COMP 40 Laboratory: Getting started with C and Interfaces with bonus technique *Make it compile; make it run; make it right*

September 9, 2011

For this lab, you will want:

1. The course handout *How to Write Compile Scripts*
2. A copy of the first homework assignment
3. A C reference, either online or in a book
4. A copy of the `Pnmrdr` interface, which goes with the first assignment (either online or printed out)
5. Access to the pnm documentation (try `man pgm`)
6. Access to the course web page on idioms for C programmers

The goal of the lab is to get you started with the new material you will need to learn to complete the homework.[1] We expect that most students will be able to finish the first part of the homework during the 75-minute lab.

Plan of the lab

The key to successful implementation is to *get an end-to-end solution working as quickly as possible*. You want a program that does *something* which you can work with and then improve. Programming is easier and more fun when your code always does something. The plan of the lab is therefore to get you set up and compiling right away, then move toward a working `brightness` program one step at a time. The slogan is

Make it compile; make it run; make it right

Steps toward `brightness`

1. Find your assigned programming partner, introduce yourself, and grab a computer. If you can't find your partner or your partner isn't in the lab, please consult the lab staff.
2. Use `git` to create a directory for the assignment, which will contain the compile script that is also linked from the web page for the first homework assignment. Running

   ```
   git clone linux.cs.tufts.edu:/comp/40/git/intro
   ```

   should create an `intro` directory containing the script `compile`. If you have trouble, you can try instead

   ```
   git clone /comp/40/git/intro
   ```

[1] Because it's the first lab, we've prepared an elaborate handout. Most labs will have shorter handouts.

which will work in the lab, but not at home. (And you might ask the course staff for help with your login shell.)

During lab we will be working only on the `brightness` program, so you will use

```
    ./compile -link brightness
```

which will link only that program.

3. Write the simplest program that does something like `brightness`. In this case, we suggest that you write a program which processes the command-line arguments. If your program is supposed to open a file, open it; if `fopen()` fails, issue a suitable error message. *The idioms for C programmers will help.*

   Instead of doing anything with a file you open, simply print a message saying that you got a file but did nothing.

4. You should now be able to compile and link your first version of `brightness`. Run

   ```
   sh compile -link brightness
   ```

5. We should check for memory errors, but nobody can resist running a brand-new program. Here are some commands to try:

   ```
   ./brightness < /dev/null
   ./brightness brightness
   ./brightness < brightness
   ./brightness one two three
   ```

   If anything nasty happens, proceed to the next step and run with `valgrind`.

6. *Now* check for memory errors and leaks using `valgrind`:

   ```
   valgrind ./brightness < /dev/null
   valgrind ./brightness brightness
   valgrind ./brightness < brightness
   valgrind ./brightness one two three
   ```

   If you get a leak, maybe your forgot to close a file after opening it? If you get a message you don't understand, ask one of the course staff helping out at lab.

7. Your next step is to stop and think about the problem. You'll think about *the work of the program* and *the names of the functions that will do that work*. We expect you to choose good names for functions, and your choices affect your grades. This part of the lab will give you practice in a *very* simple setting, and you can ask the lab TAs to check your work.

   (a) Get the lab form by running

   ```
       git clone linux.cs.tufts.edu:/comp/40/git/intro-lab
   ```

   (If your shell is broken, you may need to clone just `/comp/40/git/intro-lab`.)

   (b) Complete the exercise that you find there: identify three potential functions and three possible names for each one.

   (c) If you like, check with a member of the lab staff.

   (d) **Submit your work** by running `submit40-lab-intro`.

8. Now you can start making your nascent `brightness` program actually do some work. Learn how to use the `Pnmrdr` interface to read some portable graymap files. *This step is going to take some time and thought.* You have to take time to understand the interface. You'll be thinking about interfaces a lot in 40, so if you're not confident, now is a good time to ask the lab staff for explanations.

   Once you have an idea how the interface is supposed to work, once again start by writing something really simple—use the interface to read the file's header, then `get` exactly the right number of pixels. Don't try to compute brightness; just print a message indicating how many pixels you've gotten.

9. Recompile. Try running

   ```
   sh -x compile -link brightness
   ```

   and be sure you understand what each `gcc` command is doing—in the future, you'll write your own `compile` scripts. If you have questions about the compile script, ask the lab staff.

10. Try out your new code with actual graymaps in `/comp/40/images`:

    ```
    djpeg -grayscale /comp/40/images/erosion.jpg  | ./brightness
    djpeg -grayscale /comp/40/images/halligan.jpg | ./brightness
    ```

    If you want to see what the grayscale images look like, try piping to a viewing program, e.g.,

    ```
    djpeg -grayscale /comp/40/images/erosion.jpg  | display -
    djpeg -grayscale /comp/40/images/halligan.jpg | display -
    ```

11. Once again, run with `valgrind` to be sure you have no leaks or errors:

    ```
    djpeg -grayscale /comp/40/images/erosion.jpg  | valgrind ./brightness
    djpeg -grayscale /comp/40/images/halligan.jpg | valgrind ./brightness
    ```

12. Your penultimate step is to write a function to compute and print average brightness. If you've done things right, *your* `main` *function won't change*, except possibly at one call site. If you haven't done things right, now is the time to fix your `main` function. (Our goal is *separation of concerns*: `main`'s job is to validate the command-line arguments and come up with an open file handle; doing something with that file is a job that should be delegated to some other function.)

    Writing `brightness` is pretty easy; less easy is figuring out *how you are going to test it*. Hint: the man page for `pgm` has a sample portable graymap. Can you use a text editor to create a `pgm` file whose average brightness is known?

13. Your ultimate step is to check your work again by running under `valgrind`.

Programming technique

Successful C programmers use these techniques without even having to think about them. Make them habits!

- Compile early and often. Insanely often.
- Run frequently (but don't lose your sanity).
- The moment your code appears not to be working, use `valgrind`.
- If your code appears to be working, use `valgrind`!

If you finish the lab early

Please use your extra time constructively by *deliberately introducing leaks and errors* into your code. This exercise will help you understand `valgrind` and what it can do for you.

- Try forgetting to free the PNM reader when you are done with it.
- Try using a variable without initializing it.
- Try calling `free` instead of `Pnmrdr_free` on a PNM reader.
- Try calling `free` instead of `Pnmrdr_free` on a PNM reader, and then keep on using the PNM reader.

Each of these exercises will help you learn to understand messages from `valgrind` and how they correlate with typical programming errors. When you start tackling harder problems, this experience will be invaluable.