

```
1  /* 'dir', 'vdir' and 'ls' directory listing programs for GNU.
2  Copyright (C) 1985-2018 Free Software Foundation, Inc.
3
4  This program is free software: you can redistribute it and/or modify
5  it under the terms of the GNU General Public License as published by
6  the Free Software Foundation, either version 3 of the License, or
7  (at your option) any later version.
8
9  This program is distributed in the hope that it will be useful,
10 but WITHOUT ANY WARRANTY; without even the implied warranty of
11 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 GNU General Public License for more details.
13
14 You should have received a copy of the GNU General Public License
15 along with this program. If not, see <https://www.gnu.org/licenses/>. */
16
17 /* If ls_mode is LS_MULTI_COL,
18    the multi-column format is the default regardless
19    of the type of output device.
20    This is for the 'dir' program.
21
22    If ls_mode is LS_LONG_FORMAT,
23    the long format is the default regardless of the
24    type of output device.
25    This is for the 'vdir' program.
26
27    If ls_mode is LS_LS,
28    the output format depends on whether the output
29    device is a terminal.
30    This is for the 'ls' program. */
31
32 /* Written by Richard Stallman and David MacKenzie. */
33
34 /* Color support by Peter Anvin <Peter.Anvin@linux.org> and Dennis
35    Flaherty <dennisf@denix.elk.miles.com> based on original patches by
36    Greg Lee <lee@uhunix.uhcc.hawaii.edu>. */
37
38 #include <config.h>
39 #include <sys/types.h>
40
41 #include <termios.h>
42 #if HAVE_STROPTS_H
43 # include <stropts.h>
44 #endif
```

```
45 #include <sys/ioctl.h>
46
47 #ifdef WINSIZE_IN_PTEM
48 # include <sys/stream.h>
49 # include <sys/ptem.h>
50 #endif
51
52 #include <stdio.h>
53 #include <assert.h>
54 #include <setjmp.h>
55 #include <pwd.h>
56 #include <getopt.h>
57 #include <signal.h>
58 #include <selinux/selinux.h>
59 #include <wchar.h>
60
61 #if HAVE_LANGINFO_CODESET
62 # include <langinfo.h>
63 #endif
64
65 /* Use SA_NOCLDSTOP as a proxy for whether the sigaction machinery is
66    present. */
67 #ifndef SA_NOCLDSTOP
68 # define SA_NOCLDSTOP 0
69 # define sigprocmask(How, Set, Oset) /* empty */
70 # define sigset_t int
71 # if ! HAVE_SIGINTERRUPT
72 # define siginterrupt(sig, flag) /* empty */
73 # endif
74 #endif
75
76 /* NonStop circa 2011 lacks both SA_RESTART and siginterrupt, so don't
77    restart syscalls after a signal handler fires. This may cause
78    colors to get messed up on the screen if 'ls' is interrupted, but
79    that's the best we can do on such a platform. */
80 #ifndef SA_RESTART
81 # define SA_RESTART 0
82 #endif
83
84 #include "system.h"
85 #include <fnmatch.h>
86
87 #include "acl.h"
88 #include "argmatch.h"
89 #include "c-strcase.h"
90 #include "dev-ino.h"
```

```

91 #include "die.h"
92 #include "error.h"
93 #include "filenamecat.h"
94 #include "hard-locale.h"
95 #include "hash.h"
96 #include "human.h"
97 #include "filemode.h"
98 #include "filevercmp.h"
99 #include "idcache.h"
100 #include "ls.h"
101 #include "mbswidth.h"
102 #include "mpsort.h"
103 #include "obstack.h"
104 #include "quote.h"
105 #include "smack.h"
106 #include "stat-size.h"
107 #include "stat-time.h"
108 #include "strftime.h"
109 #include "xdectoint.h"
110 #include "xstrtol.h"
111 #include "areadlink.h"
112 #include "mbsalign.h"
113 #include "dircolors.h"
114 #include "xgethostname.h"
115 #include "c-ctype.h"
116 #include "canonicalize.h"
117
118 /* Include <sys/capability.h> last to avoid a clash of <sys/types.h>
119    include guards with some premature versions of libcap.
120    For more details, see <https://bugzilla.redhat.com/483548>. */
121 #ifdef HAVE_CAP
122 # include <sys/capability.h>
123 #endif
124
125 #define PROGRAM_NAME (ls_mode == LS_LS ? "ls" \
126                      : (ls_mode == LS_MULTI_COL \
127                        ? "dir" : "vdir"))
128
129 #define AUTHORS \
130     proper_name ("Richard M. Stallman"), \
131     proper_name ("David MacKenzie")
132
133 #define obstack_chunk_alloc malloc
134 #define obstack_chunk_free free
135
136 /* Return an int indicating the result of comparing two integers.

```

```

137 Subtracting doesn't always work, due to overflow. */
138 #define longdiff(a, b) ((a) < (b) ? -1 : (a) > (b))
139
140 /* Unix-based readdir implementations have historically returned a dirent.d_ino
141    value that is sometimes not equal to the stat-obtained st_ino value for
142    that same entry. This error occurs for a readdir entry that refers
143    to a mount point. readdir's error is to return the inode number of
144    the underlying directory -- one that typically cannot be stat'ed, as
145    long as a file system is mounted on that directory. RELIABLE_D_INO
146    encapsulates whether we can use the more efficient approach of relying
147    on readdir-supplied d_ino values, or whether we must incur the cost of
148    calling stat or lstat to obtain each guaranteed-valid inode number. */
149
150 #ifndef READDIR_LIES_ABOUT_MOUNTPOINT_D_INO
151 # define READDIR_LIES_ABOUT_MOUNTPOINT_D_INO 1
152 #endif
153
154 #if READDIR_LIES_ABOUT_MOUNTPOINT_D_INO
155 # define RELIABLE_D_INO(dp) NOT_AN_INODE_NUMBER
156 #else
157 # define RELIABLE_D_INO(dp) D_INO (dp)
158 #endif
159
160 #if ! HAVE_STRUCT_STAT_ST_AUTHOR
161 # define st_author st_uid
162 #endif
163
164 enum filetype
165 {
166     unknown,
167     fifo,
168     chardev,
169     directory,
170     blockdev,
171     normal,
172     symbolic_link,
173     sock,
174     whiteout,
175     arg_directory
176 };
177
178 /* Display letters and indicators for each filetype.
179    Keep these in sync with enum filetype. */
180 static char const filetype_letter[] = "?pcdb-lswd";
181
182 /* Ensure that filetype and filetype_letter have the same

```

```

183     number of elements. */
184 verify (sizeof filetype_letter - 1 == arg_directory + 1);
185
186 #define FILETYPE_INDICATORS
187 {
188     C_ORPHAN, C_FIFO, C_CHR, C_DIR, C_BLK, C_FILE,
189     C_LINK, C_SOCK, C_FILE, C_DIR
190 }
191
192 enum acl_type
193 {
194     ACL_T_NONE,
195     ACL_T_LSM_CONTEXT_ONLY,
196     ACL_T_YES
197 };
198
199 struct fileinfo
200 {
201     /* The file name. */
202     char *name;
203
204     /* For symbolic link, name of the file linked to, otherwise zero. */
205     char *linkname;
206
207     /* For terminal hyperlinks. */
208     char *absolute_name;
209
210     struct stat stat;
211
212     enum filetype filetype;
213
214     /* For symbolic link and long listing, st_mode of file linked to, otherwise
215        zero. */
216     mode_t linkmode;
217
218     /* security context. */
219     char *scontext;
220
221     bool stat_ok;
222
223     /* For symbolic link and color printing, true if linked-to file
224        exists, otherwise false. */
225     bool linkok;
226
227     /* For long listings, true if the file has an access control list,
228        or a security context. */

```

```

229     enum acl_type acl_type;
230
231     /* For color listings, true if a regular file has capability info. */
232     bool has_capability;
233
234     /* Whether file name needs quoting. tri-state with -1 == unknown. */
235     int quoted;
236 };
237
238 #define LEN_STR_PAIR(s) sizeof(s) - 1, s
239
240 /* Null is a valid character in a color indicator (think about Epson
241    printers, for example) so we have to use a length/buffer string
242    type. */
243
244 struct bin_str
245 {
246     size_t len;                /* Number of bytes */
247     const char *string;        /* Pointer to the same */
248 };
249
250 #if ! HAVE_TCGETPGRP
251 # define tcgetpgrp(Fd) 0
252 #endif
253
254 static size_t quote_name (char const *name,
255                          struct quoting_options const *options,
256                          int needs_general_quoting,
257                          const struct bin_str *color,
258                          bool allow_pad, struct obstack *stack,
259                          char const *absolute_name);
260 static size_t quote_name_buf (char **inbuf, size_t bufsz, char *name,
261                              struct quoting_options const *options,
262                              int needs_general_quoting, size_t *width,
263                              bool *pad);
264 static char *make_link_name (char const *name, char const *linkname);
265 static int decode_switches (int argc, char **argv);
266 static bool file_ignored (char const *name);
267 static uintmax_t gobble_file (char const *name, enum filetype type,
268                              ino_t inode, bool command_line_arg,
269                              char const *dirname);
270 static const struct bin_str * get_color_indicator (const struct fileinfo *f,
271                                                    bool symlink_target);
272 static bool print_color_indicator (const struct bin_str *ind);
273 static void put_indicator (const struct bin_str *ind);
274 static void add_ignore_pattern (const char *pattern);

```

```

275 static void attach (char *dest, const char *dirname, const char *name);
276 static void clear_files (void);
277 static void extract_dirs_from_files (char const *dirname,
278                                     bool command_line_arg);
279 static void get_link_name (char const *filename, struct fileinfo *f,
280                           bool command_line_arg);
281 static void indent (size_t from, size_t to);
282 static size_t calculate_columns (bool by_columns);
283 static void print_current_files (void);
284 static void print_dir (char const *name, char const *realname,
285                       bool command_line_arg);
286 static size_t print_file_name_and_frills (const struct fileinfo *f,
287                                           size_t start_col);
288 static void print_horizontal (void);
289 static int format_user_width (uid_t u);
290 static int format_group_width (gid_t g);
291 static void print_long_format (const struct fileinfo *f);
292 static void print_many_per_line (void);
293 static size_t print_name_with_quoting (const struct fileinfo *f,
294                                       bool symlink_target,
295                                       struct obstack *stack,
296                                       size_t start_col);
297 static void prep_non_filename_text (void);
298 static bool print_type_indicator (bool stat_ok, mode_t mode,
299                                  enum filetype type);
300 static void print_with_separator (char sep);
301 static void queue_directory (char const *name, char const *realname,
302                             bool command_line_arg);
303 static void sort_files (void);
304 static void parse_ls_color (void);
305
306 static void getenv_quoting_style (void);
307
308 /* Initial size of hash table.
309 Most hierarchies are likely to be shallower than this. */
310 #define INITIAL_TABLE_SIZE 30
311
312 /* The set of 'active' directories, from the current command-line argument
313 to the level in the hierarchy at which files are being listed.
314 A directory is represented by its device and inode numbers (struct dev_ino).
315 A directory is added to this set when ls begins listing it or its
316 entries, and it is removed from the set just after ls has finished
317 processing it. This set is used solely to detect loops, e.g., with
318 mkdir loop; cd loop; ln -s ../loop sub; ls -RL */
319 static Hash_table *active_dir_set;
320

```

```

321 #define LOOP_DETECT (!active_dir_set)
322
323 /* The table of files in the current directory:
324
325     'cwd_file' points to a vector of 'struct fileinfo', one per file.
326     'cwd_n_alloc' is the number of elements space has been allocated for.
327     'cwd_n_used' is the number actually in use. */
328
329 /* Address of block containing the files that are described. */
330 static struct fileinfo *cwd_file;
331
332 /* Length of block that 'cwd_file' points to, measured in files. */
333 static size_t cwd_n_alloc;
334
335 /* Index of first unused slot in 'cwd_file'. */
336 static size_t cwd_n_used;
337
338 /* Whether files needs may need padding due to quoting. */
339 static bool cwd_some_quoted;
340
341 /* Whether quoting style _may_ add outer quotes,
342    and whether aligning those is useful. */
343 static bool align_variable_outer_quotes;
344
345 /* Vector of pointers to files, in proper sorted order, and the number
346    of entries allocated for it. */
347 static void **sorted_file;
348 static size_t sorted_file_alloc;
349
350 /* When true, in a color listing, color each symlink name according to the
351    type of file it points to. Otherwise, color them according to the 'ln'
352    directive in LS_COLORS. Dangling (orphan) symlinks are treated specially,
353    regardless. This is set when 'ln=target' appears in LS_COLORS. */
354
355 static bool color_symlink_as_referent;
356
357 static char const *hostname;
358
359 /* mode of appropriate file for colorization */
360 #define FILE_OR_LINK_MODE(File) \
361     ((color_symlink_as_referent && (File)->linkok) \
362      ? (File)->linkmode : (File)->stat.st_mode)
363
364
365 /* Record of one pending directory waiting to be listed. */
366

```



```

367 struct pending
368 {
369     char *name;
370     /* If the directory is actually the file pointed to by a symbolic link we
371        were told to list, 'realname' will contain the name of the symbolic
372        link, otherwise zero. */
373     char *realname;
374     bool command_line_arg;
375     struct pending *next;
376 };
377
378 static struct pending *pending_dirs;
379
380 /* Current time in seconds and nanoseconds since 1970, updated as
381    needed when deciding whether a file is recent. */
382
383 static struct timespec current_time;
384
385 static bool print_scontext;
386 static char UNKNOWN_SECURITY_CONTEXT[] = "?";
387
388 /* Whether any of the files has an ACL. This affects the width of the
389    mode column. */
390
391 static bool any_has_acl;
392
393 /* The number of columns to use for columns containing inode numbers,
394    block sizes, link counts, owners, groups, authors, major device
395    numbers, minor device numbers, and file sizes, respectively. */
396
397 static int inode_number_width;
398 static int block_size_width;
399 static int nlink_width;
400 static int sccontext_width;
401 static int owner_width;
402 static int group_width;
403 static int author_width;
404 static int major_device_number_width;
405 static int minor_device_number_width;
406 static int file_size_width;
407
408 /* Option flags */
409
410 /* long_format for lots of info, one per line.
411    one_per_line for just names, one per line.
412    many_per_line for just names, many per line, sorted vertically.

```

```

113 horizontal for just names, many per line, sorted horizontally.
114 with_commas for just names, many per line, separated by commas.
115
116 -l (and other options that imply -l), -1, -C, -x and -m control
117 this parameter. */
118
119 enum format
120 {
121     long_format,           /* -l and other options that imply -l */
122     one_per_line,         /* -1 */
123     many_per_line,        /* -C */
124     horizontal,           /* -x */
125     with_commas           /* -m */
126 };
127
128 static enum format format;
129
130 /* 'full-iso' uses full ISO-style dates and times. 'long-iso' uses longer
131 ISO-style timestamps, though shorter than 'full-iso'. 'iso' uses shorter
132 ISO-style timestamps. 'locale' uses locale-dependent timestamps. */
133 enum time_style
134 {
135     full_iso_time_style,   /* --time-style=full-iso */
136     long_iso_time_style,   /* --time-style=long-iso */
137     iso_time_style,        /* --time-style=iso */
138     locale_time_style      /* --time-style=locale */
139 };
140
141 static char const *const time_style_args[] =
142 {
143     "full-iso", "long-iso", "iso", "locale", NULL
144 };
145 static enum time_style const time_style_types[] =
146 {
147     full_iso_time_style, long_iso_time_style, iso_time_style,
148     locale_time_style
149 };
150 ARGMATCH_VERIFY (time_style_args, time_style_types);
151
152 /* Type of time to print or sort by. Controlled by -c and -u.
153 The values of each item of this enum are important since they are
154 used as indices in the sort functions array (see sort_files()). */
155
156 enum time_type
157 {
158     time_mtime,           /* default */

```

```

459     time_ctime,                /* -c */
460     time_atime,               /* -u */
461     time_numtypes             /* the number of elements of this enum */
462 };
463
464 static enum time_type time_type;
465
466 /* The file characteristic to sort by. Controlled by -t, -S, -U, -X, -v.
467    The values of each item of this enum are important since they are
468    used as indices in the sort functions array (see sort_files()). */
469
470 enum sort_type
471 {
472     sort_none = -1,           /* -U */
473     sort_name,                /* default */
474     sort_extension,           /* -X */
475     sort_size,                /* -S */
476     sort_version,             /* -v */
477     sort_time,                /* -t */
478     sort_numtypes             /* the number of elements of this enum */
479 };
480
481 static enum sort_type sort_type;
482
483 /* Direction of sort.
484    false means highest first if numeric,
485    lowest first if alphabetic;
486    these are the defaults.
487    true means the opposite order in each case.  -r */
488
489 static bool sort_reverse;
490
491 /* True means to display owner information.  -g turns this off. */
492
493 static bool print_owner = true;
494
495 /* True means to display author information. */
496
497 static bool print_author;
498
499 /* True means to display group information.  -G and -o turn this off. */
500
501 static bool print_group = true;
502
503 /* True means print the user and group id's as numbers rather
504    than as names.  -n */

```

```

506 static bool numeric_ids;
507
508 /* True means mention the size in blocks of each file. -s */
509
510 static bool print_block_size;
511
512 /* Human-readable options for output, when printing block counts. */
513 static int human_output_opts;
514
515 /* The units to use when printing block counts. */
516 static uintmax_t output_block_size;
517
518 /* Likewise, but for file sizes. */
519 static int file_human_output_opts;
520 static uintmax_t file_output_block_size = 1;
521
522 /* Follow the output with a special string. Using this format,
523 Emacs' dired mode starts up twice as fast, and can handle all
524 strange characters in file names. */
525 static bool dired;
526
527 /* 'none' means don't mention the type of files.
528 'slash' means mention directories only, with a '/'. 
529 'file_type' means mention file types.
530 'classify' means mention file types and mark executables.
531
532 Controlled by -F, -p, and --indicator-style. */
533
534 enum indicator_style
535 {
536     none,          /* --indicator-style=none */
537     slash,         /* -p, --indicator-style=slash */
538     file_type,     /* --indicator-style=file-type */
539     classify        /* -F, --indicator-style=classify */
540 };
541
542 static enum indicator_style indicator_style;
543
544 /* Names of indicator styles. */
545 static char const *const indicator_style_args[] =
546 {
547     "none", "slash", "file-type", "classify", NULL
548 };
549 static enum indicator_style const indicator_style_types[] =
550 {

```

```

551 none, slash, file_type, classify
552 };
553 ARGMATCH_VERIFY (indicator_style_args, indicator_style_types);
554
555 /* True means use colors to mark types. Also define the different
556 colors as well as the stuff for the LS_COLORS environment variable.
557 The LS_COLORS variable is now in a termcap-like format. */
558
559 static bool print_with_color;
560
561 static bool print_hyperlink;
562
563 /* Whether we used any colors in the output so far. If so, we will
564 need to restore the default color later. If not, we will need to
565 call prep_non_filename_text before using color for the first time. */
566
567 static bool used_color = false;
568
569 enum when_type
570 {
571     when_never,                /* 0: default or --color=never */
572     when_always,              /* 1: --color=always */
573     when_if_tty                /* 2: --color=tty */
574 };
575
576 enum Dereference_symlink
577 {
578     Deref_UNDEFINED = 1,
579     Deref_NEVER,
580     Deref_COMMAND_LINE_ARGUMENTS,    /* -H */
581     Deref_COMMAND_LINE_SYMLINK_TO_DIR, /* the default, in certain cases */
582     Deref_ALWAYS                      /* -L */
583 };
584
585 enum indicator_no
586 {
587     C_LEFT, C_RIGHT, C_END, C_RESET, C_NORM, C_FILE, C_DIR, C_LINK,
588     C_FIFO, C_SOCKET,
589     C_BLK, C_CHR, C_MISSING, C_ORPHAN, C_EXEC, C_DOOR, C_SETUID, C_SETGID,
590     C_STICKY, C_OTHER_WRITABLE, C_STICKY_OTHER_WRITABLE, C_CAP, C_MULTIHARDLINK,
591     C_CLR_TO_EOL
592 };
593
594 static const char *const indicator_name[] =
595 {
596     "lc", "rc", "ec", "rs", "no", "fi", "di", "ln", "pi", "so",

```

```

597 "bd", "cd", "mi", "or", "ex", "do", "su", "sg", "st",
598 "ow", "tw", "ca", "mh", "cl", NULL
599 };
600
601 struct color_ext_type
602 {
603     struct bin_str ext;           /* The extension we're looking for */
604     struct bin_str seq;          /* The sequence to output when we do */
605     struct color_ext_type *next; /* Next in list */
606 };
607
608 static struct bin_str color_indicator[] =
609 {
610     { LEN_STR_PAIR ("033["),          /* lc: Left of color sequence */
611     { LEN_STR_PAIR ("m"),             /* rc: Right of color sequence */
612     { 0, NULL },                     /* ec: End color (replaces lc+rs+rc) */
613     { LEN_STR_PAIR ("0"),             /* rs: Reset to ordinary colors */
614     { 0, NULL },                     /* no: Normal */
615     { 0, NULL },                     /* fi: File: default */
616     { LEN_STR_PAIR ("01;34"),         /* di: Directory: bright blue */
617     { LEN_STR_PAIR ("01;36"),         /* ln: Symlink: bright cyan */
618     { LEN_STR_PAIR ("33"),            /* pi: Pipe: yellow/brown */
619     { LEN_STR_PAIR ("01;35"),         /* so: Socket: bright magenta */
620     { LEN_STR_PAIR ("01;33"),         /* bd: Block device: bright yellow */
621     { LEN_STR_PAIR ("01;33"),         /* cd: Char device: bright yellow */
622     { 0, NULL },                     /* mi: Missing file: undefined */
623     { 0, NULL },                     /* or: Orphaned symlink: undefined */
624     { LEN_STR_PAIR ("01;32"),         /* ex: Executable: bright green */
625     { LEN_STR_PAIR ("01;35"),         /* do: Door: bright magenta */
626     { LEN_STR_PAIR ("37;41"),         /* su: setuid: white on red */
627     { LEN_STR_PAIR ("30;43"),         /* sg: setgid: black on yellow */
628     { LEN_STR_PAIR ("37;44"),         /* st: sticky: black on blue */
629     { LEN_STR_PAIR ("34;42"),         /* ow: other-writable: blue on green */
630     { LEN_STR_PAIR ("30;42"),         /* tw: ow w/ sticky: black on green */
631     { LEN_STR_PAIR ("30;41"),         /* ca: black on red */
632     { 0, NULL },                     /* mh: disabled by default */
633     { LEN_STR_PAIR ("033[K"),        /* cl: clear to end of line */
634     };
635
636 /* FIXME: comment */
637 static struct color_ext_type *color_ext_list = NULL;
638
639 /* Buffer for color sequences */
640 static char *color_buf;
641
642 /* True means to check for orphaned symbolic link, for displaying

```

```

643     colors. */
644
645 static bool check_symlink_color;
646
647 /* True means mention the inode number of each file. -i */
648
649 static bool print_inode;
650
651 /* What to do with symbolic links. Affected by -d, -F, -H, -l (and
652    other options that imply -l), and -L. */
653
654 static enum Dereference_symlink dereference;
655
656 /* True means when a directory is found, display info on its
657    contents. -R */
658
659 static bool recursive;
660
661 /* True means when an argument is a directory name, display info
662    on it itself. -d */
663
664 static bool immediate_dirs;
665
666 /* True means that directories are grouped before files. */
667
668 static bool directories_first;
669
670 /* Which files to ignore. */
671
672 static enum
673 {
674     /* Ignore files whose names start with '.', and files specified by
675        --hide and --ignore. */
676     IGNORE_DEFAULT,
677
678     /* Ignore '.', '..', and files specified by --ignore. */
679     IGNORE_DOT_AND_DOTDOT,
680
681     /* Ignore only files specified by --ignore. */
682     IGNORE_MINIMAL
683 } ignore_mode;
684
685 /* A linked list of shell-style globbing patterns. If a non-argument
686    file name matches any of these patterns, it is ignored.
687    Controlled by -I. Multiple -I options accumulate.
688    The -B option adds '*~' and '*.~' to this list. */

```

```

690 struct ignore_pattern
691 {
692     const char *pattern;
693     struct ignore_pattern *next;
694 };
695
696 static struct ignore_pattern *ignore_patterns;
697
698 /* Similar to IGNORE_PATTERNS, except that -a or -A causes this
699    variable itself to be ignored. */
700 static struct ignore_pattern *hide_patterns;
701
702 /* True means output nongraphic chars in file names as '?'.
703    (-q, --hide-control-chars)
704    qmark_funny_chars and the quoting style (-Q, --quoting-style=WORD) are
705    independent. The algorithm is: first, obey the quoting style to get a
706    string representing the file name; then, if qmark_funny_chars is set,
707    replace all nonprintable chars in that string with '?'. It's necessary
708    to replace nonprintable chars even in quoted strings, because we don't
709    want to mess up the terminal if control chars get sent to it, and some
710    quoting methods pass through control chars as-is. */
711 static bool qmark_funny_chars;
712
713 /* Quoting options for file and dir name output. */
714
715 static struct quoting_options *filename_quoting_options;
716 static struct quoting_options *dirname_quoting_options;
717
718 /* The number of chars per hardware tab stop. Setting this to zero
719    inhibits the use of TAB characters for separating columns. -T */
720 static size_t tabsize;
721
722 /* True means print each directory name before listing it. */
723
724 static bool print_dir_name;
725
726 /* The line length to use for breaking lines in many-per-line format.
727    Can be set with -w. */
728
729 static size_t line_length;
730
731 /* The local time zone rules, as per the TZ environment variable. */
732
733 static timezone_t localtime;
734

```



```

735  /* If true, the file listing format requires that stat be called on
736  each file. */
737
738  static bool format_needs_stat;
739
740  /* Similar to 'format_needs_stat', but set if only the file type is
741  needed. */
742
743  static bool format_needs_type;
744
745  /* An arbitrary limit on the number of bytes in a printed timestamp.
746  This is set to a relatively small value to avoid the need to worry
747  about denial-of-service attacks on servers that run "ls" on behalf
748  of remote clients. 1000 bytes should be enough for any practical
749  timestamp format. */
750
751  enum { TIME_STAMP_LEN_MAXIMUM = MAX (1000, INT_STRLEN_BOUND (time_t)) };
752
753  /* strftime formats for non-recent and recent files, respectively, in
754  -l output. */
755
756  static char const *long_time_format[2] =
757  {
758      /* strftime format for non-recent files (older than 6 months), in
759      -l output. This should contain the year, month and day (at
760      least), in an order that is understood by people in your
761      locale's territory. Please try to keep the number of used
762      screen columns small, because many people work in windows with
763      only 80 columns. But make this as wide as the other string
764      below, for recent files. */
765      /* TRANSLATORS: ls output needs to be aligned for ease of reading,
766      so be wary of using variable width fields from the locale.
767      Note %b is handled specially by ls and aligned correctly.
768      Note also that specifying a width as in %5b is erroneous as strftime
769      will count bytes rather than characters in multibyte locales. */
770      N_("%b %e %Y"),
771      /* strftime format for recent files (younger than 6 months), in -l
772      output. This should contain the month, day and time (at
773      least), in an order that is understood by people in your
774      locale's territory. Please try to keep the number of used
775      screen columns small, because many people work in windows with
776      only 80 columns. But make this as wide as the other string
777      above, for non-recent files. */
778      /* TRANSLATORS: ls output needs to be aligned for ease of reading,
779      so be wary of using variable width fields from the locale.
780      Note %b is handled specially by ls and aligned correctly.

```

```

781     Note also that specifying a width as in %5b is erroneous as strftime
782     will count bytes rather than characters in multibyte locales. */
783     N_("%b %e %H:%M")
784 };
785
786 /* The set of signals that are caught. */
787
788 static sigset_t caught_signals;
789
790 /* If nonzero, the value of the pending fatal signal. */
791
792 static sig_atomic_t volatile interrupt_signal;
793
794 /* A count of the number of pending stop signals that have been received. */
795
796 static sig_atomic_t volatile stop_signal_count;
797
798 /* Desired exit status. */
799
800 static int exit_status;
801
802 /* Exit statuses. */
803 enum
804 {
805     /* "ls" had a minor problem. E.g., while processing a directory,
806     ls obtained the name of an entry via readdir, yet was later
807     unable to stat that name. This happens when listing a directory
808     in which entries are actively being removed or renamed. */
809     LS_MINOR_PROBLEM = 1,
810
811     /* "ls" had more serious trouble (e.g., memory exhausted, invalid
812     option or failure to stat a command line argument. */
813     LS_FAILURE = 2
814 };
815
816 /* For long options that have no equivalent short option, use a
817 non-character as a pseudo short option, starting with CHAR_MAX + 1. */
818 enum
819 {
820     AUTHOR_OPTION = CHAR_MAX + 1,
821     BLOCK_SIZE_OPTION,
822     COLOR_OPTION,
823     DEREFERENCE_COMMAND_LINE_SYMLINK_TO_DIR_OPTION,
824     FILE_TYPE_INDICATOR_OPTION,
825     FORMAT_OPTION,
826     FULL_TIME_OPTION,

```

```

827 GROUP_DIRECTORIES_FIRST_OPTION,
828 HIDE_OPTION,
829 HYPERLINK_OPTION,
830 INDICATOR_STYLE_OPTION,
831 QUOTING_STYLE_OPTION,
832 SHOW_CONTROL_CHARS_OPTION,
833 SI_OPTION,
834 SORT_OPTION,
835 TIME_OPTION,
836 TIME_STYLE_OPTION
837 };
838
839 static struct option const long_options[] =
840 {
841     {"all", no_argument, NULL, 'a'},
842     {"escape", no_argument, NULL, 'b'},
843     {"directory", no_argument, NULL, 'd'},
844     {"dired", no_argument, NULL, 'D'},
845     {"full-time", no_argument, NULL, FULL_TIME_OPTION},
846     {"group-directories-first", no_argument, NULL,
847      GROUP_DIRECTORIES_FIRST_OPTION},
848     {"human-readable", no_argument, NULL, 'h'},
849     {"inode", no_argument, NULL, 'i'},
850     {"kibibytes", no_argument, NULL, 'k'},
851     {"numeric-uid-gid", no_argument, NULL, 'n'},
852     {"no-group", no_argument, NULL, 'G'},
853     {"hide-control-chars", no_argument, NULL, 'q'},
854     {"reverse", no_argument, NULL, 'r'},
855     {"size", no_argument, NULL, 's'},
856     {"width", required_argument, NULL, 'w'},
857     {"almost-all", no_argument, NULL, 'A'},
858     {"ignore-backups", no_argument, NULL, 'B'},
859     {"classify", no_argument, NULL, 'F'},
860     {"file-type", no_argument, NULL, FILE_TYPE_INDICATOR_OPTION},
861     {"si", no_argument, NULL, SI_OPTION},
862     {"dereference-command-line", no_argument, NULL, 'H'},
863     {"dereference-command-line-symlink-to-dir", no_argument, NULL,
864      DEREFERENCE_COMMAND_LINE_SYMLINK_TO_DIR_OPTION},
865     {"hide", required_argument, NULL, HIDE_OPTION},
866     {"ignore", required_argument, NULL, 'I'},
867     {"indicator-style", required_argument, NULL, INDICATOR_STYLE_OPTION},
868     {"dereference", no_argument, NULL, 'L'},
869     {"literal", no_argument, NULL, 'N'},
870     {"quote-name", no_argument, NULL, 'Q'},
871     {"quoting-style", required_argument, NULL, QUOTING_STYLE_OPTION},
872     {"recursive", no_argument, NULL, 'R'},

```

```

873 {"format", required_argument, NULL, FORMAT_OPTION},
874 {"show-control-chars", no_argument, NULL, SHOW_CONTROL_CHARS_OPTION},
875 {"sort", required_argument, NULL, SORT_OPTION},
876 {"tabsize", required_argument, NULL, 'T'},
877 {"time", required_argument, NULL, TIME_OPTION},
878 {"time-style", required_argument, NULL, TIME_STYLE_OPTION},
879 {"color", optional_argument, NULL, COLOR_OPTION},
880 {"hyperlink", optional_argument, NULL, HYPERLINK_OPTION},
881 {"block-size", required_argument, NULL, BLOCK_SIZE_OPTION},
882 {"context", no_argument, 0, 'Z'},
883 {"author", no_argument, NULL, AUTHOR_OPTION},
884 {GETOPT_HELP_OPTION_DECL},
885 {GETOPT_VERSION_OPTION_DECL},
886 {NULL, 0, NULL, 0}
887 };
888
889 static char const *const format_args[] =
890 {
891     "verbose", "long", "commas", "horizontal", "across",
892     "vertical", "single-column", NULL
893 };
894 static enum format const format_types[] =
895 {
896     long_format, long_format, with_commas, horizontal, horizontal,
897     many_per_line, one_per_line
898 };
899 ARGMATCH_VERIFY (format_args, format_types);
900
901 static char const *const sort_args[] =
902 {
903     "none", "time", "size", "extension", "version", NULL
904 };
905 static enum sort_type const sort_types[] =
906 {
907     sort_none, sort_time, sort_size, sort_extension, sort_version
908 };
909 ARGMATCH_VERIFY (sort_args, sort_types);
910
911 static char const *const time_args[] =
912 {
913     "atime", "access", "use", "ctime", "status", NULL
914 };
915 static enum time_type const time_types[] =
916 {
917     time_atime, time_atime, time_atime, time_ctime, time_ctime
918 };

```

```

919 ARGMATCH_VERIFY (time_args, time_types);
920
921 static char const *const when_args[] =
922 {
923     /* force and none are for compatibility with another color-ls version */
924     "always", "yes", "force",
925     "never", "no", "none",
926     "auto", "tty", "if-tty", NULL
927 };
928 static enum when_type const when_types[] =
929 {
930     when_always, when_always, when_always,
931     when_never, when_never, when_never,
932     when_if_tty, when_if_tty, when_if_tty
933 };
934 ARGMATCH_VERIFY (when_args, when_types);
935
936 /* Information about filling a column. */
937 struct column_info
938 {
939     bool valid_len;
940     size_t line_len;
941     size_t *col_arr;
942 };
943
944 /* Array with information about column filledness. */
945 static struct column_info *column_info;
946
947 /* Maximum number of columns ever possible for this display. */
948 static size_t max_idx;
949
950 /* The minimum width of a column is 3: 1 character for the name and 2
951    for the separating white space. */
952 #define MIN_COLUMN_WIDTH      3
953
954
955 /* This zero-based index is used solely with the --dired option.
956    When that option is in effect, this counter is incremented for each
957    byte of output generated by this program so that the beginning
958    and ending indices (in that output) of every file name can be recorded
959    and later output themselves. */
960 static size_t dired_pos;
961
962 #define DIRED_PUTCHAR(c) do {putchar ((c)); ++dired_pos;} while (0)
963
964 /* Write S to STREAM and increment DIRED_POS by S_LEN. */

```

```

965 #define DIRED_FPUTS(s, stream, s_len) \
966     do {fputs (s, stream); dired_pos += s_len;} while (0)
967
968 /* Like DIRED_FPUTS, but for use when S is a literal string. */
969 #define DIRED_FPUTS_LITERAL(s, stream) \
970     do {fputs (s, stream); dired_pos += sizeof (s) - 1;} while (0)
971
972 #define DIRED_INDENT()
973     do
974     {
975         if (dired)
976             DIRED_FPUTS_LITERAL (" ", stdout);
977     }
978     while (0)
979
980 /* With --dired, store pairs of beginning and ending indices of file names. */
981 static struct obstack dired_obstack;
982
983 /* With --dired, store pairs of beginning and ending indices of any
984    directory names that appear as headers (just before 'total' line)
985    for lists of directory entries. Such directory names are seen when
986    listing hierarchies using -R and when a directory is listed with at
987    least one other command line argument. */
988 static struct obstack subdired_obstack;
989
990 /* Save the current index on the specified obstack, OBS. */
991 #define PUSH_CURRENT_DIREN_POS(obs)
992     do
993     {
994         if (dired)
995             obstack_grow (obs, &dired_pos, sizeof (dired_pos));
996     }
997     while (0)
998
999 /* With -R, this stack is used to help detect directory cycles.
1000    The device/inode pairs on this stack mirror the pairs in the
1001    active_dir_set hash table. */
1002 static struct obstack dev_ino_obstack;
1003
1004 /* Push a pair onto the device/inode stack. */
1005 static void
1006 dev_ino_push (dev_t dev, ino_t ino)
1007 {
1008     void *vdi;
1009     struct dev_ino *di;
1010     int dev_ino_size = sizeof *di;

```

```

1011     obstack_blank (&dev_ino_obstack, dev_ino_size);
1012     vdi = obstack_next_free (&dev_ino_obstack);
1013     di = vdi;
1014     di--;
1015     di->st_dev = dev;
1016     di->st_ino = ino;
1017 }
1018
1019 /* Pop a dev/ino struct off the global dev_ino_obstack
1020    and return that struct. */
1021 static struct dev_ino
1022 dev_ino_pop (void)
1023 {
1024     void *vdi;
1025     struct dev_ino *di;
1026     int dev_ino_size = sizeof *di;
1027     assert (dev_ino_size <= obstack_object_size (&dev_ino_obstack));
1028     obstack_blank_fast (&dev_ino_obstack, -dev_ino_size);
1029     vdi = obstack_next_free (&dev_ino_obstack);
1030     di = vdi;
1031     return *di;
1032 }
1033
1034 /* Note the use commented out below:
1035 #define ASSERT_MATCHING_DEV_INO(Name, Di)          \
1036     do                                             \
1037     {                                             \
1038         struct stat sb;                          \
1039         assert (Name);                          \
1040         assert (0 <= stat (Name, &sb));          \
1041         assert (sb.st_dev == Di.st_dev);         \
1042         assert (sb.st_ino == Di.st_ino);         \
1043     }                                             \
1044     while (0)
1045 */
1046
1047 /* Write to standard output PREFIX, followed by the quoting style and
1048    a space-separated list of the integers stored in OS all on one line. */
1049
1050 static void
1051 dired_dump_obstack (const char *prefix, struct obstack *os)
1052 {
1053     size_t n_pos;
1054
1055     n_pos = obstack_object_size (os) / sizeof (dired_pos);
1056     if (n_pos > 0)

```

```

1057 {
1058     size_t *pos = (size_t *) obstack_finish (os);
1059     fputs (prefix, stdout);
1060     for (size_t i = 0; i < n_pos; i++)
1061         printf ("%lu", (unsigned long int) pos[i]);
1062     putchar ('\n');
1063 }
1064 }
1065
1066 /* Return the address of the first plain %b spec in FMT, or NULL if
1067 there is no such spec. %5b etc. do not match, so that user
1068 widths/flags are honored. */
1069
1070 static char const * _GL_ATTRIBUTE_PURE
1071 first_percent_b (char const *fmt)
1072 {
1073     for (; *fmt; fmt++)
1074         if (fmt[0] == '%')
1075             switch (fmt[1])
1076             {
1077                 case 'b': return fmt;
1078                 case '%': fmt++; break;
1079             }
1080     return NULL;
1081 }
1082
1083 static char RFC3986[256];
1084 static void
1085 file_escape_init (void)
1086 {
1087     for (int i = 0; i < 256; i++)
1088         RFC3986[i] |= c_isalnum (i) || i == '~' || i == '-' || i == '.' || i == '_';
1089 }
1090
1091 /* Read the abbreviated month names from the locale, to align them
1092 and to determine the max width of the field and to truncate names
1093 greater than our max allowed.
1094 Note even though this handles multibyte locales correctly
1095 it's not restricted to them as single byte locales can have
1096 variable width abbreviated months and also precomputing/caching
1097 the names was seen to increase the performance of ls significantly. */
1098
1099 /* max number of display cells to use.
1100 As of 2018 the abmon for Arabic has entries with width 12.
1101 It doesn't make much sense to support wider than this
1102 and locales should aim for abmon entries of width <= 5. */

```



```

1103 enum { MAX_MON_WIDTH = 12 };
1104 /* abformat[RECENT][MON] is the format to use for timestamps with
1105    recentness RECENT and month MON. */
1106 enum { ABFORMAT_SIZE = 128 };
1107 static char abformat[2][12][ABFORMAT_SIZE];
1108 /* True if precomputed formats should be used. This can be false if
1109    nl_langinfo fails, if a format or month abbreviation is unusually
1110    long, or if a month abbreviation contains '%'. */
1111 static bool use_abformat;
1112
1113 /* Store into ABMON the abbreviated month names, suitably aligned.
1114    Return true if successful. */
1115
1116 static bool
1117 abmon_init (char abmon[12][ABFORMAT_SIZE])
1118 {
1119     #ifndef HAVE_NL_LANGINFO
1120         return false;
1121     #else
1122         size_t required_mon_width = MAX_MON_WIDTH;
1123         size_t curr_max_width;
1124         do
1125         {
1126             curr_max_width = required_mon_width;
1127             required_mon_width = 0;
1128             for (int i = 0; i < 12; i++)
1129             {
1130                 size_t width = curr_max_width;
1131                 char const *abbr = nl_langinfo (ABMON_1 + i);
1132                 if (strchr (abbr, '%'))
1133                     return false;
1134                 size_t req = mbsalign (abbr, abmon[i], ABFORMAT_SIZE,
1135                                         &width, MBS_ALIGN_LEFT, 0);
1136                 if (! (req < ABFORMAT_SIZE))
1137                     return false;
1138                 required_mon_width = MAX (required_mon_width, width);
1139             }
1140         }
1141         while (curr_max_width > required_mon_width);
1142
1143         return true;
1144     #endif
1145 }
1146
1147 /* Initialize ABFORMAT and USE_ABFORMAT. */
1148

```

```

1150 static void
1151 abformat_init (void)
1152 {
1153     char const *pb[2];
1154     for (int recent = 0; recent < 2; recent++)
1155         pb[recent] = first_percent_b (long_time_format[recent]);
1156     if (! (pb[0] || pb[1]))
1157         return;
1158     char abmon[12][ABFORMAT_SIZE];
1159     if (! abmon_init (abmon))
1160         return;
1161
1162     for (int recent = 0; recent < 2; recent++)
1163     {
1164         char const *fmt = long_time_format[recent];
1165         for (int i = 0; i < 12; i++)
1166         {
1167             char *nfmt = abformat[recent][i];
1168             int nbytes;
1169
1170             if (! pb[recent])
1171                 nbytes = snprintf (nfmt, ABFORMAT_SIZE, "%s", fmt);
1172             else
1173             {
1174                 if (! (pb[recent] - fmt <= MIN (ABFORMAT_SIZE, INT_MAX)))
1175                     return;
1176                 int prefix_len = pb[recent] - fmt;
1177                 nbytes = snprintf (nfmt, ABFORMAT_SIZE, "%.*s%s%s",
1178                                     prefix_len, fmt, abmon[i], pb[recent] + 2);
1179             }
1180
1181             if (! (0 <= nbytes && nbytes < ABFORMAT_SIZE))
1182                 return;
1183         }
1184     }
1185
1186     use_abformat = true;
1187 }
1188
1189 static size_t
1190 dev_ino_hash (void const *x, size_t table_size)
1191 {
1192     struct dev_ino const *p = x;
1193     return (uintmax_t) p->st_ino % table_size;
1194 }

```

```

1196 static bool
1197 dev_ino_compare (void const *x, void const *y)
1198 {
1199     struct dev_ino const *a = x;
1200     struct dev_ino const *b = y;
1201     return SAME_INODE (*a, *b) ? true : false;
1202 }
1203
1204 static void
1205 dev_ino_free (void *x)
1206 {
1207     free (x);
1208 }
1209
1210 /* Add the device/inode pair (P->st_dev/P->st_ino) to the set of
1211 active directories. Return true if there is already a matching
1212 entry in the table. */
1213
1214 static bool
1215 visit_dir (dev_t dev, ino_t ino)
1216 {
1217     struct dev_ino *ent;
1218     struct dev_ino *ent_from_table;
1219     bool found_match;
1220
1221     ent = xmalloc (sizeof *ent);
1222     ent->st_ino = ino;
1223     ent->st_dev = dev;
1224
1225     /* Attempt to insert this entry into the table. */
1226     ent_from_table = hash_insert (active_dir_set, ent);
1227
1228     if (ent_from_table == NULL)
1229     {
1230         /* Insertion failed due to lack of memory. */
1231         xalloc_die ();
1232     }
1233
1234     found_match = (ent_from_table != ent);
1235
1236     if (found_match)
1237     {
1238         /* ent was not inserted, so free it. */
1239         free (ent);
1240     }

```

```

1242     return found_match;
1243 }
1244
1245 static void
1246 free_pending_ent (struct pending *p)
1247 {
1248     free (p->name);
1249     free (p->realname);
1250     free (p);
1251 }
1252
1253 static bool
1254 is_colored (enum indicator_no type)
1255 {
1256     size_t len = color_indicator[type].len;
1257     char const *s = color_indicator[type].string;
1258     return ! (len == 0
1259              || (len == 1 && STRNCMP_LIT (s, "0") == 0)
1260              || (len == 2 && STRNCMP_LIT (s, "00") == 0));
1261 }
1262
1263 static void
1264 restore_default_color (void)
1265 {
1266     put_indicator (&color_indicator[C_LEFT]);
1267     put_indicator (&color_indicator[C_RIGHT]);
1268 }
1269
1270 static void
1271 set_normal_color (void)
1272 {
1273     if (print_with_color && is_colored (C_NORM))
1274     {
1275         put_indicator (&color_indicator[C_LEFT]);
1276         put_indicator (&color_indicator[C_NORM]);
1277         put_indicator (&color_indicator[C_RIGHT]);
1278     }
1279 }
1280
1281 /* An ordinary signal was received; arrange for the program to exit. */
1282
1283 static void
1284 sighandler (int sig)
1285 {
1286     if (! SA_NOCLDSTOP)

```

```

1287     signal (sig, SIG_IGN);
1288     if (! interrupt_signal)
1289         interrupt_signal = sig;
1290 }
1291
1292 /* A SIGTSTP was received; arrange for the program to suspend itself. */
1293
1294 static void
1295 stophandler (int sig)
1296 {
1297     if (! SA_NOCLDSTOP)
1298         signal (sig, stophandler);
1299     if (! interrupt_signal)
1300         stop_signal_count++;
1301 }
1302
1303 /* Process any pending signals. If signals are caught, this function
1304    should be called periodically. Ideally there should never be an
1305    unbounded amount of time when signals are not being processed.
1306    Signal handling can restore the default colors, so callers must
1307    immediately change colors after invoking this function. */
1308
1309 static void
1310 process_signals (void)
1311 {
1312     while (interrupt_signal || stop_signal_count)
1313     {
1314         int sig;
1315         int stops;
1316         sigset_t oldset;
1317
1318         if (used_color)
1319             restore_default_color ();
1320         fflush (stdout);
1321
1322         sigprocmask (SIG_BLOCK, &caught_signals, &oldset);
1323
1324         /* Reload interrupt_signal and stop_signal_count, in case a new
1325            signal was handled before sigprocmask took effect. */
1326         sig = interrupt_signal;
1327         stops = stop_signal_count;
1328
1329         /* SIGTSTP is special, since the application can receive that signal
1330            more than once. In this case, don't set the signal handler to the
1331            default. Instead, just raise the uncatchable SIGSTOP. */
1332         if (stops)

```

```
1333     {
1334         stop_signal_count = stops - 1;
1335         sig = SIGSTOP;
1336     }
1337 else
1338     signal (sig, SIG_DFL);
1339
1340     /* Exit or suspend the program. */
1341     raise (sig);
1342     sigprocmask (SIG_SETMASK, &oldset, NULL);
1343
1344     /* If execution reaches here, then the program has been
1345        continued (after being suspended). */
1346 }
1347 }
1348
1349 /* Setup signal handlers if INIT is true,
1350    otherwise restore to the default. */
1351
1352 static void
1353 signal_setup (bool init)
1354 {
1355     /* The signals that are trapped, and the number of such signals. */
1356     static int const sig[] =
1357     {
1358         /* This one is handled specially. */
1359         SIGTSTP,
1360
1361         /* The usual suspects. */
1362         SIGALRM, SIGHUP, SIGINT, SIGPIPE, SIGQUIT, SIGTERM,
1363 #ifdef SIGPOLL
1364         SIGPOLL,
1365 #endif
1366 #ifdef SIGPROF
1367         SIGPROF,
1368 #endif
1369 #ifdef SIGVTALRM
1370         SIGVTALRM,
1371 #endif
1372 #ifdef SIGXCPU
1373         SIGXCPU,
1374 #endif
1375 #ifdef SIGXFSZ
1376         SIGXFSZ,
1377 #endif
1378     };
```

```

1379     enum { nsigs = ARRAY_CARDINALITY (sig) };
1380
1381     #if ! SA_NOCLDSTOP
1382         static bool caught_sig[nsigs];
1383     #endif
1384
1385     int j;
1386
1387     if (init)
1388     {
1389         #if SA_NOCLDSTOP
1390             struct sigaction act;
1391
1392             sigemptyset (&caught_signals);
1393             for (j = 0; j < nsigs; j++)
1394             {
1395                 sigaction (sig[j], NULL, &act);
1396                 if (act.sa_handler != SIG_IGN)
1397                     sigaddset (&caught_signals, sig[j]);
1398             }
1399
1400             act.sa_mask = caught_signals;
1401             act.sa_flags = SA_RESTART;
1402
1403             for (j = 0; j < nsigs; j++)
1404                 if (sigismember (&caught_signals, sig[j]))
1405                 {
1406                     act.sa_handler = sig[j] == SIGTSTP ? stophandler : sighandler;
1407                     sigaction (sig[j], &act, NULL);
1408                 }
1409             #else
1410             for (j = 0; j < nsigs; j++)
1411             {
1412                 caught_sig[j] = (signal (sig[j], SIG_IGN) != SIG_IGN);
1413                 if (caught_sig[j])
1414                 {
1415                     signal (sig[j], sig[j] == SIGTSTP ? stophandler : sighandler);
1416                     siginterrupt (sig[j], 0);
1417                 }
1418             }
1419             #endif
1420         }
1421         else /* restore. */
1422         {
1423             #if SA_NOCLDSTOP
1424                 for (j = 0; j < nsigs; j++)

```

```
1425         if (sigismember (&caught_signals, sig[j]))
1426             signal (sig[j], SIG_DFL);
1427     #else
1428         for (j = 0; j < nsigs; j++)
1429             if (caught_sig[j])
1430                 signal (sig[j], SIG_DFL);
1431     #endif
1432     }
1433 }
1434
1435 static inline void
1436 signal_init (void)
1437 {
1438     signal_setup (true);
1439 }
1440
1441 static inline void
1442 signal_restore (void)
1443 {
1444     signal_setup (false);
1445 }
1446
1447 int
1448 main (int argc, char **argv)
1449 {
1450     int i;
1451     struct pending *thispend;
1452     int n_files;
1453
1454     initialize_main (&argc, &argv);
1455     set_program_name (argv[0]);
1456     setlocale (LC_ALL, "");
1457     bindtextdomain (PACKAGE, LOCALEDIR);
1458     textdomain (PACKAGE);
1459
1460     initialize_exit_failure (LS_FAILURE);
1461     atexit (close_stdout);
1462
1463     assert (ARRAY_CARDINALITY (color_indicator) + 1
1464             == ARRAY_CARDINALITY (indicator_name));
1465
1466     exit_status = EXIT_SUCCESS;
1467     print_dir_name = true;
1468     pending_dirs = NULL;
1469
1470     current_time.tv_sec = TYPE_MINIMUM (time_t);
```



```

1471 current_time_tv_nsec = -1;
1472
1473 i = decode_switches (argc, argv);
1474
1475 if (print_with_color)
1476     parse_ls_color ();
1477
1478 /* Test print_with_color again, because the call to parse_ls_color
1479    may have just reset it -- e.g., if LS_COLORS is invalid. */
1480 if (print_with_color)
1481 {
1482     /* Avoid following symbolic links when possible. */
1483     if (is_colored (C_ORPHAN)
1484         || (is_colored (C_EXEC) && color_symlink_as_referent)
1485         || (is_colored (C_MISSING) && format == long_format))
1486         check_symlink_color = true;
1487 }
1488
1489 if (dereference == DEREf_UNDEFINED)
1490     dereference = ((immediate_dirs
1491                    || indicator_style == classify
1492                    || format == long_format)
1493                   ? DEREf_NEVER
1494                   : DEREf_COMMAND_LINE_SYMLINK_TO_DIR);
1495
1496 /* When using -R, initialize a data structure we'll use to
1497    detect any directory cycles. */
1498 if (recursive)
1499 {
1500     active_dir_set = hash_initialize (INITIAL_TABLE_SIZE, NULL,
1501                                     dev_ino_hash,
1502                                     dev_ino_compare,
1503                                     dev_ino_free);
1504
1505     if (active_dir_set == NULL)
1506         xalloc_die ();
1507
1508     obstack_init (&dev_ino_obstack);
1509 }
1510
1511 localtime = tzalloc (getenv ("TZ"));
1512
1513 format_needs_stat = sort_type == sort_time || sort_type == sort_size
1514     || format == long_format
1515     || print_scontext
1516     || print_block_size;
1517 format_needs_type = (! format_needs_stat

```

```

1517         && (recursive
1518             || print_with_color
1519             || indicator_style != none
1520             || directories_first));
1521
1522 if (dired)
1523 {
1524     obstack_init (&dired_obstack);
1525     obstack_init (&subdired_obstack);
1526 }
1527
1528 if (print_hyperlink)
1529 {
1530     file_escape_init ();
1531
1532     hostname = xgethostname ();
1533     /* The hostname is generally ignored,
1534        so ignore failures obtaining it. */
1535     if (! hostname)
1536         hostname = "";
1537 }
1538
1539 cwd_n_alloc = 100;
1540 cwd_file = xmalloc (cwd_n_alloc, sizeof *cwd_file);
1541 cwd_n_used = 0;
1542
1543 clear_files ();
1544
1545 n_files = argc - i;
1546
1547 if (n_files <= 0)
1548 {
1549     if (immediate_dirs)
1550         gobble_file (".", directory, NOT_AN_INODE_NUMBER, true, "");
1551     else
1552         queue_directory (".", NULL, true);
1553 }
1554 else
1555 do
1556     gobble_file (argv[i++], unknown, NOT_AN_INODE_NUMBER, true, "");
1557 while (i < argc);
1558
1559 if (cwd_n_used)
1560 {
1561     sort_files ();
1562     if (!immediate_dirs)

```

```

1563     extract_dirs_from_files (NULL, true);
1564     /* 'cwd_n_used' might be zero now. */
1565 }
1566
1567 /* In the following if/else blocks, it is sufficient to test 'pending_dirs'
1568    (and not pending_dirs->name) because there may be no markers in the queue
1569    at this point. A marker may be enqueued when extract_dirs_from_files is
1570    called with a non-empty string or via print_dir. */
1571 if (cwd_n_used)
1572 {
1573     print_current_files ();
1574     if (pending_dirs)
1575         DIRED_PUTCHAR ('\n');
1576 }
1577 else if (n_files <= 1 && pending_dirs && pending_dirs->next == 0)
1578     print_dir_name = false;
1579
1580 while (pending_dirs)
1581 {
1582     thispend = pending_dirs;
1583     pending_dirs = pending_dirs->next;
1584
1585     if (LOOP_DETECT)
1586     {
1587         if (thispend->name == NULL)
1588         {
1589             /* thispend->name == NULL means this is a marker entry
1590                indicating we've finished processing the directory.
1591                Use its dev/ino numbers to remove the corresponding
1592                entry from the active_dir_set hash table. */
1593             struct dev_ino di = dev_ino_pop ();
1594             struct dev_ino *found = hash_delete (active_dir_set, &di);
1595             /* ASSERT_MATCHING_DEV_INO (thispend->realname, di); */
1596             assert (found);
1597             dev_ino_free (found);
1598             free_pending_ent (thispend);
1599             continue;
1600         }
1601     }
1602
1603     print_dir (thispend->name, thispend->realname,
1604               thispend->command_line_arg);
1605
1606     free_pending_ent (thispend);
1607     print_dir_name = true;
1608 }

```

```

1610 if (print_with_color && used_color)
1611 {
1612     int j;
1613
1614     /* Skip the restore when it would be a no-op, i.e.,
1615     when left is "\033[" and right is "m".  */
1616     if (!(color_indicator[C_LEFT].len == 2
1617         && memcmp (color_indicator[C_LEFT].string, "\033[", 2) == 0
1618         && color_indicator[C_RIGHT].len == 1
1619         && color_indicator[C_RIGHT].string[0] == 'm'))
1620         restore_default_color ();
1621
1622     fflush (stdout);
1623
1624     signal_restore ();
1625
1626     /* Act on any signals that arrived before the default was restored.
1627     This can process signals out of order, but there doesn't seem to
1628     be an easy way to do them in order, and the order isn't that
1629     important anyway.  */
1630     for (j = stop_signal_count; j; j--)
1631         raise (SIGSTOP);
1632     j = interrupt_signal;
1633     if (j)
1634         raise (j);
1635 }
1636
1637 if (dired)
1638 {
1639     /* No need to free these since we're about to exit.  */
1640     dired_dump_obstack ("//DIRED//", &dired_obstack);
1641     dired_dump_obstack ("//SUBDIRED//", &subdired_obstack);
1642     printf ("//DIRED-OPTIONS// --quoting-style=%s\n",
1643         quoting_style_args[get_quoting_style (filename_quoting_options)]);
1644 }
1645
1646 if (LOOP_DETECT)
1647 {
1648     assert (hash_get_n_entries (active_dir_set) == 0);
1649     hash_free (active_dir_set);
1650 }
1651
1652 return exit_status;
1653 }
1654

```

```

1655  /* Set the limit on the value given by SPEC. Return true if
1656  successful. 0 means no limit on line length. */
1657
1658  static bool
1659  set_line_length (char const *spec)
1660  {
1661      uintmax_t val;
1662
1663      /* Treat too-large values as if they were SIZE_MAX, which is
1664      effectively infinity. */
1665      switch (xstrtoumax (spec, NULL, 0, &val, ""))
1666      {
1667          case LONGINT_OK:
1668              line_length = MIN (val, SIZE_MAX);
1669              return true;
1670
1671          case LONGINT_OVERFLOW:
1672              line_length = SIZE_MAX;
1673              return true;
1674
1675          default:
1676              return false;
1677      }
1678  }
1679
1680  /* Set all the option flags according to the switches specified.
1681  Return the index of the first non-option argument. */
1682
1683  static int
1684  decode_switches (int argc, char **argv)
1685  {
1686      char *time_style_option = NULL;
1687
1688      bool sort_type_specified = false;
1689      bool kibibytes_specified = false;
1690
1691      qmark_funny_chars = false;
1692
1693      /* initialize all switches to default settings */
1694
1695      switch (ls_mode)
1696      {
1697          case LS_MULTI_COL:
1698              /* This is for the 'dir' program. */
1699              format = many_per_line;
1700              set_quoting_style (NULL, escape_quoting_style);

```

```

1701     break;
1702
1703 case LS_LONG_FORMAT:
1704     /* This is for the 'vdir' program. */
1705     format = long_format;
1706     set_quoting_style (NULL, escape_quoting_style);
1707     break;
1708
1709 case LS_LS:
1710     /* This is for the 'ls' program. */
1711     if (isatty (STDOUT_FILENO))
1712     {
1713         format = many_per_line;
1714         set_quoting_style (NULL, shell_escape_quoting_style);
1715         /* See description of qmark_funny_chars, above. */
1716         qmark_funny_chars = true;
1717     }
1718     else
1719     {
1720         format = one_per_line;
1721         qmark_funny_chars = false;
1722     }
1723     break;
1724
1725 default:
1726     abort ();
1727 }
1728
1729 time_type = time_mtime;
1730 sort_type = sort_name;
1731 sort_reverse = false;
1732 numeric_ids = false;
1733 print_block_size = false;
1734 indicator_style = none;
1735 print_inode = false;
1736 dereference = DEREf_UNDEFINED;
1737 recursive = false;
1738 immediate_dirs = false;
1739 ignore_mode = IGNORE_DEFAULT;
1740 ignore_patterns = NULL;
1741 hide_patterns = NULL;
1742 print_scontext = false;
1743
1744 getenv_quoting_style ();
1745
1746 line_length = 80;

```

```

1747 {
1748     char const *p = getenv ("COLUMNS");
1749     if (p && *p && ! set_line_length (p))
1750         error (0, 0,
1751             _("ignoring invalid width in environment variable COLUMNS: %s"),
1752             quote (p));
1753 }
1754
1755 #ifdef TIOCGWINSZ
1756 {
1757     struct winsize ws;
1758
1759     if (ioctl (STDOUT_FILENO, TIOCGWINSZ, &ws) != -1
1760         && 0 < ws.ws_col && ws.ws_col == (size_t) ws.ws_col)
1761         line_length = ws.ws_col;
1762 }
1763 #endif
1764
1765 {
1766     char const *p = getenv ("TABSIZ");
1767     tabsize = 8;
1768     if (p)
1769     {
1770         unsigned long int tmp_ulong;
1771         if (xstrtoul (p, NULL, 0, &tmp_ulong, NULL) == LONGINT_OK
1772             && tmp_ulong <= SIZE_MAX)
1773         {
1774             tabsize = tmp_ulong;
1775         }
1776         else
1777         {
1778             error (0, 0,
1779                 _("ignoring invalid tab size in environment variable TABSIZE: %s"),
1780                 quote (p));
1781         }
1782     }
1783 }
1784
1785 while (true)
1786 {
1787     int oi = -1;
1788     int c = getopt_long (argc, argv,
1789                         "abcdghiklmnopqrstuvw:ABCDEFGHILNQIRST:UXZ1",
1790                         long_options, &oi);
1791     if (c == -1)
1792         break;

```

```

1794 switch (c)
1795 {
1796     case 'a':
1797         ignore_mode = IGNORE_MINIMAL;
1798         break;
1799
1800     case 'b':
1801         set_quoting_style (NULL, escape_quoting_style);
1802         break;
1803
1804     case 'c':
1805         time_type = time_ctime;
1806         break;
1807
1808     case 'd':
1809         immediate_dirs = true;
1810         break;
1811
1812     case 'f':
1813         /* Same as enabling -a -U and disabling -l -s. */
1814         ignore_mode = IGNORE_MINIMAL;
1815         sort_type = sort_none;
1816         sort_type_specified = true;
1817         /* disable -l */
1818         if (format == long_format)
1819             format = (isatty (STDOUT_FILENO) ? many_per_line : one_per_line);
1820         print_block_size = false;        /* disable -s */
1821         print_with_color = false;        /* disable --color */
1822         print_hyperlink = false;        /* disable --hyperlink */
1823         break;
1824
1825     case FILE_TYPE_INDICATOR_OPTION: /* --file-type */
1826         indicator_style = file_type;
1827         break;
1828
1829     case 'g':
1830         format = long_format;
1831         print_owner = false;
1832         break;
1833
1834     case 'h':
1835         file_human_output_opts = human_output_opts =
1836             human_autoscale | human_SI | human_base_1024;
1837         file_output_block_size = output_block_size = 1;
1838         break;

```



```
1840     case 'i':
1841         print_inode = true;
1842         break;
1843
1844     case 'k':
1845         kibibytes_specified = true;
1846         break;
1847
1848     case 'l':
1849         format = long_format;
1850         break;
1851
1852     case 'm':
1853         format = with_commas;
1854         break;
1855
1856     case 'n':
1857         numeric_ids = true;
1858         format = long_format;
1859         break;
1860
1861     case 'o':  /* Just like -l, but don't display group info. */
1862         format = long_format;
1863         print_group = false;
1864         break;
1865
1866     case 'p':
1867         indicator_style = slash;
1868         break;
1869
1870     case 'q':
1871         qmark_funny_chars = true;
1872         break;
1873
1874     case 'r':
1875         sort_reverse = true;
1876         break;
1877
1878     case 's':
1879         print_block_size = true;
1880         break;
1881
1882     case 't':
1883         sort_type = sort_time;
1884         sort_type_specified = true;
```

```
1885         break;
1886
1887     case 'u':
1888         time_type = time_atime;
1889         break;
1890
1891     case 'v':
1892         sort_type = sort_version;
1893         sort_type_specified = true;
1894         break;
1895
1896     case 'w':
1897         if (! set_line_length (optarg))
1898             die (LS_FAILURE, 0, "%s: %s", _("invalid line width"),
1899                 quote (optarg));
1900         break;
1901
1902     case 'x':
1903         format = horizontal;
1904         break;
1905
1906     case 'A':
1907         ignore_mode = IGNORE_DOT_AND_DOTDOT;
1908         break;
1909
1910     case 'B':
1911         add_ignore_pattern ("*~");
1912         add_ignore_pattern ("*.~");
1913         break;
1914
1915     case 'C':
1916         format = many_per_line;
1917         break;
1918
1919     case 'D':
1920         dired = true;
1921         break;
1922
1923     case 'F':
1924         indicator_style = classify;
1925         break;
1926
1927     case 'G':                                     /* inhibit display of group info */
1928         print_group = false;
1929         break;
1930
```

```
1931     case 'H':
1932         dereference = Deref_Command_Line_Arguments;
1933         break;
1934
1935     case DEREference_Command_Line_Symlink_To_Dir_Option:
1936         dereference = Deref_Command_Line_Symlink_To_Dir;
1937         break;
1938
1939     case 'I':
1940         add_ignore_pattern (optarg);
1941         break;
1942
1943     case 'L':
1944         dereference = Deref_Always;
1945         break;
1946
1947     case 'N':
1948         set_quoting_style (NULL, literal_quoting_style);
1949         break;
1950
1951     case 'Q':
1952         set_quoting_style (NULL, c_quoting_style);
1953         break;
1954
1955     case 'R':
1956         recursive = true;
1957         break;
1958
1959     case 'S':
1960         sort_type = sort_size;
1961         sort_type_specified = true;
1962         break;
1963
1964     case 'T':
1965         tabsize = xnumtoulmax (optarg, 0, 0, SIZE_MAX, "",
1966                                _("invalid tab size"), LS_FAILURE);
1967         break;
1968
1969     case 'U':
1970         sort_type = sort_none;
1971         sort_type_specified = true;
1972         break;
1973
1974     case 'X':
1975         sort_type = sort_extension;
1976         sort_type_specified = true;
```

```
1977     break;
1978
1979 case '1':
1980     /* -1 has no effect after -l. */
1981     if (format != long_format)
1982         format = one_per_line;
1983     break;
1984
1985 case AUTHOR_OPTION:
1986     print_author = true;
1987     break;
1988
1989 case HIDE_OPTION:
1990     {
1991         struct ignore_pattern *hide = xmalloc (sizeof *hide);
1992         hide->pattern = optarg;
1993         hide->next = hide_patterns;
1994         hide_patterns = hide;
1995     }
1996     break;
1997
1998 case SORT_OPTION:
1999     sort_type = XARGMATCH ("--sort", optarg, sort_args, sort_types);
2000     sort_type_specified = true;
2001     break;
2002
2003 case GROUP_DIRECTORIES_FIRST_OPTION:
2004     directories_first = true;
2005     break;
2006
2007 case TIME_OPTION:
2008     time_type = XARGMATCH ("--time", optarg, time_args, time_types);
2009     break;
2010
2011 case FORMAT_OPTION:
2012     format = XARGMATCH ("--format", optarg, format_args, format_types);
2013     break;
2014
2015 case FULL_TIME_OPTION:
2016     format = long_format;
2017     time_style_option = bad_cast ("full-iso");
2018     break;
2019
2020 case COLOR_OPTION:
2021     {
2022         int i;
```

```

2021     if (optarg)
2022         i = XARGMATCH ("--color", optarg, when_args, when_types);
2023     else
2024         /* Using --color with no argument is equivalent to using
2025            --color=always. */
2026         i = when_always;
2027
2028     print_with_color = (i == when_always
2029                        || (i == when_if_tty
2030                            && isatty (STDOUT_FILENO)));
2031
2032     if (print_with_color)
2033     {
2034         /* Don't use TAB characters in output. Some terminal
2035            emulators can't handle the combination of tabs and
2036            color codes on the same line. */
2037         tabsize = 0;
2038     }
2039     break;
2040 }
2041
2042 case HYPERLINK_OPTION:
2043 {
2044     int i;
2045     if (optarg)
2046         i = XARGMATCH ("--hyperlink", optarg, when_args, when_types);
2047     else
2048         /* Using --hyperlink with no argument is equivalent to using
2049            --hyperlink=always. */
2050         i = when_always;
2051
2052     print_hyperlink = (i == when_always
2053                       || (i == when_if_tty
2054                           && isatty (STDOUT_FILENO)));
2055
2056     break;
2057 }
2058
2059 case INDICATOR_STYLE_OPTION:
2060     indicator_style = XARGMATCH ("--indicator-style", optarg,
2061                                  indicator_style_args,
2062                                  indicator_style_types);
2063
2064     break;
2065
2066 case QUOTING_STYLE_OPTION:
2067     set_quoting_style (NULL,
2068                        XARGMATCH ("--quoting-style", optarg,

```

```

2069             quoting_style_vals));
2070
2071         break;
2072
2073     case TIME_STYLE_OPTION:
2074         time_style_option = optarg;
2075         break;
2076
2077     case SHOW_CONTROL_CHARS_OPTION:
2078         qmark_funny_chars = false;
2079         break;
2080
2081     case BLOCK_SIZE_OPTION:
2082     {
2083         enum strtol_error e = human_options (optarg, &human_output_opts,
2084                                             &output_block_size);
2085
2086         if (e != LONGINT_OK)
2087             xstrtol_fatal (e, oi, 0, long_options, optarg);
2088         file_human_output_opts = human_output_opts;
2089         file_output_block_size = output_block_size;
2090     }
2091     break;
2092
2093     case SI_OPTION:
2094         file_human_output_opts = human_output_opts =
2095             human_autoscale | human_SI;
2096         file_output_block_size = output_block_size = 1;
2097         break;
2098
2099     case 'Z':
2100         print_scontext = true;
2101         break;
2102
2103     case_GETOPT_HELP_CHAR;
2104
2105     case_GETOPT_VERSION_CHAR (PROGRAM_NAME, AUTHORS);
2106
2107     default:
2108         usage (LS_FAILURE);
2109     }
2110 }
2111
2112 if (! output_block_size)
2113 {
2114     char const *ls_block_size = getenv ("LS_BLOCK_SIZE");
2115     human_options (ls_block_size,

```

```

2115         &human_output_opts, &output_block_size);
2116     if (ls_block_size || getenv ("BLOCK_SIZE"))
2117     {
2118         file_human_output_opts = human_output_opts;
2119         file_output_block_size = output_block_size;
2120     }
2121     if (kibibytes_specified)
2122     {
2123         human_output_opts = 0;
2124         output_block_size = 1024;
2125     }
2126 }
2127
2128 /* Determine the max possible number of display columns. */
2129 max_idx = line_length / MIN_COLUMN_WIDTH;
2130 /* Account for first display column not having a separator,
2131 or line_lengths shorter than MIN_COLUMN_WIDTH. */
2132 max_idx += line_length % MIN_COLUMN_WIDTH != 0;
2133
2134 enum quoting_style qs = get_quoting_style (NULL);
2135 align_variable_outer_quotes = format != with_commas
2136                               && format != one_per_line
2137                               && (line_length || format == long_format)
2138                               && (qs == shell_quoting_style
2139                                   || qs == shell_escape_quoting_style
2140                                   || qs == c_maybe_quoting_style);
2141 filename_quoting_options = clone_quoting_options (NULL);
2142 if (qs == escape_quoting_style)
2143     set_char_quoting (filename_quoting_options, ' ', 1);
2144 if (file_type <= indicator_style)
2145 {
2146     char const *p;
2147     for (p = &"*=>@|"[indicator_style - file_type]; *p; p++)
2148         set_char_quoting (filename_quoting_options, *p, 1);
2149 }
2150
2151 dirname_quoting_options = clone_quoting_options (NULL);
2152 set_char_quoting (dirname_quoting_options, ':', 1);
2153
2154 /* --dired is meaningful only with --format=long (-l).
2155 Otherwise, ignore it.  FIXME: warn about this?
2156 Alternatively, make --dired imply --format=long? */
2157 if (dired && (format != long_format || print_hyperlink))
2158     dired = false;
2159
2160 /* If -c or -u is specified and not -l (or any other option that implies -l),

```

and no sort-type was specified, then sort by the ctime (-c) or atime (-u).
The behavior of ls when using either -c or -u but with neither -l nor -t
appears to be unspecified by POSIX. So, with GNU ls, '-u' alone means
sort by atime (this is the one that's not specified by the POSIX spec),
-lu means show atime and sort by name, -lut means show atime and sort
by atime. */

```
if ((time_type == time_ctime || time_type == time_atime)
    && !sort_type_specified && format != long_format)
{
    sort_type = sort_time;
}

if (format == long_format)
{
    char *style = time_style_option;
    static char const posix_prefix[] = "posix-";

    if (!style)
        if (! (style = getenv ("TIME_STYLE")))
            style = bad_cast ("locale");

    while (STREQ_LEN (style, posix_prefix, sizeof posix_prefix - 1))
    {
        if (! hard_locale (LC_TIME))
            return optind;
        style += sizeof posix_prefix - 1;
    }

    if (*style == '+')
    {
        char *p0 = style + 1;
        char *p1 = strchr (p0, '\n');
        if (! p1)
            p1 = p0;
        else
        {
            if (strchr (p1 + 1, '\n'))
                die (LS_FAILURE, 0, _("invalid time style format %s"),
                    quote (p0));
            *p1++ = '\0';
        }
        long_time_format[0] = p0;
        long_time_format[1] = p1;
    }
    else
```



```

2207 {
2208     ptrdiff_t res = argmatch (style, time_style_args,
2209                               (char const *) time_style_types,
2210                               sizeof (*time_style_types));
2211     if (res < 0)
2212     {
2213         /* This whole block used to be a simple use of XARGMATCH.
2214         but that didn't print the "posix"-prefixed variants or
2215         the "+"-prefixed format string option upon failure. */
2216         argmatch_invalid ("time style", style, res);
2217
2218         /* The following is a manual expansion of argmatch_valid,
2219         but with the added "+ ..." description and the [posix-]
2220         prefixes prepended. Note that this simplification works
2221         only because all four existing time_style_types values
2222         are distinct. */
2223         fputs (_("Valid arguments are:\n"), stderr);
2224         char const *const *p = time_style_args;
2225         while (*p)
2226             fprintf (stderr, " - [posix-]%s\n", *p++);
2227         fputs (_(" - +FORMAT (e.g., +%H:%M) for a 'date'-style"
2228                 " format\n"), stderr);
2229         usage (LS_FAILURE);
2230     }
2231     switch (res)
2232     {
2233     case full_iso_time_style:
2234         long_time_format[0] = long_time_format[1] =
2235             "%Y-%m-%d %H:%M:%S.%N %Z";
2236         break;
2237
2238     case long_iso_time_style:
2239         long_time_format[0] = long_time_format[1] = "%Y-%m-%d %H:%M";
2240         break;
2241
2242     case iso_time_style:
2243         long_time_format[0] = "%Y-%m-%d ";
2244         long_time_format[1] = "%m-%d %H:%M";
2245         break;
2246
2247     case locale_time_style:
2248         if (hard_locale (LC_TIME))
2249         {
2250             for (int i = 0; i < 2; i++)
2251                 long_time_format[i] =
2252                     dcgettext (NULL, long_time_format[i], LC_TIME);

```

```

2253     }
2254 }
2255 }
2256
2257     abformat_init ();
2258 }
2259
2260     return optind;
2261 }
2262
2263 /* Parse a string as part of the LS_COLORS variable; this may involve
2264 decoding all kinds of escape characters. If equals_end is set an
2265 unescaped equal sign ends the string, otherwise only a : or \0
2266 does. Set *OUTPUT_COUNT to the number of bytes output. Return
2267 true if successful.
2268
2269 The resulting string is *not* null-terminated, but may contain
2270 embedded nulls.
2271
2272 Note that both dest and src are char **; on return they point to
2273 the first free byte after the array and the character that ended
2274 the input string, respectively. */
2275
2276 static bool
2277 get_funky_string (char **dest, const char **src, bool equals_end,
2278                  size_t *output_count)
2279 {
2280     char num;                                /* For numerical codes */
2281     size_t count;                            /* Something to count with */
2282     enum {
2283         ST_GND, ST_BACKSLASH, ST_OCTAL, ST_HEX, ST_CARET, ST_END, ST_ERROR
2284     } state;
2285     const char *p;
2286     char *q;
2287
2288     p = *src;                                /* We don't want to double-indirect */
2289     q = *dest;                                /* the whole darn time. */
2290
2291     count = 0;                                /* No characters counted in yet. */
2292     num = 0;
2293
2294     state = ST_GND;                            /* Start in ground state. */
2295     while (state < ST_END)
2296     {
2297         switch (state)
2298         {

```

```

2299     case ST_GND:                                /* Ground state (no escapes) */
2300         switch (*p)
2301         {
2302             case ':':
2303             case '\0':
2304                 state = ST_END;                    /* End of string */
2305                 break;
2306             case '\\':
2307                 state = ST_BACKSLASH; /* Backslash escape sequence */
2308                 ++p;
2309                 break;
2310             case '^':
2311                 state = ST_CARET; /* Caret escape */
2312                 ++p;
2313                 break;
2314             case '=':
2315                 if (equals_end)
2316                 {
2317                     state = ST_END; /* End */
2318                     break;
2319                 }
2320                 FALLTHROUGH;
2321             default:
2322                 *(q++) = *(p++);
2323                 ++count;
2324                 break;
2325         }
2326         break;
2327
2328     case ST_BACKSLASH:                            /* Backslash escaped character */
2329         switch (*p)
2330         {
2331             case '0':
2332             case '1':
2333             case '2':
2334             case '3':
2335             case '4':
2336             case '5':
2337             case '6':
2338             case '7':
2339                 state = ST_OCTAL;                    /* Octal sequence */
2340                 num = *p - '0';
2341                 break;
2342             case 'x':
2343             case 'X':
2344                 state = ST_HEX;                      /* Hex sequence */

```

```
2345     num = 0;
2346     break;
2347 case 'a':                               /* Bell */
2348     num = '\a';
2349     break;
2350 case 'b':                               /* Backspace */
2351     num = '\b';
2352     break;
2353 case 'e':                               /* Escape */
2354     num = 27;
2355     break;
2356 case 'f':                               /* Form feed */
2357     num = '\f';
2358     break;
2359 case 'n':                               /* Newline */
2360     num = '\n';
2361     break;
2362 case 'r':                               /* Carriage return */
2363     num = '\r';
2364     break;
2365 case 't':                               /* Tab */
2366     num = '\t';
2367     break;
2368 case 'v':                               /* Vtab */
2369     num = '\v';
2370     break;
2371 case '?':                               /* Delete */
2372     num = 127;
2373     break;
2374 case '_':                               /* Space */
2375     num = ' ';
2376     break;
2377 case '\0':                             /* End of string */
2378     state = ST_ERROR;                  /* Error! */
2379     break;
2380 default:                               /* Escaped character like \ ^ : = */
2381     num = *p;
2382     break;
2383 }
2384 if (state == ST_BACKSLASH)
2385 {
2386     *(q++) = num;
2387     ++count;
2388     state = ST_GND;
2389 }
2390 ++p;
```

```

2391     break;
2392
2393 case ST_OCTAL:                                     /* Octal sequence */
2394     if (*p < '0' || *p > '7')
2395     {
2396         *(q++) = num;
2397         ++count;
2398         state = ST_GND;
2399     }
2400     else
2401         num = (num << 3) + (*(p++) - '0');
2402     break;
2403
2404 case ST_HEX:                                       /* Hex sequence */
2405     switch (*p)
2406     {
2407         case '0':
2408         case '1':
2409         case '2':
2410         case '3':
2411         case '4':
2412         case '5':
2413         case '6':
2414         case '7':
2415         case '8':
2416         case '9':
2417             num = (num << 4) + (*(p++) - '0');
2418             break;
2419         case 'a':
2420         case 'b':
2421         case 'c':
2422         case 'd':
2423         case 'e':
2424         case 'f':
2425             num = (num << 4) + (*(p++) - 'a') + 10;
2426             break;
2427         case 'A':
2428         case 'B':
2429         case 'C':
2430         case 'D':
2431         case 'E':
2432         case 'F':
2433             num = (num << 4) + (*(p++) - 'A') + 10;
2434             break;
2435         default:
2436             *(q++) = num;

```

```

2437         ++count;
2438         state = ST_GND;
2439         break;
2440     }
2441     break;
2442
2443     case ST_CARET:                                /* Caret escape */
2444         state = ST_GND;                            /* Should be the next state... */
2445         if (*p >= '@' && *p <= '~')
2446             {
2447                 *(q++) = *(p++) & 037;
2448                 ++count;
2449             }
2450         else if (*p == '?')
2451             {
2452                 *(q++) = 127;
2453                 ++count;
2454             }
2455         else
2456             state = ST_ERROR;
2457         break;
2458
2459     default:
2460         abort ();
2461     }
2462 }
2463
2464 *dest = q;
2465 *src = p;
2466 *output_count = count;
2467
2468 return state != ST_ERROR;
2469 }
2470
2471 enum parse_state
2472 {
2473     PS_START = 1,
2474     PS_2,
2475     PS_3,
2476     PS_4,
2477     PS_DONE,
2478     PS_FAIL
2479 };
2480
2481
2482 /* Check if the content of TERM is a valid name in dircolors. */

```

```

2484 static bool
2485 known_term_type (void)
2486 {
2487     char const *term = getenv ("TERM");
2488     if (! term || ! *term)
2489         return false;
2490
2491     char const *line = G_line;
2492     while (line - G_line < sizeof (G_line))
2493     {
2494         if (STRNCMP_LIT (line, "TERM ") == 0)
2495         {
2496             if (fnmatch (line + 5, term, 0) == 0)
2497                 return true;
2498         }
2499         line += strlen (line) + 1;
2500     }
2501
2502     return false;
2503 }
2504
2505 static void
2506 parse_ls_color (void)
2507 {
2508     const char *p;                /* Pointer to character being parsed */
2509     char *buf;                    /* color_buf buffer pointer */
2510     int ind_no;                   /* Indicator number */
2511     char label[3];                /* Indicator label */
2512     struct color_ext_type *ext;   /* Extension we are working on */
2513
2514     if ((p = getenv ("LS_COLORS")) == NULL || *p == '\0')
2515     {
2516         /* LS_COLORS takes precedence, but if that's not set then
2517            honor the COLORTERM and TERM env variables so that
2518            we only go with the internal ANSI color codes if the
2519            former is non empty or the latter is set to a known value. */
2520         char const *colorterm = getenv ("COLORTERM");
2521         if (! (colorterm && *colorterm) && ! known_term_type ())
2522             print_with_color = false;
2523         return;
2524     }
2525
2526     ext = NULL;
2527     strcpy (label, "??");
2528

```

```

2529  /* This is an overly conservative estimate, but any possible
2530  LS_COLORS string will *not* generate a color_buf longer than
2531  itself, so it is a safe way of allocating a buffer in
2532  advance. */
2533  buf = color_buf = xstrdup (p);
2534
2535  enum parse_state state = PS_START;
2536  while (true)
2537  {
2538      switch (state)
2539      {
2540          case PS_START: /* First label character */
2541              switch (*p)
2542              {
2543                  case ':':
2544                      ++p;
2545                      break;
2546
2547                  case '*':
2548                      /* Allocate new extension block and add to head of
2549                      linked list (this way a later definition will
2550                      override an earlier one, which can be useful for
2551                      having terminal-specific defs override global). */
2552
2553                      ext = xmalloc (sizeof *ext);
2554                      ext->next = color_ext_list;
2555                      color_ext_list = ext;
2556
2557                      ++p;
2558                      ext->ext.string = buf;
2559
2560                      state = (get_funky_string (&buf, &p, true, &ext->ext.len)
2561                          ? PS_4 : PS_FAIL);
2562                      break;
2563
2564                  case '\\0':
2565                      state = PS_DONE; /* Done! */
2566                      goto done;
2567
2568                  default: /* Assume it is file type label */
2569                      label[0] = *(p++);
2570                      state = PS_2;
2571                      break;
2572              }
2573          break;
2574

```



```

2575     case PS_2:                                /* Second label character */
2576         if (*p)
2577             {
2578                 label[1] = *(p++);
2579                 state = PS_3;
2580             }
2581         else
2582             state = PS_FAIL;                    /* Error */
2583         break;
2584
2585     case PS_3:                                /* Equal sign after indicator label */
2586         state = PS_FAIL;                        /* Assume failure... */
2587         if (*(p++) == '=') /* It *should* be... */
2588             {
2589                 for (ind_no = 0; indicator_name[ind_no] != NULL; ++ind_no)
2590                     {
2591                         if (STREQ (label, indicator_name[ind_no]))
2592                             {
2593                                 color_indicator[ind_no].string = buf;
2594                                 state = (get_funky_string (&buf, &p, false,
2595                                                             &color_indicator[ind_no].len)
2596                                         ? PS_START : PS_FAIL);
2597                                 break;
2598                             }
2599                     }
2600                 if (state == PS_FAIL)
2601                     error (0, 0, _("unrecognized prefix: %s"), quote (label));
2602             }
2603         break;
2604
2605     case PS_4:                                /* Equal sign after *.ext */
2606         if (*(p++) == '=')
2607             {
2608                 ext->seq.string = buf;
2609                 state = (get_funky_string (&buf, &p, false, &ext->seq.len)
2610                         ? PS_START : PS_FAIL);
2611             }
2612         else
2613             state = PS_FAIL;
2614         break;
2615
2616     case PS_FAIL:
2617         goto done;
2618
2619     default:
2620         abort ();

```

```

2621     }
2622 }
2623 done:
2624
2625 if (state == PS_FAIL)
2626 {
2627     struct color_ext_type *e;
2628     struct color_ext_type *e2;
2629
2630     error (0, 0,
2631            _("unparsable value for LS_COLORS environment variable"));
2632     free (color_buf);
2633     for (e = color_ext_list; e != NULL; /* empty */)
2634     {
2635         e2 = e;
2636         e = e->next;
2637         free (e2);
2638     }
2639     print_with_color = false;
2640 }
2641
2642 if (color_indicator[C_LINK].len == 6
2643     && !STRNCMP_LIT (color_indicator[C_LINK].string, "target"))
2644     color_symlink_as_referent = true;
2645 }
2646
2647 /* Set the quoting style default if the environment variable
2648 QUOTING_STYLE is set.  */
2649
2650 static void
2651 getenv_quoting_style (void)
2652 {
2653     char const *q_style = getenv ("QUOTING_STYLE");
2654     if (q_style)
2655     {
2656         int i = ARGMATCH (q_style, quoting_style_args, quoting_style_vals);
2657         if (0 <= i)
2658             set_quoting_style (NULL, quoting_style_vals[i]);
2659         else
2660             error (0, 0,
2661                    _("ignoring invalid value of environment variable QUOTING_STYLE: %s"),
2662                    quote (q_style));
2663     }
2664 }
2665
2666 /* Set the exit status to report a failure.  If SERIOUS, it is a

```

```

2667     serious failure; otherwise, it is merely a minor problem. */
2668
2669 static void
2670 set_exit_status (bool serious)
2671 {
2672     if (serious)
2673         exit_status = LS_FAILURE;
2674     else if (exit_status == EXIT_SUCCESS)
2675         exit_status = LS_MINOR_PROBLEM;
2676 }
2677
2678 /* Assuming a failure is serious if SERIOUS, use the printf-style
2679    MESSAGE to report the failure to access a file named FILE. Assume
2680    errno is set appropriately for the failure. */
2681
2682 static void
2683 file_failure (bool serious, char const *message, char const *file)
2684 {
2685     error (0, errno, message, quoteaf (file));
2686     set_exit_status (serious);
2687 }
2688
2689 /* Request that the directory named NAME have its contents listed later.
2690    If REALNAME is nonzero, it will be used instead of NAME when the
2691    directory name is printed. This allows symbolic links to directories
2692    to be treated as regular directories but still be listed under their
2693    real names. NAME == NULL is used to insert a marker entry for the
2694    directory named in REALNAME.
2695    If NAME is non-NULL, we use its dev/ino information to save
2696    a call to stat -- when doing a recursive (-R) traversal.
2697    COMMAND_LINE_ARG means this directory was mentioned on the command line. */
2698
2699 static void
2700 queue_directory (char const *name, char const *realname, bool command_line_arg)
2701 {
2702     struct pending *new = xmalloc (sizeof *new);
2703     new->realname = realname ? xstrdup (realname) : NULL;
2704     new->name = name ? xstrdup (name) : NULL;
2705     new->command_line_arg = command_line_arg;
2706     new->next = pending_dirs;
2707     pending_dirs = new;
2708 }
2709
2710 /* Read directory NAME, and list the files in it.
2711    If REALNAME is nonzero, print its name instead of NAME;
2712    this is used for symbolic links to directories.

```

```

2713     COMMAND_LINE_ARG means this directory was mentioned on the command line. */
2714
2715 static void
2716 print_dir (char const *name, char const *realname, bool command_line_arg)
2717 {
2718     DIR *dirp;
2719     struct dirent *next;
2720     uintmax_t total_blocks = 0;
2721     static bool first = true;
2722
2723     errno = 0;
2724     dirp = opendir (name);
2725     if (!dirp)
2726     {
2727         file_failure (command_line_arg, _("cannot open directory %s"), name);
2728         return;
2729     }
2730
2731     if (LOOP_DETECT)
2732     {
2733         struct stat dir_stat;
2734         int fd = dirfd (dirp);
2735
2736         /* If dirfd failed, endure the overhead of using stat. */
2737         if ((0 <= fd
2738             ? fstat (fd, &dir_stat)
2739             : stat (name, &dir_stat)) < 0)
2740         {
2741             file_failure (command_line_arg,
2742                          _("cannot determine device and inode of %s"), name);
2743             closedir (dirp);
2744             return;
2745         }
2746
2747         /* If we've already visited this dev/inode pair, warn that
2748            we've found a loop, and do not process this directory. */
2749         if (visit_dir (dir_stat.st_dev, dir_stat.st_ino))
2750         {
2751             error (0, 0, ("%s: not listing already-listed directory"),
2752                  quotef (name));
2753             closedir (dirp);
2754             set_exit_status (true);
2755             return;
2756         }
2757
2758         dev_ino_push (dir_stat.st_dev, dir_stat.st_ino);

```

```

2759 }
2760
2761 clear_files ();
2762
2763 if (recursive || print_dir_name)
2764 {
2765     if (!first)
2766         DIRED_PUTCHAR ('\n');
2767     first = false;
2768     DIRED_INDENT ();
2769
2770     char *absolute_name = NULL;
2771     if (print_hyperlink)
2772     {
2773         absolute_name = canonicalize_filename_mode (name, CAN_MISSING);
2774         if (! absolute_name)
2775             file_failure (command_line_arg,
2776                          _("error canonicalizing %s"), name);
2777     }
2778     quote_name (realname ? realname : name, dirname_quoting_options, -1,
2779                NULL, true, &subdired_obstack, absolute_name);
2780
2781     free (absolute_name);
2782
2783     DIRED_FPUTS_LITERAL (":\n", stdout);
2784 }
2785
2786 /* Read the directory entries, and insert the subfiles into the 'cwd_file'
2787 table. */
2788
2789 while (1)
2790 {
2791     /* Set errno to zero so we can distinguish between a readdir failure
2792     and when readdir simply finds that there are no more entries. */
2793     errno = 0;
2794     next = readdir (dirp);
2795     if (next)
2796     {
2797         if (! file_ignored (next->d_name))
2798         {
2799             enum filetype type = unknown;
2800
2801             #if HAVE_STRUCT_DIRENT_D_TYPE
2802             switch (next->d_type)
2803             {
2804                 case DT_BLK: type = blockdev; break;

```

```

2805         case DT_CHR: type = chardev; break;
2806         case DT_DIR: type = directory; break;
2807         case DT_FIFO: type = fifo; break;
2808         case DT_LNK: type = symbolic_link; break;
2809         case DT_REG: type = normal; break;
2810         case DT_SOCK: type = sock; break;
2811 # ifdef DT_WHT
2812         case DT_WHT: type = whiteout; break;
2813 # endif
2814     }
2815 #endif
2816     total_blocks += gobble_file (next->d_name, type,
2817                                RELIABLE_D_INO (next),
2818                                false, name);
2819
2820     /* In this narrow case, print out each name right away, so
2821        ls uses constant memory while processing the entries of
2822        this directory. Useful when there are many (millions)
2823        of entries in a directory. */
2824     if (format == one_per_line && sort_type == sort_none
2825         && !print_block_size && !recursive)
2826     {
2827         /* We must call sort_files in spite of
2828            "sort_type == sort_none" for its initialization
2829            of the sorted_file vector. */
2830         sort_files ();
2831         print_current_files ();
2832         clear_files ();
2833     }
2834 }
2835 }
2836 else if (errno != 0)
2837 {
2838     file_failure (command_line_arg, _("reading directory %s"), name);
2839     if (errno != EOVERFLOW)
2840         break;
2841 }
2842 else
2843     break;
2844
2845 /* When processing a very large directory, and since we've inhibited
2846    interrupts, this loop would take so long that ls would be annoyingly
2847    uninterruptible. This ensures that it handles signals promptly. */
2848 process_signals ();
2849 }
2850

```

```

2851 if (closedir (dirp) != 0)
2852 {
2853     file_failure (command_line_arg, _("closing directory %s"), name);
2854     /* Don't return; print whatever we got. */
2855 }
2856
2857 /* Sort the directory contents. */
2858 sort_files ();
2859
2860 /* If any member files are subdirectories, perhaps they should have their
2861    contents listed rather than being mentioned here as files. */
2862
2863 if (recursive)
2864     extract_dirs_from_files (name, false);
2865
2866 if (format == long_format || print_block_size)
2867 {
2868     const char *p;
2869     char buf[LONGEST_HUMAN_READABLE + 1];
2870
2871     DIRED_INDENT ();
2872     p = _("total");
2873     DIRED_FPUTS (p, stdout, strlen (p));
2874     DIRED_PUTCHAR (' ');
2875     p = human_readable (total_blocks, buf, human_output_opts,
2876                         ST_NBLOCKSIZE, output_block_size);
2877     DIRED_FPUTS (p, stdout, strlen (p));
2878     DIRED_PUTCHAR ('\n');
2879 }
2880
2881 if (cwd_n_used)
2882     print_current_files ();
2883 }
2884
2885 /* Add 'pattern' to the list of patterns for which files that match are
2886    not listed. */
2887
2888 static void
2889 add_ignore_pattern (const char *pattern)
2890 {
2891     struct ignore_pattern *ignore;
2892
2893     ignore = xmalloc (sizeof *ignore);
2894     ignore->pattern = pattern;
2895     /* Add it to the head of the linked list. */
2896     ignore->next = ignore_patterns;

```

```

2897 ignore_patterns= ignore;
2898 }
2899
2900 /* Return true if one of the PATTERNS matches FILE. */
2901
2902 static bool
2903 patterns_match (struct ignore_pattern const *patterns, char const *file)
2904 {
2905     struct ignore_pattern const *p;
2906     for (p = patterns; p; p = p->next)
2907         if (fnmatch (p->pattern, file, FNM_PERIOD) == 0)
2908             return true;
2909     return false;
2910 }
2911
2912 /* Return true if FILE should be ignored. */
2913
2914 static bool
2915 file_ignored (char const *name)
2916 {
2917     return ((ignore_mode != IGNORE_MINIMAL
2918             && name[0] == '.')
2919             && (ignore_mode == IGNORE_DEFAULT || ! name[1 + (name[1] == '.')]))
2920         || (ignore_mode == IGNORE_DEFAULT
2921             && patterns_match (hide_patterns, name))
2922         || patterns_match (ignore_patterns, name));
2923 }
2924
2925 /* POSIX requires that a file size be printed without a sign, even
2926 when negative. Assume the typical case where negative sizes are
2927 actually positive values that have wrapped around. */
2928
2929 static uintmax_t
2930 unsigned_file_size (off_t size)
2931 {
2932     return size + (size < 0) * ((uintmax_t) OFF_T_MAX - OFF_T_MIN + 1);
2933 }
2934
2935 #ifdef HAVE_CAP
2936 /* Return true if NAME has a capability (see linux/capability.h) */
2937 static bool
2938 has_capability (char const *name)
2939 {
2940     char *result;
2941     bool has_cap;

```



```

2943     cap_t cap_d = cap_get_file (name);
2944     if (cap_d == NULL)
2945         return false;
2946
2947     result = cap_to_text (cap_d, NULL);
2948     cap_free (cap_d);
2949     if (!result)
2950         return false;
2951
2952     /* check if human-readable capability string is empty */
2953     has_cap = !!*result;
2954
2955     cap_free (result);
2956     return has_cap;
2957 }
2958 #else
2959 static bool
2960 has_capability (char const *name _GL_UNUSED)
2961 {
2962     errno = ENOTSUP;
2963     return false;
2964 }
2965 #endif
2966
2967 /* Enter and remove entries in the table 'cwd_file'. */
2968
2969 static void
2970 free_ent (struct fileinfo *f)
2971 {
2972     free (f->name);
2973     free (f->linkname);
2974     free (f->absolute_name);
2975     if (f->scontext != UNKNOWN_SECURITY_CONTEXT)
2976     {
2977         if (is_smack_enabled ())
2978             free (f->scontext);
2979         else
2980             freecon (f->scontext);
2981     }
2982 }
2983
2984 /* Empty the table of files. */
2985 static void
2986 clear_files (void)
2987 {
2988     for (size_t i = 0; i < cwd_n_used; i++)

```

```

2989     {
2990         struct fileinfo *f = sorted_file[i];
2991         free_ent (f);
2992     }
2993
2994     cwd_n_used = 0;
2995     cwd_some_quoted = false;
2996     any_has_acl = false;
2997     inode_number_width = 0;
2998     block_size_width = 0;
2999     nlink_width = 0;
3000     owner_width = 0;
3001     group_width = 0;
3002     author_width = 0;
3003     scontext_width = 0;
3004     major_device_number_width = 0;
3005     minor_device_number_width = 0;
3006     file_size_width = 0;
3007 }
3008
3009 /* Return true if ERR implies lack-of-support failure by a
3010    getxattr-calling function like getfilecon or file_has_acl. */
3011 static bool
3012 errno_unsupported (int err)
3013 {
3014     return (err == EINVAL || err == ENOSYS || is_ENOTSUP (err));
3015 }
3016
3017 /* Cache *getfilecon failure, when it's trivial to do so.
3018    Like getfilecon/lgetfilecon, but when F's st_dev says it's doesn't
3019    support getting the security context, fail with ENOTSUP immediately. */
3020 static int
3021 getfilecon_cache (char const *file, struct fileinfo *f, bool deref)
3022 {
3023     /* st_dev of the most recently processed device for which we've
3024        found that [l]getfilecon fails indicating lack of support. */
3025     static dev_t unsupported_device;
3026
3027     if (f->stat.st_dev == unsupported_device)
3028     {
3029         errno = ENOTSUP;
3030         return -1;
3031     }
3032     int r = 0;
3033     #ifdef HAVE_SMACK
3034     if (is_smack_enabled ())

```

```

3035     r = smack_new_label_from_path (file, "security.SMACK64", deref,
3036                                     &f->scontext);
3037     else
3038 #endif
3039         r = (deref
3040              ? getfilecon (file, &f->scontext)
3041                : lgetfilecon (file, &f->scontext));
3042     if (r < 0 && errno_unsupported (errno))
3043         unsupported_device = f->stat.st_dev;
3044     return r;
3045 }
3046
3047 /* Cache file_has_acl failure, when it's trivial to do.
3048    Like file_has_acl, but when F's st_dev says it's on a file
3049    system lacking ACL support, return 0 with ENOTSUP immediately. */
3050 static int
3051 file_has_acl_cache (char const *file, struct fileinfo *f)
3052 {
3053     /* st_dev of the most recently processed device for which we've
3054        found that file_has_acl fails indicating lack of support. */
3055     static dev_t unsupported_device;
3056
3057     if (f->stat.st_dev == unsupported_device)
3058     {
3059         errno = ENOTSUP;
3060         return 0;
3061     }
3062
3063     /* Zero errno so that we can distinguish between two 0-returning cases:
3064        "has-ACL-support, but only a default ACL" and "no ACL support". */
3065     errno = 0;
3066     int n = file_has_acl (file, &f->stat);
3067     if (n <= 0 && errno_unsupported (errno))
3068         unsupported_device = f->stat.st_dev;
3069     return n;
3070 }
3071
3072 /* Cache has_capability failure, when it's trivial to do.
3073    Like has_capability, but when F's st_dev says it's on a file
3074    system lacking capability support, return 0 with ENOTSUP immediately. */
3075 static bool
3076 has_capability_cache (char const *file, struct fileinfo *f)
3077 {
3078     /* st_dev of the most recently processed device for which we've
3079        found that has_capability fails indicating lack of support. */
3080     static dev_t unsupported_device;

```

```

3082     if (f->stat.st_dev == unsupported_device)
3083     {
3084         errno = ENOTSUP;
3085         return 0;
3086     }
3087
3088     bool b = has_capability (file);
3089     if ( !b && errno_unsupported (errno))
3090         unsupported_device = f->stat.st_dev;
3091     return b;
3092 }
3093
3094 static bool
3095 needs_quoting (char const* name)
3096 {
3097     char test[2];
3098     size_t len = quotearg_buffer (test, sizeof test , name, -1,
3099                                   filename_quoting_options);
3100     return *name != *test || strlen (name) != len;
3101 }
3102
3103 /* Add a file to the current table of files.
3104    Verify that the file exists, and print an error message if it does not.
3105    Return the number of blocks that the file occupies. */
3106 static uintmax_t
3107 gobble_file (char const *name, enum filetype type, ino_t inode,
3108              bool command_line_arg, char const *dirname)
3109 {
3110     uintmax_t blocks = 0;
3111     struct fileinfo *f;
3112
3113     /* An inode value prior to gobble_file necessarily came from readdir,
3114        which is not used for command line arguments. */
3115     assert (! command_line_arg || inode == NOT_AN_INODE_NUMBER);
3116
3117     if (cwd_n_used == cwd_n_alloc)
3118     {
3119         cwd_file = xnrealloc (cwd_file, cwd_n_alloc, 2 * sizeof *cwd_file);
3120         cwd_n_alloc *= 2;
3121     }
3122
3123     f = &cwd_file[cwd_n_used];
3124     memset (f, '\0', sizeof *f);
3125     f->stat.st_ino = inode;
3126     f->filetype = type;

```

```

3128 f->quoted = -1;
3129 if ((! cwd_some_quoted) && align_variable_outer_quotes)
3130 {
3131     /* Determine if any quoted for padding purposes. */
3132     f->quoted = needs_quoting (name);
3133     if (f->quoted)
3134         cwd_some_quoted = 1;
3135 }
3136
3137 if (command_line_arg
3138     || print_hyperlink
3139     || format_needs_stat
3140     /* When coloring a directory (we may know the type from
3141     direct.d_type), we have to stat it in order to indicate
3142     sticky and/or other-writable attributes. */
3143     || (type == directory && print_with_color
3144         && (is_colored (C_OTHER_WRITABLE)
3145             || is_colored (C_STICKY)
3146             || is_colored (C_STICKY_OTHER_WRITABLE)))
3147     /* When dereferencing symlinks, the inode and type must come from
3148     stat, but readdir provides the inode and type of lstat. */
3149     || ((print_inode || format_needs_type)
3150         && (type == symbolic_link || type == unknown)
3151         && (dereference == DEREFS_ALWAYS
3152             || color_symlink_as_referent || check_symlink_color))
3153     /* Command line dereferences are already taken care of by the above
3154     assertion that the inode number is not yet known. */
3155     || (print_inode && inode == NOT_AN_INODE_NUMBER)
3156     || (format_needs_type
3157         && (type == unknown || command_line_arg
3158             /* --indicator-style=classify (aka -F)
3159             requires that we stat each regular file
3160             to see if it's executable. */
3161             || (type == normal && (indicator_style == classify
3162                 /* This is so that --color ends up
3163                 highlighting files with these mode
3164                 bits set even when options like -F are
3165                 not specified. Note we do a redundant
3166                 stat in the very unlikely case where
3167                 C_CAP is set but not the others. */
3168                 || (print_with_color
3169                     && (is_colored (C_EXEC)
3170                         || is_colored (C_SETUID)
3171                         || is_colored (C_SETGID)
3172                         || is_colored (C_CAP)))

```

[illegible]

```

3219         if (!need_lstat)
3220             break;
3221
3222         /* stat failed because of ENOENT, maybe indicating a dangling
3223         symlink. Or stat succeeded, FULL_NAME does not refer to a
3224         directory, and --dereference-command-line-symlink-to-dir is
3225         in effect. Fall through so that we call lstat instead. */
3226     }
3227     FALLTHROUGH;
3228
3229     default: /* DEREf_NEVER */
3230         err = lstat (full_name, &f->stat);
3231         do_deref = false;
3232         break;
3233     }
3234
3235     if (err != 0)
3236     {
3237         /* Failure to stat a command line argument leads to
3238         an exit status of 2. For other files, stat failure
3239         provokes an exit status of 1. */
3240         file_failure (command_line_arg,
3241                     _("cannot access %s"), full_name);
3242         if (command_line_arg)
3243             return 0;
3244
3245         f->name = xstrdup (name);
3246         cwd_n_used++;
3247
3248         return 0;
3249     }
3250
3251     f->stat_ok = true;
3252
3253     /* Note has_capability() adds around 30% runtime to 'ls --color' */
3254     if ((type == normal || S_ISREG (f->stat.st_mode))
3255         && print_with_color && is_colored (C_CAP))
3256         f->has_capability = has_capability_cache (full_name, f);
3257
3258     if (format == long_format || print_scontext)
3259     {
3260         bool have_scontext = false;
3261         bool have_acl = false;
3262         int attr_len = getfilecon_cache (full_name, f, do_deref);
3263         err = (attr_len < 0);
3264

```

```

3265     if (err == 0)
3266     {
3267         if (is_smack_enabled ())
3268             have_scontext = ! STREQ ("_", f->scontext);
3269         else
3270             have_scontext = ! STREQ ("unlabeled", f->scontext);
3271     }
3272 else
3273 {
3274     f->scontext = UNKNOWN_SECURITY_CONTEXT;
3275
3276     /* When requesting security context information, don't make
3277     ls fail just because the file (even a command line argument)
3278     isn't on the right type of file system. I.e., a getfilecon
3279     failure isn't in the same class as a stat failure. */
3280     if (is_ENOTSUP (errno) || errno == ENODATA)
3281         err = 0;
3282 }
3283
3284 if (err == 0 && format == long_format)
3285 {
3286     int n = file_has_acl_cache (full_name, f);
3287     err = (n < 0);
3288     have_acl = (0 < n);
3289 }
3290
3291 f->acl_type = (!have_scontext && !have_acl
3292               ? ACL_T_NONE
3293               : (have_scontext && !have_acl
3294                 ? ACL_T_LSM_CONTEXT_ONLY
3295                 : ACL_T_YES));
3296 any_has_acl |= f->acl_type != ACL_T_NONE;
3297
3298 if (err)
3299     error (0, errno, "%s", quotef (full_name));
3300 }
3301
3302 if (S_ISLNK (f->stat.st_mode)
3303     && (format == long_format || check_symlink_color))
3304 {
3305     struct stat linkstats;
3306
3307     get_link_name (full_name, f, command_line_arg);
3308     char *linkname = make_link_name (full_name, f->linkname);
3309
3310     /* Use the slower quoting path for this entry, though

```



```

3311     don't update CWD_SOME_QUOTED since alignment not affected. */
3312     if (linkname && f->quoted == 0 && needs_quoting (f->linkname))
3313         f->quoted = -1;
3314
3315     /* Avoid following symbolic links when possible, ie, when
3316     they won't be traced and when no indicator is needed. */
3317     if (linkname
3318         && (file_type <= indicator_style || check_symlink_color)
3319         && stat (linkname, &linkstats) == 0)
3320     {
3321         f->linkok = true;
3322
3323         /* Symbolic links to directories that are mentioned on the
3324         command line are automatically traced if not being
3325         listed as files. */
3326         if (!command_line_arg || format == long_format
3327             || !S_ISDIR (linkstats.st_mode))
3328         {
3329             /* Get the linked-to file's mode for the filetype indicator
3330             in long listings. */
3331             f->linkmode = linkstats.st_mode;
3332         }
3333     }
3334     free (linkname);
3335 }
3336
3337 if (S_ISLNK (f->stat.st_mode))
3338     f->filetype = symbolic_link;
3339 else if (S_ISDIR (f->stat.st_mode))
3340 {
3341     if (command_line_arg && !immediate_dirs)
3342         f->filetype = arg_directory;
3343     else
3344         f->filetype = directory;
3345 }
3346 else
3347     f->filetype = normal;
3348
3349 blocks = ST_NBLOCKS (f->stat);
3350 if (format == long_format || print_block_size)
3351 {
3352     char buf[LONGEST_HUMAN_READABLE + 1];
3353     int len = mbswidth (human_readable (blocks, buf, human_output_opts,
3354                                         ST_NBLOCKSIZE, output_block_size),
3355                         0);
3356     if (block_size_width < len)

```

```

3357         block_size_width = len;
3358     }
3359
3360     if (format == long_format)
3361     {
3362         if (print_owner)
3363         {
3364             int len = format_user_width (f->stat.st_uid);
3365             if (owner_width < len)
3366                 owner_width = len;
3367         }
3368
3369         if (print_group)
3370         {
3371             int len = format_group_width (f->stat.st_gid);
3372             if (group_width < len)
3373                 group_width = len;
3374         }
3375
3376         if (print_author)
3377         {
3378             int len = format_user_width (f->stat.st_author);
3379             if (author_width < len)
3380                 author_width = len;
3381         }
3382     }
3383
3384     if (print_scontext)
3385     {
3386         int len = strlen (f->scontext);
3387         if (scontext_width < len)
3388             scontext_width = len;
3389     }
3390
3391     if (format == long_format)
3392     {
3393         char b[INT_BUFSIZE_BOUND (uintmax_t)];
3394         int b_len = strlen (umaxtostr (f->stat.st_nlink, b));
3395         if (nlink_width < b_len)
3396             nlink_width = b_len;
3397
3398         if (S_ISCHR (f->stat.st_mode) || S_ISBLK (f->stat.st_mode))
3399         {
3400             char buf[INT_BUFSIZE_BOUND (uintmax_t)];
3401             int len = strlen (umaxtostr (major (f->stat.st_rdev), buf));
3402             if (major_device_number_width < len)

```

```

3403         major_device_number_width = len;
3404         len = strlen (umaxtostr (minor (f->stat.st_rdev), buf));
3405         if (minor_device_number_width < len)
3406             minor_device_number_width = len;
3407         len = major_device_number_width + 2 + minor_device_number_width;
3408         if (file_size_width < len)
3409             file_size_width = len;
3410     }
3411     else
3412     {
3413         char buf[LONGEST_HUMAN_READABLE + 1];
3414         uintmax_t size = unsigned_file_size (f->stat.st_size);
3415         int len = mbswidth (human_readable (size, buf,
3416                                           file_human_output_opts,
3417                                           1, file_output_block_size),
3418                             0);
3419         if (file_size_width < len)
3420             file_size_width = len;
3421     }
3422 }
3423 }
3424
3425 if (print_inode)
3426 {
3427     char buf[INT_BUFSIZE_BOUND (uintmax_t)];
3428     int len = strlen (umaxtostr (f->stat.st_ino, buf));
3429     if (inode_number_width < len)
3430         inode_number_width = len;
3431 }
3432
3433 f->name = xstrdup (name);
3434 cwd_n_used++;
3435
3436 return blocks;
3437 }
3438
3439 /* Return true if F refers to a directory. */
3440 static bool
3441 is_directory (const struct fileinfo *f)
3442 {
3443     return f->filetype == directory || f->filetype == arg_directory;
3444 }
3445
3446 /* Put the name of the file that FILENAME is a symbolic link to
3447    into the LINKNAME field of 'f'.  COMMAND_LINE_ARG indicates whether
3448    FILENAME is a command-line argument. */

```

```

3440 static void
3441 get_link_name (char const *filename, struct fileinfo *f, bool command_line_arg)
3442 {
3443     f->linkname = areadlink_with_size (filename, f->stat.st_size);
3444     if (f->linkname == NULL)
3445         file_failure (command_line_arg, _("cannot read symbolic link %s"),
3446                     filename);
3447 }
3448
3449 /* If LINKNAME is a relative name and NAME contains one or more
3450    leading directories, return LINKNAME with those directories
3451    prepended; otherwise, return a copy of LINKNAME.
3452    If LINKNAME is NULL, return NULL.  */
3453
3454 static char *
3455 make_link_name (char const *name, char const *linkname)
3456 {
3457     if (!linkname)
3458         return NULL;
3459
3460     if (IS_ABSOLUTE_FILE_NAME (linkname))
3461         return xstrdup (linkname);
3462
3463     /* The link is to a relative name.  Prepend any leading directory
3464        in 'name' to the link name.  */
3465     size_t prefix_len = dir_len (name);
3466     if (prefix_len == 0)
3467         return xstrdup (linkname);
3468
3469     char *p = xmalloc (prefix_len + 1 + strlen (linkname) + 1);
3470
3471     /* PREFIX_LEN usually specifies a string not ending in slash.
3472        In that case, extend it by one, since the next byte *is* a slash.
3473        Otherwise, the prefix is "/", so leave the length unchanged.  */
3474     if ( ! ISSLASH (name[prefix_len - 1]))
3475         ++prefix_len;
3476
3477     stpcpy (stpncpy (p, name, prefix_len), linkname);
3478     return p;
3479 }
3480
3481 /* Return true if the last component of NAME is '.' or '..'
3482    This is so we don't try to recurse on '././././... */
3483
3484 static bool

```

```

3495 basename_is_dot_or_dotdot (const char *name)
3496 {
3497     char const *base = last_component (name);
3498     return dot_or_dotdot (base);
3499 }
3500
3501 /* Remove any entries from CWD_FILE that are for directories,
3502    and queue them to be listed as directories instead.
3503    DIRNAME is the prefix to prepend to each dirname
3504    to make it correct relative to ls's working dir;
3505    if it is null, no prefix is needed and "." and ".." should not be ignored.
3506    If COMMAND_LINE_ARG is true, this directory was mentioned at the top level,
3507    This is desirable when processing directories recursively. */
3508
3509 static void
3510 extract_dirs_from_files (char const *dirname, bool command_line_arg)
3511 {
3512     size_t i;
3513     size_t j;
3514     bool ignore_dot_and_dot_dot = (dirname != NULL);
3515
3516     if (dirname && LOOP_DETECT)
3517     {
3518         /* Insert a marker entry first. When we dequeue this marker entry,
3519            we'll know that DIRNAME has been processed and may be removed
3520            from the set of active directories. */
3521         queue_directory (NULL, dirname, false);
3522     }
3523
3524     /* Queue the directories last one first, because queueing reverses the
3525        order. */
3526     for (i = cwd_n_used; i-- != 0; )
3527     {
3528         struct fileinfo *f = sorted_file[i];
3529
3530         if (is_directory (f)
3531             && (! ignore_dot_and_dot_dot
3532                 || ! basename_is_dot_or_dotdot (f->name)))
3533         {
3534             if (!dirname || f->name[0] == '/')
3535                 queue_directory (f->name, f->linkname, command_line_arg);
3536             else
3537             {
3538                 char *name = file_name_concat (dirname, f->name, NULL);
3539                 queue_directory (name, f->linkname, command_line_arg);
3540                 free (name);

```

```

3541     }
3542     if (f->filetype == arg_directory)
3543         free_ent (f);
3544 }
3545 }
3546
3547 /* Now delete the directories from the table, compacting all the remaining
3548 entries. */
3549
3550 for (i = 0, j = 0; i < cwd_n_used; i++)
3551 {
3552     struct fileinfo *f = sorted_file[i];
3553     sorted_file[j] = f;
3554     j += (f->filetype != arg_directory);
3555 }
3556 cwd_n_used = j;
3557 }
3558
3559 /* Use strcoll to compare strings in this locale. If an error occurs,
3560 report an error and longjmp to failed_strcoll. */
3561
3562 static jmp_buf failed_strcoll;
3563
3564 static int
3565 xstrcoll (char const *a, char const *b)
3566 {
3567     int diff;
3568     errno = 0;
3569     diff = strcoll (a, b);
3570     if (errno)
3571     {
3572         error (0, errno, _("cannot compare file names %s and %s"),
3573             quote_n (0, a), quote_n (1, b));
3574         set_exit_status (false);
3575         longjmp (failed_strcoll, 1);
3576     }
3577     return diff;
3578 }
3579
3580 /* Comparison routines for sorting the files. */
3581
3582 typedef void const *V;
3583 typedef int (*qsortFunc)(V a, V b);
3584
3585 /* Used below in DEFINE_SORT_FUNCTIONS for _df_ sort function variants.
3586 The do { ... } while(0) makes it possible to use the macro more like

```

```

3587 a statement, without violating C89 rules: */
3588 #define DIRFIRST_CHECK(a, b)
3589 do
3590 {
3591     bool a_is_dir = is_directory ((struct fileinfo const *) a);
3592     bool b_is_dir = is_directory ((struct fileinfo const *) b);
3593     if (a_is_dir && !b_is_dir)
3594         return -1; /* a goes before b */
3595     if (!a_is_dir && b_is_dir)
3596         return 1; /* b goes before a */
3597 }
3598 while (0)
3599
3600 /* Define the 8 different sort function variants required for each sortkey.
3601    KEY_NAME is a token describing the sort key, e.g., ctime, atime, size.
3602    KEY_CMP_FUNC is a function to compare records based on that key, e.g.,
3603    ctime_cmp, atime_cmp, size_cmp. Append KEY_NAME to the string,
3604    '[rev_][x]str{cmp/coll}[_df]_', to create each function name. */
3605 #define DEFINE_SORT_FUNCTIONS(key_name, key_cmp_func)
3606 /* direct, non-dirfirst versions */
3607 static int xstrcoll_##key_name (V a, V b)
3608 { return key_cmp_func (a, b, xstrcoll); }
3609 static int _GL_ATTRIBUTE_PURE strcmp_##key_name (V a, V b)
3610 { return key_cmp_func (a, b, strcmp); }
3611
3612 /* reverse, non-dirfirst versions */
3613 static int rev_xstrcoll_##key_name (V a, V b)
3614 { return key_cmp_func (b, a, xstrcoll); }
3615 static int _GL_ATTRIBUTE_PURE rev_strcmp_##key_name (V a, V b)
3616 { return key_cmp_func (b, a, strcmp); }
3617
3618 /* direct, dirfirst versions */
3619 static int xstrcoll_df_##key_name (V a, V b)
3620 { DIRFIRST_CHECK (a, b); return key_cmp_func (a, b, xstrcoll); }
3621 static int _GL_ATTRIBUTE_PURE strcmp_df_##key_name (V a, V b)
3622 { DIRFIRST_CHECK (a, b); return key_cmp_func (a, b, strcmp); }
3623
3624 /* reverse, dirfirst versions */
3625 static int rev_xstrcoll_df_##key_name (V a, V b)
3626 { DIRFIRST_CHECK (a, b); return key_cmp_func (b, a, xstrcoll); }
3627 static int _GL_ATTRIBUTE_PURE rev_strcmp_df_##key_name (V a, V b)
3628 { DIRFIRST_CHECK (a, b); return key_cmp_func (b, a, strcmp); }
3629
3630 static inline int
3631 cmp_ctime (struct fileinfo const *a, struct fileinfo const *b,
3632            int (*cmp) (char const *, char const *))

```

```

3633 {
3634     int diff = timespec_cmp (get_stat_ctime (&b->stat),
3635                             get_stat_ctime (&a->stat));
3636     return diff ? diff : cmp (a->name, b->name);
3637 }
3638
3639 static inline int
3640 cmp_mtime (struct fileinfo const *a, struct fileinfo const *b,
3641            int (*cmp) (char const *, char const *))
3642 {
3643     int diff = timespec_cmp (get_stat_mtime (&b->stat),
3644                             get_stat_mtime (&a->stat));
3645     return diff ? diff : cmp (a->name, b->name);
3646 }
3647
3648 static inline int
3649 cmp_atime (struct fileinfo const *a, struct fileinfo const *b,
3650            int (*cmp) (char const *, char const *))
3651 {
3652     int diff = timespec_cmp (get_stat_atime (&b->stat),
3653                             get_stat_atime (&a->stat));
3654     return diff ? diff : cmp (a->name, b->name);
3655 }
3656
3657 static inline int
3658 cmp_size (struct fileinfo const *a, struct fileinfo const *b,
3659           int (*cmp) (char const *, char const *))
3660 {
3661     int diff = longdiff (b->stat.st_size, a->stat.st_size);
3662     return diff ? diff : cmp (a->name, b->name);
3663 }
3664
3665 static inline int
3666 cmp_name (struct fileinfo const *a, struct fileinfo const *b,
3667           int (*cmp) (char const *, char const *))
3668 {
3669     return cmp (a->name, b->name);
3670 }
3671
3672 /* Compare file extensions.  Files with no extension are 'smallest'.
3673    If extensions are the same, compare by file names instead.  */
3674
3675 static inline int
3676 cmp_extension (struct fileinfo const *a, struct fileinfo const *b,
3677               int (*cmp) (char const *, char const *))
3678 {

```



```

3679 char const *base1 = strrchr (a->name, '.');
3680 char const *base2 = strrchr (b->name, '.');
3681 int diff = cmp (base1 ? base1 : "", base2 ? base2 : "");
3682 return diff ? diff : cmp (a->name, b->name);
3683 }
3684
3685 DEFINE_SORT_FUNCTIONS (ctime, cmp_ctime)
3686 DEFINE_SORT_FUNCTIONS (mtime, cmp_mtime)
3687 DEFINE_SORT_FUNCTIONS (atime, cmp_atime)
3688 DEFINE_SORT_FUNCTIONS (size, cmp_size)
3689 DEFINE_SORT_FUNCTIONS (name, cmp_name)
3690 DEFINE_SORT_FUNCTIONS (extension, cmp_extension)
3691
3692 /* Compare file versions.
3693 Unlike all other compare functions above, cmp_version depends only
3694 on filevercmp, which does not fail (even for locale reasons), and does not
3695 need a secondary sort key. See lib/filevercmp.h for function description.
3696
3697 All the other sort options, in fact, need xstrcoll and strcmp variants,
3698 because they all use a string comparison (either as the primary or secondary
3699 sort key), and xstrcoll has the ability to do a longjmp if strcoll fails for
3700 locale reasons. Lastly, filevercmp is ALWAYS available with gnuilib. */
3701 static inline int
3702 cmp_version (struct fileinfo const *a, struct fileinfo const *b)
3703 {
3704     return filevercmp (a->name, b->name);
3705 }
3706
3707 static int xstrcoll_version (V a, V b)
3708 { return cmp_version (a, b); }
3709 static int rev_xstrcoll_version (V a, V b)
3710 { return cmp_version (b, a); }
3711 static int xstrcoll_df_version (V a, V b)
3712 { DIRFIRST_CHECK (a, b); return cmp_version (a, b); }
3713 static int rev_xstrcoll_df_version (V a, V b)
3714 { DIRFIRST_CHECK (a, b); return cmp_version (b, a); }
3715
3716
3717 /* We have 2~3 different variants for each sort-key function
3718 (for 3 independent sort modes).
3719 The function pointers stored in this array must be dereferenced as:
3720
3721     sort_variants[sort_key][use_strcmp][reverse][dirs_first]
3722
3723 Note that the order in which sort keys are listed in the function pointer
3724 array below is defined by the order of the elements in the time_type and

```

```

3725     sort_type enums! */
3726
3727 #define LIST_SORTFUNCTION_VARIANTS(key_name)
3728 {
3729     {
3730         { xstrcoll_##key_name, xstrcoll_df_##key_name },
3731         { rev_xstrcoll_##key_name, rev_xstrcoll_df_##key_name },
3732     },
3733     {
3734         { strcmp_##key_name, strcmp_df_##key_name },
3735         { rev_strcmp_##key_name, rev_strcmp_df_##key_name },
3736     }
3737 }
3738
3739 static qsortFunc const sort_functions[][2][2][2] =
3740 {
3741     LIST_SORTFUNCTION_VARIANTS (name),
3742     LIST_SORTFUNCTION_VARIANTS (extension),
3743     LIST_SORTFUNCTION_VARIANTS (size),
3744
3745     {
3746         {
3747             { xstrcoll_version, xstrcoll_df_version },
3748             { rev_xstrcoll_version, rev_xstrcoll_df_version },
3749         },
3750
3751         /* We use NULL for the strcmp variants of version comparison
3752            since as explained in cmp_version definition, version comparison
3753            does not rely on xstrcoll, so it will never longjmp, and never
3754            need to try the strcmp fallback. */
3755         {
3756             { NULL, NULL },
3757             { NULL, NULL },
3758         }
3759     },
3760
3761     /* last are time sort functions */
3762     LIST_SORTFUNCTION_VARIANTS (mtime),
3763     LIST_SORTFUNCTION_VARIANTS (ctime),
3764     LIST_SORTFUNCTION_VARIANTS (atime)
3765 };
3766
3767 /* The number of sort keys is calculated as the sum of
3768    the number of elements in the sort_type enum (i.e., sort_numtypes)
3769    the number of elements in the time_type enum (i.e., time_numtypes) - 1
3770    This is because when sort_type==sort_time, we have up to

```

```

3771 time_numtypes possible sort keys.
3772
3773 This line verifies at compile-time that the array of sort functions has been
3774 initialized for all possible sort keys. */
3775 verify (ARRAY_CARDINALITY (sort_functions)
3776         == sort_numtypes + time_numtypes - 1 );
3777
3778 /* Set up SORTED_FILE to point to the in-use entries in CWD_FILE, in order.  */
3779
3780 static void
3781 initialize_ordering_vector (void)
3782 {
3783     for (size_t i = 0; i < cwd_n_used; i++)
3784         sorted_file[i] = &cwd_file[i];
3785 }
3786
3787 /* Sort the files now in the table.  */
3788
3789 static void
3790 sort_files (void)
3791 {
3792     bool use_strcmp;
3793
3794     if (sorted_file_alloc < cwd_n_used + cwd_n_used / 2)
3795     {
3796         free (sorted_file);
3797         sorted_file = xmalloc (cwd_n_used, 3 * sizeof *sorted_file);
3798         sorted_file_alloc = 3 * cwd_n_used;
3799     }
3800
3801     initialize_ordering_vector ();
3802
3803     if (sort_type == sort_none)
3804         return;
3805
3806     /* Try strcoll.  If it fails, fall back on strcmp.  We can't safely
3807     ignore strcoll failures, as a failing strcoll might be a
3808     comparison function that is not a total order, and if we ignored
3809     the failure this might cause qsort to dump core.  */
3810
3811     if (! setjmp (failed_strcoll))
3812         use_strcmp = false;      /* strcoll() succeeded */
3813     else
3814     {
3815         use_strcmp = true;
3816         assert (sort_type != sort_version);

```

```

3817     initialize_ordering_vector ();
3818 }
3819
3820 /* When sort_type == sort_time, use time_type as subindex. */
3821 mpsort ((void const **) sorted_file, cwd_n_used,
3822         sort_functions[sort_type + (sort_type == sort_time ? time_type : 0)]
3823         [use_strcmp][sort_reverse]
3824         [directories_first]);
3825 }
3826
3827 /* List all the files now in the table. */
3828
3829 static void
3830 print_current_files (void)
3831 {
3832     size_t i;
3833
3834     switch (format)
3835     {
3836     case one_per_line:
3837         for (i = 0; i < cwd_n_used; i++)
3838         {
3839             print_file_name_and_frills (sorted_file[i], 0);
3840             putchar ('\n');
3841         }
3842         break;
3843
3844     case many_per_line:
3845         if (! line_length)
3846             print_with_separator (' ');
3847         else
3848             print_many_per_line ();
3849         break;
3850
3851     case horizontal:
3852         if (! line_length)
3853             print_with_separator (' ');
3854         else
3855             print_horizontal ();
3856         break;
3857
3858     case with_commas:
3859         print_with_separator (',');
3860         break;
3861
3862     case long_format:

```

```

3863     for (i = 0; i < cwd_n_used; i++)
3864     {
3865         set_normal_color ();
3866         print_long_format (sorted_file[i]);
3867         DIRED_PUTCHAR ('\n');
3868     }
3869     break;
3870 }
3871 }
3872
3873 /* Replace the first %b with precomputed aligned month names.
3874    Note on glibc-2.7 at least, this speeds up the whole 'ls -lU'
3875    process by around 17%, compared to letting strftime() handle the %b. */
3876
3877 static size_t
3878 align_nstrftime (char *buf, size_t size, bool recent, struct tm const *tm,
3879                 timezone_t tz, int ns)
3880 {
3881     char const *nfmt = (use_abformat
3882                         ? abformat[recent][tm->tm_mon]
3883                         : long_time_format[recent]);
3884     return strftime (buf, size, nfmt, tm, tz, ns);
3885 }
3886
3887 /* Return the expected number of columns in a long-format timestamp,
3888    or zero if it cannot be calculated. */
3889
3890 static int
3891 long_time_expected_width (void)
3892 {
3893     static int width = -1;
3894
3895     if (width < 0)
3896     {
3897         time_t epoch = 0;
3898         struct tm tm;
3899         char buf[TIME_STAMP_LEN_MAXIMUM + 1];
3900
3901         /* In case you're wondering if localtime_rz can fail with an input time_t
3902            value of 0, let's just say it's very unlikely, but not inconceivable.
3903            The TZ environment variable would have to specify a time zone that
3904            is 2*31-1900 years or more ahead of UTC. This could happen only on
3905            a 64-bit system that blindly accepts e.g., TZ=UTC+20000000000000.
3906            However, this is not possible with Solaris 10 or glibc-2.3.5, since
3907            their implementations limit the offset to 167:59 and 24:00, resp. */
3908         if (localtime_rz (localtz, &epoch, &tm))

```

```

3909     {
3910         size_t len = align_nstrftime (buf, sizeof buf, false,
3911                                         &tm, localtime, 0);
3912         if (len != 0)
3913             width = mbsnwidth (buf, len, 0);
3914     }
3915
3916     if (width < 0)
3917         width = 0;
3918 }
3919
3920 return width;
3921 }
3922
3923 /* Print the user or group name NAME, with numeric id ID, using a
3924    print width of WIDTH columns. */
3925
3926 static void
3927 format_user_or_group (char const *name, unsigned long int id, int width)
3928 {
3929     size_t len;
3930
3931     if (name)
3932     {
3933         int width_gap = width - mbswidth (name, 0);
3934         int pad = MAX (0, width_gap);
3935         fputs (name, stdout);
3936         len = strlen (name) + pad;
3937
3938         do
3939             putchar (' ');
3940         while (pad--);
3941     }
3942     else
3943     {
3944         printf ("%*lu ", width, id);
3945         len = width;
3946     }
3947
3948     dired_pos += len + 1;
3949 }
3950
3951 /* Print the name or id of the user with id U, using a print width of
3952    WIDTH. */
3953
3954 static void

```

```

3955 format_user (uid_t u, int width, bool stat_ok)
3956 {
3957     format_user_or_group (! stat_ok ? "?" :
3958                          (numeric_ids ? NULL : getuser (u)), u, width);
3959 }
3960
3961 /* Likewise, for groups. */
3962
3963 static void
3964 format_group (gid_t g, int width, bool stat_ok)
3965 {
3966     format_user_or_group (! stat_ok ? "?" :
3967                          (numeric_ids ? NULL : getgroup (g)), g, width);
3968 }
3969
3970 /* Return the number of columns that format_user_or_group will print. */
3971
3972 static int
3973 format_user_or_group_width (char const *name, unsigned long int id)
3974 {
3975     if (name)
3976     {
3977         int len = mbswidth (name, 0);
3978         return MAX (0, len);
3979     }
3980     else
3981     {
3982         char buf[INT_BUFSIZE_BOUND (id)];
3983         sprintf (buf, "%lu", id);
3984         return strlen (buf);
3985     }
3986 }
3987
3988 /* Return the number of columns that format_user will print. */
3989
3990 static int
3991 format_user_width (uid_t u)
3992 {
3993     return format_user_or_group_width (numeric_ids ? NULL : getuser (u), u);
3994 }
3995
3996 /* Likewise, for groups. */
3997
3998 static int
3999 format_group_width (gid_t g)
4000 {

```

```

4001 return format_user_or_group_width(numeric_ids ? NULL : getgroup(g), g);
4002 }
4003
4004 /* Return a pointer to a formatted version of F->stat.st_ino,
4005    possibly using buffer, BUF, of length BUFLen, which must be at least
4006    INT_BUFSIZE_BOUND (uintmax_t) bytes. */
4007 static char *
4008 format_inode (char *buf, size_t buflen, const struct fileinfo *f)
4009 {
4010     assert (INT_BUFSIZE_BOUND (uintmax_t) <= buflen);
4011     return (f->stat_ok && f->stat.st_ino != NOT_AN_INODE_NUMBER
4012             ? umaxtostr (f->stat.st_ino, buf)
4013             : (char *) "?");
4014 }
4015
4016 /* Print information about F in long format. */
4017 static void
4018 print_long_format (const struct fileinfo *f)
4019 {
4020     char modebuf[12];
4021     char buf
4022         [LONGEST_HUMAN_READABLE + 1           /* inode */
4023          + LONGEST_HUMAN_READABLE + 1         /* size in blocks */
4024          + sizeof (modebuf) - 1 + 1           /* mode string */
4025          + INT_BUFSIZE_BOUND (uintmax_t)      /* st_nlink */
4026          + LONGEST_HUMAN_READABLE + 2         /* major device number */
4027          + LONGEST_HUMAN_READABLE + 1         /* minor device number */
4028          + TIME_STAMP_LEN_MAXIMUM + 1        /* max length of time/date */
4029         ];
4030     size_t s;
4031     char *p;
4032     struct timespec when_timespec;
4033     struct tm when_local;
4034
4035     /* Compute the mode string, except remove the trailing space if no
4036        file in this directory has an ACL or security context. */
4037     if (f->stat_ok)
4038         filemodestring (&f->stat, modebuf);
4039     else
4040     {
4041         modebuf[0] = filetype_letter[f->filetype];
4042         memset (modebuf + 1, '?', 10);
4043         modebuf[11] = '\0';
4044     }
4045     if (! any_has_acl)
4046         modebuf[10] = '\0';

```



```

4047 else if (f->acl_type == ACL_T_LSM_CONTEXT_ONLY)
4048     modebuf[10] = '.';
4049 else if (f->acl_type == ACL_T_YES)
4050     modebuf[10] = '+';
4051
4052 switch (time_type)
4053 {
4054     case time_ctime:
4055         when_timespec = get_stat_ctime (&f->stat);
4056         break;
4057     case time_mtime:
4058         when_timespec = get_stat_mtime (&f->stat);
4059         break;
4060     case time_atime:
4061         when_timespec = get_stat_atime (&f->stat);
4062         break;
4063     default:
4064         abort ();
4065 }
4066
4067 p = buf;
4068
4069 if (print_inode)
4070 {
4071     char hbuf[INT_BUFSIZE_BOUND (uintmax_t)];
4072     sprintf (p, "%*s ", inode_number_width,
4073             format_inode (hbuf, sizeof hbuf, f));
4074     /* Increment by strlen (p) here, rather than by inode_number_width + 1.
4075        The latter is wrong when inode_number_width is zero. */
4076     p += strlen (p);
4077 }
4078
4079 if (print_block_size)
4080 {
4081     char hbuf[LONGEST_HUMAN_READABLE + 1];
4082     char const *blocks =
4083         (! f->stat_ok
4084          ? "?"
4085          : human_readable (ST_NBLOCKS (f->stat), hbuf, human_output_opts,
4086                           ST_NBLOCKSIZE, output_block_size));
4087     int pad;
4088     for (pad = block_size_width - mbswidth (blocks, 0); 0 < pad; pad--)
4089         *p++ = ' ';
4090     while ((*p++ = *blocks++))
4091         continue;
4092     p[-1] = ' ';

```

```

4093 }
4094
4095 /* The last byte of the mode string is the POSIX
4096 "optional alternate access method flag". */
4097 {
4098     char hbuf[INT_BUFSIZE_BOUND (uintmax_t)];
4099     sprintf (p, "%s %*s ", modebuf, nlink_width,
4100             ! f->stat_ok ? "?" : umaxtostr (f->stat.st_nlink, hbuf));
4101 }
4102 /* Increment by strlen (p) here, rather than by, e.g.,
4103 sizeof modebuf - 2 + any_has_acl + 1 + nlink_width + 1.
4104 The latter is wrong when nlink_width is zero. */
4105 p += strlen (p);
4106
4107 DURED_INDENT ();
4108
4109 if (print_owner || print_group || print_author || print_scontext)
4110 {
4111     DURED_FPUTS (buf, stdout, p - buf);
4112
4113     if (print_owner)
4114         format_user (f->stat.st_uid, owner_width, f->stat_ok);
4115
4116     if (print_group)
4117         format_group (f->stat.st_gid, group_width, f->stat_ok);
4118
4119     if (print_author)
4120         format_user (f->stat.st_author, author_width, f->stat_ok);
4121
4122     if (print_scontext)
4123         format_user_or_group (f->scontext, 0, scontext_width);
4124
4125     p = buf;
4126 }
4127
4128 if (f->stat_ok
4129     && (S_ISCHR (f->stat.st_mode) || S_ISBLK (f->stat.st_mode)))
4130 {
4131     char majorbuf[INT_BUFSIZE_BOUND (uintmax_t)];
4132     char minorbuf[INT_BUFSIZE_BOUND (uintmax_t)];
4133     int blanks_width = (file_size_width
4134                         - (major_device_number_width + 2
4135                           + minor_device_number_width));
4136     sprintf (p, "%*s, %*s ",
4137             major_device_number_width + MAX (0, blanks_width),
4138             umaxtostr (major (f->stat.st_rdev), majorbuf),

```

```

4139         minor_device_number_width,
4140         umaxtostr (minor (f->stat.st_rdev), minorbuf));
4141     p += file_size_width + 1;
4142 }
4143 else
4144 {
4145     char hbuf[LONGEST_HUMAN_READABLE + 1];
4146     char const *size =
4147         (! f->stat_ok
4148          ? "?"
4149          : human_readable (unsigned_file_size (f->stat.st_size),
4150                           hbuf, file_human_output_opts, 1,
4151                           file_output_block_size));
4152     int pad;
4153     for (pad = file_size_width - mbswidth (size, 0); 0 < pad; pad--)
4154         *p++ = ' ';
4155     while ((*p++ = *size++))
4156         continue;
4157     p[-1] = ' ';
4158 }
4159
4160 s = 0;
4161 *p = '\1';
4162
4163 if (f->stat_ok && localtime_rz (localtz, &when_timespec.tv_sec, &when_local))
4164 {
4165     struct timespec six_months_ago;
4166     bool recent;
4167
4168     /* If the file appears to be in the future, update the current
4169        time, in case the file happens to have been modified since
4170        the last time we checked the clock. */
4171     if (timespec_cmp (current_time, when_timespec) < 0)
4172         gettimeofday (&current_time);
4173
4174     /* Consider a time to be recent if it is within the past six months.
4175        A Gregorian year has 365.2425 * 24 * 60 * 60 == 31556952 seconds
4176        on the average. Write this value as an integer constant to
4177        avoid floating point hassles. */
4178     six_months_ago.tv_sec = current_time.tv_sec - 31556952 / 2;
4179     six_months_ago.tv_nsec = current_time.tv_nsec;
4180
4181     recent = (timespec_cmp (six_months_ago, when_timespec) < 0
4182              && (timespec_cmp (when_timespec, current_time) < 0));
4183
4184     /* We assume here that all time zones are offset from UTC by a

```

```

4185         whole number of seconds. */
4186         s = align_nstrftime (p, TIME_STAMP_LEN_MAXIMUM + 1, recent,
4187                             &when_local, localtime, when_timespec.tv_nsec);
4188     }
4189
4190     if (s || !*p)
4191     {
4192         p += s;
4193         *p++ = ' ';
4194
4195         /* NUL-terminate the string -- fputs (via DURED_FPUTS) requires it. */
4196         *p = '\0';
4197     }
4198     else
4199     {
4200         /* The time cannot be converted using the desired format, so
4201            print it as a huge integer number of seconds. */
4202         char hbuf[INT_BUFSIZE_BOUND (intmax_t)];
4203         sprintf (p, "%*s ", long_time_expected_width (),
4204                 (! f->stat_ok
4205                  ? "?"
4206                  : timetostr (when_timespec.tv_sec, hbuf)));
4207         /* FIXME: (maybe) We discarded when_timespec.tv_nsec. */
4208         p += strlen (p);
4209     }
4210
4211     DURED_FPUTS (buf, stdout, p - buf);
4212     size_t w = print_name_with_quoting (f, false, &dired_obstack, p - buf);
4213
4214     if (f->filetype == symbolic_link)
4215     {
4216         if (f->linkname)
4217         {
4218             DURED_FPUTS_LITERAL (" -> ", stdout);
4219             print_name_with_quoting (f, true, NULL, (p - buf) + w + 4);
4220             if (indicator_style != none)
4221                 print_type_indicator (true, f->linkmode, unknown);
4222         }
4223     }
4224     else if (indicator_style != none)
4225         print_type_indicator (f->stat_ok, f->stat.st_mode, f->filetype);
4226 }
4227
4228 /* Write to *BUF a quoted representation of the file name NAME, if non-NULL,
4229    using OPTIONS to control quoting. *BUF is set to NAME if no quoting
4230    is required. *BUF is allocated if more space required (and the original

```

```

4231  *BUF is not deallocated).
4232  Store the number of screen columns occupied by NAME's quoted
4233  representation into WIDTH, if non-NULL.
4234  Store into PAD whether an initial space is needed for padding.
4235  Return the number of bytes in *BUF. */
4236
4237  static size_t
4238  quote_name_buf (char **inbuf, size_t bufsize, char *name,
4239                  struct quoting_options const *options,
4240                  int needs_general_quoting, size_t *width, bool *pad)
4241  {
4242      char *buf = *inbuf;
4243      size_t displayed_width IF_LINT ( = 0);
4244      size_t len = 0;
4245      bool quoted;
4246
4247      enum quoting_style qs = get_quoting_style (options);
4248      bool needs_further_quoting = qmark_funny_chars
4249                                  && (qs == shell_quoting_style
4250                                       || qs == shell_always_quoting_style
4251                                       || qs == literal_quoting_style);
4252
4253      if (needs_general_quoting != 0)
4254      {
4255          len = quotearg_buffer (buf, bufsize, name, -1, options);
4256          if (bufsize <= len)
4257          {
4258              buf = xmalloc (len + 1);
4259              quotearg_buffer (buf, len + 1, name, -1, options);
4260          }
4261
4262          quoted = (*name != *buf) || strlen (name) != len;
4263      }
4264      else if (needs_further_quoting)
4265      {
4266          len = strlen (name);
4267          if (bufsize <= len)
4268              buf = xmalloc (len + 1);
4269          memcpy (buf, name, len + 1);
4270
4271          quoted = false;
4272      }
4273      else
4274      {
4275          len = strlen (name);
4276          buf = name;

```

```

4277     quoted = false;
4278 }
4279
4280 if (needs_further_quoting)
4281 {
4282     if (MB_CUR_MAX > 1)
4283     {
4284         char const *p = buf;
4285         char const *plimit = buf + len;
4286         char *q = buf;
4287         displayed_width = 0;
4288
4289         while (p < plimit)
4290             switch (*p)
4291             {
4292                 case ' ': case '!': case '"': case '#': case '%':
4293                 case '&': case '\\': case '(': case ')': case '*':
4294                 case '+': case ',': case '-': case '.': case '/':
4295                 case '0': case '1': case '2': case '3': case '4':
4296                 case '5': case '6': case '7': case '8': case '9':
4297                 case ':': case ';': case '<': case '=': case '>':
4298                 case '?':
4299                 case 'A': case 'B': case 'C': case 'D': case 'E':
4300                 case 'F': case 'G': case 'H': case 'I': case 'J':
4301                 case 'K': case 'L': case 'M': case 'N': case 'O':
4302                 case 'P': case 'Q': case 'R': case 'S': case 'T':
4303                 case 'U': case 'V': case 'W': case 'X': case 'Y':
4304                 case 'Z':
4305                 case '[': case '\\': case ']': case '^': case '_':
4306                 case 'a': case 'b': case 'c': case 'd': case 'e':
4307                 case 'f': case 'g': case 'h': case 'i': case 'j':
4308                 case 'k': case 'l': case 'm': case 'n': case 'o':
4309                 case 'p': case 'q': case 'r': case 's': case 't':
4310                 case 'u': case 'v': case 'w': case 'x': case 'y':
4311                 case 'z': case '{': case '|': case '}': case '~':
4312                     /* These characters are printable ASCII characters. */
4313                     *q++ = *p++;
4314                     displayed_width += 1;
4315                     break;
4316                 default:
4317                     /* If we have a multibyte sequence, copy it until we
4318                        reach its end, replacing each non-printable multibyte
4319                        character with a single question mark. */
4320                     {
4321                         mbstate_t mbstate = { 0, };
4322                         do

```

```

4323 {
4324     wchar_t wc;
4325     size_t bytes;
4326     int w;
4327
4328     bytes = mbrtowc (&wc, p, plimit - p, &mbstate);
4329
4330     if (bytes == (size_t) -1)
4331     {
4332         /* An invalid multibyte sequence was
4333         encountered. Skip one input byte, and
4334         put a question mark. */
4335         p++;
4336         *q++ = '?';
4337         displayed_width += 1;
4338         break;
4339     }
4340
4341     if (bytes == (size_t) -2)
4342     {
4343         /* An incomplete multibyte character
4344         at the end. Replace it entirely with
4345         a question mark. */
4346         p = plimit;
4347         *q++ = '?';
4348         displayed_width += 1;
4349         break;
4350     }
4351
4352     if (bytes == 0)
4353         /* A null wide character was encountered. */
4354         bytes = 1;
4355
4356     w = wcwidth (wc);
4357     if (w >= 0)
4358     {
4359         /* A printable multibyte character.
4360         Keep it. */
4361         for (; bytes > 0; --bytes)
4362             *q++ = *p++;
4363         displayed_width += w;
4364     }
4365     else
4366     {
4367         /* An unprintable multibyte character.
4368         Replace it entirely with a question

```

```

4369                                     mark.  */
4370                                     p += bytes;
4371                                     *q++ = '?';
4372                                     displayed_width += 1;
4373                                     }
4374                                     }
4375                                     while (! mbsinit (&mbstate));
4376                                     }
4377                                     break;
4378                                     }
4379
4380                                     /* The buffer may have shrunk.  */
4381                                     len = q - buf;
4382                                     }
4383                                     else
4384                                     {
4385                                         char *p = buf;
4386                                         char const *plimit = buf + len;
4387
4388                                         while (p < plimit)
4389                                         {
4390                                             if (! isprint (to_uchar (*p)))
4391                                                 *p = '?';
4392                                             p++;
4393                                         }
4394                                         displayed_width = len;
4395                                     }
4396                                     }
4397                                     else if (width != NULL)
4398                                     {
4399                                         if (MB_CUR_MAX > 1)
4400                                             displayed_width = mbsnwidth (buf, len, 0);
4401                                         else
4402                                         {
4403                                             char const *p = buf;
4404                                             char const *plimit = buf + len;
4405
4406                                             displayed_width = 0;
4407                                             while (p < plimit)
4408                                             {
4409                                                 if (isprint (to_uchar (*p)))
4410                                                     displayed_width++;
4411                                                 p++;
4412                                             }
4413                                         }
4414                                     }

```



```

4416  /* Set padding to better align quoted items,
4417  and also give a visual indication that quotes are
4418  not actually part of the name. */
4419  *pad = (align_variable_outer_quotes && cwd_some_quoted && ! quoted);
4420
4421  if (width != NULL)
4422      *width = displayed_width;
4423
4424  *inbuf = buf;
4425
4426  return len;
4427  }
4428
4429  static size_t
4430  quote_name_width (const char *name, struct quoting_options const *options,
4431                   int needs_general_quoting)
4432  {
4433      char smallbuf[BUFSIZ];
4434      char *buf = smallbuf;
4435      size_t width;
4436      bool pad;
4437
4438      quote_name_buf (&buf, sizeof smallbuf, (char *) name, options,
4439                     needs_general_quoting, &width, &pad);
4440
4441      if (buf != smallbuf && buf != name)
4442          free (buf);
4443
4444      width += pad;
4445
4446      return width;
4447  }
4448
4449  /* %XX escape any input out of range as defined in RFC3986,
4450  and also if PATH, convert all path separators to '/'. */
4451  static char *
4452  file_escape (const char *str, bool path)
4453  {
4454      char *esc = xnmalloc (3, strlen (str) + 1);
4455      char *p = esc;
4456      while (*str)
4457      {
4458          if (path && ISSLASH (*str))
4459          {
4460              *p++ = '/';

```

```

4462         str++;
4463     }
4464     else if (RFC3986[to_uchar (*str)])
4465         *p++ = *str++;
4466     else
4467         p += sprintf (p, "%%02x", to_uchar (*str++));
4468 }
4469 *p = '\0';
4470 return esc;
4471 }
4472
4473 static size_t
4474 quote_name (char const *name, struct quoting_options const *options,
4475             int needs_general_quoting, const struct bin_str *color,
4476             bool allow_pad, struct obstack *stack, char const *absolute_name)
4477 {
4478     char smallbuf[BUFSIZ];
4479     char *buf = smallbuf;
4480     size_t len;
4481     bool pad;
4482
4483     len = quote_name_buf (&buf, sizeof smallbuf, (char *) name, options,
4484                           needs_general_quoting, NULL, &pad);
4485
4486     if (pad && allow_pad)
4487         DIRED_PUTCHAR (' ');
4488
4489     if (color)
4490         print_color_indicator (color);
4491
4492     /* If we're padding, then don't include the outer quotes in
4493        the --hyperlink, to improve the alignment of those links. */
4494     bool skip_quotes = false;
4495
4496     if (absolute_name)
4497     {
4498         if (align_variable_outer_quotes && cwd_some_quoted && ! pad)
4499         {
4500             skip_quotes = true;
4501             putchar (*buf);
4502         }
4503         char *h = file_escape (hostname, /* path= */ false);
4504         char *n = file_escape (absolute_name, /* path= */ true);
4505         /* TODO: It would be good to be able to define parameters
4506            to give hints to the terminal as how best to render the URI.
4507            For example since ls is outputting a dense block of URIs

```

```

4507         it would be best to not underline by default, and only
4508         do so upon hover etc. */
4509         printf ("\033]8;;file://%s%s%s\a", h, *n == '/' ? "" : "/", n);
4510         free (h);
4511         free (n);
4512     }
4513
4514     if (stack)
4515         PUSH_CURRENT_DIREN_POS (stack);
4516
4517     fwrite (buf + skip_quotes, 1, len - (skip_quotes * 2), stdout);
4518
4519     diren_pos += len;
4520
4521     if (stack)
4522         PUSH_CURRENT_DIREN_POS (stack);
4523
4524     if (absolute_name)
4525     {
4526         fputs ("\033]8;;\a", stdout);
4527         if (skip_quotes)
4528             putchar (*(buf + len - 1));
4529     }
4530
4531     if (buf != smallbuf && buf != name)
4532         free (buf);
4533
4534     return len + pad;
4535 }
4536
4537 static size_t
4538 print_name_with_quoting (const struct fileinfo *f,
4539                         bool symlink_target,
4540                         struct obstack *stack,
4541                         size_t start_col)
4542 {
4543     const char* name = symlink_target ? f->linkname : f->name;
4544
4545     const struct bin_str *color = print_with_color ?
4546                                     get_color_indicator (f, symlink_target) : NULL;
4547
4548     bool used_color_this_time = (print_with_color
4549                                 && (color || is_colored (C_NORM)));
4550
4551     size_t len = quote_name (name, filename_quoting_options, f->quoted,
4552                             color, !symlink_target, stack, f->absolute_name);

```

```

4553 process_signals ();
4554 if (used_color_this_time)
4555 {
4556     prep_non_filename_text ();
4557
4558     /* We use the byte length rather than display width here as
4559        an optimization to avoid accurately calculating the width,
4560        because we only output the clear to EOL sequence if the name
4561        _might_ wrap to the next line. This may output a sequence
4562        unnecessarily in multi-byte locales for example,
4563        but in that case it's inconsequential to the output. */
4564     if (line_length
4565         && (start_col / line_length != (start_col + len - 1) / line_length))
4566         put_indicator (&color_indicator[C_CLR_TO_EOL]);
4567 }
4568
4569 return len;
4570 }
4571
4572 static void
4573 prep_non_filename_text (void)
4574 {
4575     if (color_indicator[C_END].string != NULL)
4576         put_indicator (&color_indicator[C_END]);
4577     else
4578     {
4579         put_indicator (&color_indicator[C_LEFT]);
4580         put_indicator (&color_indicator[C_RESET]);
4581         put_indicator (&color_indicator[C_RIGHT]);
4582     }
4583 }
4584
4585 /* Print the file name of 'f' with appropriate quoting.
4586    Also print file size, inode number, and filetype indicator character,
4587    as requested by switches. */
4588
4589 static size_t
4590 print_file_name_and_frills (const struct fileinfo *f, size_t start_col)
4591 {
4592     char buf[MAX (LONGEST_HUMAN_READABLE + 1, INT_BUFSIZE_BOUND (uintmax_t))];
4593
4594     set_normal_color ();
4595
4596     if (print_inode)
4597         printf ("%*s ", format == with_commas ? 0 : inode_number_width,

```

```

4599         format_inode (buf, sizeof buf, f));
4600
4601     if (print_block_size)
4602         printf ("%*s ", format == with_commas ? 0 : block_size_width,
4603             ! f->stat_ok ? "?"
4604             : human_readable (ST_NBLOCKS (f->stat), buf, human_output_opts,
4605                 ST_NBLOCKSIZE, output_block_size));
4606
4607     if (print_scontext)
4608         printf ("%*s ", format == with_commas ? 0 : scontext_width, f->scontext);
4609
4610     size_t width = print_name_with_quoting (f, false, NULL, start_col);
4611
4612     if (indicator_style != none)
4613         width += print_type_indicator (f->stat_ok, f->stat.st_mode, f->filetype);
4614
4615     return width;
4616 }
4617
4618 /* Given these arguments describing a file, return the single-byte
4619    type indicator, or 0. */
4620 static char
4621 get_type_indicator (bool stat_ok, mode_t mode, enum filetype type)
4622 {
4623     char c;
4624
4625     if (stat_ok ? S_ISREG (mode) : type == normal)
4626     {
4627         if (stat_ok && indicator_style == classify && (mode & S_IXUGO))
4628             c = '*';
4629         else
4630             c = 0;
4631     }
4632     else
4633     {
4634         if (stat_ok ? S_ISDIR (mode) : type == directory || type == arg_directory)
4635             c = '/';
4636         else if (indicator_style == slash)
4637             c = 0;
4638         else if (stat_ok ? S_ISLNK (mode) : type == symbolic_link)
4639             c = '@';
4640         else if (stat_ok ? S_ISFIFO (mode) : type == fifo)
4641             c = '|';
4642         else if (stat_ok ? S_ISSOCK (mode) : type == sock)
4643             c = '=';
4644         else if (stat_ok && S_ISDOOR (mode))

```

```

4645         c = '>';
4646     else
4647         c = 0;
4648 }
4649 return c;
4650 }
4651
4652 static bool
4653 print_type_indicator (bool stat_ok, mode_t mode, enum filetype type)
4654 {
4655     char c = get_type_indicator (stat_ok, mode, type);
4656     if (c)
4657         DIRED_PUTCHAR (c);
4658     return !!c;
4659 }
4660
4661 /* Returns if color sequence was printed. */
4662 static bool
4663 print_color_indicator (const struct bin_str *ind)
4664 {
4665     if (ind)
4666     {
4667         /* Need to reset so not dealing with attribute combinations */
4668         if (is_colored (C_NORM))
4669             restore_default_color ();
4670         put_indicator (&color_indicator[C_LEFT]);
4671         put_indicator (ind);
4672         put_indicator (&color_indicator[C_RIGHT]);
4673     }
4674
4675     return ind != NULL;
4676 }
4677
4678 /* Returns color indicator or NULL if none. */
4679 static const struct bin_str* _GL_ATTRIBUTE_PURE
4680 get_color_indicator (const struct fileinfo *f, bool symlink_target)
4681 {
4682     enum indicator_no type;
4683     struct color_ext_type *ext;           /* Color extension */
4684     size_t len;                          /* Length of name */
4685
4686     const char* name;
4687     mode_t mode;
4688     int linkok;
4689     if (symlink_target)
4690     {

```

```

4691     name = f->linkname;
4692     mode = f->linkmode;
4693     linkok = f->linkok ? 0 : -1;
4694 }
4695 else
4696 {
4697     name = f->name;
4698     mode = FILE_OR_LINK_MODE (f);
4699     linkok = f->linkok;
4700 }
4701
4702 /* Is this a nonexistent file? If so, linkok == -1. */
4703
4704 if (linkok == -1 && is_colored (C_MISSING))
4705     type = C_MISSING;
4706 else if (!f->stat_ok)
4707 {
4708     static enum indicator_no filetype_indicator[] = FILETYPE_INDICATORS;
4709     type = filetype_indicator[f->filetype];
4710 }
4711 else
4712 {
4713     if (S_ISREG (mode))
4714     {
4715         type = C_FILE;
4716
4717         if ((mode & S_ISUID) != 0 && is_colored (C_SETUID))
4718             type = C_SETUID;
4719         else if ((mode & S_ISGID) != 0 && is_colored (C_SETGID))
4720             type = C_SETGID;
4721         else if (is_colored (C_CAP) && f->has_capability)
4722             type = C_CAP;
4723         else if ((mode & S_IXUGO) != 0 && is_colored (C_EXEC))
4724             type = C_EXEC;
4725         else if ((1 < f->stat.st_nlink) && is_colored (C_MULTIHARDLINK))
4726             type = C_MULTIHARDLINK;
4727     }
4728     else if (S_ISDIR (mode))
4729     {
4730         type = C_DIR;
4731
4732         if ((mode & S_ISVTX) && (mode & S_IWOTH)
4733             && is_colored (C_STICKY_OTHER_WRITABLE))
4734             type = C_STICKY_OTHER_WRITABLE;
4735         else if ((mode & S_IWOTH) != 0 && is_colored (C_OTHER_WRITABLE))
4736             type = C_OTHER_WRITABLE;

```

```

4737         else if ((mode & S_ISVTX) != 0 && is_colored (C_STICKY))
4738             type = C_STICKY;
4739     }
4740     else if (S_ISLNK (mode))
4741         type = C_LINK;
4742     else if (S_ISFIFO (mode))
4743         type = C_FIFO;
4744     else if (S_ISSOCK (mode))
4745         type = C_SOCK;
4746     else if (S_ISBLK (mode))
4747         type = C_BLK;
4748     else if (S_ISCHR (mode))
4749         type = C_CHR;
4750     else if (S_ISDOOR (mode))
4751         type = C_DOOR;
4752     else
4753     {
4754         /* Classify a file of some other type as C_ORPHAN. */
4755         type = C_ORPHAN;
4756     }
4757 }
4758
4759 /* Check the file's suffix only if still classified as C_FILE. */
4760 ext = NULL;
4761 if (type == C_FILE)
4762 {
4763     /* Test if NAME has a recognized suffix. */
4764
4765     len = strlen (name);
4766     name += len;           /* Pointer to final \0. */
4767     for (ext = color_ext_list; ext != NULL; ext = ext->next)
4768     {
4769         if (ext->ext.len <= len
4770             && c_strncasecmp (name - ext->ext.len, ext->ext.string,
4771                             ext->ext.len) == 0)
4772             break;
4773     }
4774 }
4775
4776 /* Adjust the color for orphaned symlinks. */
4777 if (type == C_LINK && !linkok)
4778 {
4779     if (color_symlink_as_referent || is_colored (C_ORPHAN))
4780         type = C_ORPHAN;
4781 }
4782

```



```

4783     const struct bin_str *const s
4784         = ext ? &(ext->seq) : &color_indicator[type];
4785
4786     return s->string ? s : NULL;
4787 }
4788
4789 /* Output a color indicator (which may contain nulls). */
4790 static void
4791 put_indicator (const struct bin_str *ind)
4792 {
4793     if (! used_color)
4794     {
4795         used_color = true;
4796
4797         /* If the standard output is a controlling terminal, watch out
4798            for signals, so that the colors can be restored to the
4799            default state if "ls" is suspended or interrupted. */
4800
4801         if (0 <= tcgetpgrp (STDOUT_FILENO))
4802             signal_init ();
4803
4804         prep_non_filename_text ();
4805     }
4806
4807     fwrite (ind->string, ind->len, 1, stdout);
4808 }
4809
4810 static size_t
4811 length_of_file_name_and_frills (const struct fileinfo *f)
4812 {
4813     size_t len = 0;
4814     char buf[MAX (LONGEST_HUMAN_READABLE + 1, INT_BUFSIZE_BOUND (uintmax_t))];
4815
4816     if (print_inode)
4817         len += 1 + (format == with_commas
4818                     ? strlen (umaxtostr (f->stat.st_ino, buf))
4819                     : inode_number_width);
4820
4821     if (print_block_size)
4822         len += 1 + (format == with_commas
4823                     ? strlen (! f->stat_ok ? "?"
4824                                : human_readable (ST_NBLOCKS (f->stat), buf,
4825                                                    human_output_opts, ST_NBLOCKSIZE,
4826                                                    output_block_size))
4827                     : block_size_width);
4828

```

```

4829     if (print_scontext)
4830         len += 1 + (format == with_commas ? strlen (f->scontext) : scontext_width);
4831
4832     len += quote_name_width (f->name, filename_quoting_options, f->quoted);
4833
4834     if (indicator_style != none)
4835     {
4836         char c = get_type_indicator (f->stat_ok, f->stat.st_mode, f->filetype);
4837         len += (c != 0);
4838     }
4839
4840     return len;
4841 }
4842
4843 static void
4844 print_many_per_line (void)
4845 {
4846     size_t row;                                /* Current row. */
4847     size_t cols = calculate_columns (true);
4848     struct column_info const *line_fmt = &column_info[cols - 1];
4849
4850     /* Calculate the number of rows that will be in each column except possibly
4851        for a short column on the right. */
4852     size_t rows = cwd_n_used / cols + (cwd_n_used % cols != 0);
4853
4854     for (row = 0; row < rows; row++)
4855     {
4856         size_t col = 0;
4857         size_t filesno = row;
4858         size_t pos = 0;
4859
4860         /* Print the next row. */
4861         while (1)
4862         {
4863             struct fileinfo const *f = sorted_file[filesno];
4864             size_t name_length = length_of_file_name_and_frills (f);
4865             size_t max_name_length = line_fmt->col_arr[col++];
4866             print_file_name_and_frills (f, pos);
4867
4868             filesno += rows;
4869             if (filesno >= cwd_n_used)
4870                 break;
4871
4872             indent (pos + name_length, pos + max_name_length);
4873             pos += max_name_length;
4874         }

```

```

4875     putchar ('\n');
4876 }
4877 }
4878
4879 static void
4880 print_horizontal (void)
4881 {
4882     size_t filesno;
4883     size_t pos = 0;
4884     size_t cols = calculate_columns (false);
4885     struct column_info const *line_fmt = &column_info[cols - 1];
4886     struct fileinfo const *f = sorted_file[0];
4887     size_t name_length = length_of_file_name_and_frills (f);
4888     size_t max_name_length = line_fmt->col_arr[0];
4889
4890     /* Print first entry. */
4891     print_file_name_and_frills (f, 0);
4892
4893     /* Now the rest. */
4894     for (filesno = 1; filesno < cwd_n_used; ++filesno)
4895     {
4896         size_t col = filesno % cols;
4897
4898         if (col == 0)
4899         {
4900             putchar ('\n');
4901             pos = 0;
4902         }
4903         else
4904         {
4905             indent (pos + name_length, pos + max_name_length);
4906             pos += max_name_length;
4907         }
4908
4909         f = sorted_file[filesno];
4910         print_file_name_and_frills (f, pos);
4911
4912         name_length = length_of_file_name_and_frills (f);
4913         max_name_length = line_fmt->col_arr[col];
4914     }
4915     putchar ('\n');
4916 }
4917
4918 /* Output name + SEP + ' '. */
4919
4920 static void

```

```

4921 print_with_separator (char sep)
4922 {
4923     size_t filesno;
4924     size_t pos = 0;
4925
4926     for (filesno = 0; filesno < cwd_n_used; filesno++)
4927     {
4928         struct fileinfo const *f = sorted_file[filesno];
4929         size_t len = line_length ? length_of_file_name_and_frills (f) : 0;
4930
4931         if (filesno != 0)
4932         {
4933             char separator;
4934
4935             if (! line_length
4936                 || ((pos + len + 2 < line_length)
4937                     && (pos <= SIZE_MAX - len - 2)))
4938             {
4939                 pos += 2;
4940                 separator = ' ';
4941             }
4942             else
4943             {
4944                 pos = 0;
4945                 separator = '\n';
4946             }
4947
4948             putchar (sep);
4949             putchar (separator);
4950         }
4951
4952         print_file_name_and_frills (f, pos);
4953         pos += len;
4954     }
4955     putchar ('\n');
4956 }
4957
4958 /* Assuming cursor is at position FROM, indent up to position TO.
4959    Use a TAB character instead of two or more spaces whenever possible. */
4960
4961 static void
4962 indent (size_t from, size_t to)
4963 {
4964     while (from < to)
4965     {
4966         if (tabsize != 0 && to / tabsize > (from + 1) / tabsize)

```

```

4967     {
4968         putchar ('\t');
4969         from += tabsize - from % tabsize;
4970     }
4971     else
4972     {
4973         putchar (' ');
4974         from++;
4975     }
4976 }
4977 }
4978
4979 /* Put DIRNAME/NAME into DEST, handling '.' and '/' properly. */
4980 /* FIXME: maybe remove this function someday. See about using a
4981 non-malloc'ing version of file_name_concat. */
4982
4983 static void
4984 attach (char *dest, const char *dirname, const char *name)
4985 {
4986     const char *dirnamep = dirname;
4987
4988     /* Copy dirname if it is not ".". */
4989     if (dirname[0] != '.' || dirname[1] != 0)
4990     {
4991         while (*dirnamep)
4992             *dest++ = *dirnamep++;
4993         /* Add '/' if 'dirname' doesn't already end with it. */
4994         if (dirnamep > dirname && dirnamep[-1] != '/')
4995             *dest++ = '/';
4996     }
4997     while (*name)
4998         *dest++ = *name++;
4999     *dest = 0;
5000 }
5001
5002 /* Allocate enough column info suitable for the current number of
5003 files and display columns, and initialize the info to represent the
5004 narrowest possible columns. */
5005
5006 static void
5007 init_column_info (void)
5008 {
5009     size_t i;
5010     size_t max_cols = MIN (max_idx, cwd_n_used);
5011
5012     /* Currently allocated columns in column_info. */

```

```

5013 static size_t column_info_alloc;
5014
5015 if (column_info_alloc < max_cols)
5016 {
5017     size_t new_column_info_alloc;
5018     size_t *p;
5019
5020     if (max_cols < max_idx / 2)
5021     {
5022         /* The number of columns is far less than the display width
5023            allows. Grow the allocation, but only so that it's
5024            double the current requirements. If the display is
5025            extremely wide, this avoids allocating a lot of memory
5026            that is never needed. */
5027         column_info = xnrealloc (column_info, max_cols,
5028                                 2 * sizeof *column_info);
5029         new_column_info_alloc = 2 * max_cols;
5030     }
5031     else
5032     {
5033         column_info = xnrealloc (column_info, max_idx, sizeof *column_info);
5034         new_column_info_alloc = max_idx;
5035     }
5036
5037     /* Allocate the new size_t objects by computing the triangle
5038        formula  $n * (n + 1) / 2$ , except that we don't need to
5039        allocate the part of the triangle that we've already
5040        allocated. Check for address arithmetic overflow. */
5041     {
5042         size_t column_info_growth = new_column_info_alloc - column_info_alloc;
5043         size_t s = column_info_alloc + 1 + new_column_info_alloc;
5044         size_t t = s * column_info_growth;
5045         if (s < new_column_info_alloc || t / column_info_growth != s)
5046             xalloc_die ();
5047         p = xnmalloc (t / 2, sizeof *p);
5048     }
5049
5050     /* Grow the triangle by parceling out the cells just allocated. */
5051     for (i = column_info_alloc; i < new_column_info_alloc; i++)
5052     {
5053         column_info[i].col_arr = p;
5054         p += i + 1;
5055     }
5056
5057     column_info_alloc = new_column_info_alloc;
5058 }

```

```

5060     for (i = 0; i < max_cols; ++i)
5061     {
5062         size_t j;
5063
5064         column_info[i].valid_len = true;
5065         column_info[i].line_len = (i + 1) * MIN_COLUMN_WIDTH;
5066         for (j = 0; j <= i; ++j)
5067             column_info[i].col_arr[j] = MIN_COLUMN_WIDTH;
5068     }
5069 }
5070
5071 /* Calculate the number of columns needed to represent the current set
5072 of files in the current display width. */
5073
5074 static size_t
5075 calculate_columns (bool by_columns)
5076 {
5077     size_t filesno; /* Index into cwd_file. */
5078     size_t cols; /* Number of files across. */
5079
5080     /* Normally the maximum number of columns is determined by the
5081 screen width. But if few files are available this might limit it
5082 as well. */
5083     size_t max_cols = MIN (max_idx, cwd_n_used);
5084
5085     init_column_info ();
5086
5087     /* Compute the maximum number of possible columns. */
5088     for (filesno = 0; filesno < cwd_n_used; ++filesno)
5089     {
5090         struct fileinfo const *f = sorted_file[filesno];
5091         size_t name_length = length_of_file_name_and_frills (f);
5092
5093         for (size_t i = 0; i < max_cols; ++i)
5094         {
5095             if (column_info[i].valid_len)
5096             {
5097                 size_t idx = (by_columns
5098                     ? filesno / ((cwd_n_used + i) / (i + 1))
5099                     : filesno % (i + 1));
5100                 size_t real_length = name_length + (idx == i ? 0 : 2);
5101
5102                 if (column_info[i].col_arr[idx] < real_length)
5103                 {
5104                     column_info[i].line_len += (real_length

```

```

5105         - column_info[i].col_arr[idx]);
5106         column_info[i].col_arr[idx] = real_length;
5107         column_info[i].valid_len = (column_info[i].line_len
5108                                     < line_length);
5109     }
5110 }
5111 }
5112 }
5113
5114 /* Find maximum allowed columns. */
5115 for (cols = max_cols; 1 < cols; --cols)
5116 {
5117     if (column_info[cols - 1].valid_len)
5118         break;
5119 }
5120
5121 return cols;
5122 }
5123
5124 void
5125 usage (int status)
5126 {
5127     if (status != EXIT_SUCCESS)
5128         emit_try_help ();
5129     else
5130     {
5131         printf (_("Usage: %s [OPTION]... [FILE]...\n"), program_name);
5132         fputs (_("\n
5133 List information about the FILES (the current directory by default).\n\
5134 Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.\n\
5135 "), stdout);
5136
5137         emit_mandatory_arg_note ();
5138
5139         fputs (_("\n
5140 -a, --all                do not ignore entries starting with .\n\
5141 -A, --almost-all       do not list implied . and ..\n\
5142     --author             with -l, print the author of each file\n\
5143 -b, --escape            print C-style escapes for nongraphic characters\n\
5144 "), stdout);
5145         fputs (_("\n
5146     --block-size=SIZE    with -l, scale sizes by SIZE when printing them;\n\
5147                          e.g., '--block-size=M'; see SIZE format below\n\
5148 "), stdout);
5149         fputs (_("\n
5150 -B, --ignore-backups    do not list implied entries ending with ~\n\

```



```

5151 -c                                with -lt: sort by, and show, ctime (time of last\n\
5152                                modification of file status information);\n\
5153                                with -l: show ctime and sort by name;\n\
5154                                otherwise: sort by ctime, newest first\n\
5155 "), stdout);
5156     fputs (_("\n\
5157 -C                                list entries by columns\n\
5158 --color[=WHEN]                  colorize the output; WHEN can be 'always' (default\n\
5159 \n\
5160                                if omitted), 'auto', or 'never'; more info below\n\
5161 \n\
5162 -d, --directory                list directories themselves, not their contents\n\
5163 -D, --dired                    generate output designed for Emacs' dired mode\n\
5164 "), stdout);
5165     fputs (_("\n\
5166 -f                                do not sort, enable -aU, disable -ls --color\n\
5167 -F, --classify                append indicator (one of */=>@|) to entries\n\
5168 --file-type                    likewise, except do not append '*' \n\
5169 --format=WORD                  across -x, commas -m, horizontal -x, long -l,\n\
5170                                single-column -l, verbose -l, vertical -C\n\
5171 --full-time                    like -l --time-style=full-iso\n\
5172 "), stdout);
5173     fputs (_("\n\
5174 -g                                like -l, but do not list owner\n\
5175 "), stdout);
5176     fputs (_("\n\
5177 --group-directories-first\n\
5178                                group directories before files;\n\
5179                                can be augmented with a --sort option, but any\n\
5180                                use of --sort=none (-U) disables grouping\n\
5181 "), stdout);
5182     fputs (_("\n\
5183 -G, --no-group                in a long listing, don't print group names\n\
5184 "), stdout);
5185     fputs (_("\n\
5186 -h, --human-readable          with -l and -s, print sizes like 1K 234M 2G etc.\n\
5187 --si                          likewise, but use powers of 1000 not 1024\n\
5188 "), stdout);
5189     fputs (_("\n\
5190 -H, --dereference-command-line\n\
5191                                follow symbolic links listed on the command line\n\
5192 --dereference-command-line-symlink-to-dir\n\
5193                                follow each command line symbolic link\n\
5194                                that points to a directory\n\
5195 --hide=PATTERN                do not list implied entries matching shell PATTERN\n\
5196 \n\

```

```

5197             (overridden by -a or -A)\n\
5198     "), stdout);
5199     fputs (_("\n
5200         --hyperlink[=WHEN]      hyperlink file names; WHEN can be 'always'\n\
5201                                 (default if omitted), 'auto', or 'never'\n\
5202     "), stdout);
5203     fputs (_("\n
5204         --indicator-style=WORD  append indicator with style WORD to entry names:\n\
5205     \n\
5206                                 none (default), slash (-p),\n\
5207                                 file-type (--file-type), classify (-F)\n\
5208     -i, --inode                  print the index number of each file\n\
5209     -I, --ignore=PATTERN        do not list implied entries matching shell PATTERN\n\
5210     \n\
5211     "), stdout);
5212     fputs (_("\n
5213         -k, --kibibytes          default to 1024-byte blocks for disk usage;\n\
5214                                 used only with -s and per directory totals\n\
5215     "), stdout);
5216     fputs (_("\n
5217         -l                        use a long listing format\n\
5218         -L, --dereference        when showing file information for a symbolic\n\
5219                                 link, show information for the file the link\n\
5220                                 references rather than for the link itself\n\
5221         -m                        fill width with a comma separated list of entries\n\
5222     \n\
5223     "), stdout);
5224     fputs (_("\n
5225         -n, --numeric-uid-gid    like -l, but list numeric user and group IDs\n\
5226         -N, --literal            print entry names without quoting\n\
5227         -o                        like -l, but do not list group information\n\
5228         -p, --indicator-style=slash\n\
5229                                 append / indicator to directories\n\
5230     "), stdout);
5231     fputs (_("\n
5232         -q, --hide-control-chars print ? instead of nongraphic characters\n\
5233         --show-control-chars    show nongraphic characters as-is (the default,\n\
5234                                 unless program is 'ls' and output is a terminal)\n\
5235     \n\
5236         -Q, --quote-name         enclose entry names in double quotes\n\
5237         --quoting-style=WORD    use quoting style WORD for entry names:\n\
5238                                 literal, locale, shell, shell-always,\n\
5239                                 shell-escape, shell-escape-always, c, escape\n\
5240                                 (overrides QUOTING_STYLE environment variable)\n\
5241     "), stdout);
5242     fputs (_("\n

```

```

5243     -r, --reverse                reverse order while sorting\n\
5244     -R, --recursive            list subdirectories recursively\n\
5245     -s, --size                  print the allocated size of each file, in blocks\n\
5246     ), stdout);
5247     fputs (_("\n\
5248     -S                          sort by file size, largest first\n\
5249     --sort=WORD                 sort by WORD instead of name: none (-U), size (-S)\n\
5250     ,\n\
5251                                time (-t), version (-v), extension (-X)\n\
5252     --time=WORD                 with -l, show time as WORD instead of default\n\
5253                                modification time: atime or access or use (-u);\n\
5254     \n\
5255                                ctime or status (-c); also use specified time\n\
5256                                as sort key if --sort=time (newest first)\n\
5257     ), stdout);
5258     fputs (_("\n\
5259     --time-style=TIME_STYLE    time/date format with -l; see TIME_STYLE below\n\
5260     ), stdout);
5261     fputs (_("\n\
5262     -t                          sort by modification time, newest first\n\
5263     -T, --tabsize=COLS         assume tab stops at each COLS instead of 8\n\
5264     ), stdout);
5265     fputs (_("\n\
5266     -u                          with -lt: sort by, and show, access time;\n\
5267                                with -l: show access time and sort by name;\n\
5268                                otherwise: sort by access time, newest first\n\
5269     -U                          do not sort; list entries in directory order\n\
5270     -v                          natural sort of (version) numbers within text\n\
5271     ), stdout);
5272     fputs (_("\n\
5273     -w, --width=COLS          set output width to COLS. 0 means no limit\n\
5274     -x                          list entries by lines instead of by columns\n\
5275     -X                          sort alphabetically by entry extension\n\
5276     -Z, --context              print any security context of each file\n\
5277     -1                          list one file per line. Avoid '\\n' with -q or -b\n\
5278     \n\
5279     ), stdout);
5280     fputs (HELP_OPTION_DESCRIPTION, stdout);
5281     fputs (VERSION_OPTION_DESCRIPTION, stdout);
5282     emit_size_note ();
5283     fputs (_("\n\
5284     \n\
5285     The TIME_STYLE argument can be full-iso, long-iso, iso, locale, or +FORMAT.\n\
5286     FORMAT is interpreted like in date(1). If FORMAT is FORMAT1<newline>FORMAT2,\n\
5287     then FORMAT1 applies to non-recent files and FORMAT2 to recent files.\n\
5288     TIME_STYLE prefixed with 'posix-' takes effect only outside the POSIX locale.\n\

```

```
5289 Also the TIME_STYLE environment variable sets the default style to use.\n\n5290 "), stdout);\n5291     fputs (_("\n\n5292 \n\n5293 Using color to distinguish file types is disabled both by default and\n\n5294 with --color=never.  With --color=auto, ls emits color codes only when\n\n5295 standard output is connected to a terminal.  The LS_COLORS environment\n\n5296 variable can change the settings.  Use the dircolors command to set it.\n\n5297 "), stdout);\n5298     fputs (_("\n\n5299 \n\n5300 Exit status:\n\n5301 0  if OK,\n\n5302 1  if minor problems (e.g., cannot access subdirectory),\n\n5303 2  if serious trouble (e.g., cannot access command-line argument).\n\n5304 "), stdout);\n5305     emit_ancillary_info (PROGRAM_NAME);\n5306 } \n5307 exit (status);\n5308 }
```