Week 5 - Friday

CS222

Last time

- What did we talk about last time?
- Finished arrays
- Strings

Questions?

Project 2

String functions

Function	Use
`strcpy(char destination[], char source[])`	Copies **source** into **destination**
`strncpy(char destination[], char source[], size_t n)`	Copies the first **n** characters of **source** into **destination**
`strcat(char destination[], char source[])`	Concatenates **source** onto **destination**
`strncat(char destination[], char source[], size_t n)`	Concatenates the first **n** characters of **source** onto **destination**
`strcmp(char string1[], char string2[])`	Returns negative if **string1** comes before **string2**, positive if **string1** comes after **string2**, zero if they are the same
`strncmp(char string1[], char string2[], size_t n)`	Same as **strcmp()**, but only compares the first **n** characters
`strchr(char string[], char c)`	Returns pointer to first occurrence of **c** in **string** (or **NULL**)
`strstr(char haystack[], char needle[])`	Returns pointer to first occurrence of **needle** in **haystack** (or **NULL**)
`strlen(char string[])`	Returns length of **string**

String library

- To use the C string library
  - `#include <string.h>`
- There are a few more functions tied to memory copying and finding the last rather than the first occurrence of something
- There is also a string tokenizer which works something like the **`split()`** method in Java
  - It's much harder to use
- Functions in the string library go until they hit a null character
  - They make no guarantees about staying within memory bounds

String operations

- They're all done with the string library!
- Remember that strings are arrays
- There is no concatenation with +
- There is no equality with ==
  - You can compare using == without getting a warning, but it is meaningless to do so
- You cannot assign one string to another with = because they are arrays
  - You will eventually be able to do something similar with pointers

Pointers

Pointers

- A **pointer** is a variable that holds an address
- Often this address is to another variable
- Sometimes it's to a piece of memory that is mapped to file I/O or something else
- Important operations:
  - Reference (**&**) gets the address of something
  - Dereference (*****) gets the contents of a pointer

Declaration of a pointer

- We typically want a pointer that points to a certain kind of thing
- To declare a pointer to a particular type

```
type * name;
```

- Example of a pointer with type **`int`**:

```
int * pointer;
```

Whitespace doesn't matter!

- Students sometimes get worried about where the asterisk goes
- Some (like me) prefer to put it up against the type:

```
char* reference;
```

- Some like to put it against the variable:

```
char *reference;
```

- It is possible to have it hanging in the middle:

```
char * reference;
```

- Remember, whitespace doesn't matter in C

Reference operator

- A fundamental operation is to find the address of a variable
- This is done with the reference operator (**&**)

```
int value = 5;
int* pointer;
pointer = &value;
//pointer has value's address
```

- We usually can't predict what the address of something will be

Dereference operator

- The reference operator doesn't let you do much
- You can get an address, but so what?
- Using the dereference operator, you can read and write the contents of the address

```
int value = 5;
int* pointer;
pointer = &value;
printf("%d", *pointer); //prints 5
*pointer = 900; //value just changed!
```

Aliasing

- Java doesn't have pointers
  - But it does have references
  - Which are basically pointers that you can't do arithmetic on
- Like Java, pointers allow us to do aliasing
  - Multiple names for the same thing

```
int wombat = 10;
int* pointer1;
int* pointer2;
pointer1 = &wombat;
pointer2 = pointer1;
*pointer1 = 7;
printf("%d %d %d", wombat, *pointer1, *pointer2);
```

Pointer arithmetic

- One of the most powerful (and most dangerous) qualities of pointers in C is that you can take arbitrary offsets in memory
- When you add to (or subtract from) a pointers, it jumps the number of bytes in memory of the size of the type it points to

```
int a = 10;
int b = 20;
int c = 30;
int* value = &b;
value++;
printf("%d", *value); //what does it print?
```

Arrays are pointers too

- An array **is** a pointer
  - It is pre-allocated a fixed amount of memory to point to
  - You can't make it point at something else
- For this reason, you can assign an array directly to a pointer

```
int numbers[] = {3, 5, 7, 11, 13};
int* value;

value = numbers;
value = &numbers[0]; //exactly equivalent

//What about the following?
value = &numbers;
```

Surprisingly, pointers are arrays too

- Well, no, they aren't
- But you can use array subscript notation (**[ ]**) to read and write the contents of offsets from an initial pointer

```
int numbers[] = {3, 5, 7, 11, 13};
int* value = numbers;

printf("%d", value[3] ); //prints 11
printf("%d", *(value + 3) ); //prints 11
value[4] = 19; //changes 13 to 19
```

Don't try this at home!

- We can use a pointer to scan through a string

```
char s[] = "Hello World!"; //13 chars

char* t = s;
do
{
  printf("(%p): %c %3d 0x%X\n",
     t, *t,    (int)*t, (int)*t);
} while (*t++); //why does this work?
```

Or what if we pretend...

- That it's an **`int`** pointer

```
char s[] = "Hello World!"; //13 chars
int* bad = (int*)s; //unwise...

do
{
  printf("(%p): %12d 0x%08X\n", bad,
                 *bad, *bad);
} while (*bad++);
```

void pointers

- What if you don't know what you're going to point at?
- You can use a **void***, which is an address to…something!
- You have to cast it to another kind of pointer to use it
- You can't do pointer arithmetic on it
- It's not useful very often

```
char s[] = "Hello World!";
void* address = s;
int* thingy = (int*)address;
printf("%d\n", *thingy);
```

Why do we care about pointers?

- There are some tricks you can do by accessing memory with pointers
- You can pass pointers to functions allowing you to change variables from outside the function
- Next week we are going to start allocating memory dynamically
  - Arrays of arbitrary size
  - Structs (sort of like classes without methods)
- We need pointers to point to this allocated memory

Functions that can change arguments

- In general, data is passed **by value**
- This means that a variable cannot be changed for the function that calls it
- Usually, that's good, since we don't have to worry about functions screwing up our data
- It's annoying if we need a function to return more than one thing, though
- Passing a pointer is equivalent to passing the original data **by reference**

Example

- Let's think about a function that can change the values of its arguments

```
void swapIfOutOfOrder(int* a, int* b)
{
  int temp;
  if( *a > *b )
  {
     temp = *a;
     *a = *b;
     *b = temp;
  }
}
```

How do you call such a function?

- You have to pass the addresses (pointers) of the variables directly

```
int x = 5;
int y = 3;
swapIfOutOfOrder(&x, &y);//will swap x and y
```

- With normal parameters, you can pass a variable or a literal
- However, you **cannot** pass a reference to a literal

```
swapIfOutOfOrder(&5, &3); //insanity
```

Lab 5

Upcoming

Next time…

- More on pointers
- Command line arguments

Reminders

- **Finish Project 2**
  - **Due tonight by midnight**
- Exam 1 next Friday