Week 5 - Monday

CS222

Last time

- What did we talk about last time?
- Processes
- Lab 4

Questions?

Project 2

Quotes

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

Waldi Ravens

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
- 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
- There is no **`length`** member or **`length()`** method
- It is possible to find out how many bytes a statically allocated array uses with **`sizeof`**
  - But you can only do that in the function where the array is defined!

```
int list[100];
int size = sizeof(list);          //400
int length = size/sizeof(int); //100
```

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

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

Passing arrays to functions

- Using an array in a function where it wasn't created is a little different
- You have to pass in the length
- The function receiving the array has no other way to know what the length is
  - **`sizeof`** will not work because it is based on what is known at compile time
- 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];
int i = 0;
for( 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 are 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

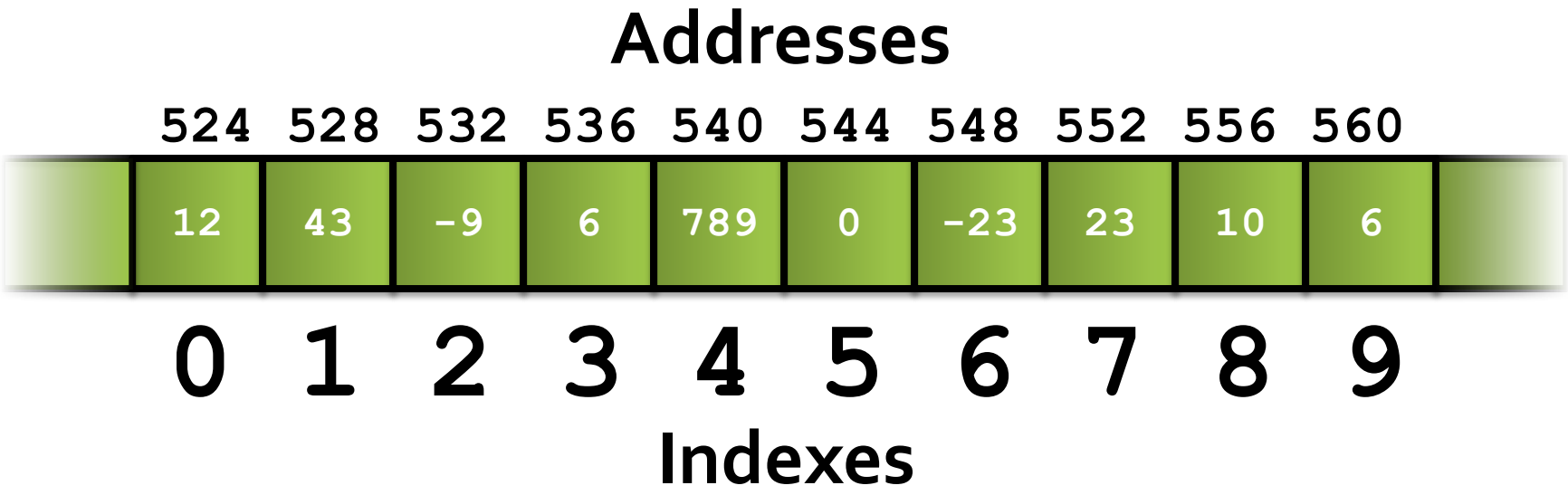
Array Memory

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])
{
  int i = 0;
  int j = 0;
  for( i = 0; i < 8; i++ )
     for( 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

Review of Compiling Multiple Files

Components

- C files
  - All the sources files that contain executable code
  - Should end with **`.c`**
- Header files
  - Files containing extern declarations and function prototypes
  - Should end with **`.h`**
- 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
```

Upcoming

Next time…

- Strings

Reminders

- Keep reading K&R chapter 5
- Keep working on Project 2
  - Due Friday
- Exam 1 next Monday