Computer Science 50
Fall 2011
Scribe Notes

Week 2 Wednesday: September 14, 2011
Andrew Sellergren

Contents

Computer Science 50
Fall 2011
Scribe Notes

Week 2 Wednesday: September 14, 2011
Andrew Sellergren

1 Announcements and Demos (0:00–4:00, 8:00–10:00)

- Ever wanted to see Star Wars drawn completely in ASCII art? Here is your chance. ASCII art is simply a form of drawing that uses only the characters from ASCII.
- To set the stage for our discussion of cryptography, we'll watch a scene from Spaceballs in which King Roland reveals that the password to the air shield is 12345.
- Office Hours will be held tonight and tomorrow night in Lowell from 9 p.m. to midnight. The houses are working on installing additional access points so that wireless speeds will be improved.
- If you're having issues with the Appliance, please reach out to us via help@cs50.net or otherwise. We also have a new solution for those of you with older or otherwise resource-limited computers like netbooks that will allow you to connect to our servers.

2 Cryptography (4:00–8:00)

- Odds are that some of you out there have a password that's not too much stronger than King Roland's. Even so, your password would not be stored as 12345, but more likely as something like this:

  ```
  {crypt}$1$LlBcWwQn$pxTB3yAjbVS/HTD2xuXFI0
  ```

  For security purposes, passwords are not generally stored in plaintext, but in ciphertext. Thus, when you are told by some IT person that you can't recover your password, but you can change it, that's because he does not have access to the plaintext, but only the ciphertext, which he is unable to decipher.
- Passwords are stored in ciphertext on servers to minimize the impact of a malicious user gaining access. If this user steals the list of user's passwords, he still has to decrypt them, which is mathematically very difficult. Even the server itself is unable to decrypt passwords in a timely fashion. To do this, it would need to store the secret key that was used to encrypt the passwords, which would defeat the purpose since a malicious user who gained access to the server would then be able to steal the list of user's passwords as well as the secret key used to encrypt them.
- So if the server can't decrypt the ciphertext version of your password, how does it know that you typed in the right password when you login? The ciphertext version of your password, or more properly the *hash* of your password, is the result of a mathematical operation performed on the plaintext. This mathematical operation is deterministic, so if it is applied to what you typed in when logging in, the result should match the hash stored on the server.

- If you read in the news that a website was hacked and its users' passwords were stolen, it might be the case that the website wasn't even encrypting passwords before storing them. In any case, if you were one of the users of this website, your best bet is to change your password immediately.

3 From Last Time (10:00–34:00)

- Recall our attempt to programmatically output the lyrics of the annoying song "99 Bottles of Beer on the Wall":

```
/**************************************************************************
 * 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;
    }
```

First, we did some error checking, deciding not to reprompt the user if he provided bad input, but instead to exit with an return code of 1. Second, we print out the four-line stanza for each of `n` bottles of beer. We dynamically insert the number of bottles of beer using `printf` and formatting strings, which even allow us to print out the value of `i - 1` in the last line of each stanza.

- Throughout the semester, we'll encourage you to bang on your code before submitting it, mostly because we'll be doing the exact same thing when we grade it. Your code may work perfectly for 99 normal cases, but if it doesn't work for 1 or 2 so-called *corner cases*, then you're on the hook. Think about the real-world implications of improperly handling corner cases. This is what results in your frustration with a certain program when it crashes if you type some unusual key combination, for example. In this program, corner cases would be unusual numbers that the user provides, perhaps 0 or 1. We want to make sure that our program never prints out "-1 bottle(s) of beer on the wall." In fact, it won't because of our loop condition `i > 0`. But, to be sure of that, we should try providing it unusual inputs anyway. This would help us identify a mistake such as using `i >= 0` instead.

- Question: how do you use `return 1` to get another integer from the user? You don't. This was a design decision on our part that if the user provides bad input, the program will exit rather than reprompt him.

- Question: how do you use the return code of 1? Soon, we'll introduce debuggers, which will allow you to examine the return codes of your programs as well as step through them line by line.

- How can we improve on the current implementation? First, we can handle the corner case of there being only 1 bottle of beer left and thus a need for singular words in the stanza. Second, we can make our `main` function a little cleaner by using *hierarchical decomposition*. If you find yourself copying and pasting snippets of code in your programs, it's a good sign that you should be using hierarchical decomposition instead, which is to say that you should be breaking your program down into smaller functions. Thus far, we've only written a single function, `main`, for each of our programs, but here we see it can be useful to write more than one:

```
/**************************************************************************
 * 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);
}
```

As before, we quit if the user provides input. This time, however, the actual printing of the song is accomplished in two lines. We've effectively outsourced the singing of the song to a function named `chorus`. Within `chorus`, we now have `printf` statements with more than one formatting string. This, in addition to the two lines of code at the beginning of `chorus`, addresses the grammar problem we ran into before whereby the singular case wasn't handled very elegantly.

- When defining the function `chorus`, we specified its return value and the type of input it takes. In this case, `chorus` returns nothing so its return type is `void`. The input it takes is a single argument of type `int`. Because this integer represents a number of bottles, we chose the name `b` to represent it. Thereafter in the function, whatever we passed into `chorus` will be referenced by the name `b`. So in `main`, we pass a variable named `n` to `chorus`, but the `chorus` function will actually make a copy of this variable and name it `b`.
- What about those first two lines of `chorus`? Well, they might make more sense if we rewrote them as follows:

```
string s1;
if (b == 1)
    s1 = "bottle";
else
    s1 = "bottles";

string s2;
if (b == 2)
    s2 = "bottle";
else
    s2 = "bottles";
```

First, we declare a string named `s1`. Second, we initialize `s1` to the value "bottle" if the number of bottles is 1, else we initialize it to the value "bottles." Similarly, we declare and initialize `s2` afterward. After we've declared and initialized `s1` and `s2` like so, we substitute them into the lyrics where previously we had the word "bottle(s)" for every case. Note that we need two different strings because when there are 2 bottles, the last line of the stanza must read "1 bottle of beer on the wall" whereas when there is 1 bottle, the last line must read "0 bottles of beer on the wall." The first two lines of code in `chorus` are actually just a more succinct way of writing these if-else conditions. Also note that in the expanded if-else conditions above, we had to declare the variables `s1` and `s2` outside of the if and else blocks in order that they would exist outside of those blocks. We called this a problem of *scope* last time.

- Question: if you declare a variable in `main`, will it be accessible to other functions? No. Again, this is a problem of scope. A good rule of thumb is that variables only exist between the curly braces within which they are declared.

- One last thing to take away from this program: above our definition of `main`, we have the following lines of code:

```
// function prototype
void chorus(int b);
```

  Because C is an older language, we have to tell the compiler that the `chorus` function exists before we try to use it. If we remove this *function prototype*, we'll get a compiler error stating "implicit declaration of function." We could actually fix this error by moving the entire implementation of `chorus` above `main`, but as your programs get more and more complex, this solution will be less and less ideal. Imagine you are reading through someone else's code to figure out what it does. You don't want to read through a number of helper functions before you get to `main`, which is the heart of the program. So, to compromise, we write function prototypes at the tops of our programs[1] and the function implementations at the bottoms of our programs.[2]

- Question: why do you have to tell the compiler what the type of a function's output is; couldn't it infer this from your program? Yes and no. In short, this is a way of improving the probability of correctness in your program by forcing this type to match the type of the values you return at different places in your program, which might be nested deep within several conditions.

[1] Or in separate files.
[2] Or in separate files.

4 Scope and the Stack (34:00–59:00)

4.1 buggy3.c

- As we mentioned earlier, variables belong to the functions that manipulate them. This is the concept of *scope*. Variable scope can introduce some subtle bugs, as we'll soon see in `buggy3.c`:

```
/**************************************************************************
 * buggy3.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Should swap two variables' values, but doesn't!
 * Can you find the bug?
 **************************************************************************/

#include <stdio.h>


// function prototype
void swap(int a, int b);


int
main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %d\n", x);
    printf("y is %d\n", y);
    printf("Swapping...\n");
    swap(x, y);
    printf("Swapped!\n");
    printf("x is %d\n", x);
    printf("y is %d\n", y);
}


/*
 * Swap arguments' values.
 */

void
swap(int a, int b)
```

```
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Everything looks fine, but when we run the program, we get the following output:

```
x is 1
y is 2
Swapping...
Swapped!
x is 1
y is 2
```

- Could it be that we left out the function prototype for `swap`? No, because the compiler would've yelled at us.
- Looking at the implementation of `swap`, it appears correct. We store the value of one variable in a temporary variable, then overwrite its value, then overwrite the other variable's value with the value stored in the temporary variable. We couldn't simply write `a = b` and `b = a` because we would lose the value of `a` and both would assume the value of `b`.[3]
- It appears that `a` and `b` are being properly swapped, but the values aren't being stored in `x` and `y`. What's really happening when we call `swap` is that we're passing in copies of the variables `x` and `y`, not the variables themselves. So while `swap` successfully swaps the values of `a` and `b`, it doesn't actually have any effect on the values of `x` and `y`. At the end of `swap`, before it returns, `a` has the value 2, `b` has the value 1, `x` has the value 1, and `y` has the value 2. As soon as `swap` returns, however, `a`, and `b` are forgotten and `x` and `y` retain their original values. In a few weeks, we'll be able to fix this problem by using *pointers*, but for now just assume that functions cannot modify variables that are defined outside their scope.

4.2 The Stack

- What's actually going on in memory when we deal with these variables? Variables are stored in RAM, a temporary kind of memory, as opposed to on the hard disk, a more permanent kind of memory. We can visualize a computer's RAM like so:

[3] As a sidenote, it is actually possible to swap two variables' values without using a temporary variable, but more on that later.

According to this visualization, we can number all of the bytes in a computer's RAM from 0 to 2 billion or so (assuming your computer has 2 GB of RAM). When we execute a program, then, the variables that belong to `main` (first its arguments or parameters, then its locals) will be assigned to the lowest numbered bytes in the program's memory, which appear here at the bottom of this rectangle. When `main` calls another function, say `foo`, that function's variables get stacked on top of `main`'s in memory.

- Interestingly, often when a website is hacked it is by a buffer overrun attack, which is just a fancy way of saying that a malicious user provided some input that was too large to be stored properly in one of these inner rectangles and consequently overwrote some of the data in another inner rectangle. As this visualization shows, `foo` lives separately from `main`, so it doesn't (or shouldn't) have access to `main`'s variables.

4.3 Global Variables

- One way to fix this problem of scope is to use *global variables*. These variables are unique in that they can be referenced by any function in a program. Take a look at `global.c` to see how they are used:

```
/**************************************************************************
 * global.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Increments variables.
 *
 * Demonstrates use of global variable and issue of scope.
```

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

#include <stdio.h>


// global variable
int x;

// function prototype
void increment(void);


int
main(void)
{
    printf("x is now %d\n", x);
    printf("Initializing...\n");
    x = 1;
    printf("Initialized!\n");
    printf("x is now %d\n", x);
    printf("Incrementing...\n");
    increment();
    printf("Incremented!\n");
    printf("x is now %d\n", x);
}


/*
 * Increments x.
 */

void
increment(void)
{
    x++;
}
```

We declare the variable `x` outside the scope of both `increment` and `main`, right above the function protoype. This solves our problem in that `x` can be referenced in any of the functions in this file, but it's not the cure-all that it might seem. Global variables are generally considered bad style because they're hard to keep track of. If you're using code from other libraries, for example, the names of your global variables might inadvertently conflict with the names of library variables and introduce subtle bugs into your program.

- Note that the `increment` function returns `void` and takes `void` as arguments because it can reference `x` without it having been passed as an argument.

4.4 Segmentation Faults

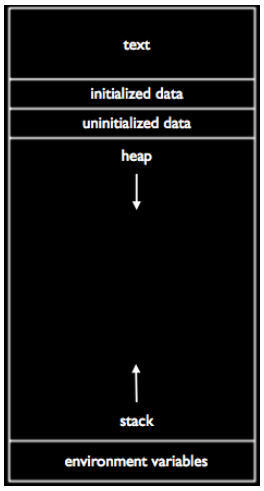
- Let's mess with your minds a little bit by writing a program called `bad.c` that looks as follows:

```
#include <stdio.h>

void bad(int n);

int
main(void)
{
    int n = 0;
    bad(n);
}

void
bad(int i)
{
    printf("%d\n",i);
    i++;
    bad(i);
}
```

  We could actually call `bad`'s argument `n` instead of `i`, since it's in a different scope than `main`.

- Why is this program bad?[4] Because `bad` always calls itself, it will repeat infinitely. Not only that, each time it calls itself, another chunk of memory will be reserved for its variables and parameters, as our visualization of the stack showed. Since our outer rectangle is finitely big, we'll eventually run out of space for adding inner rectangles. That is, we'll run out of memory, or worse, we'll overwrite memory that is being used for other purposes by the operating system. Let's look at a more complete visualization of a program's memory:

[4] Because that's it's name, duh.



Notice that the stack grows upward as we call more and more functions, but something called the heap grows downward. As this visualization suggests, the two can potentially collide if one or the other takes up too much space. When we compile and run `bad.c`, this happens very quickly and we get a message saying that a *segmentation fault* has occurred. This is the operating system's safety mechanism which kills a program when it tries to touch memory that doesn't belong to it. You may have heard of the concept of cracking a program, meaning that the need for an authorization code or a serial number is bypassed so that you can install Microsoft Word or Photoshop without purchasing it. In effect, these cracks are simply touching memory that they don't own and causing the program to skip past the simple if condition that asks the user to provide an authorization code or serial number.

5 Arrays and Command-line Arguments (59:00–73:00)

- Conceptually, strings are simply collections of `char`'s, stored one after another in memory. If we were to store "hello" as a string, each character of it would be assigned one byte of memory and an additional byte of memory would be stored at the end. This additional byte is `\0`, which signifies the end of a string and actually represents a byte whose bits are all 0. Note that there isn't a `\0` after every word you enter, but only after every string you enter, since a string can comprise multiple words and a space can be represented as a `char` just like any other letter.
- Recall from FruitcraftRPG that Scratch allowed you to store a collection of related variables in what was called a list. This is equivalent to an array in C.

5.1 argv1.c

- Thus far, we have be defining `main` as a function that takes no arguments, represented as `void`. However, we can actually define `main` as taking arguments which we pass into the program at the command line. We've already made use of command-line arguments every time we ran the program `make`. Running `make beer4`, for example, passes "beer4" as a command-line argument to `make`. Let's see how we might handle these command-line arguments in `argv1.c`:

```
/**************************************************************************
 * argv1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints command-line arguments, one per line.
 *
 * Demonstrates use of argv.
 **************************************************************************/

#include <stdio.h>


int
main(int argc, char *argv[])
{
    // print arguments
    printf("\n");
    for (int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    printf("\n");
}
```

`argc` holds the number of command-line arguments that you passed to the program. By default, the 0^{th} argument is always the name of your program, so `argc` will always be at least 1.

- Recall that we've been using `string` as a shorthand for `char *`, so we could rewrite the second argument to `main` as `string argv[]` if we include the CS50 Library.

- `argv` is a variable that holds the command-line arguments that were passed into the program. The square brackets after `argv` indicate that it's an array. In fact, it is an array of strings. In order to access the strings it holds, you can index into it by specifying a number between the square brackets. `argv[0]`, for example, gives the 0^{th} argument which is always the name of the program.

- When we compile and run this program, we see that it prints out the command-line arguments we provided it.

5.2 argv2.c

- `argv2.c` demonstrates what a string actually is:

```
/**************************************************************************
 * argv2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints command-line arguments, one character per line.
 *
 * Demonstrates argv as a two-dimensional array.
 **************************************************************************/

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


int
main(int argc, char *argv[])
{
    // print arguments
    printf("\n");
    for (int i = 0; i < argc; i++)
    {
        for (int j = 0, n = strlen(argv[i]); j < n; j++)
            printf("%c\n", argv[i][j]);
        printf("\n");
    }
}
```

When we compile and run `argv2.c`, we see that it prints out the command-line arguments, one character per line. Programmatically, then, we are iterating not only over the arguments, but over each character in each argument. We can do this because `argv` is an array of strings and a string is actually an array of characters.

- Notice that in the first part of parentheses after the second `for`, we initialize not just `j`, but also `n`. We do this simply by separating them with a comma. `strlen` returns the length of a string. Because `argv` is actually an array of strings, we know that `argv[i]` is a string. Going one level deeper[5], we can access the j^{th} character in `argv[i]` like so: `argv[i][j]`.

[5] Deeper and deeper.

6 Teaser (73:00–75:00)

- As a teaser for next time, check out how using strings as arrays and characters as integers can allow us to implement Caesar's cipher, just as little Ralphie discovered in this clip from A Christmas Story.