Computer Science 50
Fall 2010
Scribe Notes

Week 5 Monday: October 4, 2010
Andrew Sellergren

Contents

Computer Science 50
Fall 2010
Scribe Notes

Week 5 Monday: October 4, 2010
Andrew Sellergren

1 Announcements and Demos (5:00–13:00)

1.1 Announcements

- This is CS50.
- 1 new handout.
- Quiz 0 is coming up on Wednesday October 13! This Wednesday, we'll provide a handout that decribes in detail where you're to report (it won't be Sanders). Think of it as an opportunity to demonstrate the concepts you've mastered, not to regurgitate what you've heard in lecture. You won't be asked to write entire programs, but only a few dozen lines of code in bits and pieces. We understand the constraints of coding without the aid of a compiler to catch your mistakes. Check out the quizzes page for past quizzes and review material.
- Do avail yourself of the scribe notes on the lectures page.[1] Andrew[2] does an unparalleled job of turning David's gobbledygook, haphazard lectures into eloquent, melodious prose. You might even call him[3] The Voice of CS50.
- The regularly scheduled walkthrough this week will be a course-wide review session for the quiz. No problem set will be due next week.
- Today is the fifth Monday of the semester, so be sure to submit your grading status change requests if you have them.

1.2 Demos

- Bugs have grown increasingly complex since the first, a literal insect, was discovered inside a computer. For those of you who have worked with Windows machines, you might have seen the Blue Screen of Death a time or two. Although you still won't be able to decipher it completely, you might at least recognize that this blue screen provides some hexadecimal numbers that more than likely correspond to memory addresses.
- Other error messages are more amusing, including ones that claim that memory cannot be "read," that a program failed because of "no error," or that ask you to press a key to reboot when a keyboard is not detected. Amusingly, ATMs, electronic displays, and computers in display cases are just as prone to these errors as any other computer. And, of course, the most famous WTF[4] error is "PC LOAD LETTER," as popularized by Office Space.

[1] Yeah, I realize the pointlessness of writing that here in the scribe notes. Bite me.
[2] That's me!
[3] That's me!
[4] Stands for "where's the fun?"

2 Problem Set 4 (0:00–5:00)

- For Problem Set 4, you'll be tasked with implementing the popular logic game called Sudoku. Again you'll be handed distribution code and again you'll be asked to fill in the blanks. Playing around with the staff solution, you'll notice that we've created the game board using ASCII art as we did with the Game of Fifteen for Problem Set 3. Here, however, there's no need to clear the entire board and redraw it after each move thanks to a graphical library named `ncurses`.
- For those unfamiliar with the game, Sudoku requires a player to fill in nine 9x9 boxes with the digits 1 through 9 such that no two of the same digit occupy the same box, row, or column. In the Hacker Edition, the appropriate blanks will be highlighted in red if a number is placed incorrectly. Also in the Hacker Edition, if you hold down "h" on the keyboard, the game will progressively solve itself.
- Check out binary sudoku!

3 From Last Time (13:00–18:00)

- We represent the state of a computer's RAM using the visual of a stack with frames that get added on and lopped off as a program calls and returns from functions. The heap is the area of memory which is allocated dynamically at runtime. Uninitialized local variables which are stored on the stack may have junk values in them because their memory has been recycled. This is why it is crucial to initialize your variables before you ever use them.
- When a function returns and its stack frame is lopped off, the memory it occupied is not actually cleared. You can imagine, then, that if a function stores something like a password in a local variable and then returns, there is the potential for the password to be later hijacked.
- The classical use case for the heap is when the amount of memory a user will need is not known at compile time. In the quizzes example, it is fairly shortsighted to hardcode the number of quizzes that will be averaged before the program is run. Instead, we can prompt the user for the number of quizzes and then allocate memory on the heap as necessary using `malloc`.

4 A Closer Look at Memory (18:00–50:00)

4.1 `bar.c`

- Let's take a look at `bar.c` as an example we can use to practice debugging with GDB:

```
/**************************************************************************
 * bar.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Offers opportunities to play with pointers with GDB.
 **************************************************************************/

#include <stdio.h>


int foo(int n);
void bar(int m);

int
main(void)
{
    int a;
    char * s = "hello, world";
    printf("%s\n", &s[7]);
    a = 5;
    foo(a);
    return 0;
}

int
foo(int n)
{
    int b;
    b = n;
    b *= 2;
    bar(b);
    return b;
}

void
bar(int m)
{
    printf("Hi, I'm bar!\n");
}
```

Now that the training wheels are off, we'll be using `char *` to denote a string. As such, we can manipulate strings as we would any other pointer using what's called pointer arithmetic. Because the characters of a string are actually contiguous bytes in memory, writing `&s[7]` will give

us the substring of "hello, world" that starts at index 7. The address-of operator gives us the memory location of the character at index 7 and the remainder of the substring is contiguous in memory. So long as we index into a location before the string's null terminator, this syntax will be valid.

- If we compile `bar` and run it with GDB, setting a breakpoint on `main`, we see output like the following:

```
(gdb) break main
Breakpoint 1 at 0x80483fd: file bar.c, line 20.
```

`0x80483fd` is the memory address of `main` on the stack. What's the deal with the `0x` at the beginning of this memory addresses? It's simply a convention to indicate that this number is in *hexadecimal*. How do we count in hexadecimal? Whereas in binary you have 2 possible values for each digit and in decimal you have 10 possible values for each digit, in hexadecimal you have 16 possible values for each digit. Check out the conversion chart below:[5]

$0_{hex} = 0_{dec} = 0_{oct}$	0	0	0	0
$1_{hex} = 1_{dec} = 1_{oct}$	0	0	0	1
$2_{hex} = 2_{dec} = 2_{oct}$	0	0	1	0
$3_{hex} = 3_{dec} = 3_{oct}$	0	0	1	1
$4_{hex} = 4_{dec} = 4_{oct}$	0	1	0	0
$5_{hex} = 5_{dec} = 5_{oct}$	0	1	0	1
$6_{hex} = 6_{dec} = 6_{oct}$	0	1	1	0
$7_{hex} = 7_{dec} = 7_{oct}$	0	1	1	1
$8_{hex} = 8_{dec} = 10_{oct}$	1	0	0	0
$9_{hex} = 9_{dec} = 11_{oct}$	1	0	0	1
$A_{hex} = 10_{dec} = 12_{oct}$	1	0	1	0
$B_{hex} = 11_{dec} = 13_{oct}$	1	0	1	1
$C_{hex} = 12_{dec} = 14_{oct}$	1	1	0	0
$D_{hex} = 13_{dec} = 15_{oct}$	1	1	0	1
$E_{hex} = 14_{dec} = 16_{oct}$	1	1	1	0
$F_{hex} = 15_{dec} = 17_{oct}$	1	1	1	1

Colors in HTML are often represented in hexadecimal as RGB triples, e.g. `0xff0000` for red, `0x00ff00` for green and `0x0000ff` for blue as well as

[5] Source: Wikipedia.

everything in between. Generally, it's not going to useful to you to know where your program is located in RAM, but it might be useful to pay attention to the last few digits to know where things are relative to each other. In a few weeks, when we get to the forensics problem set, you'll see memory locations written in hexadecimal. One advantage of this is that memory addresses are much shorter than they would be if we wrote them in decimal.

- Returning to `bar`, we type `run` to get to the interesting part:

```
(gdb) run
Starting program: /home/malan/src4/bar

Breakpoint 1, main () at bar.c:20
20   char * s = "hello, world";
(gdb) list
15
16 int
17 main(void)
18 {
19   int a;
20   char * s = "hello, world";
21   printf("%s\n", &s[7]);
22   a = 5;
23   foo(a);
24   return 0;
(gdb) print a
$1 = 134513803
```

  As you can see when we print the value of `a`, it contains junk because we didn't initialize it.

- Even `s` proves to be just a number when we print it out:

```
(gdb) print s
$2 = 0x312ff4 "|}\030"
```

  Recall that pointers are actually implemented as integers, so this makes sense. This memory address is actually junk, however, because the line of code that initializes `s` hasn't executed yet. In fact, what is stored at that memory address, printed between double quotation marks after the memory address itself, is also junk.

- Question: what happens if we print `s+30`? We get the following:

```
(gdb) print s+30
$3 = 0x313012 "\037"
```

Thirty bytes past `s`, we still get junk. If you opt to take CS61 after this course, you'll get plenty of experience with this kind of low-level memory manipulation, as with one of the problem sets, you'll be asked to extract passwords from a program's binary.

- After we execute line 20, `s` takes on the value of a valid memory address that actually stores the string we're interested in:

```
(gdb) next
21   printf("%s\n", &s[7]);
(gdb) print s
$4 = 0x8048534 "hello, world"
```

- When we execute the next line of code, we see that "world" is printed out. We can also verify that accessing index 7 of our string returns the letter "w":

```
(gdb) next
world
22   a = 5;
(gdb) print s[7]
$5 = 119 'w'
```

- Now, to verify the syntax we used to grab the "world" substring, we print `&s[7]`:

```
(gdb) print &s[7]
$6 = 0x804853b "world"
```

What we've printed is `s` plus 7 bytes which turns out to be a valid string just as `s` was. GDB prints character after character starting at this memory address and only stops when it finds the null terminator.

- Printing `a` now will give a junk value until we initialize it:

```
(gdb) print a
$7 = 134513803
(gdb) next
23   foo(a);
(gdb) print a
$8 = 5
```

- Because we want to examine the state of the program within the `foo` function, we're going to step into it using the aptly named `step` command. We'll then also step into the `bar` function:

```
(gdb) step
foo (n=5) at bar.c:31
31   b = n;
(gdb) next
32   b *= 2;
(gdb) next
33   bar(b);
(gdb) print n
$9 = 5
(gdb) print b
$10 = 10
(gdb) step
bar (m=10) at bar.c:40
40   printf("Hi, I'm bar!\n");
(gdb) print m
$11 = 10
```

Using GDB, we are also able to confirm that the layout of memory is as we expect. Instead of printing out the values of variables, let's print out their memory addresses:

```
(gdb) print &a
$1 = (int *) 0xbffff678
(gdb) next
21   printf("%s\n", &s[7]);
(gdb) next
world
22   a = 5;
(gdb) next
23   foo(a);
(gdb) step
foo (n=5) at bar.c:31
31   b = n;
(gdb) print &b
$2 = (int *) 0xbffff64c
```

The memory addresses of `a` and `b` are very close in value, but it appears that `b`'s is actually less than `a`'s. Even though we think of the stack as growing upward, the memory addresses are actually getting smaller. This actually poses a security threat because if you store a very large number as a local variable in a function, you can potentially overwrite the lower frames on the stack and hijack the program.

4.2 Buffer Overrun Attacks

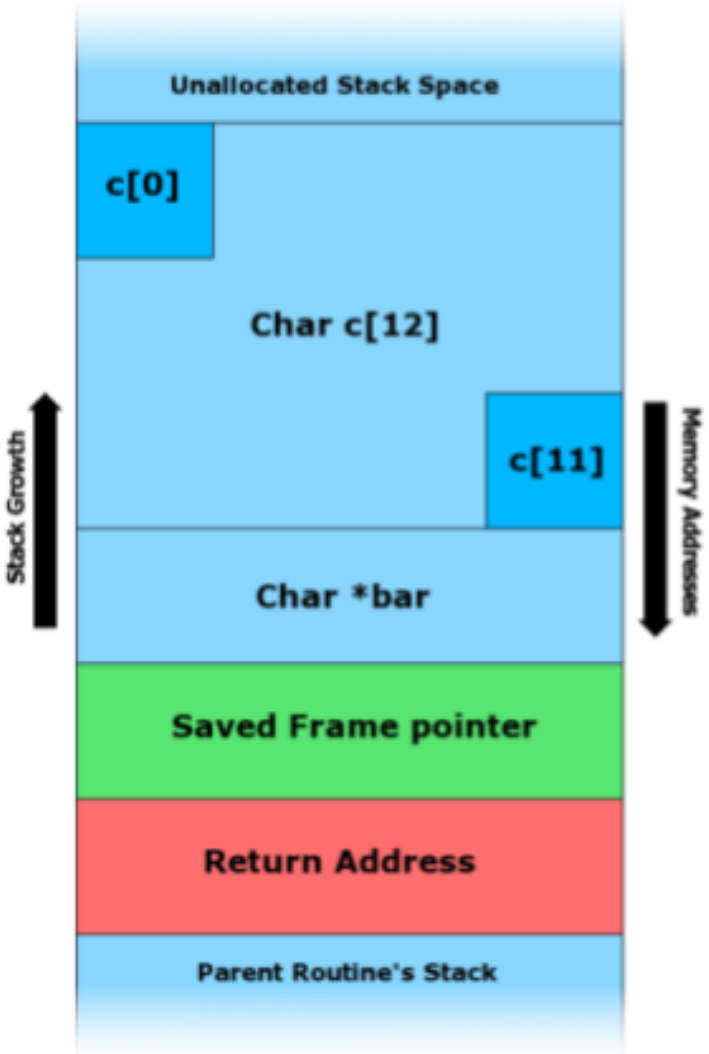
- The following code demonstrates the security vulnerability we just discussed:

```
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   memcpy(c, bar, strlen(bar));  // no bounds checking...
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

Don't be confused by the `char **argv` notation. Although we usually write `char *argv[]`, remember that arrays are also pointers, so the two are equivalent.

- One problem with the program above is that we're not checking to make sure the user provided a command-line argument. Instead, we're blindly assuming that they did and then passing it to a function named `foo`.
- Within `foo`, we declare an array `c` that can store 12 characters. We then call `memcpy`, which, as its name implies, copies the memory from the user-provided command-line argument into `c`. However, we do no bounds checking on `c`, so if the user-provided command-line argument is longer than 12 characters, we'll end up overwriting memory that isn't ours. If the command-line argument is extremely long, say, 1000 characters, our program might fail with a segmentation fault. However, if it's not much longer than 12 characters, we can overwrite important information on the stack, including the return address of the function that called `foo`, without causing the program to fail. We can visualize this like so:[6]

[6]Source: Wikipedia.



As the red rectangle indicates, the stack is used to store not only function frames and parameters, but also a function's return address. This way, when a function finishes executing, the program will know where in memory to return in order to continue.

- As you can see from this diagram, if we write more than 12 characters to `c`, we're going to first overwrite the value of `bar` followed by something called the saved frame pointer, which allows the computer to remember where in RAM it currently is. Next, we'll overwrite or corrupt the return address. If an adversary knows where on the stack this return address is,

he can overwrite it with a valid but different value which contains some malicious code he wants to execute. Then, instead of returning to the correct memory address, the program will jump to the malicious code instead.

- Many of these security vulnerabilities are discovered by trial and error. If you can cause a program to crash, you've discovered a bug that you can potentially exploit with a little more savvy.
- Buffer overrun attacks are only possible in a few low-level languages such as C. More modern programming languages like Java have built-in security features which prevent them.
- Question: what if you know exactly how much memory you're going to need, do you still need to do bounds checking? Theoretically, yes, but in the real world, you're not going to be the only programmer on a project or the only user of a program, so you need to account for malicious or unknowing users.
- Question: what is a parameter? It's an argument provided to a function.
- Question: to recap, what is stored on the stack and what is stored on the heap? Local variables, function arguments, and return addresses are stored on the stack. Anything that you use `malloc` to store goes on the heap.

5 Structs (50:00–68:00)

5.1 `structs1.c`

- Before we go over the syntax for defining a `struct`, let's take a look at `structs1.c` to see how to use one:

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

So far we've only talked about primitive data structures, but what if we want to store a chunk of related information about a single entity? For example, a student has a name, a dorm, a phone number, etc. Can we collect all these pieces of information into a single data structure?

- Even though this is a fairly straightforward definition of a `student` type, we've abstracted it away into a separate header file so that it might be

used by many different programs, including `structs1.c` and `structs2.c`. These will include this header file by writing:

```
#include "structs.h"
```

Here we're using quotation marks instead of angle brackets because the header file is local rather than on the server.

- We define the `student` type within `structs.h` using the `typedef` command:

```
/**************************************************************************
 * 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.

- 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.

- Question: how is a struct stored in memory? A struct essentially contains enough space for its component data types to be contiguous in memory. In the case of `student`, an `int` is stored next to two `char *`'s for a total of 12 bytes chunked together.[7]

[7] It's possible that the compiler will give you more than 12 bytes of memory so that your data types line up cleanly, but we won't concern ourselves with that.

- Question: what happens when we declare an array of `student`'s? If we ask for an array of size 3, then we're getting 3 structs that are contiguous in memory, for a total chunk size of 36 bytes.
- For the past few weeks, we've actually been writing buggy code whenever we call `GetString` from the CS50 Library. Under the hood, `GetString` calls `malloc` in order to store the string on the heap. But because memory on the heap isn't freed until we explicitly do so, our programs have been leaking memory. From now on, we'll need to call `free` on any data type that's been stored on the heap and that we're done using.
- If we compile and run `structs1`, we can, in fact, store information related to 3 students and call out any of them that are in Mather.

5.2 `structs2.c`

- `structs2.c` is slightly more compelling because it enables us to store more permanently the data we've collected from the user:

```
/**************************************************************************
 * structs.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);

    // let's save these students to disk
    FILE *fp = fopen("database", "w");
    if (fp != NULL)
    {
        for (int i = 0; i < STUDENTS; i++)
        {
            fprintf(fp, "%d\n", class[i].id);
            fprintf(fp, "%s\n", class[i].name);
            fprintf(fp, "%s\n", class[i].house);
        }
        fclose(fp);
    }

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

This program is exactly the same as `structs1.c` except for a chunk of code at the bottom. `fopen` takes two arguments: a filename and a file mode. If there exists no file with the given filename, it creates a new one so long as the file mode is set to "w" for write. Assuming it was successful in creating the file, `fopen` returns a pointer-to-`FILE`. If it wasn't successful, it will return `NULL`, which is why we check for it before attempting to write to it.

- Once we're sure we have a valid file pointer, we loop over our array of structs and access the elements of each struct. Notice that instead of printing them out however, we're passing them to a function called `fprintf` which writes them to the file pointer we just created.
- Generally speaking, when you quit a program without having explicitly freed the memory you allocated, it will still be returned to the operating system. However, best practice is to release it by passing it to `free`. We will require it from here on out!
- When we make and run `structs2` and input the same data as before, we see that a file named `database` is created.

6 The CS50 Library (68:00–75:00)

- Let's peek under the hood of the CS50 Library, particularly at `GetString`:

```
/*
 * Reads a line of text from standard input and returns it as a string,
 * sans trailing newline character.  (Ergo, if user inputs only "\n",
 * returns "" not NULL.)  Leading and trailing whitespace is not ignored.
 * Returns NULL upon error or no input whatsoever (i.e., just EOF).
 */

string
GetString(void)
{
    // growable buffer for chars
    string buffer = NULL;

    // capacity of buffer
    unsigned int capacity = 0;

    // number of chars actually in buffer
    unsigned int n = 0;

    // character read or EOF
    int c;

    // iteratively get chars from standard input
    while ((c = fgetc(stdin)) != '\n' && c != EOF)
    {
        // grow buffer if necessary
        if (n + 1 > capacity)
        {
            // determine new capacity: start at CAPACITY then double
```

```
            if (capacity == 0)
                capacity = CAPACITY;
            else if (capacity <= (UINT_MAX / 2))
                capacity *= 2;
            else
            {
                free(buffer);
                return NULL;
            }

            // extend buffer's capacity
            string temp = realloc(buffer, capacity * sizeof(char));
            if (temp == NULL)
            {
                free(buffer);
                return NULL;
            }
            buffer = temp;
        }

        // append current character to buffer
        buffer[n++] = c;
    }

    // return NULL if user provided no input
    if (n == 0 && c == EOF)
        return NULL;

    // minimize buffer
    string minimal = malloc((n + 1) * sizeof(char));
    strncpy(minimal, buffer, n);
    free(buffer);

    // terminate string
    minimal[n] = '\0';

    // return string
    return minimal;
}
```

To begin, we assume that the user's input will be 0 bytes in length. This is safe because we know we're going to grow our storage in kind with the user's input, so we might as well err on the lower side to begin with. We're keeping track of both the potential size of the user's input in `capacity` and the actual size of the user's input in `n`. Both are of type `unsigned int` because we know that the size of the user's input will never be negative.

- The function `fgetc` is grabbing one character at a time from `stdin` which essentially represents the keyboard. We then check if that character is the special character `EOF` which signifies that the user has finished giving input, perhaps via the return key. At the end of the loop, we add this character to the buffer which we'll eventually return. What we glossed over in the beginning were the lines of code that check if adding another character to the buffer would overflow it and, if so, ask for more memory for the buffer.
- In addition to `malloc`, we call `realloc` which reuses and expands the memory we've already asked for.
- Although the other functions in the CS50 Library return different data types, you'll notice that most of them depend on `GetString`.
- The major takeaway here is that freeing memory is extremely important. The reason your computer may feel like it's crawling if it's been left on for a long time is that programs you've executed failed to free some of the memory they were using. As a result, your operating system has less memory to work with. Besides the performance issues of failing to free memory, there are also security implications. More on that next time.