Computer Science 50
Fall 2008
Scribe Notes

Week 2 Wednesday: October 1, 2008
Andrew Sellergren, Anjuli Kannan

Contents

Computer Science 50
Fall 2008
Scribe Notes

Week 2 Wednesday: October 1, 2008
Andrew Sellergren, Anjuli Kannan

1 Announcements (0:00–8:00)

- David is a kid at heart.[1]
- 0 new handouts today.
- Walkthroughs are online (and are very well produced thanks to Keito and Chris). By default, they focus on the Standard Edition of the problem sets.
- The latest Office Hours are always posted online (over 100 per week!).
- If you still have Scratch Boards, please return them with a slip of paper noting your name, username, and ID number.
- "Lunch with David" on Fridays: e-mail `rsvp@cs50.net` if you'd like to join.
- Problem Set 2 will be released on Friday at 7 p.m. If you turn it in by 7:15 p.m. the same day, you will be exempt from all future problem sets and will be permitted to take over David's job as professor of the course.[2]
- Former head TF Thomas Carreiro wishes to announce a Facebook tech talk.[3]
- David calls the headset-style microphone the "Britney Spears mic." FYI.

2 Arrays and Pre-processor Directives (8:00–30:00)

- Let's say we want to write a program to average the test scores of CS 50 students.
  1. Pull up a terminal window.
  2. `demo.c`:

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

int
main{int argc, char *argv[])
{
    float quiz1;
    float quiz2;

    printf("Quiz 1: ");
```

[1] He has indicated to me that he watches *A Christmas Story* nightly.
[2] This is *still* not true.
[3] No, they won't be answering questions about the new layout. Sorry, we just have to deal with it.

```
    quiz1 = GetFloat();

    printf("Quiz 2: ");
    quiz2 = GetFloat();

    float ave = (quiz1 + quiz 2) / 2;

    printf("Ave: %f\n", ave);
}
```

Don't forget the `#include`'s and order of operations!

3. Oops, it didn't compile with the command `gcc demo.c`. What went wrong? We need to tell GCC where to find the CS 50 header file `cs50.h`. Use the flag `-lcs50`. We can also use the `-o` flag to specify the output file. However, be sure not to type something like `-o demo.c`. This will overwrite your source code![4]
4. What if we want 9 quizzes instead of 2? Should we simply copy and paste?[5]
5. Let's use a loop! And an array! See `array1.c`:

```
// number of quizzes per term
#define QUIZZES 2

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

[4] DUCY?
[5] Rhetorical question. LDO.

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

What's the deal with the `#define`? It's a way of dynamically defining variables at compile time. When you go to run GCC, the word `QUIZZES` will be replaced with the number you specify (in this case 2) everywhere that it occurs in your program. And the array `grades[]`? If we check out `howstuffworks.com` we see a great visual representation of an array. For example, an array like `int a[4]` is represented in memory as four contiguous chunks of memory, each of the same size as a single `int`. This, however, is also dangerous: something called a buffer overrun exploit takes advantage of this type of memory access.

6. Walk through the code above and make sure you know what's going on! Notice that we access the i^{th} element of an array by writing `grades[i]` and that we write `sum = sum + grades[i]` in shorthand as `sum+ += grades[i]`. In this spirit of conciseness, can you rewrite the code so that there is only one loop instead of two?
7. What is going on with the line that begins `average = (int)`? By adding 0.5 and casting the `float` to an `int`, we can round `average` to the nearest whole number. What if we didn't want to use this trick? We can also use the function `round` from the `math.h` library. Again, we'll have to tell GCC that we're going to use this library by typing the flag `-lm` at the command line. Check it out in `array2.c`!

- So our program scales to account for a greater number of quizzes, but what's the downside? We have to re-compile every time we want to change the number of quizzes. We'll tackle this problem next week when we look at memory management and the heap. For now, let's look at the potential bugs and security holes that can arise from using arrays incorrectly.[6]

3 The Problem with Arrays (30:00–38:00)

- Let's take a look at `buggy6.c`:

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

// number of quizzes per term
#define QUIZZES 2

int
main(int argc, char *argv[])
{
    float grades[QUIZZES];
```

[6] Man, David looks funny with the Britney Spears mic, doesn't he?

```
    int i;

    // ask user for scores
    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();
    }

    // print scores
    for (i = 0; i < 3; i++)
        printf("%.2f\n", grades[i]);
}
```

- What's wrong with this program? Our second `for` loop has a *magic number*: the terminating condition is `i < 3`, but it's not clear what 3 means in the context of this program. Usually this is simply a matter of poor style, but in this case, the number also disagrees with the `QUIZZES`. Now, we're printing out 3 quiz scores even though we've only prompted the user for 2! What happens when we run it?
- The first few times we run the program, we get the value `0.00` printed out as the third quiz score. What if we change the terminating condition to `i < 4`? We get `-1.52`. Now we're hacking![7]
- How about 100? 1000? 100000? In the last case, we're touching memory about 1 megabyte away from what we "own." When we do this, often we'll get a *segmentation fault* and a file called `core` will be generated to help you debug.

4 Strings as Arrays (38:00–56:00)

- We can treat strings as arrays because they, too, are stored in contiguous memory. A string is really just a series of chunks of memory, each of the same size as a `char`. After the last character in the string, there's a `NULL` character (often represented as `'\0'`) to denote the end. Let's take advantage of this in `string1.c`:

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

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

[7] You are officially a h4xx0r.

```
{
    char c;
    int i;
    string s;

    // get line of text
    s = GetString();

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

Notice that we can loop through a string just as we would an array. Each of the characters in the string is analogous to an element in an array.

- Why do we have to check `s != NULL`? Well, we can't assume that our users are goody two-shoes. They might feed us a `NULL` character by typing `Ctrl+D` when prompted. This will break our program.[8]
- Take a look at `string2.c`. Can you figure out what the optimization is? Whereas in `string1.c` we were calling `strlen{}` at every iteration of the loop, in `string2.c`, we only call it once. This is perfectly acceptable since the length of the string presumably won't change between iterations of the loops. This is an improvement in speed which we make at the expense of space (a common theme in computer science). In this case, the amount of space we use is negligible, so the optimization is a no-brainer.
- Let's tie this idea together with the concept of casting. Take a look at `capitalize.c`.
- This program only capitalizes legitimate letters. How do we achieve this? For starters, the loop is taken from `string2.c`. What about the lines below:

```
if (s[i] >= a && s[i] <= z)
    printf("%c", s[i] - (a - A));
else
    printf("%c", s[i]);
```

[8] This is a bad thing.

Type casting these `char`'s as `int`'s allows us to subtract them! We're also using an array to access each of the characters in the string. Be sure to walk through all of the code in `capitalize.c` since it ties together a lot of what we've been talking about for the last few weeks!

- Where would you go to find all this out? Try typing `man 3 <command_name>` or `man <command_name>` at the command line of your terminal window. This will bring up the Linux manual page for the particular command that you want to read more about.[9]

5 Back to Command-Line Arguments (56:00–62:00)

- We've already seen how we can tell GCC that we're going to use certain libraries by invoking the `-lcs50` and `-lm` flags. What else can we do from the command line?
- Finally, we're going to start using that `int main()` stuff you've been typing at the top of every file you write in C!
- Notice first that `argv[]` is actually an array (or, technically speaking, a vector). Its type is actually a `char *`, which means it's a *pointer*. We'll talk more about this next week. For now, let's look at `argv1.c`.
- This program simply prints out the command-line arguments that we feed it. How does it achieve this?
- Notice that we iterate from `0` to `argc` in our `for` loop. What is it that we're printing during each iteration of the loop? Each element of the array `argv[]` is actually one of the command-line arguments that we fed to the program. Thus, `argv[]` is actually an array of strings, or more technically, an array of arrays of characters.
- Next, `argv2.c` takes it a step further and prints each character of every command-line argument on separate lines. Take a look at how we do this, namely by accessing `argv[]` as a two-dimensional array, i.e. `argv[i][j]`.

6 Cryptography (62:00–79:00)

- Remember "Or fher gb qevax Ibhe binygvar!"?
- Cryptography usually relies on a *secret key* known only to the sender and receiver. In Ralphie's case, this was the Captain Midnight decoder ring. In the case of the Germans during WWII, this was the Enigma machine.
- How do we crack the code? Well, we could do a frequency analysis and try to map the most frequently occurring ciphertext character to the most

[9] OMG! I can't believe David said RTFM in lecture!

frequently occurring plaintext characters in the language (e.g. "e"). Or, we could just steal the Enigma machine.[10]

- One of the earliest examples of cryptography was the Caesar cipher. This is very easy to implement, but also very easy to crack.[11] Simply shift or rotate all of the plaintext characters in your message by a single integer value which will then become your *secret key*; if that value takes you past the letter "z", simply loop back and start counting from "a" again.

- Slightly more advanced is the Vigenère's cipher. Here, we rotate all of the plaintext characters in our message by numerous different integer values, each of which is mapped by a character in a key word or phrase. For example, if "foobar" is our key word, then the first letter of our plaintext message will be rotated by "f" or 5, the second letter will be rotated by "o" or 14, etc.

- How much better is the Vigenère's cipher than the Caesar cipher?[12] When we compare these ciphers, we talk about their keyspaces. Whereas the keyspace of the Caesar cipher is 26, the keyspace of the Vigenère's cipher is 26^n, where n is the number of letters in our key word. Comparatively, the keyspace of DES, a modern method, is 72,000,000,000,000,000! Roughly speaking, the keyspace is the worst-case number of possible keys that a malicious user would need to consider in order to crack the code by brute force.

- What's the catch? If you've ever seen the padlock icon in the bottom of your browser window, it means that it is encrypting information with the web site you are accessing. Does this mean that you are both sharing a single key? Fortunately, no. Now, there are two: a private key and a public key. The math works out such that you can encrypt information to a web site like Amazon using its public key (and vice versa, Amazon can encrypt information using *your* public key) and the only number which will *reverse* this process is the private key known only to you. All of this security relies upon the fact that computers still have trouble factoring very large prime numbers, even by brute force.

- For next week: how to factor very large prime numbers.[13]

[10] This is a terrible idea. See http://www.imdb.com/title/tt0141926/.
[11] Despite what David says, the Caesar cipher is actually unbreakable.
[12] Not better at all. Take my word for it.
[13] JK!!!111 ROFL!!!111 No, srsly, the world as we know it would end. If you figure this out, please, for everyone's sake, just use it to get rich gradually and privately.