Week 4, continued

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

Files, Headers, and Hex

- One of the topics for today is digital forensics, recovering information, and Problem Set 4 will be in this domain.
- David used to work in the district attorney's office, attempting to find evidence from hard drive and memory cards that the police brought in.
  - The portrayal of this process in TV and movies might look like this[1], where characters, in clips from various shows and films, say phrases like "zoom" and "enhance," that magically cause images to reveal details previously unseen.
- But when we really try to "enhance" images, like that of Daven, we eventually see the pixels that compose the image, because there are only a finite of bits in the image. (The bad guy in the reflection in his eye will only be 6 pixels, no matter how far we try to zoom!)
- We'll explore this in Problem Set 4 with **file I/O**, where I/O just means input/output. None of the programs we've worked with so far have been saving to disk by creating or changing files.
- An example of a file is a JPEG, which is simply an image file.

[1] http://youtu.be/LhF_56SxrGk

- What's interesting is that files can typically be identified by certain patterns of bits. Different files, like a JPEG or a PNG (image file) or a GIF (image file) or a Word document or a Excel spreadsheet, will have different patterns of bits, and those patterns are usually at the top of the file, so when a computer opens it, it can recognize, say, a JPEG as an image, and display it to the user as a graphic. Or, it might look like a Word doc, so let's show it to the user as an essay.
- For instance, the first three bytes of a JPEG are:

```
255  216  255
```

- Next week we'll be poking more deeply into files to see what's always been there.
- But first, realize that "what's there" generally isn't written in decimal numbers like above.
- Computer scientists actually tend to express numbers in hexadecimal, as opposed to decimal or binary.
- Recall that decimal uses 10 digits, 0-9, while binary is composed of 2 digits, 0 and 1.
- **Hexadecimal** means that we will have 16 such digits, 0-9 and a, b, c, d, e, f.

  # "a" is 10, "b" is 11, and so on.
- How can this be useful? Well let's write out the bits that represent these numbers:

```
     255      216      255
11111111  11011000  11111111
```

- This is interesting because a byte has 8 bits, and if we break each byte into two chunks of 4 bits, each set of 4 bits will correspond to exactly one hexadecimal digit:

```
      255       216       255
1111 1111  1101 1000  1111 1111
   f    f     d 8     f    f
```

- To make this more readable, we remove the whitespace and add `0x`, just to signify that the characters in the last row are in hexadecimal:

```
      255       216       255
1111 1111  1101 1000  1111 1111
   f    f     d 8     f    f
     0xff     0xd8    0xff
```

- Note that we can also convert two hexadecimal digits to 8 bits in binary, or one byte, making it especially useful for representing binary data.

- Another image file format is a bitmap file, **BMP**. One example of an image in that format is `bliss.bmp`, a very familiar rolling green hill set against a blue cloudy sky (the default Windows XP wallpaper on millions of PCs).

  # A bitmap is just a pattern of pixels, or dots, a "map of bits," if you will.

- What's interesting, though, is that its beginnings are more than just a few bytes. Its **header** has a whole bunch of numbers, bytes, with their orders and values determined years ago by its author, Microsoft. Indeed, Microsoft has named the types of those values things like `WORD` and `DWORD` and `LONG`, but those are simply data types like `int`, different names for the same thing.
- So when someone clicks on a BMP file, the image is only shown because the operating system (or image-viewing program, really) noticed all of these bits at the beginning of the file and recognized that it was a BMP. More on this later.

Structs

- Let's look at a simpler file first:

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

#include "structs.h"

// number of students
#define STUDENTS 3

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

- Let's say we want to start creating a database of every student, and start by saving the name of a student and their house.
- We see that there are `3` such `STUDENTS`, so we can probably use something like a `for` loop to `GetString` a name and house for each of them. And we can start by instinctively:

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

#include "structs.h"

// number of students
#define STUDENTS 3

int main(void)
{
    string names[STUDENTS];
    string houses[STUDENTS];

    for (int i = 0; i < STUDENTS; i++)
    {
        names[i] = GetString();
        houses[i] = GetString();
    }

    // TODO later...

}
```

# So this is correct, as we create arrays to store the `names` and `houses`, and iterate through for the number of students, even if it's not very user-friendly.

- But this is not very good design. What if they had an ID number or email or Twitter handle, or more details to associate? To add it we'd have to do something like this:

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

#include "structs.h"

// number of students
#define STUDENTS 3

int main(void)
{
    string names[STUDENTS];
    string houses[STUDENTS];
    int ids[STUDENTS];
    string twitters[STUDENTS];
    ...
}
```

 - But soon, if we keep adding, we'll have something pretty big and unwieldy.

- We can use a higher-level data structure to hold something of a type `student`, and we see an example of this in `structs.h`[2]:

```
#include <cs50.h>

// structure representing a student
typedef struct
{
    string name;
    string house;
}
student;
```

 - In fact, we've already been using `structs` in Problem Set 3, since there's no such thing as a `GRect` or `GOval` in C. The SPL has those data types, implemented with the approach shown above.

- The keywords **typedef** and **struct** on line 4 just mean define a type — a structure — that is a container for multiple things, and inside that structure will be a `string` called

[2] http://cdn.cs50.net/2014/fall/lectures/4/w/src4w/structs.h

`name` and a `string` called `house`, and the entire structure will be called `student` for convenience.

- `student` is now a data type just like `int` and `string` and `GRect` and others.
- Now we can do something like this, in `structs-0.c`[3]:

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

#include "structs.h"

// number of students
#define STUDENTS 3

int main(void)
{
    // declare students
    student students[STUDENTS];
...
```

  # Note that we have one `array` named `students`, with each element of the type `student`. There are `STUDENTS` (which we've defined in line 8 to be `3`) elements in the `students` array.

- How do we access `name` and `house` and other fields, or items, in a `struct`? Well we scroll down in `structs-0.c`:

[3] http://cdn.cs50.net/2014/fall/lectures/4/w/src4w/structs-0.c

```
...
int main(void)
{
    // declare students
    student students[STUDENTS];

    // populate students with user's input
    for (int i = 0; i < STUDENTS; i++)
    {
        printf("Student's name: ");
        students[i].name = GetString();

        printf("Student's house: ");
        students[i].house = GetString();
    }
...
```

- We index into the array in line 11, and use a new syntax of `.name` to get the field called `name`.
- We'll return to `struct`s soon!
  - As an aside, the scene in the Windows XP wallpaper[4] is now yellow and gloomy - or it was when someone went back to get another photo[5]!

Quick Reminder

- Speaking of images, here's another one of Daven at Fire and Ice, a reminder that CS50 Lunch is Friday at 1:15pm as usual, RSVP at http://cs50.harvard.edu/rsvp.

GDB

- So where did we leave off on Monday? We introduced this problem of swapping two variables `a` and `b` with a temporary variable called `tmp`:

[4] http://en.wikipedia.org/wiki/Bliss_(image)
[5] http://en.wikipedia.org/wiki/Bliss_(image)#mediaviewer/File:Bliss_(location).jpg

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

# But remember that the problem is that it only swaps the variables locally, in the function's own memory (recall the trays from Annenberg that represent each function's slice of memory, and how the `swap` function doesn't have access to the variables in `main`, but rather copies).

- Let's open our friend `noswap.c`[6]:

[6] http://cdn.cs50.net/2014/fall/lectures/4/m/src4m/noswap.c

```
/**
 * noswap.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Should swap two variables' values, but doesn't!  How come?
 */

#include <stdio.h>

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

/**
 * Fails to swap arguments' values.
 */
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

- In lines 16 and 17 we initialized `x` and `y`, had `printfs`, and then called `swap` on line 22. `swap`, on line 31, might look correct on first glance, but isn't.
- Let's warm up by investigating with our new friend `gdb`:

```
jharvard@appliance (~/Dropbox/src4w): gdb ./noswap
Reading symbols from ./noswap...done.
```

- Now let's just run it:

```
(gdb) run
Starting program: /home/jharvard/noswap
x is 1
y is 2
Swapping...
Swapped!
x is 1
y is 2
[Inferior 1 (process 4922) exited normally]
```

- That wasn't quite so useful, since it just ran the program. Let's pause execution with `break`, and we can specify a line like `break 10`, but let's just `break` at the `main` function:

```
(gdb) break main
Breakpoint 1 at 0x80484ac: file noswap.c, line 16.
```

  # Indeed, if we glance back at our source code, line 16 is the first command in `main`.

- Now we can run the program again:

```
(gdb) run
Starting program: /home/jharvard/noswap
Breakpoint 1, main () at noswap.c:16
16      int x = 1;
```

- And it's paused. Let's `print x`:

```
(gdb) print x
$1 = 0
```

  # But it's `0`. `gdb` has paused execution to just before line 16, so `x` has no assigned value but rather whatever was left in memory prior (a **garbage value**), and here we got lucky with a clean value of `0`.

- Let's say `next` and `print x` again:

```
(gdb) next
17      int y = 2;
(gdb) print x
$2 = 1
```

- And indeed we see `1`. What if we `print y`?

```
(gdb) print y
$3 = 134514064
```

- We see another garbage value as expected, with those bits from some other program that used the memory last to store something. Not to worry, we can proceed:

```
(gdb) next
19      printf("x is %i\n", x);
(gdb) print y
$4 = 2
```

- And `y` is `2` as expected. Moving on:

```
(gdb) n
x is 1
20      printf("y is %i\n", y);
(gdb) n
y is 2
21      printf("Swapping...\n");
(gdb) n
Swapping...
22      swap(x, y);
```

- But now we want to go inside `swap`, so we use the `step` command:

```
(gdb) step
swap (a=1, b=2) at noswap.c:33
33      int tmp = a;
(gdb) print tmp
$5 = -1209908752
```

- `tmp` has a garbage value again, but we can see it's correct after we initialize it.
- And remember we have `a` and `b` from the source code where `swap` is declared as `void swap(int a, int b)`, so it refers to the values passed in as `a` and `b`, and remember that those are copies of `x` and `y` held by `main`:

```
(gdb) next
34        a = b;
(gdb) print tmp
$6 = 1
(gdb) print a
$7 = 1
(gdb) next
35        b = tmp;
(gdb) print a
$8 = 2
```

- Now we see that `a` is `2`, after we said `next` and executed line 34. We can also check that `b` is `1` and `tmp` is still there:

```
(gdb) next
36    }
(gdb) print a
$9 = 2
(gdb) print b
$10 = 1
(gdb) print tmp
$11 = 1
```

- Let's say `continue` to finish the program:

```
(gdb) continue
Continuing.
Swapped!
x is 1
y is 2
[Inferior 1 (process 4946) exited normally]
(gdb)
```

- `gdb` didn't fix the problem, but helped us realize that our code didn't have an impact.

Strings

- So let's solve this problem. We're now peeling back a layer of abstraction, and realizing that a `string` doesn't actually exist, and instead is a `char*` with a name of `string`.

- Let's open `compare-0.c`[7]:

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

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

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

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

- We get two strings, and try to compare them. We make it and try it out:

```
jharvard@appliance (~/Dropbox/src4w): make compare-0
clang -ggdb3 -O0 -std=c99 -Wall -Werror    compare-0.c  -lcs50 -lm -o
 compare-0
jharvard@appliance (~/Dropbox/src4w): ./compare-0
Say something: Daven
Say something: Rob
You typed different things!
jharvard@appliance (~/Dropbox/src4w): ./compare-0
Say something: Gabe
Say something: Gabe
You typed different things!
jharvard@appliance (~/Dropbox/src4w): ./compare-0
Say something: Zamyla
```

[7] http://cdn.cs50.net/2014/fall/lectures/4/w/src4w/compare-0.c

```
Say something: Zamyla
You typed different things!
jharvard@appliance (~/Dropbox/src4w):
```

- So what's going on? The first time we got what we expected, but then we passed in two strings that were the same! All we're doing in the source code is getting them and checking if they are `==`.
- Well it turns out that, when we say something like:

```
string s = GetString();
  -----
  |   |
  -----
```

# where the box below `s` is storing `s`.

- Now `GetString`, in our first attempt, returned to us `Daven`, and also a `\0`, like so:

```
string s = GetString();
  -----    -------------------------
  |   |    | D | a | v | e | n |\0 |
  -----    -------------------------
```

- It looks like `Daven\0` is made up of many bytes, and if a `string` is only 4 bytes, 32 bits, how can we fit the entire string inside `s`? Well, the row of squares containing `Daven\0` are just bytes in memory. We can think of each byte as having a certain **address**, just as buildings might have an address like 33 Oxford Street, 34 Oxford Street, or 35 Oxford Street, and here, as an example, we start numbering each byte in memory:

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

- And now, we can sort of guess that `GetString` returns not the entire `string`, but rather the address to the `string`: `Daven` "lives" at `0x1`. So `s` only contains the address to that string:

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

```
  -----    ------------------------
  |0x1|    | D | a | v | e | n |\0 |
  -----    ------------------------
            0x1 0x2 0x3 0x4 0x5 0x6
```

- And this has been going on since we first introduced a `string`. All we get when we ask for a `string` is the location of where it begins.

  # Incidentally, programmers can put any address into a variable and try to jump to that area in memory, and we'll see how that could be problematic next time.

- But how do we know where a `string` ends, and the next `string` begins? Well, the `\0`, special null character, tells us when a `string` ends.
- So now if we look at the code of `compare.c`:

```
...
    // try (and fail) to compare strings
    if (s == t)
    {
        printf("You typed the same thing!\n");
    }
...
```

- we see that this fails since `s` and `t` are pointing to different addresses, since `t` is another `string`, and we're comparing the locations rather than the first character of each one, then the next, and so on:

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

string t = GetString();
  -----    ------------------------
  |...|    |   |   |   |   |   |   |
  -----    ------------------------
            ...
```

- So let’s fix this problem. If we had to implement it ourselves, we might compare letters in the two strings, one at a time, until we reached the end of one or both of them. But we don’t need to, thanks to the `strcmp` function as shown in `compare-1.c`[8]:

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

int main(void)
{
    // get line of text
    printf("Say something: ");
    char* s = GetString();

    // get another line of text
    printf("Say something: ");
    char* t = GetString();

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

- Notice that we use `strcmp` in line 18, which will return a negative number, or a positive number, or zero. Zero would mean that they are equal, and a positive or negative number would mean something like greater than or less than, if you wanted to alphabetize those strings.
- We’ve also switched to `char*` in lines 9 and 13, and really that’s the same as `string`, which we made up. For now, just know that the `*` means the address of something, and so a `char*`, as opposed to `int*`, means an address of a `char`.

[8] http://cdn.cs50.net/2014/fall/lectures/4/w/src4w/compare-1.c

- Going back to the board,

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

string t = GetString();
  -----    -------------------------
  |...|    |   |   |   |   |   |   |
  -----    -------------------------
            ...
```

  # that box with `0x1` is really a `char*`.

- Let's open `copy-0.c`[9]:

[9] http://cdn.cs50.net/2014/fall/lectures/4/w/src4w/copy-0.c

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

int main(void)
{
    // get line of text
    printf("Say something: ");
    string s = GetString();
    if (s == NULL)
    {
        return 1;
    }

    // try (and fail) to copy string
    string t = s;

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

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

    // success
    return 0;
}
```

- So in lines 10-14 we get a `string s` and checks that it's not `NULL` in case something went wrong. Otherwise, we might start going to invalid addresses in memory, and cause more and more problems.
- Let's try to copy the string in line 17, and capitalize the first character of `t`, `t[0]`, in line 23. And then we print them.
- But what's really happening, and where is the bug? Let's go back to line 17, where we set `string t` to `s`:

```
string t  =  s;
  ------     ------
  |0x50|     |0x50|
  ------     ------
```

- So we're setting `t` to point to the same address as `s`, but that just means when we change `t[0]`, the first letter in `t`, we also change `s[0]` since `s` points to the same thing:

```
string t  =  s;
  ------     ------
  |0x50|     |0x50|
  ------     ------

             ---------------------------------
        ...  | g | a | b | e |\0 |   |   |   |
             ---------------------------------
             0x50
```

- Let's run `copy-0`:

```
jharvard@appliance (~/Dropbox/src4w): ./copy-0
Say something: gabe
Capitalizing copy...
Original: Gabe
Copy:     Gabe
```

Memory Allocation

- Hm, we can *address* this problem with `copy-1.c`[10]:

[10] http://cdn.cs50.net/2014/fall/lectures/4/w/src4w/copy-1.c

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

int main(void)
{
    // get line of text
    printf("Say something: ");
    char* s = GetString();
    if (s == NULL)
    {
        return 1;
    }

    // allocate enough space for copy
    char* t = malloc((strlen(s) + 1) * sizeof(char));
    if (t == NULL)
    {
        return 1;
    }

    // copy string, including '\0' at end
    for (int i = 0, n = strlen(s); i <= n; i++)
    {
        t[i] = s[i];
    }

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

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

    // free memory
    free(s);
    free(t);

    // success
    return 0;
}
```

- This looks really complicated, but let's talk about the concept first. We'll use a loop to copy it character by character, but now we need to explicitly **allocate memory** for `t`:

```
string s = GetString();
  ------   --------------------
  |0x50|   | g | a | b | e |\0 |
  ------   --------------------
           0x50

string t
  ------   --------------------
  |    |   |                  |
  ------   --------------------
```

- We declared a `string t`, but how do we assign a value to it? Well, let's look at line 17, reproduced below:

```
char* t = malloc((strlen(s) + 1) * sizeof(char));
```

  # We started by writing `char* t`, which is creating a variable `t` that will store an address to a character, creating that left-hand box for `t`. Then we call `malloc`, a function that allocates a certain amount of memory. The argument we pass in is the size of the memory that we want, which is `strlen(s)`, the number of characters in `s`, `+ 1` for the `\0` character ending the string, all multiplied by the `sizeof` a `char`. (We know a character is one byte, but that may, rarely, vary from computer to computer.) Finally, `malloc` will return the address of that chunk of memory, which might be anywhere:

```
string s = GetString();
  ------   --------------------
  |0x50|   | g | a | b | e |\0 |
  ------   --------------------
           0x50

string t
  ------   --------------------
  |0x88|   |                  |
  ------   --------------------
           0x88
```

- Now we can access the memory as an array in a `for` loop, reproduced below:

```
    // copy string, including '\0' at end
    for (int i = 0, n = strlen(s); i <= n; i++)
    {
        t[i] = s[i];
    }
```

- We can do this because each string is stored with characters next to one another, so we can access them with this array notation.

- To recap, a `string` all this time was just an address of a character, a pointer, which in turn is just a number, that we conventionally write in hexadecimal.
- We also check if `t == NULL` because we might ask for more memory than `malloc` is able to give.
- And one final thing, if we return to what we were just looking at, we can replace line 4 below with line 5:

```
    // copy string, including '\0' at end
    for (int i = 0, n = strlen(s); i <= n; i++)
    {
        // t[i] = s[i];
        *(t + i) = *(s + i);
    }
```

- The `*` symbol can actually be used for two purposes. We've seen `char* t = …` which is declaring that `t` is a pointer to a `char`, but if we use `*` without a word like `char` in front of it, it becomes a **dereference operator**. That just means "go there" - if an address, like 33 Oxford Street, was written on paper like *(33 Oxford Street), then we would just go there.
- `t` is the address of the new piece of memory, and `s` is the address of the original piece, and `i` goes from `0` to `1` to `2` to `3` etc, so `t + i` is just another number, since these are all addresses with number values.
- So on the first pass of the loop, with `i = 0`, we're going to copy `g` from `0x50` to `0x88`:

```
string s = GetString();
  ------   ---------------------
  |0x50|   | g | a | b | e |\0 |
  ------   ---------------------
           0x50

string t
  ------   ---------------------
  |0x88|   | g |   |   |   |   |
  ------   ---------------------
           0x88
```

# On the next pass, `i = 1`, we'll copy `a` from `0x50 + 1`, `0x51`, to `0x88 + 1`, `0x89`, and you can see how it's going to proceed:

```
string s = GetString();
  ------   ---------------------
  |0x50|   | g | a | b | e |\0 |
  ------   ---------------------
           0x50

string t
  ------   ---------------------
  |0x88|   | g | a |   |   |   |
  ------   ---------------------
           0x88
```

- Let's look at a final program:

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

    x = malloc(sizeof(int));

    *x = 42;

    *y = 13;

    y = x;

    *y = 13;
}
```

  - It first declares two variables, `x` and `y` that aren't integers, but pointers to integers. Then we say `x = malloc(sizeof(int));`, or "give me enough memory to store an `int`", and the address returned by `malloc` will be stored in `x`.
  - Meanwhile, `*x = 42` is going to the address stored in `x`, and putting `42` in it.
  - Then we do the same with `y`, going to its address and putting `13` in it. But wait, `y` is probably a garbage value, some number left over from previous programs, but not contain an address to memory we should use to store an `int`. It's like trying to go into a building you don't own or have permission to enter, and bad things will happen.
- As an aside, David still remembers where he was when he understood pointers, sitting with his TF in the back of Eliot dining hall. So don't worry if none of this makes sense just yet (though I hope these notes are helpful)!
- Let's watch Pointer Fun with Binky[11].
  - Binky is a clay … figure that talks about this code with a narrator, using a "magic wand of dereferencing" to show what we just explained, in a different way.
  - There are three basic rules:
    - "Pointer and pointee are separate - don't forget to set up the pointee." (Don't forget to `malloc` something for `y`!)
    - "Dereference a pointer to access its pointee." (Use `*x` to go to the address stored in `x`!)
    - "Assignment (=) between pointers makes them point to the same pointee." (`x = y` sets them to the same address.)

[11] http://www.cs.stanford.edu/cslibrary/PointerFunCBig.avi