CHAPTER 24

Parsing Command-Line Options

Most Linux programs allow the user to specify command-line options. Such options perform a wide variety of functions but are fairly uniform in their syntax. **Short options** consist of a - character followed by a single alphanumeric character. **Long options**, common in GNU utilities, consist of two - characters followed by a string made up of letters, numbers, and hyphens. Either type of option may be followed by an argument. A space separates a short option from its arguments; either a space or an = separates a long option from an argument.

There are many ways of parsing command-line options. The most popular method is parsing the `argv` array by hand. The `getopt()` and `getopt_long()` library functions provide some assistance for option parsing. `getopt()` is provided by many Unix implementations, but it supports only short options. The `getopt_long()` function is available on Linux and allows automated parsing of both short and long options.

A library called popt exists specifically for option parsing. It includes a number of advantages over the `getopt()` functions.

- It does not make use of global variables, which allows it to be used when multiple passes are needed to parse `argv`.
- It can parse an arbitrary array of `argv`-style elements. This allows popt to be used for parsing command-line-style strings from any source.
- It provides a standard method of option aliasing. Programs that use popt can easily allow users to add new command-line options, which are defined as combinations of already-existing options. This allows the user to define new, complex behaviors or change the default behaviors of existing options.

Like `getopt_long()`, the popt library supports short and long style options.

The popt library is highly portable and should work on any POSIX platform. The latest version is always available from ftp://ftp.redhat.com/pub/redhat/code/popt/

It may be redistributed under either the GNU General Public License or the GNU Library General Public License, at the distributor's discretion.

24.1 Basic popt Usage

24.1.1 The Option Table

Applications provide popt with information on their command-line options through an array of `struct poptOption` structures.

```
#include <popt.h>

struct poptOption {
    const char * longName; /* may be NULL */
    char shortName;        /* may be '\0' */
    int argInfo;
    void * arg;            /* depends on argInfo */
    int val;               /* 0 means do not return, just update flag */
};
```

Table 24.1 popt Argument Types

Value	**Description**	`arg` **Type**
`POPT_ARG_NONE`	No argument is expected	`int`
`POPT_ARG_STRING`	No type checking should be performed	`char *`
`POPT_ARG_INT`	An integer argument is expected	`int`
`POPT_ARG_LONG`	A long integer is expected	`long`

Each member of the table defines a single option that may be passed to the program. Long and short options are considered a single option that may occur in two different forms. The first two members, `longName` and `shortName`, define the names of the option; the first is a long name, and the latter is a single character.

The `argInfo` member tells popt what type of argument is expected after the argument. If no option is expected, `POPT_ARG_NONE` should be used. The rest of the valid values are summarized in Table 24.1.[1]

The next element, `arg`, allows popt to automatically update program variables when the option is used. If `arg` is `NULL`, it is ignored and popt takes no special action. Otherwise, it should point to a variable of the type indicated in the right-most column of Table 24.1.

If the option takes no argument (`argInfo` is `POPT_ARG_NONE`), the variable pointed to by `arg` is set to 1 when the option is used. If the option does take an argument, the variable that `arg` points to is updated to reflect the value of the argument. Any string is acceptable for `POPT_ARG_STRING` arguments, but `POPT_ARG_INT` and `POPT_ARG_LONG` arguments are converted to the appropriate type, and an error is returned if the conversion fails.

The final option, `val`, is the value popt's parsing function should return when the option is encountered. If it is 0, the parsing function parses the next command-line argument rather than return.

The final structure in the table should have all the pointer values set to `NULL` and all the arithmetic values set to 0, marking the end of the table.

1. `getopt()` connoisseurs will note that `argInfo` is the only field of `struct poptOption` that is not directly analogous to a field in the `getopt_long()` argument table. The similarity between the two allows for easy transitions from `getopt_long()` to popt.

24.1.2 Creating a Context

popt can interleave the parsing of multiple command-line sets. It allows this by keeping all the state information for a particular set of command-line arguments in a `poptContext` data structure, an opaque type that should not be modified outside the popt library.

New popt contexts are created by `poptGetContext()`.

```
#include <popt.h>

poptContext poptGetContext(char * name, int argc, char ** argv,
                           struct poptOption * options, int flags);
```

The first parameter, `name`, is used only for alias handling (discussed later). It should be the name of the application whose options are being parsed, or should be `NULL` if no option aliasing is desired. The next two arguments specify the command-line arguments to parse. These are generally passed to `poptGetContext()` exactly as they were passed to the program's `main()` function. The `options` parameter points to the table of command-line options, which was described in the previous section. The final parameter, `flags`, is not currently used but should always be specified as 0 for compatibility with future versions of the popt library.

A `poptContext` keeps track of which options have already been parsed and which remain, among other things. If a program wishes to restart option processing of a set of arguments, it can reset the `poptContext` by passing the context as the sole argument to `poptResetContext()`.

When argument processing is complete, the process should free the `poptContext` as it contains dynamically allocated components. The `poptFreeContext()` function takes a `poptContext` as its sole argument and frees the resources the context is using.

Here are the prototypes of both `poptResetContext()` and `poptFreeContext()`.

```
#include <popt.h>

void poptFreeContext(poptContext con);
void poptResetContext(poptContext con);
```

24.1.3 Parsing the Command Line

After an application has created a `poptContext`, it may begin parsing arguments. The `poptGetNextOpt()` performs the actual argument parsing.

```
#include <popt.h>

int poptGetNextOpt(poptContext con);
```

Taking the context as its sole argument, this function parses the next command-line argument found. After finding the next argument in the option table, the function fills in the object pointed to by the option table entry's `arg` pointer if it is not `NULL`. If the `val` entry for the option is non-0, the function then returns that value. Otherwise, `poptGetNextOpt()` continues on to the next argument.

`poptGetNextOpt()` returns -1 when the final argument has been parsed, and other negative values when errors occur. This makes it a good idea to keep the `val` elements in the options table greater than 0.

If all of the command-line options are handled through `arg` pointers, command-line parsing is reduced to the following line of code:

```
rc = poptGetNextOpt(poptcon);
```

Many applications require more complex command-line parsing than this, however, and use the following structure.

```
while ((rc = poptGetNextOpt(poptcon)) > 0) {
    switch (rc) {
        /* specific arguments are handled here */
    }
}
```

When returned options are handled, the application needs to know the value of any arguments that were specified after the option. There are two ways to discover them. One is to ask popt to fill in a variable with the value of the option through the option table's `arg` elements. The other is to use `poptGetOptArg()`.

```
#include <popt.h>

char * poptGetOptArg(poptContext con);
```

This function returns the argument given for the final option returned by `poptGetNextOpt()`, or it returns `NULL` if no argument was specified.

24.1.4 Leftover Arguments

Many applications take an arbitrary number of command-line arguments, such as a list of file names. When popt encounters an argument that does not begin with a -, it assumes it is such an argument and adds it to a list of leftover arguments. Three functions allow applications to access such arguments:

`char * poptGetArg(poptContext con);`
: This function returns the next leftover argument and marks it as processed.

`char * poptPeekArg(poptContext con);`
: The next leftover argument is returned but not marked as processed. This allows an application to look ahead into the argument list, without modifying the list.

`char ** poptGetArgs(poptContext con);`
: All the leftover arguments are returned in a manner identical to `argv`. The final element in the returned array points to `NULL`, indicating the end of the arguments.

24.2 Error Handling

All of the popt functions that can return errors return integers. When an error occurs, a negative error code is returned. Table 24.2 summarizes the error codes that occur. Here is a more detailed discussion of each error.

Table 24.2 popt Errors

Error	Description
POPT_ERROR_NOARG	An argument is missing for an option.
POPT_ERROR_BADOPT	An option's argument could not be parsed.
POPT_ERROR_OPTSTOODEEP	Option aliasing is nested too deeply.
POPT_ERROR_BADQUOTE	Quotations do not match.
POPT_ERROR_BADNUMBER	An option could not be converted to a number.
POPT_ERROR_OVERFLOW	A given number was too big or too small.

POPT_ERROR_NOARG

An option that requires an argument was specified on the command line, but no argument was given. This can be returned only by poptGetNextOpt().

POPT_ERROR_BADOPT

An option was specified in argv but is not in the option table. This error can be returned only from poptGetNextOpt().

POPT_ERROR_OPTSTOODEEP

A set of option aliases is nested too deeply. Currently, popt follows options only 10 levels to prevent infinite recursion. Only poptGetNextOpt() can return this error.

POPT_ERROR_BADQUOTE

A parsed string has a quotation mismatch (such as a single quotation mark). poptParseArgvString(), poptReadConfigFile(), or poptReadDefaultConfig() can return this error.

POPT_ERROR_BADNUMBER

A conversion from a string to a number (int or long) failed due to the string containing nonnumeric characters. This occurs when poptGetNextOpt() is processing an argument of type POPT_ARG_INT or POPT_ARG_LONG.

POPT_ERROR_OVERFLOW

A string-to-number conversion failed because the number was too large or too small. Like POPT_ERROR_BADNUMBER, this error can occur only when poptGetNextOpt() is processing an argument of type POPT_ARG_INT or POPT_ARG_LONG.

`POPT_ERROR_ERRNO`
: A system call returned with an error, and `errno` still contains the error from the system call. Both `poptReadConfigFile()` and `poptReadDefaultConfig()` can return this error.

Two functions are available to make it easy for applications to provide good error messages.

`const char * poptStrerror(const int error);`
: This function takes a popt error code and returns a string describing the error, just as with the standard `strerror()` function.

`char * poptBadOption(poptContext con, int flags);`
: If an error occurred during `poptGetNextOpt()`, this function returns the option that caused the error. If the `flags` argument is set to `POPT_BADOPTION_NOALIAS`, the outermost option is returned. Otherwise, `flags` should be 0, and the option that is returned may have been specified through an alias.

These two functions make popt error handling trivial for most applications. When an error is detected from most of the functions, an error message is printed along with the error string from `poptStrerror()`. When an error occurs during argument parsing, code similiar to the following displays a useful error message.

```
fprintf(stderr, "%s: %s\n",
        poptBadOption(optCon, POPT_BADOPTION_NOALIAS),
        poptStrerror(rc));
```

24.3 Option Aliasing

One of the primary benefits of using popt over `getopt()` is the ability to use option aliasing. This lets the user specify options that popt expands into other options when they are specified. If the standard grep program made use of popt, users could add a `--text` option that expanded to `-i -n -E -2` to let them more easily find information in text files.

24.3.1 Specifying Aliases

Aliases are normally specified in two places: /etc/popt and the .popt file in the user's home directory (found through the `HOME` environment variable). Both files have the same format, an arbitrary number of lines formatted like this:

```
appname alias newoption expansion
```

The *appname* is the name of the application, which must be the same as the `name` parameter passed to `poptGetContext()`. This allows each file to specify aliases for multiple programs. The `alias` keyword specifies that an alias is being defined; currently popt configuration files support only aliases, but other abilities may be added in the future. The next option is the option that should be aliased, and it may be either a short or a long option. The rest of the line specifies the expansion for the alias. It is parsed similarly to a shell command, which allows `\`, `"`, and `'` to be used for quoting. If a backslash is the final character on a line, the next line in the file is assumed to be a logical continuation of the line containing the backslash, just as in shell.

The following entry would add a `--text` option to the `grep` command, as suggested at the beginning of this section.

```
grep alias --text -i -n -E -2
```

24.3.2 Enabling Aliases

An application must enable alias expansion for a `poptContext` before calling `poptGetNextArg()` for the first time. There are three functions that define aliases for a context.

`int poptReadDefaultConfig(poptContext con, int flags);`

This function reads aliases from /etc/popt and the .popt file in the user's home directory. Currently, `flags` should be `NULL`, as it is provided only for future expansion.

`int poptReadConfigFile(poptContext con, char * fn);`

The file specified by `fn` is opened and parsed as a popt configuration file. This allows programs to use program-specific configuration files.

`int poptAddAlias(poptContext con, struct poptAlias alias, int flags);`

Occasionally, processes want to specify aliases without having to read them from a configuration file. This function adds a new alias to a context. The `flags` argument should be 0, as it is currently reserved for future expansion. The new alias is specified as a `struct poptAlias`, which is defined as:

```
struct poptAlias {
    char * longName;           /* may be NULL */
    char shortName;            /* may be '\0' */
    int argc;
    char ** argv;              /* must be free()able */
};
```

The first two elements, `longName` and `shortName`, specify the option that is aliased. The final two, `argc` and `argv`, define the expansion to use when the aliases option is encountered.

24.4 Parsing Argument Strings

Although popt is usually used for parsing arguments already divided into an `argv`-style array, some programs need to parse strings that are formatted identically to command lines. To facilitate this, popt provides a function that parses a string into an array of string, using rules similiar to normal shell parsing.

```
#include <popt.h>

int poptParseArgvString(char * s, int * argcPtr, char *** argvPtr);
```

The string `s` is parsed into an `argv`-style array. The integer pointed to by the second parameter, `argcPtr`, contains the number of elements parsed, and the pointer pointed to by the final parameter is set to point to the newly created array. The array is dynamically allocated and should be `free()`ed when the application is finished with it.

The `argvPtr` created by `poptParseArgvString()` is suitable to pass directly to `poptGetContext()`.

24.5 Handling Extra Arguments

Some applications implement the equivalent of option aliasing but need to do so through special logic. The `poptStuffArgs()` function allows an application to insert new arguments into the current `poptContext`.

```
#include <popt.h>

int poptStuffArgs(poptContext con, char ** argv);
```

The passed `argv` must have a `NULL` pointer as its final element. When `poptGetNextOpt()` is next called, the "stuffed" arguments are the first to be parsed. popt returns to the normal arguments once all the stuffed arguments have been exhausted.

24.6 Sample Application

Robin, the sample application on pages 274–281 of Chapter 15, uses popt for its argument parsing. It provides a good example of how the popt library is generally used.

RPM, a popular Linux package management program, makes heavy use of popt's features. Many of its command-line arguments are implemented through popt aliases, which makes RPM an excellent example of how to take advantage of the popt library. For more information on RPM, see http://www.rpm.org