Computer Science 50
Fall 2011
Scribe Notes

Week 3 Monday: September 19, 2011
Andrew Sellergren

Contents

Computer Science 50
Fall 2011
Scribe Notes

Week 3 Monday: September 19, 2011
Andrew Sellergren

1 Announcements and Demos (0:00–4:00, 10:00–12:00)

- A joke from an undergrad not even taking CS50:

  > Q: Why did the computer programmer get upset when he forgot how to program?
  > A: Because he had to start again from Scratch!

- If you still have a Scratch Board, please drop it off with one of the TFs or CAs this week!
- You can resection at cs50.net/resection. Feel free to e-mail section@cs50.net if we seem to have forgotten about your request.
- Join us for CS50 Lunch this Friday at 1:15 p.m.! RSVP here. We'll vary the times and locations to allow everyone the possible of joining at least once.
- Wednesday Office Hours have proven quite popular. They will now be held at Quincy House. Other sessions will be held tonight in Pfoho, tomorrow in Leverett, and Thursday in Lowell.
- For those of you more comfortable, we're looking to fork off special Hacker hours to give you help on the Hacker Edition of problem sets.
- Problem Set 0 and Problem Set 1 will be returned to you with comments in the next few days. For the remainder of the semester, we'll aim to give feedback on your problem sets within one week of their deadlines.
- Please leave more detailed notes than simply "it doesn't work" when posting to help.cs50.net! It's just not enough for us to go on. Try stating the exact problem you're seeing, the symptoms you're experiencing, the specific error message, the steps to reproduce, etc. What have you tried that's not worked? What operating system are you running? The idea is that if you provide enough detail, we can reply to you with an answer rather than another question. Our goal (which we've met so far with one exception) is to respond to your questions within 24 hours!
- It's that time of the semester where you might be asking yourself "Should I stay or should I go now?"[1] Realize that if you're feeling like posting to HarvardFML about CS50, you're not alone. We know that this course is difficult and frustrating at times! If you keep at it, you'll eventually leave behind the mistakes you're making now and even come to look at them as silly. Keep in mind that you didn't see anyone crying at the CS50 Fair. They were all extremely proud at what they had accomplished! If you're already feeling stressed out as a result of CS50, consider altering your work habits. For example, you might be able to start problem sets

[1] If you go there *will* be trouble. Well, actually, there won't be any trouble, just a lot of fun you'll be missing out on!

earlier so that you can go to Office Hours for help. The frustration you feel is a part of every programmer's life, irrespective of his experience level. **Please reach out to David, Rob, Matt, or any of the other teaching fellows with any and all of your concerns!**

2 Security (4:00–10:00, 12:00–22:00)

- One of the files we included in the source code for last week was `iUnlock.c`, which was used to jailbreak one of the original iPhone operating systems. Although this code is slightly beyond our knowledge at this point in the course, we can scroll through it and see some familiar syntax. For example, the `#include` lines at the top and function definitions with return types and arguments. Hopefully this gives you a sense of real-world applications of what we're learning.

- Today is David Shares His E-mail Day, so we'll look at one from his spam folder now. Addressed to "Harvard College Webmail Account Subscriber," this e-mail is an example of what's known as a *phishing attack* in which a malicious user attempts to trick you into giving him your credentials by posing as a legitimate service that you use. As we'll see later in the semester, it's incredibly easy to write a program that sends out e-mails to thousands of addresses all at once and even spoofs a legitimate-sounding e-mail address. In fact, during his freshman year at Harvard, David once sent an e-mail posing as his roommate as a gag,[2] but unfortunately forgot to disable his e-mail signature, making it very obvious who had pulled the prank.

- Interestingly, phishing attacks like this one that appears in David's spam folder are a pretty effective numbers game. If a malicious user sends e-mails like this to 1 million people and only 1% of them end up falling for it, the malicious user has still managed to hijack 10,000 accounts.

- Another common attack vector for e-mail spam is attached images with somewhat wavy text. What's going on here? Images make it somewhat more difficult for spam filters to do text analysis. In addition, by making the text wavy, spammers hope to confuse the optical character recognition (OCR) technology which enables text analysis in images.

- Secret-key cryptography is a type of cryptography that relies on a secret shared only between you (let's call you Alice) and authorized parties (we'll call them Bob). This secret is used to transform plaintext into ciphertext. One of the simplest implementations of secret-key cryptography is a rotation cipher (formally called Caesar's cipher) whereby instead of writing the letter "A," we write the letter "B," and instead of the letter "B," we write the letter "C," and so on. In this scenario, the secret key is just 1,

[2] Please do not follow his example!

the number of places by which letters are shifted. ROT-13 is a variant of this rotation cipher in which the secret key is 13.

- How secure is Caesar's cipher? Not very. If an adversary doesn't know the secret key, he need only try a maximum of 25 possibilities before he'll discover it. Of course, he can do this quite quickly with the help of a computer. Incidentally, we exposed you to Caesar's cipher 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."

- Slightly more secure than Caesar's cipher is Vigenère's cipher, which you'll be implementing in the second part of Problem Set 2. 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. If the letter "A" appears in your key, then you rotate the plaintext character by 0. If the letter "B" appears in your key, then you rotate the plaintext character by 1. Let's say our secret key is ABC and the plaintext we want to encrypt is "hello." Then we'll rotate the letters as follows:

```
plaintext    hello
key        + ABCAB (01201)
ciphertext   hfnlp
```

  In this way, there are 26^n possible keys, where n is the length of the word you use as a key. If our keyword is only 5 letters long, there are already some 11 million possible keys that an adversary might have to try to discover our secret. Of course, this is a lot of possible keys to try by hand, but not all that many to try using a computer. Thankfully, there are more secure methods of cryptography which we'll discuss later.

- 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.

3 From Last Time (22:00–42:00)

- Last week, we implemented a program that prints out the lyrics to "99 Bottles of Beer on the Wall" and made several optimizations to our code to clean it up a bit. We started off with something like this:

```
#include <stdio.h>

int
main(void)
{
    printf("How many bottles of beer will there be?\n");
    n = GetInt();
}
```

Oops, already we've made some mistakes. We can't use `GetInt` without including the CS50 Library. Also, we need to declare the type of `n` before we can initialize it. Let's fix these mistakes and press on:

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

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

    for (int i = n; i >= 0; i--)
    {
        printf("%d bottles of beer on the wall.\n", i);
        printf("%d bottles of beer.\n", i);
        printf("Take one down, pass it around.\n");
        printf("%d bottles of beer on the wall.\n", i - 1);
    }
}
```

We begin at `n`, the number provided by the user, and iterate down to 0. On each iteration, we substitute in the current number of bottles, stored in `i`, into our lyrics.

- Now we've got a nice little shell of a program. Any problems with it? Well, if we compile and run this program with 99 as input, we see that the last line prints "-1 bottles of beer on the wall." This is because our loop condition is `i >= 0` instead of `i > 0` as it should be. Also, if we run this program with -1 as input, it doesn't yell at us, but it also doesn't print anything. This is because we're not doing any error checking on the number the user provided. Let's fix these problems like so:

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

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

    if (n < 1)
    {
        printf("Please input a positive number.\n");
        return 1;
    }

    for (int i = n; i > 0; i--)
    {
        printf("%d bottles of beer on the wall.\n", i);
        printf("%d bottles of beer.\n", i);
        printf("Take one down, pass it around.\n");
        printf("%d bottles of beer on the wall.\n", i - 1);
    }

    return 0;
}
```

Here, we also decided to explicitly return 0 at the end of the program. Not strictly necessary, but can be a good idea as our programs get more complicated.

- Question: can you use `%i`? Yes, it's synonymous with `%d`.
- Rather than jump into prefabbed code, let's rip this apart ourselves so you can see thought process behind hierarchical decomposition. In the program we've just written, the chunk of code that actually prints the lyrics feels like it can be factored out into a separate function. It "feels" this way because the chunk of code is repeated on every iteration of the loop and it takes only a single input, the number of bottles currently on the wall. If nothing else, moving this chunk of code to a separate function will help clean up `main` so that it is easier to see what it's actually doing. Let's see how it might look to move the loop, as well, to a separate function:

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

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

```
    if (n < 1)
    {
        printf("Please input a positive number.\n");
        return 1;
    }

    chorus(n);

    return 0;
}

void
chorus(int bottles)
{
    for (int i = bottles; i > 0; i--)
    {
        printf("%d bottles of beer on the wall.\n", i);
        printf("%d bottles of beer.\n", i);
        printf("Take one down, pass it around.\n");
        printf("%d bottles of beer on the wall.\n", i - 1);
    }
}
```

Now that we've done this, reading through `main` is very quick and informative: first we ask the user for some input; second, we do some a sanity check on this input; third, we use this input to print out the chorus of the song. Before, we had to actually parse the code comprising the loop to know what the program did, but now this is a detail that has been abstracted away and replaced with a function whose name alone tells us what it does.

- But there's a bug in this program that will prevent it from compiling. We forgot the function prototype! Recall that we need to tell the compiler what the `chorus` function takes as input and gives as output *before* we call it. We can do so with the following line of code placed at the top of our program:

  ```
  void chorus(int bottles);
  ```

- Still, there remains a bug. This one is less critical, but it's annoying nonetheless: at one point, we're printing "1 bottles," which is grammatically incorrect. To fix this, we need only add an if condition in the `chorus` function that will handle the case where the number of bottles of beer on the wall is only 1:

```
void
chorus(int bottles)
{
    for (int i = bottles; i > 0; i--)
    {
        if (i == 1)
        {
            printf("%d bottle of beer on the wall.\n", i);
            printf("%d bottle of beer.\n", i);
        }
        else
        {
            printf("%d bottles of beer on the wall.\n", i);
            printf("%d bottles of beer.\n", i);
        }
        printf("Take one down, pass it around.\n");
        printf("%d bottles of beer on the wall.\n", i - 1);
    }
}
```

Now our program is grammatically correct, but it feels a little clunky. We copied and pasted two `printf` statements that are identical except for one word. Let's try to affect only that one word, then, in our if-else statement:

```
void
chorus(int bottles)
{
    for (int i = bottles; i > 0; i--)
    {
        string s;
        if (i == 1)
            s = "bottle";
        else
            s = "bottles";

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

Remember that we must declare `s` outside of the if-else statement lest we run into problems of scope. To clean this up even more, we can use the *ternary operator* like so:

```
void
chorus(int bottles)
{

    for (int i = bottles; i > 0; i--)
    {
        string s = (i == 1) ? "bottle" : "bottles";

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

We still actually have one remaining grammar issue, but since it's very similar to the one we just solved, we can stop here.

- Question: can you use this `? :` shorthand for if-else if-else as well as if-else? Yes, it's possible (with some more colons), but it's much less readable, so we would advise against it.
- Question: what happens if you define a function with the same name as a function defined in one of the header files you've included. The compiler will yell at you. Because there's no notion of namespaces or packages in C, you can't reuse function names.
- Question: should we declare `s` outside the for loop so that it won't be reallocated on every iteration of the loop? Thankfully, the compiler is smart enough to realize that it only needs to allocate space for `s` once even if it's declared inside the for loop, so this isn't necessary.
- Question: can you include two libraries with the same function name? No, the compiler will yell that the function was previously declared.

4 More Problems with Scope (42:00–58:00)

- Let's take a look at `buggy4.c` to see another example of a problem with scope:

```
/**************************************************************************
 * buggy4.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Should increment a variable, but doesn't!
```

```
 * Can you find the bug?
 ********************************************************************************/

#include <stdio.h>


// function prototype
void increment(void);


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


/*
 * Tries to increment x.
 */

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

This program is so buggy that it won't even compile. The problem is that `x` is declared and initialized only in `main`, not in `increment`. We can't add 1 to `x` in `increment` because `x` doesn't exist within the scope of `increment`. Okay, so let's try fixing this by declaring `x` within `increment`:

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

This program actually will compile because `x` has now been declared. However, it hasn't been initialized, so when we try to increment it, we'll

be adding 1 to whatever garbage value is in `x` at runtime. This garbage value is whatever was left over in `x`'s memory from the last time it was used. This is definitely not what we want.

- Not only are we adding 1 to a garbage value, we're not having any effect on the original variable `x` that was declared in `main`. Because `main` and `increment` have different scopes, the `x` from `increment` is completely separate from the `x` in `main`. Thus adding 1 to the former has no effect on the latter.

5 Strings and Arrays (48:00–73:00)

- Arrays in C are collections of variables of the same type stored in contiguous memory. You can think of an array in C as a row of mailboxes in a dorm or apartment building. Each mailbox represents one element in the array and can store one variable of whatever type the array was declared to store.
- If our row has 5 mailboxes in it, the maximum length of a word we could store in this row is 4. This is because we need space for the null terminator (`\0`) which signifies the end of the string.

5.1 `string1.c`

- We've already seen that strings can be treated as arrays, so let's have some fun with them:

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

int
main(void)
{
    // ask user for string
    string s = GetString();

    int n = strlen(s);
}
```

To find the length of the string we just got from the user, we'll call a function named `strlen`. But we can't remember where this function is declared, so how can we figure it out? Recall that we can look up the manual entry for this function by typing `man strlen` at the command line. In the Synopsis section, the manual entry will tell you that `strlen` is declared in `string.h`. Also in the Synopsis section, we see that the function prototype looks as follows:

```
size_t strlen(const char *s);
```

Apparently, this function doesn't take a `string` type as an argument, but rather a `const char *`. As a teaser, the `*` means this type is a *pointer*, or actually a memory address. We'll come back to this. In fact, the `string` data type we declared in the CS50 Library is just a synonym for `const char *`. What about the `size_t`? It's another custom data type which, in this case, is synonymous with `int`. Know that you can get similar information from the C Reference linked from the course website.

- Now that we know what library to include, we can move on and use the return value of `strlen`:

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

int
main(void)
{
    // ask user for string
    string s = GetString();

    int n = strlen(s);

    for (int i = 0; i < n; i++)
    {
        printf("%c\n", s[i]);
    }
}
```

This bracket notation accesses the `i`$^{\text{th}}$ character of `s`. When we compile and run this program, we see that it prints out the string we provide, one character per line. Woohoo!

- What would have happened if we had messed up and used `i <= n` as our loop condition? When we compile and rerun, it appears we get a blank line printed after the last character of our string since this is how the compiler is interpreting `\0` in this case. Hmm, doesn't seem to be too bad. Incidentally, `strlen` doesn't count the null terminator when returning the number of characters in a string.

- How about if we use `i <= n * 100` as our loop condition? We'll be iterating off the end of our array and accessing whatever garbage values are left in memory surrounding the array. Some of these garbage values are even unprintable.

- As we move farther and farther off the end of the array, we'll eventually be touching memory that the operating system realizes isn't ours. When this happens, we'll cause a segmentation fault. We saw this last week when we wrote a function that called itself infinitely. When you see the message "core dumped," it means that a file named `core` is actually created in the directory in which you ran your program. This file allows you to trace back through the state of your program at the time it failed. More on that later when we discuss debuggers.

- As a sidenote, in this week's source code, we've included a file named `cs50.h`. Yes, this is the same one we've been including all along! Every time you've included this in a program,[3] the compiler has actually copied and pasted the contents of this file into the top of the file you wrote. By doing so, it makes sure that functions defined in the CS50 Library will be declared before you actually call them in your code. One line in `cs50.h` worth noting:

  ```
  typedef char *string;
  ```

  This line allows us to refer to the data type `char *` as `string`. If we scroll down a bit more, we can peek at a comment describing what `GetString` does:

  ```
  /*
   * Reads a line of text from standard input and returns it as a
   * string (char *), sans trailing newline character.  (Ergo, if
   * user inputs only "\n", returns "" not NULL.)  Returns NULL
   * upon error or no input whatsoever (i.e., just EOF).  Leading
   * and trailing whitespace is not ignored.  Stores string on heap
   * (via malloc); memory must be freed by caller to avoid leak.
   */

  string GetString(void);
  ```

  Notice that this documentation indicates that strings are stripped of newline characters before being returned. Also, in the event of an error or "no input," something called `NULL` is returned. Perhaps a malicious user succeeded in getting `GetString` to return even without hitting Enter. Or perhaps he's provided us a string that is too long to be stored in memory. In both those cases, `GetString` will return a special value called `NULL`.

5.2 `string2.c`

- To make things a little more interesting in lecture, the code we look at on the overhead will be stripped of comments. However, all of the comments are available in the handouts and online.

[3] And compiled it with a certain flag; more on that later.

- We read just a minute ago that it's possible for `GetString` to return a special value called `NULL`. In `string2.c`, we explicitly handle that case:

```
/**************************************************************************
 * string2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints a given string one character per line.
 *
 * Demonstrates strings as arrays of chars with slight optimization.
 **************************************************************************/

#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, n = strlen(s); i < n; i++)
        {
            printf("%c\n", s[i]);
        }
    }
}
```

All we've done is added a check to make sure that `s` isn't `NULL` before looping through its characters. In terms of design, consider for a moment why the above might be better or worse than the following:

```
for (int i = 0; n < strlen(s); i++)
{
    printf("%c\n", s[i]);
}
```

The way we had it at first, with the call to `strlen` in the initializations section of the for loop rather than the threshold section, is more efficient.

Why? If our call to `strlen` is in the threshold section of the loop, it will be executed on every iteration of the loop. In contrast, if our call to `strlen` is in the initializations section of the loop, it will only be executed once, before the first iteration of the loop. This kind of inefficiency, while not glaring or incorrect, is an example of poor design.

5.3 `array1.c`

- Examine the first line of the following program's `main` function and see if you can tell what it's doing:

```
/**************************************************************************
 * array1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Computes a student's average across 2 quizzes.
 *
 * Demonstrates C's math library.
 **************************************************************************/

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


// number of quizzes per term
#define QUIZZES 2


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

    // compute average
    float sum = 0;
    for (int i = 0; i < QUIZZES; i++)
```

```
        sum += grades[i];
    int average = (int) round(sum / QUIZZES);

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

It's declaring an array of `float` of length `QUIZZES`. When declaring an array, its length goes in between the square brackets. `QUIZZES` is a constant that we define in this line of code:

```
#define QUIZZES 2
```

After this line, the word `QUIZZES` will be treated as the number 2. Defining constants like this is a matter of not only good style (because it's more readable), but also good design (because it can be easily changed and the compiler can make some optimizations). By convention, constants are written in all capital letters.

- The first for loop in the above program initializes our array with quiz scores provided by the user. The second for loops sums up those quiz scores and calculates their average. Because we're generous, we're rounding the average up to the nearest integer.
- Why not just declare two separate variables of type `float`? In this case, it might be easy to do so, but what if we want to calculate the average of 100 quizzes instead of 2? In that case, declaring separate variables would be quite tedious and wasteful.