CS61C Spring 2015 Homework 1: beargit

TAs: Martin Maas, Sagar Karandikar

Due Sunday, February 8, 2015 @ 11:59pm

- Update 2/2 7:00 PM: Fixed spaces for "beargit status" sample output, added clarifications for "beargit rm", fixed minor typos

Goals

- Learn more about git by building a simpler version, called beargit (Go Bears!)
- Write a substantial C program

What is beargit?

git is a great tool for managing source code and other files. However, even great tools can be used for evil; what if someone uses it to create git commits with hideous messages such as "St****rd RuLEz!"? So in this homework, you will be developing your own version of git, which will put an end to such behavior by requiring every commit message to contain the words "GO BEARS!". ;-)
At its core, beargit is essentially a simpler version of git, which you should have become familiar with in Lab 0. beargit can track individual files in the current working directory (no subdirectories!). It maintains a .beargit/ subdirectory containing information about your repository. For each commit that the user makes, a directory is created inside the .beargit/ directory (.beargit/<ID>, where <ID> is the ID of the commit). The .beargit/ directory additionally contains two files: .index (a list of all currently tracked files, one per line, no duplicates) and .prev (contains the ID of the last commit, or 0..0 if there is no commit yet). Each .beargit/<ID> directory contains a copy of each tracked file (as well as the .index file) at the time of the commit, a .msg file that contains the commit message (one line) and a .prev file that contains the commit ID of the previous commit.

Key differences between beargit and git:

- The only supported commands are init, add, rm, commit, status and log. For each of them, only the most basic command line options are supported.
- beargit does not track diffs between files. Instead, each time you make a commit, it simply copies all files that are being tracked into the .beargit/<ID> directory (where <ID> is the commit ID).
- Commit IDs are not based on cryptographic hash functions, but instead are a fixed sequence of 40-character strings that only contain '6', '1' and 'c' (why we chose those characters is left as an exercise to the reader).
- Any commits with a commit message that does not contain "GO BEARS!" (with exactly this capitalization and spelling) will be rejected with an error message.
- No user, date or other additional information is tracked by beargit. It does not allow to track subdirectories, or files starting with '.'.
- The rm command only causes beargit to stop tracking a file, but does not delete it from the file system.

Files:

- beargit.c - This is the file that you will fill in with your implementation of beargit.
- beargit.h - Do not edit - This file contains declarations of various constructs in beargit.c along with convenient #defines. See the "Important Numbers" section below.
- main.c - Do not edit - Contains the main for beargit (which parses command line options and calls into the functions defined in beargit.c).
- Makefile - Do not edit - This tells the program make how to build your code when you run the make command. This is a convenient alternative to having to repeatedly type long commands involving gcc.
- util.h - Do not edit - Contains helper functions that you may wish to use when completing the assignment.

You should only modify and submit beargit.c. Our autograder will overwrite all other files with a fresh copy.

Important Numbers: (see beargit.h)

In lecture, you learned about using #define to define constants as a single source of truth. You should use the appropriate constants from beargit.h whenever you are using any of the following numbers:

- Commit ID lengths are limited to 40 characters (not including the null terminator).
- Filenames are limited to 512 characters (including null terminator).
- Commit messages are limited to 512 characters in length (including null terminator).

Preliminaries:

For this homework, you will be using some C functionality that you may not be familiar with. We will now highlight some of these features:

C library functions:

You may wish to familiarize yourself with the following C library functions: fprintf, sprintf, fopen (and fclose, fwrite, etc.), strcmp, strlen, strtok, and fgets. You can find documentation of the C library here (use the search box at the top to find out about each function). Make sure not to stray away from the "C library" section, as the linked website also contains C++ documentation.

When you look at the existing code in beargit.c, you will see examples of how these functions can be used to achieve the desired functionality. We recommend trying to understand the provided functions first, before starting to implement your own.

Handling I/O (more than just printf):

Unix machines use a concept called "streams" to handle arbitrary I/O. We will need two of these output streams in this homework. The first is stdout, which is where your output goes when you call printf. We will use stdout to output all output indicating a "successful" action. The other output stream is stderr, which is where we will output all error messages. By default, both of these streams, stdout and stderr, are printed to your screen when you run a program.

Outputting to either stdout or stderr can be done similarly to using printf. The only change is that you use the fprintf function, and the first argument you pass in must be either "stderr" or "stdout" (without quotes).

```
[inside your C code]
fprintf(stdout, "%d\n", 3); // prints the number 3 to stdout, along with a newline
fprintf(stderr, "%d\n", 4); // prints the number 4 to stderr, along with a newline
```

If you want to know what messages went to stdout or stderr, you can forward them to a file instead of the terminal, by appending 2>log_err and/or 1>log_out to your command (e.g., ./my_program 2>log_err). This will forward everything from stderr to the file log_err (and equivalently for stdout if you add 1>log_out).

Included helper functions:

To make life easier for you, we provide helper functions for common operations that you will encounter while implementing beargit. You fill find these in utils.h. Here is a brief overview of each of these functions:

- void fs_mkdir(const char* dirname): Create a new directory of name dirname
- void fs_rm(const char* filename): Delete the file filename
- void fs_mv(const char* src, const char* dst): Move the file src to dst, potentially overwriting it
- void fs_cp(const char* src, const char* dst): Copy the file src to dst, potentially overwriting it
- void write_string_to_file(const char* filename, const char* str): Create or overwrite the file filename and write str into it, including the NULL-character
- void read_string_from_file(const char* filename, char* str, int size): Open the file filename and read its content into the location pointed to by str; limit the amount to read to at most size bytes, including the NULL character

The last two functions should only be used together. Specifically, don't try to use read_string_from_file to read multi-line files, but only for single strings that you previously wrote into a file using write_string_to_file.

While these functions perform some basic checks to prevent you from accidentally overwriting important files, be careful whenever you call any function that modifies the file system. There is always a risk of unintentionally deleting or overwriting files, especially when working on your own machine!

Setup:

To get starter code for this homework, we will be using git. First, you must enter into your cs61c class repository in your class account (this is the repository you created in last week's lab; probably located in ~/work). Once in this directory, run the following:

```
git remote add hw1_starter git@github.com:cs61c-spring2015/hw1_starter
git fetch hw1_starter
git merge hw1_starter/master -m "merge hw1 skeleton code"
```

Once you complete these steps, you will have the hw1 directory inside of your repository, which contains the files you need to begin working.
As you develop your code, you should make commits in your git repo and push them as you see fit. Be sure not to get confused between beargit, which you are writing for this homework, and git, which is managing your real class repository.

Required functionality:

While the version of beargit that we've given to you compiles, you can't do much except call beargit init to create a new repository, and call beargit add <file> to start tracking a file. Everything else you need to implement yourself!
We recommend that you implement the beargit commands in this order, as this makes testing easier:

1. beargit status
2. beargit rm
3. beargit commit
4. beargit log

For each of these, you need to implement one of the functions below (but feel free to define new helper functions to make things easier). We give you an outline of each function's job, as well as the errors you need to be able to detect, and the output you need to produce.
Note: Whenever you see the notation <something>, you should replace it with the appropriate value for *something*, without the angle brackets

Testing your code in CS61C

Unlike CS classes you may have taken in the past, we will not provide you with a full autograder for the assignment. Instead, you should devise a methodology to test your code to ensure that it performs as you intend it to. The autograder that produces your final grade will include many more test cases than the autograder/sanity check provided with the homework.
But... why?
When you write production code in the "real" world (and upper division classes), much of the time you will not be provided with any test cases to validate your code against (not even a sanity check). The ability to write good test cases is just as important a skill for a programmer as the ability to write functioning code.
The test cases you write for this homework won't be submitted or graded, but we may ask you to submit test cases for future assignments.
Automated basic tests
To make life a bit easier for you, we are providing you with two ways to test your implementation. The first one is an automated testing tool that will run your implementation against a series of basic tests to determine whether its output is sensible. Note that this is just a small subset of the tests that the actual autograder will be running. Even if your program passes all these tests, it may still fail on some of the test cases in the autograder. You therefore shouldn't rely on this tool for your testing but consider it a sanity check.
To run these tests, go into the main source directory and type make check. You will see output similar to the following:

```
Running test cases...

    [  OK  ] beargit_test_add_0
    [  OK  ] beargit_test_add_1
    [  OK  ] beargit_test_rm_0
    [ FAIL ] beargit_test_rm_1 file is missing but no (or incorrect) error message (error type: OUTPUT)
    [ FAIL ] beargit_test_status_0 expected 4 lines of output but found 0 (error type: OUTPUT)
    [ FAIL ] beargit_test_status_1 expected 2 lines of output but found 0 (error type: OUTPUT)
    [  OK  ] beargit_test_commit_0
    [ FAIL ] beargit_test_commit_1 successful commit should not display any output (error type: OUTPUT)
    [ FAIL ] beargit_test_log_0 there are no commits, but no correct error message was shown (error type: OUTPUT)
    [ FAIL ] beargit_test_log_1 there are no commits, but no correct error message was shown (error type: OUTPUT)

TESTS PASSING: 4 / 10
```

You should pay close attention to the error messages, as they are designed to give you a hint what is going wrong.
Manual testing
For your own testing, we provide you with a script that creates a new test directory for you (called test), which you can use to experiment with your implementation in a fresh directory (where it will generate the .beargit/* files and directories). Every time you use the script, your previous directory will be deleted, so you can start afresh. **Be careful to not leave any important data in the test directory!**
To run the script and create a new test directory, run the following in your hw1 directory (this will automatically move you into the test directory and add your beargit executable to the PATH, so that you can run it):

```
$ source init_test
```

You can then run commands such as:

```
$ beargit init
$ touch test.txt
$ beargit add test.txt
```

Step 1: The status command

Functionality:

The status command in beargit should read the file .beargit/.index and print a line for each tracked file. The exact format is described below. Unlike git status, beargit status should not print anything about untracked files.

Output to stdout:

```
$ beargit status
Tracked files:

    <file1>
    [...]
    <fileN>

<N> files total
```

For each file in the above output, <file*> should be replaced with the filename of that file.

Return value and output to stderr:

This function should always return 0 (indicating success) and should never output to stderr.

Step 2: The rm command

Hint: You may want to have a look at the provided implementation of beargit add before implementing this command.

Functionality:

The rm command in beargit takes in a single argument, which specifies the file to remove from the index (which is stored in the file .beargit/.index). If the filename passed in is not currently being tracked, you should print an error as indicated below. Note that this behavior is different from git in that it doesn't delete the file from your file system.

Output to stdout:

None.

Return value and output to stderr:

If the filename specified in the provided argument exists in the index, the function should return 0 and produce no output on stderr. If the filename specified does not exist in the index, the function should return 1 and output the following to stderr:

```
$ beargit rm FILE_THAT_IS_NOT_TRACKED.txt
ERROR: File <filename> not tracked
```

Step 3: The commit command

Functionality:

The commit command involves a couple of steps:

- First, check whether the commit string contains "GO BEARS!". If not, display an error message.
- Read the ID of the previous last commit from .beargit/.prev
- Generate the next ID (newid) in such a way that:
    1. All characters of the id are either 6, 1 or c
    2. Generating 100 IDs in a row will generate 100 IDs that are all unique (*Hint: you can do this in such a way that you go through all possible IDs before you repeat yourself. Some of the ideas from the number representation class may help you!*)
- Generate a new directory .beargit/<newid> and copy .beargit/.index, .beargit/.prev and all tracked files into the directory.
- Store the commit message (<msg>) into .beargit/<newid>/.msg
- Write the new ID into .beargit/.prev.

IMPORTANT RULE THAT WILL AFFECT YOUR GRADE IF YOU DON'T READ IT!

Now that we have your attention: when implementing the code that checks whether the commit message includes GO BEARS!, you are not allowed to use any library functions, including any of the str* ones you may have seen before.

Output to stdout:

None.

Return value and output to stderr:

If the commit message does not contain the exact string "GO BEARS!", then you must output the following to stderr and return 1:

```
$ beargit commit -m "G-O- -B-E-A-R-S-!"
ERROR: Message must contain "GO BEARS!"
```

If the commit message does contain the string "GO BEARS!", then the function should produce no output and return 0.

Step 4: The log command

Functionality:
The goal of the log command is to print out all recent commits. See below for the individual steps:

- List all commits, latest to oldest. .beargit/.prev contains the ID of the latest commit, and each directory .beargit/ contains a .prev file pointing to that commit's predecessor.
- For each commit, print the commit's ID followed by the commit message (see below for the exact format).

Output to stdout:

```
$ beargit log
[BLANK LINE]
commit <ID1>
        <msg1>
[BLANK LINE]
commit <ID2>
        <msg2>
[...]
commit <IDN>
        <msgN>
[BLANK LINE]
```

Return value and output to stderr:
If there are no commits to the beargit repo, beargit should return 1 and output the following to stderr:

```
[assume that no commits have been made]
$ beargit log
ERROR: There are no commits!
```

If there are commits, you should produce the output indicated in the "Output to stdout" section above and return 0.

Submission

There are two steps required to submit hw1. Failure to perform both steps will result in loss of credit:

1. First, you must submit using the standard unix submit program on the instructional servers. This assumes that you followed the earlier instructions and did all of your work inside of your git repository. To submit, follow these instructions after logging into your cs61c-XX class account:

   ```
   $ cd ~/work                          # your git repo, should contain a directory called hw1 with your soln
   $ cd hw1
   $ submit hw1
   ```

   Once you type submit hw1, follow the prompts generated by the submission system. It will tell you when your submission has been successful and you can confirm this by looking at the output of glookup -t.

2. Additionally, you must submit hw1 to your GitHub repository. To do so, follow these instructions after logging into your cs61c-XX class account:

   ```
   $ cd ~/work                          # your git repo, should contain a directory called hw1 with your soln
   $ git add -u                         # should add all modified files in hw1 directory (must include beargit.c)
   $ git commit -m "Homework 1 submission"
   $ git tag -f "hw1"                   # The tag MUST be "hw1". Failure to do so will result in loss of credit.
   $ git push origin master --tags      # Note the "--tags" at the end. This pushes tags to github
   ```

Resubmitting
If you need to re-submit, you can follow the same set of steps that you would if you were submitting for the first time. The only exception to this is in the very last step, git push origin master --tags, where you may get an error like the following:

```
(21:28:08 Sun Feb 01 2015 cs61c-ta@hive12 Linux x86_64)
~/work $ git push origin master --tags
Counting objects: 22, done.
Delta compression using up to 8 threads.
```

```
Compressing objects: 100% (19/19), done.
Writing objects: 100% (21/21), 9.73 KiB | 0 bytes/s, done.
Total 21 (delta 4), reused 0 (delta 0)
To git@github.com:cs61c-spring2015/cs61c-ta
       bf20433..d1ff9ed  master -> master
  ! [rejected]        hw1 -> hw1 (already exists)
error: failed to push some refs to 'git@github.com:cs61c-spring2015/cs61c-ta'
hint: Updates were rejected because the tag already exists in the remote.
```

If this occurs, simply run the following instead of git push origin master --tags:

```
$ git push -f origin master --tags
```

Note that in general, force pushes should be used with caution. They will overwrite your remote repository with information from your local copy. As long as you have not damaged your local copy in any way, this will be fine.