CS 220: Introduction to Parallel Computing

Strings

Lecture 5

Today's Agenda

- C Function Documentation
- Array Review
- Strings

Today's Agenda

- **C Function Documentation**
- Array Review
- Strings

C Function Documentation

- Unix has a utility called `man` – short for ‘manual’
- Entries in the Unix manual are called ‘man pages’
  - Many times your Google searches will locate man pages that have been converted to HTML
- There are several sections of man pages:
  1. User Commands
  2. System Calls
  3. C Library Functions
  4. …And many more

Reading `man` Pages

- Simple as entering `man <query>` in your terminal
  - man man
- You can also specify the section:
  - man 3 printf
  - This is important for our class: we need section 3 for C functions
- If you're not terminal-inclined, I also recommend this page:
  - http://en.cppreference.com/w/c

Today's Agenda

- C Function Documentation
- **Array Review**
- Strings

Array Review

- An array can contain several values of the same type
- Declare them like so: `int my_array[100];`
- Under the hood, arrays behave much like pointers:

```
     numbers[i]     *(numbers + i)
```

- When you pass an array to a function, it **decays** to a pointer
  - We lose both **type** and **dimension**
  - When this happens, we can't use **sizeof** to get the number of elements in the array

Thinking About Arrays & Pointers

- Here's a question: why do we have to use **&** here?

```
printf("Enter value %d: ", i);
scanf("%f", &numbers[i]);
```

- Arrays are basically pointers, so we shouldn't need to get the address, right?
- Recall what's going on behind the scenes:

```
numbers[i]     *(numbers + i)
```

- We're dereferencing the pointer to get the value it points at

Arrays & Pointers

- This means that we could do something like:

```
scanf("%d", &(*(&list[0])));
```

- Which could be written:

```
scanf("%d", &(*(&(*(list + 0)))));
```

- Yeah, that's really clear!
- The thing to remember:
  - When you access an array element with [ ], C is automatically dereferencing the pointer for you

Today's Agenda

- C Function Documentation
- Array Review
- **Strings**

C Strings

- In C, strings are nothing more than an array of characters:
```
    char str[] = "Hello World!";
```
  Or, as a pointer:
```
    char *str  = "Hello World!";
```
- Note, there **is** a difference between these two examples!
  - Array version: can be modified
  - Pointer version: **cannot** be modified

Mutability

- When you initialize a string like this:

```
char str[] = "Hello World!";
```

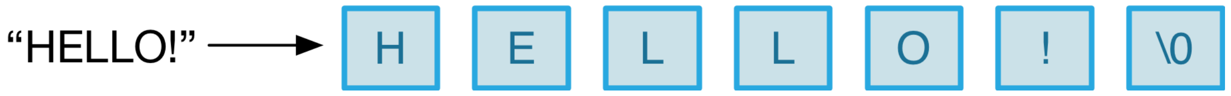
- The contents will be **copied** into the array and you can modify them (it is **mutable**)
- But when you do this:

```
char *str  = "Hello World!";
```

- You're just creating a pointer to a **string literal**
  - Embedded into your program (**immutable**)

Strings as Arrays

- Let’s look at a C string:



- Note how our string contains 6 characters, but the array representation has 7
- The \0 is the **NUL** byte, a control character
  - We write it with two characters, but in memory it only takes the space of a single character

What’s the use of NUL?

- First, the presence of the NUL byte indicates a **string** rather than just a plain old array of characters
- As we know, we can’t always reliably determine how large an array is unless we keep track of its size
  - Array decay
  - When working with the C string library, this would be extremely cumbersome!
- NUL allows the string manipulation functions to determine where the string ends

Character Arrays vs. Strings

- There is a subtle difference between a plain character array and a string
- A **string** is terminated by **`NUL`** (`\0`)
- If you use a function that expects a string, make sure it contains the NUL byte
  - Not doing so will likely lead to segmentation violations (invalid memory accesses)
  - Why?

The C String Library

- `#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)
- `strtok` – tokenize the string (split it up)
- Documentation available in the `man` pages

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:

```
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
  - strcpy will go ahead and fill the rest with \0

Reading a String

- 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

Getting String Lengths

```c
char str[] = "Hello";
/* Does not include \0: */
printf("Length = %zd\n", strlen(str));
```

Demo: Working with Strings