Lab 11: Command Line Processing

This lab will introduce you to the facilties POSIX (and GNU) provide for command line processing and environment variable queries. These are the standard ways to pass parameters to a program.

The distinction between "parameter" and "datum" is a fine one. Technically, all parameters are (input) data, but one way to make a distinction might be to say that parameters are distinct, single data values (or a small list of them) that have a significant effect on how a program operates. They may be of any type, including (frequently) Booleans. Parameters can be specified as command line options. They may also be environment variables. Data, on the other hand, usually comes in large quantities and is often stored in files. The *name* of a file may be an option, but its contents are data.

The distinction between "option" and "argument" is also worth mentioning. Not surprisingly, options are (usually) optional: programs do something even if you don't provide any options. Arguments are lower-level: A command line option will require one or more arguments (the first one beginning with "`-`") to implement. Also, arguments may not be options:

In:

```
cc -g -o foo foo.c
```

"`-g`" is an option and "`-o foo`" is an option, but "`foo.c`" is an argument, not an option.

Many programs use options to give the user flexibility in just what the program does. Providing them on the command line rather than, say, prompting the user for them allows the programs to be incorporated into scripts. The result can be very powerful. Virtually all POSIX shell commands provide them.

This page will be available as

> http://www.tricity.wsu.edu/~bobl/cpts360/lab11_getopt/writeup.html

Calling it up in a browser will allow you to cut-and-paste code in the following and save typing. You may also download whole files from the lab link on the course web page.

We'll start with a program that plays a simple but fun game and add command line options and environment variables to permit the user to change the length and/or difficulty of the game.

Every part after Part 1 depends on the part that went right before it, so be sure that part is working before you move on.

Part 1: The `lander` Program

A classic computer game called "lunar lander" simulates this:

> You're at the controls of a probe landing on the surface of the Moon. The probe starts out at a given height above the surface travelling downward at a given velocity with a given amount of fuel. At (simulated) one second intervals, you determine how much fuel to burn (from 0 to a given maximum) to slow the probe's fall. If you land at less than 2 meters per second, you land safely. Otherwise, you crash. If you run out of fuel, the probe will free-fall and (probably) crash.

For this part, of the lab:

- Download `lander_pt1.c` via the lab link. Originally, the lander landed on Earth's Moon, but this version has been modified to land on Titan, the largest moon of Saturn.
- Write a `Makefile` to compile `lander_pt1.c` to `lander_pt1`.

- Do so and run the program a couple of times to get a feel for the game. (This will help with testing.) Don't spend too much time on it. You can play the game as much as you like after you complete and submit the rest of the lab.

Part 2: Creating a `parameters` Structure and Using Environment Variables

In this part, we'll modify `lander` to use a `struct` to hold just one parameter (for now) and use environment variables to set it. Here are the steps:

- Copy `lander_pt1.c` to a file `lander_pt2.c` and make the following changes to that.
- Include `stdlib.h` to get the prototypes of `strtod()` and `getenv()`, which you'll use later.
- Create this global `struct` after all the `#include`s:

  ```
  struct {
      double maxThrust;
  } parameters;
  ```

  It's couth to put all globals (including `static` globals) into a single `struct` like this. With `gdb` you can print out all globals with a single `print` command (here, "`p parameters`").
- In `getThrust()`, replace instances of `MAX_THRUST_DEFAULT` with `parameters.maxThrust`.
- Add this code to the file just before `main()`:

  ```
  double setDefaultDouble(char *envName, double dflt)
  /*
   * If there's an environment variable named `envName`, convert it to a
   * double and return it. Otherwise, return the default value `dflt`.
   */
  {
      char *env = getenv(envName);

      return ( env ? strtod(env, NULL) : dflt );
  }
  ```

  You may also download it as `set_default_double.c` from the lab link.

  This is a handy little function to deal with environment variables and defaults. Read the comments.
- In `main()`, enter:

  ```
  parameters.maxThrust = setDefaultDouble("MAX_THRUST", MAX_THRUST_DEFAULT);
  ```
- Enhance the `Makefile` to compile `lander_pt2.c` to `lander_pt2` as well as continue to build `lander_pt1`.
- Do so and test using the new `MAX_THRUST` environment variable. People who play the game regularly could set this variable to change the default.

Part 3: Calling `getopt(3)`

In this part, we'll use `getopt(3)` to process command line arguments in a convenient and flexible way.

- Copy `lander_pt2.c` to a file `lander_pt3.c` and make the following changes.
- Include `getopt.h` and `assert.h` to get `getopt()` and `assert()`, which you'll use later.
- In `main()`:
    - Add a declaration of an `int` variable `optionLetter`.
    - Add the following loop right after the `setDefaultDouble()` call:

```
while ((optionLetter = getopt(argc, argv, "t:")) != -1) {
    switch (optionLetter) {

    case 't':
        parameters.maxThrust = strtod(optarg, NULL);
        break;

    default:
        assert(0); // just in case
    }
}
```

- Enhance the `Makefile` to compile `lander_pt3.c` to `lander_pt3` as well as continue to build `lander_pt1` and `lander_pt2`.
- Do so and test setting the maximum thrust with the "`-t`" command line option.

Part 4: Adding More Options

Following the pattern set in Parts 2 & 3, we'll add a few more parameters to give the user more control and flexibility. We'll make use of this table:

environment variable	`#define` default	`parameter` member	(short) option
`INITIAL_FUEL`	`INITIAL_FUEL_DEFAULT`	`initialFuel`	`-f`
`GRAVITY`	`GRAVITY_DEFAULT`	`gravity`	`-g`
`INITIAL_HEIGHT`	`INITIAL_HEIGHT_DEFAULT`	`initialHeight`	`-h`
`MAX_THRUST`	`MAX_THRUST_DEFAULT`	`maxThrust`	`-t`
`INITIAL_VELOCITY`	`INITIAL_VELOCITY_DEFAULT`	`initialVelocity`	`-v`

Perform these steps:

- Copy `lander_pt3.c` to a file `lander_pt4.c` and make the following changes.
- Replace all references to the `*_DEFAULT #defines` in the second column of the table with corresponding references to the `parameter` member in the third column. (You've already done `MAX_THRUST`. It's just in the table for reference.)

- In `main()`:
  - Add `setDefaultDouble()` calls to set the members of `parameters` as we did for `parameters.maxThrust`. Use the environment variable names in the the first column of the table.
  - Modify the third argument of the `getopt()` call in `main()` to allow for the options in the fourth column of the table. Note that each of them takes an argument which `optarg` will be set to. `optarg` is a global `char *` that is declared in `getopt.h`, so you don't need to. As with `"t:"`, each option letter needs to be followed by a colon to indicate the use of `optarg`.
  - Add `case`s to the `switch` to set the appropriate `options` member. Use `strtod(3)` to convert `optarg` to a `double`.
- Enhance the `Makefile` to compile `lander_pt4.c` to `lander_pt4`. as well as continue to build `lander_pt1`, `lander_pt2`, and `lander_pt3`.
- Do so and test setting the various parameters using both command line options and environment variables.

Part 5: Calling `getopt_long(3)`

Long (or "keyword") options are especially useful in scripts, when you might not remember what the short option did when you look at the script six months later. In this part, we'll add long option versions to all the short options so that:

```
$ lander_pt5 -f200
```

means the same thing as:

```
$ lander_pt5 --initial-fuel=200
```

Here are the steps:

- Copy `lander_pt4.c` to a file `lander_pt5.c` and make the following changes.
- In `main()`, referring to the `getopt_long(3)` man page:
  - Create an array of `struct options` (this `struct` is defined in `getopt(3)` to add these long option names and map them to these short options:

long option	short option
`gravity`	`-g`
`initial-fuel`	`-f`
`initial-height`	`-h`
`initial-velocity`	`-v`
`max-thrust`	`-t`

  - Change the `getopt()` call to a `getopt_long()` call.
- Enhance the `Makefile` to build `lander_pt5.c` to `lander_pt5`. Be sure that it can continue to build `lander_pt1`, `lander_pt2`, `lander_pt3`, and `lander_pt4` as before.
- Do the make and test setting the various options using long command line options.

- Submit a tarball with all the `lander_pt*.c` files and `Makefile` and via Canvas. Include `set_default_double.c` if you have it as a separate file. (So "`$ make`" should compile everything.)