Computer Science 50 | Week 4 Wednesday: September 28, 2011
Fall 2011 | Andrew Sellergren
Scribe Notes

Contents

1 Announcements and Demos (0:00–1:00, 3:00–5:00)

- If you haven't already, check out the Google search for recursion and see if you get the joke.
- Improved wifi is now available here in Sanders. The SSID is cs50 and the password is 12345678. Wireless has also been bolstered in the house dining halls to prepare for the onslaught of CS50 Office Hours.
- If David hasn't responded to your questions about Pass/Fail, don't give up! He's a very busy man,[1] so feel free to ping him again if necessary.
- We'll soon be having some Pointer Fun With Binky thanks to Nick Parlante, a CS professor at Stanford.[2]

2 From Last Time (1:00–3:00)

- We looked at a few different sorting algorithms and their big O runtimes. Here is a bit of vocabulary that will allow you to refer to big O runtimes:

$O(1)$	"constant"
$O(\log n)$	"logarithmic"
$O(n)$	"linear"
$O(n \log n)$	"supralinear"
$O(n^2)$	"quadratic"
$O(n^c)$	"polynomial"
$O(n!)$	"factorial"

  Thus far, linear runtimes have been mostly unrealistic whereas quadratic runtimes have been exceedingly slow. So-called supralinear runtimes are something of a sweet spot in between.

3 Pointers (3:00–13:00)

- Recall our problems in the past few weeks involving writing a swap function that actually worked. Let's reenact this problem using pseudocode:

```
a = OJ;
b = milk;
```

  `a` and `b` represent the cups of orange juice and milk onstage. Our goal is to swap them so that `a` contains milk and `b` contains orange juice.

- If we take the following approach, we're going to get a nasty OJ+milk combination:

[1] Wii Tennis is super important.
[2] And a consultant at Google!

```
a = b;
b = a;
```

Once we pour the orange juice into the milk, we can't possibly hope to separate out just the milk in order to pour it into the other cup.

- We can, however, achieve a proper swap if we make use of a temporary variable, represented by an extra, empty cup onstage. Now we can pour the OJ into the extra, empty cup (`tmp`), then pour the milk into the cup that originally held the OJ (`a`), and finally pour the OJ into the cup that originally held the milk (`b`):

```
tmp = a;
a = b;
b = tmp;
```

- This seems to work as is, but what happens when we factor it out into a separate function?

```
void
swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

When we call this function, we're actually passing in copies of `a` and `b`. So we our original two cups of OJ and milk remain untouched. Meanwhile, we have two separate cups of OJ and milk which will be properly swapped. As soon as `swap` returns, though, we forget about the cups that have been swapped and go back to the original two cups that haven't been touched. If we want `swap` to work correctly, we actually need to hand it the original two cups, not copies of them. How do we do this programmatically?

- In the context of our program, the cups `a` and `b` are more or less bolted to the table. We can't really pass them to the `swap` function because they live in `main`'s memory. However, we can pass *references* to `a` and `b` so that `swap` can actually find them in `main`'s memory. You can visualize this as passing two pieces of paper to `swap` which have the locations, or the memory addresses, of `a` and `b` written on them. Now `swap` is able to find `a` and `b` and swap their values in their original locations.

- How much does this process cost us? The pieces of paper that we passed actually represent *pointers*, or memory addresses, each of which is 32 bits in size.[3]

[3]Assuming a 32-bit architecture. If the architecture is 64-bit, then each pointer is 64 bits in size.

- In order to pass pointers to our `swap` function, we need to modify its definition slightly:

```
void
swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

4 Hexadecimal (13:00–19:00)

- We've already learned how to count in decimal[4] and binary, but we need to learn a third base system: base-16 or hexadecimal. Since we now know that the smallest practical unit of memory is a byte, each number in binary should probably be represented with 8 bits like so:

```
00000000
00000001
00000010
```

- Hexadecimal is the base system typically used for expressing memory addresses. Although generally we won't care what number a memory address is, we do care that *is* a memory address, so it's important that we learn to recognize hexadecimal so that we can recognize memory addresses.

- Hexadecimal numbers conventionally begin with `0x`. Whereas in binary there are 2 possible digits and in decimal there are 10 possible digits, in hexadecimal there are 16 possible digits. Where do the extra 6 digits come from? Once we get to 9, we begin using the letters A through F:

```
0x00
0x01
0x02
0x03
0x04
0x05
0x06
0x07
0x08
0x09
0x0A
0x0B
0x0C
```

[4]Hopefully before this class.

```
0x0D
0x0E
0x0F
0x10
```

- Interestingly, hexadecimal is used to represent not only memory addresses, but also RGB colors. For example, 0xFF0000 is the color red in HTML. The first two digits represent the amount of red, the second two digits represent the amount of green, and the third two digits represent the amount of blue. Hence, RGB.

5 String Comparison (19:00–46:00)

5.1 `compare1.c`

- Let's try to hammer out a program we'll call `compare1.c` that compares two strings passed to us by the user and announces whether they are equal or not:

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

int
main(void)
{
    printf(``Say something: '');
    string s1 = GetString();

    printf(``Say something again: '');
    string s2 = GetString();

    if (s1 == s2)
        printf(``You typed the same thing!\n'');
    else
        printf(``You typed different things!\n'');
}
```

Seems like it should work as intended.[5] When we compile and run this program, typing "hello" at both prompts, the program tells us that we typed different things. What's going on? Turns out we can't compare strings using the `==` operator. When we peeked under the hood of the CS50 Library, we found that `string` is just a synonym for `char *`. The `*` indicates that this is a pointer type. We've just learned that pointers are memory addresses, so when we ask if `s1` is equal to `s2`, we're actually

[5]You know that whenever I say that, I'm about to say, "But it doesn't! (gasp)" Sad how predictable I am.

asking if these two memory addresses are the same. Since they are clearly different string instances, their memory addresses aren't the same.

- If strings are actually just 32-bit pointers, then how does `GetString` work? Based on the length of the text the user provides, `GetString` allocates a contiguous chunk of memory large enough to hold this text, plus an extra byte to store the null terminator. To allocate this chunk of memory, `GetString` calls `malloc`. But if `s1` and `s2` are only 32 bits each, they can't possibly store all of the user's text (unless somehow it was only 3 or fewer characters long). So after `GetString` allocates the memory for the user's text, it returns the address of that memory so that we can find it later. In fact, it returns only the address of the first byte of the memory that it allocated, which store only the first character of the user-provided text; the other bytes are contiguous, so we can easily find the other characters the user typed simply by iterating until we find the null terminator.

- Let's actually examine `s1` and `s2` as pointers:

  ```
  printf(‘‘s1 is %p’’, s1);
  printf(‘‘s2 is %p’’, s2);
  ```

  Now when we compile and run this program, we get the following (among the usual output):

  ```
  s1 is 0x8856030
  s2 is 0x8856040
  ```

  As you can see, although these hexadecimal values are very close, they are not equal.

5.2 `compare2.c`

- So how do we compare strings, then? Well, we can probably guess that there's a built-in function for that. To find out if that's the case, we can navigate to the C Reference on the course website and browse around. Indeed, we see that there is a function named `strcmp` whose purpose is to "compare two strings."

- The description of `strcmp` indicates that it takes two `char *` as arguments. Actually, they're `const char *`. The `const` simply means that this function will not (and cannot) modify the strings we pass to it. As we scroll down, we see that this function doesn't return a boolean, but rather an integer value. This integer value is 0 if the strings are equal, negative if the first string is less than the second string, and positive if the first string is greater than the second string. "Less than" and "greater than" pertain to alphabetical order in this context, not length. This could be useful in implementing binary search, for example.

- Let's update our program to use `strcmp`:

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

  int
  main(void)
  {
      printf(``Say something: '');
      string s1 = GetString();

      printf(``Say something again: '');
      string s2 = GetString();

      if (strcmp(s1, s2) == 0)
          printf(``You typed the same thing!\n'');
      else
          printf(``You typed different things!\n'');
  }
  ```

  Notice that we need to include the `string.h` library (as indicated by the documentation page) in order for this program to compile. When we do, we see that typing "hello" at both prompts gives us "You typed the same thing!"

- Woohoo, it's working! Now we're gonna be annoying and say that you're not allowed to use `strcmp`. Rather, you have to figure out how to compare two strings yourself. Well, actually, this shouldn't be too hard since we need only iterate through both strings, comparing them one character at a time, returning early as soon as two characters are different:

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

  int
  main(void)
  {
      printf(``Say something: '');
      string s1 = GetString();

      printf(``Say something again: '');
      string s2 = GetString();

      for (int i = 0; i < strlen(s1); i++)
  ```

```
    {
        if (s1[i] != s2[i])
        {
            printf(‘‘You typed different things!\n’’);
            return 1;
        }
    }

    printf(‘‘You typed the same thing!\n’’);
}
```

Is this program correct? Not quite. If we type "cat" as our first string and "caterpillar" as our second string, the program tells us that we typed the same thing. This is because we iterated only up to the length of the first string (3), and since the first 3 letters of both strings are the same, we never found a character that differs between them.

- If our first string is much longer than our second string, we're going to iterate off the end of the second string, which is also a concern. To fix this, we could simply compare the lengths of our two strings and return early if they're different:

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

int
main(void)
{
    printf(‘‘Say something: ’’);
    string s1 = GetString();

    printf(‘‘Say something again: ’’);
    string s2 = GetString();

    if (strlen(s1) != strlen(s2))
    {
        printf(‘‘Obviously not the same!\n’’);
        return 1;
    }

    for (int i = 0; i < strlen(s1); i++)
    {
        if (s1[i] != s2[i])
        {
            printf(‘‘You typed different things!\n’’);
            return 1;
        }
    }

    printf(``You typed the same thing!\n'');
}
```

Clearly, if the two strings differ in their length, then they are not the same. With this construction, we know that the strings are of equal length by the time we reach our for loop.

- One optimization we can make is to move the call to `strlen` into the initializations of the for loop like so:

```
for (int i = 0, n = strlen(s1); i < n; i++)
{
    if (s1[i] != s2[i])
    {
        printf(``You typed different things!\n'');
        return 1;
    }
}
```

Previously, `strlen` was called on each iteration of the loop. Here, it will only be called once. Actually, technically, we already called it once before when we compared it to the length of `s2`, but this isn't as expensive of a mistake.

6 More Fun with Pointers (46:00–70:00)

6.1 `GetString`

- The declaration of `GetString` in `cs50.h`, which lives in `/usr/include/` on the Appliance, is as follows:

```
/*
 * 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);
```

Note that although the user terminates his input by hitting Enter, the CS50 Library assumes that that newline character should be excluded from the string. Thus, if the user only hits Enter, then `GetString` will store the empty string, which takes exactly 1 byte to store (the null terminator).

- What exactly is `NULL`? `NULL` is actually a pointer which stores the memory address 0x0. Because this memory address is **never** accessible by a program, it exists as a special sentinel value. From now on, when we want to check if a pointer has a legitimate value, we will check that it is not equal to `NULL`.

6.2 `copy1.c`

- Let's try writing a program that will copy a string:

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

int
main(void)
{
    printf(``Say something: '');
    char *s1 = GetString();
    if (s1 == NULL)
    {
        printf(``An error occurred.\n'');
        return 1;
    }

    char *s2 = s1;

    printf(``s1 is %s\n'', s1);
    printf(``s2 is %s\n'', s2);

}
```

Now that we know that `GetString` can return `NULL` when something went wrong,—for example, if the user gave Ctrl + D as input or if he entered a string that was so long it couldn't be stored in RAM—we must check its return value before continuing on. Although it might seem harmless, failing to do this check can result in very worrisome bugs and security vulnerabilities.

- When we compile and run this program, typing "hello" at the prompt, we get the following output:

```
s1 is hello
s2 is hello
```

Seems to work.[6] However, what if we add a snippet of code in to capitalize only one of the copies of the string?

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

int
main(void)
{
    printf(‘‘Say something: ’’);
    char *s1 = GetString();
    if (s1 == NULL)
    {
        printf(‘‘An error occurred.\n’’);
        return 1;
    }

    char *s2 = s1;

    // capitalize just one of the strings
    s2[0] = toupper(s2[0]);

    printf(‘‘s1 is %s\n’’, s1);
    printf(‘‘s2 is %s\n’’, s2);

}
```

`toupper` is a function that converts a lowercase letter to uppercase. Now when we compile and run this program, typing “hello” at the prompt, we get the following output:

```
s1 is Hello
s2 is Hello
```

Well, the capitalization worked, but unfortunately it worked on both copies of the string. What’s going on here is that `s1` and `s2` both point to the same chunk of memory. If the chunk of memory storing the string “hello” is at memory address 0x1234, then both `s1` and `s2` have the value 0x1234. When we capitalize the first character of `s2`, we’re also capitalizing the first character of `s1`, since they are one and the same. Incidentally, the actual memory address of the string is unimportant, so when we talk about pointers, we generally don’t visualize them as memory addresses like 0x1234 but rather as arrows that literally point to a chunk of memory.

[6]You know what comes next. Groan.

- To fix this problem, we'll need to allocate a separate chunk of memory that is equal in size to the chunk of memory that stores our string. Then, we'll insert into this new chunk of memory each of the characters of the string we want to copy. To allocate a chunk of memory, we'll call `malloc`, the same function that `GetString` uses.

6.3 `copy2.c`

- To get our feet wet using `malloc`, let's start by hardcoding a call to it with the input 6, i.e. enough to store the string "hello":

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

int
main(void)
{
    printf(``Say something: '');
    char *s1 = GetString();
    if (s1 == NULL)
    {
        printf(``An error occurred.\n'');
        return 1;
    }

    char *s2 = malloc(6);

    // capitalize just one of the strings
    s2[0] = toupper(s2[0]);

    printf(``s1 is %s\n'', s1);
    printf(``s2 is %s\n'', s2);

}
```

  6 bytes is enough to store each of the characters of "hello" as well as the null terminator. But if we want to be more flexible with the strings that this program can copy, we need only generalize what 6 represents: it's actually just the length of the user's string plus 1 additional character to store the null terminator. So we can improve our design as follows:

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

```
int
main(void)
{
    printf(``Say something: '');
    char *s1 = GetString();
    if (s1 == NULL)
    {
        printf(``An error occurred.\n'');
        return 1;
    }

    char *s2 = malloc(strlen(s1) + 1);

    // capitalize just one of the strings
    s2[0] = toupper(s2[0]);

    printf(``s1 is %s\n'', s1);
    printf(``s2 is %s\n'', s2);

}
```

The source code on the course website also multiplies this amount `strlen(s1) + 1` by `sizeof(char)` which is not strictly necessary, but is technically more correct. Without this multiplication, our program assumes that the `char` type takes exactly one byte, but that might not always be true. With this multiplication, our program will dynamically ask for the number of bytes that a `char` type requires using a call to `sizeof`.

- We have space to store our copy, now, so let's fill it up:

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

int
main(void)
{
    printf(``Say something: '');
    char *s1 = GetString();
    if (s1 == NULL)
    {
        printf(``An error occurred.\n'');
        return 1;
    }
```

```
        char *s2 = malloc((strlen(s1) + 1) * sizeof(char));
        if (s2 == NULL)
        {
            printf(``An error occurred (e.g., out of memory).\n'');
            return 1;
        }
        for (int i = 0, n = strlen(s1) + 1; i < n; i++)
        {
            s2[i] = s1[i];
        }

        // capitalize just one of the strings
        s2[0] = toupper(s2[0]);

        printf(``s1 is %s\n'', s1);
        printf(``s2 is %s\n'', s2);

    }
```

Note that we iterate up through one greater than the length of `s1` so that we copy the null terminator into `s2`. Also, we check that `malloc` didn't return `NULL` when we asked for memory for `s2`. FInally, we've included `string.h` since we're again using `strlen`.

- When we compile and run this program, typing "hello" at the prompt, we get the following output:

```
s1 is hello
s2 is Hello
```

  It finally worked!

6.4 Binky

- Thus far, we've only looked at pointers to chunks of memory that store characters. But there are, of course, pointers to other data types. Let's look at the following toy program which demonstrates how to store an integer value:

```
int
main(void)
{
    int *x;
    int *y;

    x = malloc(sizeof(int));
    *x = 42;
}
```

The `*` notation, in addition to being used in the declaration of pointers, is used in the *dereferencing* of pointers. That is, when you want to ask for what a pointer is pointing to, you put a `*` in front of it. So `*x = 42` says to store the value of 42 in the memory that `x` is pointing to.

- Let's use this notation to initialize `y` as well:

```
int
main(void)
{
    int *x;
    int *y;

    x = malloc(sizeof(int));
    *x = 42;

    *y = 13;

    y = x;

    *y = 13;
}
```

Anything wrong here?[7] Oops, we haven't asked for any memory for `y`, so we have no idea where it's pointing. When we ask to assign the number 13 to the memory it's pointing to, we're asking for trouble, probably in the form of a segmentation fault.

- Once we assign to `y` the value of `x`, both are pointing to the chunk of memory that we allocated for `x`. Thus when we dereference `y` and store the value 13, we're overwriting the value 42 that we originally stored there.

- This toy program is the subject of the long-anticipated Pointer Fun With Binky!

[7]The answer to that question is always yes.