Computer Science 50
Fall 2009
Scribe Notes

Week 4 Wednesday: September 30, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 4 Wednesday: September 30, 2009
Andrew Sellergren

1 Geek Humor (0:00–10:00)

- I wish this were a joke.
- We should temper our teasing by noting that Microsoft has graciously supported us via an MSDN Academic Alliance, which allows you to download many different flavors of Windows, including Windows 7.
- If you're going to host a launch party, don't forget your notes. The sad part, honestly, is that all of the host parties have already filled up. Seriously, who is going along with this? Better to go to Harvard Hungama this Saturday in Lowell Dining Hall.
- We also have access to virtual machine software, which allows you to run one operating system inside of another without the pain of dual booting.

2 Announcements (10:00–12:00)

- This is CS 50.
- 0 new handouts.
- Regarding questions pertaining to your code: now that you've been empowered to use GDB, you are encouraged to begin the debugging process yourself before turning to us via office hours, the bulletin board, or the e-mail list. That's not to say, of course, that we won't help you when you are still stuck, but simply that you should at least begin to tackle the problem on your own. Check out this quick reference for some of the commands.
- The online grades tool is now ready. This is meant to be a sanity check so that you can be sure the official grades we recorded are those that we assigned to you on your problem sets.

3 More on Memory (12:00–65:00)

- Recall that we are allocated 32 bits for an `int` that we declare, 8 bits for a `char`, 64 bits for a `double`, etc. When we declare an array of 4 `int`'s, we actually get 4 32-bit chunks of contiguous memory (plus a 5th 32-bit chunk to store the null terminator). What's stored in that memory to begin with? Junk.
- We can start thinking of the name of an array as representing the array's address in memory. Because memory in an array is contiguous, we can calculate each element's location based on its index and the size of the data type. This gives us *random access* to the array's contents.

- Now that the training wheels are off, we can begin to talk about a string as a `char *`, which, like an array, is really just a pointer to the first character of the string. The remainder of the string is stored contiguously in memory, so we have random access to each of its characters just like we do to each of an array's elements.

3.1 Pointer Arithmetic

- Let's take a look at `pointers1.c`:

```
/**************************************************************************
 * pointers1.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(int argc, char *argv[])
{
    // 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);
}
```

What exactly is being returned by `GetString()`? It's not actually the all of the characters that make up the string, but rather a pointer to that string. `GetString()` is actually assigning a chunk of memory based on

how many characters the user types and then returning to the program the memory address of the first byte of that string.

- Why are we checking for `NULL`? If the user hits Ctrl + D (the EOF signal) we want to be sure we're not trying to manipulate data that doesn't exist. Just as importantly, however—another corner case—if the user gives us a really long string, `GetString()` will return `NULL` if there isn't enough memory available to store it. We can figure out that it does this by reading `cs50.h`, the header file which contains the function declarations for CS 50's library.

- Recall that the second step for using an external library's code in your programs (the first being to include the header file) is to link to the library at compile-time, as with `-lcs50` or `-lm`. This tells the compiler to add in the actual function definitions that have already been compiled, probably into a `.o` file.

- The declaration of `GetString()` tells us that it returns an empty string (“”) when the user hits Enter, not `NULL`. The empty string is actually represented in memory as `'\0'` whereas `NULL` is not represented as anything in memory.

- Also, the newline character is stripped from the user's input before it is returned by `GetString()`.

- How do we make `string` a synonym for `char *`? Like so:

  ```
  typedef char *string;
  ```

  FYI, it's not necessary to have the asterisk right next to the data type it's pointing to, but it does make it clearer.

- What's going on in the for loop of `pointers1.c`? First things first, the value of `n`, as assigned by `strlen`, will be the length of the string in human terms, e.g. 3 for “foo.” And if we're storing “foo” in `s`, and, let's say the characters `'f'`, `'o'`, `'o'`, and `'\0'` are stored at memory locations 0x10, 0x11, 0x12, and 0x13 (separated by single bytes because each `char` is only 1 byte), then what's stored in `s` is actually 0x10. So now imagine we're on the second iteration of the loop and `i` equals 1. The expression `(s+i)` will then evaluate to 0x11 and the asterisk in front tells us to “go to the memory address that follows,” so we check memory address 0x11 and `o` gets printed out.

- As we'll find out, this *pointer arithmetic* takes into account what data type the pointer is pointing to. So the same expression, `(s+i)`, would work to grab each element of an array of `int`'s pointed to by `s`.

- Incidentally, note that we initialize `n` to the length of string `s` so that we don't have to call the function on every iteration of the loop.

- What does `free(s)` accomplish? `GetString()` is actually pretty poorly implemented. Since we don't know how much memory the program is going to need to store the arbitrary number of characters typed by the user, we have to allocate more than is necessary. And, in C, if allocated memory is not explicitly freed, so to speak, then the program assumes it's still being used. This is called a memory leak. If you've ever left your computer on for days at a time (or Firefox open for an extended period of time, ugh), you might have noticed that more and more of your computer's RAM is being taken up but not actually being used. This is due to memory leaks.
- To rectify this problem, we'll begin calling the function `free()`, which decommissions the memory allocated by a call to `malloc()`, as in `GetString()`.
- `pointers2.c` is an example of what we alluded to earlier, namely that pointer arithmetic works on arrays of data types other than strings, like `int`'s for example:

```
/**************************************************************************
 * pointers2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Iterates over an array of ints.
 *
 * Demonstrates pointer arithmetic.
 **************************************************************************/

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


int
main(int argc, char *argv[])
{
    int numbers[] = {1, 2, 3, 4, 5};

    printf("Size of array is %d.\n", sizeof(numbers));
    printf("Size of each element is %d.\n", sizeof(numbers[0]));
    for (int i = 0, n = sizeof(numbers) / sizeof(numbers[0]); i < n; i++)
        printf("%d\n", *(numbers+i));
}
```

For now, ignore the initialization of `n` and pretend that the terminating condition of the loop reads `i < 5`. In other words, it loops as many times

as there are elements in `numbers[]`. When we run the program, we see that it actually prints out the elements of `numbers[]`. Despite how it appears, this pointer arithmetic is not buggy. Even though we're only adding 1 to the memory address of `numbers` on each iteration of the loop, GCC knows that that 1 refers not to 1 byte but to 1 `int`, or 4 bytes. So that's the amount that it increments by.

3.2 Copying Strings

- Let's say we want to write a program that copies one string to another and then capitalizes the new version. Take a look at `copy1.c`:

```
/****************************************************************************
 * copy1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Tries and fails to copy two strings.
 *
 * Demonstrates strings as pointers to arrays.
 ****************************************************************************/

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


int
main(int argc, char *argv[])
{
    // get line of text
    printf("Say something: ");
    char *s1 = GetString();
    if (s1 == NULL)
        return 1;

    // try (and fail) to copy string
    char *s2 = s1;

    // change "copy"
    printf("Capitalizing copy...\n");
    if (strlen(s2) > 0)
        s2[0] = toupper(s2[0]);
```

```
    // print original and "copy"
    printf("Original: %s\n", s1);
    printf("Copy:     %s\n", s2);

    // free memory
    free(s1);
}
```

When we run this program, however, we see that both the original string and the copy are capitalized. The problem is that when we initialize `s2`, we assign to it the value of `s1`, which is simply the memory address of the string. Thus both `s1` and `s2` have the same memory address stored, so they both point to the same string. If we modify the contents of memory pointed to by `s2`, we also modify the contents of memory pointed to by `s1`.

- How do we fix this? We need a new chunk of memory which we can fill with the original string so that it is truly a copy. This we accomplish in `copy2.c`:

```
/**************************************************************************
 * copy2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Copies a string.
 *
 * Demonstrates strings as pointers to arrays.
 **************************************************************************/

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


int
main(int argc, char *argv[])
{
    // get line of text
    printf("Say something: ");
    char *s1 = GetString();
    if (s1 == NULL)
```

```
        return 1;

    // allocate enough space for copy
    char *s2 = malloc((strlen(s1) + 1) * sizeof(char));
    if (s2 == NULL)
        return 1;

    // copy string
    int n = strlen(s1);
    for (int i = 0; i < n; i++)
        s2[i] = s1[i];
    s2[n] = '\0';

    // change copy
    printf("Capitalizing copy...\n");
    if (strlen(s2) > 0)
        s2[0] = toupper(s2[0]);

    // print original and copy
    printf("Original: %s\n", s1);
    printf("Copy:     %s\n", s2);

    // free memory
    free(s1);
    free(s2);
}
```

Now, we're beginning to empower you with the tools to take user input using your own methods rather than relying on CS 50's library. The primary tool for this will be `malloc()`, which will give you the memory you need to store the user's input. `malloc()` takes as its only argument a number of bytes. If we want to dynamically figure out the number of bytes in the user's input, we do so by knowing that each character in the string is a single byte and that the number of characters in the string is the length of the string plus one extra character for the null terminator. Thus, `strlen(s1) + 1)`. To be explicit, we'll also multiply this by `sizeof(char)`, even though we know it to be 1. If we're talk about a string "foo," then we're passing the value 4 to `malloc()`.

- After we allocate the required memory, we iterate over the characters of `s1` and copy them to `s2`. We explicitly add a null terminator although we could simply iterate until `i <= n` to do the same. The former approach might be better, though, because it handles the case where `s1` is corrupt.
- Why do we check that `strlen(s2) > 0` before capitalizing its first letter? If there's nothing in `s2`, then we'll be asking for the first character of nothing, or in other words touching memory that isn't explicitly ours,

which would likely cause a seg fault. Note that these kinds of errors are often elusive because they appear only intermittently. As David showed in lecture, we can sometimes get away with iterating over the first 100 elements of `s1` even when the actual string length is considerably less. If we bump that number up to much more than 100, we can cause a seg fault. The size of `core`, which we can view by executing the `ls -lh` command, is almost half a megabyte, which corresponds to the amount of memory the program was using at the time it forcibly quit. We can examine `core`, by running the following command:

```
gdb <name of program> core
```

- Let's use GDB to step through `copy2`:

```
29          char *s2 = malloc((strlen(s1) + 1) * sizeof(char));
(gdb) p s2
$2 = 0xbf9f0518 "x\005\237?P??\001"
(gdb) n
30          if (s2 == NULL)
(gdb) p s2
$3 = 0x804a008 "\0202??\0202??"
```

Notice that after the call to `malloc`, we have different garbage being stored in `s2`. This is because `malloc` has assigned to it a new memory address which is still uninitialized.

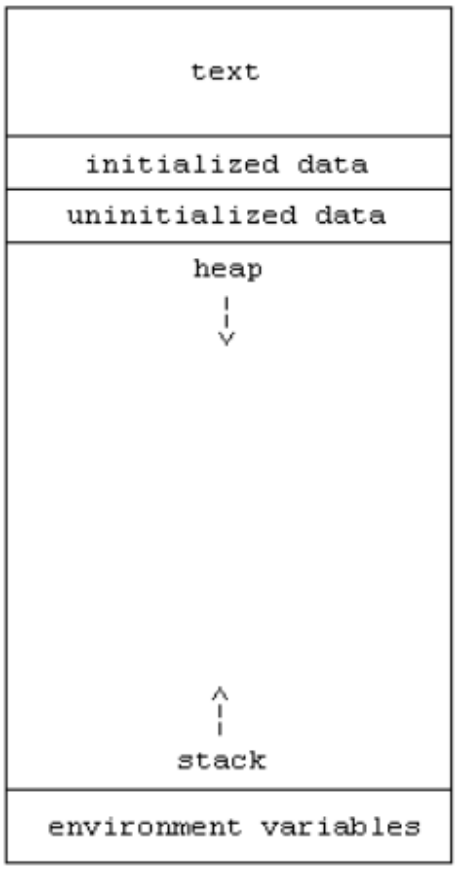
- If we step through the first few iterations of the loop and then print `s2`, we see that it does get assigned the proper value:

```
(gdb) display s2
1: s2 = 0x804a008 "\020\202?\020\202?"
(gdb) n
36              s2[i] = s1[i];
1: s2 = 0x804a008 "\020\202?\020\202?"
(gdb) n
35          for (int i = 0; i < 1000; i++)
1: s2 = 0x804a008 "f\202?\020\202?"
(gdb) n
36              s2[i] = s1[i];
1: s2 = 0x804a008 "f\202?\020\202?"
(gdb) n
35          for (int i = 0; i < 1000; i++)
1: s2 = 0x804a008 "fo?\020\202?"
(gdb) n
36              s2[i] = s1[i];
1: s2 = 0x804a008 "fo?\020\202?"
```

```
(gdb) n
35          for (int i = 0; i < 1000; i++)
1: s2 = 0x804a008 "foo?\020\202?"
(gdb) n
36              s2[i] = s1[i];
1: s2 = 0x804a008 "foo?\020\202?"
(gdb) n
35          for (int i = 0; i < 1000; i++)
1: s2 = 0x804a008 "foo"
```

Note that `s2` does in fact store "foo," which displays properly in GDB as soon as the null terminator is added to the end. If we execute `continue`, now, we'll get all sorts of errors relating to memory mismanagement.

- Recall our depiction of the stack and heap:



Previously, we noted that it was possible for the stack to overrun the heap and cause a seg fault if we called a recursive function too many time

because our finite amount of memory would be exhausted by frame after frame being allocated for each function call. What exactly is the heap? For our purposes now, it is where dynamically allocated memory is taken from. So when you call `malloc()`, the chunk of memory that is allocated comes from the heap. In the same way that the stack can overrun the heap, the heap can overrun the stack and cause a seg fault.

- What's on top (at least pictorially) of the heap? Uninitialized data and initialized data. This includes things like global variables, e.g. `board[][]` and `d` in `fifteen.c`. What's interesting is that because we've declared them one after the other and not initialized either, they get stored in contiguous memory above the heap. If you were to say, iterate past the bounds of `board`, then, you might overwrite the value of `d`.
- What is the text segment of the program? The actual 0's and 1's that compose your program on disk. The computer has to be able to read those bits from somewhere, so it puts them at the very top of memory.
- One of the disadvantages of having this very regimented layout of memory is that it's predictable. Malicious users can exploit this in a number of ways, the most common of which is to inject executable code into a program. Software cracks are generally the result of this kind of exploitation in which a user has figured out how to circumvent the serial number check, for example. "Jailbreaking" the iPhone is a good example of this, as it exploits some Apple code that fails to check the bounds of an array. In general, one of the approaches to exploiting a program is to simply "bang on it," by which we mean give it inputs the programmer might not have been expecting. We encourage you to do this to your own code to look for possible bugs (because we the staff will do the same!). If a malicious user can induce a program to crash, then he can begin to examine where it crashed and possibly use this as an entry point for his attack.
- In the case of the iPhone, the cat-and-mouse game continues. The exploit was quickly patched, only to be broken shortly thereafter. The advantage goes to the malicious users because they need only find a single hole whereas the programmers behind it have to find *all* the holes in order to plug them.
- Check out The Evolution of a Programmer which speaks to the problem of overengineering solutions to problems.

4 CS 50's Library Revisited (65:00–78:00)

- `cs50.h` contains instructions for how to compile your programs on an environment other than NICE, if you ever feel so inclined. Effectively, you'll be creating a `.o` file and then moving it to a designated location on your Unix or Linux system.

- Let's take a look at the source code for `GetInt()`:

```
/*
 * int
 * GetInt()
 *
 * Reads a line of text from standard input and returns it as an
 * int in the range of [-2^31 + 1, 2^31 - 2], if possible; if text
 * does not represent such an int, user is prompted to retry.  Leading
 * and trailing whitespace is ignored.  For simplicity, overflow is not
 * detected.  If line can't be read, returns INT_MAX.
 */

int
GetInt()
{
    // try to get an int from user
    while (true)
    {
        // get line of text, returning INT_MAX on failure
        string line = GetString();
        if (line == NULL)
            return INT_MAX;

        // return an int if only an int (possibly with
        // leading and/or trailing whitespace) was provided
        int n; char c;
        if (sscanf(line, " %d %c", &n, &c) == 1)
        {
            free(line);
            return n;
        }
        else
        {
            free(line);
            printf("Retry: ");
        }
    }
}
```

  The syntax `while(1)` induces an infinite loop, which in this case isn't really a bad thing since we explicitly break out of it by executing the `return` statement. You'll notice that almost all of the functions in CS 50's library use `GetString()`. This is consistent with our suggestion to you that you factor out common code instead of copying and pasting it.

- Why do we return `INT_MAX` instead of, say, 0 on error? Well, if we returned

0 then the user would never be able to type 0. Instead we chose the largest `int` possible, the idea being that we would simply lower by one the range of possible values that can be stored in an `int`. This would suggest that you should've been checking all this time not whether `GetInt()` returned `NULL`, but whether it returned `INT_MAX`. For the last few weeks, however, this has been beyond the scope of our computer science knowledge, so don't worry, we forgive you.

- Take a look at the call to `sscanf()`, which is used to store the user's input. Notice we're providing it with two pointers by using the `&`, or "address-of" operator, for `n` and `c`. We also place a space in front of the format specifier to indicate any number of spaces that the user types. `sscanf()` returns the number of items successfully matched. If input doesn't match the type of the variable in which it is to be stored, the match is unsuccessful. Because the `%d` comes first, this is the first data type it tries to match. So if no integer is given at all, then `sscanf()` returns 0. Why the `%c` then? If the user tries to get fancy and provides both a number and a string, then `sscanf()` will return 2 because two matches were successfully made. That's why we're checking that the return value is exactly one, which means we matched an integer and *only* an integer.

- Notice that we're only calling `free()` on `line`. This is because you can only free memory that was allocated on the heap. `n` and `c` are local variables. What's useful about the heap is that its data persists even after functions return.

- Finally, let's examing `GetString()`, which does the grunt work of the library:

```
/*
 * string
 * 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()
{
    // 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;
}
```

The function `fgetc()` is used to grab each character of the user's input as long as it isn't a newline character or the EOF character.

- The first if condition within this loop is checking that `buffer` is large enough to store one more character. If it's not, then we need to grow it dynamically. Here, we're doubling it in size every time it needs to grow. We do this via a call to `realloc()`, which, as its name implies, reallocates memory that's already been in use (if possible).
- The three lines of code commented by "minimize buffer" do the job of returning only the number of bytes minimally needed to store the user's input. This is to avoid wasting resources.