Step-by-Step into Argp

by Ben Asselstine

Preamble

This section describes the licensing of this book and it's associated parts.

This book is licensed under the terms of the GNU Free Documentation License version 1.3[1] or any later version. The source code for the various examples in this book are licensed under the GNU General Public License version 3 or at your option any later version[2]. These licenses encourage redistribution of this book; so feel free to share this book with a friend.

1 http://www.gnu.org/copyleft/fdl.html
2 http://www.gnu.org/licenses/gpl-3.0-standalone.html

Introduction

Welcome to your step-by-step journey for learning Argp, the GNU System's own command-line parsing facility. This book covers all aspects of Argp; from parsing simple command-line options to mixing sets of options from libraries into your own programs. The pace of this book is deliberately relaxed and focuses on incremental learning by example for beginner programmers.

Argp (pronounced Argh-Pee) was created in 1995 by Miles Bader for the GNU project[3]. Later on it was merged into the GNU C Standard Library; the special library that all C programs on the computer use by default. From the start Argp has shown itself to be an easy way to incorporate command-line processing into programs, such that the programs *look* and *behave* in a standard way. Argp is also available on non-GNU systems like FreeBSD, Microsoft Windows as an external library.

This book focuses on making simple command-line programs that use Argp for processing command-line options and arguments. The programs are text-based and run within a shell (sometimes called a terminal). These programs fall into a genre called CLI (command-line interface), or TUI (text user interface), or KUI (keyboard user interface). The GNU system has thousands of programs that fall into this genre, and this book will help you create your own command-line based programs.

This book is for programmers who already know a little about the C language and want to make more complicated command-line programs that look and behave and a standard way.

Let's get started and experience the joy of command-line processing with Argp.

3 Argp was created for the GNU Hurd, the GNU System's own kernel.

Conventions in this Book

This section describes common features and terms in this book.

The term “GNU System” refers to the popular GNU+Linux systems available from vendors such as Red Hat, and Ubuntu. The GNU project started in 1984 to create a completely free operating system.

The term “option” is often used a short form for “command-line option”.

When C code is displayed in the book, it takes the following form:

```
#include <stdio.h>
int
main (int argc, char **argv)
{
  printf (“hello world\n”);
}
```

All code examples are indented according to the GNU standard[4]. They appear as if they were indented with the `indent` program.

Commands must be entered in the shell are depicted like this:

```
$ make step1
```

When the names of commands or functions, or command-line fragments appear in the middle of sentences, they appear like `this` and `that`.

4 http://www.gnu.org/prep/standards/standards.html

What you will need

This section describes what you will need to follow the steps in this book.

The following programs are required:

1. gcc
2. make
3. binutils (ar, ranlib)
4. GNU Readline library

An editor will also be needed to write your programs in. You are encouraged to use GNU Emacs if you are familiar with it. Otherwise another text-based editor is required, like `vim` or `nano`.

Additionally a web-browser is required to download the example programs from: http://svn.sv.nongnu.org/viewvc/trunk/?root=argpbook. Cut and pasting the example programs from a PDF can be annoying because indentation is not retained. It is hoped that these example programs will be a convenience to the reader.

Command-line Options

This section explains what command-line options are and how they look and behave on a GNU System.

Command-line options are a convenient way of instructing a program to act in a particular way. Some systems call command-line options *switches*, because they make your program switch behaviours. Like a train on a railroad track, a railroad switch makes the train go in a new direction. Command-line options are created to make your program go in new directions.

Programs have different options because they can do different things, yet *the options are the same* in important ways. Command-line options have a standard way of looking and behaving.

There are two kinds of options; long ones and short ones. Argp supports both kinds.

Short options have a hyphen followed by a letter. For example: "`-f`" as seen in the `rm` command, or "`-c`" as seen in the `wc` command. Short options can be *ganged* together into a set. Consequently, "`ls -la`" is equivalent to "`ls -l -a`".

Long options have two hyphens followed by a word with no spaces. For example: "`--foo`". Long options can be very descriptive as seen in this `ls` long option: "`--dereference-command-line-symlink-to-dir`". Long options cannot be ganged together, and they often have a corresponding short option equivalent.

You are not required to type the whole long option name to make the program understand it: you only need to type enough of the long option name to uniquely identify it among other long options in the same program. To illustrate this feature try running "`ls --hel`" instead of "`ls --help`".

To make long options even easier to use, some shells (like the Bash shell) have extensions that allow the tab-completion of long-options.

Users expect options like `--help` and `--version` to be available in your

program. Argp provides a `--help` option to your program by default.

Command-line options are always optional to the operation of a program – this is why they are called *options*. A mandatory option is an oxymoron.

A real-life example of `--help` can be found by running the "`sum`" program with `--help` as an option.

Text 1: A program's customary --help display.

```
$ sum --help
Usage: sum [OPTION]... [FILE]...
Print checksum and block counts for each FILE.

  -r              use BSD sum algorithm, use 1K blocks
  -s, --sysv      use System V sum algorithm, use 512 bytes blocks
      --help     display this help and exit
      --version  output version information and exit

With no FILE, or when FILE is -, read standard input.

Report sum bugs to bug-coreutils@gnu.org
GNU coreutils home page: <http://www.gnu.org/software/coreutils/>
General help using GNU software: <http://www.gnu.org/gethelp/>
Report sum translation bugs to <http://translationproject.org/team/>
```

This is the common look and feel of the `--help` option. When users see this kind of help they feel comfortable with it because they've seen the common layout in so many other programs. The help format always starts off with a Usage line, followed by a short description of what the program does, followed by short descriptions of what each of the program's options do.

The square brackets and the dot-dot-dots might be new to you: they are inspired by BNF. The square brackets mean that the term is optional, and the dot-dot-dot means that the term is repeatable. The usage line shows us that the `sum` program takes many options or none at all, and also many files, or no files at all.

The program offers the standard `--help`, and `--version` options as well as

other options that are particular to the program's behaviour. We can see that `-s` is a short option that has a long option equivalent in `--sysv`. There is also some extra information shown for obtaining help on the `sum` program on the web.

Command-line options come in many shapes and sizes, and later on in this book you will be shown how Argp can help you handle them so that your program looks and behaves in the standard way with respect to command-line options.

Command-line Arguments

This section explains what command-line arguments are.

Most command-line programs take arguments as well as options. These arguments can be filenames, as seen in the "`cp`" command.

Text 2: Providing an option and two arguments to the cp command.

```
$ cp -v foo bar
`foo'->`bar'
$
```

In other programs command-line arguments can be usernames, or IP addresses, or numbers, or anything else you can think of. Arguments can be mandatory, like in the "`cp`" command, or they can be entirely optional, like in the "`ls`" command. If a program doesn't get the correct number of command-line arguments it will refuse to operate. Later on we'll see how Argp lets you (the programmer) collect the arguments given to your program.

Sometimes an argument can look *exactly* like an option. For example, maybe you have a directory called "`--foo`" and you want to remove it. Running "`rmdir --foo`" will cause the `rmdir` program to report that it doesn't have a `--foo` option. Because the argument is indistinguishable from the form of a long option we need to give the program an extra hint that we're providing it an argument and not an option. To provide that hint we use "`--`" to signify that there aren't any more options on this command-line. Here is an illustration of how to use the feature:

Text 3: Providing arguments that are indistinguishable from options.

```
$ mkdir --foo
mkdir: unrecognized option '--foo'
Try `mkdir --help' for more information.
$ mkdir -- --foo
$ ls
--foo
$ rmdir --foo
rmdir: unrecognized option '--foo'
Try `rmdir --help' for more information.
$ rmdir -- --foo
```

The use of "`--`" is the standard way to handling arguments that are

indistinguishable from long or short options. Argp provides this functionality to your program by default.

Later on in this book, you will see how to process command-line arguments in your Argp-enabled program.

Arguments to Options

This section describes command-line options that have their own arguments.

Like programs, options can also take arguments. Arguments that belong to options can be as varied as those belonging to a program. An option can have a mandatory argument, an optional argument, or no argument at all. This section is about options that have mandatory or optional arguments.

A practical example of an option that has an argument is in the "`fold`" command. Let's take a look at what the fold program's `--help` shows:

Text 4: Help that depicts an option that has a mandatory argument.

```
$ fold  --help
Usage: fold [OPTION]... [FILE]...
Wrap input lines in each FILE (standard input by default), writing to
standard output.

Mandatory arguments to long options are mandatory for short options too.
  -b, --bytes         count bytes rather than columns
  -s, --spaces        break at spaces
  -w, --width=WIDTH   use WIDTH columns instead of 80
      --help     display this help and exit
      --version  output version information and exit

Report fold bugs to bug-coreutils@gnu.org
GNU coreutils home page: <http://www.gnu.org/software/coreutils/>
General help using GNU software: <http://www.gnu.org/gethelp/>
Report fold translation bugs to <http://translationproject.org/team/>
```

Here we see the `--width` option that requires a number as an argument. It is an error to provide the `--width` option without an argument, and the program will fail to run if you do so. You can provide an argument to the `--width` option in many different ways:

Text 5: Correct ways of providing mandatory arguments to options.

```
$ echo "hello there" | fold -w3
hel
lo
the
re
$ echo "hello there" | fold -w 4
hell
o th
ere
$ echo "hello there" | fold --width 5
hello
 ther
e
$ echo "hello there" | fold --width=6
hello
there
```

There are a few invalid ways to provide a mandatory argument to an option. For example: these ways do not work:

Text 6: Incorrect ways of providing mandatory arguments to options.

```
$ fold --width5
fold: unrecognized option '--width5'
Try `fold --help' for more information.
$ fold -w=5
fold: invalid number of columns: `=5'
```

The short option with an optional argument is a noteworthy case. They are difficult to find in the wild; but one exists in the `-j` option of GNU Make:

Text 7: A short option with an optional argument.

```
  -j [N], --jobs[=N]          Allow N jobs at once; infinite jobs with no arg.
  -k, --keep-going            Keep going when some targets can't be made.
```

The noteworthy aspect of this `-j` option (and all other short options with optional arguments) is that it can be ganged together with `-k` but not when `-k` follows `-j`! Consequently "`make -kj`" works fine but "`make -jk`" does not because "`k`" is specified as the number of jobs, which is an invalid number of jobs. In an earlier section we showed how an argument can be mistaken for an

option, and we had to use "`--`" as a hint to fix the problem. In this is case we have a ganged short option being mistaken for an option's argument. There is no hint we can give the program to detect this situation, other than not ganging the `-j` and `-k` together. Unlike short options, long options with optional arguments require an equal sign separating the option name from the argument value.

Later on in this book we'll find out how Argp lets you make and handle options that have both mandatory or optional arguments, and how it does some useful error-checking by default.

Step Zero: Your First Argp Program

This section will show you how to write, compile and run the simplest Argp program.

Let's get started. Cut and paste the following C code into a file called `step0.c`:

Text 8: step0.c the simplest Argp-enabled program.

```
#include <stdio.h>
#include <argp.h>

int
main (int argc, char **argv)
{
 return argp_parse (0, argc, argv, 0, 0, 0);
}
```

The Argp facility uses the `argp_parse` function to initiate parsing. For now we pass in `argc`, and `argv` so that Argp can know about the incoming command-line. The other parameters enable other functionality and will be covered in later sections. We include the "`argp.h`" file so that our program knows about all things argp-ish.

Compile the program by typing the following into a shell:

Text 9: Compiling the step0 program.

```
$ make step0
cc     step0.c   -o step0
```

Now you can run your first Argp-enabled program. Let's take a look at what your new program can do!

Text 10: Running the step0 program to see the help.

```
$ ./step0 --help
Usage: step0 [OPTION...]

  -?, --help                 Give this help list
      --usage                Give a short usage message
```

Fantastic! With very little effort our `step0` program has that standard look and feel of other command-line programs. We get a `--help` option for free, and also another option called `--usage`. Let's try running the program with that `--usage` option:

Text 11: Running the step0 program to see the usage.

```
$ ./step0 --usage
Usage: step0 [-?] [--help] [--usage]
```

The usage option shows a very brief display of help which is useful for a quick reminder of what options the program has. Let's see what else we can do with our new program:

Text 12: Having fun with the step0 program.

```
$ ./step0 foo
step0: Too many arguments
Try `step0 --help' or `step0 --usage' for more information.
$ ./step0 --us
Usage: step0 [-?] [--help] [--usage]
$ ./step0 --foo
./step0: unrecognized option '--foo'
Try `step0 --help' or `step0 --usage' for more information.
$ ./step0 --usage --help
Usage: step0 [-?] [--help] [--usage]
$ ./step0 --
$
```

This example demonstrates the following things about Argp:

1. It does some automatic error reporting for our program. By default it disallows any arguments at all.
2. We can type "`--us`" instead of "`--usage`" and it gives the same effect.
3. It does some automatic error reporting for options that are not supported by our program.
4. It exits after parsing `--usage` (and `--help`).
5. It honours the "`--`" convention of denoting the end of options on a command-line.

It's a useless program, but it does a lot more than you'd expect because it's Argp-enabled. Why is this program called `step0`? C Programmers often have to start counting at zero, so the convention was adopted for the various steps (programs) in this book. In the next step we'll add our own option to the

program.

Step One

In this section the most frequently used elements of the Argp syntax are demonstrated. We cover the argp struct, the argp_option struct, and the parser function.

Let's add a `-d` option to our program that will show a period on the screen. Cut and paste the following C code into your editor and save it as `step1.c`. The highlighted regions represent new code from the previous step.

Text 13: step1.c : an Argp-enabled program with one option.

```
#include <stdio.h>
#include <argp.h>

static int
parse_opt (int key, char *arg,
           struct argp_state *state)
{
  switch (key)
    {
    case 'd': printf (".\n"); break;
    }
  return 0;
}

int
main (int argc, char **argv)
{
  struct argp_option options[] =
    {
      { 0, 'd', 0, 0, "Show a dot on the screen"},
      { 0 }
    };
  struct argp argp = { options, parse_opt };
  return argp_parse (&argp, argc, argv, 0, 0, 0);
}
```

The most important data type in Argp is the `struct argp` data type. Here we can see that it contains:

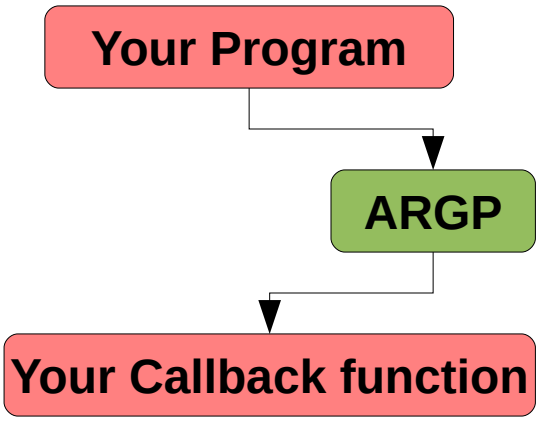
1. All of our options (just one option for now),
2. A pointer to a function that Argp will call on our behalf to assist in parsing our `-d` option (`parse_opt`).

And then we pass a pointer to our `struct argp` as the first parameter to the `argp_parse` function, to let Argp know about our shiny new `-d` option.

Our options array contains a single entry, followed by an empty terminating record. The record that describes our `-d` option has a lot of zeroes on it, (each zero meaning the field is unused,) and for now we only use the 2nd field, and 5th fields. The `'d'` means it's a "`-d`" short option we're creating, and the 5th field contains the description of the option to be shown in the `--help`. The other fields of `struct argp_option` do other things and will be revealed in later sections.

The `parse_opt` function can be a source of consternation for beginner programmers because there is no visible call to it. The function seems to hang out in space unconnected to any other function. We can see the `parse_opt` function name being given to the `struct argp`, but it's not visibly being called anywhere in the program. That's because Argp is calling our `parse_opt` function on our behalf as it goes about processing the `argc` and `argv` that we pass to `argp_parse`. If you were to look at the argp sources (and you can if you need to see the connection), you would see that it calls `parse_opt` repeatedly, over and over until it's done processing the command line. Argp is not just calling `parse_opt` for our `-d` option, it is calling `parse_opt` for every option and argument on the command line.

Drawing 1: Your program calls argp, which calls your callback function on your behalf.



Callback functions are a frequently used mechanism in C programs. Our `parse_opt` function is a *callback function* for Argp. Argp decides what the signature will be for it's callback function, and as programmers we diligently implement a function with that signature. In this case the `parse_opt` function signature has three parameters:

1. The option's unique key, which can be the short option '`d`'. Other non-character keys exist and we'll demonstrate them in later sections.
2. The value of the option's argument, as a string. Our option doesn't take an argument so we ignore this parameter in the callback function.
3. A variable that keeps Argp's state as we repeatedly iterate the callback function. Although it is not used in this program, it will be used in later programs.

And finally the callback function returns zero when everything is okay, and non-zero will stop Argp and cause the `argp_parse` function to return that non-zero error value.

When the function name is assigned to the `struct argp`, it is actually assigning the pointer to the where the function exists in memory. This is similar to how the name of an array is a pointer to the array's first element.

Our `parse_opt` function checks the key to see if Argp is currently processing the "`-d`" option or not, and when it finds that it is, it simply displays a period on the screen, followed by a newline. If `-d` is specified many times on the command-line, the callback function will be called same number of times with '`d`' as the value of the key parameter.

Let's compile the program:

Text 14: Compiling the step1 program.

```
$ make step1
cc     step1.c   -o step1
```

And now run it in the shell to see the help:

Text 15: Running the step1 program to see the help.

```
$ ./step1 --help
Usage: step1 [OPTION...]

  -d                         Show a dot on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message
$
```

But does our program work? Let's give it a spin with the `-d` option.

Text 16: Having fun with the step1 program.

```
$ ./step1 -d
.
$ ./step1 -ddd -d
.
.
.
.
$
```

Hooray! It works.

Real-life Example: Even though this is a very simple program it is far from useless. The short option without an argument appears in a myriad of programs on your system! Look at the “-f” option in the “rm” command. Or look at the “-l” option to the “ls” command. It's everywhere, and now you know the basics of how to make your own program offer a similar option with Argp.

In the next step we'll add a mandatory argument to the `-d` option.

Step Two

In this section the argument member of the argp_option struct will be described, and the arg parameter to the callback function will be demonstrated.

Let's add an argument to our `-d` option, such that it can produce many dots instead of just one. Cut and paste the following C code into your editor and save a file called `step2.c`. The highlighted regions are the parts of the code that have changed from the previous step.

Text 17: step2.c : an Argp-enabled program with an option that has a mandatory argument.

```
#include <stdio.h>
#include <argp.h>

static int
parse_opt (int key, char *arg,
           struct argp_state *state)
{
  switch (key)
    {
    case 'd':
        {
          unsigned int i;
          for (i = 0; i < atoi (arg); i++)
            printf (".");
          printf ("\n");
          break;
        }
    }
  return 0;
}

int
main (int argc, char **argv)
{
  struct argp_option options[] =
    {
        { 0, 'd', "NUM", 0, "Show some dots on the screen"},
        { 0 }
    };
  struct argp argp = { options, parse_opt, 0, 0 };
  return argp_parse (&argp, argc, argv, 0, 0, 0);
}
```

We are now using the 3rd field in the `struct argp_option`. It denotes that there is an argument to this `-d` option, and that the argument shall be called "`NUM`" in the `--help` and `--usage` displays. If we were to change this value back to 0

(or NULL), the `-d` option would cease to accept a mandatory argument.

In the `parse_opt` callback function we are using the `arg` parameter. It's coming to us (the programmer) as a string, and the value of `arg` is guaranteed not to be NULL because Argp won't let `-d` be called without an argument. A better program would have more validity checks on the contents of `arg` (a malicious user could give random text to our `-d` option), but for illustrative purposes the validity checking will be omitted.

The `arg` variable points to memory that we didn't allocate, and trying to free it would be a very bad idea. You can expect the memory that the `arg` value points to remain accessible for the duration of the program's execution. The memory that the `arg` parameter points to won't go away after the callback function completes. In our case we only need to access the value of `arg` while executing the callback function.

Let's take our new program out for a spin. Compile it, display the program's help, and try out the new and improved `-d` option:

Text 18: Compiling step2.c and take it for a spin.

```
$ make step2
cc     step2.c   -o step2
$ ./step2 --help
Usage: step2 [OPTION...]

  -d NUM                     Show some dots on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message
$ ./step2 -d1 -d2 -d 3
.
..
...
$ ./step2 -d
./step2: option requires an argument -- 'd'
Try `step2 --help' or `step2 --usage' for more information.
$ ./step2  --usage
Usage: step2 [-?] [-d NUM] [--help] [--usage]
$
```

We can see that Argp is doing some error-checking for us, and that it is automatically building the help and usage displays to include the new

argument to `-d`.

Real-life example: The mandatory argument to a short option frequently appears in many programs on your system. Look at the “-n” option to the “head” program (the option takes the number of topmost lines to show in a file.) We have already seen the “-w” option to the “fold” program. Argp makes it trivial to add short options with mandatory arguments to your program.

In the next step we will make our program accept a long option.

Step Three

In this section the 1st field of the argp_option struct is used to make the program accept a long option.

Let's make our `-d` option have a long option equivalent called `--dot`. Cut and paste the following C code into your editor and save a file called `step3.c`. The highlighted regions are the parts of the code that have changed from the previous step.

Text 19: step3.c : an Argp-enabled program with a long option that has a mandatory argument.

```
#include <stdio.h>
#include <argp.h>

static int
parse_opt (int key, char *arg,
           struct argp_state *state)
{
  switch (key)
    {
    case 'd':
        {
          unsigned int i;
          for (i = 0; i < atoi (arg); i++)
            printf (".");
          printf ("\n");
          break;
        }
    }
  return 0;
}

int
main (int argc, char **argv)
{
  struct argp_option options[] =
    {
        { "dot", 'd', "NUM", 0, "Show some dots on the screen"},
        { 0 }
    };
  struct argp argp = { options, parse_opt, 0, 0 };
  return argp_parse (&argp, argc, argv, 0, 0, 0);
}
```

With this one small change, our program now accepts the `--dot` long option. The 1st field of the `struct argp_option` controls the name of the long option.

Bad things will happen if you put spaces (or newlines, or tabs, or non-printables) in your option names, so don't do that.

Let's try it out:

Text 20: Compiling and running the step3 program.

```
$ make step3
cc     step3.c   -o step3
$ ./step3 --help
Usage: step3 [OPTION...]

  -d, --dot=NUM              Show some dots on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message

Mandatory or optional arguments to long options are also mandatory or
optional
for any corresponding short options.
$ ./step3 --dot 1 --dot=2 -d3
.
..
...
$ ./step3  --dot3
./step3: unrecognized option '--dot3'
Try `step3 --help' or `step3 --usage' for more information.
$ ./step3 --dot
./step3: option '--dot' requires an argument
Try `step3 --help' or `step3 --usage' for more information.
$./step3 --do 12
............
$ ./step3 --usage
Usage: step3 [-?] [-d NUM] [--dot=NUM] [--help] [--usage]
$
```

Here we can see that the help display now includes the `--dot` option, and it is handled in the standard way of long options. We can see that Argp provides some long option error checking for free (without the programmer having to do anything). We can also see that our program accepts the `--do` long option. This is a demonstration of how users can take shortcuts with long options. The `--do` option works because there are no other long options that start with "`do`". If there were two options that started with "`do`", and we tried to use a `--do` shortcut, the program would refuse to run and show you an error about the ambiguous option name. This all happens in Argp, you don't have to do

anything extra to have this functionality in your program.

There is a rather wordy notice at the bottom of the help display. What the notice means is that "`NUM`" is also an argument to the short `-d` option. Argp put the notice there because the help formatted in this way they can give the impression to some people that `-d` doesn't take an argument. Experienced users find the wordy notice to be redundant and you can *turn it off* by adding the text "`no-dup-args-note`" to the `ARGP_HELP_FMT` environment variable in your shell:

Text 21: Turning off the long options/short options argument notice in the bash shell.

```
$ export ARGP_HELP_FMT="no-dup-args-note"
$ ./step3 --help
Usage: step3 [OPTION...]

  -d, --dot=NUM              Show some dots on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message
```

Although it can be nice to disable this notice for experienced users, this notice is still useful to new users. That's why Argp has it turned on by default, and has this notice translated into a myriad of different spoken languages.

Real-life example: The --lines option in the "head" program is a long option equivalent to the -n short option. It is easier understand the program by reading the list of long options, and they are easier to remember. And to top it off, long options are more intuitive to use. Now you too can make your own descriptive long options that have short option equivalents.

In the next step we'll be making our program accept a long option with an optional argument.

Step Four

In this section we'll be using the 4th field of the argp_option struct to make an option accept an optional argument.

Let's make the `NUM` argument to our `--dot` option be optional. Cut and paste the following C code into your editor and save a file called `step4.c`. The highlighted regions are the parts of the code that have changed from the previous step.

Text 22: step4.c : an Argp-enabled program with a long option that has an optional argument.

```
#include <stdio.h>
#include <argp.h>

static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  switch (key)
    {
    case 'd':
       {
         unsigned int i;
         unsigned int dots = 0;
         if (arg == NULL)
           dots = 1;
         else
           dots = atoi (arg);
         for (i = 0; i < dots; i++)
           printf (".");
         printf ("\n");
         break;
       }
    }
  return 0;
}

int
main (int argc, char **argv)
{
  struct argp_option options[] =
    {
       { "dot", 'd', "NUM", OPTION_ARG_OPTIONAL, "Show some dots on the screen"},
       { 0 }
    };
  struct argp argp = { options, parse_opt, 0, 0 };
  return argp_parse (&argp, argc, argv, 0, 0, 0);
}
```

The 4th field of the `struct argp_option` is a set of flags that change how our option works. The `OPTION_ARG_OPTIONAL` flag is just one of many flags that this bitwise field accepts. The other flags will be revealed in later sections.

The `arg` parameter to our callback function `parse_opt` can now come to us as NULL which means that the `-d` or `--dot` option was specified on the command-line without an argument. When arg comes to us as non-NULL it points to the argument that was given just like in the previous step.

Let's take the `step4` program for a spin:

Text 23: Comping and running the step4 program.

```
$ make step4
cc     step4.c   -o step4
$ ./step4 --help
Usage: step4 [OPTION...]

  -d, --dot[=NUM]            Show some dots on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message
$ ./step4 --usage
Usage: step4 [-?] [-d[NUM]] [--dot[=NUM]] [--help] [--usage]
$ ./step4 -d --dot=3 --dot
.
...
.
$ ./step4 --dot 3
.
$ ./step4 -dd

$
```

Here we can see that the help and usage displays now show an optional argument to our `--dot` option. We can also see the program can accept arguments to the `--dot` option and that it also functions well without an argument given to it. The final two cases don't run as you might expect. Can you figure out why?

1. In the first case, it appears that the argument of "`3`" is not recognized, and only one dot gets displayed. Why did this happen? Long options with optional arguments require an `=` sign between the option and the argument value. The "`3`" ends up being an argument to the program, but Argp doesn't complain about that case because we're expected to handle arguments if we have a callback function. Arguments to programs will be handled in a later section.

2. In the second case, it appears that short option was successfully ganged together with another short option. After all, no error message was given. Short options with optional arguments cannot be ganged together. The second "`d`" is actually the argument given to the `-d` option. The call to "`atoi`" converts "`d`" to `0`, which causes the routine to display no dots at all, and then a newline.

Real-life Example: Try having a look at the "--backup[=CONTROL]" option of the "cp" program if you want to see a long option with an optional argument. It's easy to make your options accept an optional argument.

Exercise: Can you write a program that uses strtoul instead of atoi to check the arg string for validity?

In the next step option aliases will be demonstrated.

Step Five

In this section another option flag for the 4th field of the argp_option struct will be demonstrated: the OPTION_ALIAS flag.

Here's the `step5` program. You know what to do!

Text 24: step5.c : an Argp-enabled program with an aliased long option.

```
#include <stdio.h>
#include <argp.h>

static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  switch (key)
    {
    case 'd':
       {
         unsigned int i;
         unsigned int dots = 0;
         if (arg == NULL)
           dots = 1;
         else
           dots = atoi (arg);
         for (i = 0; i < dots; i++)
           printf (".");
         printf ("\n");
         break;
       }
    }
  return 0;
}

int
main (int argc, char **argv)
{
  struct argp_option options[] =
    {
       { "dot", 'd', "NUM", OPTION_ARG_OPTIONAL, "Show some dots on the screen"},
       { "period", 0, "FOO", OPTION_ALIAS, "Bar" },
       { 0 }
    };
  struct argp argp = { options, parse_opt, 0, 0 };
  return argp_parse (&argp, argc, argv, 0, 0, 0);
}
```

We've added a single line to our program, and it will give us a long option `--period` that is equivalent to `--dot`. The `OPTION_ALIAS` flag causes the option to inherit all fields from the previous option except for the long option name (the 1st field), and the key (the 2nd field). You can have as many aliases as you like. Some of the other fields have been filled out with dummy values to

illustrate that they are in fact ignored by Argp.

Let's see how our new `--period` option works.

Text 25: Compiling and running the step5 program.

```
$ make step5
cc     step5.c   -o step5
$ ./step5 --help
Usage: step5 [OPTION...]

  -d, --dot[=NUM], --period[=NUM]
                             Show some dots on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message
$ ./step5 --usage
Usage: step5 [-?] [-d[NUM]] [--dot[=NUM]] [--period[=FOO]] [--help]
            [--usage]
$ ./step5 --period
.
$ ./step5 --period=4
....
```

We can see that the "`FOO`" and "`Bar`" strings are correctly ignored and do not appear in the help display. The new `--period` long option appears beside the `--dot` option because it is completely equivalent to it. Argp does the grunt work of arranging the help and usage displays to accommodate our new option alias so that you don't have to.

Real-life example: The "rm" program has an option alias with the -R short option. The -r short option and the --recursive long option are equivalents to it. This illustrates that sometimes it is useful for short options to be aliases. In this case the -R option is probably provided for compatibility purposes.

In the next step we're going to add a new long option that makes the callback function call the callback function.

Step Six

In this section we're going to call the callback function from within the callback function so that we can implement a brand new option.

Let's make a new long option `--ellipsis` that displays three dots on the screen. It is functionally equivalent to `--dot=3`, and we will call parse_opt from within `parse_opt` to implement it.

Text 26: step6.c : an Argp-enabled program that calls it's own callback function.

```
#include <stdio.h>
#include <argp.h>

static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  switch (key)
    {
    case 'd':
       {
         unsigned int i;
         for (i = 0; i < atoi (arg); i++)
           printf (".");
         printf ("\n");
         break;
       }
    case 777:
      return parse_opt ('d', "3", state);
    }
  return 0;
}

int
main (int argc, char **argv)
{
  struct argp_option options[] =
    {
       { "dot", 'd', "NUM", 0, "Show some dots on the screen"},
       { "ellipsis", 777, 0, 0, "Show an ellipsis on the screen"},
       { 0 }
    };
  struct argp argp = { options, parse_opt, 0, 0 };
  return argp_parse (&argp, argc, argv, 0, 0, 0);
}
```

For the sake of simplicity, the `--dot` option has been changed to have a mandatory argument. The highlighted portions are the regions of the code that implement the new `--ellipsis` option.

We can see that our new `--ellipsis` option doesn't take an argument, and that it has a very strange key of "`777`". The 2nd field of the `struct argp_option`

(the key field) is special because Argp automatically detects if it is a viewable character to be used in a short option or not. `777` is not a printable character, and this means the `--ellipsis` option will be a long option, with no short option equivalent.

We can also see that the `parse_opt` function is being called from inside itself. This is the easiest way to execute other options. Because we want our ellipsis option to show three dots on the screen, we call `parse_opt` with the `'d'` key to activate the `--dot` option, and we pass `3` as a string as the `--dot` option's argument. We also pass the `state` variable, just because it's a required parameter to the callback function.

Let's run the program and see how it runs.

Text 27: Compiling and running the step6 program.

```
$ make step6
cc     step6.c   -o step6
$ ./step6 --help
Usage: step6 [OPTION...]

  -d, --dot=NUM              Show some dots on the screen
      --ellipsis             Show an ellipsis on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message
$ ./step6 --usage
Usage: step6 [-?] [-d NUM] [--dot=NUM] [--ellipsis] [--help] [--usage]
$ ./step6 --ellipsis
...
$ ./step6 --dot 3
...
$
```

We were unable to make this new `--ellipsis` option an alias because it does not have an argument. Instead we made a new option and had it execute our old `--dot` option with an argument of `3`. The help and usage displays show the new option, and we can see that the new option works correctly. We can also see that the ellipsis option has no short option equivalent.

Real-life example: This sort of “option equivalence” is fairly frequently used in command-line programs. If you look at the --archive option in the “cp” program, you will see that it is equivalent to the options: “-dR –preserve-all”. Now you

know how easy it is to make option equivalents in your own programs!

In the next step we will handle program arguments.

Step Seven

In this section we will demonstrate the ARGP_KEY_ARG and ARGP_KEY_END keys in the callback function, and also we will illustrate the use of the 6th parameter to argp_parse, the user-supplied data hook for the callback function.

Let's add support for one to four arguments to our program. The idea here is that the program will report an error if it doesn't get the right number of arguments.

Text 28: step7.c : An Argp-enabled program that accepts arguments. Callback funcion.

```
#include <stdio.h>
#include <argp.h>

static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  int *arg_count = state->input;
  switch (key)
    {
    case 'd':
        {
          unsigned int i;
          for (i = 0; i < atoi (arg); i++)
            printf (".");
          break;
        }
    case 777:
      return parse_opt ('d', "3", state);
    case ARGP_KEY_ARG:
        {
          (*arg_count)--;
          if (*arg_count >= 0)
            printf (" %s", arg);
        }
      break;
    case ARGP_KEY_END:
        {
          printf ("\n");
          if (*arg_count >= 4)
            argp_failure (state, 1, 0, "too few arguments");
          else if (*arg_count < 0)
            argp_failure (state, 1, 0, "too many arguments");
        }
      break;
    }
  return 0;
}
```

Here we can see that we're using two special `ARGP_KEY` keys. Argp passes in `ARGP_KEY_ARG` whenever it encounters an argument to the program, and it sets the `arg` parameter to point to that argument. When Argp has processed the

last argument it passes `ARGP_KEY_END` to our callback function.

Argp provides an error-reporting facility called `argp_failure` that is to be used when there is a problem parsing the command-line. When we use this facility our error messages have the standard look and feel.

The `state` parameter is shown to have an `input` field, and we're using it in a dual-purpose way:

1. to know how many arguments is the correct number of arguments.
2. to know how many arguments we've gotten on the command-line.

Argp also keeps track of how many arguments we've processed so far, in the `struct argp_state`. There is a member called "`arg_num`" that reports precisely this information.

The input field of the state parameter is a value that is passed to `argp_parse`. Let's have a look at the `main` function so we can see that this happening:

Text 29: step7.c : An Argp-enabled program that accepts arguments. Main function.

```
int
main (int argc, char **argv)
{
  struct argp_option options[] =
    {
      { "dot", 'd', "NUM", 0, "Show some dots on the screen"},
      { "ellipsis", 777, 0, 0, "Show an ellipsis on the screen"},
      { 0 }
    };
  int arg_count = 4;
  struct argp argp = { options, parse_opt, "WORD [WORD [WORD [WORD]]]" };
  return argp_parse (&argp, argc, argv, 0, 0, &arg_count);
}
```

And here we can see that we are indeed passing in a pointer to an int as the 6th parameter of the `argp_parse` function, and the value it points to is 4. When callback functions are employed in a C program, there is almost always a way to pass your own piece of data into the callback function. The 6th parameter of the `argp_parse` function is a way for you to pass any data you want into your callback function. For command-line processing with Argp this value is frequently a struct that contains flags that can be set by options.

We can also see that the `struct argp` has more fields that we thought. The 3rd

field of the `struct argp` holds the notation of what arguments we're expecting on this command-line. In this case we're expecting exactly 4 arguments, so we say so. For kicks we're going to call them “words” because it fits in with a sentence construction motif for the program.

Let's take the program for a spin!

Text 30: Compiling and running the step7 program.

```
$ make step7
cc     step7.c   -o step7
$ ./step7 --help
Usage: step7 [OPTION...] WORD [WORD [WORD [WORD]]]

  -d, --dot=NUM              Show some dots on the screen
      --ellipsis             Show an ellipsis on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message
$ ./step7 --usage
Usage: step7 [-?] [-d NUM] [--dot=NUM] [--ellipsis] [--help] [--usage]
            WORD [WORD [WORD [WORD]]]
$ ./step7 once upon a time
 once upon a time
$ ./step7

step7: too few arguments
$ ./step7 foo bar baz
 foo bar baz
$ ./step7 once upon a time --ellipsis
... once upon a time
$ ./step7 one two three four five
 one two three four
step7: too many arguments
$
```

First of all we can see that our argument notation has been incorporated into the help and usage displays, and it shows the user can she can give one, two, three, or four words to this program, but not none or more than four words.

When we run it with exactly four arguments, the words get printed out just as we would expect.

When we run it with no arguments, we get a newline and *then* an error message. Can you see why this is the case by looking at the callback

function?

Exercise: How would you change the callback function so that a newline only gets displayed when there are the correct number arguments? Can you remove the space before the first word that the program displays?

When we run the program with an `--ellispsis` long option at the end of the command-line, the "..." appears at the start of the sentence instead of at the end. This is normal Argp behaviour: options are always parsed before arguments. If this is unacceptable you can make Argp to honour the order of your command-line options by passing `ARGP_IN_ORDER` into the bitwise flags parameter (the 4th parameter) of the `argp_parse` function.

Lastly, we can see that our program has another bug: when we pass in five arguments to the program it still displays the first four words before displaying an error message. In a future step we will fix this bug.

Real-life example: Sometimes it is useful to accept a bounded number of arguments on the command-line. Try having a look at the "ln" program. You can see that it takes either one or two arguments just like our program takes one, two, three or four arguments. Now you can see how to make an Argp-enabled program emit an error message when it doesn't get the correct number of command-line arguments.

In the next step we'll make a program that refuses to accept any command-line arguments at all, and explore the concept of hidden options.

Step Eight

In this section we will make a program that shows an error message when the command-line has any arguments, and also we'll demonstrate another option flag, the OPTION_HIDDEN flag that makes options invisible.

Let's change the motif of our program from sentence construction, to a morse code motif. This will shrink our program but we will add a `--dash` option, and we will modify the `--ellispis` long option to be functional, but invisible in the help and usage displays. The program will not accept any arguments at all, just like our step0 program behaved.

Text 31: step8.c : An Argp-enabled program that accepts a hidden option. Callback funcion.

```
#include <stdio.h>
#include <argp.h>

static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  switch (key)
    {
    case 'd':
       {
         unsigned int i;
         unsigned int dots = 1;
         if (arg != NULL)
           dots = atoi (arg);
         for (i = 0; i < dots; i++)
           printf (".");
         break;
       }
      break;
    case 888:
      printf ("-");
      break;
    case 777:
      return parse_opt ('d', "3", state);
    case ARGP_KEY_ARG:
      argp_failure (state, 1, 0, "too many arguments");
      break;
    case ARGP_KEY_END:
      printf("\n");
      break;
    }
  return 0;
}
```

There are no new facilities shown in the callback function. We can see that we've modified the `--dot` option to take an optional argument, and we can see that we're stopping the program when we get an argument. Let's have a look

at the `main` function:

Text 32: step8.c : An Argp-enabled program that accepts a hidden option. Main function.

```
int
main (int argc, char **argv)
{
  struct argp_option options[] =
    {
      { "dot", 'd', "NUM", OPTION_ARG_OPTIONAL, "Show some dots on the screen"},
      { "ellipsis", 777, 0, OPTION_HIDDEN, "Show an ellipsis on the screen"},
      { "dash", 888, 0, 0, "Show a dash on the screen" },
      { 0 }
    };
  struct argp argp = { options, parse_opt };
  return argp_parse (&argp, argc, argv, 0, 0, 0);
}
```

We can see that we've added a new long option called `--dash`, and that we've given the `--dot` option an optional argument. We can also see that we're no longer passing in a data hook to the `argp_parse` function. The new thing shown here is the `OPTION_HIDDEN` flag on the `--ellipsis` option. This will make the option continue to operate, but it will not be shown in the help or usage displays.

Hidden options can be useful if they're not overused. One good aspect is that a hidden option doesn't clutter up the `--help` display, and it can be fully documented in the program's manual. Sometimes it is more important for the program to appear simple to use, than to show five option aliases that do the same thing. Perhaps a particular hidden-option is used only for ancient compatibility purposes. We want to accept the option so that compatibility is ensured, but we don't really want to advertise it.

One bad aspect of hidden options is that they can interfere with long option shortcuts. Let's say we had a hidden option named `--ellipsis` and a visible one called `--elation` (that displays a smiley face), the user would not be able to type "`--el`" to activate the `--elation` long option. The hidden `--ellipsis` option would confuse matters (Argp wouldn't know which option the user was referring to), even though the user can't readily see the option in the help or usage displays.

Hidden options require a balance of utility and information management

versus ease of use. Let's see the new hidden option in action.

Text 33: Compiling and running the step8 program.

```
$ make step8
cc     step8.c   -o step8
$ ./step8  --help
Usage: step8 [OPTION...]

  -d, --dot[=NUM]            Show some dots on the screen
      --dash                 Show a dash on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message
$ ./step8  --usage
Usage: step8 [-?] [-d[NUM]] [--dot[=NUM]] [--dash] [--help] [--usage]
$ ./step8  --ellipsis
...
$ ./step8  -d -d --dot --dash --dash --dash
...---
$ ./step8 --dash --
-
$ ./step8 --d
./step8: option '--d' is ambiguous
Try `step8 --help' or `step8 --usage' for more information.
$ ./step8 dot
step8: too many arguments
$
```

Just as expected, the `--ellipsis` option isn't shown in the help and usage displays, yet it functions as a valid option.

The case of the ambiguous option shortcut is also demonstrated. The program refuses to operate because Argp is unable to determine if "`--d`" means `--dash` or `--dot`.

Real-life example: Some programs take no program arguments at all. Look no further than the "tty" program for a real-life example of this. Also some programs have invisible options that are only documented in their manuals. Try looking at the "gdb" command, and it's "-c" option; it is equivalent to --core but it is not shown in the gdb's help display.

Now you know how to make your own hidden options with Argp! Go ahead and add an easter egg to your next program. In the next step we will fill out our help display with a version option and an email address for bugs.

Step Nine

In this section the argp_program_version and argp_program_bug_address global variables will be demonstrated. In addition to that, the 4th field of the argp struct (the docs field) will be used to add a description of the program to the help display. A new argp key in the callback function will be demonstrated: the ARGP_KEY_INIT key.

Let's put the finishing touches on our program. We will give it a standard `--version` option, and we'll tell people what the program does, and where to send their bug reports. We will also add a second usage to this program to illustrate alternative an way to run this program. To make this program more practical we will process the command-line arguments *after* we're finished parsing them.

Text 34: step9.c : An Argp-enabled program with a --version option and a bug address. Preamble.

```
#include <stdio.h>
#include <argp.h>
#include <argz.h>
#include <stdlib.h>

const char *argp_program_bug_address = "someone@example.com";
const char *argp_program_version = "version 1.0";

struct arguments
{
  char *argz;
  size_t argz_len;
};
```

Argp has a few global variables in it's API, and here are two of them. If the `argp_program_version` variable is set, then a `--version` long option, and a `-V` short option will be included in our program, and it will display whatever string we set the variable to, and then exit. If the `argp_program_bug_address` variable is set, it will modify the help display to include a message of the form "`Report bugs to: foo@bar`.". It is good style for a program to be able to report what version it is, and where bugs should be sent.

We can also see something called argz. Argz is another facility that originates from the GNU Standard C Library. We will use this facility to accumulate arguments as we encounter them on the command-line. The `struct arguments` will hold our arguments, and we will set it as our input data hook to the `argp_parse` function. It is a good idea to have a `struct arguments` in

your program, where it contains flags that get set when options are encountered, and the arguments as well. The `stdlib.h` file is included because argz vectors are `malloc`'d and we want to use the `free` function.

Text 35: step9.c : A program with a --version option and a bug address. Callback function.

```
static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  struct arguments *a = state->input;
  switch (key)
    {
    case 'd':
       {
         unsigned int i;
         unsigned int dots = 1;
         if (arg != NULL)
           dots = atoi (arg);
         for (i = 0; i < dots; i++)
           printf (".");
         break;
       }
    case 888:
      printf ("-");
      break;
    case ARGP_KEY_ARG:
      argz_add (&a->argz, &a->argz_len, arg);
      break;
    case ARGP_KEY_INIT:
      a->argz = 0;
      a->argz_len = 0;
      break;
    case ARGP_KEY_END:
       {
         size_t count = argz_count (a->argz, a->argz_len);
         if (count > 2)
           argp_failure (state, 1, 0, "too many arguments");
         else if (count < 1)
           argp_failure (state, 1, 0, "too few arguments");
       }
      break;
    }
  return 0;
}
```

The new element in this callback function is the `ARGP_KEY_INIT`. It gets passed into the callback function before any parsing happens. Here we are using it to initialize our `struct arguments`.

When we're finished collecting the arguments (in the `ARGP_KEY_END` case), we check to see if we have the right number of arguments. In this program the right number of arguments is exactly 1 or exactly 2 arguments. For more information on the argz facility, type "`info argz`" into your shell.

Text 36: step9.c : A program with a --version option and a bug address. Main function.

```
int
main (int argc, char **argv)
{
 struct argp_option options[] =
   {
     { "dot", 'd', "NUM", OPTION_ARG_OPTIONAL, "Show some dots on the screen"},
     { "dash", 888, 0, 0, "Show a dash on the screen" },
     { 0 }
   };
 struct argp argp = { options, parse_opt, "WORD\nWORD WORD",
   "Show some dots and dashes on the screen.\v"
     "A final newline is also shown regardless of whether any options were given." };
 struct arguments arguments;
 if (argp_parse (&argp, argc, argv, 0, 0, &arguments) == 0)
   {
     const char *prev = NULL;
     char *word;
     while ((word = argz_next (arguments.argz, arguments.argz_len, prev)))
       {
         printf (" %s", word);
         prev = word;
       }
     printf ("\n");
     free (arguments.argz);
   }
 return 0;
}
```

One new Argp element shown in this main function is the 4th field of the `struct argp`. It has a special "`\v`" character in it, which is a vertical tab. Everything before that vertical tab will be shown *before* the options in the help display, while the remainder of the string will be shown *after* the options. The purpose of this variable is two-fold:

1. To give a short description of the program.
2. To further describe the options or the operation of the program.

Another new Argp element shown in this main function is the newline in the 3rd field of the `struct argp` (the `args_doc` field). This is another way of telling your users that there are other ways of running this program.

Let's try out our new program.

Text 37: Compiling and running the step9 program.

```
$ make step9
cc     step9.c   -o step9
$ ./step9  --help
Usage: step9 [OPTION...] WORD
  or:  step9 [OPTION...] WORD WORD
Show some dots and dashes on the screen.

  -d, --dot[=NUM]            Show some dots on the screen
      --dash                 Show a dash on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

A final newline is also shown regardless of whether any options were given.

Report bugs to someone@example.com.
$ ./step9  --usage
Usage: step9 [-?V] [-d[NUM]] [--dot[=NUM]] [--dash] [--help] [--usage]
             [--version] WORD
  or:  step9 [OPTION...] WORD WORD
$ ./step9 foo bar
 foo bar
$ ./step9 foo bar baz
step9: too many arguments
$ ./step9
step9: too few arguments
$ ./step9 foo --dash -d
-. foo
$ ./step9 --version
version 1.0
$
```

In this demonstration we can see that this help output has two usages. Many commands will have many different usage lines in their help output. For an example look at the "`ln`" command; it has 4 usages. Argp makes it easy to specify alternate usages for your program.

The help output has some explanatory text below the option descriptions. It is a good practice to exercise restraint in making your help output too long. Complicated explanations do not belong in your help output; they belong in a

manual or user's guide.

We can see that the program correctly handles the cases of too many or too few arguments. We can also see that it shows error messages in the argp fashion.

Lastly we can see that the program shows the argument (`foo`) after it shows the options. This is because we're collecting the arguments in an argz vector and displaying them after the command-line is parsed. If we passed `ARGP_IN_ORDER` to the `argp_parse` function, it would have no effect on the output.

There is more than one way to obtain the *version* of a program that you're using. Some people might tell you to use the package manager on your system to see what version of a program you're running. If you are creating a new program, you are obliged to give your program an authoritative version. Some programmers even derive a certain amount of joy from the version numbering scheme they employ. As the program's author, you are *upstream* of the packagers and have the exclusive privilege of deciding on what version number your program will have. The packagers *downstream* (e.g. Red Hat and Ubuntu) from you redistribute your program if and as they see fit. Important programs like `gcc` are so heavily patched by the GNU+Linux distributions that they modify what version gets displayed when the `--version` option is given on the command-line! The good news is that your version is almost always left intact, and the modification comes in the form of a notice that this version of `gcc` was modified and built by Red Hat on a given date.

The `argp_program_version` variable limits the programmer to a simple string. You can use your own function for the version option instead by defining a function that looks like: `void foo(FILE*, struct argp_state *)` and calling it `argp_program_version_hook`.

Exercise: Try changing the step9 program to have an argp_program_version_hook function instead of a argp_program_version variable.

You now know enough to give your program some options and some arguments. It's a good idea to take a break at this point, and examine the

command line options in the programs you use, and see if you can make a CLI that mirrors theirs. The steps in this book will now get more advanced and complex.

In the next step we will use “option groups” to make our help output more readable when there are many options.

Step Ten

In this section we will explore the 6th and final field of `struct argp_option`, the `group` field. We will see how to change the ordering of options in the help output, and also how to arrange sets of options so that they appear as a cluster in the help output.

Let's make our help output a little more readable by putting our `--dash` and `--dot` options into their own group.

Text 38: step10.c : A program with a group of options. Main function.

```
int
main (int argc, char **argv)
{
 struct argp_option options[] =
   {
     { 0, 0, 0, 0, "Morse Code Options:", 7},
     { "dot", 'd', "NUM", OPTION_ARG_OPTIONAL, "Show some dots on the screen"},
     { "dash", 888, 0, 0, "Show a dash on the screen" },
     { 0, 0, 0, 0, "Informational Options:", -1},
     { "SOS", 999, 0, 0, "Give some help in morse code" },
     { 0 }
   };
 struct argp argp = { options, parse_opt, "WORD\nWORD WORD",
   "Show some dots and dashes on the screen.\v"
    "A final newline is also shown regardless of whether any options were given." };
 struct arguments arguments;
 if (argp_parse (&argp, argc, argv, 0, 0, &arguments) == 0)
   {
    const char *prev = NULL;
    char *word;
    while ((word = argz_next (arguments.argz, arguments.argz_len, prev)))
      {
       printf (" %s", word);
       prev = word;
      }
    printf ("\n");
    free (arguments.argz);
   }
 return 0;
}
```

The `group` field is more complicated than it looks. Only two of our `struct argp_option` records have it specified, as 7 and -1 respectively. These two records are called option headers, and it is the normal convention to have their text end in a colon.

The `group` field acts as a primary key to sort the options in the help output. Records that have a larger non-negative value for their group appear after

smaller non-negative groups. For example, groups with a value of 5 appear before groups with a value of 7. Groups that have negative values appear after the non-negative ones. Records that have a smaller negative value for their group appear after larger negative groups. For example groups with a value of -3 appear before groups with a value of -1. In the `argp.h` header file, it succinctly defines the order as: "`0, 1, 2, ..., n, -m, ..., -2, -1`".

The other `struct argp_option` records do not have a `group` value specified. This is because the group value is automatically set for options that occur after a option header. This means that our `--dash` and `--dot` options automatically receive a group value of 7. The normal usage of the group value is to not specify it for non-header options.

Options that have a group value of zero and do not appear after an option header retain their value of zero and appear before all other options in the help output. If we omit the group value in an option header, it is automatically set to have a value of one more than the previous option's group value. The purpose of the automated setting of group values is to make it unnecessary to supply a group value in simple cases.

There is new `--SOS` option declared in our options array, so we need to add a case block in our callback function to implement it:

Text 39: step10.c : A program with a group of options. This is a fragment of the callback function.

```
  case 888:
    printf ("-");
    break;
  case 999:
    parse_opt ('d', "3", state);
    printf (" ");
    parse_opt (888, NULL, state);
    parse_opt (888, NULL, state);
    parse_opt (888, NULL, state);
    printf (" ");
    parse_opt ('d', "3", state);
    printf ("\n");
    exit (0);
    break;
  case ARGP_KEY_ARG:
    argz_add (&a->argz, &a->argz_len, arg);
    break;
```

There isn't anything surprising in this code fragment. We're simply showing the SOS sequence and then exiting.

Let's compile `step10` and have a look at our new help output!

Text 40: Compiling and running the step10 program.

```
$ make step10
cc     step10.c   -o step
$ ./step10 --help
Usage: step10 [OPTION...] WORD
  or:  step10 [OPTION...] WORD WORD
Show some dots and dashes on the screen.

 Morse Code Options:
  -d, --dot[=NUM]            Show some dots on the screen
      --dash                 Show a dash on the screen

 Informational Options:
  -?, --help                 Give this help list
      --SOS                  Give some help in morse code
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or
optional
for any corresponding short options.

A final newline is also shown regardless of whether any options were given.

Report bugs to someone@example.com.
$ ./step10 --SOS
... --- ...
$
```

Here we can see that morse code options are grouped under a heading and followed by an empty line. Many programs have sets of options organized in this manner; for an example try looking at the help output of `automake`.

Although the `--dash` and `--dot` options have the same group value, `--dot` appears before `--dash` because the secondary sort key is the short option, and the tertiary sort key is the long option name.

In a surprising twist, the `--SOS` option is interspersed with other help

options. This interspersing of options happens because the default options also have -1 as their group value.

In the next step we'll put the `--dot` and `--dash` options into a library and instruct our program to use the options in that library.

Step Eleven

In this step the 5th field of `struct argp` (`children`) will be used to incorporate options from a library into our program. The `struct argp_child` will also be demonstrated.

Mixing options from a library into a program is an exciting feature that only Argp provides. Perhaps you have two programs and you want both programs to have some of the same options. Argp makes it easy for many programs to use the same options, without copying code.

It is common for programmers to encapsulate the functionality of a program into a library so that other programmers can use that API to do new things. In many cases the API has some configuration elements to it (even if it is limited to where the configuration file is), and the programmer has to collect that information to pass to the API. Argp lets you provide option parsers for your library so that programmers who use your API don't have to write code that does the same thing. As the writer of a library you've put a lot of thought into how other programmers are going to use your API. When you add an option parser to your library, you're thinking about the user-experience on the command-line associated with your library. You can use GNU Gettext to translate your options into other spoken-languages, so that users of your option parser don't have to do extra translation. Downstream programmers can use the option parser if they wish; and if they choose to use it, they get the standard behaviour, look and feel that you want to give your users on the command-line, and they get it for free.

Argp uses “child parsers” to mix options. To notify Argp that we want to mix options, we provide a set of child parsers to the `struct argp` (the 5th parameter) that we pass to `argp_parse`. A child parser is defined by `struct argp_child`, and it looks like this:

```
struct argp_child
{
  const struct argp *argp;
  int flags;
  const char *header;
  int group;
};
```

The idea here is that we have another “child” `struct argp` that we want to mix

with our main `struct argp`. In our example, the child `struct argp` will be an exported symbol from a library. In an interesting recursive twist, the child `struct argp` can also have it's own set of child parsers.

The `flags` field has the same semantics as the `flags` field of the `argp_parse` function, and it's okay to pass zero for this field in most cases.

The `header` field puts a heading before the child's options in the help output. When a heading is specified as an empty string in the `struct argp_child`, it has the effect of grouping the given options, but it doesn't show a heading or an empty line following the option group. This `header` field does not override the heading that might already be included in the options of the given `struct argp`.

The `group` field acts like the `group` field of `struct argp_option`. It puts these options elsewhere in the help output. The `group` field doesn't override any non-zero `group` fields in the `struct argp_option` array of the given `struct argp`.

There are limits to the mixing of options with Argp. Firstly it is important to avoid key collisions; the first option will shadow the second option that has the identical key. Secondly the child parsers cannot parse arguments on the command line; the `ARGP_KEY_ARG` value simply never gets passed into the parsing functions of child parsers. Argp only allows for options to be mixed, not argument-handling.

To demonstrate the mixing of options, let's put our `--dot` and `--dash` options into a library called libdotdash. Copy and paste the following code and put it in a file called `dotdash.c`:

Text 41: dotdash.c : A library with a group of options.

```
 #include "dotdash.h"

static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  switch (key)
    {
    case 'd':
       {
         unsigned int i;
         unsigned int dots = 1;
         if (arg != NULL)
           dots = atoi (arg);
         for (i = 0; i < dots; i++)
           printf (".");
         break;
       }
    case 888:
      printf ("-");
      break;
    }
  return 0;
}

static struct argp_option options[] =
{
   { "dot", 'd', "NUM", OPTION_ARG_OPTIONAL, "Show some dots on the screen"},
   { "dash", 888, 0, 0, "Show a dash on the screen" },
   { 0 }
};

struct argp
dotdash_argp =
{
  options, parse_opt, 0, 0, 0
};
```

Now let's make the accompanying header file `dotdash.h`:

Text 42: dotdash.h : A library with a group of options. Header file.

```
#ifndef DASHDOT_H
#define DASHDOT_H
#include <argp.h>
extern struct argp dotdash_argp;
#endif
```

Now let's make a program that uses our dotdash library. Cut and paste the

following code and put it in a file called `step11.c`:

Text 43: step11.c : A program that mixes options from a library.

```
#include <stdio.h>
#include <argp.h>
#include "dotdash.h"

static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  switch (key)
    {
    case 999:
      printf ("... --- ...");
      break;
    }
  return 0;
}

int
main (int argc, char **argv)
{
  struct argp_option options[] =
    {
      { "SOS", 999, 0, 0, "Show the SOS sequence on the screen" },
      { 0 }
    };
  struct argp_child children_parsers[] =
    {
      { &dotdash_argp, 0, "Basic Morse Code Options:", 7 },
      { 0 }
    };
  struct argp argp = { options, parse_opt, 0, 0, children_parsers };
  int retval = argp_parse (&argp, argc, argv, 0, 0, 0);
  printf ("\n");
  return retval;
}
```

Now let's compile the library and link it to our program:

Text 44: Compiling the dotdash library and linking it to the step11 program.

```
$ cc -c -o dotdash.o dotdash.c
$ ar cru libdotdash.a dotdash.o
$ ranlib libdotdash.a
$ cc step11.c -L./ -ldotdash -o step11
```

Now let's take a look at the help output that the program generates:

Text 45: Running the step11 program.

```
$ ./step11  --help
Usage: step11 [OPTION...]

      --SOS                  Show the SOS sequence on the screen

 Basic Morse Code Options:
  -d, --dot[=NUM]            Show some dots on the screen
      --dash                 Show a dash on the screen

  -?, --help                 Give this help list
      --usage                Give a short usage message

Mandatory or optional arguments to long options are also mandatory or
optional
for any corresponding short options.
$ ./step11 --SOS --dot=10
... --- .............
```

Yay! The options from the dotdash library are mixed-in with our own `--SOS` option. Now any Argp-enabled program can provide the `--dot` and `--dash` options by linking to this library, and incorporating the dotdash argp child parser.

Exercise: Try changing the `header` field of `struct argp_child` to NULL. What happens? Now try changing it to "". Can you spot the difference?

Step Twelve

In this step we will make a larger program that demonstrates the passing back of option input data from child parsers by using the `child_inputs` field of `struct argp_state`. We will also show how to parse options in a string instead of a command-line. And lastly the `help_filter` field of `struct argp` (the 6th field) will also be demonstrated.

Let's make a program that converts morse code to human-readable text. The program will have an interactive mode where you can enter in strings to be converted to morse code, and it will also have a non-interactive mode where the string is given on the command-line as argument.

The program will use the GNU Readline library to implement an interactive mode. There will be two options that control the history of interactive mode: `--no-history`, and `--history-file`. These options will be provided in a library called `libreadline-argp` and mixed into our program.

The interactive mode will accept two commands: "`quit`" (which acts like you'd expect), and "`tap`" which takes the `--dot` and `--dash` options from libdotdash. If it gets a string that isn't a command, it converts it to morse code.

First let's look at the options that control the command history. There are no new concepts introduced in the `readline-argp.c` file:

Text 46: readline-argp.c : A library that provides two readline related Argp options.

```
#include "readline-argp.h"
#include <unistd.h>

char *default_history_file = ".history";

static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  struct readline_arguments *args = state->input;
  switch (key)
    {
    case 411:
      if (access (arg, R_OK))
        args->history_file = arg;
      else if (access (arg, W_OK))
        args->history_file = arg;
      else
        argp_failure (state, 1, 0, "Cannot open file `%s' for reading", arg);
      break;
    case 511:
      args->history_file = NULL;
      break;
    case ARGP_KEY_INIT:
      args->history_file = default_history_file;
      break;
    }
  return 0;
}

static struct argp_option options[] =
{
    { "history-file", 411, "FILE", 0,
      "Specify a FILE to store the interactive history"},
    { "no-history", 511, 0, 0, "Do not record a history in interactive mode"},
    { 0 }
};

struct argp
readline_argp =
{
  options, parse_opt, 0, 0, 0
};
```

The `struct readline_arguments` is defined in the header along with `readline_argp`:

Text 47: readline-argp.h : A library that provides two readline related Argp options. Header file.

```
#ifndef READLINE_ARGP_H
#define READLINE_ARGP_H
#include <argp.h>
extern struct argp readline_argp;
extern char * default_history_file;

struct readline_arguments
{
  char *history_file;
};
#endif
```

The idea behind the `default_history_file` variable is that users of the library can set the default to whatever they want. Now let's have a look at the option parsing of the program:

Text 48: morse-tool.c : A program that converts morse code. Preamble.

```
#include <stdio.h>
#include <argp.h>
#include <argz.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>
#include "readline-argp.h"
#include "dotdash.h"

const char *argp_program_bug_address = "samuel@morse.net";
const char *argp_program_version = "version 1.0";

struct arguments
{
  int capitalize;
  char *argz;
  size_t argz_len;
  struct readline_arguments readline;
};

static struct argp_option options[] =
{
   { "capitalize", 'c', 0, 0,
     "Show morse translations in capital letters" },
   { 0 }
};
static struct argp_child children_parsers[] =
{
   { &readline_argp, 0, 0, 0 },
   { 0 }
};
```

Here we make space for the option data coming from our new readline argp

library.

Here is what the main parsing function looks like:

Text 49: morse-tool.c : A program that converts morse code. Parsing function setting child_inputs.

```
static int
parse_opt (int key, char *arg, struct argp_state *state)
{
  struct arguments *a = (struct arguments*) state->input;

  switch (key)
    {
    case 'c':
      a->capitalize = 1;
      break;
    case ARGP_KEY_ARG:
      argz_add (&a->argz, &a->argz_len, arg);
      break;
    case ARGP_KEY_INIT:
      a->argz = NULL;
      a->argz_len = 0;
      a->capitalize = 0;
      state->child_inputs[0] = &a->readline;
      break;
    }
  return 0;
}
```

Here we see how the `struct readline_arguments` gets passed into the readline argp parser. The subscript of `[0]` is used because the `struct readline_argp` is the first member of the set of child parsers (the `children_parsers` variable).

Let's have a look at the next section of the file that makes our help display a little more informative:

Text 50: morse-tool.c : A program that converts morse code. Help filter.

```
static char *
help_filter (int key, const char *text, void *input)
{
  if (key == ARGP_KEY_HELP_POST_DOC)
    {
      char *new_text = NULL;
      if (asprintf (&new_text, text, default_history_file) != -1)
        return new_text;
    }
  return (char *) text;
}

struct argp argp = { options, parse_opt, "[PHRASE]",
  "Translates to and from morse code.\vThis program starts in interactive mode when PHRASE is not
supplied on the command line.  The history for interactive mode is stored in a file called `%s' by default.",
  children_parsers, help_filter };
```

The `help_filter` field in `struct argp` points to a function that changes the appearance of the program's help output. The function will be called by Argp on every option in our array of `struct argp_option`. It is not called back for options in child parsers. The idea behind the help filter is that we can change the appearance of a particular option's help output with information at run-time instead of compile-time. The callback function gives us the `key` of the option we're dealing with, the original option help `text` (which is the `doc` field of `struct argp_option`), and the `input` parameter that we gave to `argp_parse`. The function either returns the text that was passed in (meaning no change), or a `malloc`'d string containing the new text we want the option to have. Argp will `free` the newly `malloc`'d string later on.

The `key` parameter to the function refers to the `key` field of `struct argp_option`. If the value isn't a key of an option, it can be one of the following values:

- `ARGP_KEY_HELP_PRE_DOC` : Changes the `doc` field of `struct argp`, everything before the vertical tab (`'\v'`).
- `ARGP_KEY_HELP_POST_DOC` : Changes the doc field of struct argp, everything after the vertical tab.
- `ARGP_KEY_HELP_HEADER` : Changes a header option (there can be many).
- `ARGP_KEY_HELP_EXTRA` : A way to add help text to the bottom of the help output.
- `ARGP_KEY_HELP_DUP_ARGS_NOTE` : Changes the “mandatory arguments for

long options are also mandatory for short options" message.
- `ARGP_KEY_HELP_ARGS_DOC` : Changes the `args_doc` field of `struct argp`.

Our `help_filter` callback function simply changes the `POST_DOC` section to insert the default filename for the history file. To make this easy, we put a '`%s`' into the `doc` field of `struct argp`, and then we use the `asprintf` function to fill in the value and make a new string. The `asprintf` function is another GNU specific function that appears in the GNU C Standard Library.

Let's take a look at the next section of the file, the part that implements the interactive mode of our program:

Text 51: morse-tool.c : A program that converts morse code. Parsing options in a string.

```
static void
interactive_mode (char *prompt, int capitalize)
{
  char *line;
  char *argz = NULL;
  size_t argz_len = 0;
  while ((line = readline (prompt)))
    {
      if (strncmp (line, "tap ", 4) == 0 || strcmp (line, “tap”) == 0)
        {
          /* parse the tap command with the dotdash options. */
          if (argz_create_sep (line, ' ', &argz, &argz_len) == 0)
            {
              int flags = ARGP_NO_EXIT | ARGP_NO_ARGS;
              int argc = argz_count (argz, argz_len);
              char *argv[argc + 1];
              argz_extract (argz, argz_len, argv);
              argv[argc] = 0;
              if (argp_parse (&dotdash_argp, argc, argv, flags, 0, 0) == 0)
                {
                  add_history (line);
                  printf ("\n");
                }
            }
        }
      else if (strcmp (line, "quit") == 0)
        break;
      else
        {
          if (argz_create_sep (line, ' ', &argz, &argz_len) == 0)
            {
              if (morse_process_line (argz, argz_len, capitalize) == 0)
                add_history (line);
            }
        }
    }
  return;
}
```

Here we get a string from the `readline` function and operate on it. Readline is a great library to link to if you need to ask the user for a string because it can provide a history and the allows the user to edit the string in the familiar way with the arrow keys along with delete and backspace. If the command is legal we store it to the history file.

As we planned, there are two commands, “`quit`” and “`tap`”. We want the tap command to have the dotdash options. We know from experience that `argp_parse` takes an argument vector (e.g. an argv and an argc), and argz

facility help us in crafting it. We want our tap command to have some help output, and we take special care to make it not exit our program with the `ARGP_NO_EXIT` flag. If the user gives us something that isn't a command, we break up the line into words and try to convert them to or from morse code.

The 7th and final field in struct argp is the `argp_domain` field. This is a string that will be used to identify a GNU Gettext domain that contains the various translations for hardcoded strings in Argp. This doesn't affect the strings you put in `struct argp` or `struct argp_option`, it only affects the text that originate in the Argp routines and it is not necessary to set this value to anything except 0.

We have now seen all of the fields in `struct argp`. The struct looks like this:

```
struct argp
{
  struct argp_option *options;
  error_t (*parser) (int key, char *arg, struct argp *state);
  char *args_doc;
  char *doc;
  struct argp_child *children;
  char *(*help_filter) (int key, char *text, void *input);
  char *argp_domain;
};[5]
```

For more information about how the letters are converted into morse code, see the `process_morse_line` function inside `morse.c` at: http://svn.sv.nongnu.org/viewvc/trunk/morse.c?root=argpbook.

To finish off the `morse-tool.c` file, it ends with a `main` function that does two things: firstly it kicks off the processing of the command-line for the program, and secondly it kicks off the interactive mode or batch mode of the program. Let's take a look at the `main` function of our program:

5 Const declarations are not shown for readability.

Text 52: morse-tool.c : A program that converts morse code. Parsing options in a string.

```
int
main (int argc, char **argv)
{
 struct arguments arguments;
 default_history_file = ".morse-tool.history";
 int retval = argp_parse (&argp, argc, argv, 0, 0, &arguments);
 if (retval != 0)
  return retval;

 /* silence --version and bug address in the tap command */
 argp_program_version = 0;
 argp_program_bug_address = 0;

 if (arguments.argz_len > 0)
  {
   morse_process_line (arguments.argz, arguments.argz_len,
                       arguments.capitalize);
   free (arguments.argz);
  }
 else if (arguments.argz_len == 0)
  {
   if (arguments.readline.history_file)
    read_history (arguments.readline.history_file);
   interactive_mode ("morse> ", arguments.capitalize);
   if (arguments.readline.history_file)
    write_history (arguments.readline.history_file);
  }

 return retval;
}
```

Notice how the `--version` option is suppressed *after* we run `argp_parse`, so that the `quit` and `tap` commands are less cluttered and a little bit simpler.

Okay let's compile this morse-tool program!

Text 53: Compiling the libreadline-argp library and linking it to the morse-tool program.

```
$ cc -c -o readline-argp.o readline-argp.c
$ ar cru libreadline-argp.a readline-argp.o
$ ranlib libreadline-argp.a
$ cc morse-tool.c morse.c -L./ -ldotdash -lreadline-argp -lreadline -o morse-
tool
```

The libdotdash library was built in the previous step. Let's see what the help, version and usage displays are:

Text 54: Running the morse-tool program.

```
$ ./morse-tool --help
Usage: morse-tool [OPTION...] [PHRASE]
Translates to and from morse code.

  -c, --capitalize           Show morse translations in capital letters
      --history-file=FILE    Specify a FILE to store the interactive history
      --no-history           Do not record a history in interactive mode
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

This program starts in interactive mode when PHRASE is not supplied on the
command line.  The history for interactive mode is stored in a file called
`.morse-tool.history' by default.

Report bugs to samuel@morse.net.
$ ./morse-tool --version
version 1.0
$ ./morse-tool --usage
Usage: morse-tool [-c?V] [--capitalize] [--history-file=FILE] [--no-history]
            [--help] [--usage] [--version] [PHRASE]
```

Here we can see three things:

1. That the options from libreadline-argp have been mixed-in with `morse-tool`'s `--capitalize` option.
2. That the help display shows the contents of our `default_history_file` variable.
3. That the program takes a phrase on the command line or no phrase at all.

Let's try out non-interactive mode:

Text 55: Running the morse-tool program in non-interactive mode.

```
$ ./morse-tool hello world
.... . .-.. .-.. ---  .-- --- .-. .-.. -..
$ ./morse-tool -c -- .... . .-.. .-.. ---  .-- --- .-. .-.. -..
HELLOWORLD
```

Yay it works! Now let's try out interactive mode:

Text 56: Running the morse-tool program in interactive mode.

```
$ ./morse-tool
morse> hello world
.... . .-.. .-.. ---  .-- --- .-. .-.. -..
morse> .... . .-.. .-.. ---  .-- --- .-. .-.. -..
helloworld
morse> tap --help
Usage: tap [OPTION...]

  -d, --dot[=NUM]            Show some dots on the screen
      --dash                 Show a dash on the screen
  -?, --help                 Give this help list
      --usage                Give a short usage message

Mandatory or optional arguments to long options are also mandatory or
optional
for any corresponding short options.

morse> tap --dash --dash --dash
---
morse> ---
o
morse> quit
$ cat .morse-tool.history
hello world
.... . .-.. .-.. ---  .-- --- .-. .-.. -..
tap --help
tap --dash --dash --dash
---
```

One thing you'll notice is how polished the program feels when interacting with it. This is mostly because of the functionality that libreadline provides.

If we ever wanted to translate “quit” or “tap” we can do so in non-interactive mode.

It is difficult to demonstrate libreadline doing what it does, so the history contents of the file is displayed to show that libreadline is really doing something. The next time the program is started in interactive mode, the user can press the up arrow key to see the last command given (e.g. “---”).

Exercise: Make a hidden option called --prompt to change the prompt form

“morse>” to something else. Also prevent the “arguments to short options are same options for long options” message from appearing in the tap command's help display.

This `morse-tool` program shows how easy it is to make interactive programs that take various commands with options. In the previous steps Argp parsed command-lines given to our program. Now in this step we're using Argp so that it operates on our program's data. Consider programs like `bc`, and `ftp` and how libreadline and argp could be used to implement them.

Concluding Remarks

This section describes what we've covered, what some of the limits are, and how you can learn more about Argp.

Congratulations on completing this step-by-step journey into Argp! We've conveyed a lot information! We've covered the basics of the command-line: from what long and short options are, to what arguments are and what option arguments are, and how they all fit together. We've covered the basics of command-line processing. We've seen how to collect mandatory and optional arguments on the command-line, and how to parse long and short options in an Argp callback function, some of which have mandatory or optional arguments. We've demonstrated how to do error reporting from within an Argp callback function. We've shown how to make option aliases, hidden options and option equivalents. We've also shown how to mix Argp parsers together into a single program so that the options are interspersed into the existing options.

We've seen how to modify the help display in various ways. We've seen how to put options into groups so that they appear together in the help display. We've demonstrated how to display regular and alternative usages to the program, how to set the description of what the program does, and how to control the extra information at the bottom of the help display.

After programming with Argp for a while you can sense some of it's limitations. It is impossible to make an argp parser by `malloc`'ing the `struct`

`argp_option` array and so on; the data elements must all be pre-allocated (on the stack) and `const`. Mixing options together gives precedence to the options mixed first, so we can't override a same-named option that a programmer has given us in a `struct argp`. Arguments are not processed in child parsers.

For more information, the GNU libc manual has a section on Argp[6]. Also you can use a service like Google Code Search to search for programs that use Argp[7]. Lastly you can look inside `/usr/include/argp.h` on your system; much of it will look familiar to you, and the file is well-commented and intended for programmers to read.

Recreational programming can be a lot of fun, and now you can have fun incorporating Argp into your programs! Your Argp-enabled programs will be more powerful and easier to use. Happy hacking!

6 http://www.gnu.org/s/libc/manual/html_node/Argp.html
7 http://www.google.com/codesearch?q=include+%3Cargp.h%3E&hl=en&btnG=Search+Code