GNU `textutils`

A set of text utilities
for version 1.19, 27 June 1996

David MacKenzie et al.

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-gnu-utils@prep.ai.mit.edu`'. 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 is based on the Unix man pages in the distribution, which were originally written by David MacKenzie and updated by Jim Meyering. The original `fmt` man page was written by Ross Paterson. Francois 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`’
‘`--number-nonblank`’
: Number all nonblank output lines, starting with 1.

‘`-e`’
: Equivalent to ‘`-vE`’.

‘`-E`’
‘`--show-ends`’
: Display a ‘`$`’ after the end of each line.

‘`-n`’
‘`--number`’
: Number all output lines, starting with 1.

‘`-s`’
‘`--squeeze-blank`’
: Replace multiple adjacent blank lines with a single blank line.

‘`-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-'.

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.

'-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 *`TAB`*).

'`-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.

'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. Synopsis:

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

By default, a 5-line header is printed: two blank lines; a line with the date, the file name, and the page count; and two more blank lines. A five line footer (entirely) is also printed.

Form feeds in the input cause page breaks in the output.

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

'+*page*'
: Begin printing with page *page*.

'-*column*' Produce *column*-column output and print columns down. The column width is automatically decreased as *column* increases; unless you use the '-w' option to increase the page width as well, this option might well cause some input to be truncated.

'-a' Print columns across rather than down.

'-b' Balance columns on the last page.

'-c' 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 the output.

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

'-f'
'-F' Use a formfeed instead of newlines to separate output pages.

'-h *header*'
: Replace the file name in the header with the string *header*.

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

'-l *n*' Set the page length to *n* (default 66) lines. If *n* is less than 10, the headers and footers are omitted, as if the '-t' option had been given.

'-m' Print all files in parallel, one in each column.

'-n[*number-separator*[*digits*]]'
: Precede each column with a line number; with parallel files ('-m'), precede each line with a line number. Optional argument *number-separator* is the character to print after each number (default is *TAB*). Optional argument *digits* is the number of digits per line number (default is 5).

'-o *n*' Indent each line with *n* (default is zero) spaces wide, i.e., set the left margin. The total page width is 'n' plus the width set with the '-w' option.

'-r' Do not print a warning message when an argument *file* cannot be opened. (The exit status will still be nonzero, however.)

'-s[*c*]' Separate columns by the single character *c*. If *c* is omitted, the default is space; if this option is omitted altogether, the default is *TAB*.

'-t' Do not print the usual 5-line header and the 5-line footer on each page, and do not fill out the bottoms of pages (with blank lines or formfeeds).

'-v' Print unprintable characters in octal backslash notation.

'-w *n*' Set the page width to *n* (default is 72) columns.

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'

Loop forever trying to read more characters at the end of the file, presumably because the file is growing. Ignored if 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.

'-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=`*bytes*'
: 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, `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.

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.

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. Synopsis:

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

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.

'`-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.

'`--string=`*string*'
: Compute the message digest for *string*, instead of for a file. The result is the same as for a file that contains exactly *string*.

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

'`-w`'
'`--warn`'
: When verifying checksums, warn about improperly formated 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.

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 lines byte by byte in machine collating sequence. 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.

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.

'`-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, but use strtod(3) to arrive at the numeric values. This allows floating point numbers to be specified in scientific notation, like `1.0e-34` and `10e100`. Use this option only if there is no alternative; it is much slower than '`-n`' and numbers with too many significant digits will be compared as if they had been truncated. In addition, numbers outside the range of representable double precision floating point numbers are treated as if they were zeroes; overflow and underflow are not reported.

'`-i`' Ignore characters outside the printable ASCII range 040-0176 octal (inclusive) when sorting.

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

'`-n`' Sort numerically: the number begins each line; specifically, it consists of optional whitespace, an optional '`-`' sign, and zero or more digits, optionally followed by a decimal point and zero or more digits.

`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 decimal points 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 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. See below.

'-z' Treat the input as a set of lines, each terminated by a zero byte (ASCII NUL (Null) character) instead of a 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. If a '`-n`' or '`-M`' option is used, thus implying a '`-b`' option, the '`-b`' option is taken to apply to both the '`+`*pos*' and the '`-`*pos*' parts of a key specification. 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: Uniqify 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.

'-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*

The input files must be sorted before `comm` can be used.

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 *TAB*.

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

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`* by default.

'`-d` *delim*'
'`--delimiter=`*delim*'
: For '`-f`', fields are separated by the first character in *delim* (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.

8.2 `paste`: Merge lines of files

`paste` writes to standard output lines consisting of sequentially corresponding lines of each given file, separated by *TAB*. 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 order (not numerically) on the join fields; unless the '`-t`' option is given, they 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'
  ```

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 machine collating sequence (ASCII) 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 an ASCII 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`' directory.[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 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

[1] Version 1.9 was current when this column was written. Check the nearest GNU archive for the current version.

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

-

Table of Contents