Computer Science 50
Fall 2010
Scribe Notes

Week 3 Monday: September 20, 2010
Andrew Sellergren

Contents

Computer Science 50
Fall 2010
Scribe Notes

Week 3 Monday: September 20, 2010
Andrew Sellergren

1 Announcements and Demos (0:00–5:00)

- This is CS50.
- 1 new handout.
- David recently got an e-mail requesting that he provide his FAS username and password due to a server upgrade. He clicked on the link to do so and, to his surprise, was taken to maxfilmsonline.com. Assuming David actually fell for it and entered his username and password, what can his adversary now do with his password? He can certainly login to David's e-mail account and send e-mails as him, but truth be told, he could have impersonated David via e-mail even without his password (it's surprisingly easy). However, the adversary also know has access to a shell account, as Harvard provides a server cluster much like the CS50 Cloud that you can connect to remotely via SSH. This server cluster is called NICE for New Instructional Computing Environment. That's a little scarier since now he could perhaps start sharing files illegally or launch attacks internally on NICE. What's even scarier is how easy it is to set up a *phishing* attack like this. All he needed was to buy a domain name, copy and paste the HTML that implements Harvard's webmail interface (the HTML source code is readily viewable in any browser), and send out a few e-mails. If you're interested in security issues like this, take Computer Science 105!

2 Cryptography (5:00–21:00)

2.1 Secret-key Cryptography

- As it's name implies, secret-key cryptography relies on a key which is known only to you and the recipient of the encrypted message and is used both to encrypt and to decrypt the message.
- One problem with secret-key cryptography is that if the key is compromised, all messages, both past and future, are vulnerable to being intercepted.
- Another problem with secret-key cryptography is the challenge of sharing a key between two parties. How could you, for example, agree on a secret key with Amazon when you have no real contact with anyone in the company? Thankfully, public-key cryptography provides a solution to this quandary.
- For Problem Set 2, you'll be implementing secret-key cryptography. We exposed you to this last time with a reference to A Christmas Story in which Ralphie's secret decoder ring reveals the message "Be sure to drink your Ovaltine." Incidentally, the secret key was 13 which means that all the letters in the message were rotated around the alphabet by 13 letters, wrapping around when the end is reached. This cipher is a specific instance (called ROT-13) of Caesar's cipher.

2.2 Problem Set 2

- In Problem Set 2, you'll be writing a program that takes the key as a command-line argument, prompts the user for plaintext to encrypt, and then prints the ciphertext. As you iterate from left to right through the characters of the user-provided plaintext, you'll need to have some conditions in place that prevent punctuation from being rotated.

- As we've discussed, Caesar's cipher isn't the most secure. Because there are only 26 possible keys, a brute-force attack which attempts to decrypt a message using all possible keys won't take very long to succeed. In the remainder of Problem Set 2, you'll be implementing Vigenère's cipher, which is slightly more secure. In Vigenère's cipher, keys are actual words. As you iterate from left to right through the characters of the plaintext, you also iterate through the characters of the key, using a different one to rotate each character of the plaintext. In this way, there are 26^n possible keys, where n is the length of the key.

- In the Hacker Edition of Problem Set 2, you are tasked with decrypting a few ciphertext strings. On many systems, including Mac OS and Linux, passwords are stored in files as encrypted text. The hint we give you is that these passwords you're trying to crack aren't very secure, so making certain assumptions about them will reduce the runtime of your cracking algorithm. For example, instead of "hello," perhaps the user chose "hell0" or "he110" as his password. These are slightly more secure in that a dictionary attack in which an adversary tries all English words in a dictionary won't succeed outright.

- As inspiration for Problem Set 2, check out the scene from Spaceballs in which King Roland reveals his password to be, umm, not so secure.

- If you couldn't attend the walkthrough in person, the video is available on the Problem Sets page. Sections, as well, are a valuable resource to you.

- Per the syllabus, we use three axes to grade your problem sets. Your work will be rated poor, fair, good, better, or best for Correctness, Design, and Style. Generally speaking, correctness refers to whether or not your program actually works and adheres to the specification. Design refers to the choices you made while implementing your program: did you solve the problem in one of several intelligent, efficient possible ways? Finally, style refers mostly to the readability of your code: is it pretty-printed with meaningful variable names and plenty of comments? All in all, we aim to give you much more qualitative feedback than quantitative since our hope is that you will take our advice into consideration as you tackle the next problem set. Know also that correctness is weighted more than design which is weighted more than style. And by all means, don't think that a "good" is equivalent to a 3 out of 5, or 60%. We ultimately determine your final grade based on how you have progressed over the semester.

3 Demographics (21:00–26:00)

- As we mentioned a few lectures ago, most of you are sophomores, but the rest of you are evenly distributed among the freshman, junior, and senior classes.
- For the first time, the "less comfortable" students make up the majority of the class (46%) while the remainder is split amongst "somewhere in between" (42%) and "more comfortable" (12%).
- 77% of you have never taken a course in computer science before this class.
- Almost 200 of you have so-called normal phones (i.e. non-smartphones). By the way, we ask you because it's useful to know what the user base is for Android, BlackBerry, and iPhone apps come semester's end when some of you decide to develop them for your final projects.
- You're split about 50-50 between Mac and Windows users. Check out our Software page to take advantage of our MSDN Academic Alliance license and download the latest version of Windows.
- The most common reason for taking CS50 is as an elective!

4 More Beer (25:00–46:00)

4.1 `beer1.c`

- Let's reimplement `beer1.c` starting from scratch. To begin we need to include the pre-processor directive and the skeleton definition of `main`:

```
#include <stdio.h>

int
main (void)
{

}
```

Next, we need to print a message to the user, get his input, and store it as the number of bottles of beer to count down from:

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

int
main (void)
{
    printf("How many bottles of beer? ");
```

```
    int n = GetInt();
}
```

We need to include the CS50 library's header file because we're using `GetInt()`.

- Even if we don't explicitly instruct you to in, say, a problem set specification, checking user input is always a priority:

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

int
main (void)
{
    printf("How many bottles of beer? ");
    int n = GetInt();

    if (n < 1)
    {
        printf("Sorry, that makes no sense.\n");
        return 1;
    }
}
```

We return 1 if the user gives us bad input because anything non-zero represents an error code.

- Now we move on to the business of actually singing the song with a `for` loop:

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

int
main (void)
{
    printf("How many bottles of beer? ");
    int n = GetInt();

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

    for (int i = 99; i >= 1; i--)
```

```
    {

    }
}
```

Our `for` loop terminates at `i >= 1` because if we go all the way to 0, then -1 will be printed in the last line of the song. Finally, let's add the actual lines of the song:

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

int
main (void)
{
    printf("How many bottles of beer? ");
    int n = GetInt();

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

    for (int i = 99; i >= 1; 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);
    }
}
```

The only place where we really have to stop and think is in the last line of each verse in which we want to print `i - 1` rather than `i`.

- One major bug with our program as written above. No matter what input the user provides, we're iterating from 99 down to 1. We can easily fix this by initializing `i` to the value of `n` rather than hardcoding 99.

4.2 `beer2.c`

- We can very easily convert our `for` loop to a `while` loop like so:

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

```
int
main (void)
{
    printf("How many bottles of beer? ");
    int n = GetInt();

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

    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--;
    }
}
```

Notice we need to explicitly decrement `n` within the loop.

- What is one upsides to the this approach? With a `while` loop, we have no need for the variable `i`, so we use less memory.
- However, one downside of this approach is that our use of `n` is now destructive. That is, at the end of the loop, `n` doesn't have the same value as at the beginning. So we've lost the ability to do anything else with that value.
- In the end, which is the better approach? To be honest, they're both equally good choices. Underneath the hood, they're probably implemented exactly the same. The `for` loop affords you more fine-tuned control although it's syntax is a little uglier.

4.3 `beer3.c`

- If we want to get a little fancier, we'll actually properly handle (in a grammatical sense) the case in which "beers" becomes "beer." We do so 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(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--)
    {
        // use proper grammar
        string s1 = (i == 1) ? "bottle" : "bottles";
        string 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.

4.4 `beer4.c`

- One last optimization we can make is to abstract away the logic for printing verses into a separate function:

```
/**************************************************************************
 * 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 b);


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");
    while (n)
        chorus(n--);

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


/*
 * Sings about specified number of bottles.
 */

void
chorus(int b)
{
    // use proper grammar
    string s1 = (b == 1) ? "bottle" : "bottles";
    string 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);
}
```

`chorus` is a function that takes a single argument. We're being a little fancy here by passing it `n--` instead of `n`. All this does is accomplish the function call and the decrementation of `n` in a single line of code as opposed to two.

- We can write `while(n)` because it's equivalent to `while(n > 0)`. As long as `n` is non-zero, it evaluates to "true," so the loop continues executing.
- `chorus` has a return type of `void` since all it does it print lines to the screen. Otherwise, it's just an exact copy of the lines of code within the loop in `beer3.c`. Within `chorus`, `b` never changes value. `b` takes on different values within `chorus` because we pass different values of `n`.
- The point of this code is not to be concise at the cost of clarity, but merely to introduce you to some of the syntax you might see in textbooks and the like.

5 Arrays (46:00–72:00)

5.1 array1.c

- Recall that arrays are chunks of contiguous memory that allow you to store multiple variables of the same type in a single container. Take a look at `array1.c` that introduces us to arrays in C:

```
/**************************************************************************
 * array1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Computes a student's average across 2 quizzes.
 *
 * Demonstrates use of an array, a constant, and rounding.
 **************************************************************************/

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


// number of quizzes per term
#define QUIZZES 2


int
main(void)
{
    float grades[QUIZZES], sum;
    int average, i;

    // ask user for grades
    printf("\nWhat were your quiz scores?\n\n");
    for (i = 0; i < QUIZZES; i++)
    {
        printf("Quiz #%d of %d: ", i+1, QUIZZES);
        grades[i] = GetFloat();
    }

    // compute average
    sum = 0;
    for (i = 0; i < QUIZZES; i++)
        sum += grades[i];
    average = (int) (sum / QUIZZES + 0.5);
```

```
    // report average
    printf("\nYour average is: %d\n\n", average);
}
```

The purpose of this program is to prompt a user for his quiz grades in a particular class and then compute the average. Of course, we'd like to store these quiz grades in the same variable since if there are 10 of them, we don't want to have to declare 10 separate variables.

- The syntax for declaring an array looks like this:

  ```
  float grades[QUIZZES];
  ```

  Here, we're asking for an array that can store 2 variables of type `float`. At the top of the program, we assign `QUIZZES` as a constant that holds the number 2 using the `#define` directive. By convention, constants are written in all-capital letters in C. Defining a constant at the top of our program allows us to easily change the value it stands for throughout the program. Also, it allows the compiler to make certain optimizations.

- Notice that we can consecutively declare multiple variables of the same type by separating them with commas. In addition to an array of `float`'s, we declare a single `float` named `sum`.

- Within the loop, we assign the quiz grades to the elements of the `grades` array. Arrays are 0-indexed, so the first grade goes into `grades[0]` and the second grade goes into `grades[1]`. In each case, it is the return value of `GetFloat()`, the user's input, that is stored.

- To calculate the sum of the quiz grades, we again loop through the `grades` array and add the values together. Once we have the sum, we divide it by the number of quizzes and then cast it to an `int`. Interestingly, instead of using the `round` function, we can accomplish the same by adding 0.5 before we cast to an `int`. If the quiz total is, say, 99.6, then adding 0.5 will produce 100.1 and casting to an `int` will result in 100 (since it gets truncated). 100 is the same result we would get from `round(99.6)`. Likewise if the quiz total is 99.4, adding 0.5 and casting results in 99, just as rounding would. Try it with other examples if you're still a non-believer!

5.2 `string1.c`

- When we began working with `argv`, we mentioned that it was an array of strings. Well, since strings are really just arrays of characters, `argv` was actually an array of arrays, or a two-dimensional array.

- Since strings are just arrays of characters, we can access each character in them just as we would the elements in an array:

```
/**************************************************************************
 * string1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints a given string one character per line.
 *
 * Demonstrates strings as arrays of chars and use of strlen.
 **************************************************************************/

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


int
main(void)
{
    // get line of text
    string s = GetString();

    // print string, one character per line
    if (s != NULL)
    {
        for (int i = 0; i < strlen(s); i++)
        {
            char c = s[i];
            printf("%c\n", c);
        }
    }
}
```

Here, we iterate over all of the characters in the user-provided string and print them out one at a time, one per line, simply to demonstrate that a string is essentially an array.

- The `string` data type is one that we made up to reduce confusion in these first few weeks. `s` is actually an array of `char`'s. But even that's a bit of a white lie. Functions in C can't return entire arrays. Rather, `GetString()` returns a pointer to an array, i.e. the memory address in RAM of that first chunk of the array. Basically, it tells us where we can find the string.
- If the user doesn't actually give us a string when we prompt him for one using `GetString()`, he can wreak havoc in our program if we aren't careful. Whenever we're dealing with strings, we need to check that they don't have the special value of `NULL` before we proceed to manipulate them.

- When we talk about the stack in computer science, we're referring to the map of memory addresses for all of our computer's RAM. The very lowest of those memory addresses is reserved for the operating system, so if you are ever handed the memory address 0, something is wrong. Trying to manipulate a string that has value `NULL` is equivalent to trying to access memory address 0, which is never good.
- If we instead use `i <= strlen(s)` as the terminating condition for our loop, we'll end up iterating off the end of our array and touching memory that doesn't belong to us. Still, the program compiles and seems to work okay, printing a blank after "o." What if we iterate up to 100 or 1000000? At some point, we run out of memory and cause a segmentation fault.
- Failing to check the bounds of an array is a serious security vulnerability. A knowing adversary can use this to hijack a program and execute his own malicious code.

5.3 Search and Sort

- On the board are two arrays of integers covered by pieces of paper.
- Let's bring down a volunteer and ask him to find the value 50 in the top array. Having no foreknowledge of the array, he looks behind pieces of paper "randomly" and finds 50 on the seventh try. His self-described process was to look under each piece of paper one at a time, moving left to right across the board.
- Of course, this process isn't very efficient. In the worst case, for an array of length n, the very last number will be the one we are searching for, so we'll have to walk through n steps to find it. With no foreknowledge of the array, this is the best we can do: to brute force examine every single element of the array.
- Now let's search the second array for the number 50, this time knowing that the array is sorted but still not knowing what numbers it contains. Recall our efforts to search the phonebook in the first lecture. If we jump to the middle of the array, we find the number 124. Now we can disregard the right half of the array since our number is less than 124 and the array is sorted. With the remaining left half of the array, we again choose a number in the middle and find 51. Only one number remains to the left of 51 and since our number is less than 51, it must be our number.
- Dealing with a sorted array and using binary search, we were able to greatly reduce the number of steps it took to find the number 50. But how do we sort an array? More on that next time.