CS61C Spring 2015 Homework 2: More `beargit` and Unit Testing in C

TAs: Sagar Karandikar, Martin Maas

Due Sunday, February 15, 2015 @ 11:59pm

- Update 2/11 8:15 PM: Added a few clarifications to the functionality descriptions of `beargit checkout` and `beargit branch`.
- Update 2/9 9:10 PM: We made another fix to the hw2_starter repo. If you ran git fetch before this point, please update. If you're starting after the listed time, this change does not affect you. We've also reduced the required number of tests you must submit from 3 to 2.
- Update 2/9 4:57 PM: Fixed typo in the `beargit branch` sample output
- Update 2/9 6:00 AM: Clarification that the `fs_*` functions do not require all paths to be within `.beargit` anymore.

Goals

- Learn about branches and checkouts in `git` and add similar functionality to `beargit`.
- Learn to test your C application to aid in building robust programs.

More `beargit`? Give me a break...

Last week, you implemented a basic version of `beargit` that supports `init`, `add`, `rm`, `status`, `log` and `commit`. However, this is not very useful yet -- while you can create a beargit repository and commit data to it, there is no way to retrieve it and go back to the state of a previous commit. In the first part of this week's homework, you will complete your `beargit` implementation to make this possible. And you will implement support for branches as well!

In addition, we have included a testing framework called CUnit, which provides a nice interface for writing and running unit tests. In this homework, you'll be required to write and submit tests for your code. This should hopefully help you resolve any deficiencies that exist in your hw1 submission.

Setup:

We are making a lot of changes to `beargit`, so your HW1 will stop working temporarily until you have finished the whole implementation of HW2. Fortunately, `git` can help you with that: for this homework, you will be doing all work on a new branch within your git repository -- you can always change back to your original branch (called `master`) where you will find your previous, working copy of HW1.

Step 5 below will walk you through fetching the updated codebase. Git will, in effect, "magically" merge together your implementation and our changes to the skeleton.

Important changes since last week

- We added a new numerical constant: `BRANCHNAME_SIZE`, the maximum length of a branch name (including NULL terminator)
- We refined the fs_* functions, they now also show their arguments at an error. They also don't require all files to be within `.beargit` anymore.
- We added a new helper function: `int fs_check_dir_exists(const char* dirname)`. This function tests whether a given directory exists.
- We added `cunittests.c`, a new file where you can define unit tests to test your code.

Unit Testing in C

In order to make testing easier for you, we've hooked up a framework called CUnit to the beargit code. You can learn more about CUnit here.

One issue with Unit Testing is that by default you wouldn't be able to capture the output of calls to printf or fprintf (to stdout and stderr). Since your outputs to stdout/stderr are important for beargit, we've provided some code that replaces calls to printf/fprintf with a custom function that directs output to two files, `TEST_STDOUT` and `TEST_STDERR`. You can read these files as you would any other file, allowing your testing code to analyze the output that your functions print to the screen.

All of your unit tests will live in `cunittests.c`. We've provided two example test suites, each containing one test, along with test suite initialization functions to make your life easier. The initialization function (`int init_suite(void)` in `cunittests.c`) will destroy any existing .beargit directory, and remove old copies of `TEST_STDOUT` and `TEST_STDERR`. You will most likely not need to use the `clean_suite` function, which runs at the end of a test suite, but we have provided the stub in case you need it.

What is a "suite"?

A test suite is essentially a collection of tests. To add test suites, you can use the following boilerplate code in `cunittests.c`. Two examples of this are already provided.

Creating the Test Suite

You'll need to add the following code in `cunittester()` once per test suite:

```
[... inside of cunittester() in cunittests.c  ...]
[ Replace pSuiteN below with a suitable name, like pSuite5 ]

CU_pSuite pSuiteN = NULL; // replace N with the test number
/* add a suite to the registry */

/* You don't necessarily have to use the same init and clean functions for
 * each suite. You can specify the function names in the next line:
 */
pSuiteN = CU_add_suite("Suite_N", init_suite, clean_suite);
if (NULL == pSuiteN) {
   CU_cleanup_registry();
   return CU_get_error();
}
```

Adding Tests to the Suite

You'll need to add the lines below for each test function that you want to add to the suite. In the example below, we are adding the function `simple_sample_test` to the suite.

```
[... also inside of cunittester() in cunittests.c ...]
/* Add test named simple_sample_test to Suite #N */
if (NULL == CU_add_test(pSuiteN, "Simple Test #N", simple_sample_test))
{
   CU_cleanup_registry();
   return CU_get_error();
}
```

How Tests Are Run

CUnit performs the following actions when running a test suite:

1. Runs the suite initialization function. In the above code, this function is called `init_suite`.
2. Runs all of the tests you added to the suite. In the above example, this runs only the function named `simple_sample_test`.
3. Runs the suite cleanup function. In the above code, this function is called `clean_suite`.

The initialization function is useful, because you can use it to automatically destroy any existing `.beargit` directory before your tests run, so that you can create a new repo with `beargit init`. In the code we have given you, we do not destroy files in the cleanup function (the function is actually empty). This allows you to peek into the `.beargit` directory and the `TEST_STDOUT` and `TEST_STDERR` files in case you need to do so manually.

If you want to get started on testing right away, please skip ahead to Step 8 to see how you can run the tests for your `beargit` implementation. If you prefer to get started on finishing `beargit` first, please reed on.

How branches and checkouts work in git

You can go to any commit in the history of time if you know its ID. This is called "checking out a commit". The current state of the working directory will be completely restored to how it was during the time of that commit.

Branches in git are basically just diverging commit histories. You have an "alternate history" depending on which branch you are on. One way to think about branches is that they allow multiple commits to point to the same previous commit: two branches can have a shared history, and then at some point they do different things starting from a certain point in time.

So every commit has a predecessor, but multiple commits can actually have the same predecessor. In fact, branches themselves are just identifiers for specific commits (which are called the "HEAD" of a branch). Just like commits, you can also check out a branch: in that case, you switch to that branch's HEAD commit. You can also check out commits that are not the HEAD of any branch -- in that case, you say you are "detached", because you are not on any specific branch.

To add branches in `beargit`, not much changes: every commit still has exactly one predecessor (.prev), but multiple commits can have the same predecessor now. Branches in beargit are just pointers to specific commits. To keep things simple, we only allow beargit to commit when you are at the HEAD of a branch (i.e., when you are not detached). This allows you to "grow" each branch forwards.

When you are at any commit, you can start a new branch from there: you can say `git checkout -b <new_branchname>` to start a new branch that has the current commit as its HEAD. You can then start an alternative history by committing on this branch. When you initialize a new beargit repository, a default branch `master` is created, and its HEAD points to the 00.0 commit ID.

Visualizing Branches

To help you get a better sense of how branches actually work, you should work through the following tutorial until you are satisfied that you understand what branches do: http://pcottle.github.io/learnGitBranching/.

Required functionality:

While implementing branches may sound very complicated, it is not much additional work to what you have already implemented -- in your last homework, you have created a solid foundations to build upon, so now things get easier.

Directory structure

We will implement branches very similarly to how we implemented tracking of files. All we have to do is add a few files to our directory structure:

- `.beargit/.branches` is a file that contains a line for every branch that exists. We will call the line number on which the branch exists in this file the "branch number" (starting from 0).
- `.beargit/.current_branch` contains a single string with the name of the current branch if we are at the HEAD of some branch, or is an empty string if we are not on some branch HEAD.
- `.beargit/.branch_<branchname>` (one for every branch). This is a copy of the .prev file that belongs to the branch head (i.e., the HEAD commit of the branch)

With this information, we can now implement beargit branch and beargit checkout.

Step 5: Update your HW1 code!

Updating your hw1 code to hw2 is somewhat complicated, but `git` will do most of the heavy lifting for us. Run the following to update:

First, we will create a branch that we will use to work on hw2. This means that our hw1 code will remain as is on the `master` branch, while we work on hw2 on the `hw2` branch:

```
[... assuming you are inside your repository on a hive machine ...]
$ git checkout -b hw2
```

This creates a new branch and "switches" to that branch in the same command. Next, let's pull the updated skeleton from github and apply the changes to our branch:

```
$ git remote add hw2_starter git@github.com:cs61c-spring2015/hw2_starter
$ git fetch hw2_starter
$ git merge hw2_starter/master -m "add hw2 starter code"
```

At this point, git will automatically merge the skeleton updates and your hw1 solution code. Unfortunately in some cases, git will not be able to resolve the conflicts automatically. If git reports to you that the "automatic merge failed" when running the previous command, you will need to resolve the conflicts using the following method:

A text-based version of these same instructions can be found on the GitHub website. In the worst case, you can also grab a blank copy of `beargit.c` from the hw2_starter repo and manually copy in the code you filled into the old copy of `beargit.c`. You can get the latest `beargit.c` for hw2 here.

Once the updates have been successfully merged, you can continue with the instructions:

Lastly, you'll notice that our directory is still annoyingly named `hw1`. Let's fix that now:

```
$ git mv hw1 hw2
```

Finally, let's push this to GitHub so that we have a good starting place in case something goes wrong when completing hw2:

```
$ git commit -m 'added hw2 starter code, merged with my hw1 solution'
$ git push origin hw2
```

Note that in the above, instead of running `git push origin master`, we ran `git push origin hw2`. This pushes our `hw2` branch to GitHub. You should be able to view the contents of this branch by clicking the branches link on your repository on GitHub and selecting "hw2".

BUT WAIT! We're not done yet: After following the above steps, you will need to make two changes to your newly formed `beargit.c` file: you will need to add a `_hw1` to the function names of your `beargit_commit` and `next_commit_id` functions (so they become `beargit_commit_hw1` and `next_commit_id_hw1`; note that you only should change those two lines, not the lines where they are called). We will now explain why these changes are necessary.

Since we are now adding branching functionality to beargit, we have to change our next_commit_id function: in HW1, we generated the next commit ID solely based on its predecessor. Without branches, this would give us unique IDs, but now that we can have multiple commits with the same predecessor, the commit ID cannot just depend on the last commit -- in that case, all branches following a commit would have the same ID, which is not unique anymore.

To prevent this, we have adapted the next_commit_id function to write the branch number into the first 10 digits of the string, for a total of 3^10 possible branches. If the remainder of the commit ID uses the same technique as in HW1 to generate unique IDs, we know that the IDs on each branch are unique, and since the IDs between branches are different as well, we know that all IDs are unique.

To help you with this change, we have prepared a new `next_commit_id` function that uses your `next_commit_id` function to fill in the last 30 digits of the commit ID, while our function is filling in the rest.

This means that you have to change your `next_commit_id` function to now only generate 30 digits -- note that we give you a pointer to digit 10 of the string, so nothing in your functionality should change, except that you now operate on 30 digits instead of 40. We still require you to generate at least 100 unique IDs.

Step 6: The `branch` command

Functionality:

Beargit branch prints all the branches and puts a star in front of the current one. Do you remember beargit status? This is almost the same: you need to read the entire .branches file line by line and output it. However, you also need to check each line against the string in .current_branch. If they are the same, you need to print a * in front of it.

Note that we require you to print branches in the order of creation, from oldest to latest. Also note that if you have checked out a commit previously (in contrast to a branch), you are detached from the HEAD and don't have to print a star in front of any branch. This is even true if the commit you checked out is actually the HEAD of a branch.

Output to stdout:

```
$ beargit branch
  <branch1>
  <branch2>
  [...]
* <current_branch>
  [...]
  <branchN>
```

Return value and output to stderr:

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

Step 7: The `checkout` command

Functionality:

This is the command that is the most important feature of beargit. It allows you to restore the state of any commit in time, as well as to switch and create branches. beargit checkout has three different behaviors:

- `beargit checkout <commit_id>`: Check out a particular commit (i.e., leaving a branch HEAD if you are on it; this is called a "detached" state. You can assume that whenever you call this, you become detached, even if the commit you are checking out is some commit's HEAD).
- `beargit checkout <branch>`: Check out an existing branch and check out its head.
- `beargit checkout -b <newbranch>`: Start a new branch at the current commit.

While these behaviors look very different, they are actually very similar. First, you need to find out which of the three cases it is. We give you whether the user has provided `-b` (the new_branch bool parameter) and then the other argument, which can be either a commit ID or a branch name.

So beargit first needs to find out if you are giving it a commit or a branch name. For this, we have prepared a function `is_it_a_commit_id`, which you need to fill in. The function takes a string and returns true if and only if the string is 40 characters that are each 6, 1 or c.

Once you know whether you are dealing with a branch or a commit, you have to do one of two things:

1. If it's a commit, check out the commit by replacing the currently tracked files with those from the time of the commit.
2. If it's a branch (and you're not creating a new one), first check whether it exists. If yes, you need to switch to that branch. This means that you first store the latest commit of the current branch into the branch_branchname file, and then replace the content of current_branch by the new branch. You then read the branch_newbranch file to find out the HEAD commit of that branch, and then you check that commit out just like in 1).
3. You are creating a new branch. This is very similar to 2), but you also have to add the branch to the .branches file and instead of reading the HEAD ID from .branch_branchname, you make the current prev ID the head ID for that branch and store it into that file.

Since we are nice people, we actually implemented the functionality above for you, except for the implementation of the actual checkout! But because we had to write this homework in a rush, there are three mistakes in the beargit_checkout function -- you need to find and correct them for everything to run (one line per mistake). Consider using cgdb and printf for debugging to help you!

Note: The `beargit_checkout` function is taking two arguments: `new_branch` is true if and only if `-b` was supplied to the command, and `arg` contains the other command line argument.

After you found the mistakes, you have to write a function checkout_commit which will do the actual checkout of a commit by:

- Going through the index of the current index file, delete all those files (in the current directory; i.e., the directory where we ran beargit).
- Copy the index from the commit that is being checked out to the `.beargit` directory, and use it to copy all that commit's tracked files from the commit's directory into the current directory.
- Write the ID of the commit that is being checked out into .prev.
- In the special case that the new commit is the 00.0 commit, there are no files to copy and there is no index. Instead empty the index (but still write the ID into .prev and delete the current index files). You may wonder how we could ever check out the 00.0 commit, since it is not a valid commit ID; the answer is that if you check out a branch whose HEAD is the 00.0 commit, that checkout is expected to work (while 00.0 would not be recognized as a commit ID).

Once you are done, you should experiment with the checkout and branch functionality by creating new branches, checking out old commits and see how you can commit to different branches individually. There is a lot that can go wrong, so we recommend testing thoroughly, and writing CUnit tests.

Output to stdout:

None.

Return value and output to stderr:

If the argument is a commit ID (40 characters, each of which is '6', '1' or 'c') of a commit that exists, a branch that exists and `new_branch` is false, or a branch that doesn't exist and new_branch is true, the function should return 0 and produce no output on stderr.

If the argument is a commit ID but the commit does not exist, the function should return 1 and produce the following error:

```
$ beargit checkout 6666.66
ERROR: Commit <commit_id> does not exist
```

If the argument is a branch that exists but `new_branch` is true, the function should return 1 and produce the following error:

```
$ beargit checkout -b <branch_name>
ERROR: A branch named <branch_name> already exists
```

If the argument is a branch that does not exist but `new_branch` is false, the function should return 1 and produce the following error:

```
$ beargit checkout <branch_name>
ERROR: No branch <branch_name> exists
```

Step 8: Testing

As the final part of this assignment, you will need to write 2 test suites that each focus on a different beargit command. Each of the two test suites must have a comment at the top describing what beargit command the suite is designed to test and the kinds of error conditions the test will catch. You will write these in `cunittests.c`. This file will be turned in and a reader will look over your test code to ensure that your tests are reasonable.

We've also provided a linked list structure called `commit` inside of cunittests.c, which you may find helpful in programmatically keeping track of a sequence of commits in your test code. An example of its usage is found in `simple_log_test`.

Although you are only required to turn in 2 tests, it is highly recommended that you write additional tests to ensure that your implementation works as expected.

Running Tests

In order to run tests, you should do the following:

```
[assumes you are inside your hw2 directory]
$ make beargit-unittest
$ source init_test
$ beargit-unittest


     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/

rm: cannot remove '.beargit': No such file or directory <- You can ignore this

Suite: Suite_1
  Test: Simple Test #1 ...passed
Suite: Suite_2
  Test: Log output test ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      2      2    n/a      0        0
               tests      2      2      2      0        0
             asserts      4      4      4      0      n/a

Elapsed time =    0.007 seconds
```

Submission

There are **two** steps required to submit hw2. 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 hw2 with your soln
   $ cd hw2
   $ submit hw2
   ```

   Once you type `submit hw2`, 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 hw2 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 hw2 with your soln
   $ git add -u                                        # should add all modified files in hw2 directory (must include beargit.c)
   $ git commit -m "Homework 2 submission"
   $ git tag -f "hw2-sub"                              # The tag MUST be "hw2-sub". Failure to do so will result in loss of credit.
   $ git push origin hw2 --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 hw2 --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 hw2 --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  hw2 -> hw2
 ! [rejected]        hw2-sub -> hw2-sub (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 hw2 --tags`:

```
$ git push -f origin hw2 --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.