CS 521: Systems Programming

Strings and I/O Streams

Lecture 8

Today's Schedule

- Strings
- I/O Streams

Today's Schedule

- **Strings**
- I/O Streams

C Strings

- In C, strings are just ***special*** arrays of characters:
  - `char str[] = "Hello World!"; // Mutable (array)`
  - `char *str = "Hello World!"; // Immutable (str. literal)`
- You can't see it, but the reason these character arrays are special is because they end in `\0`
  - The `NUL` terminator
- As we already discussed, we need to either pass dimensions along with arrays **OR** include some way of knowing where they end
  - `\0` means "the end!"

Strings as Arrays

- Let's look at C strings:

- Note how our string contains 6 characters, but the array representation has 7 due to the `NUL` byte
- `\0` is a control character
  - Just like `\n`, etc., we write it with two characters but it is just shorthand for a single character
  - Its value also happens to be 0 (decimal)
  - C string functions assume this is present; if it's not, you only have an **array of characters** and your program will crash

Some C String Library Functions

- `#include <string.h>`
- `strcpy` – copy one string to another
- `strcat` – concatenate two strings
- `strcmp` – test for string equality
- `strlen` – returns the length of the string (ignoring `\0`)
- `strstr` - search for a substring inside a string
- `strchr` - search for a character inside a string
- `sprintf` - create a string using `printf` -style formatting
- `strtok` – tokenize the string (split it up)
- Remember: documentation available in the `man` pages

Avoiding Buffer Overruns

- The string functions you just saw have one weakness…
  - If they lack the `\0`, they break!
  - This can lead to bugs, crashes, or even security issues
- Most C string functions also have a version that allows you to specify a fixed length
  - `strncmp`, `strncpy`, etc.
  - Notice the `n`: `strNcpy`
- Prefer these; they're ***slightly*** safer (**if** it makes sense…)

Copying a String [1/3]

- Let's say you want to copy one string into another:
  - `char str1[] = "Hello World!";`
  - `char *str2 = str1;`
- This doesn't make a copy; it just points to `str1`
- What about:
  - `char str2[] = str1;`
- Nope: `error: array initializer must be an initializer list or string literal`

Copying a String [2/3]

- We could loop through the array and copy each character into the other, but that's a lot of work
- Better solution: strcpy
  - (let's take a quick peek at the man page)

```
char str1[] = "Hello World!";
char str2[12];
strcpy(str2, str1);
printf("%s\n", str2);
```

But wait… This code has a big problem: array size

Copying a String [3/3]

Let's fix our bug:

```
char str1[] = "Hello World!";
char str2[13];
strcpy(str2, str1);
printf("%s\n", str2);
```

We could also create a much larger array to copy into.

Getting String Lengths

We can use the `strlen` function to find out how many characters (not including the `\0`) are in a string:

```
char str[] = "Hello";
printf("Length = %zd\n", strlen(str));
```

How would this be different than `sizeof(str)`?

Comparing Strings (equality)

- We unfortunately can't use `==` to check string equality
- Instead, we use the `strcmp` function
- It compares two strings based on their sort order
- If it returns `0`, the two strings are the same:
  - `if (strcmp(str_a, str_b) == 0) { /* same! */ }`
- **The following will not work as you might expect:**
  - `if (strcmp(str_a, str_b)) { /* same! */ }`
    - There is a **VERY** good chance you'll make this mistake!

Concatenating a String

`strcat` (and `strncat`) concatenate strings:

```
char *strcat(char *dest, const char *src);
```

```
char x[128] = "Hello";
char *y = "World";

strcat(x, " ");
strcat(x, y);
strcat(x, "!");

printf("%s\n", x);
```

Be careful: ***dest*** must be initialized before using `strcat`!

Concatenation: Another Option

You can use `printf`-style format specifiers to combine strings with `sprintf` and `snprintf`:

```
char a[128];

char x[] = "Hello";
char *y = "World";
sprintf(x, "%s %s!", x, y);
```

Here, you're basically "printing" to a string.

More String Functions

- There are a ***lot*** of string functions and things you can do with strings
- We will study more of them, but this gives you the foundation you need for now
- We often use **Input/Output Streams** to read or write strings

Today's Schedule

- Strings
- **I/O Streams**

Input/Output Streams

- ***Most*** useful programs will provide some type of input or output
- Our main approach thus far is printing via `printf`
- What happens if we want input from the user? We can use `scanf`:

```
printf("Please enter your age: ");
int age;
scanf("%d", &age);
printf("You are %d years old, huh? Wow!\n", age);
```

Reading a String With scanf

Let's greet the user:

```
char str[100];
printf("Enter your name: ");
scanf("%s", str);
printf("Hi, %s!\n", str);
```

- Wait a minute! Where's our &?!
- Well, remember that when we see the `[ ]` brackets, we're grabbing the actual values (via dereference)
  - The array name only = a pointer to the first element

Input/Output Streams

- Each program gets allocated three I/O streams by default:
  - `stdout` (standard output)
  - `stderr` (standard error)
  - `stdin` (standard input)
- These streams have different functions…

stdout

- When you call `printf`, you are writing to `stdout`
- This stream is designed for general program output; for example, if you run `ls` then the list of files should display on `stdout`
- You can use your shell to redirect `stdout` to a file:
  - `ls -l / > list_of_files.txt`

stderr

- The standard error stream is used for diagnostic information
  - Log messages often print to `stderr`
  - Program "usage" messages often go there too
- This way, program output can still be passed to other programs/files but we'll still see diagnostics printed to the terminal
  - Lets us know when something went wrong
    - Demo: `find` command
- Unlike stdout, stderr is not *buffered*
  - Will be flushed to the terminal immediately
    - More on that later

stdin

- The final stream, stdin, is how we provide program input (via `scanf`, for example)
- This can be entered by the user, or we can redirect input directly into a program:
  - `./my_prog < ./test_file.txt`
    - Acts like a phantom user is typing the contents of 'test_file.txt' into the program

Special Characters

- `>` – output redirection: send stdout to a file instead of the terminal
  - `cat something.txt > something_else.txt`
- `>>` – output redirection, but will append to the file instead of overwriting
- `<` – input redirection: read from file instead of stdin

“printf debugging”

- Let’s say we’re working on a bug and want to determine what’s wrong… `printf` to the rescue!
  - *cough*, *cough*, don’t do that, use logging… we’ll talk about this later!
- Unfortunately, sometimes printing to the terminal can be misleading
- The `printf()` may execute, but the program crashes before any output is displayed
- This occurs due to **Input/Output (I/O) Buffering**

I/O Buffering

- Input/output operations are slow: they have high **latency**
  - Printing to the terminal outputs to `stdout`
  - Writing to disk or controlling an external hardware device are also I/O operations
- These devices generally operate on **buffers**
  - Example: our terminal has a 8-byte buffer; we fill up the buffer before asking it to print the text
- You may have used buffered streams in Java to get better performance
- Buffered I/O collects multiple I/O operations, combines them, and then executes them as one big operation

Flushing the Output Stream

- Sometimes when debugging your program crashes before the buffer gets cleared
  - Data is lost before the buffer is flushed
- To make the print operation happen ***now***, we need to flush the output stream:
  - `fflush(stdout);`

Why not Always Flush?

- Flushing the buffer when it's not full or at inopportune times for the OS incurs more latency
- Performing the print operation takes several steps, and that takes time
- We can compare the performance of two C programs, one that flushes I/O and one that does not
  - Demo: `flush.c`