15-122: Principles of Imperative Computation

Lab Week A, Friday — Nivedita Chopra, Rob Simmons

Setup: Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab-c .
% cd lab-c
```

You should write your code in the new files `hello.c` and `rollcall.c` in the directory `lab-c`.

Grading: This is a self-paced lab and is not for credit.

We strongly recommend finishing these tasks during or after lab, as they will help you better understand C and write code in it.

Hello, World!

Today's lab involves writing some code in C. Throughout this handout, please refer to the tutorial at `http://c0.typesafety.net/tutorial/From-C0-to-C:-Basics.html`

(1.a) The first program to try out in any programming language is one that prints out "Hello World!" (or any other string of your choice), so that you know that you are able to compile and run programs.

Create a file called `hello.c` and write some valid C code that prints out the string "Hello World!" You can refer to the file `hello.c0` for the equivalent C0 code. You'll need to use the `stdio` library for printing.

Here's the C0 version of the file you're rewriting:

```
#use <conio>
int main(){
  print("Hello World!\n");
  return 0;
}
```

If your code works, you can compile it and test it as follows:

```
% gcc -Wall -Wextra -Werror -Wshadow -std=c99 -pedantic -g hello.c
% ./a.out
Hello World!
%
```

C programs won't print out the value returned from `main()` the way C0 programs do.

Return of the roll call example

In this part, we're checking on the attendance records of various students. All the code you're writing in C is provided in C0 form in the file `rollcall.c0`, which you should refer to as you do these tasks.

(2.a) In a new file, `rollcall.c`, represent each student as a struct with fields `andrew_id` (a string), `attendance` (a boolean array) and `num_days` (a `size_t` representing the length of the array). Look at `rollcall.c0` for the C0 version of the code this struct.

(2.b) Write a function `student_new` that takes in an andrew id and a string of letters (either 'T' or 'F'). The function creates a new student struct, assigns the appropriate values, and returns the newly created struct.

(2.c) Write a function `count_present` to count the number of days that a given student was present.

(2.d) Translate the main function in `rollcall.c0` to valid C code in `rollcall.c`. Don't worry about freeing allocated memory just yet.

You should now be able to compile and run your code:

```
% gcc -Wall -Wextra -Werror -Wshadow -std=c99 -pedantic -g xalloc.c rollcall.c
% ./a.out
foobar : 4/6
acarnegie : 3/3
cobot : 3/4
scotty : 2/5
bovik : 0/0
%
```

This looks good! But if you didn't free the memory you allocated, you'll see memory leaks (hopefully not memory errors) when you run the code with `valgrind`:

```
% valgrind ./a.out
<many lines of mumbo-jumbo>
% valgrind --leak-check=full ./a.out
<even more lines of mumbo-jumbo>
```

(2.e) Try to match some of the line numbers reported by `valgrind` to lines in your C code. If you don't see line numbers, make sure you added the -g flag when compiling with gcc.

(2.f) Free all the memory you've allocated right before the `return 0` in `main()`. Re-compile and re-run with `valgrind` to confirm that you got it right.

Valgrind will report "All heap blocks were freed – no leaks are possible" if you get it right. Don't worry about leaks that are marked as "suppressed."

Hashing C strings

Recall that the Java hash function is:

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

(3.a) If we were to store students in a hash table, we would need a hash function for each student. Using the fact that strings in C are character arrays terminated by a `'\0'` character, write a hash function that performs the Java hash on an appropriate field in the student struct. Here is the header for this function:

```
int hash(stu* s);
```