Dealing with a Seg Fault (0:00-8:00)

- Take another look at buggy6.c
- Because we index up to 10 in a 3-slot array, we get a whole bunch of "junk" in addition to the three desired scores
- If we push the limits of the program and index up to 1000, we get the message Segmentation Fault (core dumped)
- This just means you screwed up. When something really bad happens, usually involving memory, the operating system puts a file in the directory that has the contents of the ram. Presumably you can go look at your core and figure out what happened.
- By listing the files in the current directory, we take a look at core and notice that it's 300K.
- If the core is of no value to you, you probably should rm it because it's so big.

Strings as Arrays (8:00-17:00)

- What is an array? A collection of variables of an identical type.
- In terms of memory, an array is a contiguous chunk of RAM that contains an indexed list of variables.
- Actually you've been seeing character arrays for quite some time. A string is really just a character array.
- Take a look at string1.c:

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


int
main(int argc, char * argv[])
{
    char c;
    int i;
    string s;

    /* get line of text */
    s = GetString();

    /* print string, one character per line */
    if (s != NULL)
    {
        for (i = 0; i < strlen(s); i++)
        {
            c = s[i];
            printf("%c\n", c);
        }
    }
}
```

- First, note that we check to see if s is NULL. NULL means 0, and in this case is an error code that is returned by GetString() if some problem occurred in the function.
- When you get a return value from a function, it is good practice to check and see if you got an error code and then deal with it appropriately. Otherwise, you could accidentally refer to a memory location that does not contain anything (or at least nothing valuable to us).
- Now look at the loop in the body of the program. Here, we loop through the string to access each character.
- We can do this because strings are actually just arrays of characters. So when you type foo in as an input string, the computer actually makes a character array consisting of |f|o|o|\0
- The last character must always be \0 to indicate the end of the string. Strings are by convention always "null-terminated".
- In the same way we can use syntax like A[i] to index into an array, we can say s[i] for some string s to index into it.
- Example. s = "foo". s[0] = f.
- The function strlen() will tell you how many characters in the string. Probably implemented as a while loop with an incrementing variable.
- When we have a for loop, the condition is executed every single time. When we have a function in the condition, it gets run every time.
- This is inefficient and actually unnecessary since strlen(s) does not change in the body of the loop
- We can rewrite this for loop as for (i = 0, n = strlen(s); i < n; i++)
- Of course, if we had a function whose value was changing, then we would want to call the function every time at the start of the function

Capitalize.c (17:00-24:00)

- Take a look at capitalize.c

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


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

    /* get line of text */
    s = GetString();

    /* capitalize text */
    for (i = 0, n = strlen(s); i < n; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
```

```
            printf("%c", s[i] - ('a' - 'A'));
        else
            printf("%c", s[i]);
    }
    printf("\n");
}
```

- This time, we have gotten rid of the NULL check. Why? Because strlen() already knows how to detect if a string is of 0 length.
- What does this program do? It takes a string and reprints it in all caps. This works because of the math related to characters. (See Monday's scribe notes.)
- You might be wondering how we knew about the library string.h and the function strlen(). You can find out about useful library functions and things on documentation websites. A good one is cppreference.com. Some others are found on the website.

Command-Line Arguments (24:00-30:00)

- Whenever we write our main function we always give it some parameters like this:
  int main(int argc, char * argv[])
- Main is actually just a function that takes two arguments
- When we type the program's name at the prompt, we can follow it by words or numbers that will get passed into the program by means of these arguments
- Argc is an int representing the number of arguments
- Argv is an array of arrays of characters (an array of strings) that contains those arguments
- Note that the computer will throw away white space and treat anything separated by space as a separate argument
- Take a look at argv1.c

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

    /* print arguments */
    printf("\n");
    for (i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    printf("\n");
}
```

- This program iterates over the array of arguments and simply prints each one by referencing argv[i]
- Notice that argv[0] is a.out – the name of the program
- So the first actual argument will always be argv[1], and will go up to argc many arguments
- Take a look at argv2.c

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

    /* print arguments */
    printf("\n");
    for (i = 0; i < argc; i++)
    {
        for (j = 0, n = strlen(argv[i]); j < n; j++)
            printf("%c\n", argv[i][j]);
        printf("\n");
    }
}
```

- Now we again iterate over each argument from the command-line.
- But now for each argument, we iterate over its characters.
- We reference arrays within arrays using [][] notation
- argv[i][j] refers to the jth character of the ith array (remember, argv is an array of arrays)

CS 50's Library (30:00-35:00)

- These functions are not written as robustly as they could be
- For instance, GetString() allocates memory for a string, but you never free it.
- We will learn more about allocating and freeing memory later on

Introduction to Cryptography (35:00-53:00)

- When we evaluate the security of cryptographic schemes, we consider the point of view of an adversary trying to break them, and translate the process of breaking the code into a math problem
- The more difficult the math problem, the better the encryption scheme
- The Caesar cipher we looked at on Monday is represented by the following equation: $c_i = (p_i + k) \% 26$
- As you can see, it is an easily solvable problem because, with the help of a computer, we could simply try all the possible rotations in a very small amount of time
- As a result, the Caesar cipher is an extremely unsecure encryption method
- Vigenere cipher is similar but uses larger keys. Rather than rotating by a number, it rotates by a "word".
- You have a multi-letter key and rotate each letter in your plain text by a different letter in your key. (And just repeat the key if it's not long enough)
- If p = HELLO,WORLD and k = FOOBAR., then we get c by doing H+F = M, E+O=S, L+O = Z, etc., to come up with c = MSZMO,NTFZE
- (Note that we do not bother rotating punctuation)
- Now our formula is $c_i = (p_i + k_i) \% 26$

- How many different keys can we have of size n? 26^n
- This is a great improvement over Caesar because the key space is much larger, so it would take an adversary a much longer time to try all possibilities.
- But we still have two problems:
    - The longer the key, the better Caesar is. But longer keys are harder to remember.
    - If Alice sends a message to Bob, both Alice and Bob must know the key. (We call this symmetric key encryption.) But how can Alice secretly communicate the key to Bob?
- For this reason, we like to use asymmetric keys in the real world.
- In this method, everybody knows Bob's public key and can use it to encrypt a message to him. But only Bob knows the private key which will allow him to decrypt messages he receives.
- Any adversary who intercepts the message cannot decrypt the message because she does not have the private key.
- Here is one encryption scheme called RSA:

    Public key: (e,n)
    Private key: (d,n)
    To encrypt: $C = M^e \bmod n$
    To decrypt: $M = C^d \bmod n$
- In order to calculate the public and private keys so that this relationship exists, we must execute the following:
    1. Choose 2 large primes p and q
    2. Compute n= p*q
    3. Choose e that is coprime to (p-1)(q-1)
    4. Compute d such that (e*d) mod (p-1)(q-1) = 1
- How is this possible? We will not prove it here, so you will just have to believe us.
- If you are unwilling to blindly accept this as fact, you should consider taking cs220 with Professor Rabin, who has invented a couple of pretty sweet encryption schemes himself