Announcements (0:00-11:00)

Allocating Memory on the Heap (11:00-29:00)

- Recall copy1.c, whose purpose was to make a copy of a string and capitalize one of them.
- It was broken, as you recall, because it merely set two pointers equal (char *s2 = s1), rather than actually copying the entire string over
- We solved this problem by actually allocating memory for the s2 using malloc(), and then copying each character of the string over into our newly allocated block of memory
- malloc() reserves memory on the heap, so we don't have to worry about losing our variables when a function gets popped off the stack
- When we reserve memory on the heap, it stays there until we give it back using free()
- If you never call free(), and your program runs for a long time, your computer could slow down or even crash because all of the RAM gets eaten up
- When RAM starts getting eaten up, your computer will start to move stuff temporarily to the hard disk. But reading/writing from the disk is extremely slow, so this is one reason your computer will slow down.
- Pointers can become dangerous because they allow you to directly access memory
- A malicious user could potentially access memory you don't want them to access
- Remember that when we use malloc we specifiy the number of bytes we need by saying something like char * s2 = malloc((strlen(s1) + 1) * sizeof(char))
- We add one to strlen to make room for the null terminator \0 which must come at the end of all strings
- What is in s2 after this line (char *s2 = s1) is executed? s2 is just 32 bits of memory. It contains the address of the memory that malloc has given us.
- Note that we have to do a NULL check, because if malloc for some reason does not want to give us this memory (e.g., if it is out of memory), it will indicate that to us by returning a NULL pointer
- NULL is just a synonym for zero
- Notice at the end of copy2.c that we have to free the memory we have allocated
- We only explicitly allocate memory s2 in this main routine, but also GetString() allocates memory for s1. Therefore, we need to free both at the end of main.
- How do we know that GetString() allocates memory. By examining the contents of cs50.c…

Exploring the CS50 Library (29:00-51:00)

- To start off, let's take a look at cs50.h
- The first thing we notice is #ifndefh _CS50_H
    - This means, if the region defined as _CS50_H has not already been included, then include it

  - We do this because every time we say something like #include <cs50.h> the compiler will actually paste in the contents of cs50.h to the file where we are including it
  - If we have multiple .c files compiled into one executable, and each includes a particular library, then that library will get pasted multiple times
  - This is bad because it leads to compiler errors about functions being declared multiple times
  - #ifndef basically means, only paste in the library if it has not already been pasted in
- Next we see the inclusion of stdbool.h
  - This library creates the enumerated type bool, since Booleans are not a built in type in C
  - An enumerated type is simply a type that can only take on the value of a fixed set of constants
  - We define these constants using all words, but the compiler actually assigns integers to each, starting from zero, for the sake of comparison
  - So the enumerated type bool, which consists of {true, false} means that bool is a type which can take on the value of true or false
  - True is actually just a 1, and false is a 0, under the hood
- Next we see the definition of the string variable type
  - Typdef allows you to create your own data type
  - “typedef char * string;” means “a char * shall henceforth also be called a string” so you can use them interchangeably
  - We do this because strings are not a built in type in C, but we want to be able to have a string type
  - In this case, typdef just hides low level details, so you guys could use strings without knowing about char *’s, but it can also be useful to make an entire structure into its own type, as we’ll see later
- Now let’s look at a function.
- 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 (code reuse)
    - 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 sscanf() 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 sscanf the opportunity to pick up an int and a char. sscanf() 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()
- GetFloat(), GetChar() and GetDouble() are similar to GetInt().

Growing a Buffer (51:00-68:00)

- Now let's look at GetString()
- So far, the function we've seen to get input from the user is scanf()
- Problem with scanf() is that you have to pass it a buffer into which it should put whatever it reads in. That buffer must necessarily be of finite size. What if the user enters something greater than the buffer size? We could end up with buffer overflow and a segmentation fault.

- To avoid this problem our strategy in GetString() is to allocate some space for the buffer initially, but to *grow* the buffer if that ends up not being enough space.
- Here's the code that allocates space initially:

```
size = SIZE;
buffer = (string) malloc(size * sizeof(char));
if (buffer == NULL)
    return NULL;
```

- SIZE is #defined in cs50.h to be 128. We make a reasonable assumption that the string entered by the user is likely to be less than 128 characters long.
- We allocate 128 chars and cast that memory into a string to be explicit about the kind of pointer buffer will be. Remember that string is a char *, or pointer to a character.
- Then we check to make sure that buffer is not NULL. That could happen if we were asking for too much memory.
- Now we start reading in characters:

```
n = 0;
while ((c = fgetc(stdin)) != '\n' && c != EOF)
{
    /* grow buffer if necessary */
    /* this part omitted */

    buffer[n++] = c;
}
```

- Let's look at the while condition to start off with. fgetc() is "file get character". We're getting input from stdin, a built in file that refers to the user's keyboard. We keep getting characters until we encounter a newline or an EOF (end of file) character.
- Now, so long as n does not equal size-1, we put the next character into the next slot of the buffer. That's all that buffer[n++] = c means.
- But when n does get to size-1, the buffer is full. In that case we have to "grow" the buffer.
- Here's the code for that (omitted above):

```
if (n + 1 > capacity)
{
    // determine new capacity: start at CAPACITY then double
    if (capacity == 0)
        capacity = CAPACITY;
    else if (capacity <= (UINT_MAX / 2))
        capacity += 2;
    else
    {
```

```
            free(buffer);
            return NULL;
        }

        // extend buffer's capacity
        string temp = realloc(buffer, capacity * sizeof(char));
        if (temp == NULL)
        {
            free(buffer);
            return NULL;
        }
        buffer = temp;
    }
```

- The catch is that we can't simply tell the operating system to extend the buffer to whatever memory happens to be next to it. There might be something already occupying that space.
- Instead, we have to ask for a bigger buffer somewhere else, and copy over everything we have into the beginning of that buffer.
- If the size is greater than half the largest possible int (UINT_MAX/2), then we have to bail.
- Otherwise, double the size and malloc the new size. Assign the newly allocated memory to tmp, make sure tmp's not NULL, and copy over what we have into tmp.
- strncopy() copies the contents of buffer into tmp but only up to n characters
- This is better than strcopy() because it checks the bounds
- Finally, we reassign our new, bigger array tmp into the variable buffer
- So, why did we choose 128 as our starting size?
- We could have started with 1 or 2 and just doubled it as needed. But this would result in a lot of work if we had large input. malloc() and free() are very expensive operations that can hurt the performance of your program if you call them too much.
- 128 seems kind of arbitrary, but represents a carefully made design decision. The idea is that growing the buffer is a pretty costly operation that wants to be avoided if at all possible.