Computer Science 50 | Week 2 Monday: September 14, 2009
Fall 2009 | Andrew Sellergren
Scribe Notes

Contents

1 Announcements (0:00–10:00)

- This is CS 50.
- Check out our first foray into cryptography.
- Know that the `root` password for CS 50's servers is a quadruple ROT13 encryption of the string "password." If you can even *begin* to understand what that means, you just might be able to hack into our servers. Godspeed.
- 1 new handout.
- Scribe notes are available for your convenience.[1]
- 9 late days are available to you throughout the semester. So if you need a 24-hour extension on any given problem set, just e-mail your TF to let him know that you'll be cashing one in. We encourage you to be conservative with them at the start of the course since the problem sets tend to get exponentially harder.
- Supersections on Monday, Tuesday, and Wednesday. For this first week, we'll simply be holding large sections which anyone is welcome to attend. Our first walkthrough was held last night and was recorded and will be placed online. The walkthroughs are meant to be authoritative how-tos for the problem sets. Section assignments will be sent out this week. If you have conflicts with your assigned section time, please feel free to get in touch with us to resolve them.
- OHs and VOHs have begun! Office Hours are a long-standing tradition of 50 and are held in the basement of the Science Center in the Terminal Room where TFs cross names off the whiteboard and do their best to handle everyone's questions. Virtual Office Hours are in the same spirit, only a different location. Login to the VTR if you feel that your question needs a little more interactive help from the TFs and you don't feel like putting pants on to leave your dorm room.
- Soon the bulletin board will be active, but in the meantime feel free to send your questions to `help@cs50.net`. Just be sure to be very specific in your descriptions of the problems since we do get a **lot** of these e-mails.
- If you did the Hacker Edition of Problem Set 1, be sure to return your sensor board after class and write your name on the slip of paper.
- Lunch with David! This tradition began last year and was funded by Harvard, but even though Harvard's days of wine and roses have come to an end (for now), some generous alumni have stepped in to foot the

[1]Wow, so meta. It's like holding up a mirror to another mirror. Or one of the acronyms that never ends. Nova of Virginia Aquatics (NOVA)! Alright I'll stop now.

bill. If you're interested in joinining David and a handful of others in an opportunity to make a large class much more intimate, then go here to RSVP. Lunch with David will be held Fridays at 1:15 PM.

- Thanks to Harvard Computer Society (HCS) for holding their Demystifying Linux seminar!

2 More About the Course (10:00–23:00)

- We know where you are. If you can't find yourself, please don't panic and e-mail David. Our large number of requests seems to have annoyed Google to the point where they might not be showing you, even if you properly submitted your problem set.
- Check out the galleries of Scratch projects past!
- Wow, I can't believe I managed to be in the VTR the same time as David. If video had been enabled, you would've seen me lying in bed in my boxers with cookie crumbs all over my stomach. My life is. . . well let's not talk about it.
- So, why did you decide to take CS 50? A few funny answers:
  - "I feel like an idiot when I use computers"
  - "Have never taken CS 50"
  - "idk. lulz. Programmers are 1337"
  - "SO I CAN HACK THE PLANET"
- The course policy on academic honesty is **very** clear and we take it **very** seriously. Suffice it to say that collaborating with a friend while speaking in English is perfectly acceptable, but holding a conversation in C or sharing code via e-mail is not acceptable. See the Academic Honesty section of the syllabus and on the second page of every problem set. We have our computer savvy to help us find similarities between problem sets, so don't take a stupid risk. We've sent 25 students to the Ad Board in the past few years. None of them enjoyed it, I can assure you. When you have questions, please ask us. And when you are pressed for time, please use your late days instead of taking shortcuts!
- On a lighter note, some fun facts about CS 50:
  - Most of you have a "normal" mobile phone.
  - None of you is using Nextel, because as everyone but David apparently knew, they merged with Sprint.
  - The House with the highest enrollment is Mather.[2]

[2]Mather : Winthrop :: a child on a tricycle : Lance Armstrong.

  - Most are taking the course for concentration credit, but almost as many are taking it as an elective.
  - And, yse, most are you are "somewhere in between" or "among those less comfortable." And most of you have taken 0 CS courses previously!

- This course isn't about learning any programming language in particular. Although we will teach you the basics of C, PHP, and JavaScript, our hope is that you will be empowered to learn any language you want at course's end. This is because all of the deep-rooted concepts of programming are independent of the language they are written in.

3 Debugging and Casting (23:00–46:00)

- As the story goes, Grace Hopper, a well-known computer scientist of her time, discovered an actual moth in the inner workings of the Mark II, which was housed at Harvard's Computation Library in 1947. Truth be told, however, the term "bug" goes back farther than this, but, hey, it makes for a nice story.

- Can you figure out the bug in this program?

```
/****************************************************************************
 * buggy1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Should print 10 asterisks but doesn't!
 * Can you find the bug?
 ***************************************************************************/

#include <stdio.h>

int
main(int argc, char *argv[])
{
    int i;

    for (i = 0; i <= 10; i++)
        printf("*");
}
```

  Well, it prints 11 asterisks instead of 10 because the termination condition is `i <= 10` rather than `i < 10`. How about `buggy2.c`?

```
/****************************************************************************
 * buggy2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Should print 10 asterisks, one per line, but doesn't!
 * Can you find the bug?
 ***************************************************************************/

#include <stdio.h>

int
main(int argc, char *argv[])
{
    int i;

    for (i = 0; i <= 10; i++)
        printf("*");
        printf("\n");
}
```

The second `printf` statement is not executed within the scope of the loop because no curly braces are placed around it and the first `printf` statement! Once we add the curly braces and recompile, we'll get one asterisk per line as we intended.

- Recall that all data in computers is stored in the form of zeroes and ones. These zeroes and ones map to actual transistors which are either on or off depending on if a switch is flipped that is either allowing or preventing electricity from flowing. Thankfully, we can put several of these binary bits together to make a byte which we can use to represent letters via the ASCII mapping. To actually access the numerical value of a character according to this mapping, we can use *typecasting*:

  ```
  int i = (int) 'A';
  char c = (char) 65;
  ```

  Here we're storing a capital A as an `int` and the number 65 as a `char`. What if we want to access the ASCII mapping of the whole alphabet? We can do so using our `ascii1.c` program:

  ```
  /****************************************************************************
   * ascii1.c
   *
  ```

```
 * Computer Science 50
 * David J. Malan
 *
 * Displays the mapping between alphabetical ASCII characters and
 * their decimal equivalents using one column.
 *
 * Demonstrates casting from int to char.
 ***************************************************************************/

#include <stdio.h>


int
main(int argc, char *argv[])
{
    int i;

    // display mapping for uppercase letters
    for (i = 65; i < 65 + 26; i++)
        printf("%c: %d\n", (char) i, i);

    // separate uppercase from lowercase
    printf("\n");

    // display mapping for lowercase letters
    for (i = 97; i < 97 + 26; i++)
        printf("%c: %d\n", (char) i, i);
}
```

65 and 97, you'll recall, are the two mappings (for uppercase A and lowercase a, respectively) that it's useful to remember. Beginning with these, we'll loop through the whole alphabet (incrementing `i` 26 times). Then we're printing a `char` followed by an `int`. The `char` placeholder is being filled in with our `int i` cast as a `char`.

- Our next program does the same, but with a little better presentation:

```
/****************************************************************************
 * ascii2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Displays the mapping between alphabetical ASCII characters and
 * their decimal equivalents using two columns.
 *
 * Demonstrates specification of width in format string.
 ***************************************************************************/

#include <stdio.h>


int
main(int argc, char *argv[])
{
    int i;

    // display mapping for uppercase letters
    for (i = 65; i < 65 + 26; i++)
        printf("%c  %d    %3d  %c\n", (char) i, i, i + 32, (char) (i + 32));
}
```

Notice the third column format string is `%3d`. This means print the decimal value in three spaces even if the number is only double-digit or single-digit. Also, we've combined two loops into one by realizing that the lowercase characters are all offset by exactly 32 from the uppercase characters in our ASCII mapping.

- Typically, a terminal window will be 80 characters wide by 24 characters tall. This will have significance as you try to build your Mario pyramid.
- `ascii3.c` shows that we can iterate over characters rather than numbers:

```
/****************************************************************************
 * ascii3.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Displays the mapping between alphabetical ASCII characters and
 * their decimal equivalents.
 *
 * Demonstrates iteration with a char.
 ***************************************************************************/

#include <stdio.h>


int
main(int argc, char *argv[])
{
    char c;
```

```
        // display mapping for uppercase letters
        for (c = 'A'; c <= 'Z'; c = (char) ((int) c + 1))
            printf("%c: %d\n", c, (int) c);
}
```

What's the deal with the update condition? Recall that `i++` is equivalent to `i = i + 1`. So for this update condition, we're casting `c` to an `int` in order to increment it. Then we're casting the result back to a `char` so we can reassign it to `c`. This is simply for clarity's sake.[3] It's not actually necessary, as we'll see in a moment!

- Let's make things a little more interesting by implementing the Battleship gameboard:

```
   1  2  3  4  5  6  7  8  9  10
A  o  o  o  o  o  o  o  o  o  o
B  o  o  o  o  o  o  o  o  o  o
C  o  o  o  o  o  o  o  o  o  o
D  o  o  o  o  o  o  o  o  o  o
E  o  o  o  o  o  o  o  o  o  o
F  o  o  o  o  o  o  o  o  o  o
G  o  o  o  o  o  o  o  o  o  o
H  o  o  o  o  o  o  o  o  o  o
I  o  o  o  o  o  o  o  o  o  o
J  o  o  o  o  o  o  o  o  o  o
```

  We'll compile it using the `make` command and then run it by writing `./battleship`. This is a security precaution, in fact. If you SSH to a server you don't know much about and you want to run a program called `foo`, what if you type `foo` at the command line and it executes a different, malicious version of that program stored elsewhere rather than the one in your directory that you were hoping to execute?

- What we're doing is just the tip of the iceberg of what's known as ASCII art among geeks. Just Google it. Don't say you weren't warned.[4]

- Before we walk through the code, let's think how we might approach to writing the program. Notice that over the last few weeks, we've only been able to print to the screen from left to right and top to bottom. So to begin, we'll probably have to print out that first row of numbers, which shouldn't be too hard. The middle of the gameboard isn't too hard, either, since we just need to print 10 lowercase o's in a row. But what about that first column of letters? Let's take a look at the code:

[3]Oh yeah, real clear, huh.
[4]Enjoy.

```
/****************************************************************************
 * battleship.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints a Battleship board.
 *
 * Demonstrates nested loop.
 ***************************************************************************/

#include <stdio.h>


int
main(int argc, char *argv[])
{
    int i, j;

    // print top row of numbers
    printf("\n   ");
    for (i = 1; i <= 10; i++)
        printf("%d  ", i);
    printf("\n");

    // print rows of holes, with letters in leftmost column
    for (i = 0; i < 10; i++)
    {
        printf("%c  ", 'A' + i);
        for (j = 1; j <= 10; j++)
            printf("o  ");
        printf("\n");
    }
    printf("\n");
}
```

Take a look at the second `for` loop. Notice we could've started at 1 and iterated *through* 10, but we chose to start from 0. This is handy in the first line when we use `i` as an offset. `'A' + 0` gives us `A`. If we forget about printing the lowercase o's, then we can take one task at a time. This isn't so bad! Now, on to functions.

4 Functions (46:00–41:00)

- Remember that Simpsons episode where Otto listens to the 99 Beers on the Wall song on his Walkman? Well this next programming task is *almost* as annoying as Everybody Loves Raymond![5]

- The fact that this song has a definite pattern gives us an opportunity to implement it by means of code which can be executed multiple times. You might be thinking loop at this point, but we're going to go one step further now and introduce *functions* to help get the job done. Let's take a look at beer1.c:

```
/****************************************************************************
 * beer1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Sings "99 Bottles of Beer on the Wall."
 *
 * Demonstrates a for loop (and an opportunity for hierarchical
 * decomposition).
 ***************************************************************************/

#include <cs50.h>
#include <stdio.h>


int
main(int argc, char *argv[])
{
    int i, n;

    // ask user for number
    printf("How many bottles will there be? ");
    n = GetInt();

    // exit upon invalid input
    if (n < 1)
    {
        printf("Sorry, that makes no sense.\n");
        return 1;
    }

    // sing the annoying song
    printf("\n");
    for (i = n; i > 0; i--)
    {
        printf("%d bottle(s) of beer on the wall,\n", i);
```

[5]Yes, those thoughts were related. Think about it.

```
            printf("%d bottle(s) of beer,\n", i);
            printf("Take one down, pass it around,\n");
            printf("%d bottle(s) of beer on the wall.\n\n", i - 1);
        }

        // exit when song is over
        printf("Wow, that’s annoying.\n");
        return 0;
    }
```

As we have been frequently now, we’re including the CS 50 library as well as `stdio.h`. Now we’re going to ask the user for an integer input. After that, for the first time, we’re going to do some error-checking. If the user gives us 0 or a negative number as input, we’re going to exit out. Probably not the best solution—it might be better to reprompt—but at least we’re handling the error case. If we didn’t, it might wreak havoc in our program. Best practice is to assume that your users are both idiotic and malicious. It’s a beautiful world, isn’t it?

- Notice that in the case of a bad input, we execute the statement `return 1`. We’ve mentioned the ability of programs to return values and, in fact, `main` is no different. What’s the use of this? Well, if you’ve ever gotten an error in a program you were running,[6] you’ve likely seen an error code to accompany it. This is similar in spirit to the exit code of our `main` program. By default, if everything executes successfully in `main`, your program returns the value 0.

- Now we’re just looping through the verses of the song, counting down from the positive number that the user provided. We can do this equivalently using a `while` loop rather than a `for` loop as we do in `beer2.c`:

```
/****************************************************************************
 * beer2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Sings "99 Bottles of Beer on the Wall."
 *
 * Demonstrates a while loop (and an opportunity for hierarchical
 * decomposition).
 ***************************************************************************/

#include <cs50.h>
```

[6]Actually, scratch that, if you *haven’t* ever gotten an error, please come find me. I just want to meet the man that God likes more than me.

```
#include <stdio.h>


int
main(int argc, char *argv[])
{
    int n;

    // ask user for number
    printf("How many bottles will there be? ");
    n = GetInt();

    // exit upon invalid input
    if (n < 1)
    {
        printf("Sorry, that makes no sense.\n");
        return 1;
    }

    // sing the annoying song
    printf("\n");
    while (n > 0)
    {
        printf("%d bottle(s) of beer on the wall,\n", n);
        printf("%d bottle(s) of beer,\n", n);
        printf("Take one down, pass it around,\n");
        printf("%d bottle(s) of beer on the wall.\n\n", n - 1);
        n--;
    }

    // exit when song is over
    printf("Wow, that's annoying.\n");
    return 0;
}
```

Notice that we're *destructively* modifying the user's input. At the end of the program, we have no record of what his original input was. That's fine for our purposes here, but it's something to keep in mind if you wanted your program to thank the user for his specific input at the end of the program.

- A note on style: `n` is fine as a kind of sum variable, `i`, `j`, `k` are generally iterators.
- Let's fix that pesky grammar problem in `beer3.c`:

```
/****************************************************************************
```

```
 * beer3.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Sings "99 Bottles of Beer on the Wall."
 *
 * Demonstrates a condition within a for loop.
 ***************************************************************************/

#include <cs50.h>
#include <stdio.h>


int
main(int argc, char *argv[])
{
    int i, n;
    string s1, s2;

    // ask user for number
    printf("How many bottles will there be? ");
    n = GetInt();

    // exit upon invalid input
    if (n < 1)
    {
        printf("Sorry, that makes no sense.\n");
        return 1;
    }

    // sing the annoying song
    printf("\n");
    for (i = n; i > 0; i--)
    {
        // use proper grammar
        s1 = (i == 1) ? "bottle" : "bottles";
        s2 = (i == 2) ? "bottle" : "bottles";

        // sing verses
        printf("%d %s of beer on the wall,\n", i, s1);
        printf("%d %s of beer,\n", i, s1);
        printf("Take one down, pass it around,\n");
        printf("%d %s of beer on the wall.\n\n", i - 1, s2);
    }
    // exit when song is over
    printf("Wow, that's annoying.\n");
    return 0;
}
```

What's with the `? :` syntax? It's actually a *ternary operator*. The first part `(i == 1)` is a condition which, if true, causes the string after the `?`, "bottle," to be assigned to `s1`. If the condition is false, then "bottles" is assigned instead. Try running `beer3` with 3 as the input and you'll see that we've corrected the grammar glitch. It's called a ternary operator because it takes three operands, as opposed to a binary operator, which takes two, or a unary operator, which takes just one.

- One downside about this approach, however, of using `s1` and `s2` is that we're wasting CPU cycles every time we execute the loop in order to assign values to them. This isn't noticeable for our tiny little program, but it's something to think about when you're writing code for say, a cell phone application where CPU cycles aren't so numerous!

- Okay, finally, let's get around to talking about functions. In `beer4.c`, we'll use a function to implement the chorusline:

```
/****************************************************************************
 * beer4.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Sings "99 Bottles of Beer on the Wall."
 *
 * Demonstrates hierarchical decomposition and parameter passing.
 ***************************************************************************/

#include <cs50.h>
#include <stdio.h>


// function prototype
void chorus(int);


int
main(int argc, char *argv[])
{
    int n;

    // ask user for number
    printf("How many bottles will there be? ");
    n = GetInt();

    // exit upon invalid input
    if (n < 1)
    {
        printf("Sorry, that makes no sense.\n");
        return 1;
    }

    // sing the annoying song
    printf("\n");
    while (n)
        chorus(n--);

    // exit when song is over
    printf("Wow, that's annoying.\n");
    return 0;
}


/*
 * void
 * chorus(int bottles)
 *
 * Sings about specified number of bottles.
 */

void
chorus(int b)
{
    string s1, s2;

    // use proper grammar
    s1 = (b == 1) ? "bottle" : "bottles";
    s2 = (b == 2) ? "bottle" : "bottles";

    // sing verses
    printf("%d %s of beer on the wall,\n", b, s1);
    printf("%d %s of beer,\n", b, s1);
    printf("Take one down, pass it around,\n");
    printf("%d %s of beer on the wall.\n\n", b - 1, s2);
}
```

Notice that `while(n)` is the same as `while(n > 0)` because as soon as `n` reaches 0, it will be false and the loop will terminate.

- Just as someone (if you ever meet him, thank him) wrote `printf` for your benefit, you can write functions like `chorus` that make your job easier and your code more readable. `chorus` is effectively a black box that takes care of the chorus line after being passed the current number of beer bottles. Why are we passing `n--` instead of just `n`? This takes care of passing the argument *and* decrementing the iterator in one line of code. Because of order of precedence, `n` is passed to the `chorus` function and `n` is only decremented after the `chorus` function is finished reading in the value of `n`.

- The syntax for declaring a custom function is the same as for declaring the main method. You need to tell the compiler what variable type the function returns (in this case, `void`, or nothing), the name of the function (`chorus` in this case), and the arguments that it takes (here, `int b`). We need to name the argument so that it can be used within the function. The `n` we pass to `chorus` becomes the variable `b` within the function itself. The lines of code in `chorus`, actually, are nearly identical to how they appeared in `beer3.c`. But, arguably, this is better both design-wise and stylistically. Notice, however, that if we run `beer4`, we get the exact same output as `beer3`.