Computer Science 50
Fall 2011
Scribe Notes

Week 5 Wednesday: October 5, 2011
Andrew Sellergren

Contents

Computer Science 50
Fall 2011
Scribe Notes

Week 5 Wednesday: October 5, 2011
Andrew Sellergren

1 Announcements and Demos (0:00–2:00, 40:00–45:00)

- On the horizon is Problem Set 5, one of David's favorites! For your first task, you will be uncovering a secret message in what appears to be nothing but jumbled colors. To do this, you'll implement a color filter much like you might've found in a cereal box when you were younger. For your second task, you'll be recovering photos that have been unintentionally (read: intentionally) deleted from a memory card. At least one of your predecessors actually made use of this program a few months after finishing CS50 when his sister accidentally formatted her memory card. He went from being "less comfortable" to salvaging 1000+ photos and saving the day!
- CS50 Lunch this Friday at 1:15 p.m.! RSVP here. We'll be hosting a dinner, as well, in future weeks.
- Quiz 0 is on Wednesday! There's no lecture on Monday. We will announce on the course website where you are to report to take the quiz (it won't be Sanders). A review session will take place this Sunday at the same time and place as the Walkthroughs. Office Hours will be held on Monday and Tuesday nights. There are four years' worth of past quizzes available on the course website for you to practice with, but keep in mind that the material has changed slightly from year to year, so don't be alarmed if there's some topic covered there that you haven't seen at all this year. Transcripts of the videos are linked from the course website, as are these scribe notes, which you should make use of![1]
- Just for fun, David uploaded the transcripts of his lectures to a free tool that creates word clouds, or visualizations in which the size of a word corresponds to the frequency with which it appears. Turns out that "just" is David's favorite word!
- The Harvard Innovation Lab is now open! This is meant to be an open space for entrepreneurial spirits to meet and discuss ideas. The week after next, we'll hold Office Hours there (complete with pizza and sodas) so you can get a feel for it, if you so choose. Shuttles will also be available to take you there, although it's not that far to walk, either.

2 The CS50 Library (2:00–40:00)

- Recall the functions that are defined by the CS50 Library:
  - `GetChar`
  - `GetDouble`
  - `GetFloat`

[1] Nothing like preaching to the choir, huh.

  - `GetInt`
  - `GetLongLong`
  - `GetString`

  We've been taking these functions for granted for several weeks now, but finally we're beginning to understand what's going on underneath the hood: for example, dynamic memory allocation on the heap.

- Now that you understand more about memory allocation, we can reveal a dirty little secret to you: the CS50 Library actually leaks memory (gasp!). When you call `malloc` to allocate memory on the heap, it is your responsibility to call `free` on that same memory so that you release it when you're done using it. Thus far, when we've used `GetString` and the other library functions, we've been calling `malloc` without ever calling `free`.

- If you've ever noticed your operating system getting slower and slower, it might be because one of your open programs has a memory leak. What this means is that it's gradually sucking up more and more of your RAM and never releasing it back to the operating system. As a result, your operating system begins to think that it is out of RAM, so it lends virtual memory (which is slower than RAM) to new programs that you open.

- If we look closer at the CS50 Library, we see that the other functions all use `GetString` to get the user's input. In every case, we check that the return value of `GetString` isn't `NULL`. If the user provides an input that is too long to be stored in memory, `GetString` will return `NULL`. If `GetString` returns `NULL`, then our wrapper functions return `INT_MAX`. This is a convention of C for functions that return integers. Because 0 and -1 and other potential error codes are valid integers that could be returned legitimately by our function, we need to find a sentinel value that is less popular that can be returned in case of error. This is where `INT_MAX` comes in: it's a number around 2 billion that is reserved for error cases since it is less likely to be a legitimate return value of an `int` function.

- Once we have checked that `GetString` has returned a non-`NULL` value, we pass the user's input, stored in `line`, to a function `sscanf` which looks for format strings within it. You can think of `sscanf` as the opposite of `printf`: it reads in as opposed to writes out. In this case, we're looking for an integer followed by a character. What we're checking, however, is if only the integer was found, in which case `sscanf` will return 1, the if condition will evaluate to true, and the integer will be returned. If both an integer and a character are captured, `sscanf` will return 2 and the user will be asked to retry. This is a clever bit of error detection: if the user types "42 f," then both `n` and `c` will be filled and we will ask him to retry; if the user types "42," then only `n` will be filled and we will return the integer 42. As you can infer, `sscanf` returns the number of placeholders it filled.

- Question: what does `sscanf` return if the user types in only a character? 0. The order of the formatting strings `%d %c` is respected so that `sscanf` has to find an integer first.
- Question: what's with the `while (true)`? It's a construction for an infinite loop. Because `true` is always true,[2] the loop will keep iterating until the user provides valid input, in which case the whole function returns.
- Question: what if the user types "42f" (no space)? `sscanf` still returns 2 because the whitespace between `%d` and `%c` is optional.
- Question: do the CS50 Library functions hang if the user provides no input? Yup. They wait indefinitely for the user to enter something valid.

2.1 scanf1.c

- `scanf1.c` demonstrates the traditional way of obtaining user input via `scanf`:

```
/**************************************************************************
 * scanf1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Reads a number from the user into an int.
 *
 * Demonstrates scanf and address-of operator.
 **************************************************************************/

#include <stdio.h>


int
main(void)
{
    int x;
    printf("Number please: ");
    scanf("%d", &x);
    printf("Thanks for the %d!\n", x);
}
```

`scanf` reads directly from keyboard input whereas `sscanf` reads from a string input. When we compile and run `scanf1`, we see that it works as long as we provide integer input. However, if we enter something like

[2] Deep.

"monkey," the program fails to detect an integer and ultimately prints whatever garbage value was in `x` to begin with. To be on the safe side in a program like this, you might choose to initialize `x` to some sentinel value like `INT_MAX` so that there's no risk you will attempt to print such a garbage value.

2.2 scanf2.c

- Now that the training wheels are off, let's try to get a string from the user without the help of the CS50 Library:

```
/**************************************************************************
 * scanf2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Reads a string from the user into memory it shouldn't.
 *
 * Demonstrates possible attack!
 **************************************************************************/

#include <stdio.h>


int
main(void)
{
    char *buffer;
    printf("String please: ");
    scanf("%s", buffer);
    printf("Thanks for the \"%s\"!\n", buffer);
}
```

- Incidentally, the fact that pointers are 32 bits on most systems is often what places restrictions on the maximum amount of RAM a computer can have. If you've ever heard of a computer that can only upgrade to 2GB of RAM, for example, it might be because its pointers have only 31 bits to store memory addresses ($2^{31} \approx 2$ billion).
- Question: are strings native to C? No. `printf` has a formatting placeholder for them, but it treats them as arrays of characters, not as a native type.
- As a sidenote, a "buffer" is really just a chunk of memory in computer-science speak.

- Assuming the user provides a string input, `scanf` will attempt to store this string in the memory pointed to by `buffer`. However, `buffer` is a pointer that hasn't been initialized with a pointee. Instead of a valid memory address, it contains some garbage value. Thus, when we try to access this garbage value as a memory address, we will probably cause a segmentation fault.
- If we try to compile this program using `make`, we're actually going to get an error regarding the uninitialized pointer. This is a safety mechanism in the form of a compiler flag that we put into place. If we compile this program using GCC instead, we will get no errors.
- To fix this, we could initialize `buffer` to `NULL` to begin with. Then we'll get "Thanks for the (null)," which will at least indicate to us that something is wrong. The real solution, however, is to initialize `buffer` with a pointer to a chunk of memory allocated by `malloc`. We might, for example, write the following:

  ```
  char *buffer = malloc(10);
  ```

  In order to use `malloc`, we need to include the `stdlib.h` library, which thus far the CS50 Library has been doing for us. Unfortunately, hard-coding the value of 10 as an argument to `malloc` isn't a great solution either. If we enter in a long enough input, we're going to overflow these 10 bytes that have been allocated for us and still cause a segmentation fault. Herein lies the motivation for the CS50 Library. In order to do this properly, we need to ask for a small chunk of memory from `malloc`, then read in the user's input one character at a time. If ever the user's input requires more space than we've allocated, we need to ask the operating system for more memory. And so on. Thankfully, the CS50 Library does all of this for us.
- Question: why did we pass `&x` to `scanf` in `scanf1.c` but `buffer` to `scanf` in `scanf2.c`? `scanf` takes a pointer for each formatting string that it attempts extract. `buffer` is already a pointer, but `x` is an `int`, so we must pass the address of `x`, or `&x` as an argument to `scanf`.

2.3 `scanf3.c`

- One final way that we could go about initializing a string is demonstrated in `scanf3.c`:

  ```
  /**************************************************************************
   * scanf3.c
   *
   * Computer Science 50
   * David J. Malan
  ```

```
 *
 * Reads a string from the user into an array (dangerously).
 *
 * Demonstrates potential buffer overflow!
 ****************************************************************************/

#include <stdio.h>


int
main(void)
{
    char buffer[16];
    printf("String please: ");
    scanf("%s", buffer);
    printf("Thanks for the \"%s\"!\n", buffer);
}
```

Here, too, we face the same problem as before when we hardcoded the value we passed to `malloc`. As soon as the user provides input that is longer than 16 characters, we're going to run the risk of causing a segmentation fault.

- Question: why do we pass `buffer` and not `&buffer` to `scanf`? `buffer`, by virtue of being an array, is already a memory address, which is exactly what `scanf` wants as an argument.
- Question: how many bytes can we store in `buffer`? 16. If instead `buffer` were an array-of-`int` of size 16, it would be 64 bytes in size. The 16 makes room for 16 of whatever data type the array stores.
- Recall that the dangers of overflowing a buffer are not just that you might cause a segmentation fault. If the buffer is a local variable, you might actually overwrite the return address of the function that's currently executing. If a malicious user is savvy enough to accomplish this, he can hijack your program and force it to execute his own injected code in what is known as a buffer overrun attack.

2.4 `pointers.c`

- Another feature of pointers is demonstrated in `pointers.c`:

```
/****************************************************************************
 * pointers.c
 *
 * Computer Science 50
 * David J. Malan
```

```
 *
 * Prints a given string one character per line.
 *
 * Demonstrates pointer arithmetic.
 ****************************************************************************/

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


int
main(void)
{
    // get line of text
   char *s = GetString();
    if (s == NULL)
        return 1;

    // print string, one character per line
    for (int i = 0, n = strlen(s); i < n; i++)
        printf("%c\n", *(s+i));

    // free string
    free(s);
}
```

We've already seen that we can access each character of a string like `s` using bracket notation. As it turns out, this bracket notation is really just syntactic sugar. That is, it's shorthand that makes it easier to express something slightly more complex. In this case, it's shorthand for the above: writing `s[i]` is actually equivalent to writing `*(s+i)`. To understand how this is true, consider the case where `i` is 0. Then `*(s+i)` is really `*(s+0)` which is really `*s`. All we're saying, then, is to dereference `s`, which we know points to the first character in the string. Thus `*s` gives us the first character of the string. We can infer, then, that `*(s+1)` will add one byte to `s` and then dereference it. So it gives us the second character in `s`, which is one byte away from the first character. And so on.

- Question: would we have to add more than `i` on each iteration (say, `4 * i`) if this were an array of integers rather than an array of characters? No. The compiler is smart enough to know that adding 1 to a memory address in this context means adding 1 times the size of the data type stored in the array.
- Question: is this syntax more efficient than bracket notation? No. The

compiler will translate bracket notation into the above, so they are effectively the same.

3 Data Forensics (45:00–56:00)

- Recall from this video that hard drives consist of spinning platters that store the 0's and 1's written by voltage fluctuations sent through a read-write head. At a higher level, though, these 0's and 1's are clustered together into files. In order to keep track of these files, some section of the hard drive is reserved for mapping filenames to memory addresses.
- Actually, a single file might contain bytes that are scattered all across the hard drive. This is called fragmentation.
- What happens when you drag a file to the trash can on Mac OS or Windows? Well, nothing. It's pretty easy to double click the trash can and restore that file that you supposedly deleted. But even if you right click the trash and "empty" it, the contents of the file aren't really deleted. Rather, the operating system simply removes that file's entry in the table that maps its name to a memory address.
- Because the contents of the deleted file remain intact on the hard drive, they can be found somewhat easily by searching the entire hard drive for that file's signature. A signature is a small chunk of bytes that uniquely identifies a file type. For example, the following signature identifies (at least with high probability) a JPEG file:

  ```
  0xff 0xd8 0xff 0xe0
  ```

- In order to truly "delete" the contents of a file, you must overwrite the bits that comprised it. One way of doing this would be to download some very large files (like episodes of The Sopranos) so that the bits of your deleted file will be reused. Another easier way would be to use the "Secure Empty Trash" option on Mac OS.
- Other ways of beefing up your security are to encrypt your hard drive and to use secure virtual memory. Encrypting your hard drive implies that a password is required to unscramble the contents of your hard drive while using secure virtual memory implies that the contents of your virtual memory, the section of the hard drive used as a substitute for RAM when you have a lot of programs open, are unreadable.

4 Structs (56:00–63:00)

4.1 `structs1.c`

- Thus far, we've worked only with data types that are native to C. But occasionally there will be a need to work with more sophisticated data

structures. One example of such a data structure is a struct, a simple example of which is defined in `structs.h`:

```
/**************************************************************************
 * structs.h
 *
 * Computer Science 50
 * David J. Malan
 *
 * Defines a student for structs{1,2}.c.
 **************************************************************************/


// structure representing a student
typedef struct
{
    int id;
    char *name;
    char *house;
}
student;
```

The `typedef struct` syntax begins our new definition of a variable type. Within the curly braces, we declare the related variables that we want to belong to this struct. Finally, after the closing curly brace, we can optionally give the struct a name. In this case, we have defined a new variable type named `student` that contains an integer and two strings representing an ID, name, and house (or dorm), respectively.

- Now that we've defined this new variable type, we can make use of it in `structs1.c`:

```
/**************************************************************************
 * structs1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Demonstrates use of structs.
 **************************************************************************/

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

```
#include "structs.h"


// class size
#define STUDENTS 3


int
main(void)
{
    // declare class
    student class[STUDENTS];

    // populate class with user's input
    for (int i = 0; i < STUDENTS; i++)
    {
        printf("Student's ID: ");
        class[i].id = GetInt();

        printf("Student's name: ");
        class[i].name = GetString();

        printf("Student's house: ");
        class[i].house = GetString();
        printf("\n");
    }

    // now print anyone in Mather
    for (int i = 0; i < STUDENTS; i++)
        if (strcmp(class[i].house, "Mather") == 0)
            printf("%s is in Mather!\n\n", class[i].name);

    // free memory
    for (int i = 0; i < STUDENTS; i++)
    {
        free(class[i].name);
        free(class[i].house);
    }
}
```

- If we compile and run `structs1.c`, it seems that we can store information related to 3 students and call out any of them that are in Mather. Nothing fancy. In fact, we could have implemented a program with the same functionality weeks ago. However, several weeks ago, we could have only made use of primitive data types, meaning that we would have been juggling numerous variables in order to keep track of the IDs, names, and

houses of 3 different students. Here, we've done so in just 1 variable: an array-of-`student`.

- So within `structs1.c`, we've declared an array named `class` that contains three instances of the variable type `student`. The rest of the program asks the user for input to populate our structs, checking for any students in Mather so that we can call them out. Notice the syntax whereby we use a period to refer to the inner elements of a struct.
- At the bottom of our program, we are being responsible in freeing the memory that was used to store the names and houses of our students. We don't need to free the IDs of the students because `GetInt` does that for us.

5 A Final Takeaway (63:00–64:00)

- Zoom and enhance does **not** work this well.