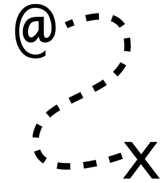
15-122: Principles of Imperative Computation, Spring 2020

Programming Homework 1: Scavenger hunt



Due: Thursday 23rd January, 2020 by 10pm

Welcome to 15-122! This first programming homework is designed as an opportunity for you to tour the course tools and workflow that we will use in 15-122.

The assignment takes the form of a scavenger hunt; you will add pieces of code to a file, `scavhunt.c0`, and then turn this file in through Autolab. Your submission should be made through the web interface of Autolab. To do so, please create a zipped tarball (`tgz`), for example:

```
% tar czvf handin.tgz scavhunt.c0 puzzle.c0
```

and then upload it to Autolab. If you are working remotely through ssh, you will have to transfer the handin.tgz file to your computer in order to upload it using the web browser.

Your score for this assignment will be the score of your last Autolab submission.

© Carnegie Mellon University 2020

1 Obtaining the handout code

Task 1 (2 points) Obtain the handout file `scavhunt.c0`, containing a function `greet`.

The handout code for this assignment is on Autolab under

Programming > Scavenger Hunt > Download Handout

Clicking on the link will download a gzipped tarball file called `scavhunt-handout.tgz` which you need to unpack. **Note that some browsers uncompress downloaded file.**

IMPORTANT: if you are using Andrew AFS, make sure that you only put code for 15-122 into your `private` directory or into another directory with appropriate AFS permissions set. Failing to do this will cause you to run afoul of the course's academic integrity policy.

```
% cd $HOME/private/15122
% tar xzvf scavhunt-handout.tgz
scavhunt-handout/
scavhunt-handout/puzzle.c0
scavhunt-handout/puzzle-test.c0
scavhunt-handout/scavhunt-main.c0
scavhunt-handout/scavhunt.c0
% cd scavhunt-handout
% ls -la
drwxr-xr-x 2 iliano users 2048 Aug 16 17:30 .
drwxr-xr-x 3 iliano users 2048 Aug 16 17:48 ..
-rw-r--r-- 1 iliano users 1587 May 16  2015 puzzle.c0
-rw-r--r-- 1 iliano users 1155 Jan 15  2015 puzzle-test.c0
-rw-r--r-- 1 iliano users  405 Jan 12  2015 scavhunt.c0
-rw-r--r-- 1 iliano users  196 Sep  1  2014 scavhunt-main.c0
```

You can't use `scavhunt-main.c0` yet, but you can test the `greet` function contained in `scavhunt.c0` using `coin`.

```
% coin -d scavhunt.c0
C0 interpreter (coin) 0.3.3 'Nickel' (r590, Mon Aug 29 12:04:13 UTC 2016)
Type '#help' for help or '#quit' to exit.
--> greet("Hello", "world");
"Hello, world!" (string)
--> println(greet("Hello", "world"));
Hello, world!
(void)
--> print(greet("Hi", "friend"));
(void)
--> print("\n");
Hi, friend!
(void)
```

© Carnegie Mellon University 2020

This demonstrates something important (and potentially confusing) about C0 programs. When you use the `print` function to print strings, the output is *buffered* and does not generally get printed out until a newline is printed. The escape sequence `\n` represents a newline, so printing the string `"\n"` empties the buffer, printing `Hi, friend!`.

2 Using the C0 tutorial

Task 2 (2 points) The C0 tutorial's page on "Statements" contains the code for a C0 file named `fact.c0`. Copy all the contents of this file (it's just one function) into `scavhunt.c0`.

The C0 tutorial at `http://c0.typesafety.net/` will help with early assignments. The point about buffered output above is explained on the page "Debugging C0 Programs" in the C0 tutorial.

3 Viewing images from AFS

Task 3 (2 points) The file `/afs/andrew/course/15/122/misc/scavhunt/snippet.png` is an image file that, when viewed, contains a C0 function. Copy the code in that image into `scavhunt.c0`.

For the next two programming assignments, it will be very helpful for you to already know how to view images that live on AFS on your own computer. The PNG file above, which is publicly readable when you're logged into a Linux cluster machine or connected with SSH to `unix.qatar.cmu.edu`, can be viewed using a program like `display`. To use one of these programs, you will either need to be on a Linux machine or you will need to use `ssh -X` to log on to `unix.qatar.cmu.edu` with X11 forwarding (if this doesn't work, try `ssh -Y`). You can also copy the file by transferring it from AFS to your computer with the `scp` command-line program or with a program like FileZilla and viewing the image with whatever built-in image-viewing software your operating system uses. Play around and find a method you like.

Once you have added the function from `snippet.png`, you can use the `cc0` compiler and the provided `scavhunt-main.c0` program to compile and run your scavenger hunt code.

```
% cc0 -d -o scavhunt scavhunt.c0 scavhunt-main.c0
% ./scavhunt
```

The `-d`, which also appeared in the call to `coin`, makes sure contracts are checked. The arguments `-o scavhunt` tells the compiler to produce an executable file named `scavhunt`. You could omit this argument and the executable file would be named `a.out`.

```
% cc0 -d scavhunt.c0 scavhunt-main.c0
% ./a.out
```

© Carnegie Mellon University 2020

4 Updates, clarifications and questions on Diderot

Task 4 (3 points) Modify the function from `snippet.png` as described in the "How to use Diderot (effectively)" post on 15-122 Diderot.

Diderot can be found at `https://www.diderot.one`. Please read through the whole "How to use Diderot (effectively)" post, as it explains some of the guidelines for how we will be using Diderot.

5 Getting feedback from Autolab

To proceed, you'll need to **submit the work you have so far to Autolab**. The instructions for doing this are on the first page of this writeup. Autolab always allows multiple submissions, and your **last submission** is the one we count. For this assignment, there is no limit to the number of times you are allowed to submit (but there will be in future homework).

Task 5 (4 points) Add a function (that takes no arguments and returns a string) to `scavhunt.c0`. It does not matter what string this function returns.

You'll be able to figure out what the name of the missing function should be by looking at the output that Autolab gives you. The autograder we use for 15-122 doesn't give a ton of feedback, but it does give some feedback when tests fail. First, it says why the test failed (for instance, `File did not compile`), and then it gives a hint. These hints aren't perfect, and they can sometimes be unintentionally misleading. (Usually, this is because of a mistake that we didn't anticipate, or because an error in an earlier part of the assignment triggered another error later on.)

Also, the hints almost always assume that the test compiled, so if the test reports that the file did not compile, you should look to the first part of the autograder's input to see the output from the C0 compiler and get an idea of what went wrong, rather than paying attention to the hint. You'll need to do this to learn the name of the mystery function needed to complete this assignment. *Don't make a habit of doing things this way.* Autolab is not your compiler, and you should usually test your code before you hand in your work. In fact, future programming assignments will limit the number of submissions you are allowed.

There are two other critical features in Autolab. First, the *Gradebook* link lets you see your performance, including the number of late days you have used. Second, you can view the code you handed in with Autolab; when there's a manually-graded piece of a programming assignment you'll get comments from TAs on Autolab.

© Carnegie Mellon University 2020

6 Puzzle hunt

This last task will require you to think a little bit harder, and also read through the C0 tutorial page on Strings and Characters at `http://c0.typesafety.net/`, as well as the C0 library documentation. You must implement three functions:

```
int f(string s1, string s2)
//@ensures 0 <= \result && \result <= string_length(s1);
//@ensures 0 <= \result && \result <= string_length(s2);
//@ensures string_equal(string_sub(s1, 0, \result), string_sub(s2, 0, \result));
/*@ensures \result == string_length(s1)
  @     || \result == string_length(s2)
  @     || string_charat(s1, \result) != string_charat(s2, \result); @*/

int g(string s)
//@requires string_length(s) > 0;
//@requires string_charat(s, 0) != string_charat(s, string_length(s) - 1);
//@ensures 0 <= \result && \result < string_length(s)-1;
//@ensures string_charat(s, \result) == string_charat(s, 0);
//@ensures string_charat(s, \result+1) != string_charat(s, 0);

string h(string s)
//@ensures is_substring(\result, s);
/*@ensures string_length(s) < 128
  @     || (string_length(\result) > 1
  @         && string_charat(\result, 0)
  @            == string_charat(\result, string_length(\result) - 1)); @*/
```

Note: some of these postconditions rely on *short-circuiting evaluation*, which is discussed in the C0 tutorial page on Booleans and in the first recitation notes.

It is possible to write loop invariants to prove that each function satisfies its postcondition. You aren't required to do that for this assignment, but you are encouraged to try! You will definitely want to write test cases for these functions and run them; it's a good idea to enable contract checking when you do so:

```
% cc0 -d -o puzzle puzzle.c0 puzzle-test.c0
% ./puzzle
```

For the first function, some test cases are provided. For the second function, the autograder's feedback will indicate what string your function failed on. For the third function, which involves finding two characters in the string that are the same, you'll need to think through test cases on your own. You might want to look up the *pigeonhole principle* to understand why the postcondition for that function looks like it does. (Hint: there are 127 different characters that can appear in C0 strings.)

Task 6 (12 points) Fill in the three functions in `puzzle.c0`. Any implementation that always satisfies the postconditions will be accepted, even though there may be multiple correct implementations that give different answers.

© Carnegie Mellon University 2020