CSE 29
Lecture 8 Summary

January 29, 2026

📣 Logistical Things

- Assignment 2 is due tonight, which includes:
  - PSet 2 on Prairie Learn
  - PA 2
  - Design Questions 2
  - We have gone over all the things you need to complete PA2
- Assignment 1 resubmission is also due tonight
  - More details on resubmission policy: https://ucsd-cse29.github.io/wi26/#assignments

🧠 Review Questions

Answers in speaker notes!

1. What does the SHA256 function do?
2. The start of the hash of my password is 0x01AB3F. What do you know about my password?
3. Write a `main()` function that prints the 1st command line argument.

strtok

What do you find interesting about this output?

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

// using strtok

int main() {
    char str[] = "Joe Politz,jpolitz@ucsd.edu,Instructor";
    printf("str@%p: %p \"%s\"\n", &str, str, str);

    char* a = strtok(str, ",");
    char* b = strtok(NULL, ",");

    printf("a@%p: %p \"%s\"\n", &a, a, a);
    printf("b@%p: %p \"%s\"\n", &b, b, b);
}

```

```
$ gcc strtok.c -o strtok
$ ./strtok
str@0x16b7b6e51: 0x16b7b6e51 "Joe Politz,jpolitz@ucsd.edu,Instructor"
a@0x16b7b6e40: 0x16b7b6e51 "Joe Politz"
b@0x16b7b6e38: 0x16b7b6e5c "jpolitz@ucsd.edu"
```

Observations & Questions from students

- On line 8, printing `&str` and `str` have the same value when formatted as a pointer (`%p`)
  - This is because `str` is a array locally declared in this function. GCC already has an implicit ampersand when printing the pointer of `str`, so doing `&str` is just being more explicit.
  - In this case if you asked for `sizeof(str)`, you would get a size corresponding to the actual size of the array (case 3 of sizeof, refer to Lecture 7 Summary)
- `strtok` *remembered* something, it returns a pointer to the same string even when it is not passed the string (passed `NULL` instead)
- If you call `strtok` without `NULL`, what happens?
  - It would start from the beginning of the input string
- Does `strtok` modify the original string?
  - Good question! We'll talk about this later on.

Lets add c and d, what do you observe?

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

// explore strtok

int main() {
  char str[] = "Joe Politz,jpolitz@ucsd.edu,Instructor"
;
  printf("str@%p: %p \"%s\"\n", &str, str, str);

  char* a = strtok(str, ",");
  char* b = strtok(NULL, ",");
  char* c = strtok(NULL, ",");
  char* d = strtok(NULL, ",");

  printf("a@%p: %p \"%s\"\n", &a, a, a);
  printf("b@%p: %p \"%s\"\n", &b, b, b);
  printf("c@%p: %p \"%s\"\n", &c, c, c);
  printf("d@%p: %p \"%s\"\n", &d, d, d);

}
```

Code

```
bash-3.2$ ./strtok
str@0x16f3bee21: 0x16f3bee21 "Joe Politz,jpolitz@ucsd.edu,I
nstructor"
a@0x16f3bee18: 0x16f3bee21 "Joe Politz"
b@0x16f3bee10: 0x16f3bee2c "jpolitz@ucsd.edu"
c@0x16f3bee08: 0x16f3bee3d "Instructor"
d@0x16f3bee00: 0x0 "(null)"
bash-3.2$
```

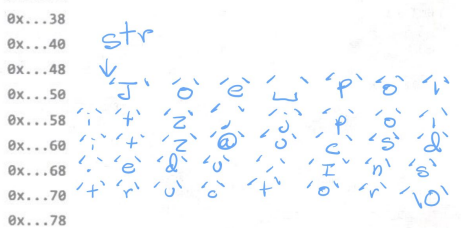
Output in terminal

Try filling out the memory diagram for this program!

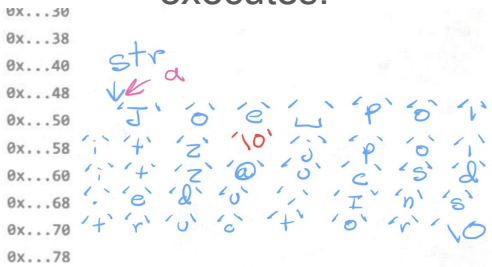
```
$ gcc strtok.c -o strtok
$ ./strtok
str@0x16b7b6e51: 0x16b7b6e51 "Joe Politz,jpolitz@ucsd.edu,Instructor"
a@0x16b7b6e40: 0x16b7b6e51 "Joe Politz"
b@0x16b7b6e38: 0x16b7b6e5c "jpolitz@ucsd.edu"
c@0x16b7b6e30: 0x16b7b6e6d "Instructor"
d@0x16b7b6e28: 0x0

str after strtok:
0x16b7b6e51: 'J' 'o' 'e' ' ' 'P' 'o' 'l' 'i'
0x16b7b6e59: 't' 'z' NUL 'j' 'p' 'o' 'l' 'i'
0x16b7b6e61: 't' 'z' '@' 'u' 'c' 's' 'd' '.'
0x16b7b6e69: 'e' 'd' 'u' NUL 'I' 'n' 's' 't'
0x16b7b6e71: 'r' 'u' 'c' 't' 'o' 'r' NUL
```

After creating
`str`

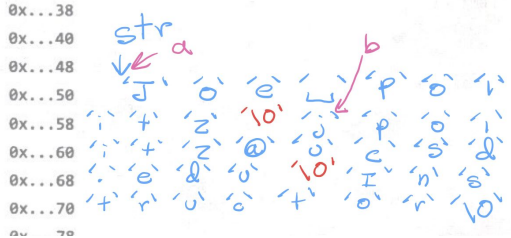


After
`a = strtok(str, ",")`
executes:

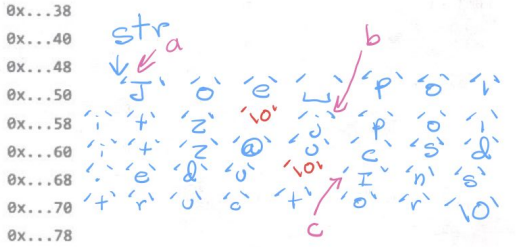


(`strtok` put a null terminator at the first comma it saw!)

After
`b = strtok(NULL, ",")`
executes:



After
`c = strtok(NULL, ",")`
executes:



Observations

- What's interesting about the address of the things stored in `a`, `b`, `c`, and `d`?
  - `a` stores something at 0x…21, `b` stores something at 0x…2c, etc.
  - The difference between 0x21 and 0x2c is 11! This is because `"Joe Politz"` is 10 bytes + 1 byte (null terminator)
- All of the pointers of the values stored in `a`, `b`, `c`, and `d` are within `str`
- The pointer of the object stored in `a` is the same as where `str` is
  - This is evidence that str is getting modified

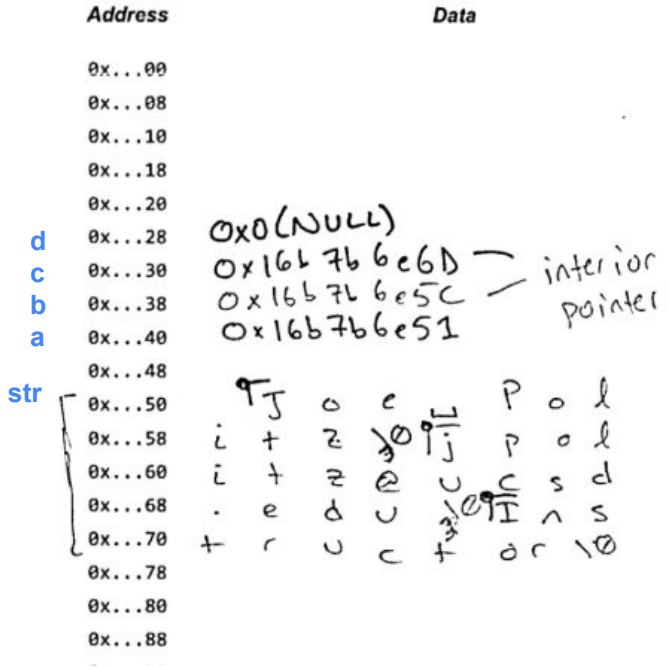
```
char *strtok(char *str, const char *delim);
```

- `strtok` has a **global variable** in a file where it is implemented that contains the string it was parsing from the last call
  - It replaces this each time `str` is not NULL
  - This is how `strtok` is keeping track of where it ends up!
- When it sees a delimiter, it replaces it with a **null terminator**, and **returns a pointer to the string**
  - `strtok` is modifying the input string
- Once it hits a null terminator in the input string, `strtok` knows that it is done

Interior Pointers

The pointers `a`, `b`, `c`, and `d` could be called **interior pointers**

- These are pointers INSIDE of an existing structure (`str`)
- `b` doesn't store a copy of a string, but rather it stores an address inside of an existing structure
- This is a cool thing that C can do
  - In other languages, you would get a copy of the string



Questions

- What if I passed in `str` to `strtok` instead of `NULL`?
  - It would restart strtok at the beginning of `str`. If you called `strtok(str, ",")` multiple times, you would get the same address over and over
- If you do `strtok` on str twice, what would happen?
  - You would keep getting the same address 0x…51 over and over.
- What if we want to start in the middle of a string?
  - We would need something to get the address to the middle of a string. For example, we could use something like indexing to get there (`&str[index]`).
- How does the function know which null terminator you are referencing?
  - `strtok` is careful and starts at the character right after the last delimiter found. It will never look at the same byte twice (unless you restart `strtok` on the same string)
- How does `strtok` actually remember?
  - There's a global variable in the file that implements `strtok`. This would be a file in `string.h`

Questions (continued)

- Could the delimiter be a character at a specific index?
  - No you can’t. strtok takes the delimiters you give it in the second argument and find those single characters as delimiters
  - Ex. If you do `strtok(str, “,;”)`, it would find both commas and semicolons and overwrite them with null terminators
  - The delimiter can’t be a index number
- If i give multiple delimiters, does it look for them in sequence?
  - No, it looks for any of them
- Could you use `strtok` to modify strings that were just modified by `strtok`?
  - Yes, they’re just normal strings now.

Shell

What is a shell?

- Shells are programs in your terminal to run your command
  - The terminal is the interface (the window/display and keyboard input mechanism), while the shell is the program that interprets and executes the commands you type into that interface
- Some examples of shells
  - bash - born again shell
  - zsh - mac default shell
  - fish - friendly interactive shell (Joe likes it)
  - Powershell - windows default shell

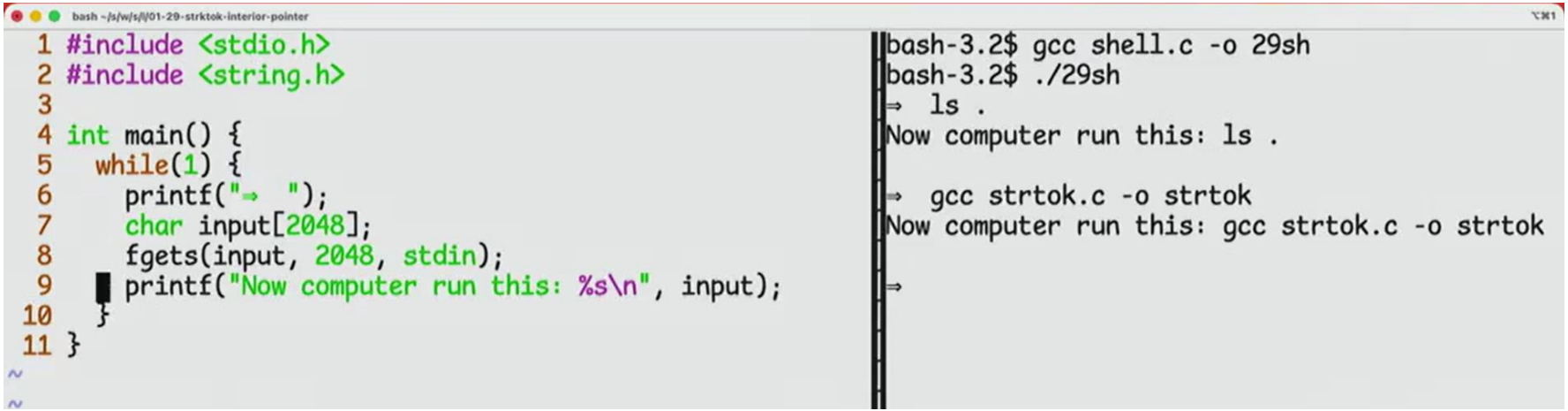
What does a shell do?

1. Prints a prompt
    a. Ex. “jpolitz[@ieng6~201] $”
    b. What prints out before your command
2. “Parses” a command and the arguments that the user types
    a. ls, cd, gcc, ./my_program arg1 arg2
3. Runs that command (by asking the OS)
4. Does this over and over again

- Other features
    a. You can see the history of your commands by using up arrow or Ctrl+R
    b. Control over working directory (look around your files)
    c. Hand control over to vim
    d. Tab for autocomplete
    e. I/O redirection (./program < input.txt > output.txt)

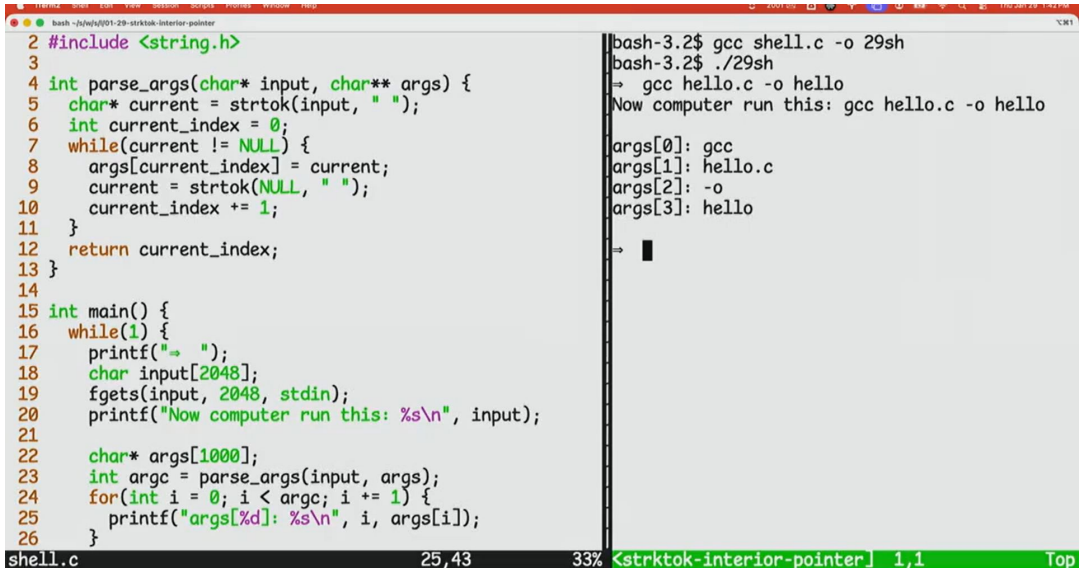
Let’s implement a shell!

1. Print out a prompt (let’s go with the ⟹ character)
2. Take in an input from user
    a. Create a big buffer called `input`, use `fgets`, and take in `stdin` (input from the terminal)



Let’s implement a shell! (cont.)

3. Ask the Operating System to do a command
   a. We have to process/parse the user input and then hand it off to the OS
   b. We need to make the string into an args array (break up the string, enter a null terminator)
   c. We can use `strtok`!

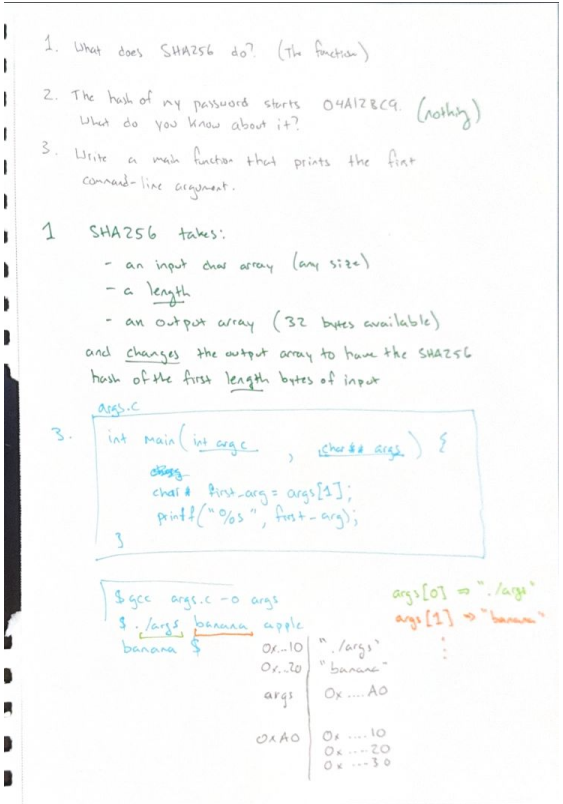


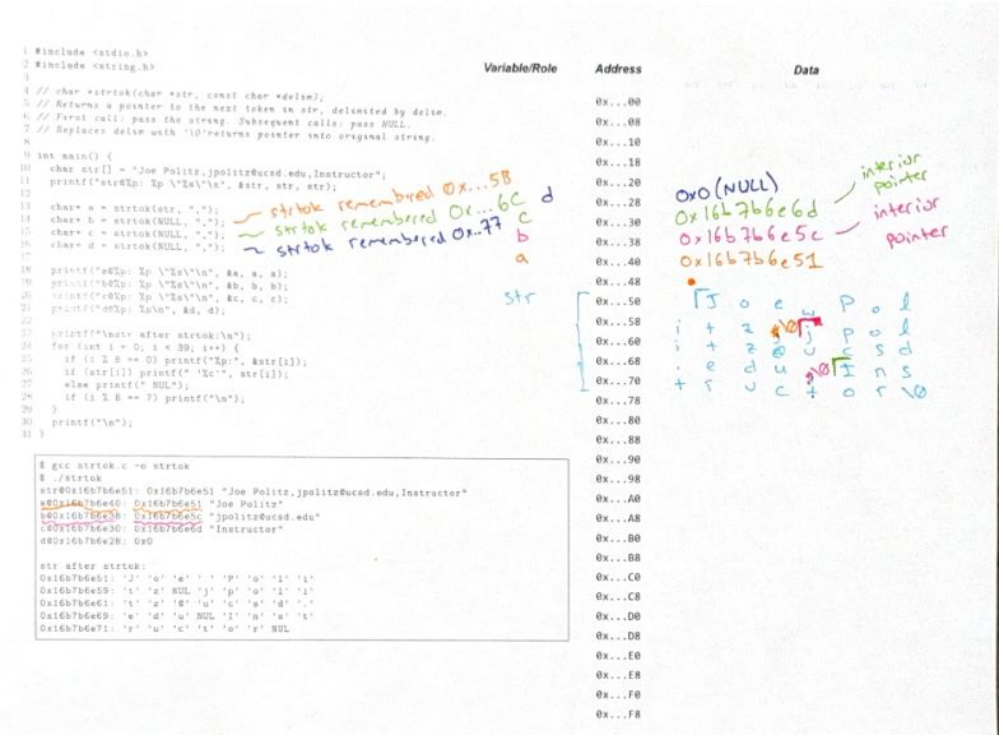
Looking Ahead

The next step of implementing the shell is to have the operating system (OS) run our commands!

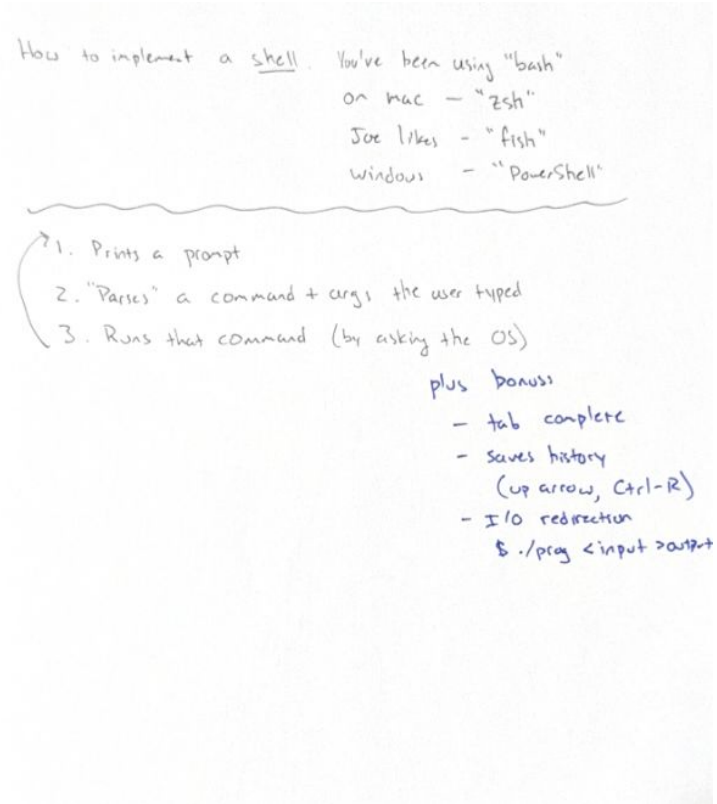
- Read about fork and exec on the textbook to learn more about this!
- We will be going over it in the following lectures
- https://diveintosystems.org/book/C13-OS/processes.html#_processes

Joe's Notes (11am)





Joe’s Notes (11am)

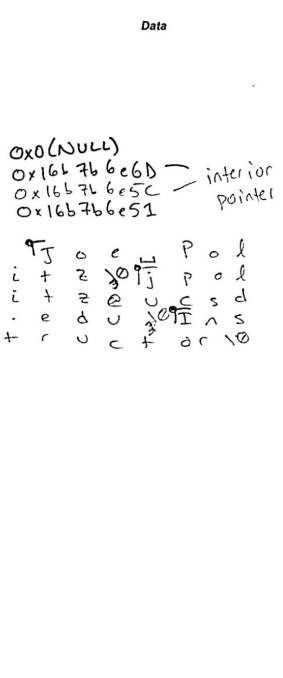


Joe's Notes (12:30pm)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  // char *strtok(char *str, const char *delim);
5  // Returns a pointer to the next token in str, delimited by delim.
6  // First call: pass the string. Subsequent calls: pass NULL.
7  // Replaces delim with '\0' returns pointer into original string.
8
9  int main() {
10   char str[] = "Joe Politz,jpolitz@ucsd.edu,Instructor";
11   printf("str@%p: %p \"%s\"\n", &str, str, str);
12
13   char* a = strtok(str, ",");
14   char* b = strtok(NULL, ",");
15   char* c = strtok(NULL, ",");
16   char* d = strtok(NULL, ",");
17
18   printf("a@%p: %p \"%s\"\n", &a, a, a);
19   printf("b@%p: %p \"%s\"\n", &b, b, b);
20   printf("c@%p: %p \"%s\"\n", &c, c, c);
21   printf("d@%p: %p\n", &d, d);
22
23   printf("\nstr after strtok:\n");
24   for (int i = 0; i < 39; i++) {
25     if (i % 8 == 0) printf("%p:", &str[i]);
26     if (str[i]) printf(" '%c'", str[i]);
27     else printf(" NUL");
28     if (i % 8 == 7) printf("\n");
29   }
30   printf("\n");
31 }
```

```
$ gcc strtok.c -o strtok
$ ./strtok
str@0x16b7b6e51: 0x16b7b6e51 "Joe Politz,jpolitz@ucsd.edu,Instructor"
a@0x16b7b6e40: 0x16b7b6e51 "Joe Politz"
b@0x16b7b6e38: 0x16b7b6e5c "jpolitz@ucsd.edu"
c@0x16b7b6e30: 0x16b7b6e6d "Instructor"
d@0x16b7b6e28: 0x0

str after strtok:
0x16b7b6e51: 'J' 'o' 'e' ' ' 'P' 'o' 'l' 'i'
0x16b7b6e59: 't' 'z' NUL 'j' 'p' 'o' 'l' 'i'
0x16b7b6e61: 't' 'z' '@' 'u' 'c' 's' 'd' '.'
0x16b7b6e69: 'e' 'd' 'u' NUL 'I' 'n' 's' 't'
0x16b7b6e71: 'r' 'u' 'c' 't' 'o' 'r' NUL
```



1. What does the SHA256 function do?
2. The start of the hash of my password is 0x01AB3F
   What do you know about my password?
3. Write a main() function that prints the 1st command line argument

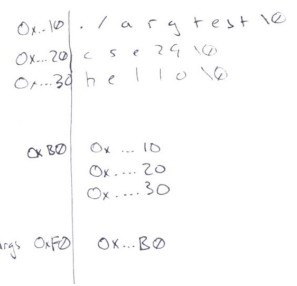
SHA256 takes:
- input char array (at least length bytes long, doesn't have to be C string)
- length
- output char array (at least 32 bytes long)

Calculates the hash of the first length bytes of input, stores the result in output (changes output)

argtest.c

```
int main(int argc, char** args) {
  char* first_arg = args[1];
  printf("%s\n", first_arg);
}
```

```
$ gcc argtest.c -o argtest
$ ./argtest cse29 hello
cse29
$
```



Joe's Notes (12:30pm)