Computer Science 50 — Week 2 Monday: September 13, 2010
Fall 2010 — Andrew Sellergren
Scribe Notes

Contents

Computer Science 50 | Week 2 Monday: September 13, 2010
Fall 2010 | Andrew Sellergren
Scribe Notes

1 Announcements and Demos (0:00–6:00)

- This is CS50.
- 1 new handout.
- Want to eat lunch with CS50 this Friday? RSVP and we'll head somewhere in the Square or to one of the dining halls. If you can't make it, don't worry, we'll try to do this every other Friday.
- If you think that small bugs don't matter, consider this: Flight 501 of the Ariane 5 expendable launch system culminated with an explosion due to an error in type casting (a 64-bit floating point was converted to a 16-bit signed integer causing arithmetic overflow). Thankfully, the rocket was unmanned and no lives were lost, but the bug caused damages estimated at $370 million. Check out the video and the Wikipedia article. According to one of our teaching fellows, this event was life-changing because it convinced him of the importance and difficulty of software engineering. So too did it convince those in charge of budgeting the European space program as they realized they needed to dedicate more resources toward rigorous testing and validation of software.
- On a lighter note, you might have seen STAR WARS ASCIIMATION as you walked into class today. This is free time taken to the extreme: the author uses ASCII art to recreate the entirety of the original Star Wars movie. For Problem Set 1, you will dabble in ASCII art, as well, albeit on a much smaller scale.
- This is CS50 (literally). If you haven't already, check out the map that depicts where you and all your classmates hail from. We built this using the Google Maps API (application programming interface), which students also used last year for one of the problem sets. Click on the marker clusters to zoom in to individual markers!

2 Problem Set 1 (6:00–14:00)

- If you found yourself overwhelmed last week with the transition from Scratch to C, don't worry, Problem Set 1 will hold your hand as you get familiar with Linux and C. That being said, if you're at all feeling left behind, don't hesitate to reach out with an e-mail, a bulletin board post, or a trip to office hours. We're here to help! And, worse, if you're thinking of dropping the course, come have a conversation with David so he can talk you out of it!
- In the Standard Edition, we challenge you to implement a program that will allow a user to guess the number of Skittles that are in the candy machine in the CS50 Lounge. To simulate this candy machine, we walk you through the use of a pseudorandom number generator. You'll then

have the opportunity to interact with your own program and try to guess the number that has been pseudorandomly chosen.

- The purpose of this first exercise is to get your feet wet with Linux and C. The problem, while simple, is one that you'll nonetheless find challenging to solve in this new programming environment.
- Next, we'll ask you to take on the role of a cashier. Generally speaking, cashiers make change by starting with the largest currency denomination possible and working their way down to the smallest. This is called a greedy algorithm. Interestingly, it will always result in the smallest number of bills and coins being used to make change.
- For the Hacker Edition, we'll throw a wrench into the works: what if you don't have any bills or coins of a particular denomination? How do you optimally make change? We'll leave that to you to figure out.
- Finally, you'll be creating an ASCII animation of a bar chart that illustrates the gender breakdown of student spotting on isawyouharvard. This will involve many of the same constructs you used in Scratch, namely loops, conditions, and a little bit of math.
- Know that you are welcome to try the Hacker Edition of each problem set regardless of your self-declared comfort level and regardless of which version you completed the week before.
- The Hacker Edition of Problem Set 1 begins with the same warm-up candy-counting exercise, but moves on to a problem involving credit card number verification. Finally, you are tasked with creating the same gender breakdown bar chart as in the Standard Edition except you must display it vertically rather than horizontally.
- Avail yourself of help@cs50.net! and the bulletin board. If your question isn't personal and doesn't involve a large portion of your own code, check the bulletin board to see if it has already been posted. If not, go ahead and post it. If your question is person or involves a large portion of your own code, best to e-mail us instead.

3 Academic Honesty (14:00–16:00)

- We take academic honesty very seriously in this course. Over the past few years, we have sent 25 students to the Ad Board. We'd like to never have to send another one.
- For your convenience, our policy is spelled out very clearly on the second page of **every** problem set we release. It describes in detail the line between collaboration and plagiarism, which essentially boils down to one guideline: don't talk in real code. If you want to discuss ideas or specific

problems, feel free to go so far as to write out pseudocode, but don't go any farther. What you write in C, PHP, SQL, JavaScript, etc., should ultimately be your own. If you are ever in doubt, feel free to contact us with your specific situation.

4 `hello, world` (16:00–21:00)

- Just so you realize that the ideas we have discussed thus far in C are perfectly applicable to other programming languages, we offer the following few "Hello World" programs in C++, Lisp, PHP, Perl, and Java.
- As an aside, connecting to the CS50 Cloud is a matter of opening up Terminal on a Mac or PuTTY on a PC. On a Mac, you'll type `ssh malan@cloud.cs50.net`,[1] enter your password, and you'll be connected. On a PC, you'll open a stored session that automatically connects to the CS50 Cloud at which point you'll be prompted for your password. The specification for Problem Set 1 and the CS50 Wiki walk you through this in detail.
- Question: why do you suggest changing the colors of Terminal on a Mac? It will help standardize the appearance of source code and program output across all students. Is it absolutely necessary? No, not at all.

4.1 `Hai.java`

- Take a look at `Hai.java`:

```
class Hai
{
    public static void main(String [] args)
    {
        System.out.println("O hai, world!");
    }
}
```

  You can see that the syntax is similar in spirit to C: there's the keywords `void` and `main` along with open and close parentheses and curly braces. The word `class` is new, but ultimately not much is different.
- To compile and execute this program, we run the commands `javac Hai.java` and `java Hai` from the command line.

4.2 `hai.php`

- The syntax for PHP is perhaps even simpler:

[1] Again, `malan` here should be replaced by your own username.

```
<?
    echo "O hai, world!\n";
?>
```

The `echo` function in PHP is very similar to `printf` in C.

- David learned C, C++, and Lisp in formal courses, but every other language he's learned he has taught himself. Honestly, that's pretty common among programmers because once you know the basic concepts, learning a new programming language is simply a matter of picking up the specific syntax.

5 Bugs (21:00–34:00)

- Let's start from scratch and create a program in C that will demonstrate the imprecision of floating point integers:

```
#include <stdio.h>

int
main (void)
{
    float x = 0.88;
    float y = 0.01;
    float z = x + y;
    printf("%f\n", z);
}
```

When we look at this as humans, we know intuitively that the value of `z` should be 0.89. In the context of Problem Set 1, this is similar to adding 88 cents to 1 cent.

- As an aside, the C Reference is a wonderful resource for looking up nitty gritty details like the formatting characters of `printf`.

- When we compile and run this program, we see the answer 0.890000 is displayed, as we expected. But what if we print out 10 decimal places instead of the default 6?

```
#include <stdio.h>

int
main (void)
{
    float x = 0.88;
    float y = 0.01;
    float z = x + y;
```

```
    printf("%.10f\n", z);
}
```

This time when we run the program (making sure to recompile it first), we get an answer of 0.8899999857. Because we have a finite number of bits being used to represent an infinite number of rational numbers, there is some approximation going on.

- Now let's add a condition to this program just to hammer this point home:

```
#include <stdio.h>

int
main (void)
{
    float x = 0.88;
    float y = 0.01;
    float z = x + y;
    printf("%.10f\n", z);
    if (z == 0.89)
        printf("EQUAL\n");
    else if (z > 0.89)
        printf("GREATER\n");
    else
        printf("LESS\n");
}
```

When we run this version of the program, we see that "LESS" is printed. How do we fix this problem in the context of Problem Set 1? The simple solution is to deal in terms of cents. $1.99 multiplied by 100 becomes 199 cents and removes the need to use floating points at all.

- What was the first bug in history? Legend has it that it was an actual bug (an insect) that found its way into a Mark I, one of which is on display in the Science Center.

5.1 `buggy1.c`

- Can you figure out why the program below doesn't print 10 asterisks as it's supposed to?

```
/*****************************************************************************
 * 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(void)
{
    for (int 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`.

5.2 `buggy2.c`

- How about `buggy2.c`, which is also supposed to print 10 asterisks, one per line, but doesn't?

```
/*****************************************************************************
 * 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(void)
{
    for (int 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.

- If you find that your code won't even compile, come back to these lecture examples and mimic them as far as possible until you figure out where you went wrong.

6 Typecasting (34:00–48:00)

- We've already seen how typecasting can be both useful and dangerous. Casting from integers to characters allows us to create the ASCII alphabet. However, casting from floating points to integers can leave us with an imprecise result.
- Toward the end of the semester, we'll be making use of more sophisticated data types. In fact, we can actually create our own data types, as we did in fact when implementing HarvardCourses. A "course," we reasoned, is an object that has a professor, a schedule, a name, a catalog number, and many other identifying attributes. Of course,[2] you won't be able to cast a "course" object to some other data type and end up with a meaningful result.

6.1 `ascii1.c`

- As `ascii1.c` demonstrates, you can actually explicitly convert an integer to an ASCII character simply by casting it:

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

[2] Pun intended.

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

This program simply prints all the letters in the alphabet, both lowercase and uppercase, along with their ASCII mappings to integers.

6.2 `ascii2.c`

- `ascii2.c` also prints the ASCII mappings, 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(void)
{
    // display mapping for uppercase letters
    for (int 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.

6.3 ascii3.c

- One last variant of this program will demonstrate that we can iterate over characters just like 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(void)
{
    // display mapping for uppercase letters
    for (char 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!

6.4 battleship.c

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

[3] Oh yeah, real clear, huh.

```
   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
```

- 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:

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

#include <stdio.h>


int
main(int argc, char *argv[])
{
    // print top row of numbers
    printf("\n   ");
    for (int i = 1; i <= 10; i++)
        printf("%d  ", i);
    printf("\n");

    // print rows of holes, with letters in leftmost column
    for (int i = 0; i < 10; i++)
    {
```

```
        printf("%c  ", 'A' + i);
        for (int 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!

- You may have noticed that the arguments we pass to `main` in the parentheses got a lot more complicated. This week we'll tease apart what exactly a string is and how you can pass it to a program as an argument. We've already seen this, in fact, when we run `gcc hello.c`. `hello.c` is an argument we pass to the `gcc` program.
- The single quotes around 'A' are intentional because it is a `char`. A `string` is enclosed by double quotes but a `char` is enclosed by single quotes.
- Question: when passing `\n` to `printf` can you use single quotes? No, because `printf` takes a string as argument.

7 Blah (48:00–66:00)

7.1 `beer1.c`

- Let's take a simple problem and try to solve it in an efficient way. We want to print out all the lyrics from "99 Bottles of Beer on the Wall," but we obviously don't want to have to hardcode every line. For starters, we know that the number 99 counts down to 1, which seems pretty easy to handle with a loop. But also, the last line of the song will be "1 bottle of beer on the wall" whereas the last line of all the previous stanzas used the word "bottles" instead. So we need to convert plural to singular when it's appropriate.
- 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(void)
{
    // ask user for number
    printf("How many bottles will there be? ");
    int 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 (int i = n; i > 0; i--)
    {
        printf("%d bottle(s) of beer on the wall,\n", i);
        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;
}
```

In this program, we actually ask the user how many bottles of beer he wants to start with. We capture this value by storing the output of the `GetInt()` function in the variable `n`.

- In the next step, we do some error checking by making sure the user's input is greater than 1. You won't see us spell out in depth the error checking you need to do in every problem set, so you should always be thinking about it yourself. Whenever you take input from the user, you need to

make sure it meets your requirements and won't break your program if it's invalid.

- Notice that in the case the user has given an integer less than 1 as input, we print an error message and then execute the line `return 1`. Since `main` is actually a function itself, it can have return values, specifically of type `int`. Generally speaking, a return value of 0, which is implicitly returned by default, means "everything went okay." Any other return value indicates an error occurred. It's useful to return different values (called sentinel values) for different errors so that when the program breaks, you know exactly where it broke. You may have experienced this on your Mac or PC when something goes wrong and a dialog pops up giving you an error code. This error code might mean nothing to you, but it means something to the programmer who wrote that piece of software.

- Moving on to the loop, we see that it iterates downward instead of upward. This is perfectly fine (and suits our purposes of counting down from 99 here) so long as your terminating condition is eventually reached. `i--` is shorthand for `i = i - 1`.

- We took a shortcut here by writing "bottle(s)" so that the last line of the song is grammatically correct.

- Is there a risk of printing -1 at any point in the song? No. Because the terminating condition is `i > 0`, the very last iteration of the loop will have `i` equal to 1. So in the last line of the song, "0 bottle(s)" will be printed.

- As a matter of good style, we should explicitly return 0 at the end of the program.

- Question: what happens if we change the terminating condition of the loop to `i >= 0`? We actually see -1 printed in the last line, as we guessed earlier.

- Question: where can you view the return values of your programs? Soon we'll be introducing you to GDB, a debugger for C programs. With this tool, you'll be able to step through your program line by line as it executes as well as view the values of variables, including the return value of `main`.

- Let's introduce a bug into this program by removing the error checking on the user's input. Now what happens when we enter -99 as our input? Instead of printing the verses of the song, the program just prints "Wow that's annoying." That's because `n` is set to -99 which is already less than 0, so the `for` loop is skipped entirely.

- What if we were to also mess up the terminating condition of the loop by writing `i < 0`? If we then enter a negative number as input to the program, it will be caught in an infinite loop. What will happen eventually when the `int` reaches negative 2 billion, it's lower limit? It might actually

wrap around to positive 2 billion as it overflows its storage and the critical bit that designates its sign gets flipped. At that point, the program will stop executing because the terminating condition `i < 0` will no longer be true.

- Related to the idea of finite storage is a hack in a Zelda game in which a very long name for your horse Epona causes a buffer overrun. We'll see this next time!