

Week 15 - Wednesday

**COMP 2400**

# Last time

- What did we talk about last time?
- Review up to Exam 2
  - Pointers
  - Dynamic memory allocation
  - Structs
  - Unions
  - Enums
  - Time

Questions?

---

# Project 6

---

# Review

---

# Final exam

- **Final exam:**
  - **Thursday, April 30, 2026**
  - **8:00 to 10:00 a.m.**
  - **50% longer than previous exams, but you'll have 100% more time**
- There will be multiple choice, short answer, and programming questions

# Practice

- Write a function that finds the median of an array
  - You'll have to sort it
- Write a function that, given a string, creates a dynamically allocated chunk of memory containing the string reversed
- Write a function that will delete an element from the singly linked list struct given in earlier slides
- Write a program that counts the **total** number of characters in all the arguments passed in through the command line
  - Ignore `argv[0]`
- Write a program to "encrypt" a file by writing a new file with exactly the same contents, except that each byte in the file is inverted
  - Old byte:  $x$
  - New byte:  $255 - x$

# File I/O

---

# Files

- Think of a file as a stream of bytes
- It is possible to read from the stream
- It is possible to write to the stream
- It is even possible to do both
- Central to the idea of a stream is also a file stream pointer, which keeps track of where in the stream you are
- We have been redirecting **stdin** from and **stdout** to files, but we can access them directly as well

# fopen ()

- To open a file, call the **fopen ()** function
- It returns a pointer to a **FILE** object
- Its first argument is the path to the file as a null-terminated string
- Its second argument is another string that says how it is being opened (for reading, writing, etc.)

```
FILE* file = fopen("data.txt", "r");
```

# fopen () arguments

- The following are legal arguments for the second string

Argument	Meaning
"r"	Open for reading. The file must exist.
"w"	Open for writing. If the file exists, all its contents will be erased.
"a"	Open for appending. Write all data to the end of the file, preserving anything that is already there.
"r+"	Open a file for reading and writing, but it must exist.
"w+"	Open a file for reading and writing, but if it exists, its contents will be erased.
"a+"	Open a file for reading and writing, but all writing is done to the end of the file.

# fprintf()

- Once you've got a file open, write to it using **fprintf()** the same way you write to the screen with **printf()**
- The first argument is the file pointer
- The second is the format string
- The third and subsequent arguments are the values

```
FILE* file = fopen("output.dat", "w");  
fprintf(file, "Yo! I got %d on it!\n", 5);
```

# fscanf()

- Once you've got a file open, write to it using **fscanf()** the same way you write to the screen with **scanf()**
- The first argument is the file pointer
- The second is the format string
- The third and subsequent arguments are pointers to the values you want to read into

```
FILE* file = fopen("input.dat", "r");  
int value = 0;  
fscanf(file, "%d", &value);
```

# Closing files

- When you're done using a file, close the file pointer using the `fclose()` function
- Files will automatically be closed when your program ends
- It's a good idea to close them as soon as you don't need them anymore
  - It takes up system resources
  - You can only have a limited number of files open at once

```
FILE* file = fopen("input.dat", "r");  
int value = 0;  
fscanf(file, "%d", &value);  
fclose(file);
```

# fputc () and putc ()

- If you need to do character by character output, you can use **fputc ()**
- The first is the character to output
- The second argument is the file pointer
- **putc ()** is an equivalent function

```
FILE* file = fopen("output.dat", "w");  
for( int i = 0; i < 100; i++ )  
    fputc('$', file);
```

# fgetc () and getc ()

- If you need to do character by character input, you can use **fgetc ()**
- The argument is the file pointer
- It returns the character value or **EOF** if there's nothing left in the file
- **getc ()** is an equivalent function

```
FILE* file = fopen("input.dat", "r");  
int count = 0;
```

```
while( fgetc(file) != EOF )  
    count++;
```

```
printf("There are %d characters in the file\n", count);
```

# Standard streams

- C programs that run on the command line have the following file pointers open by default
  - `stdin`
  - `stdout`
  - `stderr`
- You can use them where you would use other file pointers

# What is a binary file?

- Technically, **all** files are binary files
  - They all carry data stored in binary
- But some of those binary files are called **text files** because they are filled with human readable text
- When most people talk about binary files, they mean files with data that is only computer readable

# Why use binary files?

- Wouldn't it be easier to use all human readable files?
- Binary files can be more efficient
  - In binary, all `int` values are the same size, usually 4 bytes
- You can also load a chunk of memory (like a WAV header) into memory with one function call

Integer	Bytes in text representation
0	1
92	2
789	3
4551	4
10890999	8
204471262	9
-2000000000	11

# Changes to `fopen()`

- To specify that a file should be opened in binary mode, append a **b** to the mode string

```
FILE* file = fopen("output.dat", "wb");
```

```
FILE* file = fopen("input.dat", "rb");
```

- On some systems, the **b** has no effect
- On others, it changes how some characters are interpreted

# fread()

- The **fread()** function allows you to read binary data from a file and drop it directly into memory
- It takes
  - A pointer to the memory you want to fill
  - The size of each element
  - The number of elements
  - The file pointer

```
double data[100];  
FILE* file = fopen("input.dat", "rb");  
fread(data, sizeof(double), 100, file);  
fclose(file);
```

# fwrite()

- The **fwrite()** function allows for binary writing
- It can drop an arbitrarily large chunk of data into memory at once
- It takes
  - A pointer to the memory you want to write
  - The size of each element
  - The number of elements
  - The file pointer

```
short values[50];  
FILE* file = NULL;  
//fill values with data  
file = fopen("output.dat", "wb");  
fwrite(values, sizeof(short), 50, file);  
fclose(file);
```

# fseek ()

- The **fseek ()** function takes
  - The file pointer
  - The offset to move the stream pointer (positive or negative)
  - The location the offset is relative to
- Legal locations are
  - **SEEK\_SET**        From the beginning of the file
  - **SEEK\_CUR**        From the current location
  - **SEEK\_END**        From the end of the file (not always supported)

```
FILE* file = fopen("input.dat", "rb");  
int offset;  
fread(&offset, sizeof(int), 1, file); //get offset  
fseek(file, offset, SEEK_SET);
```

# Partition layout

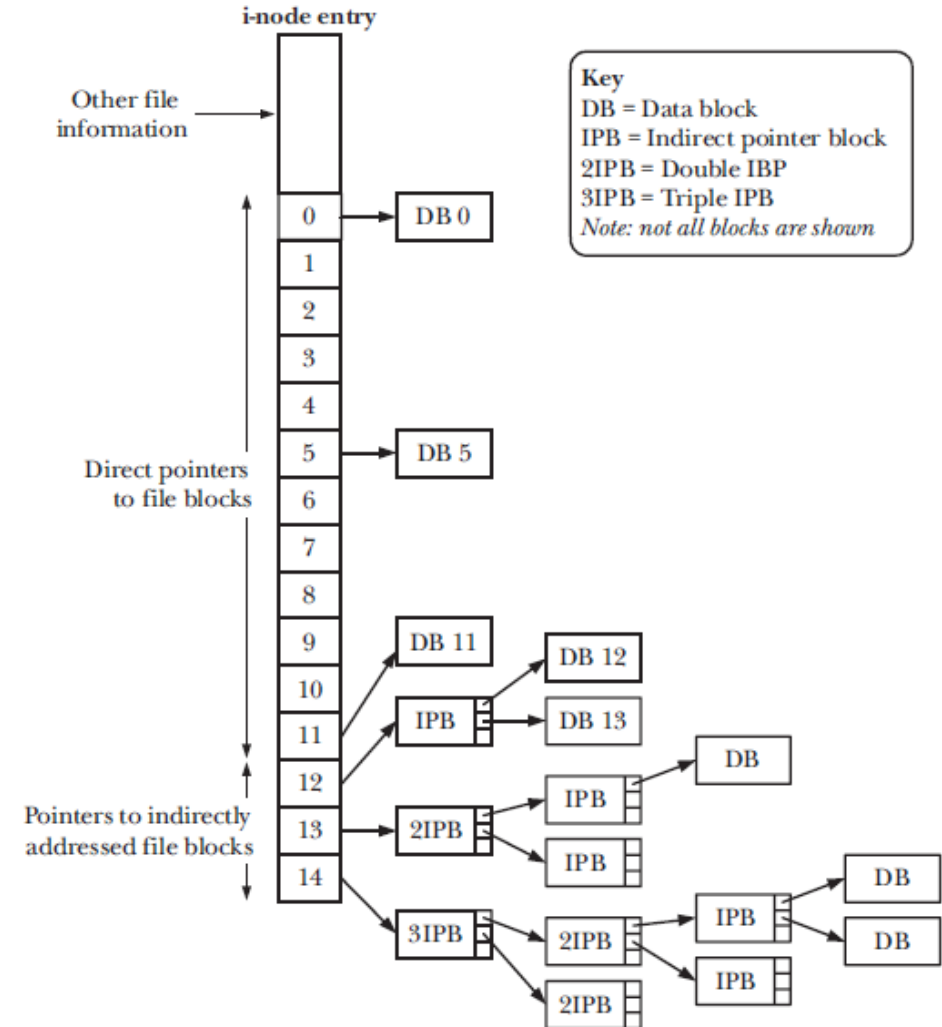
- Virtually all file systems have each partition laid out something like this



- The boot block is the first block and has information needed to boot the OS
- The superblock has information about the size of the i-node table and logical blocks
- The i-node table has entries for every file in the system
- Data blocks are the actual data in the files and take up almost all the space

# i-nodes

- Every file has an i-node in the i-node table
- Each i-node has information about the file like type (directory or not), owner, group, permissions, and size
- More importantly, each i-node has pointers to the data blocks of the file on disk
- In ext2, i-nodes have 15 pointers
  - The first 12 point to blocks of data
  - The next points to a block of pointers to blocks of data
  - The next points to a block of pointers to pointers to blocks of data
  - The last points to a block of pointers to pointers to pointers to blocks of data



# Networking

---

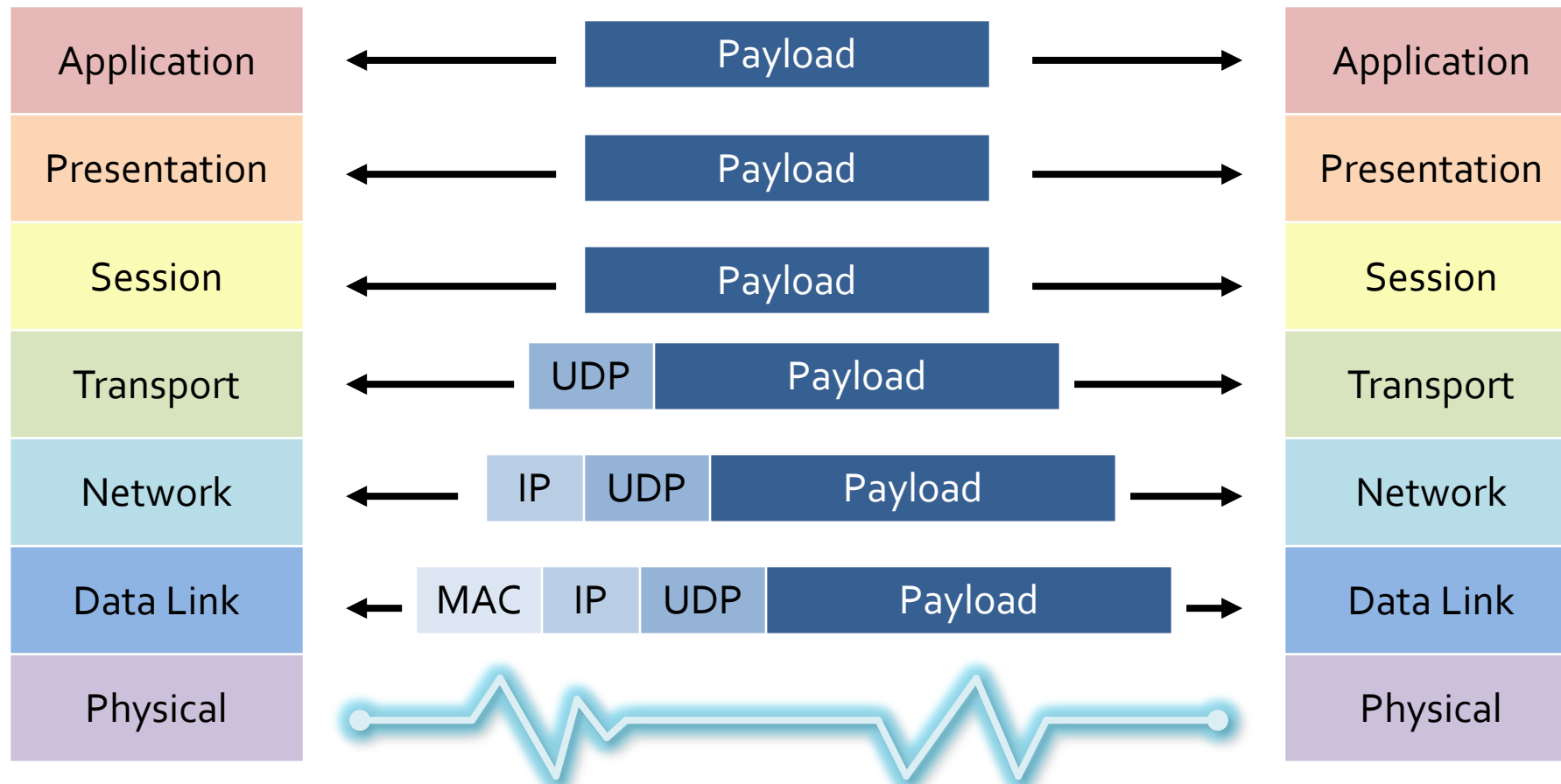
# Layers

- Not every layer is always used
- Sometimes user errors are referred to as Layer 8 problems

Layer	Name	Mnemonic	Activity	Example
7	Application	Away	User-level data	HTTP
6	Presentation	Pretzels	Data appearance, some encryption	TLS
5	Session	Salty	Sessions, sequencing, recovery	IPC and part of TCP
4	Transport	Throw	Flow control, end-to-end error detection	TCP
3	Network	Not	Routing, blocking into packets	IP
2	Data Link	Dare	Data delivery, packets into frames, transmission error recovery	Ethernet
1	Physical	Programmers	Physical communication, bit transmission	Electrons in copper

# Transparency

- The goal of the OSI model is to make lower layers transparent to upper ones



# TCP/IP

- The OSI model is sort of a sham
  - It was invented after the Internet was already in use
  - You don't need all layers
  - Some people think this categorization is not useful
- Most network communication uses TCP/IP
- We can view TCP/IP as five layers:

Layer	Action	Responsibilities	Protocols
Application	Prepare messages	User interaction	HTTP, FTP, etc.
Transport	Convert messages to segments	Sequencing, reliability, error correction	TCP or UDP
Internet	Convert segments to packets	Flow control, routing	IP
Link	Convert packets to frames	Point-to-point communication between devices on the same network	Ethernet, Wi-Fi
Physical	Transmit frames as bits	Data communication	

# TCP/IP

- A TCP/IP connection between two hosts (computers) is defined by four things
  - Source IP
  - Source port
  - Destination IP
  - Destination port
- One machine can be connected to many other machines, but the port numbers keep it straight

# Sockets

---

# Sockets

- **Sockets** are the most basic way to send data over a network in C
- A socket is **one end** of a two-way communication link between two programs
  - Just like you can plug a phone into a socket in your wall (if you are living in 1980)
  - Both programs have to have a socket
  - And those sockets have to be connected to each other
- Sockets can be used to communicate within a computer, but we'll focus on Internet sockets



# socket ()

- If you want to create a socket, you can call the **socket ()** function
- The function takes a communication domain
  - Will always be **AF\_INET** for IPv4 Internet communication
- It takes a type
  - **SOCK\_STREAM** usually means TCP
  - **SOCK\_DGRAM** usually means UDP
- It takes a protocol
  - Which will always be **0** for us
- It returns a file descriptor (an **int**)

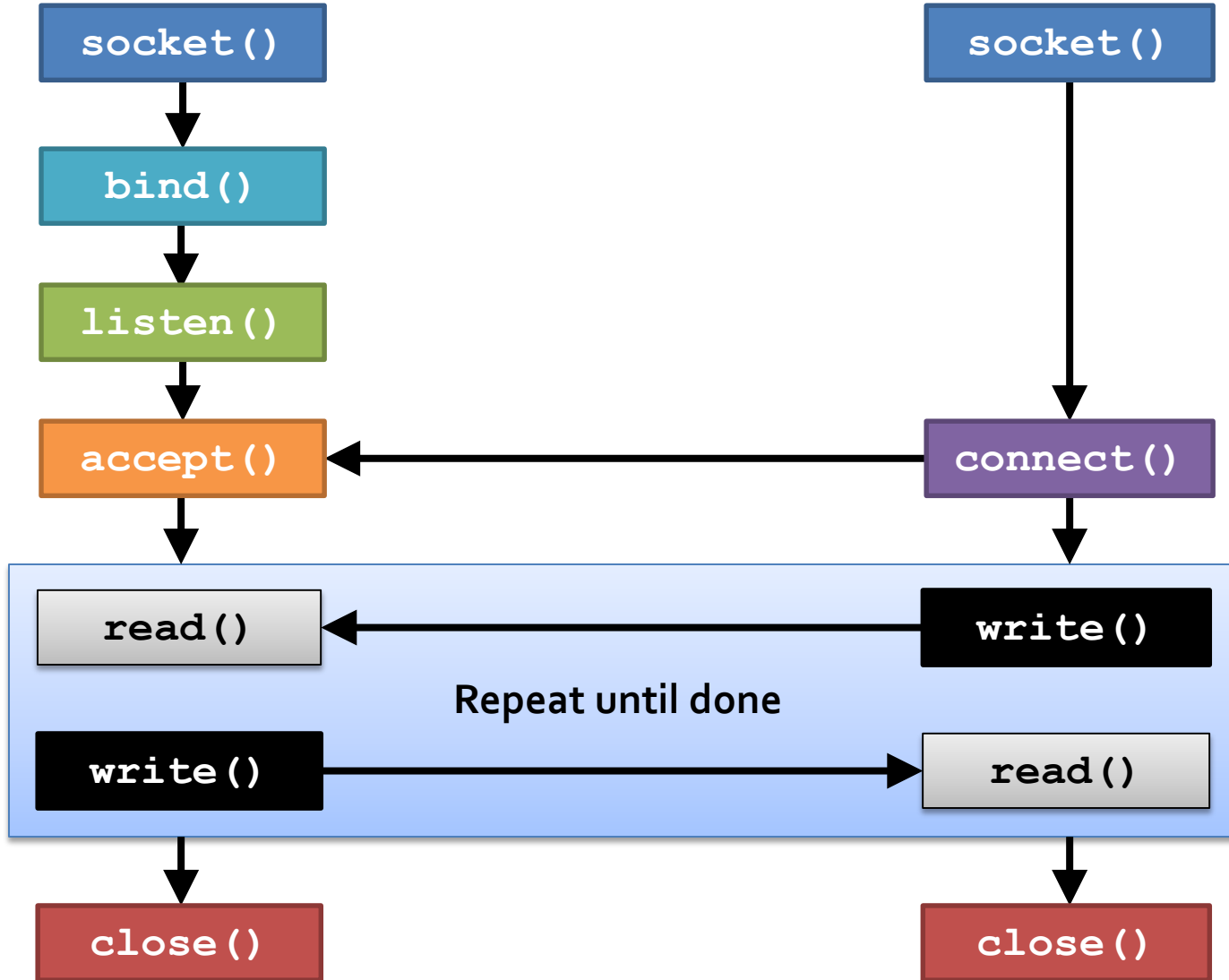
```
int sockFD = -1;  
sockFD = socket(AF_INET, SOCK_STREAM, 0);
```

# Clients vs. servers

- Using sockets is usually associated with a client-server model
- A **server** is a process that sits around waiting for a connection
  - When it gets one, it can do sends and receives
- A **client** is a process that connects to a waiting server
  - Then it can do sends and receives
- Clients and servers are processes, not computers
  - You can have many client and server processes on a single machine

# Server

# Client



# Sending

- Once you've created your socket, set up your port and address, and called **connect ()**, you can send data
  - Assuming there were no errors
  - Sending is very similar to writing to a file
- The **write ()** function takes
  - The socket file descriptor
  - A pointer to the data you want to send
  - The number of bytes you want to send
  - Just like writing to a file
- It returns the number of bytes sent

```
char* message = "Flip mode is the squad!";  
write(socketFD, message, strlen(message)+1);
```

# Receiving

- Or, once you're connected, you can also receive data
  - Receiving is very similar to reading from a file
- The **read()** function takes
  - The socket file descriptor
  - A pointer to the data you want to receive
  - The size of your buffer
  - Just like reading from a file
- It returns the number of bytes received, or **0** if the connection is closed, or **-1** if there was an error

```
char message[100];  
read(socketFD, message, 100);
```

# Servers

- Sending and receiving are the same on servers, but setting up the socket is more complex
- Steps:
  1. Create a socket in the same way as a client
  2. Bind the socket to a port
  3. Set up the socket to listen for incoming connections
  4. Accept a connection

# Function pointers

- C can have pointers to functions
- You can call a function if you have a pointer to it
- You can store these function pointers in arrays and structs
- They can be passed as parameters and returned as values
- Java doesn't have function pointers
  - Instead, you pass around objects that have methods you want
  - C# has delegates, which are similar to function pointers

# Declaring a function pointer

- The syntax is a bit ugly
- Pretend like it's a prototype for a function
  - Except take the name, put a **\*** in front, and surround that with parentheses

```
#include <math.h>
#include <stdio.h>

int main()
{
    double (*root) (double); //pointer named root
    root = &sqrt; //note there are no parentheses
    printf( "Root 3 is %lf", root(3) );
    printf( "Root 3 is %lf", (*root)(3) ); //also legal

    return 0;
}
```

C++

---

# Overview

- C++ is based on C and easier to use
  - You can declare variables anywhere
    - Not the case in the C89 standard (where all variables had to be declared right after a block starts), but our `gcc` is following the C99 standard
  - It has function overloading
  - Most people think the I/O is cleaner
- The big addition is OOP through classes
- It's an approximate superset of C that includes most C structures

# Hello, World in C++

- It's not too different from C
- We need different headers for C++ I/O

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
    return 0;
}
```

# Output in C++

- Output uses the **cout** object (of type **ostream**)
- Instead of using formatting strings, **cout** uses the idea of a stream, where objects are placed into the stream separated by the extraction operator **<<**
- The **endl** object adds a newline to the stream
  - Of course, "**\n**" works too

```
int x = 50;
```

```
cout << "There are " << x << " ways to leave your lover."  
<< endl;
```

# Input in C++

- Input uses the **cin** object (of type **istream**)
- **cin** also uses the idea of a stream, where items are read from the stream and separated by the insertion operator **>>**
- It reads items using whitespace as the separator, just like **scanf()**

```
int x = 0;
int y = 0;
int z = 0;
cout << "Enter the x, y, and z values: ";
cin >> x >> y >> z;
```

# The `string` class

- Like Java, C++ has a class for holding strings, which makes life much easier
  - It's called `string` (with a lower case 's')
- You must include `<string>` to use it
- Unlike `String` in Java, `string` is mutable
  - You can use array-style indexing to get and set individual characters

```
string a = "Can I kick it?";
string b = "Yes, you can!";
string c = a + " " + b;
c[0] = 'D';
c[1] = 'i';
c[2] = 'd';
cout << c << endl; // Prints Did I kick it? Yes, you can!
```

# The `std` namespace

- Java uses packages to keep different classes with the same name straight
- C++ uses namespaces
- The standard library includes I/O (`<iostream>`), the string class (`<string>`), STL containers (`<vector>`, `<list>`, `<deque>`, and others)
- If you use these in your program, put the following after your includes

```
using namespace std;
```

- The alternative is to specify the namespace by putting the it followed by two colons before the class name

```
std::string name = "Ghostface Killah";
```

# Functions in C++

- Regular C++ functions are very similar to functions in C
- A big difference is that prototypes are no longer optional if you want to call the function before it's defined
- Unlike C, function overloading allowed:

```
int max(int a, int b)
{
    return a > b ? a : b;
}

int max(int a, int b, int c)
{
    return max( a, max( b, c) );
}
```

# Pass by reference

- In C, all functions are pass by value
  - If you want to change an argument, you have to pass a pointer to the value
- In C++, you can specify that a parameter is pass by reference
  - Changes to it are seen on the outside
  - You do this by putting an ampersand (&) before the variable name in the header

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

# Default parameter values

- C++ also allows you to specify default values for function parameters
- If you call a function and leave off those parameters, the default values will be used
- Default parameters are only allowed for the rightmost grouping of parameters

```
void build(int width = 2, int height = 4)
{
    cout << "We built this house with " << width <<
    " by " << height << "s.";
}
```

```
build();           //We built this house with 2 by 4s.
build(3);         //We built this house with 3 by 4s.
build(6, 8);      //We built this house with 6 by 8s.
```

# The new keyword

- When you want to dynamically allocate memory in C++, you use **new** (instead of **malloc()**)
  - No cast needed
  - It "feels" a lot like Java

```
int* value = new int(); // Make an int
int* array = new int[100]; // Array of ints
Wombat* wombat = new Wombat(); // Make a Wombat
// Makes 100 Wombats with the default constructor
Wombat* zoo = new Wombat[100];
```

# The `delete` keyword

- When you want to free dynamically allocated memory in C++, use **`delete`** (instead of **`free()`**)
  - If an array was allocated, you have to use **`delete[]`**

```
int* value = new int(); // Make just one int
delete value;
```

```
Wombat* wombat = new Wombat();
delete wombat;
```

```
Wombat* zoo = new Wombat[100];
delete[] zoo; // Array delete needed
```

# Object Oriented Programming

- C++ has several classically important elements of OOP:
  - Encapsulation
  - Dynamic dispatch
  - Polymorphism
  - Inheritance
  - Self-reference

# Overloading operators

- In C++, you can **overload operators**, meaning that you can define what + means when used with classes you design
- Thus, the following *could* be legal:

```
Hippopotamus hippo;  
Sandwich club;  
Vampire dracula = club + hippo;
```

# Templates

- Allow classes and functions to be written with a generic type or value parameter, then instantiated later
- Each necessary instantiation is generated at compile time
- Appears to function like generics in Java, but works very differently under the covers
- Most of the time you will **use** templates, not create them

# Upcoming

---

# Next time...

---

- There is no next time!

# Reminders

- **Finish Project 6**
  - **Due tonight by midnight!**
- **Final exam:**
  - **Thursday, April 30, 2026**
  - **8:00 to 10:00 a.m.**