# Chapter 2

# Arguments, Options, and the Environment

In this chapter

Command-line option and argument interpretation is usually the first task of any program. This chapter examines how C (and C++) programs access their command-line arguments, describes standard routines for parsing options, and takes a look at the environment.

## 2.1 Option and Argument Conventions

The word *arguments* has two meanings. The more technical definition is "all the 'words' on the command line." For example:

```
$ ls main.c opts.c process.c
```

Here, the user typed four "words." All four words are made available to the program as its arguments.

The second definition is more informal: Arguments are all the words on the command line *except* the command name. By default, Unix shells separate arguments from each other with *whitespace* (spaces or TAB characters). Quoting allows arguments to include whitespace:

```
$ echo here are lots    of spaces
here are lots of spaces                    The shell "eats" the spaces
$ echo "here are lots    of spaces"
here are lots    of spaces                 Spaces are preserved
```

Quoting is transparent to the running program; `echo` never sees the double-quote characters. (Double and single quotes are different in the shell; a discussion of the rules is beyond the scope of this book, which focuses on C programming.)

Arguments can be further classified as *options* or *operands*. In the previous two examples all the arguments were operands: files for `ls` and raw text for `echo`.

Options are special arguments that each program interprets. Options change a program's behavior, or they provide information to the program. By ancient convention, (almost) universally adhered to, options start with a dash (a.k.a. hyphen, minus sign) and consist of a single letter. *Option arguments* are information needed by an option, as opposed to regular operand arguments. For example, the `fgrep` program's `-f` option means "use the contents of the following file as a list of strings to search for." See Figure 2.1.

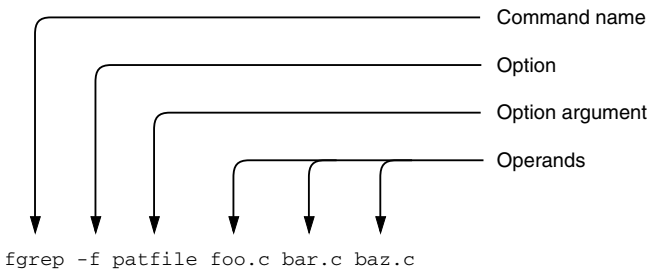


FIGURE 2.1
Command-line components

Thus, `patfile` is not a data file to search, but rather it's for use by `fgrep` in defining the list of strings to search for.

### 2.1.1 POSIX Conventions

The POSIX standard describes a number of conventions that standard-conforming programs adhere to. Nothing requires that your programs adhere to these standards, but it's a good idea for them to do so: Linux and Unix users the world over understand and use these conventions, and if your program doesn't follow them, your users will be unhappy. (Or you won't have any users!) Furthermore, the functions we discuss later in this chapter relieve you of the burden of manually adhering to these conventions for each program you write. Here they are, paraphrased from the standard:

1. Program names should have no less than two and no more than nine characters.
2. Program names should consist of only lowercase letters and digits.
3. Option names should be single alphanumeric characters. Multidigit options should not be allowed. For vendors implementing the POSIX utilities, the `-W` option is reserved for vendor-specific options.
4. All options should begin with a '`-`' character.
5. For options that don't require option arguments, it should be possible to group multiple options after a single '`-`' character. (For example, '`foo -a -b -c`' and '`foo -abc`' should be treated the same way.)
6. When an option does require an option argument, the argument should be separated from the option by a space (for example, '`fgrep -f patfile`').

The standard, however, does allow for historical practice, whereby sometimes the option and the operand could be in the same string: '`fgrep -fpatfile`'. In practice, the `getopt()` and `getopt_long()` functions interpret '`-fpatfile`' as '`-f patfile`', not as '`-f -p -a -t ...`'.

7. Option arguments should not be optional.

   This means that when a program documents an option as requiring an option argument, that option's argument must always be present or else the program will fail. GNU `getopt()` does provide for optional option arguments since they're occasionally useful.

8. If an option takes an argument that may have multiple values, the program should receive that argument as a single string, with values separated by commas or whitespace.

   For example, suppose a hypothetical program `myprog` requires a list of users for its `-u` option. Then, it should be invoked in one of these two ways:

   ```
   myprog -u "arnold,joe,jane"          Separate with commas
   myprog -u "arnold joe jane"          Separate with whitespace
   ```

   In such a case, you're on your own for splitting out and processing each value (that is, there is no standard routine), but doing so manually is usually straightforward.

9. Options should come first on the command line, before operands. Unix versions of `getopt()` enforce this convention. GNU `getopt()` does not by default, although you can tell it to.

10. The special argument '`--`' indicates the end of all options. Any subsequent arguments on the command line are treated as operands, even if they begin with a dash.

11. The order in which options are given should not matter. However, for mutually exclusive options, when one option overrides the setting of another, then (so to speak) the last one wins. If an option that has arguments is repeated, the program should process the arguments in order. For example, '`myprog -u arnold -u jane`' is the same as '`myprog -u "arnold,jane"`'. (You have to enforce this yourself; `getopt()` doesn't help you.)

12. It is OK for the order of operands to matter to a program. Each program should document such things.

13. Programs that read or write named files should treat the single argument '-' as meaning standard input or standard output, as is appropriate for the program.

Note that many standard programs don't follow all of the above conventions. The primary reason is historical compatibility; many such programs predate the codifying of these conventions.

### 2.1.2 GNU Long Options

As we saw in Section 1.4.2, "Program Behavior," page 16, GNU programs are encouraged to use long options of the form `--help`, `--verbose`, and so on. Such options, since they start with '`--`', do not conflict with the POSIX conventions. They also can be easier to remember, and they provide the opportunity for consistency across all GNU utilities. (For example, `--help` is the same everywhere, as compared with `-h` for "help," `-i` for "information," and so on.) GNU long options have their own conventions, implemented by the `getopt_long()` function:

1. For programs implementing POSIX utilities, every short (single-letter) option should also have a long option.
2. Additional GNU-specific long options need not have a corresponding short option, but we recommend that they do.
3. Long options can be abbreviated to the shortest string that remains unique. For example, if there are two options `--verbose` and `--verbatim`, the shortest possible abbreviations are `--verbo` and `--verba`.
4. Option arguments are separated from long options either by whitespace or by an `=` sign. For example, `--sourcefile=/some/file` or `--sourcefile /some/file`.
5. Options and arguments may be interspersed with operands on the command line; `getopt_long()` will rearrange things so that all options are processed and then all operands are available sequentially. (This behavior can be suppressed.)
6. Option arguments can be optional. For such options, the argument is deemed to be present if it's in the same string as the option. This works only for short options. For example, if `-x` is such an option, given '`foo -xYANKEES -y`', the argument to `-x` is '`YANKEES`'. For '`foo -x -y`', there is no argument to `-x`.

7. Programs can choose to allow long options to begin with a single dash. (This is common with many X Window programs.)

Much of this will become clearer when we examine `getopt_long()` later in the chapter.

The *GNU Coding Standards* devotes considerable space to listing all the long and short options used by GNU programs. If you're writing a program that accepts long options, see if option names already in use might make sense for you to use as well.

## 2.2 Basic Command-Line Processing

A C program accesses its command-line arguments through its parameters, `argc` and `argv`. The `argc` parameter is an integer, indicating the number of arguments there are, including the command name. There are two common ways to declare `main()`, varying in how `argv` is declared:

```
int main(int argc, char *argv[])
{
    ...
}
```

```
int main(int argc, char **argv)
{
...
}
```

Practically speaking, there's no difference between the two declarations, although the first is conceptually clearer: `argv` is an array of pointers to characters. The second is more commonly used: `argv` is a pointer to a pointer. Also, the second definition is technically more correct, and it is what we use. Figure 2.2 depicts this situation.

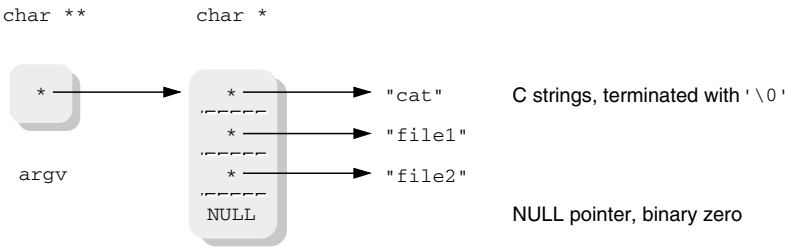


FIGURE 2.2
Memory for `argv`

By convention, `argv[0]` is the program's name. (For details, see Section 9.1.4.3, "Program Names and `argv[0]`," page 297.) Subsequent entries are the command line arguments. The final entry in the `argv` array is a `NULL` pointer.

`argc` indicates how many arguments there are; since C is zero-based, it is always true that '`argv[argc] == NULL`'. Because of this, particularly in Unix code, you will see different ways of checking for the end of arguments, such as looping until a counter is greater than or equal to `argc`, or until '`argv[i] == 0`' or while '`*argv != NULL`' and so on. These are all equivalent.

### 2.2.1 The V7 `echo` Program

Perhaps the simplest example of command-line processing is the V7 `echo` program, which prints its arguments to standard output, separated by spaces and terminated with a newline. If the first argument is `-n`, then the trailing newline is omitted. (This is used for prompting from shell scripts.) Here's the code:[1]

```
 1  #include <stdio.h>
 2
 3  main(argc, argv)                          int main(int argc, char **argv)
 4  int argc;
 5  char *argv[];
 6  {
 7      register int i, nflg;
 8
 9      nflg = 0;
10      if(argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n') {
11          nflg++;
12          argc--;
13          argv++;
14      }
15      for(i=1; i<argc; i++) {
16          fputs(argv[i], stdout);
17          if (i < argc-1)
18              putchar(' ');
19      }
20      if(nflg == 0)
21          putchar('\n');
22      exit(0);
23  }
```

Only 23 lines! There are two points of interest. First, decrementing `argc` and simultaneously incrementing `argv` (lines 12 and 13) are common ways of skipping initial arguments. Second, the check for `-n` (line 10) is simplistic. `-no-newline-at-the-end` also works. (Compile it and try it!)

---

[1] See `/usr/src/cmd/echo.c` in the V7 distribution.

Manual option parsing is common in V7 code because the `getopt()` function hadn't been invented yet.

Finally, here and in other places throughout the book, we see use of the `register` keyword. At one time, this keyword provided a hint to the compiler that the given variables should be placed in CPU registers, if possible. Use of this keyword is obsolete; modern compilers all base register assignment on analysis of the source code, ignoring the `register` keyword. We've chosen to leave code using it alone, but you should be aware that it has no real use anymore.[2]

## 2.3 Option Parsing: `getopt()` and `getopt_long()`

Circa 1980, for System III, the Unix Support Group within AT&T noted that each Unix program used ad hoc techniques for parsing arguments. To make things easier for users and developers, they developed most of the conventions we listed earlier. (The statement in the System III *intro*(1) manpage is considerably less formal than what's in the POSIX standard, though.)

The Unix Support Group also developed the `getopt()` function, along with several external variables, to make it easy to write code that follows the standard conventions. The GNU `getopt_long()` function supplies a compatible version of `getopt()`, as well as making it easy to parse long options of the form described earlier.

### 2.3.1 Single-Letter Options

The `getopt()` function is declared as follows:

```
#include <unistd.h>                                                    POSIX

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

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

The arguments `argc` and `argv` are normally passed straight from those of `main()`. `optstring` is a string of option letters. If any letter in the string is followed by a colon, then that option is expected to have an argument.

---

[2] When we asked Jim Meyering, the Coreutils maintainer, about instances of `register` in the GNU Coreutils, he gave us an interesting response. He removes them when modifying code, but otherwise leaves them alone to make it easier to integrate changes submitted against existing versions.

To use getopt(), call it repeatedly from a while loop until it returns -1. Each time that it finds a valid option letter, it returns that letter. If the option takes an argument, optarg is set to point to it. Consider a program that accepts a -a option that doesn't take an argument and a -b argument that does:

```
int oc;             /* option character */
char *b_opt_arg;

while ((oc = getopt(argc, argv, "ab:")) != -1) {
    switch (oc) {
    case 'a':
        /* handle -a, set a flag, whatever */
        break;
    case 'b':
        /* handle -b, get arg value from optarg */
        b_opt_arg = optarg;
        break;
    case ':':
        ...       /* error handling, see text */
    case '?':
    default:
        ...       /* error handling, see text */
    }
}
```

As it works, getopt() sets several variables that control error handling.

`char *optarg`
: The argument for an option, if the option accepts one.

`int optind`
: The current index in argv. When the while loop has finished, remaining operands are found in argv[optind] through argv[argc-1]. (Remember that 'argv[argc] == NULL'.)

`int opterr`
: When this variable is nonzero (which it is by default), getopt() prints its own error messages for invalid options and for missing option arguments.

`int optopt`
: When an invalid option character is found, getopt() returns either a '?' or a ':' (see below), and optopt contains the invalid character that was found.

People being human, it is inevitable that programs will be invoked incorrectly, either with an invalid option or with a missing option argument. In the normal case, getopt()

prints its own messages for these cases and returns the `'?'` character. However, you can change its behavior in two ways.

First, by setting `opterr` to `0` before invoking `getopt()`, you can force `getopt()` to remain silent when it finds a problem.

Second, if the *first* character in the `optstring` argument is a colon, then `getopt()` is silent *and* it returns a different character depending upon the error, as follows:

*Invalid option*
: `getopt()` returns a `'?'` and `optopt` contains the invalid option character. (This is the normal behavior.)

*Missing option argument*
: `getopt()` returns a `':'`. If the first character of `optstring` is not a colon, then `getopt()` returns a `'?'`, making this case indistinguishable from the invalid option case.

Thus, making the first character of `optstring` a colon is a good idea since it allows you to distinguish between "invalid option" and "missing option argument." The cost is that using the colon also silences `getopt()`, forcing you to supply your own error messages. Here is the previous example, this time with error message handling:

```
int oc;             /* option character */
char *b_opt_arg;

while ((oc = getopt(argc, argv, ":ab:")) != -1) {
    switch (oc) {
    case 'a':
        /* handle -a, set a flag, whatever */
        break;
    case 'b':
        /* handle -b, get arg value from optarg */
        b_opt_arg = optarg;
        break;
    case ':':
        /* missing option argument */
        fprintf(stderr, "%s: option `-%c' requires an argument\n",
                argv[0], optopt);
        break;
    case '?':
    default:
        /* invalid option */
        fprintf(stderr, "%s: option `-%c' is invalid: ignored\n",
                argv[0], optopt);
        break;
    }
}
```

A word about flag or option variable-naming conventions: Much Unix code uses names of the form *`xflg`* for any given option letter *`x`* (for example, `nflg` in the V7 `echo`; *`xflag`* is also common). This may be great for the program's author, who happens to know what the *`x`* option does without having to check the documentation. But it's unkind to someone else trying to read the code who doesn't know the meaning of all the option letters by heart. It is much better to use names that convey the option's meaning, such as `no_newline` for `echo`'s `-n` option.

### 2.3.2 GNU `getopt()` and Option Ordering

The standard `getopt()` function stops looking for options as soon as it encounters a command-line argument that doesn't start with a '`-`'. GNU `getopt()` is different: It scans the entire command line looking for options. As it goes along, it *permutes* (rearranges) the elements of `argv`, so that when it's done, all the options have been moved to the front and code that proceeds to examine `argv[optind]` through `argv[argc-1]` works correctly. In all cases, the special argument '`--`' terminates option scanning.

You can change the default behavior by using a special first character in `optstring`, as follows:

`optstring[0] == '+'`
: GNU `getopt()` behaves like standard `getopt()`; it returns options in the order in which they are found, stopping at the first nonoption argument. This will also be true if `POSIXLY_CORRECT` exists in the environment.

`optstring[0] == '-'`
: GNU `getopt()` returns *every* command-line argument, whether or not it represents an argument. In this case, for each such argument, the function returns the integer `1` and sets `optarg` to point to the string.

As for standard `getopt()`, if the first character of `optstring` is a `':'`, then GNU `getopt()` distinguishes between "invalid option" and "missing option argument" by returning `'?'` or `':'`, respectively. The `':'` in `optstring` can be the second character if the first character is `'+'` or `'-'`.

Finally, if an option letter in `optstring` is followed by *two* colon characters, then that option is allowed to have an optional option argument. (Say that three times fast!) Such an argument is deemed to be present if it's in the same `argv` element as the option,

and absent otherwise. In the case that it's absent, GNU `getopt()` returns the option letter and sets `optarg` to `NULL`. For example, given—

```
while ((c = getopt(argc, argv, "ab::")) != 1)
   ...
```

—for `-bYANKEES`, the return value is `'b'`, and `optarg` points to `"YANKEES"`, while for `-b` or '`-b YANKEES`', the return value is still `'b'` but `optarg` is set to `NULL`. In the latter case, `"YANKEES"` is a separate command-line argument.

### 2.3.3 Long Options

The `getopt_long()` function handles the parsing of long options of the form described earlier. An additional routine, `getopt_long_only()` works identically, but it is used for programs where *all* options are long and options begin with a single '`-`' character. Otherwise, both work just like the simpler GNU `getopt()` function. (For brevity, whenever we say "`getopt_long()`," it's as if we'd said "`getopt_long()` and `getopt_long_only()`.") Here are the declarations, from the GNU/Linux *getopt*(3) manpage:

```
#include <getopt.h>                                                    GLIBC

int getopt_long(int argc, char *const argv[],
           const char *optstring,
           const struct option *longopts, int *longindex);

int getopt_long_only(int argc, char *const argv[],
           const char *optstring,
           const struct option *longopts, int *longindex);
```

The first three arguments are the same as for `getopt()`. The next option is a pointer to an array of `struct option`, which we refer to as the *long options table* and which is described shortly. The `longindex` parameter, if not set to `NULL`, points to a variable which is filled in with the index in `longopts` of the long option that was found. This is useful for error diagnostics, for example.

#### 2.3.3.1 Long Options Table

Long options are described with an array of `struct option` structures. The `struct option` is declared in `<getopt.h>`; it looks like this:

```
struct option {
    const char *name;
    int has_arg;
    int *flag;
    int val;
};
```

The elements in the structure are as follows:

`const char *name`
: This is the name of the option, *without* any leading dashes, for example, `"help"` or `"verbose"`.

`int has_arg`
: This describes whether the long option has an argument, and if so, what kind of argument. The value must be one of those presented in Table 2.1.

  The symbolic constants are macros for the numeric values given in the table. While the numeric values work, the symbolic constants are considerably easier to read, and you should use them instead of the corresponding numbers in any code that you write.

`int *flag`
: If this pointer is `NULL`, then `getopt_long()` returns the value in the `val` field of the structure. If it's not `NULL`, the variable it points to is filled in with the value in `val` and `getopt_long()` returns `0`. If the `flag` isn't `NULL` but the long option is never seen, then the pointed-to variable is not changed.

`int val`
: This is the value to return if the long option is seen or to load into `*flag` if `flag` is not `NULL`. Typically, if `flag` is not `NULL`, then `val` is a true/false value, such as `1` or `0`. On the other hand, if `flag` is `NULL`, then `val` is usually a character constant. If the long option corresponds to a short one, the character constant should be the same one that appears in the `optstring` argument for this option. (All of this will become clearer shortly when we see some examples.)

Each long option has a single entry with the values appropriately filled in. The last element in the array should have zeros for all the values. The array need not be sorted; `getopt_long()` does a linear search. However, sorting it by long name may make it easier for a programmer to read.

TABLE 2.1
Values for `has_arg`

| Symbolic constant | Numeric value | Meaning |
|---|---|---|
| `no_argument` | 0 | The option does not take an argument. |
| `required_argument` | 1 | The option requires an argument. |
| `optional_argument` | 2 | The option's argument is optional. |

The use of `flag` and `val` seems confusing at first encounter. Let's step back for a moment and examine why it works the way it does. Most of the time, option processing consists of setting different flag variables when different option letters are seen, like so:

```
while ((c = getopt(argc, argv, ":af:hv")) != -1) {
    switch (c) {
    case 'a':
        do_all = 1;
        break;
    case 'f':
        myfile = optarg;
        break;
    case 'h':
        do_help = 1;
        break;
    case 'v':
        do_verbose = 1;
        break;
    ...                 Error handling code here
    }
}
```

When `flag` is not `NULL`, `getopt_long()` *sets the variable for you*. This reduces the three cases in the previous `switch` to one case. Here is an example long options table and the code to go with it:

```
int do_all, do_help, do_verbose;    /* flag variables */
char *myfile;

struct option longopts[] = {
   { "all",     no_argument,       & do_all,     1   },
   { "file",    required_argument, NULL,         'f' },
   { "help",    no_argument,       & do_help,    1   },
   { "verbose", no_argument,       & do_verbose, 1   },
   { 0, 0, 0, 0 }
};
...
```

```
while ((c = getopt_long(argc, argv, ":f:", longopts, NULL)) != -1) {
    switch (c) {
    case 'f':
        myfile = optarg;
        break;
    case 0:
        /* getopt_long() set a variable, just keep going */
        break;
    ...                 Error handling code here
    }
}
```

Notice that the value passed for the `optstring` argument no longer contains `'a'`, `'h'`, or `'v'`. This means that the corresponding short options are not accepted. To allow both long and short options, you would have to restore the corresponding `case`s from the first example to the `switch`.

Practically speaking, you should write your programs such that each short option also has a corresponding long option. In this case, it's easiest to have `flag` be `NULL` and `val` be the corresponding single letter.

#### 2.3.3.2 Long Options, POSIX Style

The POSIX standard reserves the `-W` option for vendor-specific features. Thus, by definition, `-W` isn't portable across different systems.

If `W` appears in the `optstring` argument followed by a semicolon (note: *not* a colon), then `getopt_long()` treats `-Wlongopt` the same as `--longopt`. Thus, in the previous example, change the call to be:

```
while ((c = getopt_long(argc, argv, ":f:W;", longopts, NULL)) != -1) {
```

With this change, `-Wall` is the same as `--all` and `-Wfile=myfile` is the same as `--file=myfile`. The use of a semicolon makes it possible for a program to use `-W` as a regular option, if desired. (For example, GCC uses it as a regular option, whereas `gawk` uses it for POSIX conformance.)

#### 2.3.3.3 `getopt_long()` Return Value Summary

As should be clear by now, `getopt_long()` provides a flexible mechanism for option parsing. Table 2.2 summarizes the possible return values and their meaning.

TABLE 2.2
`getopt_long()` return values

| Return code | Meaning |
|---|---|
| `0` | `getopt_long()` set a flag as found in the long option table. |
| `1` | `optarg` points at a plain command-line argument. |
| `'?'` | Invalid option. |
| `':'` | Missing option argument. |
| `'x'` | Option character `'x'`. |
| `-1` | End of options. |

Finally, we enhance the previous example code, showing the full `switch` statement:

```
int do_all, do_help, do_verbose;    /* flag variables */
char *myfile, *user;                /* input file, user name */

struct option longopts[] = {
   { "all",     no_argument,       & do_all,     1   },
   { "file",    required_argument, NULL,         'f' },
   { "help",    no_argument,       & do_help,    1   },
   { "verbose", no_argument,       & do_verbose, 1   },
   { "user"   , optional_argument, NULL,         'u' },
   { 0, 0, 0, 0 }
};
...
while ((c = getopt_long(argc, argv, ":ahvf:u::W;", longopts, NULL)) != -1) {
    switch (c) {
    case 'a':
        do_all = 1;
        break;
    case 'f':
        myfile = optarg;
        break;
    case 'h':
        do_help = 1;
        break;
    case 'u':
        if (optarg != NULL)
           user = optarg;
        else
           user = "root";
        break;
    case 'v':
        do_verbose = 1;
        break;
    case 0:     /* getopt_long() set a variable, just keep going */
        break;
```

```
#if 0
    case 1:
        /*
         * Use this case if getopt_long() should go through all
         * arguments.  If so, add a leading '-' character to optstring.
         * Actual code, if any, goes here.
         */
        break;
#endif
    case ':':   /* missing option argument */
        fprintf(stderr, "%s: option `-%c' requires an argument\n",
                argv[0], optopt);
        break;
    case '?':
    default:    /* invalid option */
        fprintf(stderr, "%s: option `-%c' is invalid: ignored\n",
                argv[0], optopt);
        break;
    }
}
```

In your programs, you may wish to have comments for each option letter explaining what each one does. However, if you've used descriptive variable names for each option letter, comments are not as necessary. (Compare `do_verbose` to `vflg`.)

#### 2.3.3.4 GNU `getopt()` or `getopt_long()` in User Programs

You may wish to use GNU `getopt()` or `getopt_long()` in your own programs and have them run on non-Linux systems. That's OK; just copy the source files from a GNU program or from the GNU C Library (GLIBC) CVS archive.[3] The source files are `getopt.h`, `getopt.c`, and `getopt1.c`. They are licensed under the GNU Lesser General Public License, which allows library functions to be included even in proprietary programs. You should include a copy of the file `COPYING.LIB` with your program, along with the files `getopt.h`, `getopt.c`, and `getopt1.c`.

Include the source files in your distribution, and compile them along with any other source files. In your source code that calls `getopt_long()`, use '`#include <getopt.h>`', not '`#include "getopt.h"`'. Then, when compiling, add `-I.` to the C compiler's command line. That way, the local copy of the header file will be found first.

---

[3] See `http://sources.redhat.com`.

You may be wondering, "Gee, I already use GNU/Linux. Why should I include `getopt_long()` in my executable, making it bigger, if the routine is already in the C library?" That's a good question. However, there's nothing to worry about. The source code is set up so that if it's compiled on a system that uses GLIBC, the compiled files will not contain any code! Here's the proof, on our system:

```
$ uname -a                                          Show system name and type
Linux example 2.4.18-14 #1 Wed Sep 4 13:35:50 EDT 2002 i686 i686 i386 GNU/Linux
$ ls -l getopt.o getopt1.o                          Show file sizes
-rw-r--r--    1 arnold   devel        9836 Mar 24 13:55 getopt.o
-rw-r--r--    1 arnold   devel       10324 Mar 24 13:55 getopt1.o
$ size getopt.o getopt1.o                           Show sizes included in executable
   text    data     bss     dec     hex filename
      0       0       0       0       0 getopt.o
      0       0       0       0       0 getopt1.o
```

The `size` command prints the sizes of the various parts of a binary object or executable file. We explain the output in Section 3.1, "Linux/Unix Address Space," page 52. What's important to understand right now is that, despite the nonzero sizes of the files themselves, they don't contribute anything to the final executable. (We think this is pretty neat.)

## 2.4 The Environment

The *environment* is a set of '*name*=*value*' pairs for each program. These pairs are termed *environment variables*. Each *name* consists of one to any number of alphanumeric characters or underscores ('_'), but the name may not start with a digit. (This rule is enforced by the shell; the C API can put anything it wants to into the environment, at the likely cost of confusing subsequent programs.)

Environment variables are often used to control program behavior. For example, if `POSIXLY_CORRECT` exists in the environment, many GNU programs disable extensions or historical behavior that isn't compatible with the POSIX standard.

You can decide (and should document) the environment variables that your program will use to control its behavior. For example, you may wish to use an environment variable for debugging options instead of a command-line argument. The advantage of using environment variables is that users can set them in their startup file and not have to remember to always supply a particular set of command-line options.

Of course, the disadvantage to using environment variables is that they can *silently* change a program's behavior. Jim Meyering, the maintainer of the Coreutils, put it this way:

> It makes it easy for the user to customize how the program works without changing how the program is invoked. That can be both a blessing and a curse. If you write a script that depends on your having a certain environment variable set, but then have someone else use that same script, it may well fail (or worse, silently produce invalid results) if that other person doesn't have the same environment settings.

### 2.4.1 Environment Management Functions

Several functions let you retrieve the values of environment variables, change their values, or remove them. Here are the declarations:

```
#include <stdlib.h>

char *getenv(const char *name);                          ISO C: Retrieve environment variable
int setenv(const char *name, const char *value,          POSIX: Set environment variable
           int overwrite);
int putenv(char *string);                                XSI: Set environment variable, uses string
void unsetenv(const char *name);                         POSIX: Remove environment variable
int clearenv(void);                                      Common: Clear entire environment
```

The `getenv()` function is the one you will use 99 percent of the time. The argument is the environment variable name to look up, such as `"HOME"` or `"PATH"`. If the variable exists, `getenv()` returns a pointer to the character string value. If not, it returns `NULL`. For example:

```
char *pathval;

/* Look for PATH; if not present, supply a default value */
if ((pathval = getenv("PATH")) == NULL)
    pathval = "/bin:/usr/bin:/usr/ucb";
```

Occasionally, environment variables exist, but with empty values. In this case, the return value will be non-`NULL`, but the first character pointed to will be the zero byte, which is the C string terminator, `'\0'`. Your code should be careful to check that the return value pointed to is not `NULL`. Even if it isn't `NULL`, also check that the string is not empty if you intend to use its value for something. In any case, don't just blindly use the returned value.

To change an environment variable or to add a new one to the environment, use `setenv()`:

```
if (setenv("PATH", "/bin:/usr/bin:/usr/ucb", 1) != 0) {
    /* handle failure */
}
```

It's possible that a variable already exists in the environment. If the third argument is true (nonzero), then the supplied value overwrites the previous one. Otherwise, it doesn't. The return value is `-1` if there was no memory for the new variable, and `0` otherwise. `setenv()` makes private copies of both the variable name and the new value for storing in the environment.

A simpler alternative to `setenv()` is `putenv()`, which takes a single "*`name=value`*" string and places it in the environment:

```
if (putenv("PATH=/bin:/usr/bin:/usr/ucb") != 0) {
    /* handle failure */
}
```

`putenv()` blindly replaces any previous value for the same variable. Also, and perhaps more importantly, the string passed to `putenv()` is placed *directly* into the environment. This means that if your code later modifies this string (for example, if it was an array, not a string constant), the environment is modified also. This in turn means that you should *not* use a local variable as the parameter for `putenv()`. For all these reasons `setenv()` is preferred.

> NOTE The GNU `putenv()` has an additional (documented) quirk to its behavior. If the argument string is a name, then without an = character, the named variable is *removed*. The GNU `env` program, which we look at later in this chapter, relies on this behavior.

The `unsetenv()` function removes a variable from the environment:

```
unsetenv("PATH");
```

Finally, the `clearenv()` function clears the environment entirely:

```
if (clearenv() != 0) {
    /* handle failure */
}
```

This function is not standardized by POSIX, although it's available in GNU/Linux and several commercial Unix variants. You should use it if your application must be very security conscious and you want it to build its own environment entirely from

scratch. If `clearenv()` is not available, the GNU/Linux *clearenv*(3) manpage recommends using '`environ = NULL;`' to accomplish the task.

### 2.4.2 The Entire Environment: `environ`

The correct way to deal with the environment is through the functions described in the previous section. However, it's worth a look at how things are managed "under the hood."

The external variable `environ` provides access to the environment in the same way that `argv` provides access to the command-line arguments. You must declare the variable yourself. Although standardized by POSIX, `environ` is purposely not declared by any standardized header file. (This seems to evolve from historical practice.) Here is the declaration:

```
extern char **environ;   /* Look Ma, no header file! */          POSIX
```

Like `argv`, the final element in `environ` is `NULL`. There is no "environment count" variable that corresponds to `argc`, however. This simple program prints out the entire environment:

```
/* ch02-printenv.c --- Print out the environment. */

#include <stdio.h>

extern char **environ;

int main(int argc, char **argv)
{
    int i;

    if (environ != NULL)
        for (i = 0; environ[i] != NULL; i++)
            printf("%s\n", environ[i]);

    return 0;
}
```

Although it's unlikely to happen, this program makes sure that `environ` isn't `NULL` before attempting to use it.

Variables are kept in the environment in random order. Although some Unix shells keep the environment sorted by variable name, there is no formal requirement that this be so, and many shells don't keep them sorted.

As something of a quirk of the implementation, you can access the environment by declaring a *third* parameter to `main()`:

```
int main(int argc, char **argv, char **envp)
{
    ...
}
```

You can then use `envp` as you would have used `environ`. Although you may see this occasionally in old code, we don't recommend its use; `environ` is the official, standard, portable way to access the entire environment, should you need to do so.

### 2.4.3 GNU `env`

To round off the chapter, here is the GNU version of the `env` command. This command adds variables to the environment for the duration of one command. It can also be used to clear the environment for that command or to remove specific environment variables. The program serves double-duty for us, since it demonstrates both `getopt_long()` and several of the functions discussed in this section. Here is how the program is invoked:

```
$ env --help
Usage: env [OPTION]... [-] [NAME=VALUE]... [COMMAND [ARG]...]
Set each NAME to VALUE in the environment and run COMMAND.

  -i, --ignore-environment   start with an empty environment
  -u, --unset=NAME           remove variable from the environment
      --help     display this help and exit
      --version  output version information and exit

A mere - implies -i.  If no COMMAND, print the resulting environment.

Report bugs to <bug-coreutils@gnu.org>.
```

Here are some sample invocations:

```
$ env - myprog arg1                          Clear environment, run program with args

$ env - PATH=/bin:/usr/bin myprog arg1       Clear environment, add PATH, run program

$ env -u IFS PATH=/bin:/usr/bin myprog arg1  Unset IFS, add PATH, run program
```

The code begins with a standard GNU copyright statement and explanatory comment. We have omitted both for brevity. (The copyright statement is discussed in Appendix C, "GNU General Public License," page 657. The `--help` output shown previously is enough to understand how the program works.) Following the copyright and comments

are header includes and declarations. The '`N_("string")`' macro invocation (line 93) is for use in internationalization and localization of the software, topics covered in Chapter 13, "Internationalization and Localization," page 485. For now, you can treat it as if it were the contained string constant.

```
 80  #include <config.h>
 81  #include <stdio.h>
 82  #include <getopt.h>
 83  #include <sys/types.h>
 84  #include <getopt.h>
 85
 86  #include "system.h"
 87  #include "error.h"
 88  #include "closeout.h"
 89
 90  /* The official name of this program (e.g., no `g' prefix).  */
 91  #define PROGRAM_NAME "env"
 92
 93  #define AUTHORS N_ ("Richard Mlynarik and David MacKenzie")
 94
 95  int putenv ();
 96
 97  extern char **environ;
 98
 99  /* The name by which this program was run. */
100  char *program_name;
101
102  static struct option const longopts[] =
103  {
104    {"ignore-environment", no_argument, NULL, 'i'},
105    {"unset", required_argument, NULL, 'u'},
106    {GETOPT_HELP_OPTION_DECL},
107    {GETOPT_VERSION_OPTION_DECL},
108    {NULL, 0, NULL, 0}
109  };
```

The GNU Coreutils contain a large number of programs, many of which perform the same common tasks (for example, argument parsing). To make maintenance easier, many common idioms are defined as macros. `GETOPT_HELP_OPTION_DECL` and `GETOPT_VERSION_OPTION` (lines 106 and 107) are two such. We examine their definitions shortly. The first function, `usage()`, prints the usage information and exits. The `_("string")` macro (line 115, and used throughout the program) is also for internationalization, and for now you should also treat it as if it were the contained string constant.

```
111 void
112 usage (int status)
113 {
114   if (status != 0)
115     fprintf (stderr, _("Try `%s --help' for more information.\n"),
116              program_name);
117   else
118     {
119       printf (_("\
120 Usage: %s [OPTION]... [-] [NAME=VALUE]... [COMMAND [ARG]...]\n"),
121               program_name);
122       fputs (_("\
123 Set each NAME to VALUE in the environment and run COMMAND.\n\
124 \n\
125   -i, --ignore-environment   start with an empty environment\n\
126   -u, --unset=NAME           remove variable from the environment\n\
127 "), stdout);
128       fputs (HELP_OPTION_DESCRIPTION, stdout);
129       fputs (VERSION_OPTION_DESCRIPTION, stdout);
130       fputs (_("\
131 \n\
132 A mere - implies -i.  If no COMMAND, print the resulting environment.\n\
133 "), stdout);
134       printf (_("\nReport bugs to <%s>.\n"), PACKAGE_BUGREPORT);
135     }
136   exit (status);
137 }
```

The first part of `main()` declares variables and sets up the internationalization. The functions `setlocale()`, `bindtextdomain()`, and `textdomain()` (lines 147–149) are all discussed in Chapter 13, "Internationalization and Localization," page 485. Note that this program does use the `envp` argument to `main()` (line 140). It is the only one of the Coreutils programs to do so. Finally, the call to `atexit()` on line 151 (see Section 9.1.5.3, "Exiting Functions," page 302) registers a Coreutils library function that flushes all pending output and closes `stdout`, reporting a message if there were problems. The next bit processes the command-line arguments, using `getopt_long()`.

```
139 int
140 main (register int argc, register char **argv, char **envp)
141 {
142   char *dummy_environ[1];
143   int optc;
144   int ignore_environment = 0;
145
146   program_name = argv[0];
147   setlocale (LC_ALL, "");
148   bindtextdomain (PACKAGE, LOCALEDIR);
149   textdomain (PACKAGE);
150
151   atexit (close_stdout);
```

```
152
153     while ((optc = getopt_long (argc, argv, "+iu:", longopts, NULL)) != -1)
154       {
155         switch (optc)
156           {
157           case 0:
158             break;
159           case 'i':
160             ignore_environment = 1;
161             break;
162           case 'u':
163             break;
164           case_GETOPT_HELP_CHAR;
165           case_GETOPT_VERSION_CHAR (PROGRAM_NAME, AUTHORS);
166           default:
167             usage (2);
168           }
169       }
170
171     if (optind != argc && !strcmp (argv[optind], "-"))
172       ignore_environment = 1;
```

Here are the macros, from `src/sys2.h` in the Coreutils distribution, that define the declarations we saw earlier and the '`case_GETOPT_xxx`' macros used above (lines 164–165):

```
/* Factor out some of the common --help and --version processing code.  */

/* These enum values cannot possibly conflict with the option values
   ordinarily used by commands, including CHAR_MAX + 1, etc.  Avoid
   CHAR_MIN - 1, as it may equal -1, the getopt end-of-options value.  */
enum
{
  GETOPT_HELP_CHAR = (CHAR_MIN - 2),
  GETOPT_VERSION_CHAR = (CHAR_MIN - 3)
};

#define GETOPT_HELP_OPTION_DECL \
  "help", no_argument, 0, GETOPT_HELP_CHAR
#define GETOPT_VERSION_OPTION_DECL \
  "version", no_argument, 0, GETOPT_VERSION_CHAR

#define case_GETOPT_HELP_CHAR                   \
  case GETOPT_HELP_CHAR:                        \
    usage (EXIT_SUCCESS);                       \
    break;

#define case_GETOPT_VERSION_CHAR(Program_name, Authors)                 \
  case GETOPT_VERSION_CHAR:                                             \
    version_etc (stdout, Program_name, PACKAGE, VERSION, Authors);      \
    exit (EXIT_SUCCESS);                                                \
    break;
```

The upshot of this code is that `--help` prints the usage message and `--version` prints version information. Both exit successfully. ("Success" and "failure" exit statuses are described in Section 9.1.5.1, "Defining Process Exit Status," page 300.) Given that the Coreutils have dozens of utilities, it makes sense to factor out and standardize as much repetitive code as possible.

Returning to `env.c`:

```
174     environ = dummy_environ;
175     environ[0] = NULL;
176
177     if (!ignore_environment)
178       for (; *envp; envp++)
179         putenv (*envp);
180
181     optind = 0;                     /* Force GNU getopt to re-initialize. */
182     while ((optc = getopt_long (argc, argv, "+iu:", longopts, NULL)) != -1)
183       if (optc == 'u')
184         putenv (optarg);            /* Requires GNU putenv. */
185
186     if (optind != argc && !strcmp (argv[optind], "-"))    Skip options
187       ++optind;
188
189     while (optind < argc && strchr (argv[optind], '='))   Set environment variables
190       putenv (argv[optind++]);
191
192     /* If no program is specified, print the environment and exit. */
193     if (optind == argc)
194       {
195         while (*environ)
196           puts (*environ++);
197         exit (EXIT_SUCCESS);
198       }
```

Lines 174–179 copy the existing environment into a fresh copy of the environment. The global variable `environ` is set to point to an empty local array. The `envp` parameter maintains access to the original environment.

Lines 181–184 remove any environment variables as requested by the `-u` option. The program does this by rescanning the command line and removing names listed there. Environment variable removal relies on the GNU `putenv()` behavior discussed earlier: that when called with a plain variable name, `putenv()` removes the environment variable.

After any options, new or replacement environment variables are supplied on the command line. Lines 189–190 continue scanning the command line, looking for environment variable settings of the form '*`name`*`=`*`value`*'.

Upon reaching line 192, if nothing is left on the command line, `env` is supposed to print the new environment, and exit. It does so (lines 195–197).

If arguments are left, they represent a command name to run and arguments to pass to that new command. This is done with the `execvp()` system call (line 200), which *replaces* the current program with the new one. (This call is discussed in Section 9.1.4, "Starting New Programs: The `exec()` Family," page 293; don't worry about the details for now.) If this call returns to the current program, it *failed*. In such a case, `env` prints an error message and exits.

```
200     execvp (argv[optind], &argv[optind]);
201
202     {
203       int exit_status = (errno == ENOENT ? 127 : 126);
204       error (0, errno, "%s", argv[optind]);
205       exit (exit_status);
206     }
207   }
```

The exit status values, `126` and `127` (determined on line 203), conform to POSIX. `127` means the program that `execvp()` attempted to run didn't exist. (`ENOENT` means the file doesn't have an entry in the directory.) `126` means that the file exists, but something else went wrong.

## 2.5 Summary

- C programs access their command-line arguments through the parameters `argc` and `argv`. The `getopt()` function provides a standard way for consistent parsing of options and their arguments. The GNU version of `getopt()` provides some extensions, and `getopt_long()` and `getopt_long_only()` make it possible to easily parse long-style options.
- The environment is a set of '*`name`*`=`*`value`*' pairs that each program inherits from its parent. Programs can, at their author's whim, use environment variables to change their behavior, in addition to any command-line arguments. Standard routines (`getenv()`, `setenv()`, `putenv()`, and `unsetenv()`) exist for retrieving environment variable values, changing them, or removing them. If necessary, the entire environment is available through the external variable `environ` or through the `char **envp` third argument to `main()`. The latter technique is discouraged.

## Exercises

1. Assume a program accepts options `-a`, `-b`, and `-c`, and that `-b` requires an argument. Write the manual argument parsing code for this program, without using `getopt()` or `getopt_long()`. Accept `--` to end option processing. Make sure that `-ac` works, as do `-bYANKEES`, `-b YANKEES`, and `-abYANKEES`. Test your program.
2. Implement `getopt()`. For the first version, don't worry about the case in which '`optstring[0] == ':'`'. You may also ignore `opterr`.
3. Add code for '`optstring[0] == ':'`' and `opterr` to your version of `getopt()`.
4. Print and read the GNU `getopt.h`, `getopt.c` and `getopt1.c` files.
5. Write a program that declares both `environ` and `envp` and compares their values.
6. Parsing command line arguments and options is a wheel that many people can't refrain from reinventing. Besides `getopt()` and `getopt_long()`, you may wish to examine different argument-parsing packages, such as:
   - The *Plan 9 From Bell Labs arg*(2) argument-parsing library,[4]
   - Argp,[5]
   - Argv,[6]
   - Autoopts,[7]
   - GNU Gengetopt,[8]
   - Opt,[9]
   - Popt.[10] See also the *popt*(3) manpage on a GNU/Linux system.
7. Extra credit: Why can't a C compiler completely ignore the `register` keyword? Hint: What operation *cannot* be applied to a `register` variable?

---

[4] `http://plan9.bell-labs.com/magic/man2html/2/arg`

[5] `http://www.gnu.org/manual/glibc/html_node/Argp.html`

[6] `http://256.com/sources/argv`

[7] `http://autogen.sourceforge.net/autoopts.html`

[8] `ftp://ftp.gnu.org/gnu/gengetopt/`

[9] `http://nis-www.lanl.gov/~jt/Software/opt/opt-3.19.tar.gz`

[10] `http://freshmeat.net/projects/popt/?topic_id=809`