Week 5 - Monday

COMP 2400

Last time

- What did we talk about last time?
- Scope
- Systems programming

Questions?

Project 3

Quotes

A C program is like a fast dance on a newly waxed dance floor by people carrying razors.

Waldi Ravens

Single file system

- In Windows, each drive has its own directory hierarchy
  - `C:` etc.
- In Linux, the top of the file system is the **root directory /**
  - Everything (including drives, usually mounted in `/mnt`) is under the top directory
  - `/bin` is for programs
  - `/etc` is for configuration
  - `/usr` is for user programs
  - `/boot` is for boot information
  - `/dev` is for devices
  - `/home` is for user home directories

File permissions

- Every file has a UID and GID specifying the user who owns the file and the group the file belongs to
- For each file, permissions are set that specify:
  - Whether the owner can read, write, or execute it
  - Whether other members of the group can read, write, or execute it
  - Whether anyone else on the system can read, write, or execute it
- The **`chmod`** command changes these settings (**`u`** is for owner, **`g`** is for group, and **`o`** is everyone else)
- Example that adds the execute (**`x`**) permission to others (**`o`**) on a file called **`script.sh`**:

```
chmod o+x script.sh
```

File I/O

- All I/O operations in Linux are treated like file I/O
- Printing to the screen is writing to a special file called **`stdout`**
- Reading from the keyboard is reading from a special file called **`stdin`**
- When we get the basic functions needed to open, read, and write files, we'll be able to do almost any kind of I/O

Arrays

Declaration of an array

- To declare an array of a specified **`type`** with a given **`name`** and a given **`size`**:

```
type name[ size ];
```

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

```
int list[ 100 ];
```

Differences from Java

- When you declare an array, you are creating the whole array
- There is no second instantiation step
  - It is possible to create dynamic arrays using pointers and **`malloc()`**, but we haven't talked about it yet
- You must give a fixed size (literal integer or a **`#define`** constant) for the array
  - The version of **`gcc`** we are using allows variables, but some older (and newer) versions of C do not
- These arrays sit on the stack in C
  - Creating them is fast, but inflexible
  - You have to guess the maximum amount of space you'll need ahead of time

Accessing elements of an array

- You can access an element of an array by **indexing** into it, using square brackets and a number

```
list[9] = 142;
printf("%d", list[9]);
```

- Once you have indexed into an array, that variable behaves exactly like any other variable of that type
- You can read values from it and store values into it
- **Indexing starts at 0 and stops at 1 less than the length**
  - Just like Java

Length of an array

- The length of the array must be known at compile time
  - Our version of **`gcc`** has looser rules about this, but C90 insists on true constants
- There is no **`length`** member or **`length()`** method
- It's common to keep track of how many elements are used in an array with a separate length variable

```
int list[100];
list[0] = 5;
list[1] = 17;
int length = 2;
```

Arrays start filled with garbage

- When you create an array, it is **not** automatically filled with any particular value
- Inside the array (like any variable in C) is garbage
- With regular variables, you might get a warning if you use a variable before you initialize it
- With an array, you won't

Explicit initialization

- Explicit initialization can be done with a list:

```
int primes[10] = {2, 3, 5, 7, 11, 13, 17,
  19, 23, 29};
```

- You can omit the size if you use an explicit initialization because the compiler can figure it out

```
char grades[] = {'A', 'B', 'C', 'D', 'F'};
```

memset()

- The C standard library has a function called **memset()** that can set all the bytes in a chunk of memory to a particular value
- Using it is guaranteed to be no slower than using a loop to initialize all the values in your array
  - It usually uses special instructions to set big chunks of memory at the same time

```
int values[100];
// Zeroes out array
memset(values, 0, sizeof(int)*100);
char letters[26];
// Sets array to all 'A's
memset(letters, 'A', sizeof(char)*26);
```

memcpy()

- **memset()** is mostly useful for initialization (and usually only for zeroing things out)
- **memcpy()** is a fast way to copy values from one array to another
  - Again, it's at least as fast as using your own loop
  - Again, it's somewhat dangerous since it lets you write memory places en masse

```
int cubes[100];
int copy[100];
for (int i = 0; i < 100; i++)
  cubes[i] = i*i*i;
memcpy(copy, cubes, sizeof(int)*100);
```

Passing arrays to functions

- When using an array in a different function, you usually **have** to pass in the length
- The function receiving the array has no other way to know what the length is
- The function should list an array parameter with empty square brackets on the right of the variable
- No brackets should be used on the argument when the function is called
- Like Java, arguments are passed by value, but the contents of the array are passed by reference
  - Changes made to an array in a function *are* seen by the caller

Array to function example

- Calling code:

```
int values[100];
for(int i = 0; i < 100; i++ )
  values[i] = i + 1;
reverse(values, 100);
```

Array to function example

- Function:

```
void reverse(int array[], int length)
{
  int start = 0;
  int end = length - 1;
  int temp = 0;
  while( start < end )
  {
     temp = array[start];
     array[start++] = array[end];
     array[end--] = temp;
  }
}
```

Returning arrays

- In C, you **can't** return the kind of arrays we're talking about
  - Why?
- They're allocated on the stack
- When a function returns, all its memory disappears
- If you dynamically allocate an array with **`malloc()`**, you can return a pointer to it

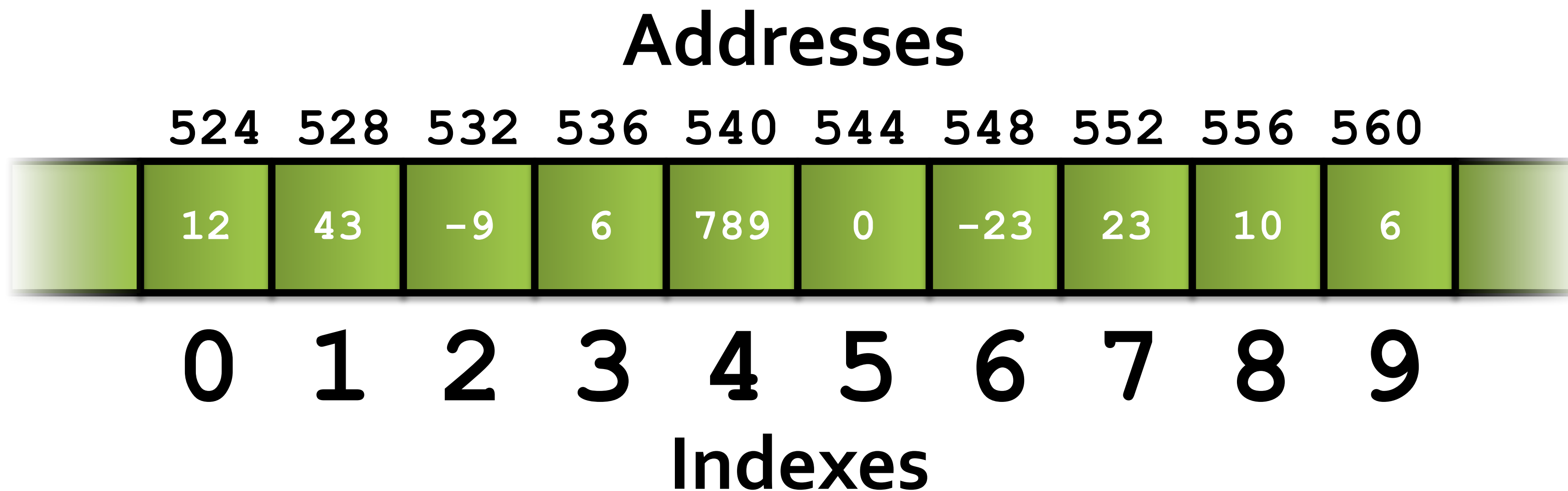
Array Memory

Memory

- An array takes up the size of each element times the length of the array
- Each array starts at some point in computer memory
- The index used for the array is actually an offset from that starting point
- That's why the first element is at index **0**

A look at memory

- We can imagine that we have an array of type **`int`** of length 10
- Let's say the array starts at address 524



Multidimensional arrays

- It is legal to declare multidimensional arrays in C

```
char board[8][8];
```

- They'll work just as you would expect
- **Except!** You have to give the second dimension when passing to a function (otherwise, it won't know how big of a step to take when going from row to row)

```
void clearBoard (char board[][8])
{
  for(int i = 0; i < 8; i++ )
    for(int j = 0; j < 8; j++ )
        board[i][j] = ' ';
}
```

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

Upcoming

Next time…

- Strings

Reminders

- Keep reading K&R chapter 5
- Start on Project 3
  - Form teams if you haven't yet!
- Exam 1 next Monday!