, is still crude; it obeys the underlying character set ordering quite blindly.

'`-f`'
'`--ignore-case`'
: Fold lower case letters to upper case for sorting.

7.5.3 Word selection and input processing

'`-b` *file*'

'`--break-file=`*file*'
: This option provides an alternative (to '`-W`') method of describing which characters make up words. It introduces the name of a file which contains a list of characters which can*not* be part of one word, this file is called the *Break file*. Any character which is not part of the Break file is a word constituent. If both options '`-b`' and '`-W`' are specified, then '`-W`' has precedence and '`-b`' is ignored.

  When GNU extensions are enabled, the only way to avoid newline as a break character is to write all the break characters in the file with no newline at all, not even at the end of the file. When GNU extensions are disabled, spaces, tabs and newlines are always considered as break characters even if not included in the Break file.

'`-i` *file*'
'`--ignore-file=`*file*'
: The file associated with this option contains a list of words which will never be taken as keywords in concordance output. It is called the *Ignore file*. The file contains exactly one word in each line; the end of line separation of words is not subject to the value of the '`-S`' option.

  There is a default Ignore file used by `ptx` when this option is not specified, usually found in '`/usr/local/lib/eign`' if this has not been changed at installation time. If you want to deactivate the default Ignore file, specify `/dev/null` instead.

'`-o` *file*'
'`--only-file=`*file*'
: The file associated with this option contains a list of words which will be retained in concordance output, any word not mentioned in this file is ignored. The file is called the *Only file*. The file contains exactly one word in each line; the end of line separation of words is not subject to the value of the '`-S`' option.

There is no default for the Only file. In the case there are both an Only file and an Ignore file, a word will be subject to be a keyword only if it is given in the Only file and not given in the Ignore file.

'`-r`'
'`--references`'
: On each input line, the leading sequence of non white characters will be taken to be a reference that has the purpose of identifying this input line on the produced permuted index. For more information about reference production, see See Section 7.5.4 [Output formatting in ptx], page 35. Using this option changes the default value for option '`-S`'.

  Using this option, the program does not try very hard to remove references from contexts in output, but it succeeds in doing so *when* the context ends exactly at the newline. If option '`-r`' is used with '`-S`' default value, or when GNU extensions are disabled, this condition is always met and references are completely excluded from the output contexts.

'`-S` *regexp*'
'`--sentence-regexp=`*regexp*'
: This option selects which regular expression will describe the end of a line or the end of a sentence. In fact, there is other distinction between end of lines or end of sentences than the effect of this regular expression, and input line boundaries have no special significance outside this option. By default, when GNU extensions are enabled and if '`-r`' option is not used, end of sentences are used. In this case, the precise *regex* is imported from GNU emacs:

  ```
  [.?!][]\"')}]*\\($\\|\t\\|  \\)[ \t\n]*
  ```

  Whenever GNU extensions are disabled or if '`-r`' option is used, end of lines are used; in this case, the default *regexp* is just:

  ```
  \n
  ```

  Using an empty *regexp* is equivalent to completely disabling end of line or end of sentence recognition. In this case, the whole file is considered to be a single big line or sentence. The user might want to disallow all truncation flag generation as well, through option '`-F ""`'. See section "Syntax of Regular Expressions" in *The GNU Emacs Manual*.

  When the keywords happen to be near the beginning of the input line or sentence, this often creates an unused area at the beginning of the output context line; when the keywords happen to be near the end of the input line or sentence, this often creates an unused area at the end of the output context line. The program tries to fill those unused areas by wrapping around context in them; the tail of the input line or sentence is used to fill the unused area on the left of the output line; the head of the input line or sentence is used to fill the unused area on the right of the output line.

  As a matter of convenience to the user, many usual backslashed escape sequences, as found in the C language, are recognized and converted to the corresponding characters by `ptx` itself.

'`-W` *regexp*'
'`--word-regexp=`*regexp*'
: This option selects which regular expression will describe each keyword. By default, if GNU extensions are enabled, a word is a sequence of letters; the *regexp* used is '`\w+`'. When GNU extensions are disabled, a word is by default anything which ends with a space, a tab or a newline; the *regexp* used is '`[^ \t\n]+`'.

  An empty *regexp* is equivalent to not using this option, letting the default dive in. See section "Syntax of Regular Expressions" in *The GNU Emacs Manual*.

  As a matter of convenience to the user, many usual backslashed escape sequences, as found in the C language, are recognized and converted to the corresponding characters by `ptx` itself.

7.5.4 Output formatting

Output format is mainly controlled by '`-O`' and '`-T`' options, described in the table below. When neither '`-O`' nor '`-T`' is selected, and if GNU extensions are enabled, the program choose an output format suited for a dumb terminal. Each keyword occurrence is output to the center of one line, surrounded by its left and right contexts. Each field is properly justified, so the concordance output could readily be observed. As a special feature, if automatic references are selected by option '`-A`' and are output before the left context, that is, if option '`-R`' is *not* selected, then a colon is added after the reference; this nicely interfaces with GNU Emacs `next-error` processing. In this default output format, each white space character, like newline and tab, is merely changed to exactly one space, with no special attempt to compress consecutive spaces. This might change in the future. Except for those white space characters, every other character of the underlying set of 256 characters is transmitted verbatim.

Output format is further controlled by the following options.

'`-g` *number*'
'`--gap-size=`*number*'
: Select the size of the minimum white gap between the fields on the output line.

'`-w` *number*'
'`--width=`*number*'
: Select the output maximum width of each final line. If references are used, they are included or excluded from the output maximum width depending on the value of option '`-R`'. If this option is not selected, that is, when references are output before the left context, the output maximum width takes into account the maximum length of all references. If this options is selected, that is, when references are output after the right context, the output maximum width does not take into account the space taken by references, nor the gap that precedes them.

'`-A`'
'`--auto-reference`'
: Select automatic references. Each input line will have an automatic reference made up of the file name and the line ordinal, with a single colon between them.

However, the file name will be empty when standard input is being read. If both '`-A`' and '`-r`' are selected, then the input reference is still read and skipped, but the automatic reference is used at output time, overriding the input reference.

'`-R`'
'`--right-side-refs`'
: In default output format, when option '`-R`' is not used, any reference produced by the effect of options '`-r`' or '`-A`' are given to the far right of output lines, after the right context. In default output format, when option '`-R`' is specified, references are rather given to the beginning of each output line, before the left context. For any other output format, option '`-R`' is almost ignored, except for the fact that the width of references is *not* taken into account in total output width given by '`-w`' whenever '`-R`' is selected.

  This option is automatically selected whenever GNU extensions are disabled.

'`-F` *string*'
'`--flac-truncation=`*string*'
: This option will request that any truncation in the output be reported using the string *string*. Most output fields theoretically extend towards the beginning or the end of the current line, or current sentence, as selected with option '`-S`'. But there is a maximum allowed output line width, changeable through option '`-w`', which is further divided into space for various output fields. When a field has to be truncated because cannot extend until the beginning or the end of the current line to fit in the, then a truncation occurs. By default, the string used is a single slash, as in '`-F /`'.

  string may have more than one character, as in '`-F ...`'. Also, in the particular case *string* is empty ('`-F ""`'), truncation flagging is disabled, and no truncation marks are appended in this case.

  As a matter of convenience to the user, many usual backslashed escape sequences, as found in the C language, are recognized and converted to the corresponding characters by `ptx` itself.

'`-M` *string*'
'`--macro-name=`*string*'
: Select another *string* to be used instead of '`xx`', while generating output suitable for `nroff`, `troff` or TEX.

'`-O`'
'`--format=roff`'
: Choose an output format suitable for `nroff` or `troff` processing. Each output line will look like:

  `.xx "`*tail*`" "`*before*`" "`*keyword_and_after*`" "`*head*`" "`*ref*`"`

  so it will be possible to write an '`.xx`' roff macro to take care of the output typesetting. This is the default output format when GNU extensions are disabled. Option '`-M`' might be used to change '`xx`' to another macro name.

  In this output format, each non-graphical character, like newline and tab, is merely changed to exactly one space, with no special attempt to compress

consecutive spaces. Each quote character: " is doubled so it will be correctly processed by `nroff` or `troff`.

'`-T`'
'`--format=tex`'
: Choose an output format suitable for TEX processing. Each output line will look like:

  `\xx {`*tail*`}{`*before*`}{`*keyword*`}{`*after*`}{`*head*`}{`*ref*`}`

  so it will be possible to write a `\xx` definition to take care of the output typesetting. Note that when references are not being produced, that is, neither option '`-A`' nor option '`-r`' is selected, the last parameter of each `\xx` call is inhibited. Option '`-M`' might be used to change '`xx`' to another macro name.

  In this output format, some special characters, like `$`, `%`, `&`, `#` and `_` are automatically protected with a backslash. Curly brackets `{`, `}` are also protected with a backslash, but also enclosed in a pair of dollar signs to force mathematical mode. The backslash itself produces the sequence `\backslash{}`. Circumflex and tilde diacritics produce the sequence `^\{ }` and `~\{ }` respectively. Other diacriticized characters of the underlying character set produce an appropriate TEX sequence as far as possible. The other non-graphical characters, like newline and tab, and all others characters which are not part of ASCII, are merely changed to exactly one space, with no special attempt to compress consecutive spaces. Let me know how to improve this special character processing for TEX.

7.5.5 The GNU extensions to `ptx`

This version of `ptx` contains a few features which do not exist in System V `ptx`. These extra features are suppressed by using the '`-G`' command line option, unless overridden by other command line options. Some GNU extensions cannot be recovered by overriding, so the simple rule is to avoid '`-G`' if you care about GNU extensions. Here are the differences between this program and System V `ptx`.

- This program can read many input files at once, it always writes the resulting concordance on standard output. On the other end, System V `ptx` reads only one file and produce the result on standard output or, if a second *file* parameter is given on the command, to that *file*.

  Having output parameters not introduced by options is a quite dangerous practice which GNU avoids as far as possible. So, for using `ptx` portably between GNU and System V, you should pay attention to always use it with a single input file, and always expect the result on standard output. You might also want to automatically configure in a '`-G`' option to `ptx` calls in products using `ptx`, if the configurator finds that the installed `ptx` accepts '`-G`'.
- The only options available in System V `ptx` are options '`-b`', '`-f`', '`-g`', '`-i`', '`-o`', '`-r`', '`-t`' and '`-w`'. All other options are GNU extensions and are not repeated in this enumeration. Moreover, some options have a slightly different meaning when GNU extensions are enabled, as explained below.
- By default, concordance output is not formatted for `troff` or `nroff`. It is rather formatted for a dumb terminal. `troff` or `nroff` output may still be selected through option '`-O`'.

- Unless '`-R`' option is used, the maximum reference width is subtracted from the total output line width. With GNU extensions disabled, width of references is not taken into account in the output line width computations.
- All 256 characters, even *NUL*s, are always read and processed from input file with no adverse effect, even if GNU extensions are disabled. However, System V `ptx` does not accept 8-bit characters, a few control characters are rejected, and the tilde `~` is condemned.
- Input line length is only limited by available memory, even if GNU extensions are disabled. However, System V `ptx` processes only the first 200 characters in each line.
- The break (non-word) characters default to be every character except all letters of the underlying character set, diacriticized or not. When GNU extensions are disabled, the break characters default to space, tab and newline only.
- The program makes better use of output line width. If GNU extensions are disabled, the program rather tries to imitate System V `ptx`, but still, there are some slight disposition glitches this program does not completely reproduce.
- The user can specify both an Ignore file and an Only file. This is not allowed with System V `ptx`.

8 Operating on fields within a line

8.1 `cut`: Print selected parts of lines

`cut` writes to standard output selected parts of each line of each input file, or standard input if no files are given or for a file name of '`-`'. Synopsis:

```
cut [option]... [file]...
```

In the table which follows, the *byte-list*, *character-list*, and *field-list* are one or more numbers or ranges (two numbers separated by a dash) separated by commas. Bytes, characters, and fields are numbered from starting at 1. Incomplete ranges may be given: '`-`*m*' means '`1-`*m*'; '*n*`-`' means '*n*' through end of line or last field.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-b` *byte-list*'
'`--bytes=`*byte-list*'
: Print only the bytes in positions listed in *byte-list*. Tabs and backspaces are treated like any other character; they take up 1 byte.

'`-c` *character-list*'
'`--characters=`*character-list*'
: Print only characters in positions listed in *character-list*. The same as '`-b`' for now, but internationalization will change that. Tabs and backspaces are treated like any other character; they take up 1 character.

'`-f` *field-list*'
'`--fields=`*field-list*'
: Print only the fields listed in *field-list*. Fields are separated by a TAB character by default.

'`-d` *input_delim_byte*'
'`--delimiter=`*input_delim_byte*'
: For '`-f`', fields are separated in the input by the first character in *input_delim_byte* (default is TAB).

'`-n`'
: Do not split multi-byte characters (no-op for now).

'`-s`'
'`--only-delimited`'
: For '`-f`', do not print lines that do not contain the field separator character.

'`--output-delimiter=`*output_delim_string*'
: For '`-f`', output fields are separated by *output_delim_string* The default is to use the input delimiter.

8.2 `paste`: Merge lines of files

`paste` writes to standard output lines consisting of sequentially corresponding lines of each given file, separated by a TAB character. Standard input is used for a file name of '`-`' or if no input files are given.

Synopsis:

```
paste [option]... [file]...
```

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-s`'
'`--serial`'
: Paste the lines of one file at a time rather than one line from each file.

'`-d` *delim-list*'
'`--delimiters` *delim-list*'
: Consecutively use the characters in *delim-list* instead of TAB to separate merged lines. When *delim-list* is exhausted, start again at its beginning.

8.3 `join`: Join lines on a common field

`join` writes to standard output a line for each pair of input lines that have identical join fields. Synopsis:

```
join [option]... file1 file2
```

Either *file1* or *file2* (but not both) can be '`-`', meaning standard input. *file1* and *file2* should be already sorted in increasing textual order on the join fields, using the collating sequence specified by the `LC_COLLATE` locale. Unless the '`-t`' option is given, the input should be sorted ignoring blanks at the start of the join field, as in `sort -b`. If the '`--ignore-case`' option is given, lines should be sorted without regard to the case of characters in the join field, as in `sort -f`.

The defaults are: the join field is the first field in each line; fields in the input are separated by one or more blanks, with leading blanks on the line ignored; fields in the output are separated by a space; each output line consists of the join field, the remaining fields from *file1*, then the remaining fields from *file2*.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-a` *file-number*'
: Print a line for each unpairable line in file *file-number* (either '`1`' or '`2`'), in addition to the normal output.

'`-e` *string*' Replace those output fields that are missing in the input with *string*.

'`-i`'
'`--ignore-case`'
: Ignore differences in case when comparing keys. With this option, the lines of the input files must be ordered in the same way. Use '`sort -f`' to produce this ordering.

'`-1` *field*'
'`-j1` *field*' Join on field *field* (a positive integer) of file 1.

'`-2` *field*'
'`-j2` *field*' Join on field *field* (a positive integer) of file 2.

'`-j` *field*' Equivalent to '`-1` *field* `-2` *field*'.

'`-o` *field-list*...'
Construct each output line according to the format in *field-list*. Each element in *field-list* is either the single character '`0`' or has the form *m.n* where the file number, *m*, is '`1`' or '`2`' and *n* is a positive field number.

A field specification of '`0`' denotes the join field. In most cases, the functionality of the '`0`' field spec may be reproduced using the explicit *m.n* that corresponds to the join field. However, when printing unpairable lines (using either of the '`-a`' or '`-v`' options), there is no way to specify the join field using *m.n* in *field-list* if there are unpairable lines in both files. To give `join` that functionality, POSIX invented the '`0`' field specification notation.

The elements in *field-list* are separated by commas or blanks. Multiple *field-list* arguments can be given after a single '`-o`' option; the values of all lists given with '`-o`' are concatenated together. All output lines – including those printed because of any -a or -v option – are subject to the specified *field-list*.

'`-t` *char*' Use character *char* as the input and output field separator.

'`-v` *file-number*'
Print a line for each unpairable line in file *file-number* (either '`1`' or '`2`'), instead of the normal output.

In addition, when GNU `join` is invoked with exactly one argument, options '`--help`' and '`--version`' are recognized. See Chapter 2 [Common options], page 2.

9 Operating on characters

This commands operate on individual characters.

9.1 `tr`: Translate, squeeze, and/or delete characters

Synopsis:

`tr` [*option*]... *set1* [*set2*]

`tr` copies standard input to standard output, performing one of the following operations:

- translate, and optionally squeeze repeated characters in the result,
- squeeze repeated characters,
- delete characters,
- delete characters, then squeeze repeated characters from the result.

The *set1* and (if given) *set2* arguments define ordered sets of characters, referred to below as *set1* and *set2*. These sets are the characters of the input that `tr` operates on. The '`--complement`' ('`-c`') option replaces *set1* with its complement (all of the characters that are not in *set1*).

9.1.1 Specifying sets of characters

The format of the *set1* and *set2* arguments resembles the format of regular expressions; however, they are not regular expressions, only lists of characters. Most characters simply represent themselves in these strings, but the strings can contain the shorthands listed below, for convenience. Some of them can be used only in *set1* or *set2*, as noted below.

Backslash escapes

A backslash followed by a character not listed below causes an error message.

'`\a`' Control-G.

'`\b`' Control-H.

'`\f`' Control-L.

'`\n`' Control-J.

'`\r`' Control-M.

'`\t`' Control-I.

'`\v`' Control-K.

'`\`*ooo*' The character with the value given by *ooo*, which is 1 to 3 octal digits,

'`\\`' A backslash.

Ranges

The notation '*m*`-`*n*' expands to all of the characters from *m* through *n*, in ascending order. *m* should collate before *n*; if it doesn't, an error results. As an example, '`0-9`' is the same as '`0123456789`'. Although GNU `tr` does

not support the System V syntax that uses square brackets to enclose ranges, translations specified in that format will still work as long as the brackets in *string1* correspond to identical brackets in *string2*.

Repeated characters

The notation '`[`*c*`*`*n*`]`' in *set2* expands to *n* copies of character *c*. Thus, '`[y*6]`' is the same as '`yyyyyy`'. The notation '`[`*c*`*]`' in *string2* expands to as many copies of *c* as are needed to make *set2* as long as *set1*. If *n* begins with '`0`', it is interpreted in octal, otherwise in decimal.

Character classes

The notation '`[:`*class*`:]`' expands to all of the characters in the (predefined) class *class*. The characters expand in no particular order, except for the `upper` and `lower` classes, which expand in ascending order. When the '`--delete`' ('`-d`') and '`--squeeze-repeats`' ('`-s`') options are both given, any character class can be used in *set2*. Otherwise, only the character classes `lower` and `upper` are accepted in *set2*, and then only if the corresponding character class (`upper` and `lower`, respectively) is specified in the same relative position in *set1*. Doing this specifies case conversion. The class names are given below; an error results when an invalid class name is given.

`alnum` Letters and digits.

`alpha` Letters.

`blank` Horizontal whitespace.

`cntrl` Control characters.

`digit` Digits.

`graph` Printable characters, not including space.

`lower` Lowercase letters.

`print` Printable characters, including space.

`punct` Punctuation characters.

`space` Horizontal or vertical whitespace.

`upper` Uppercase letters.

`xdigit` Hexadecimal digits.

Equivalence classes

The syntax '`[=`*c*`=]`' expands to all of the characters that are equivalent to *c*, in no particular order. Equivalence classes are a relatively recent invention intended to support non-English alphabets. But there seems to be no standard way to define them or determine their contents. Therefore, they are not fully implemented in GNU `tr`; each character's equivalence class consists only of that character, which is of no particular use.

9.1.2 Translating

`tr` performs translation when *set1* and *set2* are both given and the '`--delete`' ('`-d`') option is not given. `tr` translates each character of its input that is in *set1* to the corresponding character in *set2*. Characters not in *set1* are passed through unchanged. When a character appears more than once in *set1* and the corresponding characters in *set2* are not all the same, only the final one is used. For example, these two commands are equivalent:

```
tr aaa xyz
tr a z
```

A common use of `tr` is to convert lowercase characters to uppercase. This can be done in many ways. Here are three of them:

```
tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
tr a-z A-Z
tr '[:lower:]' '[:upper:]'
```

When `tr` is performing translation, *set1* and *set2* typically have the same length. If *set1* is shorter than *set2*, the extra characters at the end of *set2* are ignored.

On the other hand, making *set1* longer than *set2* is not portable; POSIX.2 says that the result is undefined. In this situation, BSD `tr` pads *set2* to the length of *set1* by repeating the last character of *set2* as many times as necessary. System V `tr` truncates *set1* to the length of *set2*.

By default, GNU `tr` handles this case like BSD `tr`. When the '`--truncate-set1`' ('`-t`') option is given, GNU `tr` handles this case like the System V `tr` instead. This option is ignored for operations other than translation.

Acting like System V `tr` in this case breaks the relatively common BSD idiom:

```
tr -cs A-Za-z0-9 '\012'
```

because it converts only zero bytes (the first element in the complement of *set1*), rather than all non-alphanumerics, to newlines.

9.1.3 Squeezing repeats and deleting

When given just the '`--delete`' ('`-d`') option, `tr` removes any input characters that are in *set1*.

When given just the '`--squeeze-repeats`' ('`-s`') option, `tr` replaces each input sequence of a repeated character that is in *set1* with a single occurrence of that character.

When given both '`--delete`' and '`--squeeze-repeats`', `tr` first performs any deletions using *set1*, then squeezes repeats from any remaining characters using *set2*.

The '`--squeeze-repeats`' option may also be used when translating, in which case `tr` first performs translation, then squeezes repeats from any remaining characters using *set2*.

Here are some examples to illustrate various combinations of options:

- Remove all zero bytes:

  ```
  tr -d '\000'
  ```

- Put all words on lines by themselves. This converts all non-alphanumeric characters to newlines, then squeezes each string of repeated newlines into a single newline:

  ```
  tr -cs 'a-zA-Z0-9' '[\n*]'
  ```

- Convert each sequence of repeated newlines to a single newline:

```
tr -s '\n'
```

- Find doubled occurrences of words in a document. For example, people often write "the the" with the duplicated words separated by a newline. The bourne shell script below works first by converting each sequence of punctuation and blank characters to a single newline. That puts each "word" on a line by itself. Next it maps all uppercase characters to lower case, and finally it runs `uniq` with the '`-d`' option to print out only the words that were adjacent duplicates.

```
#!/bin/sh
cat "$@" \
  | tr -s '[:punct:][:blank:]' '\n' \
  | tr '[:upper:]' '[:lower:]' \
  | uniq -d
```

9.1.4 Warning messages

Setting the environment variable `POSIXLY_CORRECT` turns off the following warning and error messages, for strict compliance with POSIX.2. Otherwise, the following diagnostics are issued:

1. When the '`--delete`' option is given but '`--squeeze-repeats`' is not, and *set2* is given, GNU `tr` by default prints a usage message and exits, because *set2* would not be used. The POSIX specification says that *set2* must be ignored in this case. Silently ignoring arguments is a bad idea.
2. When an ambiguous octal escape is given. For example, '`\400`' is actually '`\40`' followed by the digit '`0`', because the value 400 octal does not fit into a single byte.

GNU `tr` does not provide complete BSD or System V compatibility. For example, it is impossible to disable interpretation of the POSIX constructs '`[:alpha:]`', '`[=c=]`', and '`[c*10]`'. Also, GNU `tr` does not delete zero bytes automatically, unlike traditional Unix versions, which provide no way to preserve zero bytes.

9.2 `expand`: Convert tabs to spaces

`expand` writes the contents of each given *file*, or standard input if none are given or for a *file* of '`-`', to standard output, with tab characters converted to the appropriate number of spaces. Synopsis:

```
expand [option]... [file]...
```

By default, `expand` converts all tabs to spaces. It preserves backspace characters in the output; they decrement the column count for tab calculations. The default action is equivalent to '`-8`' (set tabs every 8 columns).

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-`*tab1*`[,`*tab2*`]...`'
'`-t` *tab1*`[,`*tab2*`]...`'
'`--tabs=`*tab1*`[,`*tab2*`]...`'
: If only one tab stop is given, set the tabs *tab1* spaces apart (default is 8). Otherwise, set the tabs at columns *tab1*, *tab2*, ... (numbered from 0), and

replace any tabs beyond the last tabstop given with single spaces. If the tabstops are specified with the '`-t`' or '`--tabs`' option, they can be separated by blanks as well as by commas.

'`-i`'
'`--initial`'
: Only convert initial tabs (those that precede all non-space or non-tab characters) on each line to spaces.

9.3 `unexpand`: Convert spaces to tabs

`unexpand` writes the contents of each given *file*, or standard input if none are given or for a *file* of '`-`', to standard output, with strings of two or more space or tab characters converted to as many tabs as possible followed by as many spaces as are needed. Synopsis:

`unexpand [`*option*`]... [`*file*`]...`

By default, `unexpand` converts only initial spaces and tabs (those that precede all non space or tab characters) on each line. It preserves backspace characters in the output; they decrement the column count for tab calculations. By default, tabs are set at every 8th column.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-`*tab1*`[,`*tab2*`]...`'
'`-t` *tab1*`[,`*tab2*`]...`'
'`--tabs=`*tab1*`[,`*tab2*`]...`'
: If only one tab stop is given, set the tabs *tab1* spaces apart instead of the default 8. Otherwise, set the tabs at columns *tab1*, *tab2*, ... (numbered from 0), and leave spaces and tabs beyond the tabstops given unchanged. If the tabstops are specified with the '`-t`' or '`--tabs`' option, they can be separated by blanks as well as by commas. This option implies the '`-a`' option.

'`-a`'
'`--all`'
: Convert all strings of two or more spaces or tabs, not just initial ones, to tabs.

10 Opening the software toolbox

This chapter originally appeared in *Linux Journal*, volume 1, number 2, in the *What's GNU?* column. It was written by Arnold Robbins.

Toolbox introduction

This month's column is only peripherally related to the GNU Project, in that it describes a number of the GNU tools on your Linux system and how they might be used. What it's really about is the "Software Tools" philosophy of program development and usage.

The software tools philosophy was an important and integral concept in the initial design and development of Unix (of which Linux and GNU are essentially clones). Unfortunately, in the modern day press of Internetworking and flashy GUIs, it seems to have fallen by the wayside. This is a shame, since it provides a powerful mental model for solving many kinds of problems.

Many people carry a Swiss Army knife around in their pants pockets (or purse). A Swiss Army knife is a handy tool to have: it has several knife blades, a screwdriver, tweezers, toothpick, nail file, corkscrew, and perhaps a number of other things on it. For the everyday, small miscellaneous jobs where you need a simple, general purpose tool, it's just the thing.

On the other hand, an experienced carpenter doesn't build a house using a Swiss Army knife. Instead, he has a toolbox chock full of specialized tools—a saw, a hammer, a screwdriver, a plane, and so on. And he knows exactly when and where to use each tool; you won't catch him hammering nails with the handle of his screwdriver.

The Unix developers at Bell Labs were all professional programmers and trained computer scientists. They had found that while a one-size-fits-all program might appeal to a user because there's only one program to use, in practice such programs are

a. difficult to write,
b. difficult to maintain and debug, and
c. difficult to extend to meet new situations.

Instead, they felt that programs should be specialized tools. In short, each program "should do one thing well." No more and no less. Such programs are simpler to design, write, and get right—they only do one thing.

Furthermore, they found that with the right machinery for hooking programs together, that the whole was greater than the sum of the parts. By combining several special purpose programs, you could accomplish a specific task that none of the programs was designed for, and accomplish it much more quickly and easily than if you had to write a special purpose program. We will see some (classic) examples of this further on in the column. (An important additional point was that, if necessary, take a detour and build any software tools you may need first, if you don't already have something appropriate in the toolbox.)

I/O redirection

Hopefully, you are familiar with the basics of I/O redirection in the shell, in particular the concepts of "standard input," "standard output," and "standard error". Briefly, "standard

input" is a data source, where data comes from. A program should not need to either know or care if the data source is a disk file, a keyboard, a magnetic tape, or even a punched card reader. Similarly, "standard output" is a data sink, where data goes to. The program should neither know nor care where this might be. Programs that only read their standard input, do something to the data, and then send it on, are called "filters", by analogy to filters in a water pipeline.

With the Unix shell, it's very easy to set up data pipelines:

```
program_to_create_data | filter1 | .... | filterN > final.pretty.data
```

We start out by creating the raw data; each filter applies some successive transformation to the data, until by the time it comes out of the pipeline, it is in the desired form.

This is fine and good for standard input and standard output. Where does the standard error come in to play? Well, think about `filter1` in the pipeline above. What happens if it encounters an error in the data it sees? If it writes an error message to standard output, it will just disappear down the pipeline into `filter2`'s input, and the user will probably never see it. So programs need a place where they can send error messages so that the user will notice them. This is standard error, and it is usually connected to your console or window, even if you have redirected standard output of your program away from your screen.

For filter programs to work together, the format of the data has to be agreed upon. The most straightforward and easiest format to use is simply lines of text. Unix data files are generally just streams of bytes, with lines delimited by the ASCII LF (Line Feed) character, conventionally called a "newline" in the Unix literature. (This is `'\n'` if you're a C programmer.) This is the format used by all the traditional filtering programs. (Many earlier operating systems had elaborate facilities and special purpose programs for managing binary data. Unix has always shied away from such things, under the philosophy that it's easiest to simply be able to view and edit your data with a text editor.)

OK, enough introduction. Let's take a look at some of the tools, and then we'll see how to hook them together in interesting ways. In the following discussion, we will only present those command line options that interest us. As you should always do, double check your system documentation for the full story.

The `who` command

The first program is the `who` command. By itself, it generates a list of the users who are currently logged in. Although I'm writing this on a single-user system, we'll pretend that several people are logged in:

```
$ who
arnold   console Jan 22 19:57
miriam   ttyp0   Jan 23 14:19(:0.0)
bill     ttyp1   Jan 21 09:32(:0.0)
arnold   ttyp2   Jan 23 20:48(:0.0)
```

Here, the '`$`' is the usual shell prompt, at which I typed `who`. There are three people logged in, and I am logged in twice. On traditional Unix systems, user names are never more than eight characters long. This little bit of trivia will be useful later. The output of `who` is nice, but the data is not all that exciting.

The `cut` command

The next program we'll look at is the `cut` command. This program cuts out columns or fields of input data. For example, we can tell it to print just the login name and full name from the '`/etc/passwd file`'. The '`/etc/passwd`' file has seven fields, separated by colons:

```
arnold:xyzzy:2076:10:Arnold D. Robbins:/home/arnold:/bin/ksh
```

To get the first and fifth fields, we would use cut like this:

```
$ cut -d: -f1,5 /etc/passwd
root:Operator
...
arnold:Arnold D. Robbins
miriam:Miriam A. Robbins
...
```

With the '`-c`' option, `cut` will cut out specific characters (i.e., columns) in the input lines. This command looks like it might be useful for data filtering.

The `sort` command

Next we'll look at the `sort` command. This is one of the most powerful commands on a Unix-style system; one that you will often find yourself using when setting up fancy data plumbing. The `sort` command reads and sorts each file named on the command line. It then merges the sorted data and writes it to standard output. It will read standard input if no files are given on the command line (thus making it into a filter). The sort is based on the character collating sequence or based on user-supplied ordering criteria.

The `uniq` command

Finally (at least for now), we'll look at the `uniq` program. When sorting data, you will often end up with duplicate lines, lines that are identical. Usually, all you need is one instance of each line. This is where `uniq` comes in. The `uniq` program reads its standard input, which it expects to be sorted. It only prints out one copy of each duplicated line. It does have several options. Later on, we'll use the '`-c`' option, which prints each unique line, preceded by a count of the number of times that line occurred in the input.

Putting the tools together

Now, let's suppose this is a large BBS system with dozens of users logged in. The management wants the SysOp to write a program that will generate a sorted list of logged in users. Furthermore, even if a user is logged in multiple times, his or her name should only show up in the output once.

The SysOp could sit down with the system documentation and write a C program that did this. It would take perhaps a couple of hundred lines of code and about two hours to write it, test it, and debug it. However, knowing the software toolbox, the SysOp can instead start out by generating just a list of logged on users:

```
$ who | cut -c1-8
arnold
```

```
miriam
bill
arnold
```

Next, sort the list:

```
$ who | cut -c1-8 | sort
arnold
arnold
bill
miriam
```

Finally, run the sorted list through `uniq`, to weed out duplicates:

```
$ who | cut -c1-8 | sort | uniq
arnold
bill
miriam
```

The `sort` command actually has a '`-u`' option that does what `uniq` does. However, `uniq` has other uses for which one cannot substitute '`sort -u`'.

The SysOp puts this pipeline into a shell script, and makes it available for all the users on the system:

```
# cat > /usr/local/bin/listusers
who | cut -c1-8 | sort | uniq
^D
# chmod +x /usr/local/bin/listusers
```

There are four major points to note here. First, with just four programs, on one command line, the SysOp was able to save about two hours worth of work. Furthermore, the shell pipeline is just about as efficient as the C program would be, and it is much more efficient in terms of programmer time. People time is much more expensive than computer time, and in our modern "there's never enough time to do everything" society, saving two hours of programmer time is no mean feat.

Second, it is also important to emphasize that with the *combination* of the tools, it is possible to do a special purpose job never imagined by the authors of the individual programs.

Third, it is also valuable to build up your pipeline in stages, as we did here. This allows you to view the data at each stage in the pipeline, which helps you acquire the confidence that you are indeed using these tools correctly.

Finally, by bundling the pipeline in a shell script, other users can use your command, without having to remember the fancy plumbing you set up for them. In terms of how you run them, shell scripts and compiled programs are indistinguishable.

After the previous warm-up exercise, we'll look at two additional, more complicated pipelines. For them, we need to introduce two more tools.

The first is the `tr` command, which stands for "transliterate." The `tr` command works on a character-by-character basis, changing characters. Normally it is used for things like mapping upper case to lower case:

```
$ echo ThIs ExAmPlE HaS MIXED case! | tr '[A-Z]' '[a-z]'
this example has mixed case!
```

There are several options of interest:

'`-c`' work on the complement of the listed characters, i.e., operations apply to characters not in the given set

'`-d`' delete characters in the first set from the output

'`-s`' squeeze repeated characters in the output into just one character.

We will be using all three options in a moment.

The other command we'll look at is `comm`. The `comm` command takes two sorted input files as input data, and prints out the files' lines in three columns. The output columns are the data lines unique to the first file, the data lines unique to the second file, and the data lines that are common to both. The '`-1`', '`-2`', and '`-3`' command line options omit the respective columns. (This is non-intuitive and takes a little getting used to.) For example:

```
$ cat f1
11111
22222
33333
44444
$ cat f2
00000
22222
33333
55555
$ comm f1 f2
        00000
11111
                22222
                33333
44444
        55555
```

The single dash as a filename tells `comm` to read standard input instead of a regular file.

Now we're ready to build a fancy pipeline. The first application is a word frequency counter. This helps an author determine if he or she is over-using certain words.

The first step is to change the case of all the letters in our input file to one case. "The" and "the" are the same word when doing counting.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | ...
```

The next step is to get rid of punctuation. Quoted words and unquoted words should be treated identically; it's easiest to just get the punctuation out of the way.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' | ...
```

The second `tr` command operates on the complement of the listed characters, which are all the letters, the digits, the underscore, and the blank. The '`\012`' represents the newline character; it has to be left alone. (The ASCII tab character should also be included for good measure in a production script.)

At this point, we have data consisting of words separated by blank space. The words only contain alphanumeric characters (and the underscore). The next step is break the data apart so that we have one word per line. This makes the counting operation much easier, as we will see shortly.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | ...
```

This command turns blanks into newlines. The '`-s`' option squeezes multiple newline characters in the output into just one. This helps us avoid blank lines. (The '`>`' is the shell's "secondary prompt." This is what the shell prints when it notices you haven't finished typing in all of a command.)

We now have data consisting of one word per line, no punctuation, all one case. We're ready to count each word:

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort | uniq -c | ...
```

At this point, the data might look something like this:

```
 60 a
  2 able
  6 about
  1 above
  2 accomplish
  1 acquire
  1 actually
  2 additional
```

The output is sorted by word, not by count! What we want is the most frequently used words first. Fortunately, this is easy to accomplish, with the help of two more `sort` options:

'`-n`' do a numeric sort, not a textual one

'`-r`' reverse the order of the sort

The final pipeline looks like this:

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort | uniq -c | sort -nr
 156 the
  60 a
  58 to
  51 of
  51 and
 ...
```

Whew! That's a lot to digest. Yet, the same principles apply. With six commands, on two lines (really one long one split for convenience), we've created a program that does something interesting and useful, in much less time than we could have written a C program to do the same thing.

A minor modification to the above pipeline can give us a simple spelling checker! To determine if you've spelled a word correctly, all you have to do is look it up in a dictionary. If it is not there, then chances are that your spelling is incorrect. So, we need a dictionary. If you have the Slackware Linux distribution, you have the file '`/usr/lib/ispell/ispell.words`', which is a sorted, 38,400 word dictionary.

Now, how to compare our file with the dictionary? As before, we generate a sorted list of words, one per line:

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort -u | ...
```

Now, all we need is a list of words that are *not* in the dictionary. Here is where the `comm` command comes in.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort -u |
> comm -23 - /usr/lib/ispell/ispell.words
```

The '`-2`' and '`-3`' options eliminate lines that are only in the dictionary (the second file), and lines that are in both files. Lines only in the first file (standard input, our stream of words), are words that are not in the dictionary. These are likely candidates for spelling errors. This pipeline was the first cut at a production spelling checker on Unix.

There are some other tools that deserve brief mention.

`grep`	search files for text that matches a regular expression
`egrep`	like `grep`, but with more powerful regular expressions
`wc`	count lines, words, characters
`tee`	a T-fitting for data pipes, copies data to files and to standard output
`sed`	the stream editor, an advanced tool
`awk`	a data manipulation language, another advanced tool

The software tools philosophy also espoused the following bit of advice: "Let someone else do the hard part." This means, take something that gives you most of what you need, and then massage it the rest of the way until it's in the form that you want.

To summarize:

1. Each program should do one thing well. No more, no less.
2. Combining programs with appropriate plumbing leads to results where the whole is greater than the sum of the parts. It also leads to novel uses of programs that the authors might never have imagined.
3. Programs should never print extraneous header or trailer data, since these could get sent on down a pipeline. (A point we didn't mention earlier.)
4. Let someone else do the hard part.
5. Know your toolbox! Use each program appropriately. If you don't have an appropriate tool, build one.

As of this writing, all the programs we've discussed are available via anonymous `ftp` from `prep.ai.mit.edu` as '`/pub/gnu/textutils-1.9.tar.gz`'.[1]

None of what I have presented in this column is new. The Software Tools philosophy was first introduced in the book *Software Tools*, by Brian Kernighan and P.J. Plauger (Addison-Wesley, ISBN 0-201-03669-X). This book showed how to write and use software tools. It was written in 1976, using a preprocessor for FORTRAN named `ratfor` (RATional FORtran). At the time, C was not as ubiquitous as it is now; FORTRAN was. The last

[1] Version 1.9 was current when this column was written. Check the nearest GNU archive for the current version. The main GNU FTP site is now `ftp.gnu.org`.

chapter presented a `ratfor` to FORTRAN processor, written in `ratfor`. `ratfor` looks an awful lot like C; if you know C, you won't have any problem following the code.

In 1981, the book was updated and made available as *Software Tools in Pascal* (Addison-Wesley, ISBN 0-201-10342-7). Both books remain in print, and are well worth reading if you're a programmer. They certainly made a major change in how I view programming.

Initially, the programs in both books were available (on 9-track tape) from Addison-Wesley. Unfortunately, this is no longer the case, although you might be able to find copies floating around the Internet. For a number of years, there was an active Software Tools Users Group, whose members had ported the original `ratfor` programs to essentially every computer system with a FORTRAN compiler. The popularity of the group waned in the middle '80s as Unix began to spread beyond universities.

With the current proliferation of GNU code and other clones of Unix programs, these programs now receive little attention; modern C versions are much more efficient and do more than these programs do. Nevertheless, as exposition of good programming style, and evangelism for a still-valuable philosophy, these books are unparalleled, and I recommend them highly.

Acknowledgment: I would like to express my gratitude to Brian Kernighan of Bell Labs, the original Software Toolsmith, for reviewing this column.

Index

-

--across ... 12
--address-radix ... 6
--all ... 46
--all-repeated ... 30
--before ... 4
--binary ... 3, 25
--body-numbering ... 5
--bytes ... 15, 17, 18, 20, 23, 39
--characters ... 39
--chars ... 23
--check-chars ... 30
--columns ... 12
--count ... 30
--crown-margin ... 10
--delimiter ... 39
--delimiters ... 40
--digits ... 22
--double-space ... 12
--elide-empty-files ... 22
--expand-tabs ... 12
--fields ... 39
--first-line-number ... 14
--follow ... 18
--footer-numbering ... 5
--form-feed ... 12
--format ... 7
--header ... 13
--header-numbering ... 5
--help ... 2
--ignore-case ... 30, 40
--indent ... 14
--initial ... 46
--join-blank-lines ... 6
--join-lines ... 13
--keep-files ... 22
--length ... 13
--line-bytes ... 20
--lines ... 17, 20, 23
--max-consecutive-size-changes ... 19
--max-line-length ... 23
--max-unchanged-stats ... 19
--merge ... 13
--no-file-warnings ... 14
--no-renumber ... 6
--number ... 3
--number-format ... 6
--number-lines ... 13
--number-nonblank ... 3
--number-separator ... 6
--number-width ... 6
--omit-header ... 14
--omit-pagination ... 15
--only-delimited ... 39
--output-delimiter ... 39
--output-duplicates ... 8
--output-tabs ... 13
--page-increment ... 5
--page_width ... 15
--pages ... 12
--pid ... 19
--prefix ... 21
--quiet ... 17, 20, 22
--read-bytes ... 7
--regex ... 4
--repeated ... 30
--retry ... 19
--section-delimiter ... 5
--sep-string ... 14
--separator ... 4, 14
--serial ... 40
--show-all ... 3
--show-control-chars ... 12
--show-ends ... 3
--show-nonprinting ... 4, 15
--show-tabs ... 4
--silent ... 17, 20, 22
--skip-bytes ... 7
--skip-chars ... 30
--skip-fields ... 30
--sleep-interval ... 19
--spaces ... 15
--split-only ... 10
--squeeze-blank ... 3
--starting-line-number ... 6
--status ... 25
--strings ... 7
--suffix ... 21
--sysv ... 24
--tabs ... 45, 46
--tagged-paragraph ... 10
--text ... 25
--traditional ... 8
--uniform-spacing ... 10
--unique ... 30
--verbose ... 17, 20
--version ... 2
--warn ... 25
--width ... 8, 11, 15, 16
--words ... 23
-1 ... 31, 40
-2 ... 31, 40
-3 ... 31
-a ... 8, 12, 40, 46

-A ... 3, 6
-b ... 3, 4, 5, 8, 15, 20, 21, 25, 27, 39
-B ... 3
-c ... 8, 10, 12, 17, 18, 23, 26, 30, 39
-C ... 20
-*column* ... 12
-*count* ... 17, 18
-d ... 5, 8, 12, 27, 30, 39, 40
-D ... 30
-e ... 3, 12, 40
-E ... 3
-f ... 5, 8, 12, 18, 21, 27, 30, 39
-F ... 12
-g ... 27
-h ... 5, 8, 13
-i ... 5, 8, 13, 27, 30, 40, 46
-j ... 7
-J ... 13
-j1 ... 40
-j2 ... 40
-k ... 22, 28
-l ... 6, 8, 13, 20, 23
-L ... 23
-m ... 13, 26
-M ... 27
-n ... 3, 6, 13, 17, 20, 22, 27
-*n* ... 30
-n ... 39
-N ... 7, 14
-o ... 8, 14, 28
-p ... 6
-q ... 17, 20, 22
-r ... 4, 14, 24, 27
-s ... 3, 4, 6, 7, 10, 14, 15, 22, 24, 30, 39, 40
-S ... 14
-t ... 4, 7, 10, 14, 25, 28, 45, 46
-T ... 4, 15
-*tab* ... 45, 46
-u ... 4, 10, 28, 30
-v ... 4, 6, 8, 15, 17, 20
-w ... 6, 8, 11, 15, 16, 23, 25, 30
-W ... 15
-*width* ... 11
-x ... 8
-z ... 22, 28

+

+*count* ... 18
+*first_page*[:*last_page*] ... 12
+*n* ... 30

1

128-bit checksum ... 24
16-bit checksum ... 23

A

across columns ... 12
all duplicate lines, outputting ... 30
`alnum` ... 43
`alpha` ... 43
ASCII dump of files ... 6

B

backslash escapes ... 42
binary and text I/O in cat ... 3
binary input files ... 25
`blank` ... 43
blank lines, numbering ... 6
blanks, ignoring leading ... 27
body, numbering ... 4
BSD `sum` ... 24
BSD `tail` ... 18
bugs, reporting ... 1
byte count ... 23

C

case folding ... 27
`cat` ... 3
characters classes ... 43
checking for sortedness ... 26
checksum, 128-bit ... 24
checksum, 16-bit ... 23
`cksum` ... 24
`cntrl` ... 43
`comm` ... 31
common field, joining on ... 40
common lines ... 31
common options ... 2
comparing sorted files ... 31
concatenate and write files ... 3
context splitting ... 21
converting tabs to spaces ... 45
copying files ... 3
CRC checksum ... 24
crown margin ... 10
`csplit` ... 21
`cut` ... 39
cyclic redundancy check ... 24

D

deleting characters ... 44
`descriptor` follow option ... 18
differing lines ... 31
`digit` ... 43
double spacing ... 12
down columns ... 12
duplicate lines, outputting ... 30

E

empty lines, numbering ... 6
entire files, output of ... 3
equivalence classes ... 43
`expand` ... 45

F

field separator character ... 28
file contents, dumping unambiguously ... 6
file offset radix ... 6
fingerprint, 128-bit ... 24
first part of files, outputting ... 17
`fmt` ... 10
`fold` ... 15
folding long input lines ... 15
footers, numbering ... 4
formatting file contents ... 10

G

general numeric sort ... 27
`graph` ... 43
growing files ... 18

H

`head` ... 17
headers, numbering ... 4
help, online ... 2
hex dump of files ... 6

I

indenting lines ... 14
initial part of files, outputting ... 17
initial tabs, converting ... 46
input tabs ... 12
introduction ... 1

J

`join` ... 40

K

Knuth, Donald E. ... 10

L

last part of files, outputting ... 17
`LC_COLLATE` ... 26, 31, 40
`LC_CTYPE` ... 26
`LC_NUMERIC` ... 27
`LC_TIME` ... 27
left margin ... 14
line count ... 23
line numbering ... 4
line-breaking ... 10
line-by-line comparison ... 31
`ln` format for `nl` ... 6
logical pages, numbering on ... 4
`lower` ... 43

M

`md5sum` ... 24
merging files ... 39
merging files in parallel ... 11
merging sorted files ... 26
message-digest, 128-bit ... 24
months, sorting by ... 27
multicolumn output, generating ... 11

N

`name` follow option ... 18
`nl` ... 4
numbering lines ... 4
numeric sort ... 27

O

octal dump of files ... 6
`od` ... 6
operating on characters ... 42
operating on sorted files ... 26
output file name prefix ... 20, 21
output file name suffix ... 21
output of entire files ... 3
output of parts of files ... 17
output tabs ... 13

overwriting of input, allowed ... 28

P

paragraphs, reformatting ... 10
parts of files, output of ... 17
`paste` ... 39
phone directory order ... 27
pieces, splitting a file into ... 20
Plass, Michael F. ... 10
POSIX.2 ... 1
`POSIXLY_CORRECT` ... 45
`pr` ... 11
`print` ... 43
printing, preparing files for ... 11
`ptx` ... 32
`punct` ... 43

R

radix for file offsets ... 6
ranges ... 42
reformatting paragraph text ... 10
repeated characters ... 43
reverse sorting ... 27
reversing files ... 4
`rn` format for `nl` ... 6
`rz` format for `nl` ... 6

S

screen columns ... 15
section delimiters of pages ... 5
sentences and line-breaking ... 10
`sort` ... 26
sort field ... 28
sort zero-terminated lines ... 28
sorted files, operations on ... 26
sorting files ... 26
`space` ... 43
specifying sets of characters ... 42
`split` ... 20
splitting a file into pieces ... 20
splitting a file into pieces by context ... 21
squeezing blank lines ... 3
squeezing repeat characters ... 44
string constants, outputting ... 7
`sum` ... 23
summarizing files ... 23
System V `sum` ... 24

T

tabs to spaces, converting ... 45
tabstops, setting ... 45
`tac` ... 4
tagged paragraphs ... 10
`tail` ... 17
telephone directory order ... 27
text input files ... 25
text, reformatting ... 10
`TMPDIR` ... 26
topological sort ... 31
total counts ... 23
`tr` ... 42
translating characters ... 44
`tsort` ... 31
type size ... 7

U

`unexpand` ... 46
`uniq` ... 29
unique lines, outputting ... 30
uniquify files ... 29
uniquifying output ... 28
unprintable characters, ignoring ... 27
`upper` ... 43

V

verifying MD5 checksums ... 25
version number, finding ... 2

W

`wc` ... 23
word count ... 23
wrapping long input lines ... 15

X

`xdigit` ... 43

Table of Contents

1 Introduction 1

2 Common options 2

3 Output of entire files 3

- 3.1 `cat`: Concatenate and write files 3
- 3.2 `tac`: Concatenate and write files in reverse 4
- 3.3 `nl`: Number lines and write files 4
- 3.4 `od`: Write files in octal or other formats 6

4 Formatting file contents 10

- 4.1 `fmt`: Reformat paragraph text 10
- 4.2 `pr`: Paginate or columnate files for printing 11
- 4.3 `fold`: Wrap input lines to fit in specified width 15

5 Output of parts of files 17

- 5.1 `head`: Output the first part of files 17
- 5.2 `tail`: Output the last part of files 17
- 5.3 `split`: Split a file into fixed-size pieces 20
- 5.4 `csplit`: Split a file into context-determined pieces 21

6 Summarizing files 23

- 6.1 `wc`: Print byte, word, and line counts 23
- 6.2 `sum`: Print checksum and block counts 23
- 6.3 `cksum`: Print CRC checksum and byte counts 24
- 6.4 `md5sum`: Print or check message-digests 24

7 Operating on sorted files 26

- 7.1 `sort`: Sort text files 26
- 7.2 `uniq`: Uniquify files 29
- 7.3 `comm`: Compare two sorted files line by line 31
- 7.4 `tsort`: Topological sort 31
- 7.5 `ptx`: Produce permuted indexes 32
  - 7.5.1 General options 32
  - 7.5.2 Charset selection 33
  - 7.5.3 Word selection and input processing 33
  - 7.5.4 Output formatting 35
  - 7.5.5 The GNU extensions to `ptx` 37

8 Operating on fields within a line 39

8.1 `cut`: Print selected parts of lines 39
8.2 `paste`: Merge lines of files 39
8.3 `join`: Join lines on a common field 40

9 Operating on characters 42

9.1 `tr`: Translate, squeeze, and/or delete characters 42
9.1.1 Specifying sets of characters 42
9.1.2 Translating 44
9.1.3 Squeezing repeats and deleting 44
9.1.4 Warning messages 45
9.2 `expand`: Convert tabs to spaces 45
9.3 `unexpand`: Convert spaces to tabs 46

10 Opening the software toolbox 47

Toolbox introduction 47
I/O redirection 47
The `who` command 48
The `cut` command 49
The `sort` command 49
The `uniq` command 49
Putting the tools together 49

Index 55