Week 4, continued

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Files, Headers, and Hex

- Depictions of trying to recover digital information in TV and movies often look like this[1], where characters say phrases like "zoom" and "enhance," that magically cause images to reveal details previously unseen.
  - # Sorry if we ruin some TV and movies for you, as you'll now be able to recognize when characters that are supposedly computer experts are making no sense at all.
- If we really try to "enhance" images, like that of TF Mary, we eventually see the pixels that compose the image, because there are only a finite number 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 can try to smooth so the image looks less pixellated, but there's still no more information in the photo than was contained in the original pixels.
- One of the topics for today is digital forensics, recovering information, and Problem Set 4 will be in this domain. You'll be manipulating and generating image files 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.
- One way to represent an image is with a **bitmap**, or BMP:

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

```
11000011
10111101
01011010
01111110
01011010
01100110
11000011
```

If we use 0 to represent black and 1 to represent white (which is standard), this bitmap produces a smiley face.

- What if we want to implement color? With 1 bit, we can only represent two states (black or white). To represent more states, we need more bits per pixel (sometimes 8, more commonly 24).
- JPEG is a file format you've likely used if you've ever taken a photo or looked at one on Facebook.

You'll be working with JPEGs on Problem Set 4 to recover photos from a camera memory card that David accidentally deleted.

- Even complex file formats like JPEGs can typically be identified by certain patterns of bits. Different file types, 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, called **signatures**, 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
```

- We've written these in decimal above, and we've worked with binary before, but 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.
- So you'll be looking in Problem Set 4 not for the decimal numbers `255`, `216`, `255`, but for the hexadecimal bytes `0xff`, `0xd8`, `0xff`.
- The debugger we've been working with in the CS50 IDE, `gdb`, might show you values in hexadecimal rather than decimal or binary.
- Another image file format that we referred to vaguely earlier 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).

  # As an aside, the scene in the Windows XP wallpaper[2] is now yellow and gloomy - or it was when someone went back to get another photo[3]!

- 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

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

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.
- All files are just 0s and 1s under the hood, and humans have defined conventions of what patterns of bits correspond to what kinds of data.

2. Structs

- C only has a couple more features that we haven't already discussed, one of which is called a **struct**.
- Let's say we want a variable to represent a student, with values such as name, age/birthdate, ID number, etc associated with it. How might we represent that student in code?

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

int main(void)
{
    string name;
    string dorm;
    int id;

    // do something with these variables
}
```

- What if we want to store two students?

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

int main(void)
{
    string name;
    string dorm;
    int id;

    string name2;
    string dorm2;
    int id2;

    // do something with these variables
}
```

- Wait, but we've already solved this problem of copy-pasting before - we can use an array instead.

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

int main(void)
{
    string names[3];
    string dorms[3];
    int ids[3];

    // do something with these variables

}
```

- But this is still pretty unwieldy - we don't really care about individual id numbers, just about the collection of data associated with a student, so it would be better if we could define a variable that somehow represented a student so then we could do things like this directly:

```
student s;
student t;

student class[3];
```

- 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`[4]:

```
#include <cs50.h>

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

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

  # The convention is to put typedefs like this into a separate header (or `.h`) file.

  # To `#include` a header file from the current directory, rather than one installed on your system, we use `#include "filename.h"` (with double-quotes) rather than `#include <stdio.h>` (with angle brackets).

- `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`[5]:

[4] http://cdn.cs50.net/2015/fall/lectures/4/w/src4w/structs.h
[5] http://cdn.cs50.net/2015/fall/lectures/4/w/src4w/structs-0.c

```
#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 an array named `students`, with each element of the type `student`. There are `STUDENTS` (which we've defined as a constant in line 8 to be `3`, using `#define`) elements in the `students` array.

- How do we access `name` and `house` and other fields, or items, in a `struct`?

```
...
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`.

- Now in `structs-1.c`[6], we bring files into the picture:

```
...
    // save students to disk
    FILE* file = fopen("students.csv", "w");
    if (file != NULL)
    {
        for (int i = 0; i < STUDENTS; i++)
        {
            fprintf(file, "%s,%s\n", students[i].name, students[i].dorm);
        }
        fclose(file);
    }

    // free memory
    for (int i = 0; i < STUDENTS; i++)
    {
        free(students[i].name);
        free(students[i].dorm);
    }
}
```

- We open a new file called `students.csv` for writing (hence the `"w"` argument), and then write the students' information to the file using `fprintf` - just like `printf`, but for printing to files rather than directly to the terminal.
- We'll come back to this, but `NULL` is a special value that can be returned by functions if something has gone wrong, so checking that `file != NULL` is just conventional error checking.
- We then `fclose` the file and `free` the memory used by the students' information (more on this later as well).
- If we compile and run this program, we'll see that the file is indeed created with the contents we expect:

```
jharvard@ide50:~/workspace/src4w $ ./structs-1
Student's name: Andi
Student's dorm: Berkeley
Student's name: Rob
```

[6] http://cdn.cs50.net/2015/fall/lectures/4/w/src4w/structs-1.c

```
Student's dorm: Thayer
Student's name: Maria
Student's dorm: Mather
jharvard@ide50:~/workspace/src4w $ ls
[... other files ...]
students.csv
```

```
# And if we open students.csv, we see that it now contains:
```

```
Andi,Berkeley
Rob,Thayer
Maria,Mather
```

- If we run the program again, we'd overwrite this file, because we opened this in `"w"` mode. If we instead wanted to add to the end, we could use `fopen` in `"a"` (append) mode instead.
- CSVs are useful because they can be opened natively by any standard spreadsheet program, but are not a proprietary format like `.xls`.
- This is a stepping stone to being able to persist information permanently, and you'll see a lot more of this on Problem Set 4.

3. Quick Reminder

- CS50 Lunch is Friday at 1:15pm as usual, RSVP at http://cs50.harvard.edu/rsvp.

4. Strings and Pointers

- We took some training wheels off on Monday, revealing that `string` doesn't exist - it's actually an alias for `char*` that we've created for you in the CS50 Library.
- Let's look back at `compare-0.c`[7]:

[7] http://cdn.cs50.net/2015/fall/lectures/4/w/src4w/compare-0.c

```
#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");
    }
}
```

- Recall that this didn't behave as expected:

```
jharvard@ide50:~/workspace/src4w $ ./compare-0
Say something: mom
Say something: mom
You typed different things!
```

- We clarified that strings are actually stored by their addresses in memory, rather than the actual sequence of characters. 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.
- So let's fix this problem. If we had to implement it ourselves, we might iterate through the two strings, comparing letters 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");
        }
    }
}
```

- Now we've made it clear that the string variables are actually `char*` variables, meaning that they will contain an address, not a string. `GetString()` doesn't return

[8] http://cdn.cs50.net/2015/fall/lectures/4/w/src4w/compare-1.c

a string directly, in the sense of a sequence of characters; it returns the address in memory of a string. Again, we're checking that neither of these is `NULL`.

One reason `GetString()` might return `NULL` is if the user provided too long a string and the program ran out of memory to store it in.

- 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.
- Now we can compare as we intend:

```
jharvard@ide50:~/workspace/src4w $ ./compare-1
Say something: mom
Say something: mom
You typed the same thing!
jharvard@ide50:~/workspace/src4w $ ./compare-1
Say something: mom
Say something: Mom
You typed different things!
```

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

[9] http://cdn.cs50.net/2015/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 check 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.
- We try to copy the string in line 17, and capitalize the first character of `t`, `t[0]`, in line 23. Then we print both strings.
- Let's run `copy-0`:

```
jharvard@ide50:~/workspace/src4w $ ./copy-0
```

[10] http://cdn.cs50.net/2015/fall/lectures/4/w/src4w/copy-1.c

```
Say something: mom
Capitalizing copy...
Original: Mom
Copy:     Mom
```

- Both the original and the copy appear to have been capitalized. 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|
  ------     ------

             ----------------------------
        ...  | m | o | m |\0 |   |   |   |
             ----------------------------
             0x50
              =>
             ----------------------------
        ...  | M | o | m |\0 |   |   |   |
             ----------------------------
             0x50
```

```
# Since both strings point to the same chunk of memory, both see the change.
```

5. Memory Allocation

- To fix this problem, we need another chunk of memory to put the copy in. See `copy-1.c`[10]:

```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`:

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

```
char* s = GetString();
  ------   -----------------
  |0x50|   | m | o | m |\0 |
  ------   -----------------
           0x50

char* t
  ------   -----------------
  |0x88|   |               |
  ------   -----------------
           0x88
```

  # We allocate enough memory for `strlen(s) + 1` chars, because we need a space for the `\0` character. (Although a `char` is essentially always 1 byte, we use the operator `sizeof` to make sure that we're allocating enough space even if we were on a system that used more than 1 byte per char. This is more relevant if we were allocating space for a type like `int` that's more likely to take up different amounts of space on different systems.)

- 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.

This method of moving around in memory is called **pointer arithmetic**, because we're doing math directly on addresses (by taking advantage of the fact that when we store a string, or when `malloc` gives us a chunk of memory, the addresses in that chunk of memory are consecutive!). This is just like if we were given *(33 Oxford Street + 1) to mean "go to the address one after 33 Oxford Street", so we'd go to 34 Oxford Street.

So on the first pass of the loop, with `i = 0`, we're going to copy `m` from `0x50` to `0x88`:

```
char* s = GetString();
  ------   -----------------
  |0x50|   | m | o | m |\0 |
  ------   -----------------
           0x50

char* t
  ------   -----------------
  |0x88|   | m |   |   |   |
  ------   -----------------
```

```
          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:

```
char* s = GetString();
  ------   -----------------
  |0x50|   | m | o | m |\0 |
  ------   -----------------
           0x50

char* t
  ------   -----------------
  |0x88|   | m | o | m |\0 |
  ------   -----------------
           0x88
```

So now when we capitalize the copy, the original string is unaffected:

```
char* s = GetString();
  ------   -----------------
  |0x50|   | m | o | m |\0 |
  ------   -----------------
           0x50

char* t
  ------   -----------------
  |0x88|   | M | o | m |\0 |
  ------   -----------------
           0x88
```

- When we do this:

```
*t = toupper(*t);
```

…we're only capitalizing the first letter, because `*t` refers to the contents of just the single address `t`. It's a pointer to a `char`, and your program has no way of knowing that char is part of a longer string unless you start iterating through it to look for the `\0` at the end.

- Bracket notation and pointer dereference notation are functionally equivalent here, but brackets are actually just **syntactic sugar** (simpler or clearer syntax abstracted on top of more complicated syntax) for dereferencing the pointer and doing pointer arithmetic.
- Recall that we introduced this problem of swapping two variables `a` and `b` with a temporary variable called `tmp`:

```
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 slice of memory (the `swap` function doesn't have access to the variables in `main`, but rather copies).

- We need a way to pass variables not by copy, but actually access the original variables in main. We can now solve this problem by passing `swap` the addresses where the actual values are in the memory belonging to `main`.
- Let's see this work in `swap.c`[11]:

[11] http://cdn.cs50.net/2015/fall/lectures/4/w/src4w/swap.c

```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);
    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;
    *a = *b;
    *b = tmp;
}
```

`tmp` is still just an integer, but `a` and `b` are addresses, or pointers.

In the parameter list of the `swap` function, the `*` before `a` and `b` signify that they are pointers, while in the actual code of the `swap` function, `*a` and `*b` signify dereferencing those pointers, i.e., going to those addresses and getting the values there.

But now we also need to pass the addresses of `x` and `y` to `swap`, not their values. We use `&x` and `&y` to mean "the address of x" and "the address of y".

- So to recap:

  # if `x` is a variable containing an `int` (or some other data type), we refer to it just as `x` if we want the value it contains (the `int` or whatever), and if we want its address, we use `&x`.

  # if `a` is a variable containing a pointer, e.g., an `int*` (or a pointer to any other data type), we refer to it as just `a` if we want the address, the pointer itself, and we use `*a` if we want the value contained at that address.

- 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 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, we haven't given `y` a value! So it's probably a garbage value, some number left over from previous programs, but not 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.

- Let’s watch Pointer Fun with Binky[12].
  # 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.)

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