Assignmnent 0: Testing Software

Author: Peter Alvaro
For: CSE 13S-02
Version 2

January 2024

1 Introduction

Brad Daylight, junior software engineer at Birdcat LLC, has recently inherited responsibility for maintaining a collection of programs written by people who no longer work for the company. It is now *his* job to make sure that this software works correctly, even though he had no role in the decisions that were made writing the code[1].

To make matters worse, this software was written in the C programming language, a memory-unsafe language from the late 1900s that Brad has only this week started to learn. In fact, in some cases, he does not even have the source code anymore, but needs to test binary programs without looking at their implementation!

2 Counting birds with `calc`

Everyone in the bird counting department has, over the years, found the need to implement a simple arithmetic calculator in order to simplify bird-counting operations. There is no need for all of these redundant programs, but some correct implementation should be retained (because who wants to write it again?) All of the programs were written to adhere to this specification (or spec for short):

A calculator program takes 2 tiny integers as command line arguments. If the arguments are not integers, it should print "BAD INPUT" to the console (and return something other than zero, to indicate that something went wrong). If there are fewer than two arguments, it should print "NOT ENOUGH INPUT" and return non-zero. If either of the integers is not tiny (that is, if either is less than -512 or more than 512), it should print "TOO BIG" and return non-zero. Otherwise, it should print the sum of the two integers and return zero. Every print statement should be followed by a new line.

For example:

```
> calc 5 187
192

> calc 3 3
6

> calc 3
NOT ENOUGH INPUT
```

[1] Read about "technical debt": `https://en.wikipedia.org/wiki/Technical_debt`.

Brad's current plan is to create a collection of shell scripts, each of which is intended to test a particular property of an implementation of `calc`. "I should probably read the spec carefully and write down all of the properties that I want to test before I start writing any code," he thinks. But immediately forgetting himself, he spins up an editor and begins writing a simple test: a shell script that he has named "basic_addition.sh" and whose contents are as follows:

```
./calc 3 4 > 3.4.txt
echo 7 > 7.txt
diff 3.4.txt 7.txt
```

Shell scripting is probably unfamiliar to most of you, so let's walk through this example slowly, one piece at a time. The first thing to point out is that a shell script is nothing more than a sequence of commands that Brad could have typed at the command line, and which will be executed in sequence when the script runs. If you are ever confused about what is happening when a shell script runs, you can try running the individual pieces by hand.

3 Shell Scripting

One of the reasons we write shell scripts is to avoid writing the same commands over and over; another is to make sure that we do *exactly* the same sequence of things every time. Sometimes it is tempting to test code by running it manually and simply looking at what it does. But this approach is prone to errors, and can become awkward when we want to show someone else that the code is running correctly (or incorrectly). When we are testing code, repeatability is very important.

```
./calc 3 4 > 3.4.txt
```

The first line of the script is doing more than one thing. First, it is invoking the `calc` program as we saw in the examples above, by giving it the integer arguments 3 and 4. But instead of allowing it to print its output – the number 7, he hopes – to the screen, he has used the command redirection operator '>' to save the program's output to a new file called '3.4.txt'. There is nothing special about this sequence of characters. Bob could have chosen any file name he liked, but chose '3.4.txt' as a mnemonic to remind him that this file contains the (text) output of the sum of 3 and 4. When you start compiling different versions of the `calc` program later in this assignment, you can try running the command yourself and viewing its output.

```
echo 7 > 7.txt
```

The second line of the program does something similar to the first. This time, instead of running the `calc` program it is running the built-in program "echo" – whose job is merely to print its arguments to the screen – to place the number 7 in the file named '7.txt'. Instead of printing to the screen, we again redirect the output to a file. Try running echo yourself in the shell to see how it behaves (for example, 'echo hi' or 'echo "words back to the screen"').

```
diff 3.4.txt 7.txt
```

All of the action happens in this last line. Here, Brad uses the built-in program "diff" to compare the contents of the two files he has just created: 3.4.txt, which contains the sum of 3 and 4 as reported by the program being tested, and the number 7, the "true" sum of 3 and 4. If the contents of the two file are the same, then this program works correctly for this input. Victory! If not, we would like to know not only THAT they disagree, but how. Diff works in exactly this way: it prints nothing, and returns 0, when the contents of its inputs match. It returns something other than zero, and prints a description of how the two inputs differed, otherwise.

Because the invocation of diff is the last line in Brad's shell script, his script will return whatever diff returns. This means that to run this test, Brad can merely run his script. When he runs it against a program that correctly adds 3 to 4, he sees something like this:

```
> sh basic_addition.sh
```

Nothing happens! But this is expected, since the test passed, and diff had nothing to print because its input files were identical. He checks the return value of the script just to be sure:

```
> echo $?
0
```

OK, the test passed. As a sanity check, he temporarily rewrites his test script in a way that he expects to fail:

```
./calc 3 4 > 3.4.txt
echo 8 > 8.txt
diff 3.4.txt 8.txt
```

He runs the altered script:

```
> sh basic_addition.sh
1c1
< 7
---
> 8
```

Now he sees the expected output from diff. The input file 3.4.txt contains "7" but the file 8.txt disagrees, containing 8. Confident now that the test script works as intended and would catch a bug if one existed for these inputs, he undoes his tweak to the test script.

As the glow of triumph begins to fade, Brad ponders how many more tests he should write. He is not convinced that a calculator that always adds 3 and 4 correctly is necessarily a correct calculator. At the same time, he does not have time to write tests for every pair of natural numbers. Somewhere between these two extremes must lie a reasonable test plan.

Because he knows he will need it later, Brad builds a test harness – a special shell script that he can call to run all of his tests later. The basic logic of his test harness works like this:

- Run every test script (e.g., basic_arithmetic.sh is a single test script) in the shell. For each,
  - If it returns something other than zero (meaning it failed), exit and report an error.
- If all the test scripts return zero, return zero.

Do not worry so much about the details of this somewhat more sophisticated shell script, called "runner.sh". The important thing is that you can run it to automatically exercise the suite of tests that you will be writing for this assignment. We hope that you will follow similar patterns to test all the code you write in upcoming assignments.

4 The sample test script

One of the things that is specified in the design spec is the "return value" of a program. When a program in C ends, it is good practice to specify an return value that indicates whether a program has succeeded or failed. It is general convention to use 0 for success and any non zero value for failures. [2] This means that you will be checking a few things. First, the text outputted by the program, and second, the return value. As we briefly mentioned earlier, you can use the variable `$?` to find the return value of the last ran program. To incorporate this, let us write a more advanced test script.

```
# Creates the example output
echo 8 > 8.txt

# Runs the program
```

[2] It's actually quite a bit more complicated, look into "errno" for more information.

```
./calc 3 4 > 3.4.txt

# Ensures exit code is Zero
if [ $? -ne 0 ]; then
    echo "invalid exit code" $?
    rm 3.4.txt
    rm 8.txt
    exit 1
fi

# Ensures differences *are* found
diff 3.4.txt 8.txt
if [ $? -eq 0 ]; then
    echo "Somehow, the output of 3+4 is 8!"
    rm 3.4.txt
    rm 8.txt
    exit 1
fi

# Prints a message on success
echo "Test does not find that 3+4 = 8: PASS"

# Cleans up files created
rm 3.4.txt
rm 8.txt
exit 0
```

You may notice that this test script is much longer. This is because it does a few extra things.

- There are comments, which begin with `#` indicating what each line does
- It ensures that the exit code returned by the program is zero. [3]
- It prints descriptive message if an error is encountered (and if the test passes) [4]
- It produces its own return value using `exit`
- It uses if statements to figure out if things are or are not equal to zero. [5]
- It cleans up the files it creates afterwards. [6]

This extended script is now stored in `asgn0/tests/test_badmath.sh`. For this assignment, all the tests you create will be stored in that folder, and will follow the naming convention `test_SOMETHING.sh`.

5 Your Task

Your task in this assignment is to help Brad find which of the implementations of the bird counting programs follow the specifications. To do this, you will be writing tests, and run them on each of the provided programs. The first step in this is Help Brad write tests. Brad has found all the various versions of the code lying around on company servers and put them all in a folder. To avoid confusion, each version is named after the cat that authored it. Brad has also created one test case and a test harness called `runner.sh`, which he planned on using to test all the versions. Unfortunately, Brad was recently diagnosed with a bad case of eepy sleepies, and has to leave the rest to you.

[3] This is not going to be the case for every test!

[4] this is not required, but it might make your life easier

[5] Bash is very specific about if statements and for loops. It is recommended that you use this program as a starting point to avoid debugging syntax errors.

[6] This is not required, but is useful for cleanliness

5.1 Problem 1 (to be answered in the Design Plan):

Describe a test plan for the various implementations of calc. Given that the program cannot be tested on every possible input, what are three examples of tests that are implied by the spec but not checked by basic_arithmetic.sh?

5.2 Problem 2 (to be answered in the Design Plan):

You may have noticed that the spec doesn't say anything about what to do if the user supplies too many arguments at the command line. Unfortunately, ambiguity in specifications is very common in practice and can be very confusing for programmers. If you were asked to implement calc given this spec, you would be forced to choose from among at least three interpretations of what calc should do when called with, say, three arguments 3, 4, and 5:

- Calc should sum *all* of the arguments, and print 12.
- Calc should ignore the third argument, and print the sum of the first two: 7.
- Calc should print an error message and return non-zero.

Luckily for you, you are not the programmer (for now). Should your test scripts check what the program does when given more than two arguments? Why or why not?

5.3 Problem 3 (to be submitted in your repo)

Write a collection of shell scripts that test the various implementations of the calculator program. All of this should be done on your Linux system, and will later be submitted to GitLab. We have ways of knowing if you do not use Linux to write these tests. You may write as many scripts as you like, and may give them whatever names you like (though we recommend intuitive names that say something about what the script checks). Each script must work in the same way as basic_arithmetic.sh:

- The script can be run at the command line with no arguments.
- The script may assume that the calc program has been compiled, is in the current directory, and is named 'calc.'
- The script should test calc in some way by running it and checking its output.
- The script should communicate a successful test by returning 0, and a failed test by returning something other than zero. When a test fails, it is helpful to print something that describes the error.
- The script must be stored in the `tests` directory.
- The script must be named `test_SOMETHING.sh`

5.4 Finishing up (to be submitted in your repo)

Round out Brad's test suite by adding at least three more test scripts. You can test your test scripts by compiling the various implementations of `calc` that you can find in the `compiled_cats` folder. The simplest way to do this is to use the script `runner.sh`. You can do this by running `./runner.sh`. Assuming that you followed the instructions correctly, the script will output a list of programs and what tests they failed. You can tinker with the variables in the top of `runner.sh` if you want to get more or less information. When you submit your assignment, we will grade it in exactly the same way: by using your tests to check a variety of implementations of `calc` with a variety of bugs, and making sure that your tests both catch bugs and report when an implementation is correct.

Some sample implementations are included in the repository. Most of them contain at least one bug. Some of these bugs are easier to catch than others. Most of your functionality grade from this assignment will come from your ability to catch simple bugs. Some more complicated bugs that require more thinking will be worth extra credit.

That's it!

6 A note on git and submissions

Because this is the first assignment, we realize that some of you may be unfamiliar with git. We recommend that you start by reading this guide `https://13s-docs.jessie.id/usage/git_setup.html`, so you know how git works and how to use it. Please read this guide in its entirety as it will help you throughout this class. There are other wiki pages that may help you as well. Please read it closely.

Finally, be sure to commit and push often, as you are very likely to lose progress if you don't.

6.1 A note on architectures

Because of Apple's switch to ARM cpus, we are forced to lose some consistency. You may notice that we often have to have different instructions for arm and x86 systems. This will be especially important when running any code that we compile for you. In this assignment, that's the `compiled_cats` folders. For future assignments, that will be the example binaries that we may give you. We will always give you x86 and arm versions of all compiled code to make this easy for you, but it is your job to figure out which one you need to run on your computer.

6.2 How to go about this assignment

By this point, you should already have your VM set up with SSH access as well. This will allow you to do things such as copy paste and change font size on your terminal. To access your files, you should do the following (this will not change very much between assignments [7], so get used to this procedure):

1. Navigate to git.ucsc.edu. You should see at least 2 projects, one with your cruzid for this quarter and one called "resources".
2. Clone the resources repository on your virtual machine.
3. In a separate folder, clone your personal repository. Note that it is a bad idea to clone one git repository inside another. Bad things will come of that.
4. Navigate to the resources repository and move the needed files to your personal repository. You will be submitting most of the code that we provide in your git repository. See the deliverables section of each assignment for more info.
5. Using any tool that can create a PDF, begin working on your design. Be sure to name it `design.pdf`. Do not name it anything else.
6. Begin working on your code. The name for this will change from assignment to assignment. For this assignment, we simply want your tests to be `.sh` files which begin with the word "test".
7. When the design is due, commit and push it to git. See canvas for how that is submitted.
8. When the assignment is due, commit, push and submit to canvas.

6.3 Deliverables

For this assignment, you must submit (in your git repository)

1. Both `compiled_cats` folders.
2. `runner.sh`
3. The `tests` folder with all your tests (be sure that they follow the naming convention)
4. `design.pdf`

[7] but you wont have to do the first 3 steps again