Week 5

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

Notes

- What you're reading now is a set of notes for reference and review of material. We want you more engaged in lecture than just taking notes! Find the complete collection at http://cs50.harvard.edu/lectures.

Pointer Review

- Remember that last time we discovered that a `string` was actually `char*`, which meant that every time we stored the return value of `GetString`, we were actually just storing an address to a character in memory:

```
string s = GetString();
  -----    -------------------------
  |0x1|    | D | a | v | e | n |\0 |
  -----    -------------------------
            0x1 0x2 0x3 0x4 0x5 0x6
```

  # `GetString` gets a chunk of memory from the operating system by calling `malloc`, putting letters from the user into that memory (followed by a `\0` character), and then returning a variable we call a **pointer** that is the address of where the first character (since it's a `char*`) is in memory.

    # And remember that we can move left to right through the `string` in linear time, until we reach `\0`, which will tell us we are at the end of the `string`.

- Last time we also looked at this code, which didn't actually compare the two `string`s:

```
string s = GetString();
string t = GetString();

if (s == t)
{
    printf("You typed the same thing!\n");
}
else
{
    printf("You typed different things!\n");
}
```

    # This program starts by asking the user for two `string`s, and then uses `s == t` to try to compare them, but fails to do so correctly.

- A volunteer from the audience, Janelle, helps us by first drawing a picture of what's happening:

```
s                0     1     2
-------        ------------------------------
| 0x1 |        |  g  |  a  |  b  |  e  | \0  |
-------        ------------------------------
               0x1   0x2   0x3   ...
t
-------        ------------------------------
|0x10 |        |  g  |  a  |  b  |  e  | \0  |
-------        ------------------------------
               0x10  0x11  0x12  ...
```

    # The boxes on the right are the `strings`s from `GetString` stored in memory, with their addresses below (the 0, 1, 2 at the top of the first `string` is just the index of each box, were you to treat the memory as an array).

    # The boxes on the left are the values of `s` and `t`, so when we try to compare them we see that they are different values, though what we really wanted to was to compare the two `string`s.

- Let’s look at a fix:

```c
char* s = GetString();
char* t = GetString();

if (s != NULL && t != NULL)
{
    if (strcmp(s, t) == 0)
    {
        printf("You typed the same thing!\n");
    }
    else
    {
        printf("You typed different things!\n");
    }
}
```

# First, we checked that both `s` and `t` are actually addresses and not `NULL` (like if the `string` a user typed is too long).

  # And if we were to try to do something with `NULL`, like passing it in to a function, most of the time the function or program will crash, so we should error-check. This is a good habit to get into, any time we use a variable that could be `NULL`.

# Then we use `strcmp` that stands for "string comparison" that compares the `string`s rather than the addresses, by going down both of them one character at at time.

# We made up all of these addresses and we just learned that the addresses are stored in `s` and `t`, so we can simplify this diagram by using just arrows. If `s` is a pointer, we can make it an arrow that literally points to what it refers to. The actual addresses don’t really matter, so we can cross them out, and just know that *some* address is there:

```
s
-------         ------------------------------
| ~~0x1~~ | -----> |  g  |  a  |  b  |  e  | \0  |
-------         ------------------------------
                ~~0x1~~   ~~0x2~~   ~~0x3~~   ...
t
-------         ------------------------------
|~~0x10~~ | -----> |  g  |  a  |  b  |  e  | \0  |
   -------         ------------------------------
                   ~~0x10~~  ~~0x11~~  ~~0x12~~  ...
```

- We also looked at this last time:

```
string s = GetString();
...
string t = s;
if (strlen(t) > 0)
{
    t[0] = toupper(t[0]);
}
```

  # We wanted to capitalize just the first letter in a `string t`, but we ended up changing `s` as well.

    # As an aside, a computer that has a **32-bit architecture** uses 32-bits to address memory, which is why older computers can only have a maximum of 4GB of RAM. They can only "count" as high as about 4 billion bytes, since they're using 32 bits, or 4 bytes, as addresses.

- So when we said `string t = s`, we were really doing this:

```
s
----         ------------------------------
|  | -----> |  g  |  a  |  b  |  e  | \0  |
----         ------------------------------
            ^
t           |
----        |
|  | ------/
----
```

- So now if we change `t[0]`, we're also changing `s[0]`. And remember that even though `t` and `s` are pointers, we can still use the array notation of `[0]` or `[1]` or `[2]` to jump to any element in the array.

- Let's fix our code:

```
char* s = GetString();
...
char* t = malloc((strlen(s) + 1) * sizeof(char)); ❶
...
for (int i = 0, n = strlen(s); i <= n; i++)
{
    t[i] = s[i];
}
...
if (strlen(t) > 0)
{
    t[0] = toupper(t[0]);
}
```

# At **(1)**, we are allocating a certain amount of memory — as many bytes as there are characters in `s`, with one more for the terminating character, and just in case `sizeof(char)` isn't `1`, we multiply it by whatever that value is, even though it should most likely be `1`.

# Then we copy `s` to `t`, and now that last line only changes `t`.

# And remember that when we first allocate memory for `t`, there are garbage values inside that we need to overwrite:

```
s
----        -------------------------------
|  | -----> |  g  |  a  |  b  |  e  | \0  |
----        -------------------------------

t
----        -------------------------------
|  | -----> |  ?  |  ?  |  ?  |  ?  |  ?  |
----        -------------------------------
```

Swap Again

- Finally, recall the `swap` example:

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

# This function worked in swapping `a` and `b`, but it didn't work in being able to swap the original variables in `main`, which looks like this:

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

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

- So let's look at `swap.c`[1]:

[1] http://cdn.cs50.net/2014/fall/lectures/5/m/src5m/swap.c

```
#include <stdio.h>

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

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

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

/**
 * Swap arguments' values.
 */
void swap(int* a, int* b)
{
    int tmp = *a; ❷
    *a = *b; ❸
    *b = tmp; ❹
}
```

# Notice that we have brand new syntax on line 14, the `&` s. It’s one of the last pieces of syntax we have to learn (phew) and all it does is get the address of some variable. `&x` is the address of `x`, which is an `int`.

- Before, `x` and `y` were passed as copies, but now we can pass `swap` a "treasure map" that leads to `x` and `y` that changes the same variables in memory that `main` has.
- And if we look at the `swap` function, we see that `a` and `b` are no longer `int` s but `int*` s.
- Now when we call `swap` and give it two addresses, actual locations in memory, what does it do with them? At line 25, it goes to the location with the address stored in `a`

with `*a`, and stores that `int` value in `tmp`. At line 26, it goes to the location with the address stored in `b` and stores that in the location with the address stored in `a`. Finally, at line 27, we put the value of `tmp` into the location with the address stored in `b`.

- When `main` calls `swap`, `swap`'s stack frame receives the address of `x` and `y`, whereever they may be in memory.
- And now we see that our values are swapped successfully:

```
jharvard@appliance (~/Dropbox/src5m): ./swap
x is 1
y is 2
Swapping...
Swapped!
x is 2
y is 1
```

- We can even see this with our friend `gdb`:

```
jharvard@appliance (~/Dropbox/src5m): gdb ./swap
Reading symbols from ./swap...done.
(gdb) break main
Breakpoint 1 at 0x80484ac: file swap.c, line 19.
(gdb) run
Starting program: /home/jharvard/Dropbox/src5m/swap

Breakpoint 1, main () at swap.c:19
19        int x = 1;
(gdb) print x
$1 = 0
```

- Remember that in this case `x` could be any garbage value (though we happened to get 0), since line 19 hasn't yet run.
- We confirm that `x` and `y` are indeed `1` and `2`:

```
(gdb) next
20        int y = 2;
(gdb) print x
$2 = 1
(gdb) next
22        printf("x is %i\n", x);
(gdb) print y
```

```
$3 = 2
(gdb) next
x is 1
23      printf("y is %i\n", y);
(gdb) next
y is 2
24      printf("Swapping...\n");
(gdb) print x
$4 = 1
(gdb) print y
$5 = 2
```

- But what if we said this? Notice that we're getting large hexadecimal values, `0xbfff…`, which are the actual memory addresses of `x` and `y` in memory:

```
(gdb) print &x
$6 = (int *) 0xbffff0c4
(gdb) print &y
$7 = (int *) 0xbffff0c0
```

  - Notice that the difference between the two is 4, the size of an `int`, so they must be lined up in memory.

- Stepping into the `swap` function, we see that `a` and `b` are indeeed those same addresses, which, when followed, lead to `1` and `2`, or `x` and `y` from `main`, as expected:

```
(gdb) next
Swapping...
25      swap(&x, &y);
(gdb) step
swap (a=0xbffff0f4, b=0xbffff0f0) at swap.c:36
36      int tmp = *a;
(gdb) print a
$8 = (int *) 0xbffff0c4
(gdb) print b
$9 = (int *) 0xbffff0c0
(gdb) print *a
$10 = 1
(gdb) print *b
$11 = 2
```

- You'll see more in problem sets how these are useful and get more comfortable with them!

Malloc

- All this time, when we've been using the CS50 Library to `GetString`, it's also been calling `malloc`. But let's start by looking at `scanf-0.c`[2]:

```
#include <stdio.h>

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

  # It declares a variable `x` and `printf`s a message asking for a number, but what about line 7? `scanf` "scans" the user's input in from the keyboard, and `%i` means we expect an integer. Then we pass in `&x`, because we want the input to be saved at that location. Otherwise, only a copy of `x` would be changed by `scanf`.

- So we can run it:

```
jharvard@appliance (~/Dropbox/src5m): ./scanf-0
Number please: 50
Thanks for the 50!
jharvard@appliance (~/Dropbox/src5m): ./scanf-0
Number please: -1
Thanks for the -1!
jharvard@appliance (~/Dropbox/src5m): ./scanf-0
Number please: 1.5
Thanks for the 1!
```

  # Hm, seems to work until we pass in `1.5`. Well, it expects an `int`, so it only keeps the integer part of the number.

[2] http://cdn.cs50.net/2014/fall/lectures/5/m/src5m/scanf-0.c

- Let's look at `scanf-1.c`[3]:

```
#include <stdio.h>

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

  # But we realize this is a bad example. We create a `char* buffer` and expect `scanf` to take some string and put it in memory at whatever address `buffer` points to. But `buffer` is some garbage value, so `scanf` will put the input at an address that could be anywhere in memory. And if the memory doesn't belong to us, then we'll probably cause a segmentation fault and crash our program.

- What if we did something like `scanf-2.c`[4]?

```
#include <stdio.h>

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

  # This is better, since we're declaring an array of characters, which sets aside memory, and works perfectly, until we type in 16, 17, or more characters. Then that string will partly end up in `buffer`, but overwrite whatever is beyond the boundary of that array, since we only asked for 16 bytes.

- If we in `cs50.h`[5], we see this:

[3] http://cdn.cs50.net/2014/fall/lectures/5/m/src5m/scanf-1.c
[4] http://cdn.cs50.net/2014/fall/lectures/5/m/src5m/scanf-2.c
[5] http://cdn.cs50.net/2014/fall/lectures/5/m/src5m/cs50.h

```
...
/**
 * Our own data type for string variables.
 */
typedef char* string;
...
```

# And that just makes `string` mean the same as `char*`.

- If we continue, we see code like this:

```
...
/**
 * Reads a line of text from standard input and returns the equivalent
 * char; if text does not represent a char, user is prompted to retry.
 * Leading and trailing whitespace is ignored.  If line can't be read,
 * returns CHAR_MAX.
 */
char GetChar(void);

/**
 * Reads a line of text from standard input and returns the equivalent
 * double as precisely as possible; if text does not represent a
 * double, user is prompted to retry.  Leading and trailing whitespace
 * is ignored.  For simplicity, overflow and underflow are not detected.
 * If line can't be read, returns DBL_MAX.
 */
double GetDouble(void);

/**
 * Reads a line of text from standard input and returns the equivalent
 * float as precisely as possible; if text does not represent a float,
 * user is prompted to retry.  Leading and trailing whitespace is ignored.
 * For simplicity, overflow and underflow are not detected.  If line can't
 * be read, returns FLT_MAX.
 */
float GetFloat(void);

/**
 * 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(void);
```

but no actual functions. Remember that we’re in the header file, which only has a `typedef` and prototypes.

- So we need to find the actual implementation of functions in `cs50.c`[6], in particular `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(void)
{
    // 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, " %i %c", &n, &c) == 1)
        {
            free(line);
            return n;
        }
        else
        {
            free(line);
            printf("Retry: ");
        }
    }
}
...
```

[6] http://cdn.cs50.net/2014/fall/lectures/5/m/src5m/cs50.c

# First it tells us what the range of values `GetInt` can return, and that `If line can’t be read, returns INT_MAX`. So if something went wrong, it wouldn’t return `NULL` but a special value called `INT_MAX`. We can’t return `NULL` since it’s technically pointer and we have to return an `int`, so we return `INT_MAX`, the maximum value that can be stored in an `int`, as an error code. (We haven’t been checking to make things simple, though for maximum safety we should!)

# Then we see an infinite loop with `while (true)`, that actually starts by calling `GetString`, checking that it’s not `NULL`.

# Then we call `sscanf`, which takes a `string` in memory and does any of a number of things with it. In this case, we take `line`, our `string` from the user, and looks for `" %i %c"`, which means an int, and maybe other characters. `sscanf` returns the number of variables it was able to find, so in this case we want that to be `1`, or just an `int`, rather than an `int` followed by other characters. If the user typed in `123 x`, then `sscanf` would return `2`.

# Realize that we do a lot of error-checking, so any possible input typed in will have a case in our code that will handle it (and makes sure that `cs50.h` isn’t the source of a bug in your program!).

Binky

- Let’s return to our friend from last time, Binky. Remember we had code that looked like this:

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

    x = malloc(sizeof(int));

    *x = 42;

    *y = 13;

    y = x;

    *y = 13;
}
```

# This code does nothing useful, but walking through it and seeing what's going on helps us understand pointers.

- Let's retell the story super quickly.

# The first two lines declare two pointers to integers, `int* x` and `int* y`, with some garbage values in them:

```
x -----
  | ? |
  -----

y -----
  | ? |
  -----
```

# Then we allocate enough memory to store an `int`, and stores the address of that memory in `x` with `x = malloc(sizeof(int));`.

# With `*x = 42`, we go to the address `x` and stores `42` there.

```
x -----         ------
  | --|------>  | 42 |
  -----         ------

y -----
```

```
  | ? |
  -----
```

# But then we tried to go to `y` with `*y = 13`, but since there's still a garbage value in `y`, Binky lost his head.

# We could fix it with this line, `y = x`:

```
x -----          ------
  | --|------>  | 42 |
  -----          ------
                 ^
y -----          |
  | --|--------/
  -----
```

# And now we can say `*y = 13`:

```
x -----          ------
  | --|------>  | 13 |
  -----          ------
                 ^
y -----          |
  | --|--------/
  -----
```

- Let's open `memory.c`[7] as another example:

[7] http://cdn.cs50.net/2014/fall/lectures/5/m/src5m/memory.c

```
#include <stdlib.h>

void f(void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;
}

int main(void)
{
    f();
    return 0;
}
```

  # All `main` does is call a function `f` and `return 0`. So let's look at `f`, which only has two lines. The left side of the first line is just declaring a pointer to an `int`, and the right is allocating 40 bytes (4 bytes per `int`, at least on the appliance). And remember that `malloc` gives you one contiguous chunk of memory of that size. The second line, `x[10] = 0`, is setting the *11th* element of the array to `0`, but that element doesn't exist since there are only 10 elements, indexed `0` through `9`. There may be a garbage value in `x[10]`, but that memory doesn't belong to us, and might cause problems or crash our program.

- We can run the program with some values like `x[10]` or even `x[1000]` without any problems, if we get lucky, though we may have overwritten some other variable in our program's memory.
- If we went really far out, like trying to do this:

```
...
    int* x = malloc(10 * sizeof(int));
    x[100000] = 0;
...
```

  # then we get a `Segmentation fault (core dumped)` since we *really* shouldn't be touching that piece of memory.

Valgrind

- Let’s introduce you to everyone’s best friend, `valgrind`. This is a tool that helps us notice these cases where we touch memory that doesn’t belong to us, and also where we have **memory leaks**. So far, when we’ve asked the operating system for memory with `malloc`, we haven’t returned it (which we will soon with `free`) when we finished, and so those chunks of memory are "leaked."

  # If you’ve left your computer running for some time, opening lots of programs, and it gets slower, then the problem could be with certain programs asking for memory, and forgetting about it, taking it away from other programs and slowing everything else.

- Let’s run `valgrind ./memory`:

```
jharvard@appliance (~/Dropbox/src5m): valgrind ./memory
==22478== Memcheck, a memory error detector
==22478== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==22478== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for
 copyright info
==22478== Command: ./memory
==22478==
==22478== Invalid write of size 4
==22478==    at 0x80484C0: f (memory.c:21)
==22478==    by 0x80484E1: main (memory.c:26)
==22478==  Address 0x429eaa8 is not stack'd, malloc'd or (recently) free'd
==22478==
==22478==
==22478== HEAP SUMMARY:
==22478==     in use at exit: 40 bytes in 1 blocks
==22478==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==22478==
==22478== LEAK SUMMARY:
==22478==    definitely lost: 40 bytes in 1 blocks
==22478==    indirectly lost: 0 bytes in 0 blocks
==22478==      possibly lost: 0 bytes in 0 blocks
==22478==    still reachable: 0 bytes in 0 blocks
==22478==         suppressed: 0 bytes in 0 blocks
==22478== Rerun with --leak-check=full to see details of leaked memory
==22478==
==22478== For counts of detected and suppressed errors, rerun with: -v
```

```
==22478== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

  - We see that it tells us about the `Invalid write of size 4` where we tried to write to `x[100000]`, and also the `40 bytes in 1 blocks` that we haven't returned, or `free`d.

- A final example demonstrates **buffer overflow**, which bad guys can take advantage of to compromise accounts or machines. Like the name suggests, a buffer overflow is filling a buffer, or a chunk of memory, with too much information. (When watching videos on YouTube or the like, "buffer" is used similarly where the video player is downloading lots of bytes from the Internet and holding it in a buffer, so it can play the video without pausing and downloading again constantly.)

  - Very bad things can happen. Let's look at this example:

```
#include <string.h>

void  f(char* bar)
{
    char c[12];
    strncpy(c, bar, strlen(bar));
}

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

    - `main` takes an argument, passing it to `f`, which has a `char` array called `c` of size `12`, and uses this new function called `strncpy`. With this simple line of code, we've made our entire computer vulnerable, since someone can run it with certain arguments that represent executable code, that can do things like delete files, get a command-line prompt for more access to your computer, and more.

- We'll learn more about these concepts, and this diagram, next time:

```
----------------------------
|                          |
|           text           |
|                          |
----------------------------
```

```
|      initialized data     |
-----------------------------
|     uninitialized data    |
-----------------------------
|           heap            |
|             |             |
|             |             |
|             v             |
|                           |
|                           |
|                           |
|             ^             |
|             |             |
|             |             |
|           stack           |
-----------------------------
|   environment variables   |
-----------------------------
```

- We end on this xkcd[8] until next time.

[8] http://xkcd.com/138