UNIVERSITY of WASHINGTON

LECTURE 14 · CSE 333 · Summer 2026

Midterm Review

Discussion: What stresses you out?

Reminders

Assessments:

- Exercise 6 will be released today. Due Thursday at 11:59pm
- Please show up to the midterm tomorrow in lecture!

General:

- Mid-quarter feedback this week
  - Anonymous, but submission is tracked. 1 participation pt for filling it out.
  - Be honest and critical! I can't improve without clear issues to work on ☹

MIDTERM

Logistics

WHAT

Midterm Exam

WHEN

9:40am – 10:40am

WHERE

IEB G106 (here)

MIDTERM

Logistics

WHAT

Midterm Exam

WHEN

9:40am – 10:40am

WHERE

IEB G106 (here)

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.
- Only 26su material: lectures, ex1, ex2, ex3, ex4, ex5, hw1, hw2

MIDTERM

Logistics

WHAT

Midterm Exam

WHEN

9:40am – 10:40am

WHERE

IEB G106 (here)

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.
- Only 26su material: lectures, ex1, ex2, ex3, ex4, ex5, hw1, hw2
- **Must bring your Husky card**

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.

MIDTERM

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

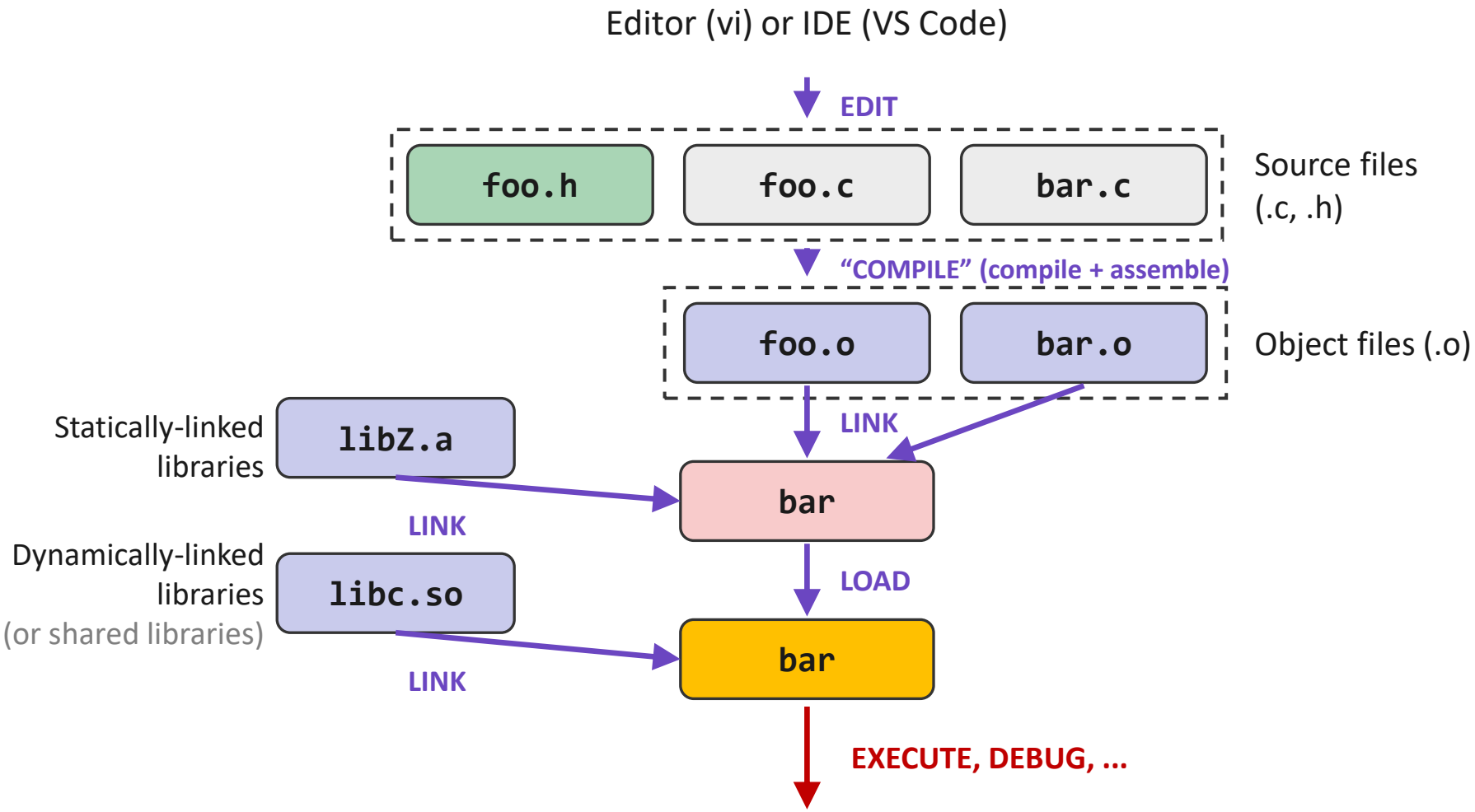
- Re-reading slides passively.
- Memorizing output without undersatning why.
- Skipping the C++ because it feels new.
- Cramming the notes page the night before.

MIDTERM

1	**Compilation**	each stage; compile vs link errors
2	**Decompilation**	compilation in reverse; what 'matching' means
3	**The memory diagram**	where each thing lives, and why
4	**Memory leaks**	the malloc with no free
5	**References and const**	aliases, and what const locks
6	**Constructors**	init lists, and members left uninitialized
7	**Inheritance**	base vs derived, and how a call is dispatched
8	**virtual and override**	the sharpest difference from Java
9	**Templates**	one recipe, many stamped-out types

Midterm Review

Compilation Process



PROBLEM: COMPILATION

EXAM-STYLE Q

The build command is missing a file

util.h

```
1 int Twice(int n);
```

main.c

```
1 #include "util.h"
2 int main(int argc, char* argv[]) {
3   printf("%d\n", Twice(21));
4   return EXIT_SUCCESS;
5 }
```

terminal

```
1 $ gcc -Wall -o prog main.c
```

QUESTION

util.c (not shown) defines Twice; the build names main.c only.

Does gcc stop while compiling, or while linking?

Name the error, then fix the command.

SOLUTION

The build command is missing a file

util.h

```
1 int Twice(int n);
```

main.c

```
1 #include "util.h"
2 int main(int argc, char* argv[]) {
3   printf("%d\n", Twice(21));
4   return EXIT_SUCCESS;
5 }
```

terminal

```
1 $ gcc -Wall -o prog main.c
```

ANSWER

Compiling main.c succeeds. util.h declares Twice, so the call type-checks and main.o is built.

Linking fails: undefined reference to `Twice' -- no object file defines it.

Fix: name util.c too, so both are compiled and linked:

$ gcc -Wall -o prog main.c util.c

What is decompilation?

- Compilation turns **source code** into a **binary** the machine runs.
- Decompilation goes the other way: start from the shipped **binary**, recover readable source.
- Nintendo never released the Pokemon source. All we have is the ROM.
- A **matching** decompilation is the gold standard: the recovered C must compile back to the **exact same bytes**.
  - If it rebuilds byte-for-byte, you have proven the source is faithful.

pokeemerald: a matching decomp

```
// src/data/battle_moves.h  (real, verbatim)
[MOVE_POUND] = {
    .effect = EFFECT_HIT,
    .power = 40,
    .type = TYPE_NORMAL,
    .accuracy = 100,
    .pp = 35,
    .priority = 0,
},
// this C compiles to the exact bytes
// found in the retail Emerald cartridge
```

- Built by **pret**, a Pokemon reverse-engineering community.
- Emerald is fully decompiled: 100% matching C.
- Rebuilds **pokeemerald.gba** byte-for-byte.
- Readable names, structs, and comments the original never shipped.
- This is the C you are learning, in a game you know.

PROBLEM: DECOMPILATION

EXAM-STYLE Q

What does a decompiler actually do?

SCENARIO

You have a shipped game as a ROM: **just raw bytes,** no source.

You want readable C you can edit and rebuild.

QUESTION

What are the two big steps from bytes to C?

And what makes a decompilation "matching"?

SOLUTION

What does a decompiler actually do?

SCENARIO

You have a shipped game as a ROM: **just raw bytes,** no source.

You want readable C you can edit and rebuild.

ANSWER

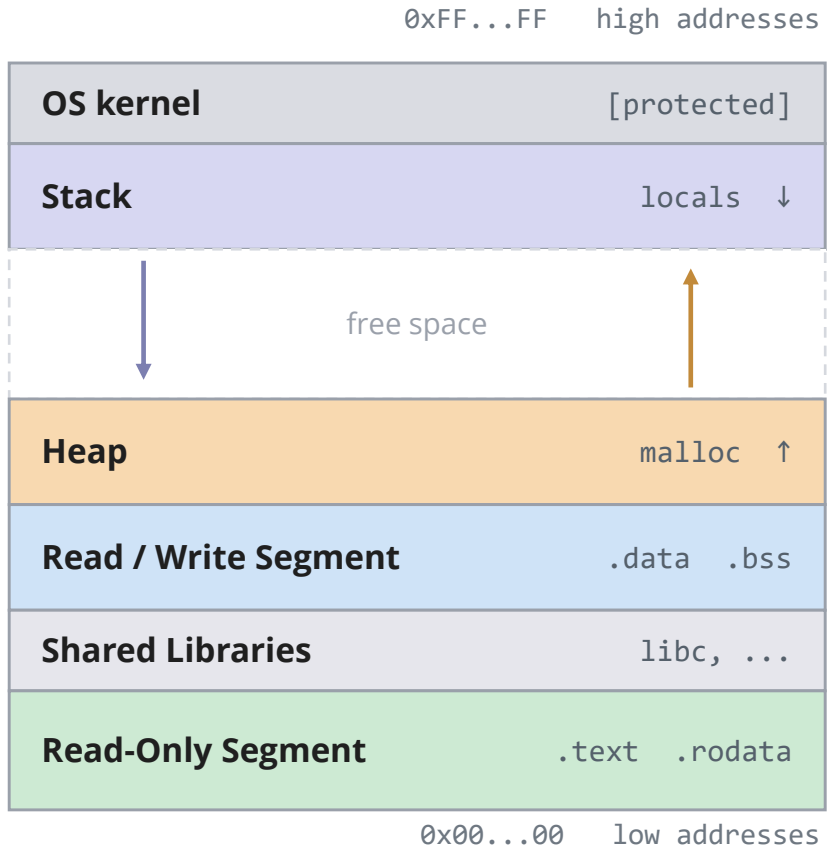
1. Disassemble: bytes to assembly. Mechanical and reliable.

2. Decompile: assembly to readable C. The human-hard part.

Matching: recompiling your C reproduces the original bytes exactly.

The whole address space at once

```
1  int g_limit = 10;       // .data  (init global)
2  int g_count;            // .bss   (zero global)
3  const char* k = "hi";   // ptr .data, "hi" .rodata
4
5  int main(int argc, char* argv[]) {
6    int local = 5;        // stack
7    int* buf =            // buf -> stack
8        malloc(40);       // *buf -> heap
9    free(buf);
10   return EXIT_SUCCESS;
11 }
```



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.

EXAM-STYLE Q

Find the leak (C)

```c
int* MakeArray(int n) {
  int* a = malloc(n * sizeof(int));
  return a;
}

int main(int argc, char* argv[]) {
  int* p = MakeArray(100);
  int* q = MakeArray(200);
  free(p);
  return EXIT_SUCCESS;
}
```

QUESTION

MakeArray does one malloc and returns the block.

main frees p but not q. Is there a leak? Which line, and the fix?

SOLUTION

Find the leak (C)

```c
int* MakeArray(int n) {
  int* a = malloc(n * sizeof(int));
  return a;
}

int main(int argc, char* argv[]) {
  int* p = MakeArray(100);
  int* q = MakeArray(200);
  free(p);
  return EXIT_SUCCESS;
}
```

ANSWER

p is fine: its block is freed on the free(p) line.

q leaks: MakeArray(200) malloc'd a block that nothing ever frees.

Fix: free(q); before the return.

EXAM-STYLE Q

Find the leak (C++)

```
1  int* MakeGrid(int rows, int cols) {
2    int* g = new int[rows * cols];
3    return g;
4  }
5
6  int main(int argc, char* argv[]) {
7    int* a = MakeGrid(4, 4);
8    int* b = MakeGrid(8, 8);
9    delete[] a;
10   return EXIT_SUCCESS;
11 }
```

QUESTION

Same shape, but now with new[] and delete[].

Which block leaks, and what exactly must free it?

SOLUTION

Find the leak (C++)

```
1  int* MakeGrid(int rows, int cols) {
2    int* g = new int[rows * cols];
3    return g;
4  }
5
6  int main(int argc, char* argv[]) {
7    int* a = MakeGrid(4, 4);
8    int* b = MakeGrid(8, 8);
9    delete[] a;
10   return EXIT_SUCCESS;
11 }
```

ANSWER

a is fine: delete[] a releases its array.

b leaks: the 64 ints from MakeGrid(8, 8) are never deleted.

Fix: delete[] b; and remember new[] pairs with delete[], never delete or free.

Pointers vs references, side by side

```
1  void IncPtr(int* p) { *p += 1; }   // pointer version
2  void IncRef(int& r) {  r += 1; }   // reference version
3
4  int main(int argc, char** argv) {
5    int a = 5;
6    IncPtr(&a);   // caller: &a,  body: *p
7    IncRef(a);    // caller:  a,  body:  r
8    // a is now 7
9    return EXIT_SUCCESS;
10 }
```

- **Both functions do the same thing**: add 1 to the caller's a.
- **Pointer:** the caller writes **&a**, the body writes ***p**.
- **Reference:** the caller writes **a**, the body writes **r**. No stars, no ampersands.

const: a promise this will not change

```
1  const int kMaxHealth = 100;   // a promise: fixed
2
3  int hp = kMaxHealth;   // ok: reading is fine
4  kMaxHealth = 120;      // error: cannot modify
```

- **const** means "I promise not to modify this."
- Reading a const value is fine; writing to it is a compile error.
- We just used it as **const Image&**: pass without copying, promise not to change.
- Used far more in C++ than in C.

const pointer

```
1  Account acc   = {7, 100};
2  Account other = {9, 500};
3
4  Account* const p = &acc;  // a const POINTER
5  p->balance = 0;   // ok: change the data
6  p = &other;       // error: pointer is fixed
```

- A pointer touches two separate things:
  - **the data** it points at (**p->balance**)
  - **its own value** (which object, **p = …**)
- **Account* const p** locks the pointer itself.
  - **p->balance = 0** is fine (data is free)
  - **p = &other** is an error (pointer is fixed)

EXAM-STYLE Q

What compiles, and what changes?

```
1 int a = 5;
2 int& r = a;
3 const int* p = &a;
4 int* const q = &a;
5
6 r = 9;       // A
7 q = &a;      // B
8 *p = 7;      // C
```

QUESTION

After line A runs, what is the value of a?

Which of B and C compile, and which are errors? Why?

SOLUTION

What compiles, and what changes?

```
1 int a = 5;
2 int& r = a;
3 const int* p = &a;
4 int* const q = &a;
5
6 r = 9;       // A
7 q = &a;      // B
8 *p = 7;      // C
```

ANSWER

A: a becomes 9. r is another name for a, so writing r writes a.

B: error. q is a const pointer (int* const): the pointer cannot be reassigned.

C: error. p is a pointer to const int: you cannot write through it.

Initialization lists

```
1  // assignment in the body: default-build, then set
2  Point::Point(int x, int y) {
3    x_ = x;   // x_ was already built, now assigned
4    y_ = y;
5  }
6
7  // initialization list: build members directly
8  Point::Point(int x, int y) : x_(x), y_(y) { }
```

❖ The **: member(value)** list runs BEFORE the body.

❖ **Assignment version:** each member is default-built first, then overwritten. Two steps.

❖ **Init-list version:** each member is built directly from the value. One step.

❖ Required for **const** members and references, which cannot be assigned.

Members init in declaration order

```
1  class IntArray {
2   public:
3    explicit IntArray(size_t size)
4        : data_(new int[size_]),   // uses size_...
5          size_(size) { }          // ...set only here!
6   private:
7    int* data_;      // declared FIRST
8    size_t size_;    // declared SECOND
9  };
```

❖ The list says **data_(...)** then **size_(size)**, so it looks like size_ is ready.

❖ **It is not.** Members init in DECLARATION order: data_ FIRST, size_ second.

❖ So **new int[size_]** reads size_ before it is set. Garbage size, wild allocation.

❖ **Fix:** declare size_ before data_ (as our IntArray does), and list the initializers in that same order.

EXAM-STYLE Q

A member left uninitialized

```
1  class Timer {
2   public:
3    Timer(int secs) : count_(secs) { }
4    int Remaining() const { return limit_ - count_; }
5   private:
6    int limit_;
7    int count_;
8  };
9
10 Timer t(10);
11 cout << t.Remaining() << endl;
```

QUESTION

The constructor sets count_, but never sets limit_.

What does t.Remaining() return? Is it well defined?

How would you fix the constructor?

SOLUTION

A member left uninitialized

```
1  class Timer {
2   public:
3    Timer(int secs) : count_(secs) { }
4    int Remaining() const { return limit_ - count_; }
5   private:
6    int limit_;
7    int count_;
8  };
9
10 Timer t(10);
11 cout << t.Remaining() << endl;
```

ANSWER

limit_ is never initialized: it holds an indeterminate (garbage) value.

So Remaining() returns garbage minus 10. Undefined, unpredictable.

Fix: Timer(int secs) : limit_(secs), count_(0) { } so every member is initialized.

EXAM-STYLE Q

A member that default-constructs

Product.h

```
1 class Product {
2  public:
3   Product() : name_("unknown") { }
4   Product(string n) : name_(n) { }
5   string Name() const { return name_; }
6  private:
7   string name_;
8 };
```

Sale.h + usage

```
1 class Sale {
2  public:
3   Sale(int q) : quantity_(q) { }
4   Product product_;
5   int quantity_;
6 };
7
8 Sale s(5);
9 cout << s.product_.Name() << endl;
```

QUESTION

Sale's constructor initializes quantity_ but never mentions product_.

What does s.product_.Name() print, and why?

SOLUTION

A member that default-constructs

Product.h

```
1  class Product {
2   public:
3    Product() : name_("unknown") { }
4    Product(string n) : name_(n) { }
5    string Name() const { return name_; }
6   private:
7    string name_;
8  };
```

Sale.h + usage

```
1  class Sale {
2   public:
3    Sale(int q) : quantity_(q) { }
4    Product product_;
5    int quantity_;
6  };
7
8  Sale s(5);
9  cout << s.product_.Name() << endl;
```

ANSWER

product_ is default-constructed. The init list omits it, so C++ calls Product's default constructor.

That default sets name_ to "unknown".

So it prints: unknown

A member with no init-list entry gets its default constructor (or garbage, for a plain type like int).

class Derived : public Base

```
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, the C++ default

```
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 with virtual

```
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.

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  };
```

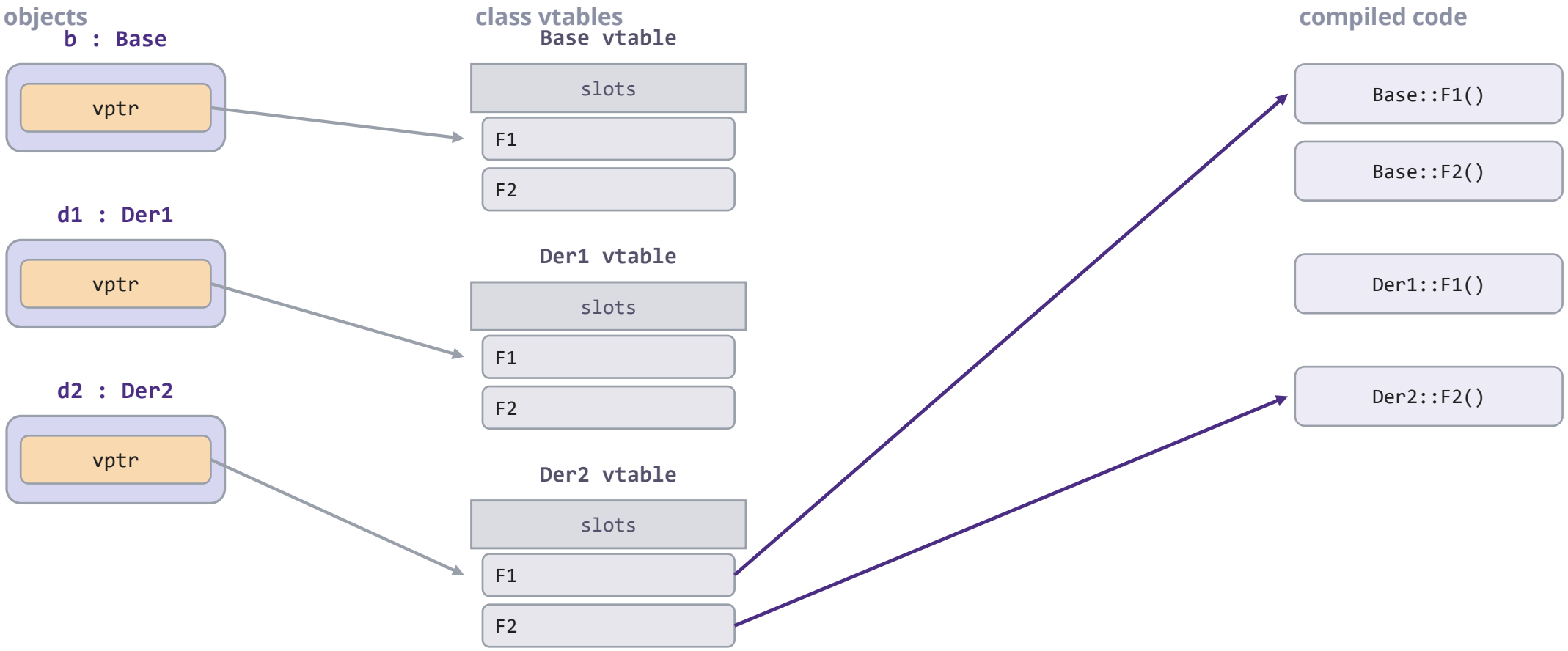
Two keywords

❖ **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.

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



EXAM-STYLE Q

Which version runs?

```
1  struct A {
2    void M1()         { cout << "A1 "; }
3    virtual void M2() { cout << "A2 "; }
4  };
5  struct B : A {
6    void M1()          { cout << "B1 "; }
7    void M2() override { cout << "B2 "; }
8  };
9
10 B b;
11 A* p = &b;
12 p->M1();
13 p->M2();
```

QUESTION

p is an A* pointing at a real B.

What does this print?

Which call is decided by the pointer type, and which by the object?

SOLUTION

Which version runs?

```
1  struct A {
2    void M1()         { cout << "A1 "; }
3    virtual void M2() { cout << "A2 "; }
4  };
5  struct B : A {
6    void M1()          { cout << "B1 "; }
7    void M2() override { cout << "B2 "; }
8  };
9
10 B b;
11 A* p = &b;
12 p->M1();
13 p->M2();
```

ANSWER

Prints **A1 B2**.

M1 is non-virtual: resolved by the pointer's type A, so A::M1.

M2 is virtual: resolved by the actual object B, so B::M2.

EXAM-STYLE Q

A derived class that overrides only some methods

```
1  struct Animal {
2    virtual void Speak()  { cout << "..."; }
3    virtual void Legs()   { cout << 4; }
4  };
5  struct Snake : Animal {
6    void Speak() override { cout << "hiss"; }
7  };
8
9  Animal* a = new Snake();
10 a->Speak();
11 a->Legs();
```

QUESTION

Snake overrides Speak but not Legs.

What do a->Speak() and a->Legs() each run and print?

SOLUTION

A derived class that overrides only some methods

```
1  struct Animal {
2    virtual void Speak()  { cout << "..."; }
3    virtual void Legs()   { cout << 4; }
4  };
5  struct Snake : Animal {
6    void Speak() override { cout << "hiss"; }
7  };
8
9  Animal* a = new Snake();
10 a->Speak();
11 a->Legs();
```

ANSWER

a->Speak(): Snake::Speak, prints "hiss".

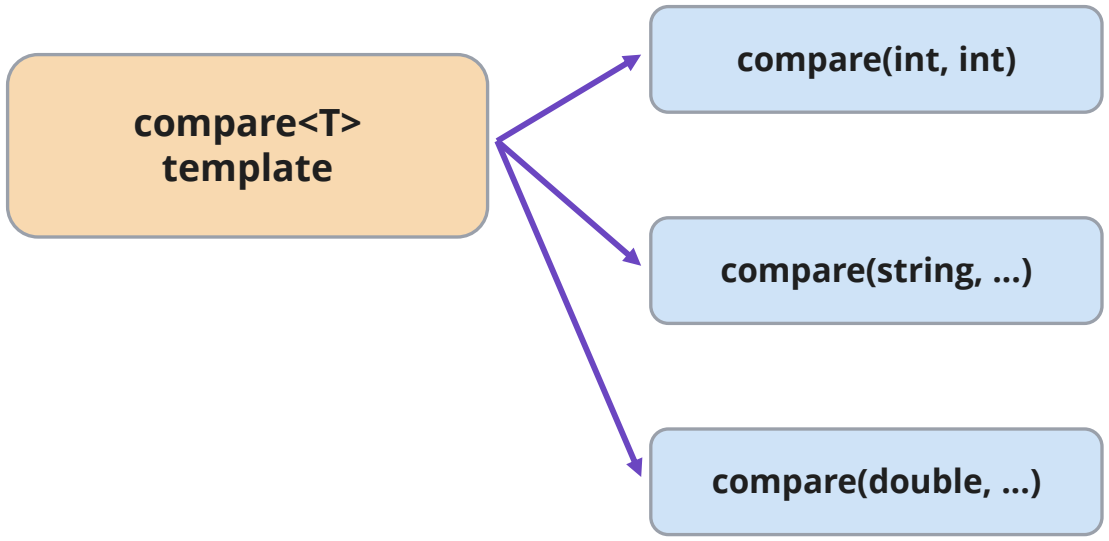
a->Legs(): not overridden, so the Snake vtable's Legs slot still points at Animal::Legs. Prints 4.

A derived class inherits the base version of any virtual it does not override.

A template is a recipe

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

The compilation problem

compare.h

```
1  template <typename T>
2  int compare(const T& a, const T& b);   // declaration only
```

compare.cc

```
1  template <typename T>
2  int compare(const T& a, const T& b) { /* ... */ }
```

main.cc

```
1  #include "compare.h"
2  compare(3, 4);   // undefined reference at link time
```

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.

Definitions go in the header

compare.h full definition in the header

```
1  template <typename T>
2  int compare(const T& a, const T& b) {
3    if (a < b) return -1;
4    if (b < a) return 1;
5    return 0;
6  }
```

compare.h declares, then #includes the .cc

```
1  template <typename T>
2  int compare(const T& a, const T& b);   // declar
3
4  #include "compare.cc"    // pulls the bodies in
```

EXAM-STYLE Q

Predict the output, or the error

```
1 template <typename T>
2 T Smaller(T a, T b) { return (a < b) ? a : b; }
3
4 cout << Smaller(4, 9)      << endl;
5 cout << Smaller('Z', 'A')  << endl;
6 cout << Smaller(3, 2.5)    << endl;
```

QUESTION

For each call, what does T become, and what prints?

Does any line fail to compile? Why?

SOLUTION

Predict the output, or the error

```
1 template <typename T>
2 T Smaller(T a, T b) { return (a < b) ? a : b; }
3
4 cout << Smaller(4, 9)       << endl;
5 cout << Smaller('Z', 'A')   << endl;
6 cout << Smaller(3, 2.5)     << endl;
```

ANSWER

Smaller(4, 9): T is int, prints 4.

Smaller('Z', 'A'): T is char, prints the letter A.

Smaller(3, 2.5): will not compile. T cannot be both int and double.

Fix: Smaller<double>(3, 2.5), or make both arguments the same type.

Designing good software questions

I may also ask you some short answer question to argue some things.

- Should a parameter be const?
- What functions should be public or private in C++?
- Should a global variable be static or extern?
- How many modules should we decompose a problem into?
- When should we use the C stdlib or the POSIX API to read a file?
- Etc…

Reminders

Assessments:

- Exercise 6 will be released today. Due Thursday at 11:59pm
- Please show up to the midterm tomorrow in lecture!

General:

- Mid-quarter feedback this week
  - Anonymous, but submission is tracked. 1 participation pt for filling it out.
  - Be honest and critical! I can't improve without clear issues to work on ☹

Questions?

Thanks for coming - see you next lecture!