LECTURE 23 · CSE 333 · Summer 2026

Final Exam Review

Discussion: What should we review first?

Logistics

WHAT

Final Exam

WHEN

Tue 8/18, 1:30-3:30

WHERE

CSE2 G10

FINAL

Logistics

WHAT

Final Exam

WHEN

Tue 8/18, 1:30-3:30

WHERE

CSE2 G10

Rules:

Standard exam rules: No talking. No devices. No peeking at other students.

Only allowed a pencil or pen. One page of handwritten notes, both sides.

Cumulative: all 26su material, with emphasis on topics after the midterm

Exam Advice

- **Skim each question before starting.** Prioritize your tasks.
- **Read each question *carefully*.** Note what it actually asks: the value, the output, or the bug.
- **Show your reasoning.** Partial credit follows a diagram or a short trace, not a lone answer.
- **Watch the C++ traps.** Pointer type vs actual object, integer vs real division, a copy that slices.
- **Budget your time.** If you get stuck, move on and come back.

FINAL

Studying Advice

DO

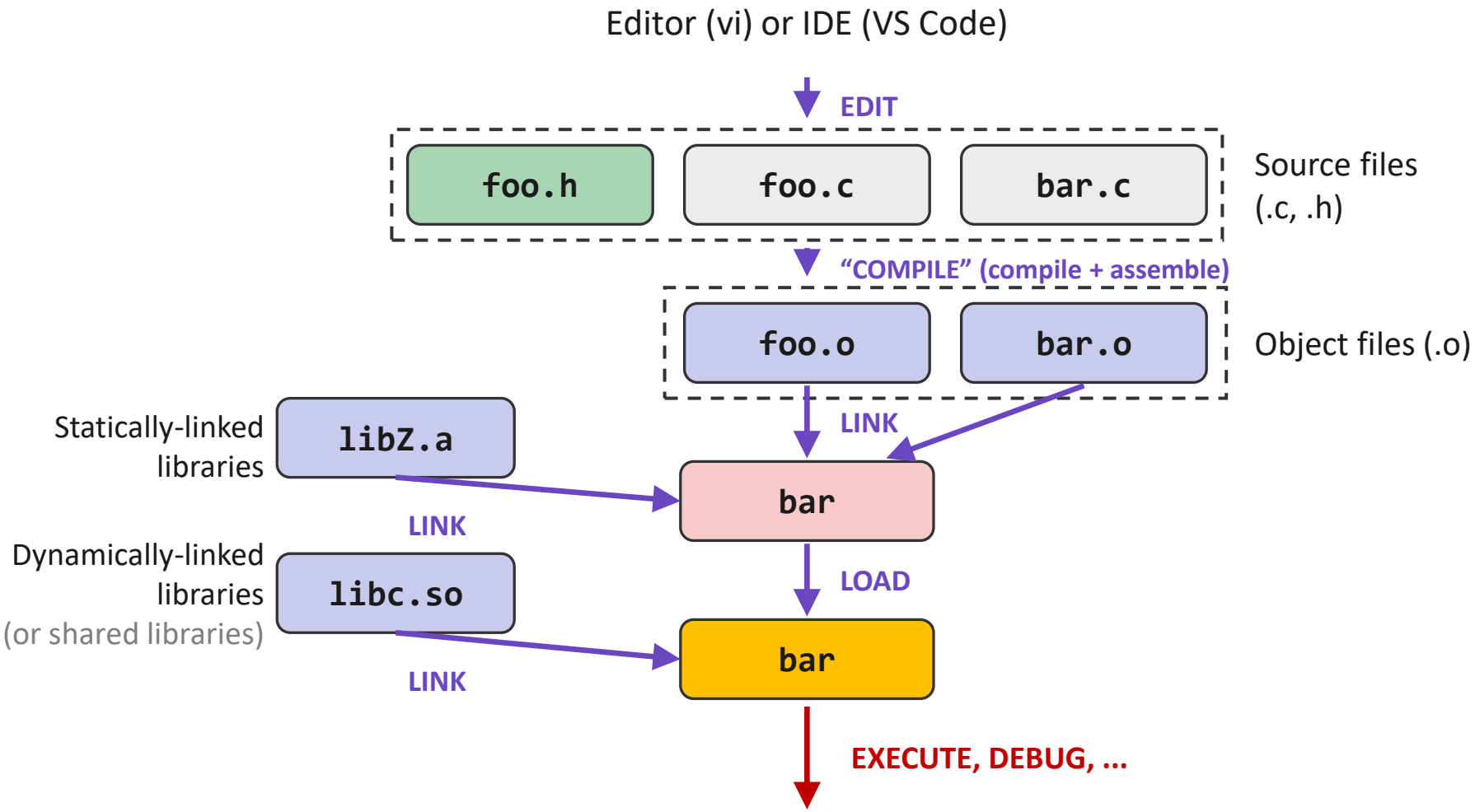
- Work old exams under time, then check.
- Draw the memory diagram by hand for every pointer problem.
- Practice the stuff that you're bad at, not the stuff you're good at.
- Build your one-page notes as a study act, not a dump.

AVOID

- Re-reading slides passively.
- Memorizing output without undersatning why.
- Skipping the C++ because it feels new.
- Cramming the notes page the night before.

Cumulative Foundations

Compilation: Source to Execution



EXAM-STYLE Q

Where does each thing live?

```c
int g = 7;                          // A
int main(int argc, char* argv[]) {
  int x = 3;                        // B
  int* p = malloc(sizeof(int));     // C
  char* s = "hi";                   // D
  return EXIT_SUCCESS;
}
```

QUESTION

For A, B, C, D, name where each value lives:

stack, heap, globals/data, or read-only.

For C and D, answer for BOTH the pointer and what it points at.

SOLUTION

Where does each thing live?

```
1 int g = 7;                          // A
2 int main(int argc, char* argv[]) {
3   int x = 3;                        // B
4   int* p = malloc(sizeof(int));     // C
5   char* s = "hi";                   // D
6   return EXIT_SUCCESS;
7 }
```

ANSWER

A g: globals/data segment.

B x, C p, D s: on the stack (local variables).

C *p: the heap block from malloc.

D "hi": read-only data; s just points at it.

Inheritance

Base and Derived Classes

```
1  #include "Stock.h"
2
3  class DividendStock : public Stock {   // is-a Stock
4   public:
5    DividendStock(...);                  // NOT inherited
6    double GetMarketValue() const override;
7    void PayDividend(double amount);     // new behavior
8   private:
9    double dividends_;                   // new state
10 };
```

Public inheritance

- **public inheritance** is like Java's extends: every non-private member of Base is part of Derived too.
- **Access:** public visible to all, private only to the class, protected visible to derived classes.
- **NOT inherited:** constructors, destructor, copy ctor, operator=. The derived class writes its own.

Static Dispatch: Compile-Time Choice

```
1  class Stock {
2   public:
3    double GetMarketValue() const;          // NOT virtual
4  };
5  class DividendStock : public Stock {
6   public:
7    double GetMarketValue() const;          // hides the base
8  };
9
10 DividendStock dividend;
11 Stock* s = &dividend;     // base ptr, derived object
12 s->GetMarketValue();      // Stock::GetMarketValue (the BASE)
```

Resolved at compile time

- ❖ With no **virtual**, C++ binds the call by the POINTER's declared type.
- ❖ s is a **Stock***, so the compiler wires the call to Stock::GetMarketValue.
- ❖ **The object is really a DividendStock,** but that does not matter here.
- ❖ **Java surprise:** every Java method is virtual, so this is the opposite of the default you expect.

Dynamic Dispatch: Runtime Choice

```
1  class Stock {
2   public:
3    virtual double GetMarketValue() const;   // opt in
4    double GetProfit() const;                // inherited
5  };
6  class DividendStock : public Stock {
7   public:
8    double GetMarketValue() const override;
9  };
10
11 Stock* s = &dividend;   // still a Stock*
12 s->GetMarketValue();    // DividendStock::GetMarketValue
13 s->GetProfit();         // inherited, uses the DERIVED value
```

Resolved at run time

- **virtual** asks for dynamic dispatch: the actual object decides.
- **The same s->GetMarketValue()** now runs the DividendStock version, through a Stock*.
- **Inherited code counts too:** GetProfit lives in Stock, but its call to GetMarketValue picks the derived one.
- **This is Java's default;** in C++ you opt in with virtual.

Takeaway: Dynamic dispatch picks the most-derived version from the actual object, even through a base pointer or from inside inherited code.

virtual and override

```
1  class Stock {
2   public:
3    virtual double GetMarketValue() const;  // opt in
4    double GetProfit() const;               // inherited
5  };
6  class DividendStock : public Stock {
7   public:
8    double GetMarketValue() const override; // Google style
9  };
```

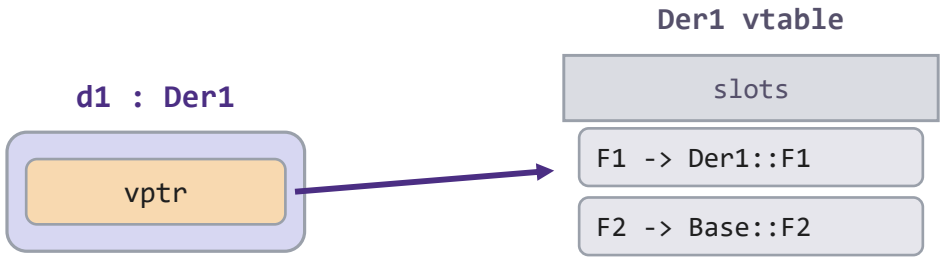
Two keywords

- **virtual** on the base method requests dynamic dispatch (this is Java's default, always on).
- **override** on the derived method says 'I mean to override', and the compiler checks the signature matches.
- **Almost always** you want virtual. Google style: use override, not virtual, in the derived class.

Takeaway: virtual on the base turns on dynamic dispatch; override on the derived catches signature bugs. Use both purposefully and follow the local convention.

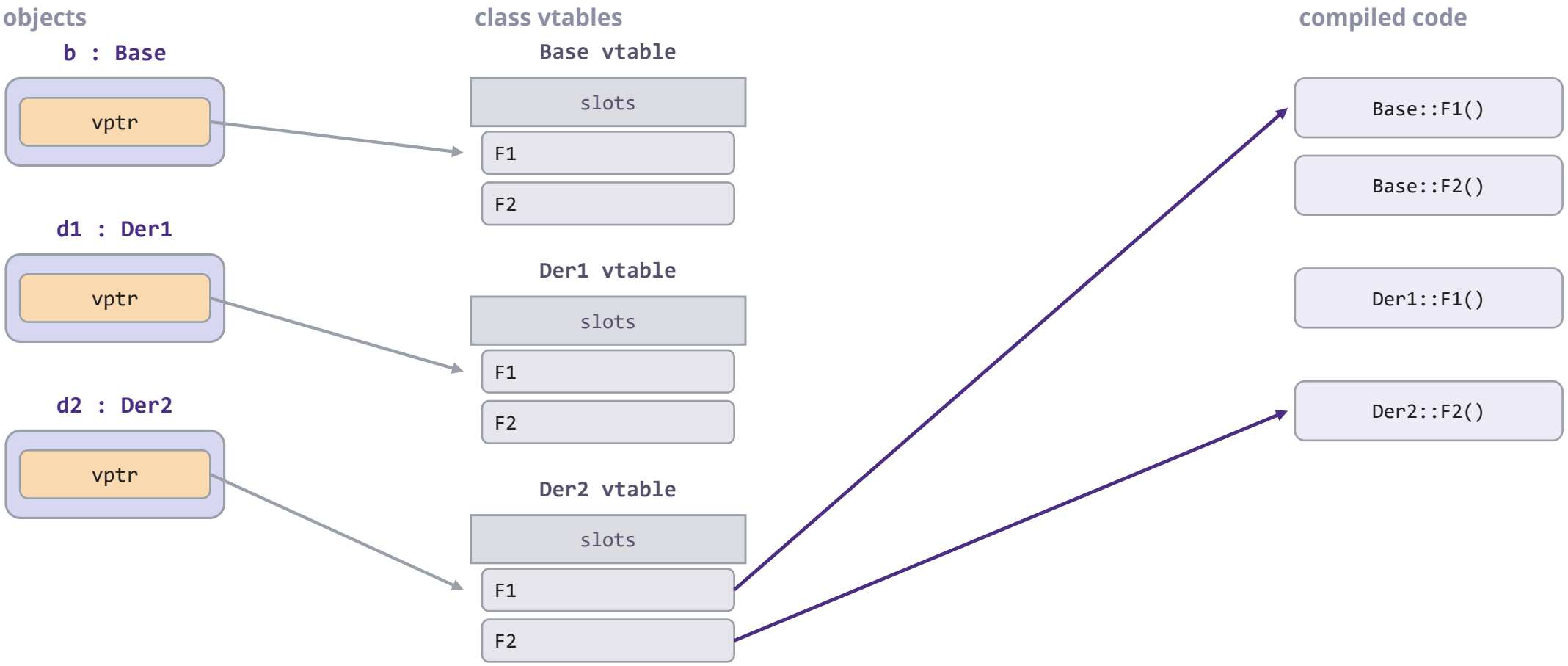
The vptr and the vtable

- **Compiled separately:** Stock.o is built from Stock.cc alone; it has never heard of DividendStock.
- So the call cannot be a hard-coded address. The answer is function pointers!
- **Any class with a virtual method** gets ONE vtable: an array of function pointers, one per virtual method, aimed at the most-derived versions.
- **Every object** gets a hidden vptr set by its constructor to point at its class's vtable.



the vptr 'remembers' the object's class

A virtual call: p->vptr->slot()



Takeaway: One vtable per class, one vptr per object. A virtual call follows p->vptr to the vtable, indexes the method's slot, and jumps to that function.

Pure virtual functions

```
1  class Shape {
2   public:
3    virtual double Area() const = 0;   // pure virtual
4    virtual ~Shape() { }
5  };
6  class Circle : public Shape {
7   public:
8    explicit Circle(double r) : r_(r) { }
9    double Area() const override { return 3.14159*r_*r_; }
10  private:
11   double r_;
12 };
13 // Shape s;                    // ERROR: abstract
14 Shape* p = new Circle(2.0);  // OK: ptr to concrete type
```

= 0 makes it pure

- **virtual double Area() const = 0;** declares the method with no body.
- **Any pure virtual** makes the class abstract: you cannot instantiate it.
- **Derived classes** must override every pure virtual to become concrete.
- **Only pure virtuals?** That is exactly a Java interface.

Takeaway: A pure virtual method (= 0) has no body and makes its class abstract. Use it for an interface every concrete subclass must fill in.

Why Base Destructors Must Be Virtual

```
1  class Base {
2   public:
3    Base()  { x_ = new int; }
4    ~Base() { delete x_; }      // NON-virtual
5    int* x_;
6  };
7  class Der1 : public Base {
8   public:
9    Der1()  { y_ = new int; }
10   ~Der1() { delete y_; }
11   int* y_;
12 };
13 Base* b = new Der1;
14 delete b;   // only ~Base runs -> y_ LEAKS (UB)
```

```
1  class Base {
2   public:
3    Base()          { x_ = new int; }
4    virtual ~Base() { delete x_; }   // virtual dtor
5    int* x_;
6  };
7  // ...
8  Base* b = new Der1;
9  delete b;   // ~Der1 THEN ~Base run -> no leak
```

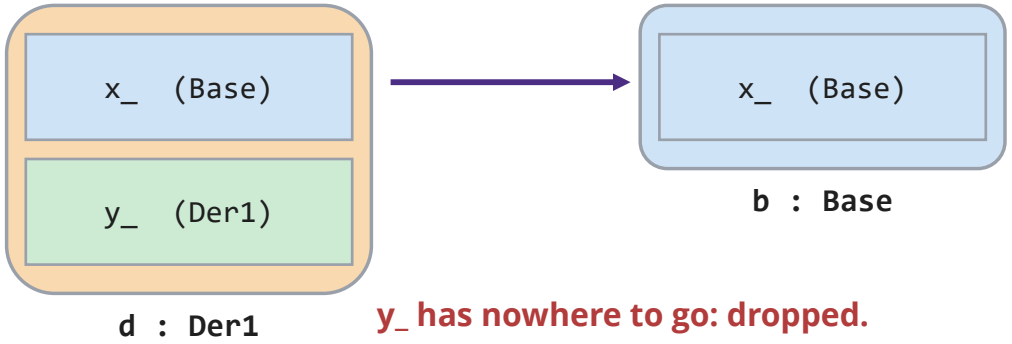
Destruction runs top-down: ~Der1 then ~Base. A non-virtual base dtor called through a Base* skips ~Der1, so y_ leaks (formally undefined behavior). Marking ~Base virtual fixes it.

Takeaway: Deleting a derived object through a base pointer with a non-virtual destructor leaks (UB). Always give a class meant for inheritance a virtual destructor.

Object Slicing: Only the Base Part Is Copied

```
1  class Base {
2   public:
3    Base(int x) : x_(x) { }
4    int x_;
5  };
6  class Der1 : public Base {
7   public:
8    Der1(int y) : Base(16), y_(y) { }
9    int y_;
10 };
11 void Foo() {
12   Base b(1);
13   Der1 d(2);
14   b = d;   // SLICED: copies x_, drops y_
15 }
```

b = d; copies only the Base part



Takeaway: Assigning a derived object to a base VALUE slices off the derived parts. A Base variable has no room for Der1's y_.

Containers Store Values, So They Can Slice

```
1  #include <vector>
2  #include <memory>
3  #include "Stock.h"
4  #include "DividendStock.h"
5
6  std::vector<Stock> v;          // stores Stock BY VALUE
7  DividendStock ds;
8  v.push_back(ds);               // OUCH: sliced to a Stock
9
10 // Fix: store pointers (ideally smart pointers)
11 std::vector<std::unique_ptr<Stock> > v2;
12 v2.push_back(std::make_unique<DividendStock>());
```

Containers copy by value

❖ **So vector<Stock>** copies a DividendStock into a Stock-sized slot: sliced.

❖ **Fix: store pointers.** A pointer is the same size for any type, so nothing is dropped.

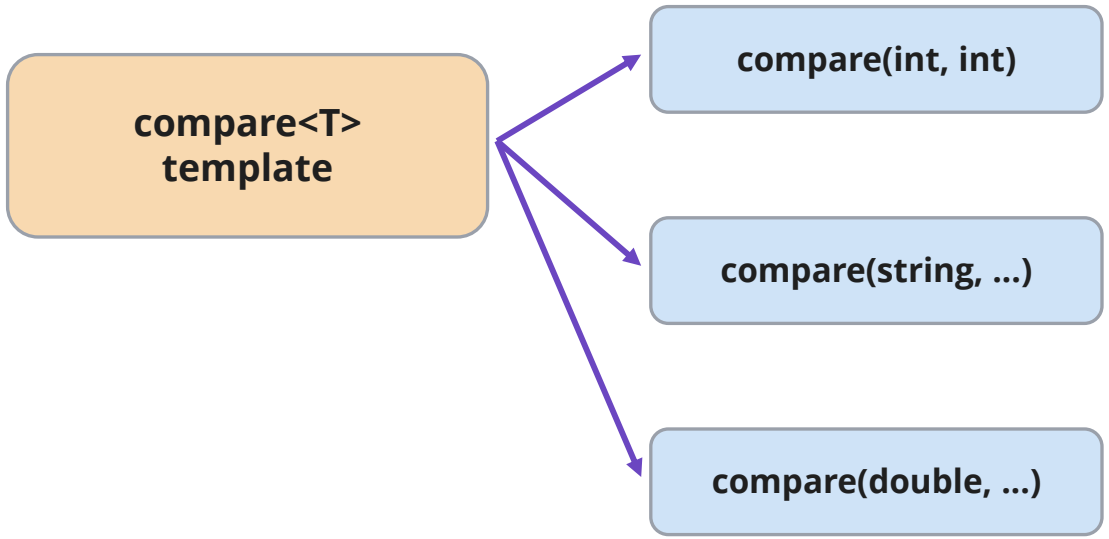
❖ **Best:** vector<unique_ptr<Stock> > so the container also owns and frees them.

Templates

Templates Are Compile-Time Recipes

Parametric polymorphism

- A **template** is code parameterized by a TYPE, written once.
- **The compiler** generates a specialized copy for each type you actually use.
- This happens at **compile time**, not run time.
- **The template itself is** not runnable code; it is a pattern for making code.



compiler emits one real function per type

Why Template Definitions Must Be Visible

```
1  // compare.h
2  template <typename T>
3  int compare(const T& a, const T& b);    // declaration
4
5  // compare.cc
6  template <typename T>
7  int compare(const T& a, const T& b) { /* ... */ }
8
9  // main.cc
10 #include "compare.h"
11 compare(3, 4);   // LINK ERROR: no code was generated
```

Why it fails to link

- To make code for **compare<int>**, the compiler must SEE the template body.
- main.cc only sees the **declaration** from the header.
- compare.cc compiles alone: nobody used T there, so no code is generated.
- **Result:** undefined reference at link time.

Template Definitions Belong in Headers

Solution 1: full definition in .h

```
1  // compare.h  (Google style: preferred)
2  template <typename T>
3  int compare(const T& a, const T& b) {
4    if (a < b) return -1;
5    if (b < a) return 1;
6    return 0;
7  }
8  // no compare.cc: definition in header
```

Solution 2: header #includes the .cc (WEIRD!)

```
1  // compare.h
2  template <typename T>
3  int compare(const T& a, const T& b);
4
5  #include "compare.cc"   // pull definitions
6
7  // compare.cc holds the template bodies
```

Takeaway: Because the compiler needs to see the body, template definitions go in the header. Google style puts the full definition there directly; that is what we do in CSE 333.

STL

(Standard Template Library)

Containers, Iterators, and Algorithms

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.

vector: a resizable array

```
1  #include <vector>
2  std::vector<T> v;
3  T thing1, thing2;           // ctor
4  v.push_back(thing1);
5  v.push_back(thing2);
6
7
```

vector<T>

- Dynamically resizable, **contiguous array.**
- **O(1)** random access: v[i], v.at(i).
- **Amortized O(1)** push_back at the end.
- **Insert/erase in the middle is** O(n): everything after shifts.
- Each push_back **copies** the element in.

Containers Copy Their Elements

By value means copies

- **An STL container holds** its own copy of each element.
- Insert an object -> the container **copies it in** (copy ctor).
- Rearranging (sorting, resizing) **copies or moves elements** around.
- **Big objects ->** copying gets expensive fast.

```
1  class Tracer {
2   public:
3    Tracer()                 { cout << "ctor\n"; }
4    Tracer(const Tracer& t) { cout << "copy ctor\n"; }
5    Tracer& operator=(const Tracer& t) {
6      cout << "operator=\n";  return *this;
7    }
8    ~Tracer()                { cout << "dtor\n"; }
9  };
```

Tracer prints on every ctor / copy / assign / dtor, so we can SEE the copies.

Polymorphic Objects Can Be Sliced in Containers

```
1  #include <vector>
2  #include <memory>
3  #include "Stock.h"
4  #include "DividendStock.h"
5
6  std::vector<Stock> v;          // stores Stock BY VALUE
7  DividendStock ds;
8  v.push_back(ds);               // OUCH: sliced to a Stock
9
10 // Fix: store pointers (ideally smart pointers)
11 std::vector<std::unique_ptr<Stock> > v2;
12 v2.push_back(std::make_unique<DividendStock>());
```

Containers copy by value

❖ **So vector<Stock>** copies a DividendStock into a Stock-sized slot: sliced.

❖ **Fix: store pointers.** A pointer is the same size for any type, so nothing is dropped.

❖ **Best:** vector<unique_ptr<Stock> > so the container also owns and frees them.

Iterators

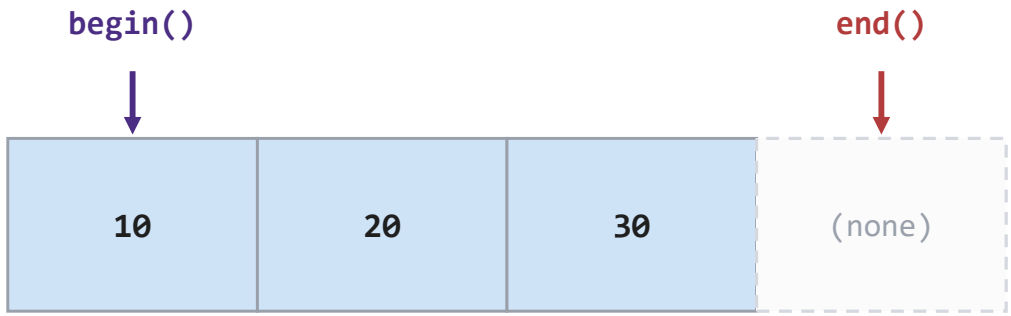
```
1  std::vector<int> v = {10, 20, 30};
2
3  // every container defines its own iterator type
4  std::vector<int>::iterator it;
5  for (it = v.begin(); it != v.end(); ++it) {
6    cout << *it << endl;   // * dereferences the iterator
7  }
```

What an iterator is

- A generalized **pointer** into a container.
- **begin()**: points at the first element.
- **end()**: one PAST the last element.
- ***it** reads it, **++it** advances it.

- Every container defines its own **iterator** type, but the loop looks the same for all of them: begin to end, dereference, increment.

The range [begin, end)



Why one-past-the-end?

- The range is **half-open**: includes begin, excludes end.
- Empty container -> **begin() == end()**, so the loop runs zero times.
- **end()** is a marker, not a real element: never dereference it.
- Size is just **end - begin** for random-access iterators.

Takeaway: STL ranges are half-open: [begin, end). end() sits one past the last element as a stop marker, which makes empty ranges and loop termination fall out cleanly.

map: ordered key to value

```
#include <map>
std::map<string, int> ages;       // key -> value, kept sor
ages["Alice"] = 30;           // insert or update
ages["Bob"]   = 25;

auto it = ages.find("Alice");
if (it != ages.end())
  cout << it->first << ": " << it->second;   // Alice: 30
```

map<Key, Value>

- ❖ Ordered associative container: a **balanced search tree.**
- ❖ Unique keys, kept **sorted**; O(log n) insert and lookup.
- ❖ Elements are **pair<const Key, Value>**.
- ❖ **m[key]** inserts or updates; **find** returns an iterator.

`it->first is the key, it->second is the value  (it is an iterator to a pair)`

map is a sorted key/value tree: unique keys, O(log n) insert and lookup, elements are pair<const Key, Value>, and an iterator gives you .first (key) and .second (value).

Choosing a Container: Browser History

The scenario

- As the user visits pages, you remember where they have been.
- Pressing **Back** returns to the MOST RECENT page, then the one before it.
- You only ever add to, or remove from, **one end.**
- You never need the 5th-oldest page by index.

Discuss with a neighbor

- Which END do add and remove happen at?
- What ordering is this? **(newest out first)**
- Which container gives **O(1) at that end?**
- Would list or map buy you anything here?

Predict first: add and remove at the same end, newest-out-first. Name the pattern.

Browser History Uses Stack Behavior

Use: a stack: vector<Url> (push_back / back / pop_back), or std::stack

```
1  vector<Url> history;          // used as a stack
2  history.push_back(url);       // visit a page
3  Url prev = history.back();    // peek newest
4  history.pop_back();           // press Back
5  // std::stack<Url> is the same idea, LIFO.
```

Why / tradeoffs

- **LIFO**: newest in, newest out.
- **vector** already gives **O(1)** push_back / pop_back at the end.
- **std::stack** is a thin adapter that just says "stack" in the type.
- No key lookup, no middle edits -> **map / list would be overkill.**

Smart Pointers

Ownership Determines Copy Behavior

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**.

Smart Pointer Ownership at a Glance

`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

The Network Stack at a Glance

#	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**

Socket Vocabulary

`host`	`port`	`connection`	`byte stream`
which machine	**which program on it**	**one conversation**	**what flows through it**
attu, or your laptop. Found by name or by IP address.	One machine runs many services. 22 is SSH, 3333 is ours.	Two endpoints, opened on purpose, kept apart from everyone else's.	Bytes, in order. Not messages. Just bytes.

Once you have these four, the code is a handful of calls similar to the file I/O you already wrote.

Sockets Are File Descriptors

```
1  int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
2  if (socket_fd == -1) return EXIT_FAILURE;
3
4  char buffer[64];
5  write(socket_fd, "hello\n", 6);
6  read(socket_fd, buffer, sizeof(buffer));
7  close(socket_fd);
```

file descriptor table

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

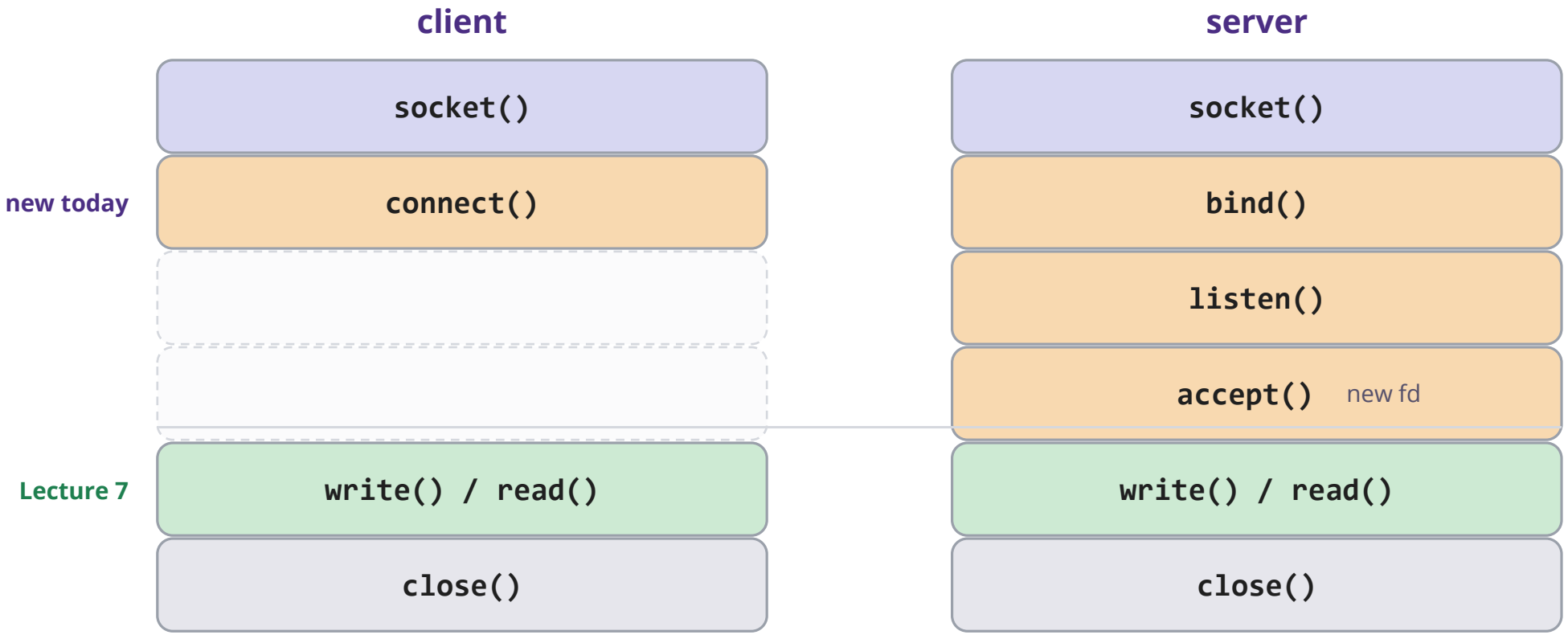
Network I/O uses familiar UNIX file operations.

TCP Does Not Preserve Message Boundaries

```
1  write(socket_fd, "hello\n", 6);
2  // receiver might observe:
3  read(socket_fd, buf, 64);  // "he"
4  read(socket_fd, buf, 64);  // "llo\n"
```

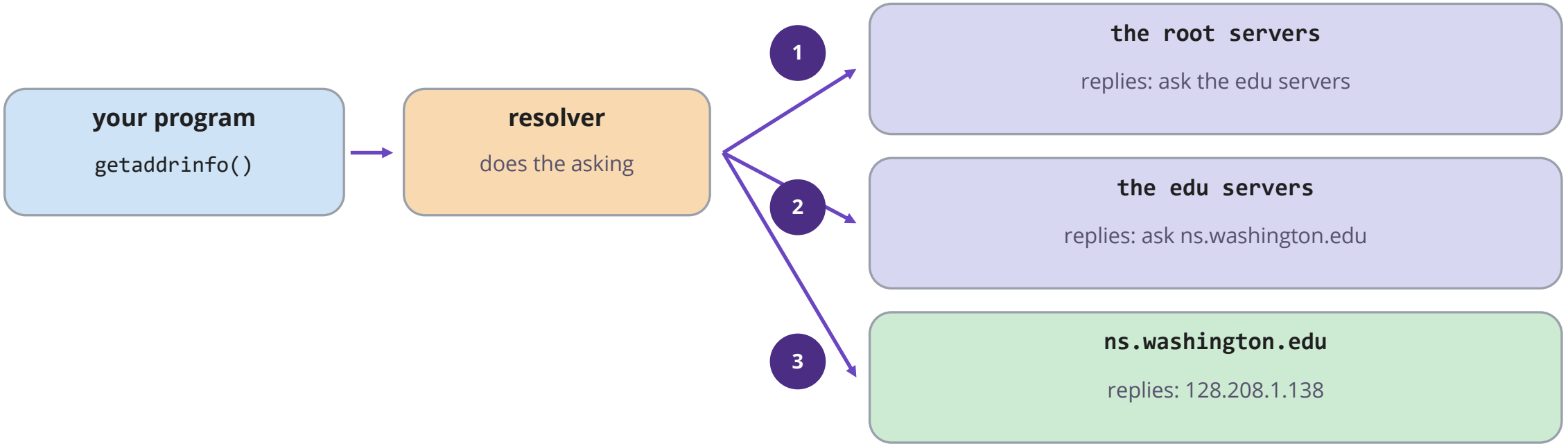
- Correct answer: **C**
- Accumulate bytes until the application message is complete

Client/Server Lifecycle



Both sides call socket(). The client names who it wants; the server claims a port and waits. Below the line, both sides are doing Lecture 7 file I/O.

DNS Resolution Path



UW was delegated this part of the tree, so ithe real answer. The resolver hands 128.208.1.138 back, and getaddrinfo() returns.

Anatomy of an HTTP Request

```
$ nc www.example.com 80
GET /index.html HTTP/1.1
Host: www.example.com
Connection: close

```

The last line is empty on purpose. That blank line is the whole framing rule.

Three parts

- **Request line:** method, path, version
- **Headers:** name, colon, value, one per line
- **A blank line:** the headers are over
- Lines end with carriage return and newline

Anatomy of an HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 137

<html><body>hello</body></html>
```

Same sort of shape as the request, but now with a status line on top and a body underneath.

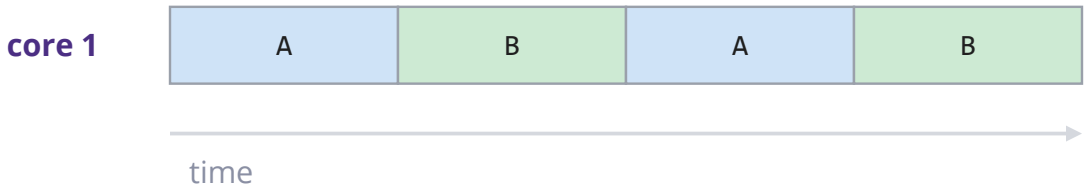
How you know when to stop

- Read until the blank line to get the headers
- Then read exactly **Content-Length** more bytes
- Without it you cannot tell a slow body from a finished one
- 200, 404, 500: the number is for the program, the text is for you

Concurrency and parallelism

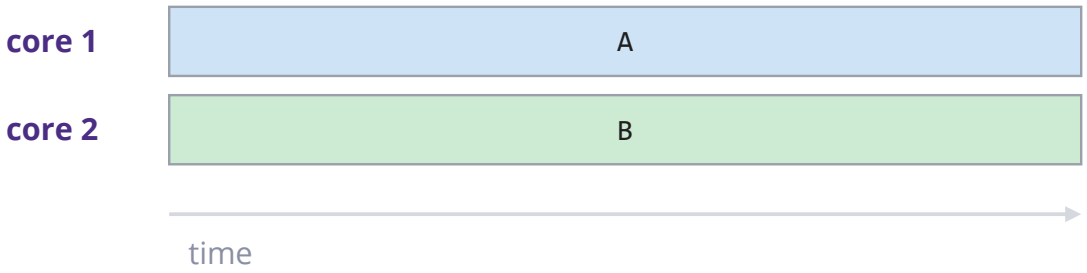
`concurrency`

Several tasks make progress during the same interval.



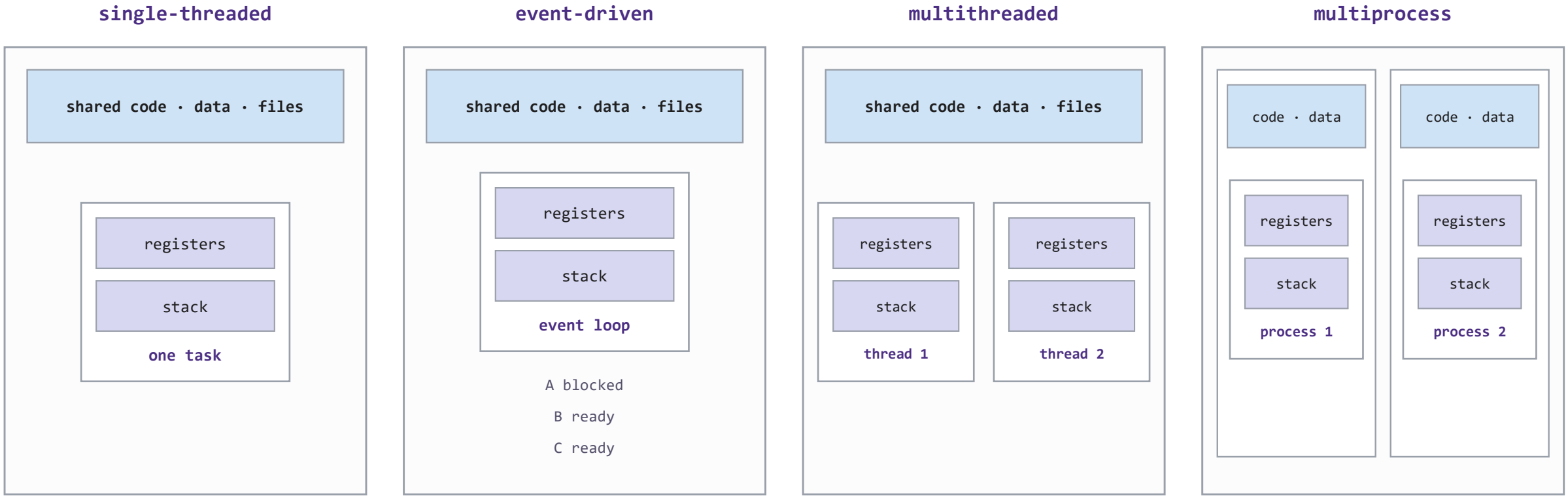
`parallelism`

Several instructions execute at the same instant.



Concurrency can improve an I/O-bound server on one core. Parallelism needs multiple cores and helps CPU-bound work.

Execution Models: What Is Shared?



Recognizing a Data Race

```
1  int total = 0;
2
3  void* Count(void* unused) {
4    (void) unused;
5    for (int i = 0; i < 1000000; i++) {
6      total++;
7    }
8    return NULL;
9  }
10
11 // four threads run Count; main joins them and prints total
```

Conflicting accesses

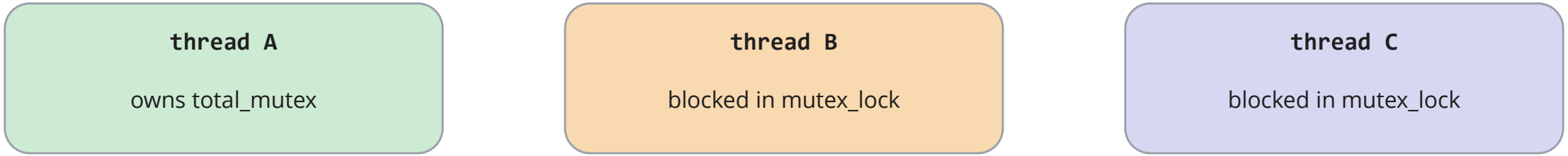
- All four workers access the same global total
- Each access in total++ includes a write
- No mutex or other synchronization orders the accesses
- The C program therefore has undefined behavior

How an Increment Gets Lost

step	thread A	A register	thread B	B register	total
1	read total	0		-	0
2		0	read total	0	0
3	add 1	1		0	0
4		1	add 1	1	0
5	write total	1		1	1
6		1	write total	1	1

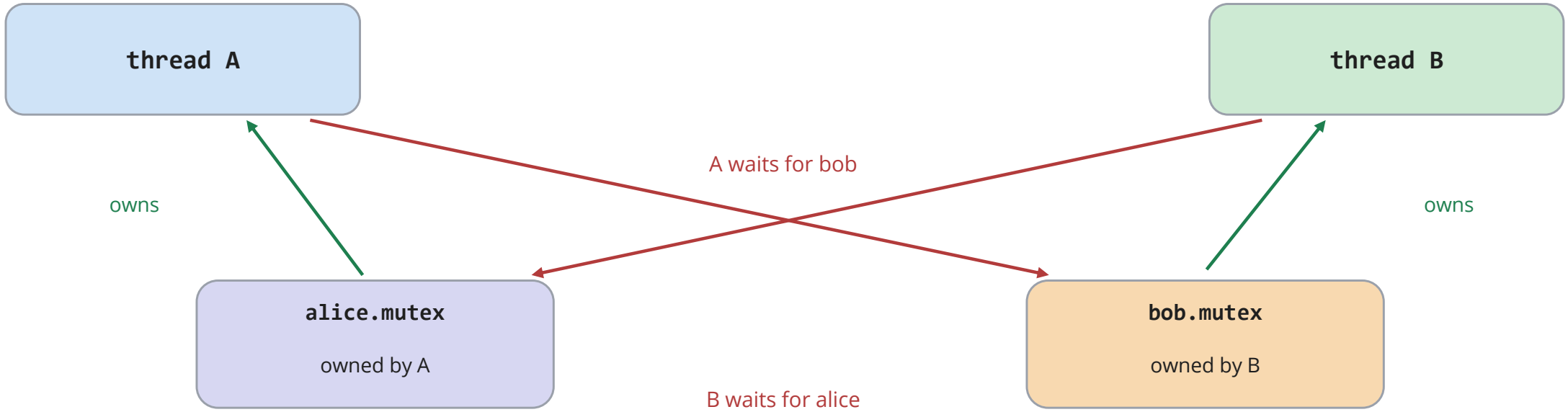
Both threads computed a locally correct 1. The second store overwrote the first store, so one increment was lost.

Mutexes Protect Critical Sections




```
1  pthread_mutex_lock(&total_mutex);    // acquire or sleep
2  total++;                             // critical section
3  pthread_mutex_unlock(&total_mutex);  // release ownership
```

Deadlock Is Circular Waiting



Neither blocked thread can execute an unlock. The cycle cannot resolve itself.

Threads vs. Processes

program state	pthread_create()	fork()
stack	one per thread	copied into child
heap and globals	shared	separate copies
file descriptors	shared table	inherited table copy
ordinary communication	read the same object	pipe, socket, file, or shared-memory mapping
failure isolation	low	higher

Questions?

Thanks for coming - see you at the final!