Computer Science 50
Fall 2010
Scribe Notes

Week 4 Wednesday: September 29, 2010
Andrew Sellergren

Contents

Computer Science 50 Week 4 Wednesday: September 29, 2010
Fall 2010 Andrew Sellergren
Scribe Notes

1 Announcements and Demos (0:00–8:00)

- This is CS50.
- 0 new handouts.
- Today when David witnessed a woman struggling with the Charlie Card machines in the Harvard T station today and offered to help, he was reminded of the importance of good design in computer science. It will be especially important if you decide to create a website for your final project as you'll need to consider all aspects of the user interface. The user interface on the Charlie Card machines, unfortunately, is optimized for all possible use cases rather than the most common use cases. You are first asked if you have a Charlie Card or if you want to buy a ticket. Once you press the button to buy a ticket, you have to choose what type of ticket you want to buy. Then, because there's no option for one-way or round-trip, you have to manually enter the value of the ticket you want to buy. Only after navigating through a half dozen more screens are you finally able to print out a ticket.
- We're not perfect by any means. After soliciting feedback on the Harvard-Courses user interface, we'll be implementing some changes that hopefully make it easier to use.

2 From Last Time (8:00–15:00)

- A few lectures ago, we teased apart `buggy3.c` and determined that a problem of variable scope was preventing our `swap` function from working properly. We'll dive more into this in a moment.
- We also mentioned the importance of not overstepping the bounds of an array. Doing so can lead to segmentation faults or leave your program vulnerable to a buffer overrun exploit. This exploit was, in fact, the basis for the original iPhone jailbreak.
- Conceptually, we think of a computer's memory as a stack. At the bottom of the stack is the frame that stores `main`'s variables `x` and `y`. When we call `swap`, another frame gets placed on top of `main`'s that stores `swap`'s local variables `a` and `b`. Herein lies the problem. When we pass `x` and `y` as arguments to `swap`, we're not passing `x` and `y` themselves but rather copies of `x` and `y`. `swap` modifies these copies rather than the original variables, so the changes are lost when the function returns.

3 Pointers (15:00–73:00)

- Version 2 of the `swap` function fixes this problem:

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

It seems like all we needed to do was put an asterisk in front of `a` and `b` wherever they appear. Of course, the solution is a little more complicated than that, but at least syntactically, it's quite simple.

- Using this notation, we're actually passing *pointers* to `swap` rather than integers. Pointers are memory addresses. By passing pointers instead of integers, we're giving `swap` the ability to modify `main`'s variables themselves rather than copies of those variables.

3.1 swap.c

- `swap.c` makes use of this new `swap` function:

```
/**************************************************************************
 * swap.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Swaps two variables' values.
 *
 * Demonstrates passing by reference.
 **************************************************************************/

#include <stdio.h>


// function prototype
void swap(int *a, int *b);


int
main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %d\n", x);
    printf("y is %d\n", y);
```

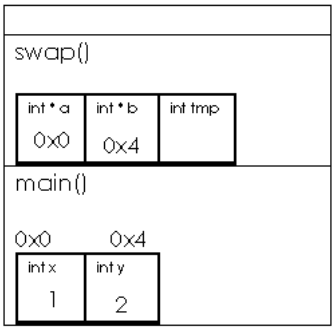
```
    printf("Swapping...\n");
    swap(&x, &y);
    printf("Swapped!\n");
    printf("x is %d\n", x);
    printf("y is %d\n", y);
}


/*
 * Swap arguments' values.
 */

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

We had to change the declaration of `swap` to indicate that it takes as arguments two pointers to `int`'s rather than two `int`'s.

- Besides the asterisk notation, we also have a change to how we call `swap`. We pass not `x` and `y`, but `&x` and `&y`. The `&` is the "address-of" operator. `&x` and `&y` are the addresses in memory of `x` and `y`, respectively.
- Inside of the `swap` function, the `*` has a slightly different meaning. Writing `int *a` declares a variable `a` of type `int *`. Writing `*a` retrieves the value stored at memory address `a`. In the latter context, `*` is the dereferencing operator.
- Let's walk through the new `swap` line by line while looking at the stack:[1]

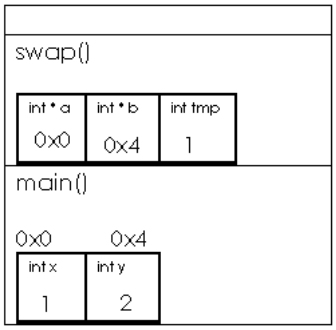
    1. `int tmp;`

[1]Note that in lecture, David uses the memory addresses 0x123 and 0x456, but we're using 0x0 and 0x4 here. The starting points of both sets are arbitrary—only the difference between them, namely 4 bytes, is significant.



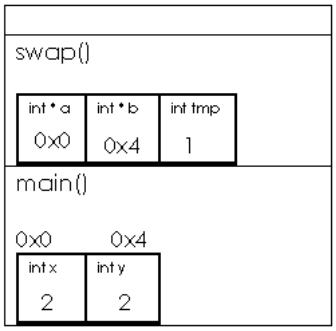
Here, again we're declaring `tmp` as an `int` local to the function `swap`. However, notice now that the variables `a` and `b` are not storing values but rather memory addresses, specifically those of `x` and `y`! These are pointers to `x` and `y`. When called upon, our function can access and change `x` and `y` directly.

2. `tmp = *a;`



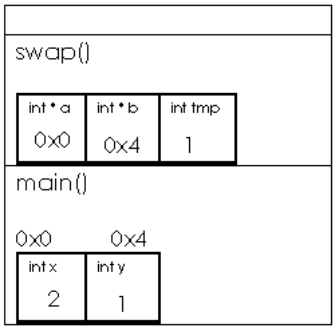
This whole line reads "assign to `tmp` whatever is stored in memory at location `a`." Once we've done that, `tmp` stores the value 1, as shown in the diagram.

3. `*a = *b;`



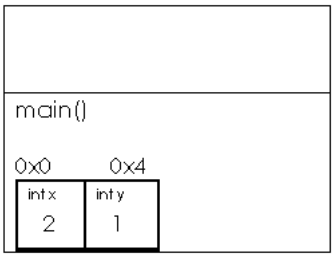
This line is a little harder to follow, but pay attention to the diagram. We're saying "assign to the memory at location `a` whatever is stored in memory at location `b`." This is the first part of the swap! We've actually modified the memory of `main`!

4. `*b = tmp;`



And the swap is complete! Note that we don't use a star in front of `tmp` in this line. That's because we don't care about where `tmp` is stored, we're only using it to temporarily hold a value. We don't care what happens to it after `swap` returns.

5. `[return;]`



Finally, `swap` returns (albeit not explicitly in the code since its type is `void`) and its stack frame gets popped off. Unlike in `buggy3.c`, however, the values of `x` and `y` have actually been swapped!

- Question: what would `*&x` return? This is equivalent to writing `x` since the `*` and `&` operators undo each other. In this context, it would return 1, then, before the swap has taken place.

3.2 `compare1.c`

- Our newly gained knowledge of pointers will allow us to figure out why `compare1.c` doesn't successfully compare two strings:

```
/**************************************************************************
 * compare1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Tries (and fails) to compare two strings.
 *
 * Demonstrates strings as pointers to arrays.
 **************************************************************************/

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


int
main(void)
{
    // get line of text
    printf("Say something: ");
    string s1 = GetString();

    // get another line of text
    printf("Say something: ");
    string s2 = GetString();

    // try (and fail) to compare strings
    if (s1 == s2)
        printf("You typed the same thing!\n");
    else
        printf("You typed different things!\n");
}
```

We're correctly using the comparison operator `==` instead of the assignment operator `=`. Writing `if (s1 = s2)` would evaluate to true whenever `s2` was non-zero. This is definitely not what we want.

- `compare1` prints out "You typed different things!" even when we provide it the same string twice. This is because `s1` and `s2` are actually pointers. The `string` data type is actually a `char *`. `s1` and `s2` store the addresses in memory of the first `char`'s in our two string inputs. Once we know the memory address of the first character, we can find the rest of the characters in the string because they are contiguous in memory. We just walk one byte at a time starting from that memory address until we find the `\0` character, the null terminator, which marks the end of the string.
- When we write `s1 == s2`, we're comparing the memory addresses of our

two string inputs, not the strings themselves. Because the memory addresses will never be the same (they *must* be stored at different locations), this expression will always return false.

3.3 copy1.c

- `copy1.c` takes the training wheels off for the first time and explicitly uses the `char *` data type to store a string:

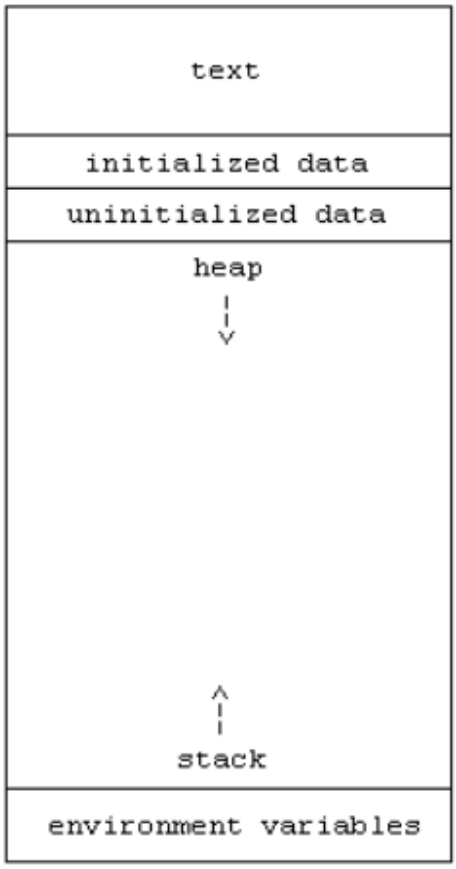
```
/**************************************************************************
 * copy1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Tries and fails to copy two strings.
 *
 * Demonstrates strings as pointers to arrays.
 **************************************************************************/

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


int
main(void)
{
    // 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);
}
```

We must check that `s1` isn't `NULL` because `GetString` will return `NULL` if there isn't enough memory to store the string the user gave as input. `NULL` actually refers to the memory address 0x00[2] which is owned by the operating system and which, if accessed by the program, will throw a fatal error.

- The line `char *s2 = s1` successfully assigns the memory address of the user's string to the `s2` variable. However, it doesn't actually make a copy of that string. `s2` points to the same string that `s1` does, so if we dereference `s2` and modify what's stored there, we'll lose our original version of the string.
- We do another sanity check using the `strlen` function to check that the user hasn't passed us an empty string. Once we're sure the string isn't empty, we capitalize the first letter by passing it to the `toupper` function. We access the first letter of the string using the bracket notation familiar to us from arrays.
- When we compile and run `copy1`, we see that both the original string and the copy are capitalized. When we converted the first letter to uppercase, it was the original string, not a copy, that we modified.
- In case it wasn't clear before, the syntax for declaring a pointer to a `char` is demonstrated above. We write `char *s2` to declare a pointer to a `char`—what we previously referred to as the `string` data type—named `s2`.
- Recall that our representation of a computer's memory included the stack, which grows upward, as well as the heap, which grows downward:

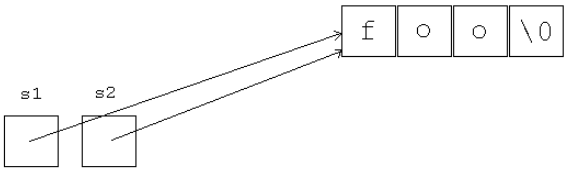
[2] We'll discuss the 0x notation shortly.



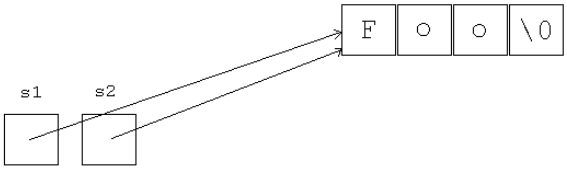
The text segment of memory stores the actual zeroes and ones that make up the program. When we declare a global variable, as we do to store the board for the Game of Fifteen, it is stored in the initialized data or uninitialized data segments.

- The heap is used for dynamic memory allocation. You the programmer won't always know how much RAM your program will need before it runs. When your program requests more memory at runtime, it will be allocated from the heap. Whenever you've called `GetString`, `GetInt`, etc. from the CS50 Library, you've been given memory from the heap.
- On most systems, pointers are actually 32-bit integers. That is, all of the memory addresses can be represented with 4 bytes. Some systems are 64-bit, which means that memory addresses are represented with 8 bytes.
- When we call `GetString` in `copy1` and provide it with "foo" as input, 4 bytes of memory are allocated on the heap, 3 for the actual letters of the string and 1 for the null terminator to mark the end of the string.

`GetString` then returns a pointer to the first character, the memory address of "f" in the heap. Let's call that memory address 71. The actual value, then, of `s1` is 71. However, since we don't really care what the actual value of `s1` is, it's more meaningful to represent `s1` as an arrow that literally points to the first character of our string. Because `s2` is assigned the same value that `s1` holds, it also points to the first character of our string. We can visualize this like so:



Once we've accessed the first letter of `s2` and converted it to uppercase, our memory looks like this:



- Keep in mind that `GetString` allocates memory for our string on the heap (using a function called `malloc`), not the stack. As a result, we don't run into problems with variable scope when `GetString`'s frame gets popped off the stack.

3.4 copy2.c

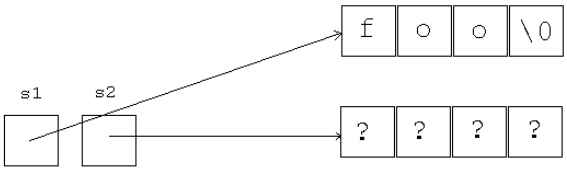
- `copy2.c` successfully copies a string and capitalizes the copy:

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

As before, we prompt the user for the string and then store the memory address of the first character of that string (which lives on the heap) in `s1`.

- 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 talking about a string "foo," then we're passing the value 4 to `malloc`.
- At this point, after we've requested 4 bytes of memory from the heap and assigned the memory address of the first byte to `s2` (as returned by `malloc`), our memory looks like this:



`s2` now points to a different chunk of memory than does `s1`. This new chunk of memory is uninitialized, so we'll put question marks to represent its contents.

- In case the user has given us a string that is too large to be stored in memory, we check that `s2` isn't `NULL`.
- To copy the actual string into the memory pointed to by `s2`, we iterate over the characters of the string. Our terminating condition is `i < strlen(s1)` which equates to `i < 3` in this case. Once the loop terminates, we need to manually place the null terminator at the end of the string.
- If we compile and run `copy2`, we see that it actually works!

3.5 Pointers to Other Data Types

- Let's write a short program that demonstrates the use pointers to integers (`int *`) rather than pointers to characters (`char *`):

```
int
main(void)
{
    int *x;
    int *y;
    x = malloc(sizeof(int));
    x = 42;
}
```

This assignment is wrong because `x` previously stored the memory address of an integer on the heap but then is overwritten to hold the number 42. We'll then lose track of that memory on the heap that we asked for, introducing a memory leak into our program. What we want is to store the number 42 in the space we've allocated on the heap. To do that we write this instead:

```
int
main(void)
{
    int *x;
    int *y;
    x = malloc(sizeof(int));
    *x = 42;
}
```

- What happens if we try to assign a value to `y` before we've allocated memory for it?

```
int
main(void)
{
    int *x;
    int *y;
    x = malloc(sizeof(int));
    *x = 42;
    *y = 13; // BAD
}
```

More than likely, this will cause a segmentation fault. When we declared `y`, we didn't initialize it to have any value. Writing `*y` attempts to access the contents of `y` as a memory address. But we have no way of knowing if that memory address is a valid one and even if it is, it's certainly not memory that belongs to us. If `y` happens to have the value 0, for example, we'll be trying to access the memory at address 0 which, as we've already seen, throws a fatal error.

- Assigning the value of `x` to `y` and then accessing the memory at `y`, however, is okay:

```
int
main(void)
{
    int *x;
    int *y;
    x = malloc(sizeof(int));
    *x = 42;
    y = x;
    *y = 13;
}
```

To visualize this program in a more fun way, let's turn to Binky![3]

[3]The professor behind this video, Nick Parlante, also works at Google. Yes, I met him, and yes, I asked him about Binky. He said Binky is doing well. If you'd like to meet him too, maybe you should think about applying to Google. Ask me how!