B

PARSING COMMAND-LINE OPTIONS

A typical UNIX command line has the following form:

---

command [*options*] *arguments*

An option takes the form of a hyphen (-) followed by a unique character identifying the option and a possible argument for the option. An option that takes an argument may optionally be separated from that argument by white space. Multiple options can be grouped after a single hyphen, and the last option in the group may be one that takes an argument. According to these rules, the following commands are all equivalent:

```
$ grep -l -i -f patterns *.c
$ grep -lif patterns *.c
$ grep -lifpatterns *.c
```

In the above commands, the *-l* and *-i* options don't have an argument, while the *-f* option takes the string *patterns* as its argument.

Since many programs (including some of the example programs in this book) need to parse options in the above format, the facility to do so is encapsulated in a standard library function, *getopt()*.

```
#include <unistd.h>

extern int optind, opterr, optopt;
extern char *optarg;

int getopt(int argc, char *const argv[], const char *optstring);
```

See main text for description of return value

The *getopt()* function parses the set of command-line arguments given in *argc* and *argv*, which would normally be taken from the arguments of the same name to *main()*. The *optstring* argument specifies the set of options that *getopt()* should look for in *argv*. This argument consists of a sequence of characters, each of which identifies an option. SUSv3 specifies that *getopt()* should permit at least the characters in the 62-character set `[a-zA-Z0-9]` as options. Most implementations allow other characters as well, with the exception of `:`, `?`, and `-`, which have special meaning to *getopt()*. Each option character may be followed by a colon (`:`), indicating that this option expects an argument.

We parse a command line by calling *getopt()* repeatedly. Each call returns information about the next unprocessed option. If an option was found, the option character is returned as the function result. If the end of the option list was reached, *getopt()* returns −1. If an option has an argument, *getopt()* sets the global variable *optarg* to point to that argument.

Note that the function result of *getopt()* is *int*. We must not assign the result of *getopt()* to a variable of type *char*, because the comparison of the *char* variable with −1 won't work on systems where *char* is unsigned.

> If an option doesn't have an argument, then the *glibc getopt()* implementation (like most other implementations) sets *optarg* to `NULL`. However, SUSv3 doesn't specify this behavior, so applications can't portably rely on it (nor is it usually needed).
>
> SUSv3 specifies (and *glibc* implements) a related function, *getsubopt()*, that parses option arguments that consist of one of more comma-separated strings of the form *name[=value]*. See the *getsubopt(3)* manual page for details.

On each call to *getopt()*, the global variable *optind* is updated to contain the index of the next unprocessed element of *argv*. (When multiple options are grouped in a single word, *getopt()* does some internal bookkeeping to keep track of which part of the word is next to be processed.) The *optind* variable is automatically set to 1 before the first call to *getopt()*. There are two circumstances where we may make use of this variable:

- If *getopt()* returns −1, indicating that no more options are present and *optind* is less than *argc*, then *argv[optind]* is the location of the next nonoption word from the command line.
- If we are processing multiple command-line vectors or rescanning the same command line, then we must explicitly reset *optind* to 1.

The *getopt()* function returns −1, indicating the end of the option list, in the following circumstances:

- The end of the list described by *argc* plus *argv* was reached (i.e., *argv[optind]* is NULL).
- The next unprocessed word in *argv* does not start with an option delimiter (i.e., *argv[optind][0]* is not a hyphen).
- The next unprocessed word in *argv* consists of a single hyphen (i.e., *argv[optind]* is -). Some commands understand such a word as an argument with a special meaning, as described in Section 5.11.
- The next unprocessed word in *argv* consists of two hyphens (--). In this case, *getopt()* silently consumes the two hyphens and *optind* is adjusted to point to the next word after the double hyphen. This syntax enables a user to indicate the end of the options of a command, even when the next word on the command line (after the double hyphen) looks like an option (i.e., starts with a hyphen). For example, if we want to use *grep* to search for the string *−k* inside a file, then we would write *grep −− −k myfile*.

Two kinds of errors may occur as *getopt()* processes an option list. One error arises when an option that is not specified in *optstring* is encountered. The other error occurs when an argument is not supplied to an option that expects one (i.e., the option appears at the end of the command line). The rules about how *getopt()* handles and reports these errors are as follows:

- By default, *getopt()* prints an appropriate error message on standard error and returns the character ? as its function result. In this case, the global variable *optopt* returns the erroneous option character (i.e., the one that is unrecognized or whose argument is missing).
- The global variable *opterr* can be used to suppress the error messages printed by *getopt()*. By default, this variable is set to 1. If we set it to 0, then *getopt()* doesn't print error messages, but otherwise behaves as described in the preceding point. The program can detect the error via the ? function result and display a customized error message.
- Alternatively, we may suppress error messages by specifying a colon (:) as the first character in *optstring* (doing so overrides the effect of setting *opterr* to 0). In this case, an error is reported as with setting *opterr* to 0, except that an option with a missing argument is reported by returning : as the function result. This difference in return values allows us to distinguish the two types of errors (unrecognized option and missing option argument), if we need to do so.

The above error-reporting alternatives are summarized in Table B-1.

Table B-1: *getopt()* error-reporting behavior

Error-reporting method	*getopt()* displays error message?	Return for unrecognized option	Return for missing argument
default (*opterr == 1*)	Y	?	?
opterr == 0	N	?	?
: at start of *optstring*	N	?	:

Example program

Listing B-1 demonstrates the use of *getopt()* to parse the command line for two options: the *–x* option, which doesn't expect an argument, and the *–p* option which does expect an argument. This program suppresses error messages from *getopt()* by specifying a colon (:) as the first character in *optstring*.

To allow us to observe the operation of *getopt()*, we include some *printf()* calls to display the information returned by each *getopt()* call. On completion, the program prints some summary information about the specified options and also displays the next nonoption word on the command line, if there is one. The following shell session log shows the results when we run this program with different command-line arguments:

```
$ ./t_getopt -x -p hello world
opt =120 (x); optind = 2
opt =112 (p); optind = 4
-x was specified (count=1)
-p was specified with the value "hello"
First nonoption argument is "world" at argv[4]
$ ./t_getopt -p
opt = 58 (:); optind = 2; optopt =112 (p)
Missing argument (-p)
Usage: ./t_getopt [-p arg] [-x]
$ ./t_getopt -a
opt = 63 (?); optind = 2; optopt = 97 (a)
Unrecognized option (-a)
Usage: ./t_getopt [-p arg] [-x]
$ ./t_getopt -p str -- -x
opt =112 (p); optind = 3
-p was specified with the value "str"
First nonoption argument is "-x" at argv[4]
$ ./t_getopt -p -x
opt =112 (p); optind = 3
-p was specified with the value "-x"
```

Note that in the last example above, the string *–x* was interpreted as an argument to the *–p* option, rather than as an option.

Listing B-1: Using *getopt()*

getopt/t_getopt.c

```
#include <ctype.h>
#include "tlpi_hdr.h"

#define printable(ch) (isprint((unsigned char) ch) ? ch : '#')

static void             /* Print "usage" message and exit */
usageError(char *progName, char *msg, int opt)
{
    if (msg != NULL && opt != 0)
        fprintf(stderr, "%s (-%c)\n", msg, printable(opt));
    fprintf(stderr, "Usage: %s [-p arg] [-x]\n", progName);
    exit(EXIT_FAILURE);
}
```

```
int
main(int argc, char *argv[])
{
    int opt, xfnd;
    char *pstr;

    xfnd = 0;
    pstr = NULL;

    while ((opt = getopt(argc, argv, ":p:x")) != -1) {
        printf("opt =%3d (%c); optind = %d", opt, printable(opt), optind);
        if (opt == '?' || opt == ':')
            printf("; optopt =%3d (%c)", optopt, printable(optopt));
        printf("\n");

        switch (opt) {
        case 'p': pstr = optarg;        break;
        case 'x': xfnd++;               break;
        case ':': usageError(argv[0], "Missing argument", optopt);
        case '?': usageError(argv[0], "Unrecognized option", optopt);
        default:  fatal("Unexpected case in switch()");
        }
    }

    if (xfnd != 0)
        printf("-x was specified (count=%d)\n", xfnd);
    if (pstr != NULL)
        printf("-p was specified with the value \"%s\"\n", pstr);
    if (optind < argc)
        printf("First nonoption argument is \"%s\" at argv[%d]\n",
                argv[optind], optind);
    exit(EXIT_SUCCESS);
}
```

getopt/t_getopt.c

GNU-specific behavior

By default, the *glibc* implementation of *getopt()* implements a nonstandard feature: it allows options and nonoptions to be intermingled. Thus, for example, the following are equivalent:

```
$ ls -l file
$ ls file -l
```

In processing command lines of the second form, *getopt()* permutes the contents of *argv* so that all options are moved to the beginning of the array and all nonoptions are moved to the end of the array. (If *argv* contains an element pointing to the word `--`, then only the elements preceding that element are subject to permutation and interpretation as options.) In other words, the `const` declaration of *argv* in the *getopt()* prototype shown earlier is not actually true for *glibc*.

Permuting the contents of *argv* is not permitted by SUSv3 (or SUSv4). We can force *getopt()* to provide standards-conformant behavior (i.e., to follow the rules

listed earlier for determining the end of the option list) by setting the environment variable POSIXLY_CORRECT to any value. This can be done in two ways:

- From within the program, we can call *putenv()* or *setenv()*. This has the advantage that the user is not required to do anything. It has the disadvantages that it requires modifications of the program source code and that it changes the behavior of only that program.
- We can define the variable from the shell before we execute the program:

  ```
  $ export POSIXLY_CORRECT=y
  ```

  This method has the advantage that it affects all programs that use *getopt()*. However, it also has some disadvantages. POSIXLY_CORRECT causes other changes in the behavior of various Linux tools. Furthermore, setting this variable requires explicit user intervention (most likely by setting the variable in a shell startup file).

An alternative method of preventing *getopt()* from permuting command-line arguments is to make the first character of *optstring* a plus sign (+). (If we want to also suppress *getopt()* error messages as described above, then the first two characters of *optstring* should be +:, in that order.) As with the use of *putenv()* or *setenv()*, this approach has the disadvantage that it requires changes to the program code. See the *getopt(3)* manual page for further details.

> A future technical corrigendum of SUSv4 is likely to add a specification for the use of the plus sign in *optstring* to prevent permutation of command-line arguments.

Note that the *glibc getopt()* permuting behavior affects how we write shell scripts. (This affects developers porting shell scripts from other systems to Linux.) Suppose we have a shell script that performs the following command on all of the files in a directory:

```
chmod 644 *
```

If one of these filenames starts with a hyphen, then the *glibc getopt()* permuting behavior would cause that filename to be interpreted as an option to *chmod*. This would not happen on other UNIX implementations, where the occurrence of the first nonoption (644) ensures that *getopt()* ceases looking for options in the remainder of the command line. For most commands, (if we don't set POSIXLY_CORRECT, then) the way of dealing with this possibility in shell scripts that must run on Linux is to place the string -- before the first nonoption argument. Thus, we would rewrite the above line as follows:

```
chmod -- 644 *
```

In this particular example, which employs filename generation, we could alternatively write this:

```
chmod 644 ./*
```

Although we have used the example of filename pattern matching (globbing) above, similar scenarios can also occur as a result of other shell processing (e.g., command substitution and parameter expansion), and they can be dealt with similarly, by using a -- string to separate options from arguments.

GNU extensions

The GNU C library provides a number of extensions to *getopt()*. We briefly note the following:

- The SUSv3 specification permits options to have only mandatory arguments. In the GNU version of *getopt()*, we can place two colons after an option character in *optstring* to indicate that its argument is optional. The argument to such an option must appear in the same word as the option itself (i.e., no spaces may appear between the option and its argument). If the argument is not present, then, on return from *getopt()*, *optarg* is set to `NULL`.
- Many GNU commands allow a form of long option syntax. A long option begins with two hyphens, and the option itself is identified using a word, rather than a single character, as in the following example:

  ```
  $ gcc --version
  ```

  The *glibc* function *getopt_long()* can be used to parse such options.
- The GNU C library provides an even more sophisticated (but nonportable) API for parsing the command-line, called *argp*. This API is described in the *glibc* manual.