Week 5 - Wednesday

CS222

Last time

- What did we talk about last time?
- Arrays

Questions?

Project 2

Quotes

Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter.

Eric S. Raymond

Array example

- Write a program that reads an integer from the user saying how many values will be in a list
  - Assume no more than 100
  - If the user enters a value larger than 100, tell them to try a smaller value
- Read these values into an array
- Find
  - Maximum
  - Minimum
  - Mean
  - Variance
  - Median
  - Mode

Review of Compiling Multiple Files

Components

- C files
  - All the sources files that contain executable code
  - Should end with **`.c`**
  - Should never be **`#include`**d
  - Must be compiled with **`gcc`**
- Header files
  - Files containing extern declarations and function prototypes
  - Should end with **`.h`**
  - Should be **`#include`**d
  - Should not be compiled with **`gcc`**
- Makefile
  - File used by Unix make utility
  - Should be named either **`makefile`** or **`Makefile`**

C files

- You can have any number of **`.c`** files forming a program
- Only one of them should have a main() function
- If the functions in a **`.c`** file will be used in other files, you should have a corresponding **`.h`** file with all the prototypes for those functions
  - **`whatever.c`** should have a matching **`whatever.h`**
- Both the .c file that defines the functions and any that use them should include the header

Header files

- Sometimes header files include other header files
- For this reason, it is wise to use conditional compilation directives to avoid multiple inclusion of the contents of a header file
- For a header file called **`wombat.h`**, one convention is the following:

```
#ifndef WOMBAT_H
#define WOMBAT_H

//maybe some #includes of other headers
//lots of function prototypes

#endif
```

Compiling

- When compiling multiple files, you can do it all on one line:

```
gcc main.c utility.c wombat.c -o program
```

- Alternatively, you can compile files individually and then link them together at the end

```
gcc -c main.c
gcc -c utility.c
gcc -c wombat.c
gcc main.o utility.o wombat.o -o program
```

Makefile

- Compiling files separately is more efficient if you are only changing one or two of them
- But it's a pain to type the commands that recompile only the updated files
- That's why makefiles were invented

```
program: main.o utility.o wombat.o
	gcc main.o utility.o wombat.o -o program

main.o: main.c utility.h wombat.h
	gcc -c main.c

utility.o: utility.c utility.h
	gcc -c utility.c

wombat.o: wombat.c wombat.h
	gcc -c wombat.c

clean:
	rm -f *.o program
```

Strings

There are no strings in C

- Unfortunately, C does not recognize strings as a type
- A string in C an array of **`char`** values, ending with the null character
- Both parts are important
  - It's an array of **`char`** values which can be accessed like anything else in an array
  - Because we don't know how long a string is, we mark the end with the null character

Null character

- What is the null character?
- It is the very first char in the ASCII table and has value **0** (zero)
- It is unprintable
- You can write it
  - as a **`char`**: **`'\0'`**
  - as an **`int`**: **`0`**
  - as a constant: **`NULL`**
- It is **not** the same as **`EOF`** (which is **`-1`** as an **`int`** value)
- If you allocate memory for a string, you need enough for the length **plus** one extra for the null

String literals

- A string literal (`"yo, yo, yo!"`) in C is a `char` array somewhere in memory
- It is read-only memory with global scope
  - Maybe it's in the Global or BSS segment (or even some even more obscure segment)
- You can throw a string literal into an array:

```c
char word[] = "wombat";
```

- Doing so is **exactly** like doing the following:

```c
char word[] = {'w','o','m','b','a','t','\0'};
```

Using `printf()`

- You can print out another string using **`printf()`**

```
printf("The word of the week is: \"%s.\"\n",
"exiguous");
```

- Even **`printf()`** is only looking until it hits a null character
- What would happen in the following scenario?

```
char letters[5];
int i = 0;
for(i = 0; i < 5; i++ )
      letters[i] = 'A';

printf("The word of the week is: \"%s.\"\n",
letters);
```

Practice

- Write a function that finds the length of a string
- Write a function that reverses a string
  - First you have to find the null character

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

Quiz

Upcoming

Next time…

- Introduction to pointers
- Lab 5

Reminders

- Keep reading K&R chapter 5
- Finish Project 2
  - Due Friday
- Exam 1 next Friday