Pointer Arithmetic (0:00-8:00)

- Take a look at pointers1.c:

```
int
main(int argc, char * argv[])
{
    int i, n;
    char * s;

    // get line of text
    s = GetString();
    if (s == NULL)
        return 1;

    // print string, one character per line
    for (i = 0, n = strlen(s); i < n; i++)
        printf("%c\n", *(s+i));

    // free string
    free(s);
}
```

- This program iterates through the characters of a string and prints out each one on a separate line
- Notice that s is just a pointer to the first character of the string
- As we add 1, 2, etc. to s, we point to the second, third, etc. characters of the string
- This is called pointer arithmetic: if we add 1 to a pointer, we advance ahead by once chunk of memory equal to the size of whatever we are pointing to (in this case, a char)
- We accomplish a similar function, but with an array of numbers, in pointers2.c:

```
int
main(int argc, char * argv[])
{
    int numbers[] = {1, 2, 3, 4, 5};

    printf("Size of array is %d.\n", sizeof(numbers));
    printf("Size of each element is %d.\n", sizeof(numbers[0]));
    for (int i = 0, n = sizeof(numbers) / sizeof(numbers[0]); i < n;
i++)
        printf("%d\n", *(numbers+i));
}
```

- This time, n is the size of the number of slots in the array, which is equal to the size of the whole array sizeof(numbers) divided by the size of each element sizeof(numbers[0])
- Again, we point to each number by adding i (0, 1, 2, etc.) to the address of the first number in the array

CS50's Library Revisited (8:00-23:00)

- As you know, many of the functions you have been using (GetInt(), GetString(), for instance) are not built in to C, but rather have been written by the teaching stuff in cs50.h
- You can use these functions only by #include-ing cs50.h in you programs
- Take a look at this snippet from cs50.h:

```
/*
 * bool
 *
 * Our own data type for Boolean variables.
 */

typedef enum {FALSE, TRUE} bool;
```

- enum declares a set of constants and automatically sets them to 0, 1,2,…
- So, under the hood, bool is just an integer. FALSE is 0 and TRUE is 1.
- Why do you need to #include cs50.h?
- This file contains the function prototypes. Recall that when we don't declare functions we get warnings from gcc.
- #include essentially accomplishes a copy and paste. It take the content of cs50.h and pastes it into the .c file you're compiling, thereby giving you all the function prototypes at the top of your program. Then when gcc compiles, everything it needs is there.
- Now we take a look at the cs50.c file. Here's the definition for GetInt():

```
int
GetInt()
{
    char c;
    int n;
    string line;

    /* try to get an int from user */
    while (TRUE)
    {
        /* get line of text, returning INT_MAX on failure */
        line = GetString();
        if (line == NULL)
            return INT_MAX;

        /* return int equivalent to text if possible */
        if (sscanf(line, " %d %c", &n, &c) == 1)
        {
            free(line);
            return n;
        }
        else
        {
            free(line);
            printf("Retry: ");
        }
    }
```

```
}
```

- How does GetInt() work?
    - First just use GetString() since we went through the trouble of writing it
    - Now do a sanity check. Did GetString() work? We accomplish this by checking to see if GetString() has returned a NULL pointer.
    - If indeed something has gone wrong we return immediately. Usually we like to return 0 or -1 as an error value, but this would become problematic if the user actually did enter one of these values. Instead we return the maximum value possible, INT_MAX, which has been defined as 2^31-1.
    - To scanf() we pass &n and &c, the locations where we want to put the text read in. We pass an address in order to affect the actual contents of that memory location, rather than just passing its value.
    - We put "%d %c" to give scanf the opportunity to pick up an int and a char. scanf() then returns the number of arguments that were filled.
    - If this is 1, we're golden. Otherwise, we know that some characters have been read in and ask for a retry.
    - What's this free business? Well, GetString() fills up memory with a string equivalent to the text read in (using malloc(), of course). Whether or not this is valid text, we eventually have to free up this memory. We accomplish this using the function free()
- Check out the rest of the library if you're interested in knowing how things work!

Structures (23:00-34:00)

- Right now, the only data types you can use are the ones C gives you: char, int, etc.
- What if you wanted to make your own data type?
- For instance, we want to make a database of students, each of whom has an id, a name and a house.
- At this point in time, we'd just have to make three arrays, one of IDS, one of names, and one of houses
- Problem: there's no inherent linking among them. We just have to assume that they match up in index.
- This does not scale very well. Think about how much information FAS has about you. Having so many arrays would become very unwieldy.
- Instead, we just make a struct. Here's what the syntax looks like:

```
typedef struct
{
    int id;
    char * name;
    char * house;
}
student;
```

- Inside the curly braces we put each of the data field types, and just outside the curly braces we put the name we want to give the struct
- Now, we can declare a database of 20 students by simply typing struct students[20]
- This accomplishes the same thing as writing out those three arrays, bu also ensures that each student's data is bound up together
- We can assign the fields by typing students[0].id = 1234 or students[0].name = GetString();
- Take a look at structs1.c

```
// class STUDENTS
#define STUDENTS 3


int
main(int argc, char * argv[])
{
    // declare class
    student class[STUDENTS];

    // populate class with user's input
    for (int i = 0; i < STUDENTS; i++)
    {
        printf("Student's ID: ");
        class[i].id = GetInt();

        printf("Student's name: ");
        class[i].name = GetString();

        printf("Student's house: ");
        class[i].house = GetString();
        printf("\n");
    }

    // now print anyone in Mather
    for (int i = 0; i < STUDENTS; i++)
        if (strcmp(class[i].house, "Mather") == 0)
            printf("%s is in Mather!\n\n", class[i].name);

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

```
    }
}
```

- We make a class of size STUDENTS, which is #defined as 3
- Then, for each student, we give the user the opportunity to fill in data
- It is common to put the structure declarations in a separate header file, so it appears in struct.h
- Now, suppose we want to print out the name of any student who lives in Mather. We accomplish this with the following code snippet:

```
for (int i = 0; i < STUDENTS; i++)
   if (strcmp(class[i].house, "Mather") == 0)
      printf("%s is in Mather!\n\n", class[i].name);
```

- strcmp() takes in two strings and returns 0 if they are exactaly the same, and -1 or +1 if they are not (depending on which comes first in lexicographic values)
- Remember that we can't just use == because that tests to see if the pointers are equal, rather than to see if they are character-for-character equal
- BUT now that we have allocated all this space, we have to get rid of it at the end of the program:

```
for (int i = 0; i < STUDENTS; i++)
    {
        free(class[i].name);
        free(class[i].house);
    }
```

- Otherwise, our program could eventually run out of memory

File I/O (34:00-40:00)

- Let's look at struct.c again, this time with the intention of saving the data that we read in
- We will accomplish this using file i/o (input/output)
- To use file i/o, we need to make a file pointer. First we open a file using fopen(), then we place the result of this function into a pointer:

```
FILE *fp = fopen("database", "w");
```

- Next we check to make sure that fp is NULL (some problem occurred in opening the file) and, assuming it's not, print all of our data to the file:

```
 if (fp != NULL)
 {
     for (int i = 0; i < STUDENTS; i++)
     {
         fprintf(fp, "%d\n", class[i].id);
         fprintf(fp, "%s\n", class[i].name);
         fprintf(fp, "%s\n", class[i].house);
     }
```

```
fclose(fp);
}
```

- fprintf is just like printf, but we pass it a file pointer that says where to print the data to
- pdoes the same thing, but prints to stdout, which basically means the screen
- When we're done, we close the file (just like in Microsoft Word).
- Now we run, enter our data, nothing seems to have happened until we ls and find a file called database.txt
- Open it up and we have all of our data in text form.
- On the other side, we can use fscanf to read from a file

Problem Set 4 (40:00-45:00)

- As a grad student, David worked with a forensic investigator to gather evidence on hard drives and flash drives
- What he did was write tools to unearth data on these devices that people thought they had gotten rid of
- Beware, when you delete a file, or even format it, you don't actually get rid of everything that was there!
- Your operating system has file allocation tables that maps file names to addresses
- When you drag your essay.doc to the recycle bin, the file is no longer associated with the location, but a slice of disk is still taken up by a whole bunch of 0's and 1's representing your essay!
- So when you delete a file, nothing happens to your document in memory – the computer simply "forgets" where it is
- One of the challenges of problem set 4 is to exploit this reality to uncover data that is assumed to be lost