CS61C Summer 2015 Project 1: `beargit`

TA: Jeff Wettstein

Due Sunday, July 5, 2015 @ 11:59pm

- Update June 30, 4:25pm - Added section 6.5 which discusses a required error if you commit and are not on the HEAD of a branch

Goals

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

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 project, 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`, `branch`, `checkout`, `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`.
- `cunittests.c` - A file where you can define unit tests to test your code.
- `beargit.h` - Do not edit - This file contains declarations of various constructs in `beargit.c` along with convenient `#define`s. See the "Important Numbers" section below.
- `cunittests.h` - Do not edit - Declarations needed for use with the cunit library.
- `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 declarations you may want to use in this assignment.
- `util.c` - Do not edit - Contains implementation of the helper functions declaraed in until.h (you may want to see the specific details of what those functions are doing before you use them)

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), 10 of which are composed of the branch id.
- Filenames are limited to 512 characters (including null terminator).
- Commit messages are limited to 512 characters in length (including null terminator).
- Branch names are limited to 128 characters (including null terminator).

Preliminaries:

For this project, 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 project. 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
```

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
- `int fs_check_dir_exists(const char* dirname)`. This function tests whether a given directory exists.
- `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 the starter code for this project, we will be using `git`. You should have received an invitation to a private repository shared with your partner. First, create a local copy of your private project repository (repeat lab 0 with the new repo). Enter the repo directory and run the following:

```
git remote add proj1-starter git@github.com:cs61c-summer2015/proj1-starter
git fetch proj1-starter
git merge proj1-starter/master -m "merge proj1 skeleton code"
```

Once you complete these steps, you will have the `proj1` 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 project, 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`
5. `beargit branch`
6. `beargit checkout`

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 project.

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 project 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 three 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 interactive 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 `proj1` 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
```

Unit testing in C

In order to make automated 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 read on.

String manipulation tips and warnings

For a large portion of this project you will be dealing with manipulating strings. You can use this section as a reference if you are running into trouble

Concatinating two strings

There are two functions you can use: `strcat()` and `sprintf()`.

strcat(char * dst, char *src)

Note that `strcat(".beargit/","test")` is incorrect. ".beargit/" is a string literal which is of fixed size thus you cannot safely append test to the end of it. Instead you would want to do something of the sort:

```
char file[SIZE] = ".beargit/";
strcat(file, "test"); // Assuming size >= 14, file points to the string ".beargit/test"
```

sprintf(char * dst, char * format_string, ...)

This works exactly like printf() except that the resulting string is written into dst

Be careful of string literals!

```
char * str = "beargit/";
beargit[0] = 'a';
...
```

You may be tempted to do something like this, however it will produce a runtime error. The reason is that str points to a string literal, and string literals are stored in a read-only section. Thus if you want to be able to append to a string that you predefine in this way you must declare str as a char array. This case of declaration and initialization is one of the few cases where there is a difference between arrays and pointers.

```
char str[] = "beargit/";
beargit[0] = 'a';
...
```

In this case the string literal is still stored in read-only memory but the character array str is allocated on the stack and recieves a copy of each character in the literal. Thus since str points to a character array on the stack it is modifiable.

How can I remove newlines from strings when I'm reading in files

The function strtok() is going to help you accomplish this. Since you will be reading in a few single-line files this can be used to remove the newline from the end of the string.

```
 char * str = "...\n";
strtok(str, "\n");
```

All you need to know is this function is replacing the '\n' character with a NULL terminator thus effectively removing it from the end of your string. More information can be found in strtok() documentation if you are curious.

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. ID Length is COMMIT_ID_BYTES (not including NULL terminator)
    2. All characters of the id are either 6, 1 or c
    3. 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 number representation may help you!*)
    4. Calling next_commit_id(char* commit_id) results in commit_id being updated to a ID.
    5. The ID string consists of a branch-id (of size COMMIT_ID_BRANCH_BYTES) followed by a tag-id to fill the rest of the size of the ID. (Note: the tag-id used here has nothing to do with a git tag, git tags aren't involved in this project!)
    6. We have implemented the branch-id step for you in next_commit_id(char* commit_id). Don't worry too much about where the branch-id is coming from yet (more on that in part 5), but pay close attention to what indices in the commit_id string are being updated and how the pointer is being passed to next_commit_id_part1(). To finish the next ID generation you will need to complete next_commit_id_part1().
- 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.

NOTE: *beargit -m "GO BEARS!"* will result in an error because '!' is a special character in many shells, to avoid this issue use single quotes beargit -m 'GO BEARS!'

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 either all or a specified number of 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).
- If you pass in the -n flag (e.g. beargit -n 10), then limit the number of log records printed to the amount specified. If the -n flag is not passed, then the argument "int limit" will be set to INT_MAX.

Output to stdout:

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

Note: In order to pass our tests you must have exactly the same spacing as above!

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.

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. 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: 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 6: 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 project 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 6.5: Add a safety check to commit

We are enforcing the rule that you can only perform a commit when you are on the HEAD of a branch. Thus you need to modify commit such that if you attempt to commit and are not on the HEAD of a branch that you return an error and output:

```
... // no longer at the HEAD of a branch (i.e. checking out a specific commit id)
$ beargit commit -m 'GO BEARS!'
ERROR: Need to be on HEAD of a branch to commit
```

Note: We won't be picky about which error message appears in cases where the commit message doesn't contain "GO BEARS!" and you're not at the HEAD of a branch.

Step 7: 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 proj1 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 proj1. 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 proj1 with your soln
   $ cd proj1
   $ submit proj1
   ```

   Once you type `submit proj1`, 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 proj1 to your GitHub repository. To do so, follow these instructions after logging into your cs61c-XX class account:

   ```
   $ cd <proj1 repo directory>                      # your project git repo, should contain a directory called proj1 with your soln
   $ git add -u                                     # should add all modified files in proj1 directory (must include beargit.c)
   $ git commit -m "Project 1 submission"
   $ git tag -f "proj1-sub"                         # The tag MUST be "proj1-sub". Failure to do so will result in loss of credit.
   ```

```
$ git push origin proj1 --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 proj1 --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 proj1 --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-summer2015/cs61c-ta
   bf20433..d1ff9ed  proj1 -> proj1
 ! [rejected]        proj1-sub -> proj1-sub (already exists)
error: failed to push some refs to 'git@github.com:cs61c-summer2015/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 proj1 --tags`:

```
$ git push -f origin proj1 --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.