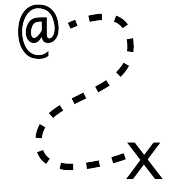


15-122: Principles of Imperative Computation, Spring 2025

Programming Homework 1: Scavenger Hunt



Due: Sunday 26th January, 2025 by 9pm EST

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 three files, `scavhunt.c0`, `compilation.c0` and `puzzle.c0`, and then turn these files in through [Autolab](https://autolab.andrew.cmu.edu/courses/15122-s25/) (<https://autolab.andrew.cmu.edu/courses/15122-s25/>). There are two ways to do so. From the terminal on Andrew Linux (via cluster or ssh) type:

```
% autolab122 handin scavhunt scavhunt.c0 compilation.c0 buggy.c0 puzzle.c0
```

Once [Autolab](https://autolab.andrew.cmu.edu/courses/15122-s25/) is done grading it, your scores will be displayed. To view error messages, type

```
% autolab122 feedback
```

Your submission can also be made through the web interface of [Autolab](https://autolab.andrew.cmu.edu/courses/15122-s25/). To do so, please create a zipped tarball (`tgz`), for example:

```
% tar -czvf handin.tgz scavhunt.c0 compilation.c0 buggy.c0 puzzle.c0
```

and then upload it to [Autolab](https://autolab.andrew.cmu.edu/courses/15122-s25/). There, you can also see scores and feedback of your last submission.

For this assignment, you can make as many submissions as you want. Future programming assignments will have a limit to the number of submissions you can make before incurring a grade penalty.

Your score for this assignment will be the score of your last [Autolab](https://autolab.andrew.cmu.edu/courses/15122-s25/) submission. The same will be true of all future assignments.

1 Obtaining the handout code

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

To do so, go to your `private/15122` directory

```
% cd ~/private/15122
```

and type¹

```
% autolab122 download scavhunt
```

This will create a subdirectory called `scavhunt` containing this file (and a few others):

```
% ls scavhunt
```

```
puzzle.c0  puzzle-test.c0  README.txt  scavhunt.c0  scavhunt-main.c0
```

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.

In all assignments, you will find the file `README.txt`. It contains instructions for submitting and testing, as well as an overview of the files in this assignment. We recommend referring back to this file as you complete the assignment.

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

```
% cd scavhunt
```

```
% coin -d -lconio scavhunt.c0
```

```
C0 interpreter (coin) 1.0.0 'Quarter' (v20210822, Sun Aug 22 11:15:15 EDT 2021)
```

```
Type '#help' for help or '#quit' to exit.
```

```
--> greet("Hello", "world");
```

```
"Hello, world!" (string)
```

```
--> printf("%s\n", greet("Hello", "world"));
```

```
Hello, world!
```

```
--> #quit
```

```
Goodbye!
```

Here, “`-lconio`” allows you to use printing functions in `coin`.

When you use `printf` (or most other C0 printing functions), the output is *buffered* and does not generally get printed out until a newline is printed. If you do not end the text you want to print with the escape sequence `\n`, which represents a newline, code compiled with `cc0` will not print it out.

¹You can also get the handout from the [Autolab](#) web site by clicking *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 files.**

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 <https://bitbucket.org/c0-lang/docs/wiki/Tutorial> will help with early assignments. The point about buffered output above is explained on the page entitled "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` (notice the leading `'/'`) 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. You can easily do this using a program like `display` if you are logged into a Linux cluster machine or connected with SSH to `unix.andrew.cmu.edu` (see the posts entitled "Laptop Setup for Mac/Windows/Linux" on [Ed](#) for how to do this). If you use VSCode with the `ssh` or `sftp` extension, you can copy the image to your `~/private/15122` directory using the `cp` command, and then open it like any other file. Alternatively, you can transfer the image from AFS to your computer with the `scp` command-line program or with a program like WinSCP, and then view it with whatever built-in image-viewing software your operating system uses. Play around and find a method you like.²

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
```

²Note that some of these methods may take a few seconds or minutes on a slow Internet connection.

4 Updates, clarifications and questions on Ed

Task 4 (2 points) Modify the function from `snippet.png` as described in the “How to use Ed (effectively)” post on 15-122 Ed.

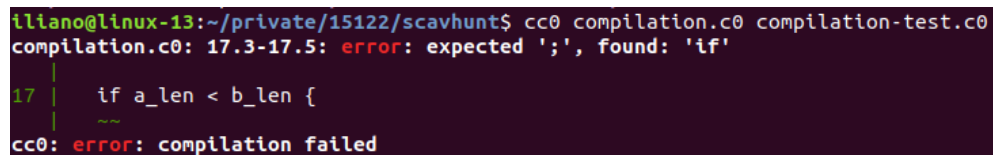
Ed can be found at <https://edstem.org/>. Please read through the whole “How to use Ed (effectively)” post, as it explains some of the guidelines for how we will be using Ed.

5 Fixing Syntax Errors

In the next task, you will be learning how to read and fix local compilation errors. Specifically, you will be making additions to the file `compilation.c0` so that it compiles. Let’s compile this file together:

```
% cc0 compilation.c0 compilation-test.c0
```

Because the file has issues, this command will display the following compilation error:



```
liiano@linux-13:~/private/15122/scavhunt$ cc0 compilation.c0 compilation-test.c0
compilation.c0: 17.3-17.5: error: expected ';', found: 'if'
17 |   if a_len < b_len {
cc0: error: compilation failed
```

The compilation error gives you a lot of handy information for figuring out what is wrong.

- It tells the name of the file where the compilation error occurred, here `compilation.c0`.
- It also tells the line number where `cc0` noticed the problem, in this case `17.3-17.5`. This means the issue starts on line 17 column 3 up to line 17 column 5 — sometimes the reported line number is a few lines after where the issue actually is.
- Next, it tells what the error is, in this case “`expected ';', found 'if'`” which means that `cc0` expected to see a semicolon but ended up finding an `if`. This says we are missing a semicolon. If we look at the previous line of code, line 15 (since line 16 is empty), we find that indeed we were missing a semicolon.

Go ahead and add a semicolon at the end of line 15.

Task 5 (2 points) Identify and fix the remaining syntax issues in this file by repeatedly compiling it and using the error message to find and correct the next issue. Once there are no more compilation errors, you have successfully completed this task.

Do compile the files even if the fix is obvious from a quick glance at the code. In this way, you will practice reading compilation messages, which will help you deal with your own compilation errors in future programming assignments.

All your fixes should be **additions** to the file `compilation.c0`. In particular, you should **not** be deleting anything. In the event you do something wrong and don’t know how to undo it, you can go back to the original `compilation.c0` by running the following command:

```
% cp compilation-backup.c0 compilation.c0
```

6 Satisfying library function contracts

Now that `compilation.c0` is free of syntax error, let's run the fruit of your labors:

```
% ./a.out
```

This will abort with a safety failure caused by the call to the library function `string_charat`.

Task 6 (1 point) Find what caused this issue by looking up the contracts of `string_charat` in the [C0 library documentation](https://c0.cs.cmu.edu/docs/c0-libraries.pdf) at <https://c0.cs.cmu.edu/docs/c0-libraries.pdf>. Then, fix it by deleting a single character in function `fixme_later`.

Once you are done, compile and execute this code again. It should now run to completion and display a friendly message.

7 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 7 (2 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. However, the hints almost always assume that the test compiled, so if the test reports that the file did not compile, you should **look at the top part of the autograder's feedback** (you will need to scroll up, if viewing it in the terminal) to see the output from the C0 compiler and get an idea of what went wrong, rather than paying attention to the hint.

Don't make a habit of doing things this way. Autolab is not your compiler, and you should test your code before you hand in your work. In fact, future programming assignments will limit substantially the number of submissions you are allowed.

There are two other critical features in the [Autolab website](https://autolab.andrew.cmu.edu/courses/15122-s25/), found at <https://autolab.andrew.cmu.edu/courses/15122-s25/>.

- 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.

8 Debugging code

Doing a programming assignment (and any programming task in general) involves two things: developing a conceptual solution and fixing all the bugs that crop up when translating it into code. This last aspect — debugging code — can take many frustrating hours if approached wrong.

In this course, we will help you develop efficient debugging habits. As a preview, the next task asks you to debug a relatively simple program. Here's how you **should** go about it:

- Using `coin`, call a suspect function on various inputs both to identify inputs that lead to incorrect outputs and later to check that your edits fix the bugs you identified. These inputs are called *test cases*.³
- Add print statements to display the value of variables and other expressions of interest on your test cases — remember to print a new line character (`\n`) before it returns! This is called *print debugging*.
- Add contracts, often `@assert` statements, to check that the value of key expressions is what you expect it to be. Make sure to enable contract checking by passing the `-d` flag to `coin`. This is called *contract debugging*.

One thing you should not do: stare at the code until you “see” the bug. This may work for tiny programs but, for the kind of code you will see in this course and beyond, it will lead to many wasted hours with nothing to show for them.

Task 8 (3 points) The code in file `buggy.c0` has three bugs. Each can be fixed by editing a single line of code (for a total of three one-line changes). Using any of the techniques above (except staring at the code), find these bugs and fix them.

For more about debugging, see our *Guides to Success* about [Debugging C0 Code](#) on the [course web site](#).

³We will see later in this course how to run test cases using `cc0`.

9 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” in the [C0 tutorial](#), as well as the [C0 library documentation](#) at <https://c0.cs.cmu.edu/docs/c0-libraries.pdf>. 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: <https://bitbucket.org/c0-lang/docs/wiki/Booleans>.

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 in `puzzle-test.c0`. 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 9 (9 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.