CSE 333
Section 7

Inheritance and Networks

UNIVERSITY of WASHINGTON

Logistics

- **HW3** due tonight (!!) **8/7 at 11:00pm**
- **Exercise 15** due on **Monday (8/11) @ 10:00am**
- **Midterm revisions during office hours this week!**
    - Please make an appointment by email!

Inheritance

Inheritance

- Motivation: Better modularize our code for similar classes!

- The public interface of a derived class inherits all **non-private** member variables and functions (**except** for `ctor`, `cctor`, `dtor`, `op=`) from its base class
  - *Similar to*: A subclass inherits from a superclass

- Aside: We will be only using **public, single** inheritance in CSE 333

Polymorphism: Dynamic Dispatch

- **Polymorphism** allows for you to access objects of related types (base and derived classes) – Allows interface usage instead of class implementation

- **Dynamic dispatch**: Implementation is determined *at* **runtime** via lookup
  - Allows you to call the **most-derived** version of the actual type of an object
  - Generally want to use this when you have a derived class

- `virtual` replaces the class's default **static dispatch** with **dynamic dispatch**

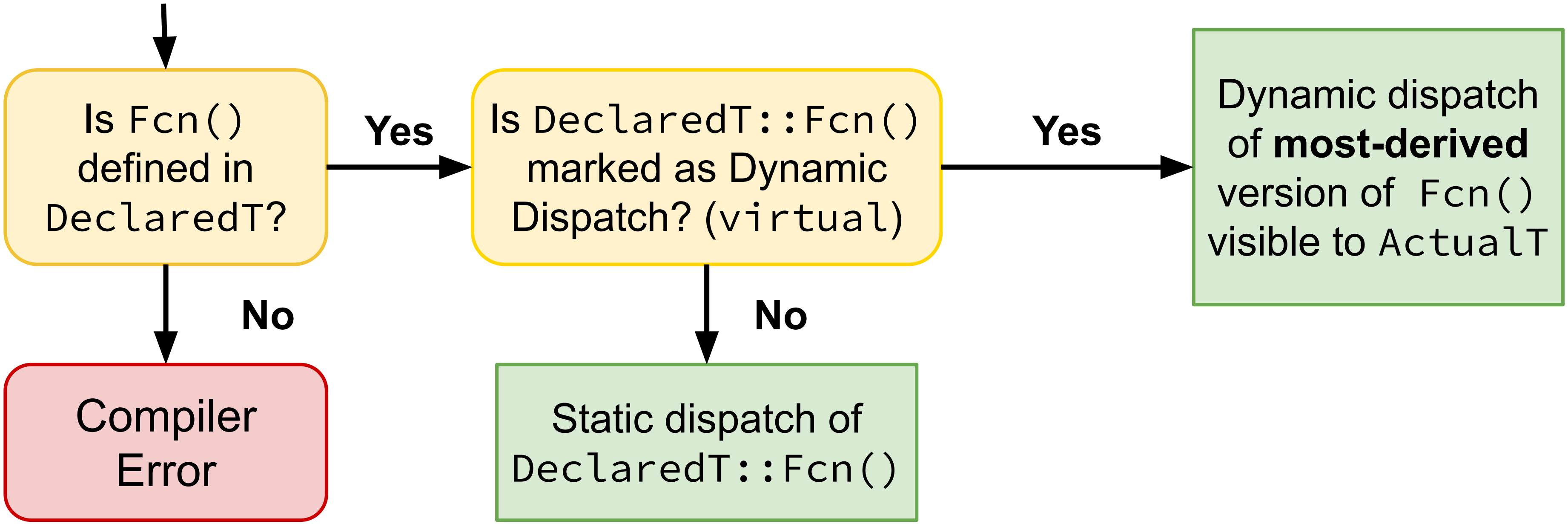
Dynamic Dispatch: Style Considerations

- Defining Dynamic Dispatch in your code base
  - Use `virtual` **only once** when first defined in the base class
    - (although in older code bases you may see it repeated on functions in subclasses)
  - All derived classes of a base class should use `override` to get the compiler to check that a function overrides a virtual function from a base class

- Use `virtual` for destructors of a base class – Guarantees all derived classes will use dynamic dispatch to ensure use of appropriate destructors

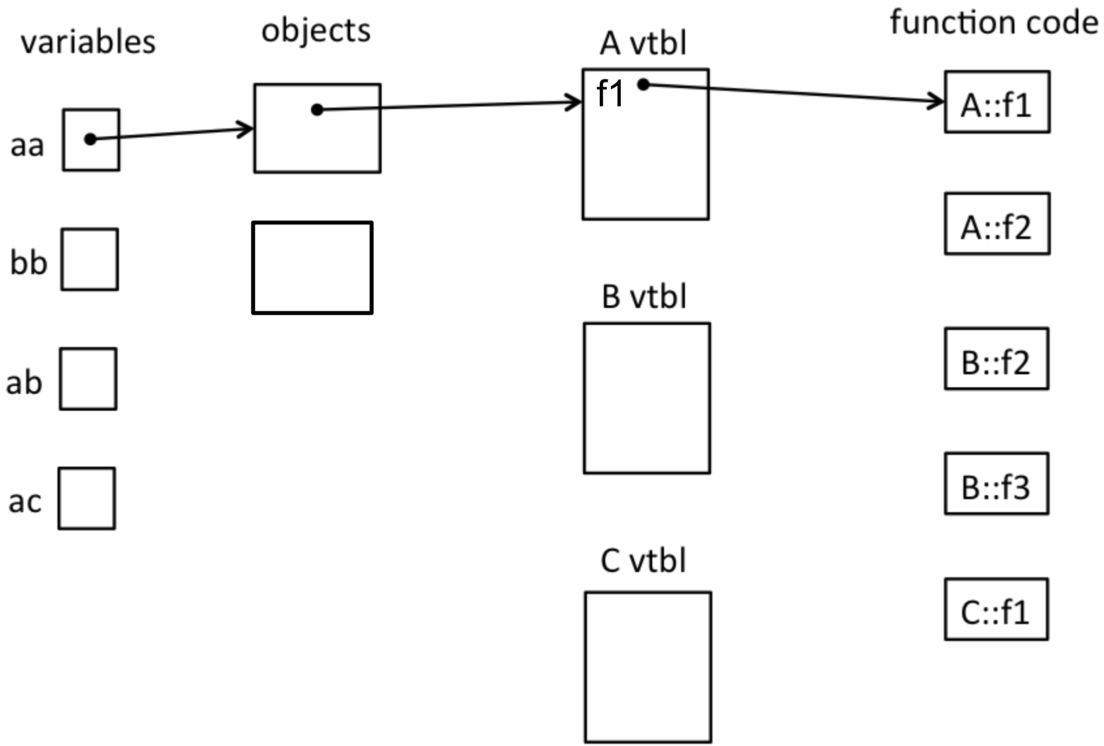
Dispatch Decision Tree

```
DeclaredT* ptr = new ActualT();
ptr->Fcn();  // which version is called?
```



Exercise 1

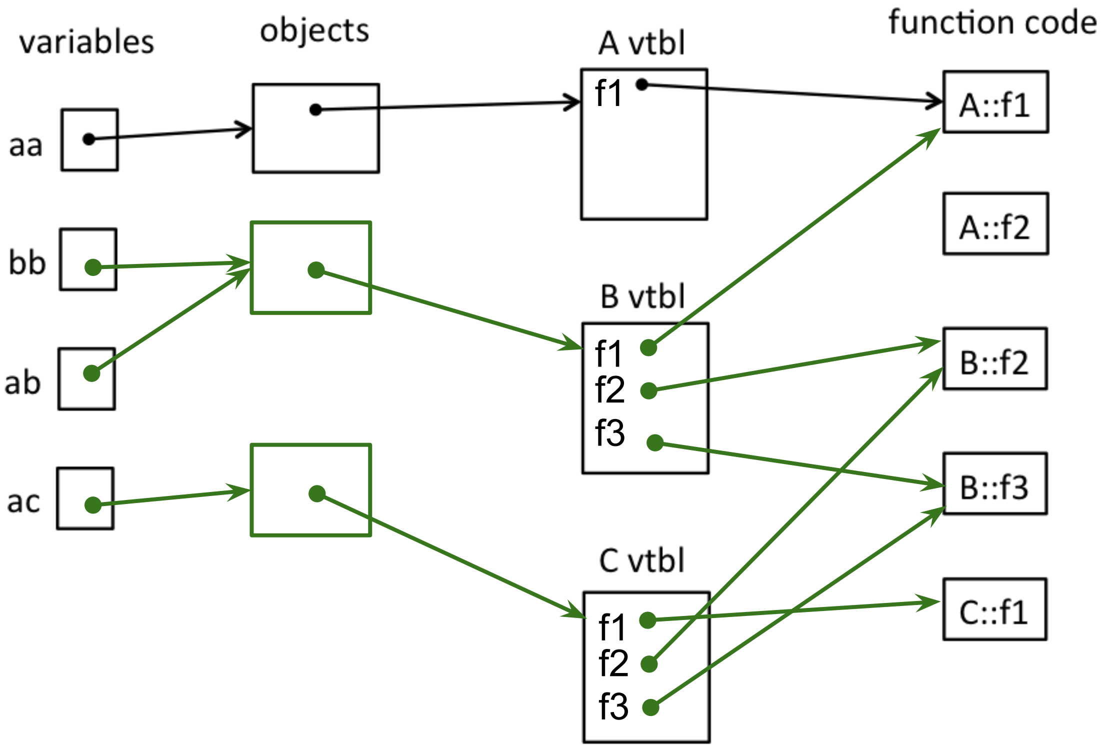
Exercise 1 (Drawing vtable diagram)



Exercise 1 Solution (pointers)

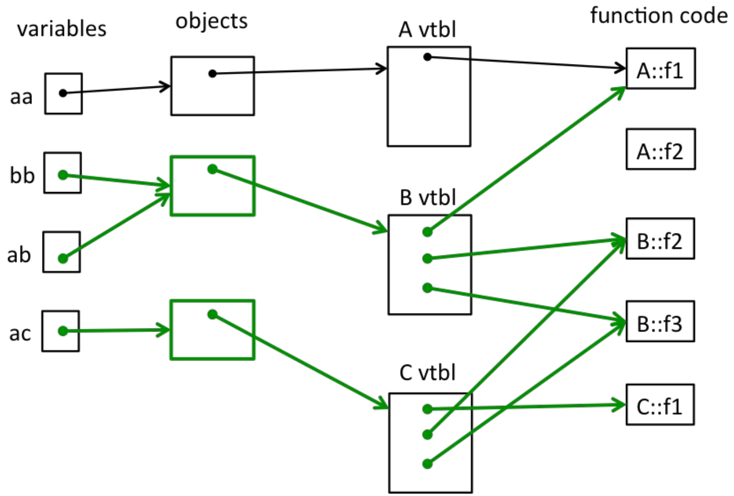
```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

```
int main() {
  A* aa = new A();
  B* bb = new B();
  A* ab = bb;
  A* ac = new C();
```



Exercise 1 Solution (output)

```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

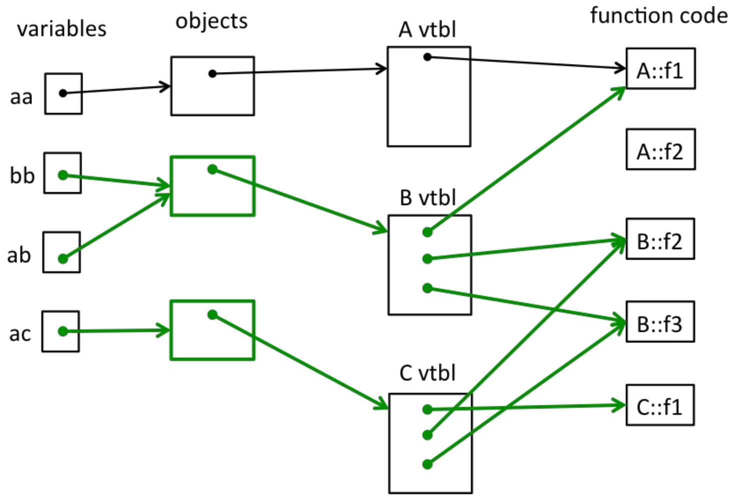



```
A* aa = new A();

aa->f1();
```

A	B	C	D
B::f2 A::f1	A::f2 C::f1	A::f2 A::f1	B::f2 C::f1

Exercise 1 Solution (output)

```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

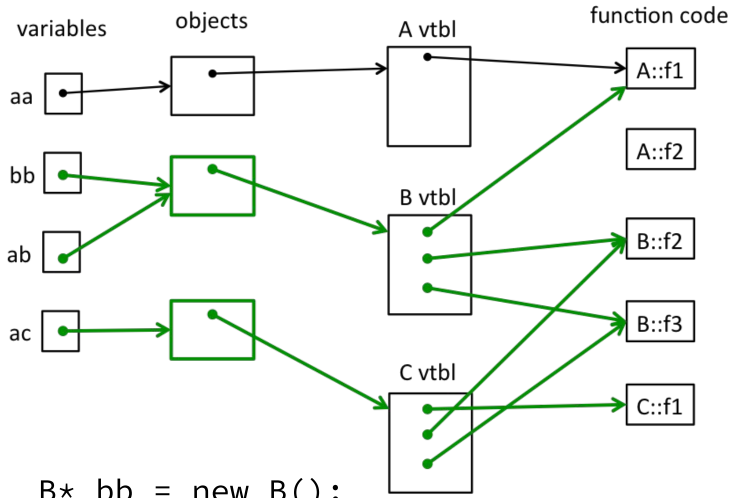



```
B* bb = new B();

bb->f1();
```

A	B	C	D
B::f2 A::f1	A::f2 C::f1	A::f2 A::f1	B::f2 C::f1

Exercise 1 Solution (output)

```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

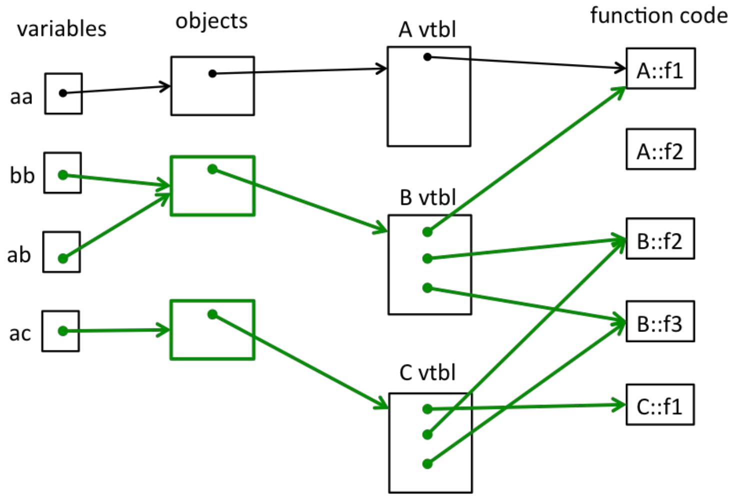



```
B* bb = new B();
A* ab = bb;

bb->f2();
cout << "----" << endl;
ab->f2();
```

A	B	C	D
B::f2 ---- B::f2	A::f2 ---- B::f2	B::f2 ---- A::f2	A::f2 ---- A::f2

Exercise 1 Solution (output)

```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

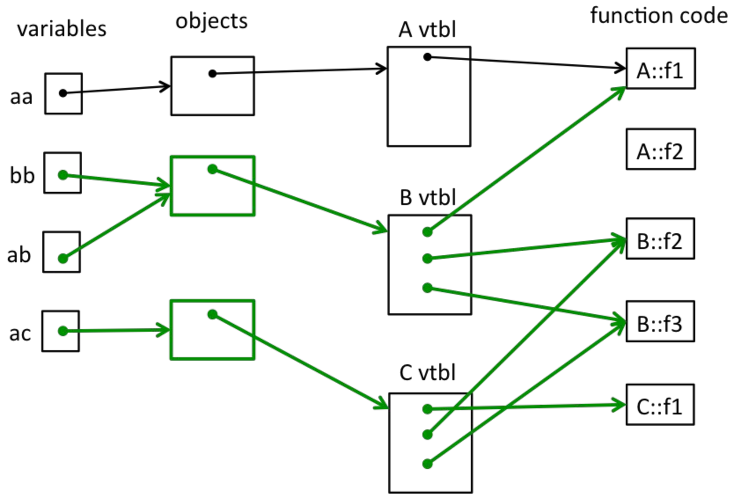



```
B* bb = new B();

bb->f3();
```

A	B	C	D
B::f2 A::f1 B::f3	A::f2 A::f1 B::f3	A::f2 C::f1 B::f3	B::f2 C::f1 B::f3

Exercise 1 Solution (output)

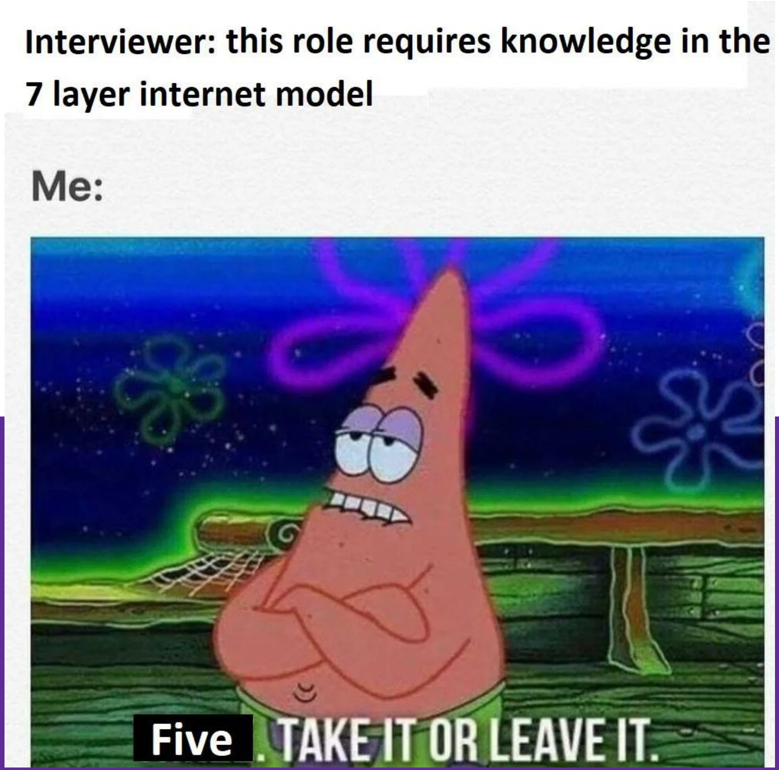
```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```



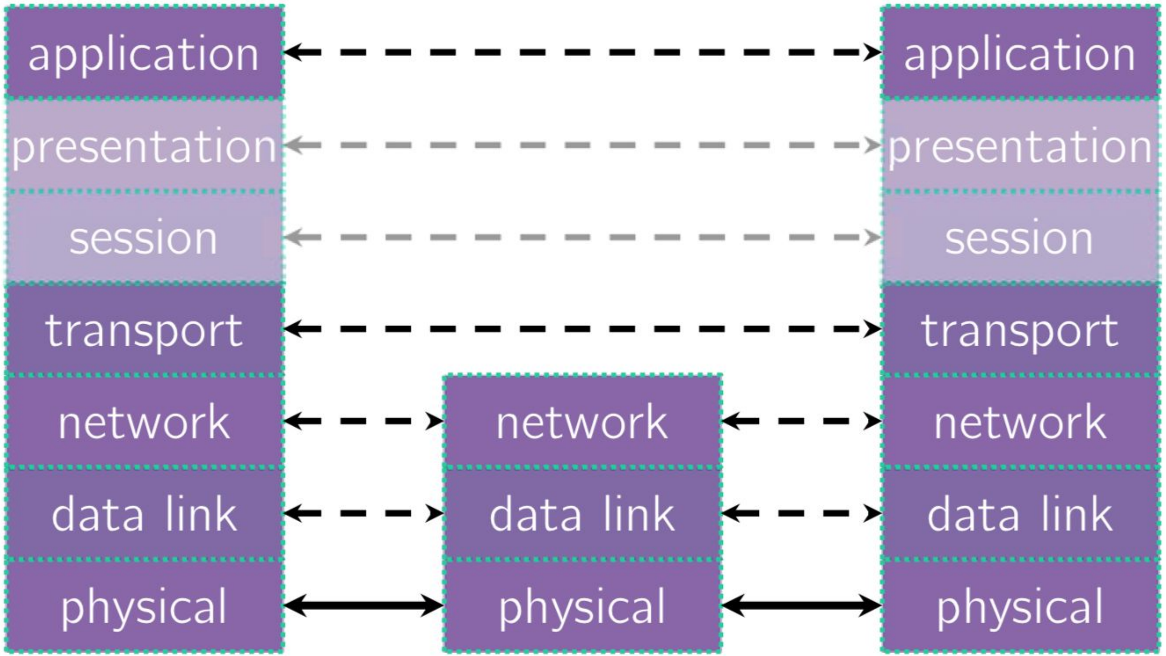

```
A* ac = new C();

ac->f1();
```

A	B	C	D
B::f2 A::f1	A::f2 C::f1	A::f2 A::f1	B::f2 C::f1

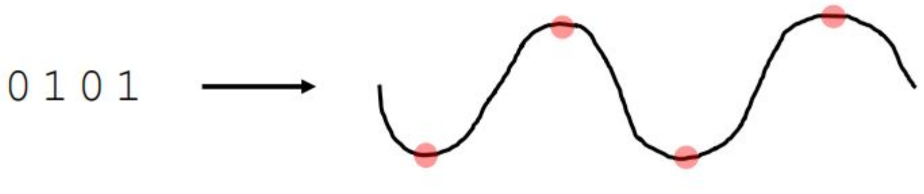
Computer Networking - At a High Level



Computer Networks: A 7-ish Layer Cake



Computer Networks: A 7-ish Layer Cake



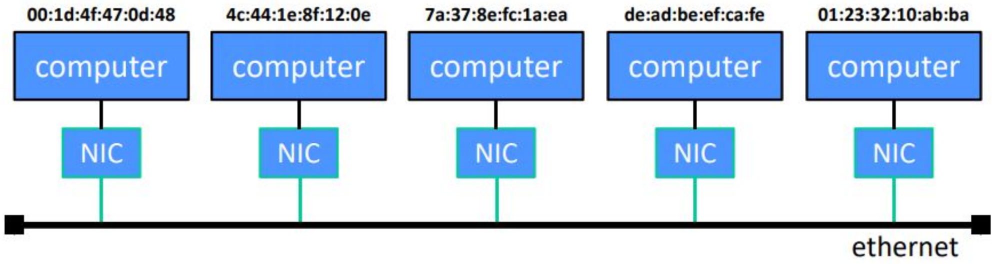
Wires, radio signals, fiber optics

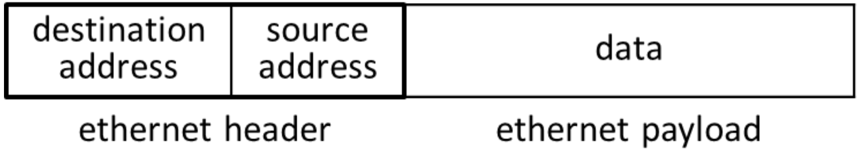
bit encoding at signal level

Computer Networks: A 7-ish Layer Cake

WiFi, ethernet.
Connecting multiple computers

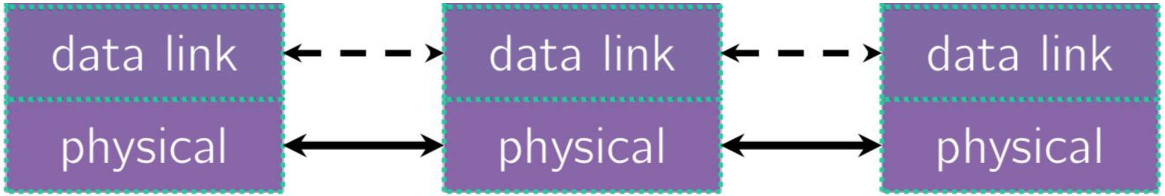
LAN



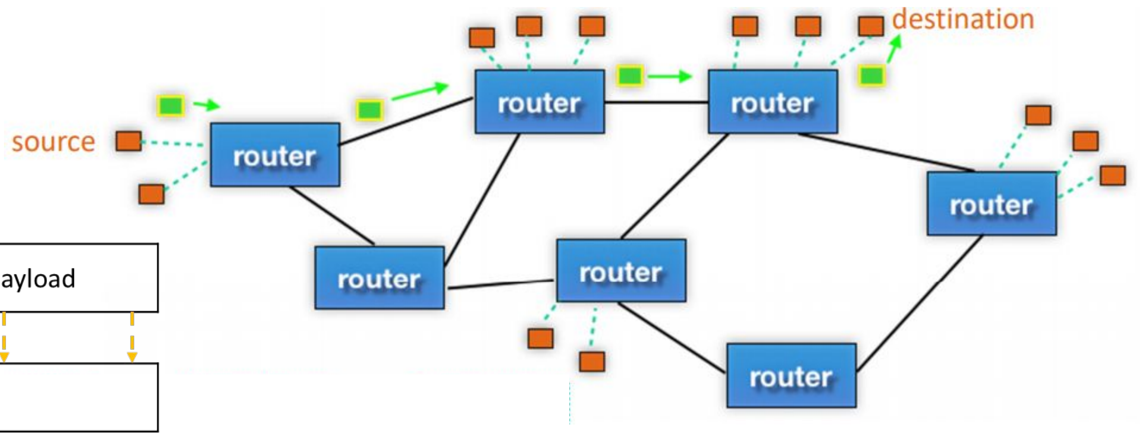


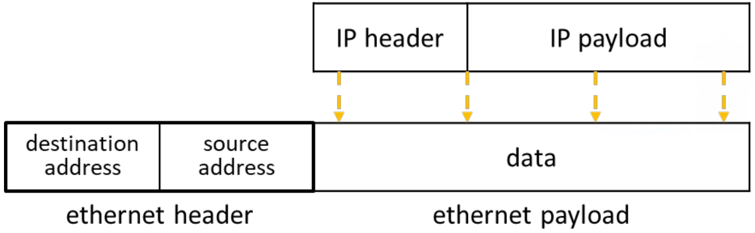
multiple computers on a local network

bit encoding at signal level



Computer Networks: A 7-ish Layer Cake

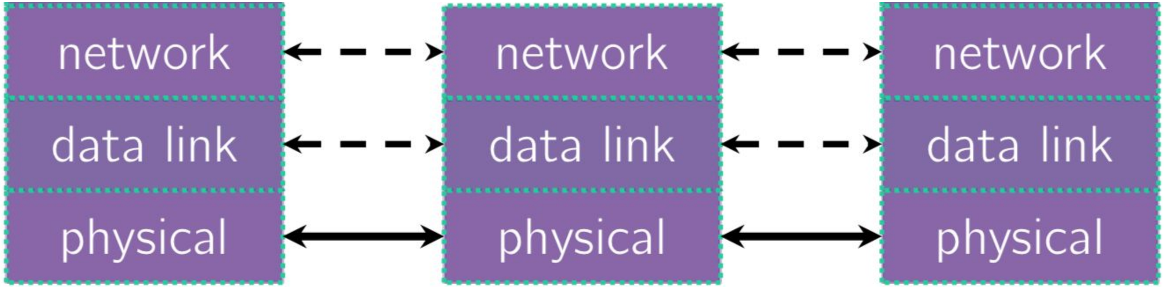




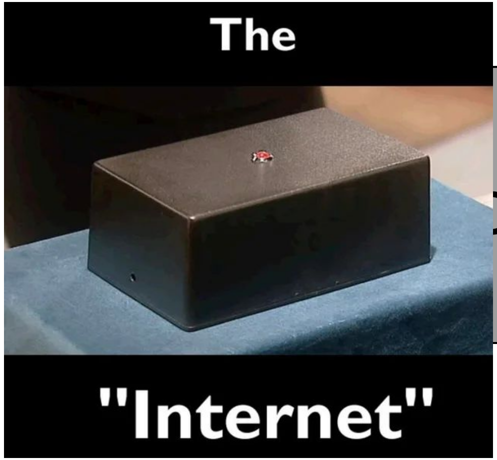
routing of packets across networks

multiple computers on a local network

bit encoding at signal level

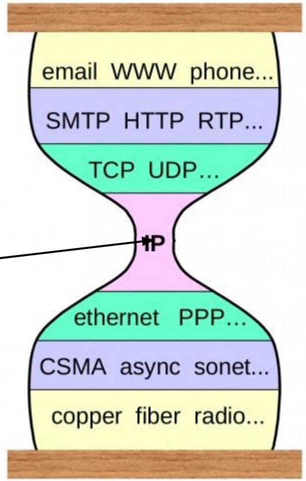


Computer Networks: A 7-ish Layer Cake



on/Interface

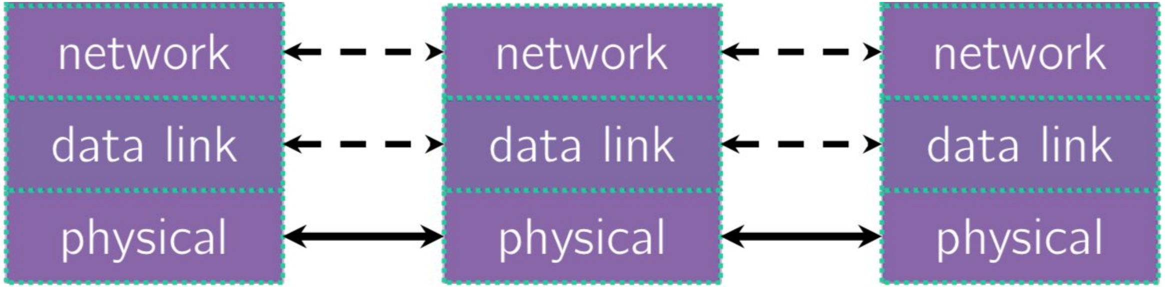
Backbone of the Internet!



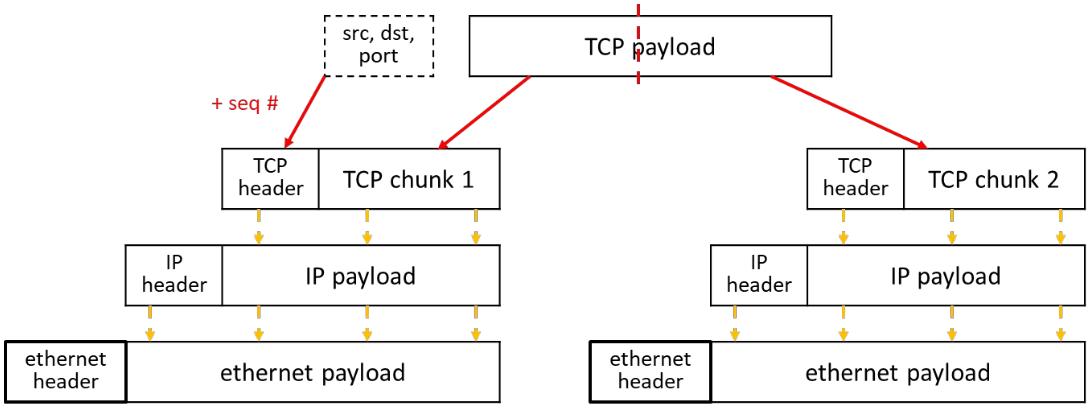
routing of packets across networks

multiple computers on a local network

bit encoding at signal level



Computer Networks: A 7-ish Layer Cake



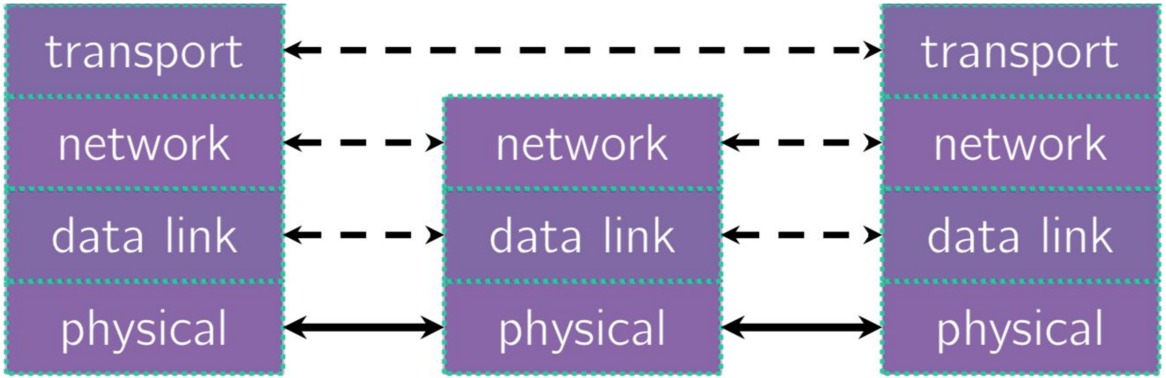
TCP, UDP, etc.

sending data end-to-end

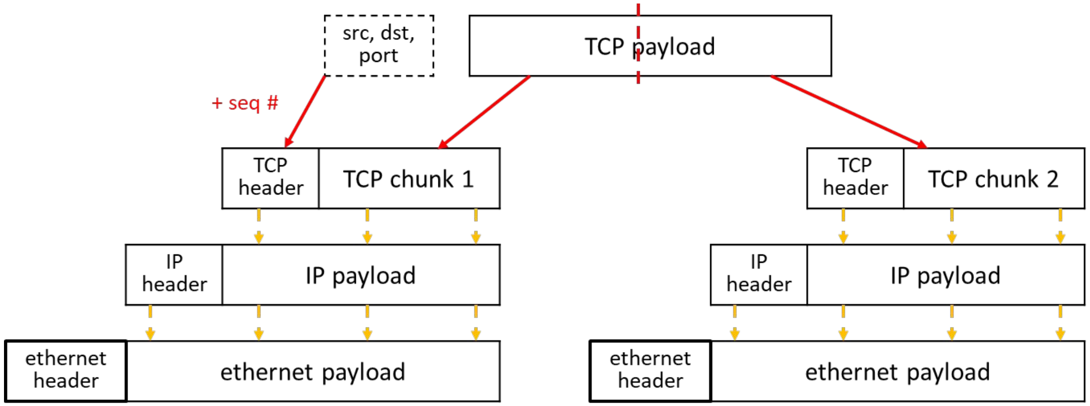
routing of packets across networks

multiple computers on a local network

bit encoding at signal level



Computer Networks: A 7-ish Layer Cake



TCP, UDP, etc.

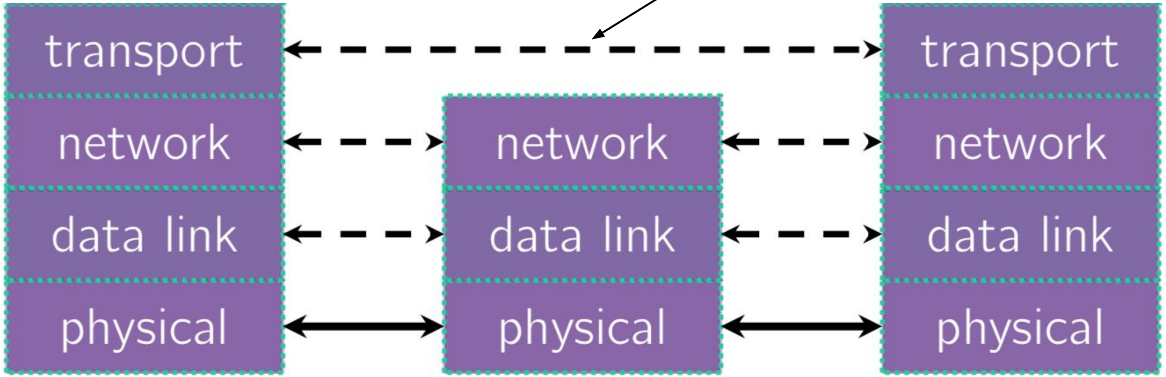
Stream abstraction!

sending data end-to-end

routing of packets across networks

multiple computers on a local network

bit encoding at signal level



Computer Networks: A 7-ish Layer Cake

HTTP, DNS, much more

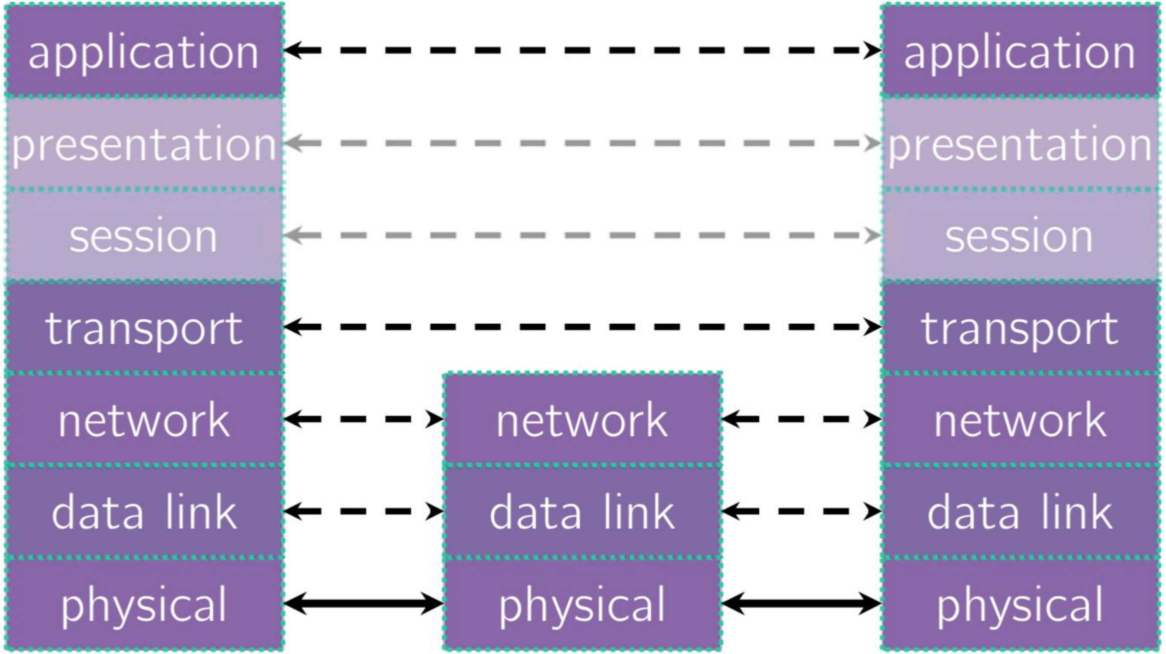
format/meaning of messages

sending data end-to-end

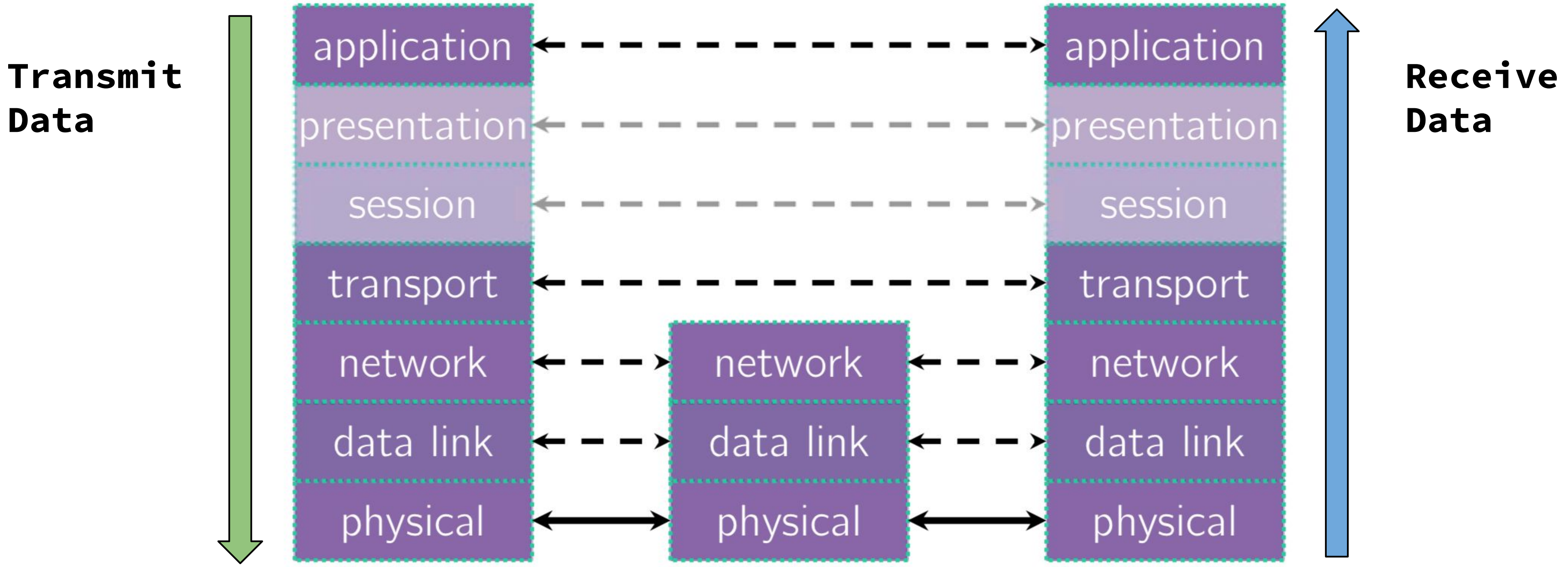
routing of packets across networks

multiple computers on a local network

bit encoding at signal level



Data Flow


Transmit
Data
application
presentation
session
transport
network
data link
physical
network
data link
physical
application
presentation
session
transport
network
data link
physical
Receive
Data

Exercise 3

Exercise 3

- **(Application Layer)** DNS:
- **(Network Layer)** IP:
- **(Transport Layer)** TCP:
- **(Transport Layer)** UDP:
- **(Application Layer)** HTTP:

Reliable transport protocol on top of IP.

Translating between IP addresses and host names.

Sending websites and data over the Internet.

Unreliable transport protocol on top of IP.

Routing packets across the Internet.

TCP versus UDP

Transmission Control Protocol (TCP):

- Connection-oriented service
- Reliable and Ordered
- Flow control

User Datagram Protocol (UDP):

- “Connectionless” service
- Unreliable packet delivery
- High speed, no feedback

TCP guarantees reliability for things like messaging or data transfers. UDP has less overhead since it doesn’t make those guarantees, but is often fine for streaming applications (e.g., YouTube or Netflix) or other applications that manage packets on their own or do not want occasional pauses for packet retransmission or recovery.

Client-Side Networking

Client-Side Networking in 5 Easy* Steps!

1. Figure out what IP address and port to talk to
2. Build a socket from the client
3. Connect to the server using the client socket and server socket
4. Read and/or write using the socket
5. Close the socket connection

Remember these are POSIX operations called using glibc C functions, though we are using them in our C++ programs

*difficulty is subjective

Sockets (Berkeley Sockets)

- Just a file descriptor for network communication
  - Defines a local endpoint for network communication
  - Built on various operating system calls

- Types of Sockets
  - Stream sockets (TCP)
  - Datagram sockets (UDP)
  - There are other types, which we will not discuss

- Each TCP socket is associated with **a TCP port number (`uint16_t`)** and **an IP address**
  - These are in network order (not host order) in TCP/IP data structures! (https://www.gnu.org/software/libc/manual/html_node/Byte-Order.html)
  - `ai_family` will help you to determine what is stored for your socket!

Understanding Socket Addresses

struct sockaddr (pointer to this struct is used as parameter type in system calls)

fam	????

....

struct sockaddr_in (IPv4)

fam	**port**	**addr**	zero

16

struct sockaddr_in6 (IPv6)

fam	**port**	flow	**addr**	scope

28

struct sockaddr_storage

fam	????

Big enough to hold either

Understanding `struct sockaddr*`

- It's just a pointer. To use it, we're going to have to dereference it and cast it to the right type (Very strange C "inheritance")
  - It is the endpoint your connection refers to

- Convert to a `struct sockaddr_storage`
  - Read the `sa_family` to determine whether it is IPv4 or IPv6
  - IPv4: `AF_INET` (macro) → cast to `struct sockaddr_in`
  - IPv6: `AF_INET6` (macro) → cast to `struct sockaddr_in6`

Step 1: Figuring out the port and IP

- Performs a **DNS Lookup** for a hostname
- Use “hints” to specify constraints (`struct addrinfo*`)
- Get back a linked list of `struct addrinfo` results

Name of host whose IP we want

```
int getaddrinfo(const char* hostname,
                const char* service,
                const struct addrinfo* hints,
                struct addrinfo** res);
```

We will set this to `nullptr` to get the default; otherwise you can specify service/port

Output parameter; `*res` is set to the first result in LL

Hints for the lookup server/refine results

Step 1: Obtaining your server's socket address

```
struct addrinfo {
    int ai_flags;                   // additional flags
    int ai_family;                  // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype;                // SOCK_STREAM, SOCK_DGRAM, 0
    int ai_protocol;                // IPPROTO_TCP, IPPROTO_UDP, 0
    size_t ai_addrlen;              // length of socket addr in bytes
    struct sockaddr* ai_addr;       // pointer to socket addr
    char* ai_canonname;             // canonical name
    struct addrinfo* ai_next;       // can have linked list of records
}
```

- `ai_addr` points to a `struct sockaddr` describing a socket address, can be IPv4 or IPv6

Steps 2 and 3: Building a Connection

2. Create a client socket to manage (returns an integer file descriptor, just like POSIX open)

```
// returns file descriptor on success, -1 on failure (errno set)
int socket(int domain,              // AF_INET, AF_INET6, etc.
           int type,                // SOCK_STREAM, SOCK_DGRAM, etc.
           int protocol);           // just put 0 (network abstraction)
```

3. Use that created client socket to connect to the server socket

```
// Connects to the server
// returns 0 on success, -1 on failure (errno set)
int connect(int sockfd,                      // socket file descriptor
            struct sockaddr* serv_addr,      // socket addr of server
            socklen_t addrlen);              // size of serv_addr
```

Usually from `getaddrinfo`!

Steps 4 and 5: Using your Connection

```
// returns amount read, 0 for EOF, -1 on failure (errno set)
ssize_t read(int fd, void* buf, size_t count);

// returns amount written, -1 on failure (errno set)
ssize_t write(int fd, void* buf, size_t count);

// returns 0 for success, -1 on failure (errno set)
int close(int fd);
```

- Same POSIX methods we used for file I/O!
  (so they require the same error checking...)

Helpful References

1. Figure out what IP address and port to talk to
    - dnsresolve.cc
2. Build a socket from the client
    - connect.cc
3. Connect to the server using the client socket and server socket
    - sendreceive.cc
4. Read and/or write using the socket
    - sendreceive.cc (same as above)
5. Close the socket connection