Week 15 - Monday

CS222

Last time

- What did we talk about last time?
- Templates
- STL
- Lab 14

Questions?

Project 6

Review up to Exam 1

What is Unix?

- It's a standard for operating systems based on a long, complex history with many companies and innovators
- The Open Group has the trademark on the term "UNIX," and you're only allowed to call your OS Unix if it meets their Single UNIX Specification
- Linux and FreeBSD and other free implementations of Unix do **not** meet this specification

Linux

- Linus Torvalds started working in 1991 to make a Unix kernel to run on an Intel 386
- He put Linus's Unix (Linux) under the GNU GPL
- The BSD distributions also gave rise to free BSD implementations (notably FreeBSD), but their usage is much less widespread than Linux
- Linux kernel version numbers are ***x***.***y***.***z*** where ***x*** is a major version, ***y*** is a minor version, and ***z*** is a minor revision

Types in C

- Basic types in C are similar to those in Java, but there are fewer

Type	Meaning	Size
`char`	Smallest addressable chunk of memory	Usually 1 byte
`short`	Short signed integer type	At least 2 bytes
`int`	Signed integer type	At least 2 bytes, usually 4 bytes
`long`	Long signed integer type	At least 4 bytes
`float`	Single precision floating point type	Usually 4 bytes
`double`	Double precision floating point type	Usually 8 bytes

- No built-in boolean type!

But, wait, it gets worse…

- Unlike Java, C has signed and unsigned versions of all of its integer types
  - Perhaps even worse, there's more than one way to specify their names

Type	Equivalent Types
`char`	`signed char`
`unsigned char`	

`short`	`signed short`
	`short int`
	`signed short int`

`unsigned short`	`unsigned short int`
`int`	`signed int`
`unsigned int`	`unsigned`

`long`	`signed long`
	`long int`
	`signed long int`

`unsigned long`	`unsigned long int`

Hello, World

- The standard Hello World program is simpler in C, since no external classes are needed

```
#include <stdio.h>

int main()
{
      printf("Hello, World!");
      return 0;
}
```

Sample makefile

- Makefiles are called **`makefile`** or **`Makefile`**

```
all:    hello

hello: hello.c
    gcc -o hello hello.c

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

Bases

- Know how to convert between all of the following:
  - Base 2 (binary)
  - Base 8 (octal)
  - Base 10 (decimal)
  - Base 16 (hexadecimal)

Literals

- By default, every integer is assumed to be a signed **`int`**
- If you want to mark a literal as **`long`**, put an **`L`** or an **`l`** at the end
  - **`long value = 2L;`**
  - Don't use **`l`**, it looks too much like **`1`**
  - There's no way to mark a literal as a **`short`**
- If you want to mark it unsigned, you can use a **`U`** or a **`u`**
  - **`unsigned int x = 500u;`**
- Every value with a decimal point is assumed to be double
- If you want to mark it as a **`float`**, put an **`f`** or an **`F`** at the end
  - **`float z = 1.0f;`**

Integers in other bases

- You can also write a literal in hexadecimal or octal
- A hexadecimal literal begins with **0x**
  - **`int a = 0xDEADBEEF;`**
  - Hexadecimal digits are **0** – **9** and **A** – **F** (upper or lower case)
- An octal literal begins with **0**
  - **`int b = 0765;`**
  - Octal digits are **0** – **7**
  - Be careful not to prepend other numbers with **0**, because they will be in octal!
- Remember, this changes only how you write the literal, not how it is stored in the computer
- Can't write binary literals

Binary representation

- Using a normal base 10 to base 2 conversion works fine for unsigned integer values
  - However many bits you've got, take the pattern of 1's and 0's and convert to decimal
- What about signed integers that are negative?
  - Most modern hardware (and consequently C and Java) use **two's complement** representation

Negative integer in two's complement

- Let's say you have a positive number ***n*** and want the representation of ***–n*** in two's complement with ***k*** bits

1. Figure out the pattern of ***k*** 0's and 1's for ***n***
2. Flip every single bit in that pattern (changing all 0's to 1's and all 1's to 0's)
   - This is called one's complement
3. Then, add 1 to the final representation as if it were positive, carrying the value if needed

Floating point representation

- Okay, how do we represent floating point numbers?
- A completely different system!
  - IEEE-754 standard
  - One bit is the sign bit
  - Then some bits are for the exponent (8 bits for float, 11 bits for double)
  - Then some bits are for the mantissa (23 bits for float, 52 bits for double)

More complexity

- They want floating point values to be unique
- So, the mantissa leaves off the first 1
- To allow for positive and negative exponents, you subtract 127 (for **`float`**, or 1023 for **`double`**) from the written exponent
- The final number is:
  - $(-1)^{\textit{sign bit}} \times 2^{(\textit{exponent} - 127)} \times 1.\textit{mantissa}$

Except even that isn't enough!

- How would you represent zero?
  - If all the bits are zero, the number is 0.0
- There are other special cases
  - If every bit of the exponent is set (but all of the mantissa are zeroes), the value is positive or negative infinity
  - If every bit of the exponent is set (and some of the mantissa bits are set), the value is positive or negative NaN (not a number)

Number	Representation
`0.0`	`0x00000000`
`1.0`	`0x3F800000`
`0.5`	`0x3F000000`
`3.0`	`0x40400000`
+Infinity	`0x7F800000`
-Infinity	`0xFF800000`
+NaN	`0x7FC00000` and others

One little endian

- For both integers and floating-point values, the **most significant bit** determines the sign
  - But is that bit on the rightmost side or the leftmost side?
  - What does left or right even mean inside a computer?
- The property is the **endianness** of a computer
- Some computers store the most significant bit first in the representation of a number
  - These are called **big-endian** machines
- Others store the least significant bit first
  - These are called **little-endian** machines

Math library

Function	Result	Function	Result
`cos(double theta)`	Cosine of **theta**	`exp(double x)`	e^x
`sin(double theta)`	Sine of **theta**	`log(double x)`	Natural logarithm of **x**
`tan(double theta)`	Tangent of **theta**	`log10(double x)`	Common logarithm of **x**
`acos(double x)`	Arc cosine of **x**	`pow(double base, double exponent)`	Raise **base** to power **exponent**
`asin(double x)`	Arc sine of **x**	`sqrt(double x)`	Square root of **x**
`atan(double x)`	Arc tangent of **x**	`ceil(double x)`	Round up value of x
`atan2(double y, double x)`	Arc tangent of **y/x**	`floor(double x)`	Round down value of **x**
`fabs(double x)`	Absolute value of **x**	`fmod(double value, double divisor)`	Remainder of dividing **value** by **divisor**

Preprocessor directives

- There are preprocessor directives which are technically not part of the C language
- These are processed before the real C compiler becomes involved
- The most important of these are
  - **`#include`**
  - **`#define`**
  - Conditional compilation directives

sizeof

- We said that the size of **`int`** is compiler dependent, right?
  - How do you know what it is?
- **`sizeof`** is a built-in operator that will tell you the size of an data type or variable in bytes

```
#include <stdio.h>

int main()
{
      printf("%d", sizeof(char));
      int a = 10;
      printf("%d", sizeof(a));
      double array[100];
      printf("%d", sizeof(array));

      return 0;
}
```

const

- In Java, constants are specified with the **`final`** modifier
- In C, you can use the keyword **`const`**
- Note that **`const`** is only a suggestion
  - The compiler will give you a warning if you try to assign things to **`const`** values, but there are ways you can even get around that

```
const double PI = 3.141592;
```

- Since you can dodge **`const`**, it isn't strong enough to be used for array size
- That's why **`#define`** is more prevalent

`char` values

- C uses one byte for a **`char`** value
- This means that we can represent the 128 ASCII characters without a problem
  - In many situations, you can use the full 256 extended ASCII sequence
  - In other cases, the (negative) characters will cause problems
- Beware the ASCII table!
  - Use it and die!

Bitwise operators

- Now that we have a deep understanding of how the data is stored in the computer, there are operators we can use to manipulate those representations
- These are:
  - **&** Bitwise AND
  - **|** Bitwise OR
  - **~** Bitwise NOT
  - **^** Bitwise XOR
  - **<<** Left shift
  - **>>** Right shift

Precedence

- Operators in every programming language have precedence
- Some of them are evaluated before others
  - Just like order of operations in math
- ***** and **/** have higher precedence than **+** and **–**
  - **=** has a very low precedence
- I don't expect you to memorize them all, **but**
  - Know where to look them up
  - Don't write confusing code

Precedence table

Type	Operators	Associativity
Primary Expression	`()` `[]` `.` `->` *expr*`++` *expr*`--`	Left to right
Unary	`*` `&` `+` `-` `!` `~` `++`*expr* `--`*expr* `(`*typecast*`)` `sizeof`	Right to left
Binary	`*` `/` `%`	Left to right
	`+` `-`	
	`>>` `<<`	
	`<` `>` `<=` `>=`	
	`==` `!=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary	`?:`	Right to left
Assignment	`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`	Right to left
Comma	`,`	Left to right

`if` statements

- Like Java, the body of an **`if`** statement will only execute if the condition is true
  - The condition is evaluated to an **int**
  - True means not zero

Sometimes this is natural and clear; at other times it can be cryptic.

- An **`else`** is used to mark code executed if the condition is false

Nesting

- We can nest **`if`** statements inside of other if statements, arbitrarily deep
- Just like Java, there is no such thing as an **`else if`** statement
- But, we can pretend there is because the entire **`if`** statement and the statement beneath it (and optionally a trailing **`else`**) is treated like a single statement

`switch` statements

- **`switch`** statements allow us to choose between many listed possibilities
- Execution will jump to the matching label or to **`default`** (if present) if none match
  - Labels must be constant (either literal values or **`#define`** constants)
- Execution will continue to fall through the labels until it reaches the end of the switch or hits a **`break`**
  - Don't leave out **`break`** statements unless you really mean to!

Three loops

- C has three loops, just like Java
  - **`while`** loop
    - You don't know how many times you want to run
  - **`for`** loop
    - You know how many times you want to run
  - **`do-while`** loop
    - You want to run at least once
- Like **`if`** statements, the condition for them will be evaluated to an **`int`**, which is true as long as it is non-zero
  - All loops execute as long as the condition is true

The comma operator

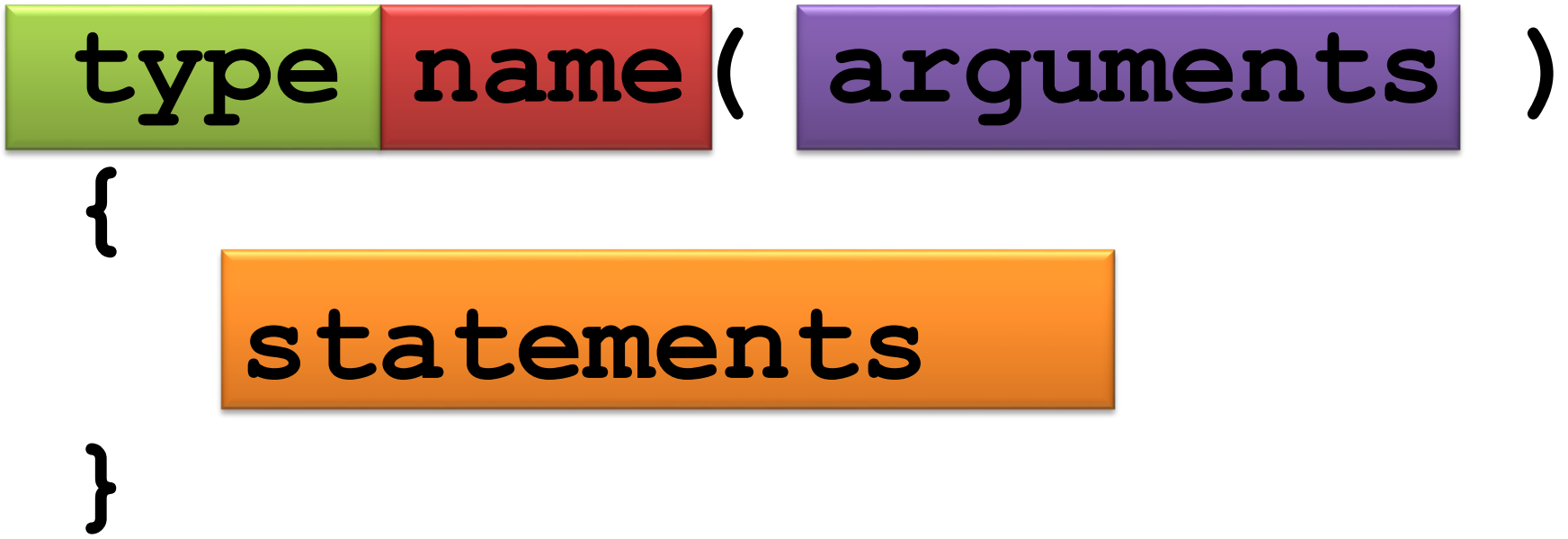
- C has a comma operator
- Expressions can be written and separated by commas
- Each will be evaluated, and the last one will give the value for the entire expression

```
int a = 10;
int b = 5;
int c = a, b, ++a, a + b; //16
```

Systems programming concepts

- Kernel
  - The part of the OS that does everything important
- Shell
  - The program you type commands into
- Users and groups
  - Users that can log in to the machines and logical groupings of them for permission purposes
- Superuser
  - The user that can do everything, often named **`root`**
- Files
  - All input and output in Unix/Linux is viewed as a file operation

Anatomy of a function definition



Differences from Java methods

- You don't have to specify a return type
  - But you **should**
  - **`int`** will be assumed if you don't
- If you start calling a function before it has been defined, it will assume it has return type **`int`** and won't bother checking its parameters

Prototypes

- Because the C language is older, its compiler processes source code in a simpler way
- It does no reasonable typechecking if a function is called before it is defined
- To have appropriate typechecking for functions, create a **prototype** for it
- Prototypes are like declarations for functions
  - They usually come in a block at the top of your source file

Return values

- C does not force you to return a value in all cases
  - The compiler may warn you, but it isn't an error
- Your function can "fall off the end"
- Sometimes it works, other times you get garbage

```
int sum(int a, int b)
{
      int result = a + b;
      return result;
}
```

```
int sum(int a, int b)
{
      int result = a + b;
}
```

Bad things

- Avoid the following constructs except when necessary:
  - **`break`**
    - Leaves loop immediately
    - Necessary for switch statements
  - **`continue`**
    - Jumps to bottom of loop immediately
- Avoid the following construct always:
  - **`goto`**

Scope

- The **scope** of a name is the part of the program where that name is visible
- In Java, scope could get complex
  - Local variables, class variables, member variables,
  - Inner classes
  - Static vs. non-static
  - Visibility issues with **`public`**, **`private`**, **`protected`**, and default
- C is simpler
  - Local variables
  - Global variables

Hiding

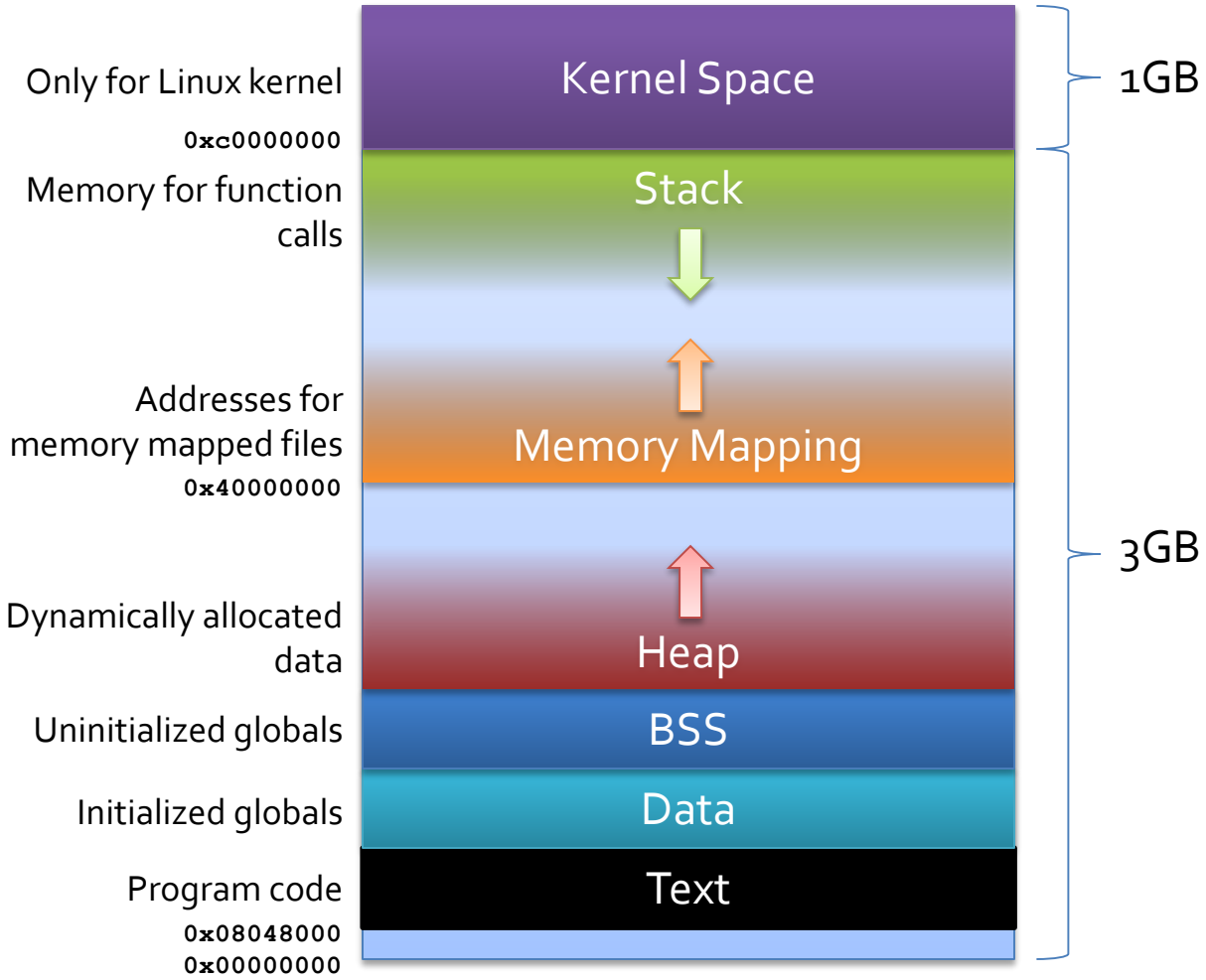
- If there are multiple variables with the same name, the one declared in the current block will be used
- If there is no such variable declared in the current block, the compiler will look outward one block at a time until it finds it
- Multiple variables can have the same name if they are declared at different scope levels
  - When an inner variable is used instead of an outer variable with the same name, it **hides** or **shadows** the outer variable
- Global variables are used only when nothing else matches
- Minimize variable hiding to avoid confusion

Processes

- A **process** is a program that is currently executing
- In memory, processes have the following segments:
  - **Text** The executable code
  - **Data** Static variables
  - **Heap** Dynamically allocated variables
  - **Stack** Area that grows and shrinks with function calls
- A **segmentation fault** is when your code tries to access a segment it's not supposed to
- A process generally executes with the same privileges as the user who started it

Process memory segments

- Layout for 32-bit architecture
  - Could only address 4GB
- Modern layouts often have random offsets for stack, heap, and memory mapping for security reasons



Compiling multiple files

- 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`**

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()`**
  - In traditional C, you must give a fixed size (literal integer or a **`#define`** constant) for the array
  - C99 supports a variable as the size, but the size is still fixed at declaration
- 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 an array uses with **`sizeof`**
  - But only in the function where it is declared!

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

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 generally 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

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**

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

Programming Practice

- Write a function that converts an **`int`** to a string representation in hexadecimal
- Write a function that converts a string representation of an integer in hexadecimal to an **`int`**

Upcoming

Next time…

- Review up to Exam 2

Reminders

- Finish Project 6
  - Due Friday by midnight