

/* Various processing of names.

Copyright (C) 1988, 1992, 1994, 1996, 1997, 1998, 1999, 2000, 2001,
2003, 2004, 2005, 2006, 2007 Free Software Foundation, Inc.

This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 3, or (at your option) any later
version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA. */

```
#include <system.h>
```

```
#include <fnmatch.h>
```

```
#include <hash.h>
```

```
#include <quotearg.h>
```

```
#include "common.h"
```

```

/* User and group names. */

struct group *getgrnam ();
struct passwd *getpwnam ();
#if ! HAVE_DECL_GETPWUID
struct passwd *getpwuid ();
#endif
#if ! HAVE_DECL_GETGRGID
struct group *getgrgid ();
#endif

/* Make sure you link with the proper libraries if you are running the
   Yellow Peril (thanks for the good laugh, Ian J.!), or, euh... NIS.
   This code should also be modified for non-UNIX systems to do something
   reasonable. */

static char *cached_uname;
static char *cached_gname;

static uid_t cached_uid;      /* valid only if cached_uname is not empty */
static gid_t cached_gid;     /* valid only if cached_gname is not empty */

/* These variables are valid only if nonempty. */
static char *cached_no_such_uname;
static char *cached_no_such_gname;

/* These variables are valid only if nonzero. It's not worth optimizing
   the case for weird systems where 0 is not a valid uid or gid. */
static uid_t cached_no_such_uid;
static gid_t cached_no_such_gid;

static void register_individual_file (char const *name);

/* Given UID, find the corresponding UNAME. */
void
uid_to_uname (uid_t uid, char **uname)
{
    struct passwd *passwd;

    if (uid != 0 && uid == cached_no_such_uid)
    {
        *uname = xstrdup ("");
        return;
    }

    if (!cached_uname || uid != cached_uid)
    {
        passwd = getpwuid (uid);
        if (passwd)
        {
            cached_uid = uid;

```

```

        assign_string (&cached_uname, passwd->pw_name);
    }
else
    {
        cached_no_such_uid = uid;
        *uname = xstrdup ("");
        return;
    }
}
*uname = xstrdup (cached_uname);
}

/* Given GID, find the corresponding GNAME. */
void
gid_to_gname (gid_t gid, char **gname)
{
    struct group *group;

    if (gid != 0 && gid == cached_no_such_gid)
    {
        *gname = xstrdup ("");
        return;
    }

    if (!cached_gname || gid != cached_gid)
    {
        group = getgrgid (gid);
        if (group)
        {
            cached_gid = gid;
            assign_string (&cached_gname, group->gr_name);
        }
        else
        {
            cached_no_such_gid = gid;
            *gname = xstrdup ("");
            return;
        }
    }
    *gname = xstrdup (cached_gname);
}

/* Given UNAME, set the corresponding UID and return 1, or else, return 0. */
int
uname_to_uid (char const *uname, uid_t *uidp)
{
    struct passwd *passwd;

    if (cached_no_such_uname
        && strcmp (uname, cached_no_such_uname) == 0)
        return 0;

```

```

if (!cached_username
    || username[0] != cached_username[0]
    || strcmp (username, cached_username) != 0)
{
    passwd = getpwnam (username);
    if (passwd)
        {
            cached_uid = passwd->pw_uid;
            assign_string (&cached_username, passwd->pw_name);
        }
    else
        {
            assign_string (&cached_no_such_username, username);
            return 0;
        }
}
*uidp = cached_uid;
return 1;
}

/* Given GNAME, set the corresponding GID and return 1, or else, return 0. */
int
gname_to_gid (char const *gname, gid_t *gidp)
{
    struct group *group;

    if (cached_no_such_gname
        && strcmp (gname, cached_no_such_gname) == 0)
        return 0;

    if (!cached_gname
        || gname[0] != cached_gname[0]
        || strcmp (gname, cached_gname) != 0)
        {
            group = getgrnam (gname);
            if (group)
                {
                    cached_gid = group->gr_gid;
                    assign_string (&cached_gname, gname);
                }
            else
                {
                    assign_string (&cached_no_such_gname, gname);
                    return 0;
                }
        }
    *gidp = cached_gid;
    return 1;
}

```

```
/* Names from the command call. */
```

```
static struct name *namelist; /* first name in list, if any */  
static struct name **nametail = &namelist; /* end of name list */
```

```
/* File name arguments are processed in two stages: first a  
name_array (see below) is filled, then the names from it  
are moved into the namelist.
```

This awkward process is needed only to implement --same-order option, which is meant to help process large archives on machines with limited memory. With this option on, namelist contains at most one entry, which diminishes the memory consumption.

However, I very much doubt if we still need this -- Sergey */

```
/* A name_array element contains entries of three types: */
```

```
#define NELT_NAME 0 /* File name */  
#define NELT_CHDIR 1 /* Change directory request */  
#define NELT_FMASK 2 /* Change fnmatch options request */
```

```
struct name_elt /* A name_array element. */  
{  
    char type; /* Element type, see NELT_* constants above */  
    union  
    {  
        const char *name; /* File or directory name */  
        int matching_flags; /* fnmatch options if type == NELT_FMASK */  
    } v;  
};
```

```
static struct name_elt *name_array; /* store an array of names */  
static size_t allocated_names; /* how big is the array? */  
static size_t names; /* how many entries does it have? */  
static size_t name_index; /* how many of the entries have we scanned? */
```

```
/* Check the size of name_array, reallocating it as necessary. */
```

```
static void  
check_name_alloc ()  
{  
    if (names == allocated_names)  
    {  
        if (allocated_names == 0)  
            allocated_names = 10; /* Set initial allocation */  
        name_array = x2nrealloc (name_array, &allocated_names,  
                                sizeof (name_array[0]));  
    }  
}
```

```
/* Add to name_array the file NAME with fnmatch options MATCHING_FLAGS */
```

```

void
name_add_name (const char *name, int matching_flags)
{
    static int prev_flags = 0; /* FIXME: Or EXCLUDE_ANCHORED? */
    struct name_elt *ep;

    check_name_alloc ();
    ep = &name_array[names++];
    if (prev_flags != matching_flags)
    {
        ep->type = NELT_FMASK;
        ep->v.matching_flags = matching_flags;
        prev_flags = matching_flags;
        check_name_alloc ();
        ep = &name_array[names++];
    }
    ep->type = NELT_NAME;
    ep->v.name = name;
}

/* Add to name_array a chdir request for the directory NAME */
void
name_add_dir (const char *name)
{
    struct name_elt *ep;
    check_name_alloc ();
    ep = &name_array[names++];
    ep->type = NELT_CHDIR;
    ep->v.name = name;
}

```

```

/* Names from external name file. */

static char *name_buffer; /* buffer to hold the current file name */
static size_t name_buffer_length; /* allocated length of name_buffer */

/* Set up to gather file names for tar. They can either come from a
   file or were saved from decoding arguments. */
void
name_init (void)
{
    name_buffer = xmalloc (NAME_FIELD_SIZE + 2);
    name_buffer_length = NAME_FIELD_SIZE;
}

void
name_term (void)
{
    free (name_buffer);
    free (name_array);
}

static int matching_flags; /* exclude_fnmatch options */

/* Get the next NELT_NAME element from name_array. Result is in
   static storage and can't be relied upon across two calls.

   If CHANGE_DIRS is true, treat any entries of type NELT_CHDIR as
   the request to change to the given directory. If filename_terminator
   is NUL, CHANGE_DIRS is effectively always false.

   Entries of type NELT_FMASK cause updates of the matching_flags
   value. */
struct name_elt *
name_next_elt (int change_dirs)
{
    static struct name_elt entry;
    const char *source;
    char *cursor;

    if (filename_terminator == '\0')
        change_dirs = 0;

    while (name_index != names)
    {
        struct name_elt *ep;
        size_t source_len;

        ep = &name_array[name_index++];
        if (ep->type == NELT_FMASK)
        {
            matching_flags = ep->v.matching_flags;

```

```

        continue;
    }

    source = ep->v.name;
    source_len = strlen (source);
    if (name_buffer_length < source_len)
    {
        do
        {
            name_buffer_length *= 2;
            if (! name_buffer_length)
                xalloc_die ();
        }
        while (name_buffer_length < source_len);

        free (name_buffer);
        name_buffer = xmalloc (name_buffer_length + 2);
    }
    strcpy (name_buffer, source);

    /* Zap trailing slashes. */

    cursor = name_buffer + strlen (name_buffer) - 1;
    while (cursor > name_buffer && ISSLASH (*cursor))
        *cursor-- = '\0';

    if (change_dirs && ep->type == NELT_CHDIR)
    {
        if (chdir (name_buffer) < 0)
            chdir_fatal (name_buffer);
    }
    else
    {
        if (unquote_option)
            unquote_string (name_buffer);
        if (incremental_option)
            register_individual_file (name_buffer);
        entry.type = ep->type;
        entry.v.name = name_buffer;
        return &entry;
    }
}

return NULL;
}

const char *
name_next (int change_dirs)
{
    struct name_elt *nelt = name_next_elt (change_dirs);
    return nelt ? nelt->v.name : NULL;
}

```

/* Gather names in a list for scanning. Could hash them later if we really care.

If the names are already sorted to match the archive, we just read them one by one. name_gather reads the first one, and it is called by name_match as appropriate to read the next ones. At EOF, the last name read is just left in the buffer. This option lets users of small machines extract an arbitrary number of files by doing "tar t" and editing down the list of files. */

```
void
name_gather (void)
{
    /* Buffer able to hold a single name. */
    static struct name *buffer;
    static size_t allocated_size;

    struct name_elt *ep;

    if (same_order_option)
    {
        static int change_dir;

        if (allocated_size == 0)
        {
            allocated_size = offsetof (struct name, name) + NAME_FIELD_SIZE + 1;
            buffer = xmalloc (allocated_size);
            /* FIXME: This memset is overkill, and ugly... */
            memset (buffer, 0, allocated_size);
        }

        while ((ep = name_next_elt (0)) && ep->type == NELT_CHDIR)
            change_dir = chdir_arg (xstrdup (ep->v.name));

        if (ep)
        {
            size_t needed_size;

            buffer->length = strlen (ep->v.name);
            needed_size = offsetof (struct name, name) + buffer->length + 1;
            if (allocated_size < needed_size)
            {
                do
                {
                    allocated_size *= 2;
                    if (! allocated_size)
                        xalloc_die ();
                }
                while (allocated_size < needed_size);

                buffer = xrealloc (buffer, allocated_size);
            }
        }
    }
}
```

```

    }
    buffer->change_dir = change_dir;
    strcpy (buffer->name, ep->v.name);
    buffer->next = 0;
    buffer->found_count = 0;
    buffer->matching_flags = matching_flags;

    namelist = buffer;
    nametail = &namelist->next;
}
else if (change_dir)
    addname (0, change_dir);
}
else
{
    /* Non sorted names -- read them all in. */
    int change_dir = 0;

    for (;;)
    {
        int change_dir0 = change_dir;
        while ((ep = name_next_elt (0)) && ep->type == NELT_CHDIR)
            change_dir = chdir_arg (xstrdup (ep->v.name));

        if (ep)
            addname (ep->v.name, change_dir);
        else
        {
            if (change_dir != change_dir0)
                addname (0, change_dir);
            break;
        }
    }
}

/* Add a name to the namelist. */
struct name *
addname (char const *string, int change_dir)
{
    size_t length = string ? strlen (string) : 0;
    struct name *name = xmalloc (offsetof (struct name, name) + length + 1);

    if (string)
        strcpy (name->name, string);
    else
        name->name[0] = 0;

    name->next = NULL;
    name->length = length;
    name->found_count = 0;
    name->matching_flags = matching_flags;
}

```

```

name->change_dir = change_dir;
name->dir_contents = NULL;

*nametail = name;
nametail = &name->next;
return name;
}

/* Find a match for FILE_NAME (whose string length is LENGTH) in the name
list. */
static struct name *
namelist_match (char const *file_name, size_t length)
{
    struct name *p;

    for (p = namelist; p; p = p->next)
    {
        if (p->name[0]
            && exclude_fnmatch (p->name, file_name, p->matching_flags))
            return p;
    }

    return NULL;
}

/* Return true if and only if name FILE_NAME (from an archive) matches any
name from the namelist. */
bool
name_match (const char *file_name)
{
    size_t length = strlen (file_name);

    while (1)
    {
        struct name *cursor = namelist;

        if (!cursor)
            return true;

        if (cursor->name[0] == 0)
        {
            chdir_do (cursor->change_dir);
            namelist = 0;
            nametail = &namelist;
            return true;
        }

        cursor = namelist_match (file_name, length);
        if (cursor)
        {
            if (!(ISSLASH (file_name[cursor->length]) && recursion_option)
                || cursor->found_count == 0)

```

```

        cursor->found_count++; /* remember it matched */
    if (starting_file_option)
    {
        free (namelist);
        namelist = 0;
        nametail = &namelist;
    }
    chdir_do (cursor->change_dir);

    /* We got a match. */
    return ISFOUND (cursor);
}

```

/* Filename from archive not found in namelist. If we have the whole namelist here, just return 0. Otherwise, read the next name in and compare it. If this was the last name, namelist->found_count will remain on. If not, we loop to compare the newly read name. */

```

if (same_order_option && namelist->found_count)
{
    name_gather (); /* read one more */
    if (namelist->found_count)
        return false;
}
else
    return false;
}
}

```

/* Returns true if all names from the namelist were processed.
P is the stat_info of the most recently processed entry.
The decision is postponed until the next entry is read if:

- 1) P ended with a slash (i.e. it was a directory)
- 2) P matches any entry from the namelist *and* represents a subdirectory or a file lying under this entry (in the terms of directory structure).

This is necessary to handle contents of directories. */

```

bool
all_names_found (struct tar_stat_info *p)
{
    struct name const *cursor;
    size_t len;

    if (test_label_option)
        return true;
    if (!p->file_name || occurrence_option == 0 || p->had_trailing_slash)
        return false;
    len = strlen (p->file_name);
    for (cursor = namelist; cursor; cursor = cursor->next)
    {
        if ((cursor->name[0] && !WASFOUND (cursor))

```

```

        || (len >= cursor->length && ISSLASH (p->file_name[cursor->length])))
    return false;
}
return true;
}

static inline int
is_pattern (const char *string)
{
    return strchr (string, '*') || strchr (string, '[') || strchr (string, '?');
}

static void
regex_usage_warning (const char *name)
{
    static int warned_once = 0;

    if (warn_regex_usage && is_pattern (name))
    {
        warned_once = 1;
        WARN ((0, 0,
            /* TRANSLATORS: The following three msgids form a single sentence.
            */
            _("Pattern matching characters used in file names. Please, "));
        WARN ((0, 0,
            _("use --wildcards to enable pattern matching, or --no-wildcards to")));
        WARN ((0, 0,
            _("suppress this warning. ")));
    }
}

/* Print the names of things in the namelist that were not matched. */
void
names_notfound (void)
{
    struct name const *cursor;

    for (cursor = namelist; cursor; cursor = cursor->next)
        if (!WASFOUND (cursor) && cursor->name[0])
        {
            regex_usage_warning (cursor->name);
            if (cursor->found_count == 0)
                ERROR ((0, 0, _("%s: Not found in archive"),
                    quotearg_colon (cursor->name)));
            else
                ERROR ((0, 0, _("%s: Required occurrence not found in archive"),
                    quotearg_colon (cursor->name)));
        }
}

/* Don't bother freeing the name list; we're about to exit. */
namelist = 0;
nametail = &namelist;

```

```
if (same_order_option)
{
    const char *name;

    while ((name = name_next (1)) != NULL)
    {
        regex_usage_warning (name);
        ERROR ((0, 0, _("%s: Not found in archive"),
            quotearg_colon (name)));
    }
}
}
```

```

/* Sorting name lists. */

/* Sort linked LIST of names, of given LENGTH, using COMPARE to order
names. Return the sorted list. Apart from the type `struct name'
and the definition of SUCCESSOR, this is a generic list-sorting
function, but it's too painful to make it both generic and portable
in C. */

static struct name *
merge_sort (struct name *list, int length,
            int (*compare) (struct name const*, struct name const*))
{
    struct name *first_list;
    struct name *second_list;
    int first_length;
    int second_length;
    struct name *result;
    struct name **merge_point;
    struct name *cursor;
    int counter;

# define SUCCESSOR(name) ((name)->next)

    if (length == 1)
        return list;

    if (length == 2)
    {
        if ((*compare) (list, SUCCESSOR (list)) > 0)
        {
            result = SUCCESSOR (list);
            SUCCESSOR (result) = list;
            SUCCESSOR (list) = 0;
            return result;
        }
        return list;
    }

    first_list = list;
    first_length = (length + 1) / 2;
    second_length = length / 2;
    for (cursor = list, counter = first_length - 1;
         counter;
         cursor = SUCCESSOR (cursor), counter--)
        continue;
    second_list = SUCCESSOR (cursor);
    SUCCESSOR (cursor) = 0;

    first_list = merge_sort (first_list, first_length, compare);
    second_list = merge_sort (second_list, second_length, compare);

```

```

merge_point = &result;
while (first_list && second_list)
  if ((*compare) (first_list, second_list) < 0)
    {
      cursor = SUCCESSOR (first_list);
      *merge_point = first_list;
      merge_point = &SUCCESSOR (first_list);
      first_list = cursor;
    }
  else
    {
      cursor = SUCCESSOR (second_list);
      *merge_point = second_list;
      merge_point = &SUCCESSOR (second_list);
      second_list = cursor;
    }
if (first_list)
  *merge_point = first_list;
else
  *merge_point = second_list;

return result;

#undef SUCCESSOR
}

/* A comparison function for sorting names. Put found names last;
   break ties by string comparison. */

static int
compare_names (struct name const *n1, struct name const *n2)
{
  int found_diff = WASFOUND(n2) - WASFOUND(n1);
  return found_diff ? found_diff : strcmp (n1->name, n2->name);
}

```

```
/* Add all the dirs under NAME, which names a directory, to the namelist.
   If any of the files is a directory, recurse on the subdirectory.
   DEVICE is the device not to leave, if the -l option is specified. */
```

```
static void
add_hierarchy_to_namelist (struct name *name, dev_t device)
{
  char *file_name = name->name;
  char *buffer = get_directory_contents (file_name, device);

  if (! buffer)
    name->dir_contents = "\0\0\0\0";
  else
    {
      size_t name_length = name->length;
      size_t allocated_length = (name_length >= NAME_FIELD_SIZE
                                ? name_length + NAME_FIELD_SIZE
                                : NAME_FIELD_SIZE);
      char *namebuf = xmalloc (allocated_length + 1);
                                /* FIXME: + 2 above? */

      char *string;
      size_t string_length;
      int change_dir = name->change_dir;

      name->dir_contents = buffer;
      strcpy (namebuf, file_name);
      if (! ISSLASH (namebuf[name_length - 1]))
        {
          namebuf[name_length++] = '/';
          namebuf[name_length] = '\0';
        }

      for (string = buffer; *string; string += string_length + 1)
        {
          string_length = strlen (string);
          if (*string == 'D')
            {
              struct name *np;

              if (allocated_length <= name_length + string_length)
                {
                  do
                    {
                      allocated_length *= 2;
                      if (! allocated_length)
                        xalloc_die ();
                    }
                  while (allocated_length <= name_length + string_length);

                  namebuf = xrealloc (namebuf, allocated_length + 1);
                }
            }
        }
    }
}
```

```
        strcpy (namebuf + name_length, string + 1);
        np = addname (namebuf, change_dir);
        add_hierarchy_to_namelist (np, device);
    }
}

free (namebuf);
}
}
```

```
/* Collect all the names from argv[] (or whatever), expand them into a
   directory tree, and sort them. This gets only subdirectories, not
   all files. */
```

```
void
collect_and_sort_names (void)
{
    struct name *name;
    struct name *next_name;
    int num_names;
    struct stat statbuf;

    name_gather ();

    if (listed_incremental_option)
        read_directory_file ();

    if (!namelist)
        addname (".", 0);

    for (name = namelist; name; name = next_name)
    {
        next_name = name->next;
        if (name->found_count || name->dir_contents)
            continue;
        if (name->matching_flags & EXCLUDE_WILDCARDS)
            /* NOTE: EXCLUDE_ANCHORED is not relevant here */
            /* FIXME: just skip regexps for now */
            continue;
        chdir_do (name->change_dir);
        if (name->name[0] == 0)
            continue;

        if (deref_stat (dereference_option, name->name, &statbuf) != 0)
        {
            stat_diag (name->name);
            continue;
        }
        if (S_ISDIR (statbuf.st_mode))
        {
            name->found_count++;
            add_hierarchy_to_namelist (name, statbuf.st_dev);
        }
    }

    num_names = 0;
    for (name = namelist; name; name = name->next)
        num_names++;
    namelist = merge_sort (namelist, num_names, compare_names);

    for (name = namelist; name; name = name->next)
```

```

name->found_count = 0;

if (listed_incremental_option)
{
    for (name = namelist; name && name->name[0] == 0; name++)
        ;
    if (name)
        name->dir_contents = append_incremental_renames (name->dir_contents);
}
}

```

```

/* This is like name_match, except that
1. It returns a pointer to the name it matched, and doesn't set FOUND
in structure. The caller will have to do that if it wants to.
2. If the namelist is empty, it returns null, unlike name_match, which
returns TRUE. */

```

```

struct name *
name_scan (const char *file_name)
{
    size_t length = strlen (file_name);

    while (1)
    {
        struct name *cursor = namelist_match (file_name, length);
        if (cursor)
            return cursor;

        /* Filename from archive not found in namelist. If we have the whole
        namelist here, just return 0. Otherwise, read the next name in and
        compare it. If this was the last name, namelist->found_count will
        remain on. If not, we loop to compare the newly read name. */

        if (same_order_option && namelist && namelist->found_count)
        {
            name_gather (); /* read one more */
            if (namelist->found_count)
                return 0;
        }
        else
            return 0;
    }
}

```

```

/* This returns a name from the namelist which doesn't have ->found
set. It sets ->found before returning, so successive calls will
find and return all the non-found names in the namelist. */
struct name *gnu_list_name;

```

```

char *
name_from_list (void)
{
    if (!gnu_list_name)

```

```

    gnu_list_name = namelist;
while (gnu_list_name
    && (gnu_list_name->found_count || gnu_list_name->name[0] == 0))
    gnu_list_name = gnu_list_name->next;
if (gnu_list_name)
{
    gnu_list_name->found_count++;
    chdir_do (gnu_list_name->change_dir);
    return gnu_list_name->name;
}
return 0;
}

void
blank_name_list (void)
{
    struct name *name;

    gnu_list_name = 0;
    for (name = namelist; name; name = name->next)
        name->found_count = 0;
}

/* Yield a newly allocated file name consisting of FILE_NAME concatenated to
   NAME, with an intervening slash if FILE_NAME does not already end in one. */
char *
new_name (const char *file_name, const char *name)
{
    size_t file_name_len = strlen (file_name);
    size_t namesize = strlen (name) + 1;
    int slash = file_name_len && ! ISSLASH (file_name[file_name_len - 1]);
    char *buffer = xmalloc (file_name_len + slash + namesize);
    memcpy (buffer, file_name, file_name_len);
    buffer[file_name_len] = '/';
    memcpy (buffer + file_name_len + slash, name, namesize);
    return buffer;
}

/* Return nonzero if file NAME is excluded. */
bool
excluded_name (char const *name)
{
    return excluded_file_name (excluded, name + FILE_SYSTEM_PREFIX_LEN (name));
}

```

```
/* Names to avoid dumping. */
static Hash_table *avoided_name_table;

/* Remember to not archive NAME. */
void
add_avoided_name (char const *name)
{
    hash_string_insert (&avoided_name_table, name);
}

/* Should NAME be avoided when archiving? */
bool
is_avoided_name (char const *name)
{
    return hash_string_lookup (avoided_name_table, name);
}

}
```

```
static Hash_table *individual_file_table;

static void
register_individual_file (char const *name)
{
    struct stat st;

    if (deref_stat (dereference_option, name, &st) != 0)
        return; /* Will be complained about later */
    if (S_ISDIR (st.st_mode))
        return;

    hash_string_insert (&individual_file_table, name);
}

bool
is_individual_file (char const *name)
{
    return hash_string_lookup (individual_file_table, name);
}
```

```
/* Return the size of the prefix of FILE_NAME that is removed after stripping NUM leading file name components. NUM must be positive. */
```

```
size_t  
stripped_prefix_len (char const *file_name, size_t num)  
{  
    char const *p = file_name + FILE_SYSTEM_PREFIX_LEN (file_name);  
    while (ISSLASH (*p))  
        p++;  
    while (*p)  
    {  
        bool slash = ISSLASH (*p);  
        p++;  
        if (slash)  
        {  
            if (--num == 0)  
                return p - file_name;  
        }  
        while (ISSLASH (*p))  
            p++;  
    }  
    return -1;  
}
```

```
/* Return nonzero if NAME contains ".." as a file name component. */
bool
contains_dot_dot (char const *name)
{
    char const *p = name + FILE_SYSTEM_PREFIX_LEN (name);

    for (;;) p++
    {
        if (p[0] == '.' && p[1] == '.' && (ISLASH (p[2]) || !p[2]))
            return 1;

        while (! ISLASH (*p))
            if (! *p++)
                return 0;
    }
}

}
```