UNIVERSITY of WASHINGTON

LECTURE 16 · CSE 333 · Summer 2026

Intro to Networking

Discussion: Favorite tongue twister?

Reminders

Assessments:

- Homework 3 due next Thursday at 11:59pm
- Exercise 7 released sometime today.

General:

- Guest lecture on Monday, August 3rd
  - Works on Google Cloud (Data center, C++ client libraries, GPU team)
  - **Definitely show up!**
- Midterm is *mostly* graded. Will be released over weekend.
  - Statistics not including personalied:

Median	Maximum	Mean	Std Dev
78.33%	100.0%	74.52%	18.83%

Review

The standard library, in three pieces

Containers: hold your data

Own their elements by value: **vector** (array), **list** (linked), **map** (tree).

Examples

```
1  vector<int> v;
2  list<string> lst;
3  map<string,int> m;
```

Iterators: the bridge

A **pointer-like cursor** that lets any algorithm walk any container.

Examples

```
1  auto b = v.begin();
2  auto e = v.end();
3  sort(b, e);
```

Algorithms: operate on data

Free functions that **transform or query** a sequence of elements.

Examples

```
1  sort( ... );
2  find( ..., x );
3  for_each( ..., f );
```

Iterators manage pointers to a container and provide an interface for the same algorithms to run on many containers.

Example #1: sort

```
1  std::vector<std::string> names = {"Bob", "Alice", "Eve", "Dan"};
2
3  std::sort(names.begin(), names.end());
4  // names: Alice  Bob  Dan  Eve
5
6  std::sort(names.begin(), names.end(), std::greater<std::string>());
7  // names: Eve  Dan  Bob  Alice
```

Example #2: count frequencies

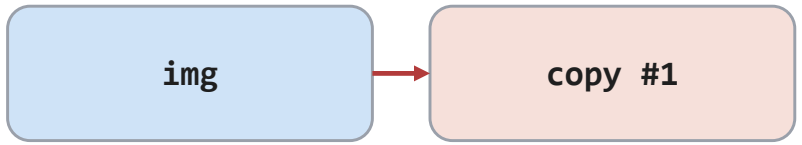
```
1  std::vector<std::string> words = {"red", "blue", "red"};
2  std::map<std::string, int> freq;
3
4  for (const std::string& w : words) {
5    freq[w]++;          // [] inserts 0 the first time by calling default constructor
6  }
7  // freq: {blue: 1, red: 2}   (kept sorted by key)
```

Smart Pointers

Containers store by value

```
1  vector<Image> gallery;
2  gallery.push_back(img);    // COPY #1 into vector
3  gallery.push_back(img2);   // may realloc ->
4                             // COPIES all again
```

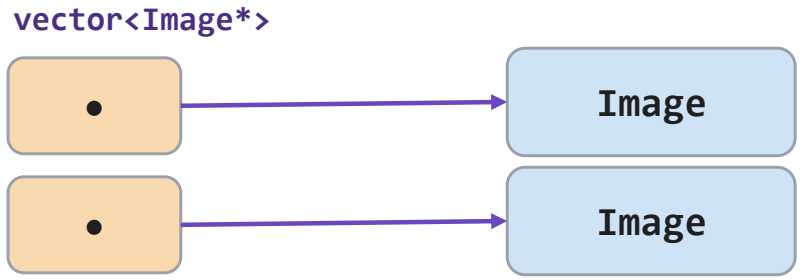
push_back copies INTO the container:



A reallocation copies **every** element again. For big objects (Images, long strings) that is noticeable.

So we store pointers instead

```
1  vector<Image*> gallery;
2  gallery.push_back(new Image("a.png"));  // store
3  gallery.push_back(new Image("b.png"));  // a ptr
4  // copying the vector copies ADDRESSES only
```

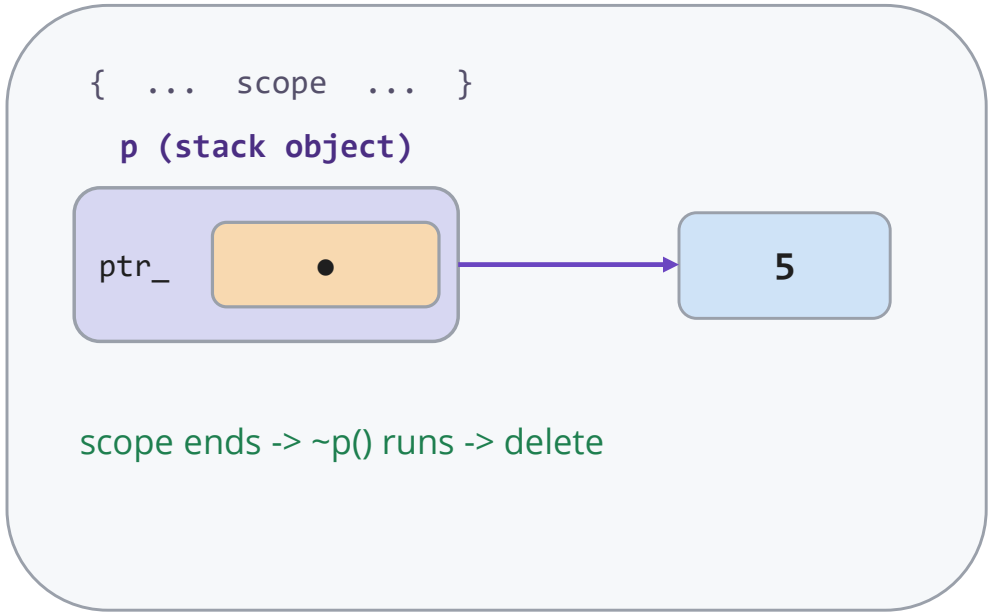


copying the vector copies addresses,
not pixels - cheap!

Let a local object own it

- **One thing in C++ is guaranteed**:
  - when a local object leaves scope, its **destructor runs** on function returns AND on exceptions.
- So wrap the raw pointer in a small **stack object** whose:
  - **constructor** takes the pointer, and
  - **destructor** deletes it - automatically, on every path.
- This pattern has a name: **RAII**

  (resource allocation is initialization)



ToyPtr with operator overloading

```
1  template <typename T> class ToyPtr {
2   public:
3    ToyPtr(T* p) : ptr_(p) { }
4    ~ToyPtr() { delete ptr_; }
5    T& operator*()  const { return *ptr_; }  // *p
6    T* operator->() const { return ptr_; }   // p->x
7
8   private:
9    T* ptr_;
10 };
```

operator* and operator->

- ❖ **operator*** replaces **deref()**, so ***p** reads/writes.
- ❖ **operator->** replaces **get()**, so **p->field** works.
- ❖ Same ownership, **nicer syntax.**
- ❖ This is how real smart pointers **feel like pointers.**

Using *p and p->

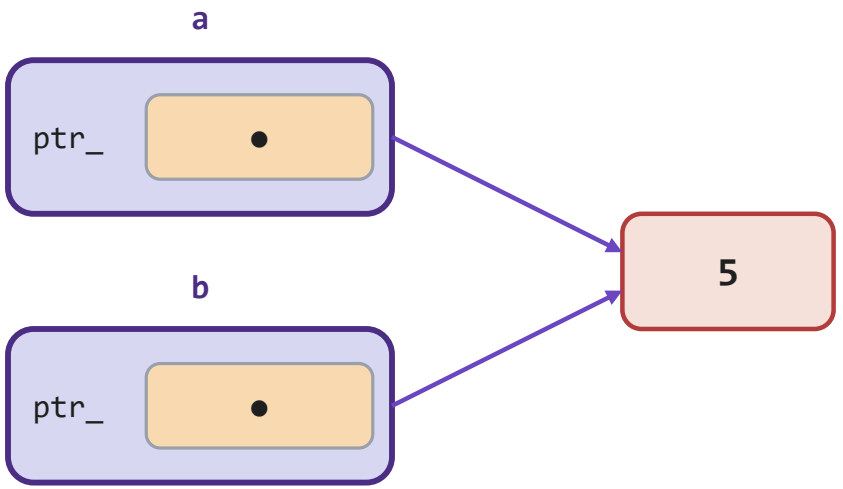
```
1  struct Node { int val; };
2
3  {
4    ToyPtr<Node> p(new Node{5});
5    cout << (*p).val;     // operator*
6    (*p).val = 10;        // write through
7    cout << p->val;       // operator-> : cleaner
8  }   // p leaves scope -> Node deleted
```

Reads like a raw pointer

- ❖ ***p** and **p->val** are exactly raw-pointer syntax.
- ❖ But when **p** leaves scope, the Node is freed for you.
- ❖ Same behavior as the method version, **without the noise.**
- ❖ That is the whole idea of a smart pointer, in ~10 lines.

The copy bug

```
1  ToyPtr<int> a(new int(5));
2  ToyPtr<int> b = a;   // shallow copy: same ptr_!
3
4  // scope ends:
5  //   ~b -> delete ptr_;
6  //   ~a -> delete ptr_;   // DOUBLE FREE
```



both destructors delete the SAME int

What should copying mean?

Copying an owner of one heap object must do WHAT?

Answer A: forbid the copy

- There is exactly **one owner**. Copying is banned.
- You may **transfer** ownership (move), not duplicate it.
- **Zero overhead.** This is **unique_ptr**.

Answer B: share and count

- Copying is fine: **many owners share one object.**
- Keep a **reference count**; free when it hits 0.
- **Small overhead.** This is **shared_ptr**.

unique_ptr: exactly one owner

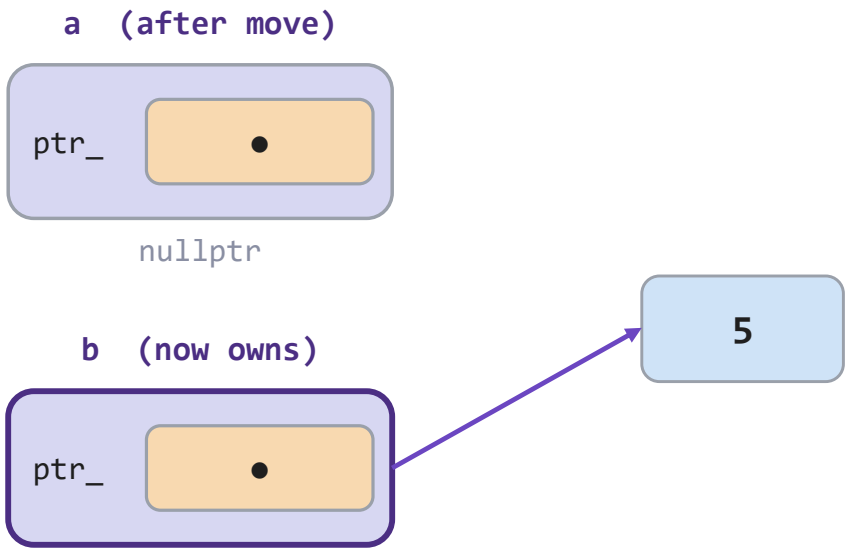
```
1  #include <memory>
2  using namespace std;
3
4  std::unique_ptr<int> a(new int(5));
5  std::unique_ptr<int> b = a;   // COMPILE ERROR: no copy
6  std::cout << *a << std::endl;      // a is the sole owner
7  // preferred:
8  auto c = std::make_unique<int>(5);
```

Sole ownership

- **#include <memory>**, type **unique_ptr<T>**.
- **Copy and assign are** deleted - the toy's bug cannot compile.
- **Frees in its destructor,** zero overhead vs a raw pointer.
- **make_unique<T>(...)** is the preferred way to create one.

Moving a unique_ptr

```
1  std::unique_ptr<int> a(new int(5));
2
3  std::unique_ptr<int> b = std::move(a);  // TRANSFER ownership
4  // a is now nullptr; b owns the 5
5
6  a.reset();           // free early (a already null)
7  int* raw = b.release();  // give up ownership (rare)
```



vector<unique_ptr<Image>>

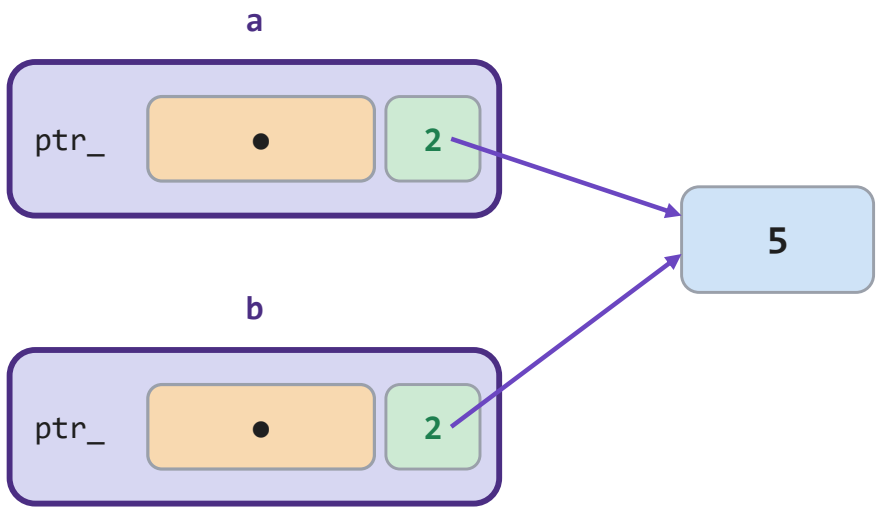
```
1  vector<unique_ptr<Image>> gallery;
2  gallery.push_back(std::make_unique<Image>("a.png"));
3  gallery.push_back(std::make_unique<Image>("b.png"));
4
5  // container MOVES pointers, never copies an Image;
6  // when gallery dies, every Image is deleted.
```

Container owns the objects

- Store **vector<unique_ptr<Image>>**.
- push_back(**move**(p)) - the container MOVES pointers, never copies Images.
- **When the vector dies,** each unique_ptr's destructor deletes its Image.
- **The leak from before** is now impossible.

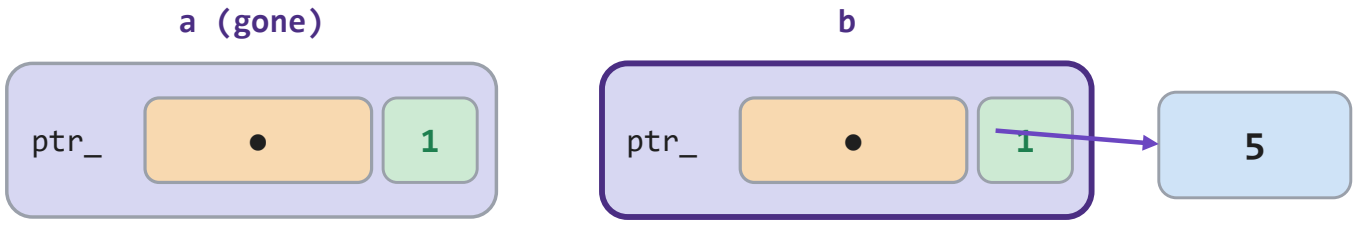
shared_ptr: count the owners

```
1  #include <memory>
2
3
4  std::shared_ptr<int> a(new int(5));  // count = 1
5  std::shared_ptr<int> b = a;          // count = 2 (copy ok)
6  std::cout << a.use_count();          // 2
```



one object, count = 2 owners

Freed when the last owner leaves



a destroyed -> count 2 -> 1, object stays

b destroyed -> count 1 -> 0 -> delete

Reference counting

- Copy -> **count + 1**.
- Destroy a shared_ptr -> **count - 1**.
- Count hits **0** -> the object is deleted, once.
- **use_count()** reports the current number of owners.
- **make_shared<T>(...)** is the preferred constructor.

Choosing: unique vs shared

Reach for unique_ptr when…

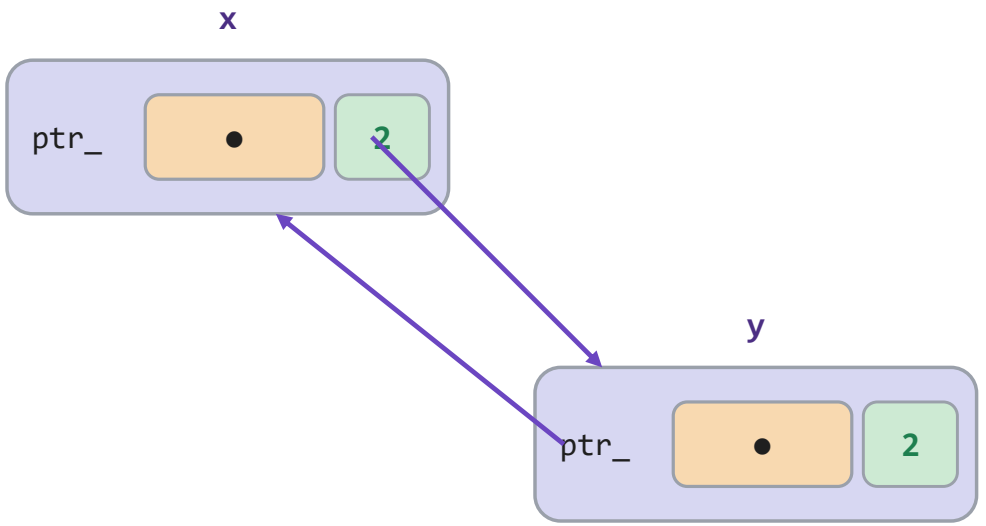
- …there is **one clear owner** at a time (the common case).
- Factory returns, members that own a resource.
- **Free, and it** documents ownership in the type.
- Move it to hand ownership on.

Reach for shared_ptr when…

- …ownership is **genuinely shared** and lifetimes overlap unpredictably.
- Graph / cache nodes reachable many ways.
- **Costs a** reference count (atomic, thread-safe).
- **If you cannot name a second owner,** use unique.

Uh-oh! Reference cycles

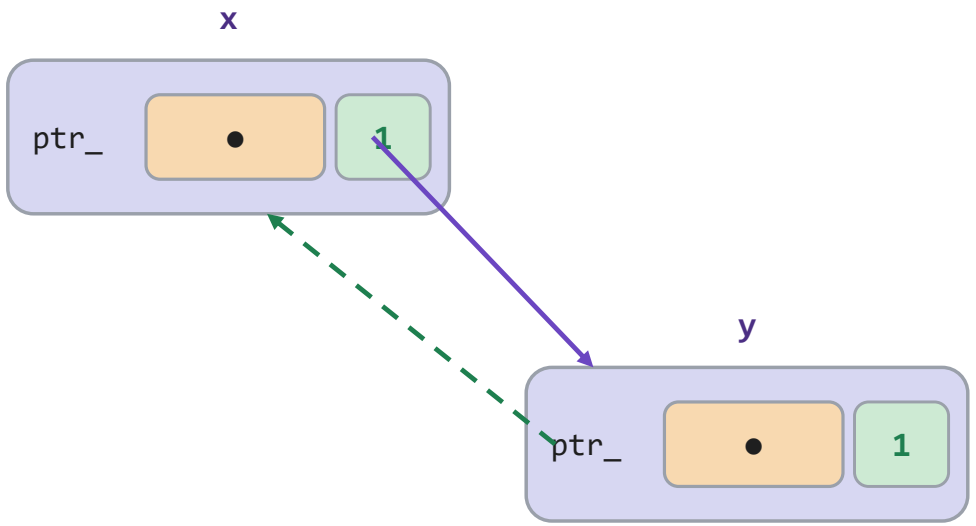
```
1  struct Node {
2      std::shared_ptr<Node> other;
3  };
4
5  auto x = std::make_shared<Node>();
6  auto y = std::make_shared<Node>();
7  x->other = y;   // y's count -> 2
8  y->other = x;   // x's count -> 2
```



x and y leave scope: 2 -> 1, never 0

weak_ptr: observe without owning

```
1  struct Node {
2    std::shared_ptr<Node> next;   // owns forward
3    std::weak_ptr<Node>   prev;   // does NOT count
4  };
5
6  // use a weak_ptr by promoting it:
7  if (auto p = n->prev.lock()) {
8    // p is a shared_ptr; object still alive
9  }
```



back-edge is weak: does not count

The three, side by side

`unique_ptr<T>`	`shared_ptr<T>`	`weak_ptr<T>`
One owner	Many owners	No ownership
Copy: DELETED	Copy: count + 1	Does not count
Move to transfer	Free at count 0	lock() -> shared_ptr
Zero overhead	use_count()	expired()
make_unique<T>()	make_shared<T>()	Breaks cycles
Your DEFAULT	Only if truly shared	Use for back-edges

Example: returning ownership

```
1  something_ptr<Image> LoadImage(const string& path) {
2    auto image = make_something<Image>(path);
3    if (!image->ok())
4      return nullptr;
5    return image;
6  }
7
8  auto logo = LoadImage("logo.png");
9  if (logo != nullptr)
10   logo->Draw();
11 // logo frees the Image when it leaves scope
```

Why type of ptr fits here?

Example: returning unique ownership

```
1  unique_ptr<Image> LoadImage(const string& path) {
2    auto image = make_unique<Image>(path);
3    if (!image->ok())
4      return nullptr;
5    return image;  // ownership moves to the caller
6  }
7
8  auto logo = LoadImage("logo.png");
9  if (logo != nullptr)
10   logo->Draw();
11 // logo frees the Image when it leaves scope
```

Why unique_ptr fits

- ❖ **LoadImage creates the object**, then hands ownership to exactly one caller.
- ❖ **Returning moves the unique_ptr**. It does not copy the Image.
- ❖ **Failure is just nullptr**, with no partial object to delete.
- ❖ Use this for factories **and any API that transfers sole ownership.**

Example: sharing one texture

```
1  class Sprite {
2   public:
3    Sprite(something_ptr<Texture> texture)
4        : texture_(move(texture)) { }
5   private:
6    something_ptr<Texture> texture_;
7  };
8
9  auto texture = make_something<Texture>("player.png");
10 Sprite player(texture);
11 Sprite reflection(texture);
12 // …
```

Why type of ptr fits here?

Example: sharing one texture

```
1  class Sprite {
2   public:
3    Sprite(shared_ptr<Texture> texture)
4        : texture_(move(texture)) { }
5   private:
6    shared_ptr<Texture> texture_;
7  };
8
9  auto texture = make_shared<Texture>("player.png");
10 Sprite player(texture);
11 Sprite reflection(texture);
12 // one Texture, three owners; freed after the last
```

Why shared_ptr fits

- **Two Sprite objects need the same Texture**, and either Sprite may be destroyed first.
- Each Sprite stores an owner**, so the Texture stays alive while either still needs it.**
- **The Texture is allocated once**. Copies only update the reference count.
- Use this only when the owners are real **and their lifetimes genuinely overlap.**

Networking

We use the Internet all day

message a friend

`phone → cloud → phone`

open a web page

`browser → many servers`

stream a lecture

`video arrives while we watch`

push to Git

`local files → remote repository`

It feels immediate. Underneath, many machines, links, and protocols cooperate to move the bytes.

What is a network?

Nodes

devices that send, receive, or forward data

Links

wired or wireless paths between nearby nodes

Protocols

shared rules for formatting and exchanging data

The Internet is a network of networks joined by routers and common protocols.

The complete seven-layer model

#	OSI layer	Responsibility	Examples	Internet stack
7	**Application**	Meaning and message rules	`HTTP, DNS`	**Application**
6	**Presentation**	Representation, encryption	`UTF-8, TLS`	**Application**
5	**Session**	Conversation coordination	`RPC session`	**Application**
4	**Transport**	Process-to-process delivery	`TCP, UDP`	**Transport**
3	**Network**	Host-to-host routing	`IP`	**Internet**
2	**Data link**	Delivery across one link	`Ethernet`	**Link**
1	**Physical**	Bits as signals	`copper, radio`	**Physical**

Layer 1: Physical

Move individual bits across one physical medium.

Copper

voltage changes

`0 1 1 0`

Fiber

light pulses

`dim bright bright dim`

Wi-Fi

radio waves

`modulated signal`

The layer provides

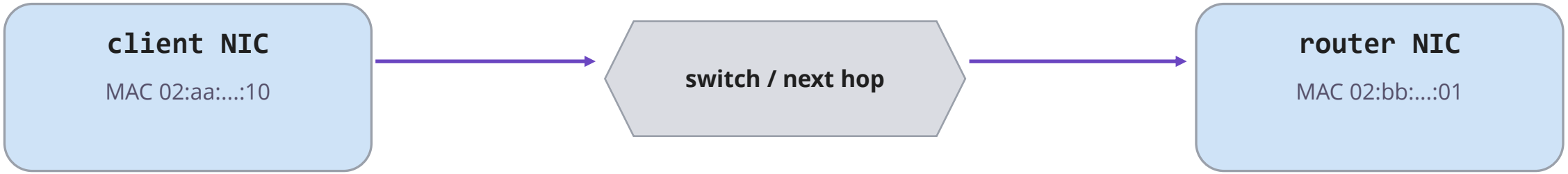
- A way to encode and detect bits on the medium

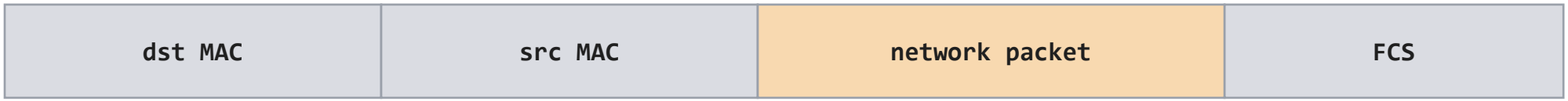
Practical checks

- Cable, link light, Wi-Fi signal, interface up/down

Layer 2: Data Link

Deliver one frame across one local link.





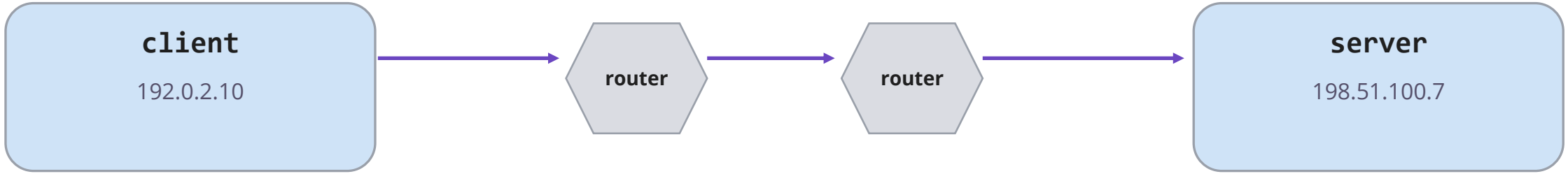
Ethernet and Wi-Fi

- Define frames and local interface addresses

Practical checks

- ip link, switches, ARP / neighbor table

Layer 3: Network



Each router inspects the destination IP and chooses a next hop.

Layer 4: Transport

simplified TCP segment

TCP provides

- Source and destination ports
- Ordered, reliable byte delivery
- Flow and congestion control

What the kernel can decide

- Port 3333 selects the listening uppercase service
- The endpoint pair keeps this client separate from other clients

Layer 5: Session

Organize a sustained conversation between applications.

1. **start** — authenticate / negotiate
2. **exchange** — many requests and responses
3. **recover** — resume or reconnect
4. **finish** — logout / orderly shutdown

In real Internet software

❖ Often handled by application protocols and libraries

Examples

❖ Login state, RPC sessions, checkpoints, reconnect tokens

Layer 6: Presentation

Give shared meaning and representation to application data.

Structure

`object → JSON / protobuf`

Encoding

`text → UTF-8 bytes`

Compression

`bytes → fewer bytes`

Encryption

`plaintext → TLS records`

Agreement matters

- Both endpoints must interpret the same bytes the same way

Layer 7: Application

Define the messages and behavior users actually need.

```
// uppercase protocol
request:  hello, networks\n
response: HELLO, NETWORKS\n
```

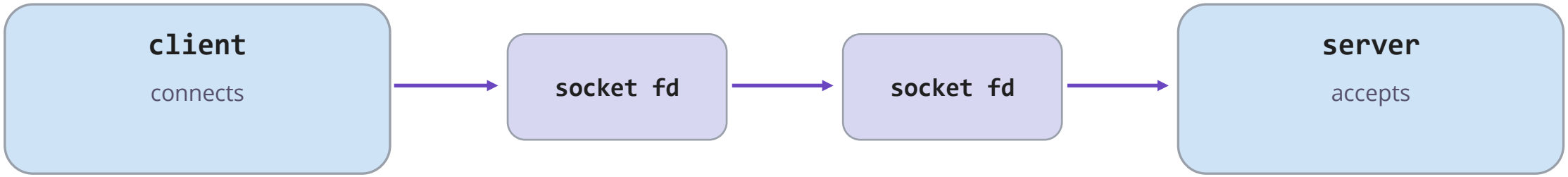
The protocol decides

- What requests and responses mean
- Where each message ends
- What errors and valid sequences look like

HTTP	web
DNS	names
SMTP	mail
SSH	remote shell

Sockets

TCP and Sockets!




```
1  write(socket_fd, "hello\n", 6);
```

A socket is a file descriptor

```c
int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
if (socket_fd == -1) return EXIT_FAILURE;

char buffer[64];
write(socket_fd, "hello\n", 6);
read(socket_fd, buffer, sizeof(buffer));
close(socket_fd);
```

file descriptor table

0	pipe	stdin
1	pipe	stdout
2	pipe	stderr
3	TCP socket	remote process

Network I/O uses familiar UNIX file operations.

Takeaway: socket() creates the handle; read(), write(), and close() use it.

Stream sockets vs datagram sockets

```
1  int fd = socket(AF_INET, SOCK_STREAM, 0);
2  // connection-oriented TCP byte stream
```

```
1  int fd = socket(AF_INET, SOCK_DGRAM, 0);
2  // connectionless UDP datagrams
```

Stream socket (TCP)

- One connected peer at each endpoint
- Reliable, ordered byte stream
- Used by SSH, HTTP connections, databases

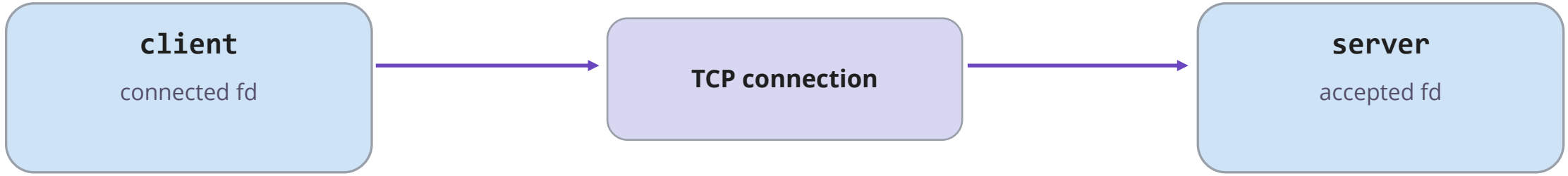
Datagram socket (UDP)

- Each send is a separate datagram
- No built-in delivery or ordering guarantee
- Used by DNS, games, real-time media

Clients connect; servers listen and accept

```
1  // client
2  int fd = socket(AF_INET, SOCK_STREAM, 0);
3  connect(fd, address, address_len);
4  write(fd, "hello\n", 6);
```

```
1  // server
2  int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
3  bind(listen_fd, address, address_len);
4  listen(listen_fd, SOMAXCONN);
5  int client_fd = accept(listen_fd, nullptr, nullptr);
```



TCP rebuilds an ordered byte stream

sender splits bytes into segments

network may reorder or lose packets

TCP handles

- Sequence numbers and reordering
- Retransmission after loss
- Flow and congestion control

The application receives

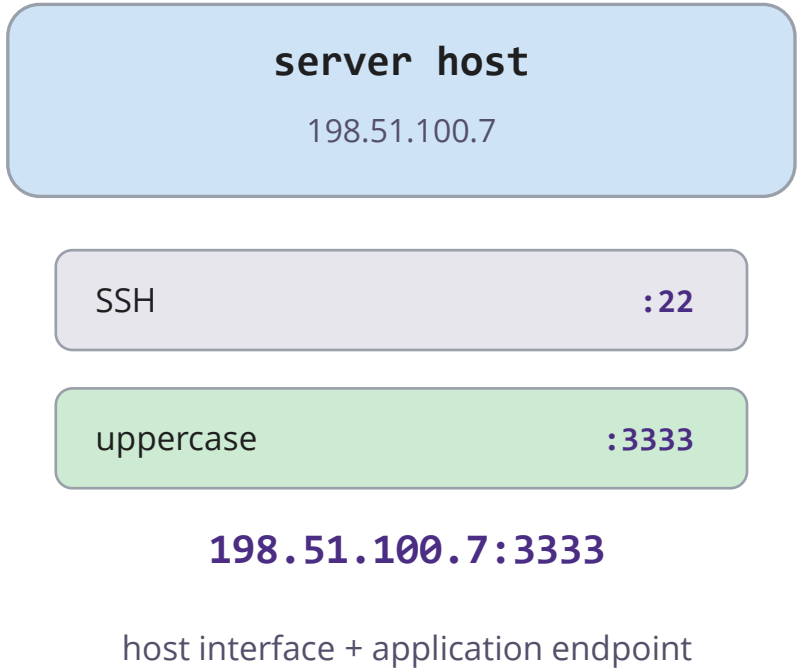
- The bytes in order: hello\n
- No packet boundaries
- No guarantee of one read per write

An endpoint combines an IP address and port

```
1  addrinfo hints = {};
2  hints.ai_socktype = SOCK_STREAM;
3  addrinfo* results = nullptr;
4  int rc = getaddrinfo("attu.cs.washington.edu",
5                       "3333", &hints, &results);
```

getaddrinfo()

- Turns a host name and service into socket addresses

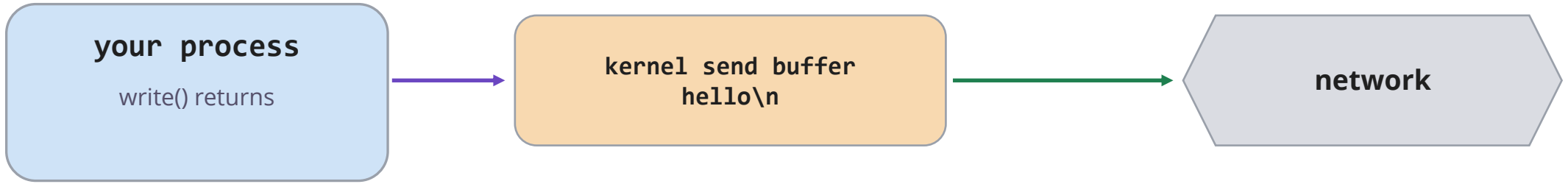


write() queues bytes in the kernel

```
1  const char* message = "hello\n";
2  ssize_t count = write(socket_fd, message, 6);
```

What the return value means

- Positive: that many bytes entered the send buffer
- -1: error; inspect errno
- The server probably has not received them yet



read() returns bytes currently available

```
1  char buffer[64];
2  ssize_t count = read(socket_fd, buffer, sizeof(buffer));
3  if (count == 0) { /* peer closed */ }
4  if (count == -1) { /* error */ }
```

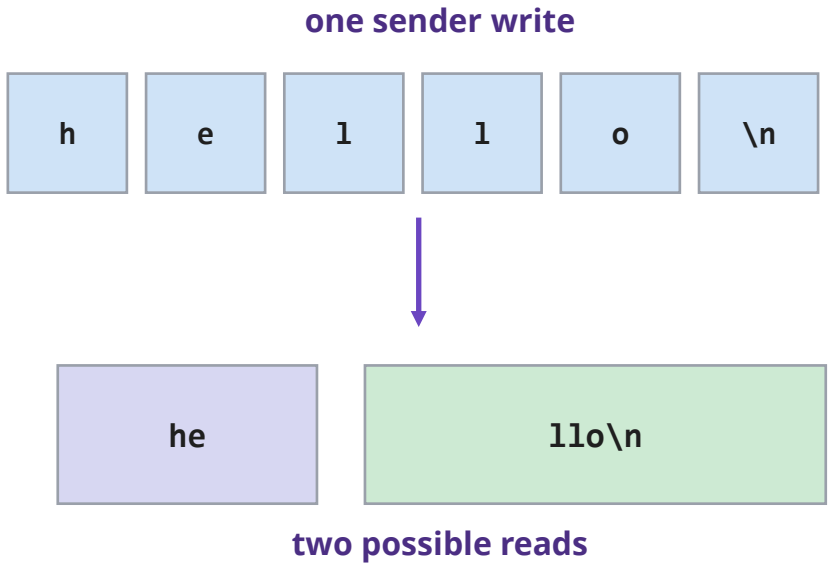
kernel receive buffer: h e l l o \n

`> 0`	available bytes copied
`0`	peer closed its write side
`-1`	error; EINTR may be retried
`waits`	buffer empty on blocking socket

A short positive count is normal, not EOF or an error.

TCP does not preserve message boundaries

```
1  string line;
2  char byte;
3  while (read(socket_fd, &byte, 1) == 1) {
4    line += byte;
5    if (byte == '\n') break;
6  }
7  // process one complete line
```



Application protocol

- Newline marks the complete request

Takeaway: TCP delivers a byte stream. Your protocol defines where messages end.

Reminders

Assessments:

- Exercise 6 due Thursday at 11:59pm
- Homework 3 due next Thursday at 11:59pm

General:

- Guest lecture on Monday, August 3rd
  - Works on Google Cloud (Data center, C++ client libraries, GPU team)
- Midterm will be graded on Thursday and released over the weekend

Questions?

Thanks for coming - see you next lecture!