LECTURE 12 · CSE 333 · Summer 2026

C++: inheritance + templates

Discussion: What's the purpose of an exam? What makes an exam bad? Or good?

Reminders

Assessments:

- Exercise 5 is due Wednesday at 11:59pm
- HW2 is due Thursday at 11:59pm

General:

- Mid-quarter feedback this week
  - Anonymous, but I see whether you completed it after 1 week
  - 1 participation pt for filling it out
  - Be honest and critical! I can't improve without clear issues to work on ☹
- 2 lectures before the midterm
  - Personalized exams?

Review

The Six Special Members

Point.c (what you see)

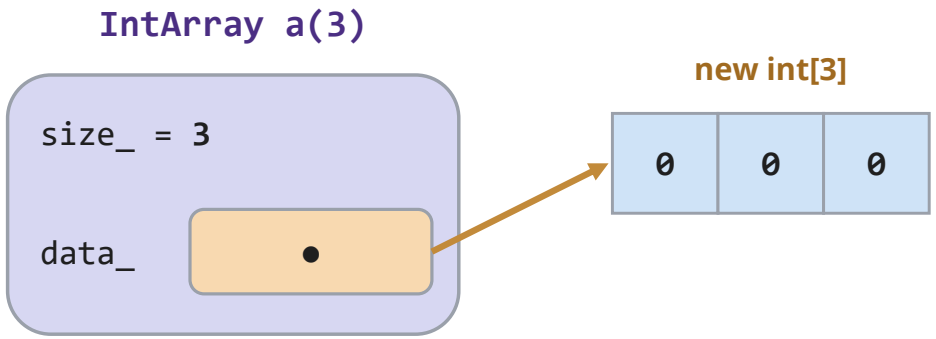
```
1  class Point {
2   private:
3    int x_;   // data member
4    int y_;
5  };
6
7
8
9
10
11
12
13
14
15
16
17
```

Point.c (what you get)

```
1  class Point {
2    Point() = default;                                  // constructor
3    Point(const Point& other) = default;                // copy constructor
4    Point& operator=(const Point& other) = default;     // copy assignment
5    Point(Poin&t&& other) = default;                    // move constructor
6    Point& operator=(Point&& other) = default;          // move assignment
7    ~Point() = default;                                 // destructor
8
9   private:
10   int x_;   // data member
11   int y_;
12 };
13
14
15
16
17
```

#1: Constructor

```
1  class IntArray {
2   public:
3    explicit IntArray(size_t size);
4    size_t Size() const { return size_; }
5    int At(size_t i) const { return data_[i]; }
6    void Set(size_t i, int v) { data_[i] = v; }
7   private:
8    size_t size_;    // declared FIRST
9    int* data_;      // heap buffer we OWN
10 };
11
12 IntArray::IntArray(size_t size) : size_(size) {
13   data_ = new int[size];
14 }
```



❖ The object holds a **pointer**, not the array. The array lives on the heap.

❖ **The ctor acquires the buffer** with new int[size] -- but nothing frees it yet.

❖ So: who calls **delete[]** on that buffer, and when?

#2: Destructor

```
1  class IntArray {
2   public:
3    explicit IntArray(size_t size);  // acquire
4    ~IntArray();                     // release
5    // ...
6   private:
7    int* data_;
8    size_t size_;
9  };
10
11 IntArray::~IntArray() {
12   delete[] data_;   // give the heap back
13 }
```

❖ **~IntArray()** runs automatically when the object dies.

- a stack object: at the end of its scope
- a heap object: when you delete it

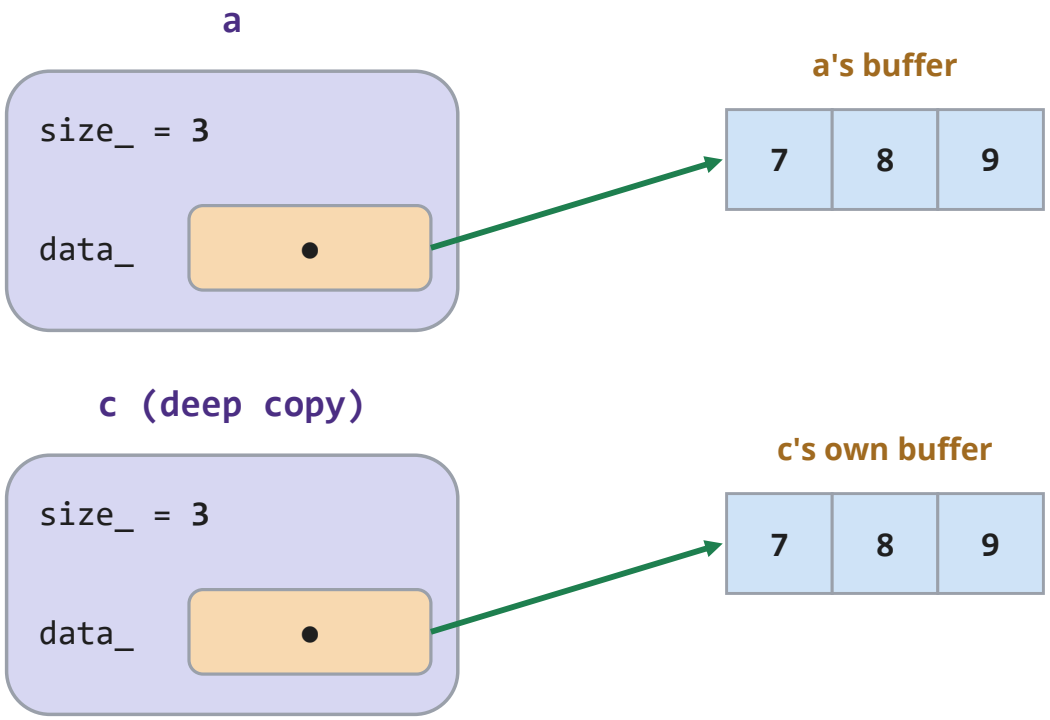
❖ **Its job:** release what the object owns, here **delete[] data_**. No return type, no params, one per class.

❖ **The private buffer can't leak:** cleanup is automatic, no delete[] at the call site.

#3: Copy Constructor

```
1  IntArray::IntArray(const IntArray& other)
2      : size_(other.size_) {
3    data_ = new int[size_];         // our own buffer
4    for (size_t i = 0; i < size_; i++) {
5      data_[i] = other.data_[i];    // copy the values
6    }
7  }
```

❖ **Allocate a fresh buffer,** then copy the elements over. Each object owns its own array.



#4: Copy Assignment

```
1  IntArray& IntArray::operator=(const IntArray& rhs) {
2    if (this != &rhs) {           // 1. self-assign?
3      delete[] data_;             // 2. free old buffer
4      size_ = rhs.size_;
5      data_ = new int[size_];     // 3. deep copy
6      for (size_t i = 0; i < size_; i++) {
7        data_[i] = rhs.data_[i];
8      }
9    }
10   return *this;                 // 4. enable chaining
11 }
```

- **Construction** builds a NEW object.
- **Assignment** overwrites one that ALREADY owns a buffer.
- So there is more to do. We walk the four steps next.

#5: Move

```
1  // move constructor: steal the buffer
2  IntArray::IntArray(IntArray&& other)
3      : size_(other.size_), data_(other.data_) {
4    other.data_ = nullptr;   // leave the source empty
5    other.size_ = 0;
6  }
7
8  // move assignment: free old, then steal
9  IntArray& IntArray::operator=(IntArray&& rhs) {
10   if (this != &rhs) {
11     delete[] data_;        // free our old buffer
12     size_ = rhs.size_;
13     data_ = rhs.data_;     // steal the pointer
14     rhs.data_ = nullptr;   // null the source
15     rhs.size_ = 0;
16   }
17   return *this;
18 }
```

❖ **Both steal** rhs's pointer, then null it so its destructor frees nothing.

❖ **No new, no element copy:** a move is a few pointer assignments.

❖ **Big Three suppressed these.** Declaring the dtor and copy ops turned off the compiler's move members; you opt back in by writing them.

❖ **= default** would be wrong: it copies data_ without nulling the source, so both free it -> double free.

Rule of Zero, Three, and Five

Rule of Zero

- **Own no resource** (like Point).
- All six defaults are already correct.
- Write **none** of the special members.

Rule of Three

- **Own a resource** (like IntArray).
- Write the three that manage it:
  - destructor
  - copy constructor
  - copy assignment

Rule of Five

- The Big Three, plus two for speed:
  - move constructor
  - move assignment
- **Declaring the Three** removed the free moves; add them back.

=default and =delete

```
1  class IntArray {
2   public:
3    IntArray() = default;   // opt back into the
4                            // synthesized default ctor
5
6    // ban copying entirely:
7    IntArray(const IntArray&) = delete;
8    IntArray& operator=(const IntArray&) = delete;
9  };
```

❖ **= default** asks for the compiler's synthesized version explicitly, so you keep it even after declaring other ctors.

❖ **= delete** removes a function. Calling it is a compile error.

❖ **Deleting the copy ctor and operator=** makes a type non-copyable, which prevents the accidental shallow copies we just saw.

❖ This is how unique-ownership types are built.

Operator Overloading

Operators are functions with special names

```
class IntArray {
 public:
  int& operator[](size_t i) { return data_[i]; }
  // ...
};

IntArray a(3);
a[0] = 7;           // write:  calls operator[]
int first = a[0];   // read:   calls operator[]
```

```
// operator== is just a function with a special name
bool operator==(const Point& a, const Point& b) {
  return a.GetX() == b.GetX()
      && a.GetY() == b.GetY();
}

Point p(1, 2), q(1, 2);
if (p == q) { ... }   // the compiler calls operator==
```

- **operator[]** makes **a[i]** read and write elements.
- It returns a reference, so a[0] = 7 assigns.
- **operator==** defines what **p == q** means.
- You already met one: **operator=** (assignment), coming up next.

Inheritance

Duplicated code across classes

asset #1

Stock

```
symbol_
total_shares_
total_cost_
current_price_
```

```
GetMarketValue()
GetProfit()
GetCost()
```

asset #2

DividendStock

```
symbol_
total_shares_
total_cost_
current_price_
dividends_
```

```
GetMarketValue()
GetProfit()
GetCost()
```

asset #3

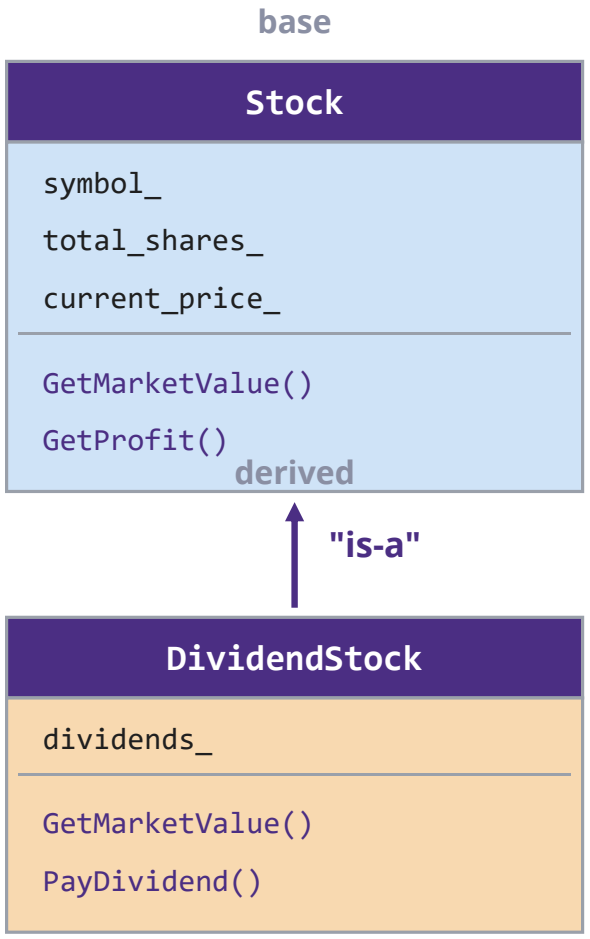
Cash

```
amount_
```

```
GetMarketValue()
```

- **Duplicated** fields and methods across Stock and DividendStock.
- **No common type:** you cannot make one vector holding a mix of assets.
- Adding an asset means copy-pasting the shared parts again.

Base and derived classes



Java	C++
Superclass	Base class
Subclass	Derived class
extends	: public Base

Why inherit?

- **Code reuse:** the derived class inherits the base's fields and methods.
- **Polymorphism:** redefine behavior while keeping the same interface.
- **Extensibility:** the derived class adds new state and behavior.

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.

A base pointer to a derived object

```
1  #include "Stock.h"
2  #include "DividendStock.h"
3
4  DividendStock dividend;
5  DividendStock* ds = &dividend;   // exact type
6  Stock* s = &dividend;            // base ptr to derived
7  // s exposes the Stock interface; the ACTUAL type of
8  // *s may decide which version actually runs.
```

PromisedType* p = new ActualType

- In Java: **Base var = new Derived();**
- In C++: **Base* p = new Derived();** (references work too)
- **ActualType** must be the same or a derived class of the promised type.
- **The pointer's type** sets what you can call; the actual object may decide which version runs.

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.

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  };
```

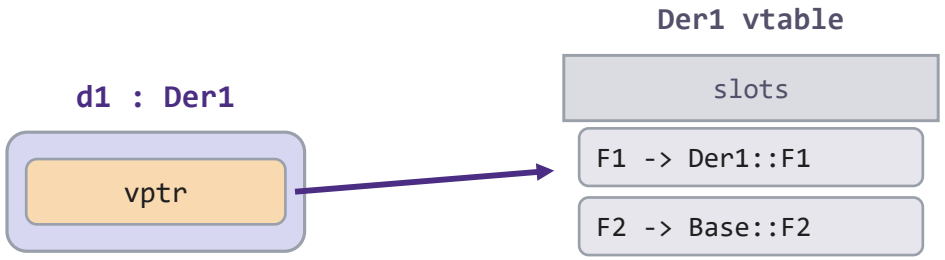
Two keywords

- **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.

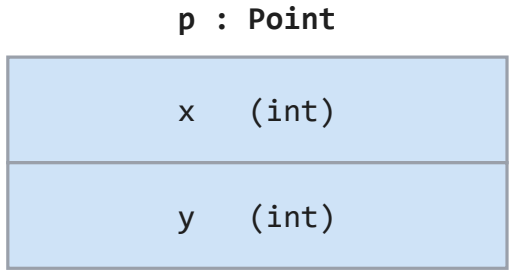


the vptr 'remembers' the object's class

Memory of a struct vs an object

C: a plain struct

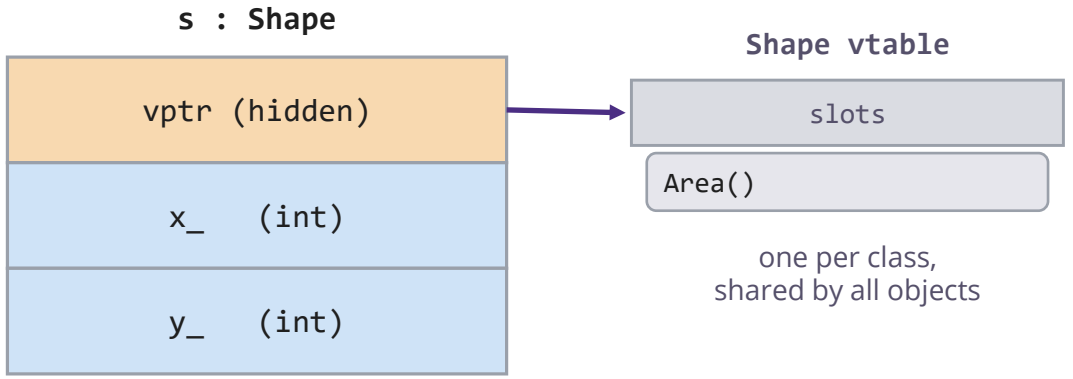
```
1  struct Point {
2    int x;
3    int y;
4  };
```



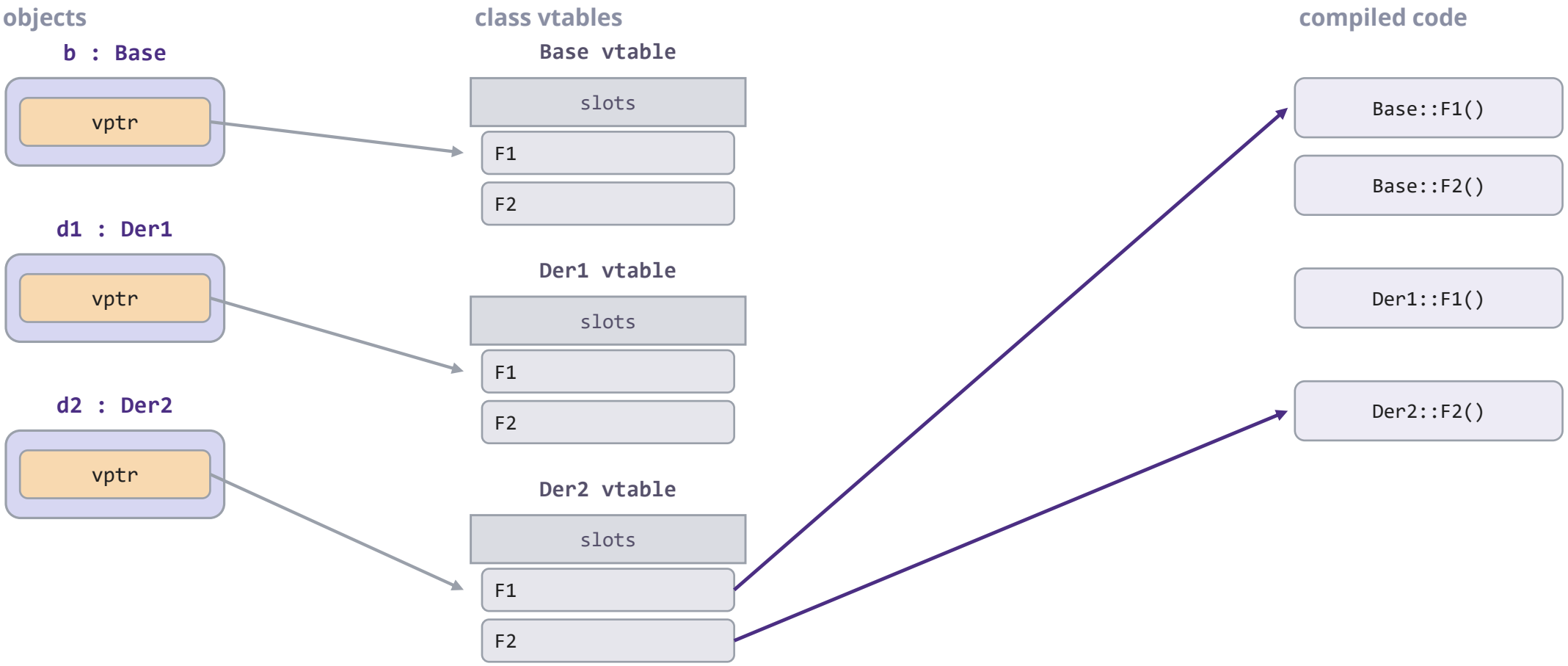
Members sit inline, in declaration order. That is the whole object.

C++: an object with a virtual method

```
1  class Shape {
2   public:
3    virtual double Area() const;
4    int x_;
5    int y_;
6  };
```



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.

M1 static, M2 virtual

```
1  class A {
2   public:
3    void M1() { cout << "a1, "; }          // static
4    virtual void M2() { cout << "a2"; }    // dynamic
5  };
6  class B : public A {
7   public:
8    void M1() { cout << "b1, "; }
9    void M2() override { cout << "b2"; }
10 };
```

call	prints	resolved by
a_ptr_a->M1()	a1,	A* type
a_ptr_a->M2()	a2	object A
a_ptr_b->M1()	a1,	A* type
a_ptr_b->M2()	b2	object B
b_ptr_b->M1()	b1,	B* type
b_ptr_b->M2()	b2	object B

a_ptr_b is an A* aimed at a B: M1 static -> a1, but M2 dynamic -> b2.

Takeaway: Non-virtual M1 follows the pointer's type; virtual M2 follows the actual object. One object, two dispatch rules, depending on the method.

Base constructed first

```
1  class Base {                    // no default ctor
2   public:
3    Base(int y) : y_(y) { }
4    int y_;
5  };
6  class Der1 : public Base {      // ERROR: needs Base()
7   public:
8    int z_;
9  };
10 class Der2 : public Base {      // OK
11  public:
12   Der2(int y, int z) : Base(y), z_(z) { }
13   int z_;
14 };
```

How construction chains

- ❖ **Base part first:** the base constructor runs before the derived body.
- ❖ **Pick the base ctor** in the derived initializer list: Der2(int y, int z) : Base(y), z_(z).
- ❖ **No default base ctor** and you do not call one? Compiler error (see Der1).
- ❖ Constructors are not inherited; Der writes its own.

Takeaway: The base part is built first. Name the base constructor in the derived initializer list, or you rely on the base's default, which may not exist.

Virtual destructors

```
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
```

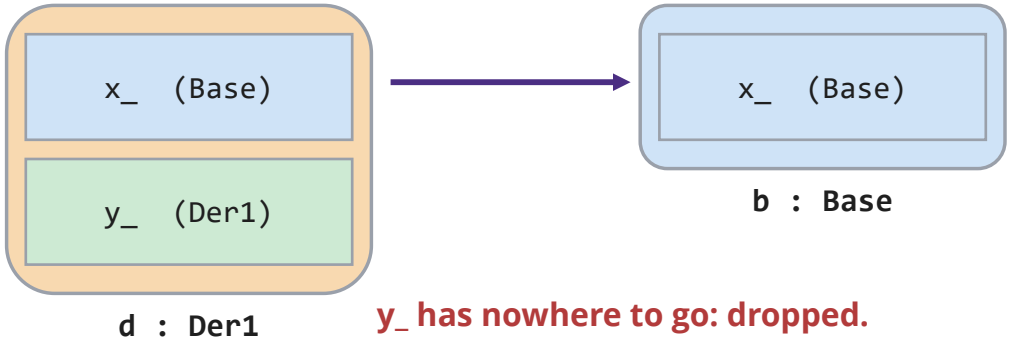
Destruction runs top-down: ~Der1 then ~Base. A non-virtual base dtor called through a Base* skips ~Der1, so y_ leaks (formally undefined behavior). Marking ~Base virtual fixes it.

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"

```
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_.

Example: vector<Stock> slices derived types

```
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.

Casting

C-style casts

```
1  double d = 3.9;
2  int n = (int) d;              // truncates 3.9 -> 3
3
4  int x = 42;
5  char* p = (char*) &x;         // reinterpret the bits
6
7  // One blunt syntax for very different operations:
8  // numeric conversion vs pointer reinterpretation.
9  // Hard to search for, easy to do the wrong thing.
```

Why C++ moved on

- **(type) expr** does everything: truncate a double, reinterpret a pointer, strip const.
- **Intent is hidden:** the reader cannot tell a safe numeric cast from a dangerous pointer reinterpret.
- **Unsearchable:** you cannot grep for '(type)' meaningfully.
- **No safety check** beyond what the compiler must allow.

Say what you mean

static_cast<T>(e)

- **Compile-time** conversion between related pointers or numeric types.
- Example: **static_cast<int>(3.9)**, or up a hierarchy. Down-casts are unchecked and risky.

dynamic_cast<T>(e)

- **Run-time checked** down-cast; needs a polymorphic base (a virtual method).
- Example: **dynamic_cast<Der1*>(bp)** returns nullptr if bp is not really a Der1.

const_cast<T>(e)

- **Adds or removes const** (or volatile), nothing else.
- Example: **const_cast<int*>(p)**. Writing through a stripped const can be UB. Use rarely.

reinterpret_cast<T>(e)

- **Raw bit reinterpretation** between unrelated types (pointer <-> int, incompatible pointers).
- Example: **reinterpret_cast<int*>(addr)**. Dangerous; used carefully in hw3.

static_cast and dynamic_cast

```
1  double d = 3.9;
2  int n = static_cast<int>(d);          // 3: numeric
3
4  Stock* s = portfolio[i];              // base pointer
5  // Is it really a DividendStock?  Ask at run time:
6  DividendStock* ds = dynamic_cast<DividendStock*>(s);
7  if (ds != nullptr) {
8    ds->PayDividend(1.0);   // safe: it really is one
9  }
```

The two you actually use

❖ **static_cast<T>(e):** compile-time conversion between numeric types or related pointers (e.g. up a hierarchy).

❖ **dynamic_cast<T>(e):** run-time CHECKED down-cast. Needs a polymorphic base (a virtual method); returns nullptr if the object is not really a T.

The other two are rare red flags. **const_cast** adds or strips const; **reinterpret_cast** reinterprets raw bits between unrelated types (used carefully in hw3). Both should make a reader pause.

Implicit conversions from constructors

```
1  class Foo {
2   public:
3    Foo(int x) : x_(x) { }   // one-arg = conversion path
4    int x_;
5  };
6  int Bar(Foo f) { return f.x_; }
7
8  int main(int argc, char** argv) {
9    return Bar(5);   // becomes Bar(Foo(5)) implicitly!
10 }
```

```
1  class Foo {
2   public:
3    explicit Foo(int x) : x_(x) { }   // no implicit path
4    int x_;
5  };
6  int Bar(Foo f) { return f.x_; }
7
8  int main(int argc, char** argv) {
9    return Bar(5);   // compiler error (good!)
10 }
```

A single-argument constructor doubles as a conversion path, so Bar(5) silently becomes Bar(Foo(5)). At most one user-defined conversion is applied. Mark such constructors **explicit** to turn that off.

Templates

The same function, three times

```
1  int compare(const int& a, const int& b) {
2    if (a < b) return -1;
3    if (b < a) return 1;
4    return 0;
5  }
6  int compare(const string& a, const string& b) {
7    if (a < b) return -1;      // exact same body
8    if (b < a) return 1;
9    return 0;
10 }
11 int compare(const double& a, const double& b) {
12   if (a < b) return -1;      // ...and again
13   if (b < a) return 1;
14   return 0;
15 }
```

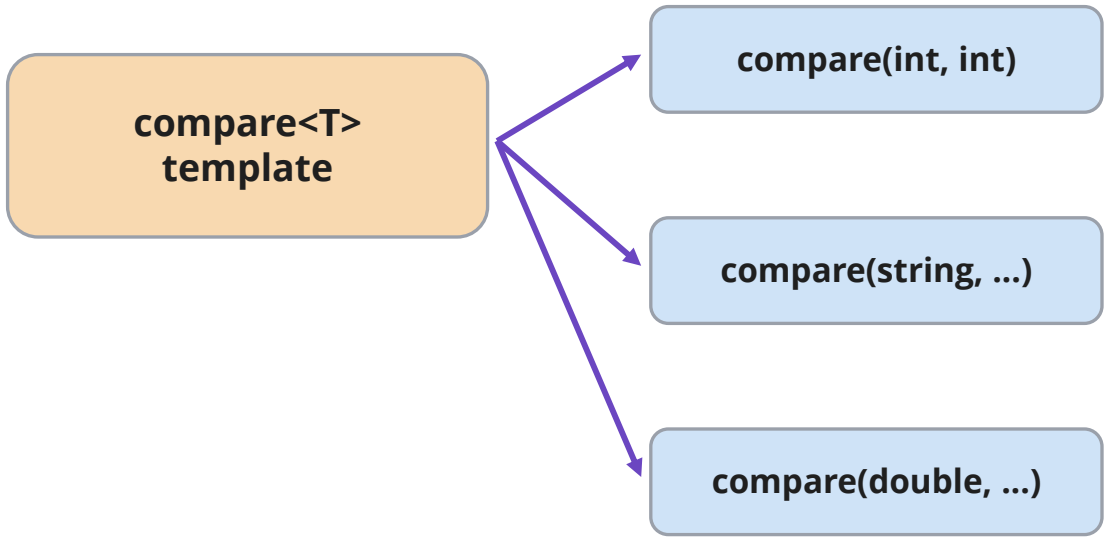
This is all boilerplate

- We want a **compare** for many types.
- Overloading forces one copy per type: int, string, double, …
- Every body is identical; only the type name changes.
- The logic only needs **<** to work on the type.
- **We want** generic (type-independent) code.

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

Writing a function template

```
1  // T is a type parameter, filled in by the compiler
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  }
```

The syntax

- **template <typename T>** introduces a type parameter named T.
- Inside, **T** is used like any other type.
- **typename** and **class** are interchangeable here.

- One definition replaces all three overloads. It works for any type T that supports **operator<**.
- Nothing is compiled yet: no code exists until someone CALLS compare with a concrete type.

Explicit vs inferred template type

```
1  compare<int>(10, 20);          // explicit:  T = int
2  compare(10, 20);               // inferred:  T = int
3  compare(3.14, 2.72);           // inferred:  T = double
4
5  string s1 = "Hello", s2 = "World";
6  compare(s1, s2);               // inferred:  T = string
7
8  compare("Hello", "World");     // T = const char*  !
9                                 // compares POINTERS, not text
```

Type inference

- **compare<int>(...)**: you name T.
- **compare(10, 20)**: compiler INFERS T from the arguments.
- **Watch string literals:** "Hello" is a const char*, so T = const char*.
- **That compares** pointer addresses, not the text.

Takeaway: Usually you let the compiler infer T from the arguments. The classic trap is passing string literals, where T becomes const char* and you compare addresses instead of characters.

The compilation problem

```
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.

Definitions go in the header

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.

Practice: function template output

```
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;   // ?
7  cout << smaller("pear", "fig") << endl;  // ?
```

What single type does T become in each call?

Predict first: for each call, write down T and the exact output before we reveal it.

Practice: function template output

```
1  template <typename T>
2  T smaller(T a, T b) { return (a < b) ? a : b; }
3
4  cout << smaller(4, 9)          << endl;   // int
5  cout << smaller('Z', 'A')      << endl;   // char
6  cout << smaller(3, 2.5)        << endl;   // compiler error
7  cout << smaller("pear", "fig") << endl;  // const char*
```

Fix for compiler error:

- Cast to a specific type
- Manually specify type by writing: `smaller<double>(3, 2.5)`

Practice: class template output

```
1  template <typename T>
2  class Box {
3   public:
4    Box(T v) : val_(v) { }
5    T Get() const { return val_; }
6   private:
7    T val_;
8  };
9
10 Box<double> a(7);              // T = ?
11 Box<int>    b(7);              // T = ?
12 cout << a.Get() / 2 << endl;   // ?
13 cout << b.Get() / 2 << endl;   // ?
```

Predict first: predict both printed lines, then decide whether a and b come from the same class.

Practice: class template output

```
1  template <typename T>
2  class Box {
3   public:
4    Box(T v) : val_(v) { }
5    T Get() const { return val_; }
6   private:
7    T val_;
8  };
9
10 Box<double> a(7);              // T = double
11 Box<int>    b(7);              // T = int
12 cout << a.Get() / 2 << endl;   // 3
13 cout << b.Get() / 2 << endl;   // 3.5
```

Predict first: predict both printed lines, then decide whether a and b come from the same class.

STL

A class template: Pair

```
1  template <typename T>
2  class Pair {
3   public:
4    Pair(const T& a, const T& b) : first_(a), second_(b) { }
5    T First()  const { return first_; }
6    T Second() const { return second_; }
7    void Swap() { T t = first_; first_ = second_; second_ = t;
8   private:
9    T first_, second_;    // members have the parameter type
10 };
```

Classes take T too

- **template <typename T>** sits above the whole class.
- Members can have type **T**: first_ and second_.
- Every method here is **inline**, so it is part of the template.
- **Swap()** swaps the two members in place.

Using Pair

```
1  Pair<string> names("Alice", "Bob");
2  names.Swap();
3  cout << names.First() << endl;   // Bob
4
5  Pair<int> point(3, 4);           // a fresh instantiation
6  cout << point.Second() << endl;  // 4
```

Instantiation on demand

- **Pair<string>** tells the compiler to stamp out that class.
- **Pair<int>** makes a second, unrelated class.
- You supply T; the compiler does the rest.

Takeaway: A class template is a factory for classes. You pick the type in the angle brackets and the compiler generates the whole class for it. This is exactly how vector, list, and map work.

Practice: function template output

```
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;   // ?
7  cout << smaller("pear", "fig") << endl;   // ?
```

Watch the types

- What single type does T become in each call?
- Does a char stay a char when it prints?
- Can one of these refuse to compile? Why?
- What are **"pear"** and **"fig"**, really?

Predict first: for each call, write down T and the exact output before we reveal it.

Practice: class template output

```
1  template <typename T>
2  class Box {
3   public:
4    Box(T v) : val_(v) { }
5    T Get() const { return val_; }
6   private:
7    T val_;
8  };
9
10 Box<double> a(7);            // int arg, T = double
11 Box<int>    b(7);
12 cout << a.Get() / 2 << endl;   // ?
13 cout << b.Get() / 2 << endl;   // ?
```

Watch the types

- **Box<double> a(7)**: what does val_ hold?
- Is **Get() / 2** integer or real division?
- Do a and b share one class, or two?

Predict first: predict both printed lines, then decide whether a and b come from the same class.

Containers store by value

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.

One copy per push_back

```
1  std::vector<Tracer> v;
2  v.reserve(3);          // no reallocation
3  Tracer t;              // ctor
4  v.push_back(t);        // copy ctor
5  v.push_back(t);        // copy ctor
6  v.push_back(t);        // copy ctor
```

```
$ ./a.out

ctor                <- Tracer t;

copy ctor           <- push_back(t)

copy ctor           <- push_back(t)

copy ctor           <- push_back(t)
```

The vector keeps its OWN copy of every element, so each **push_back** runs the copy constructor. Remove the **reserve(3)** and a growing vector copies the existing elements into the new buffer on top of that.

Takeaway: Storing by value means copying: every push_back invokes the element's copy constructor, and reallocation copies again. This is why big objects belong behind a pointer.

vector: a resizable array

```
1  #include <vector>
2  std::vector<Tracer> v;
3  Tracer t;                // ctor
4  v.push_back(t);          // copy ctor: t is COPIED in
5  v.push_back(t);          // copy ctor again
6                           // a resize may copy the old
7                           // elements into the new block
```

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.

- Run the Tracer version and you will see **copy ctor** print on every push_back, plus extra copies when the vector outgrows its buffer and reallocates.

Iterators

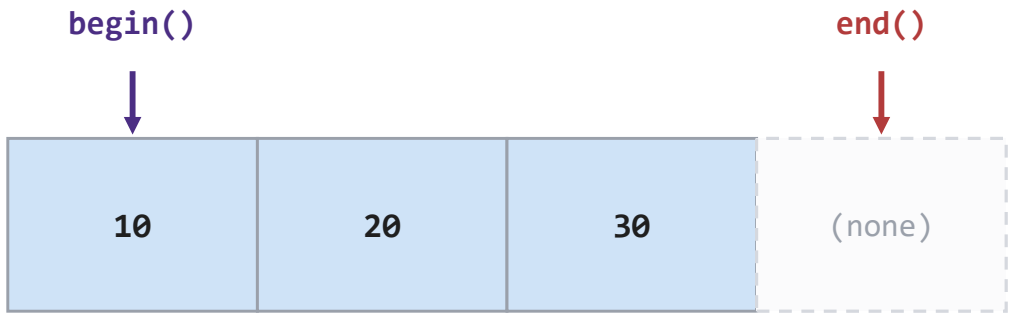
```
1  std::vector<int> v = {10, 20, 30};
2
3  // every container defines its own iterator type
4  std::vector<int>::iterator it;
5  for (it = v.begin(); it != v.end(); ++it) {
6    cout << *it << endl;   // * dereferences the iterator
7  }
```

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.

auto and range-for

```
1  // the verbose iterator loop
2  for (std::vector<int>::iterator it = v.begin();
3       it != v.end(); ++it) { cout << *it; }
4
5  // auto infers the iterator type (C++11)
6  for (auto it = v.begin(); it != v.end(); ++it) { }
7
8  // range-for hides the iterator entirely
9  for (auto& x : v) {
10   cout << x << endl;
11 }
```

C++11 conveniences

- **auto** asks the compiler to infer the type.
- Great for iterator types, which are long to spell out.
- **range-for**: for (auto& x : v) visits each element.
- Use **auto&** to avoid copying each element; **const auto&** to also forbid edits.

Takeaway: auto infers the iterator type and range-for hides the iterator entirely. Prefer for (const auto& x : v) so you walk the container without copying every element.

Algorithms work through iterators

```
1  #include <algorithm>
2
3  std::sort(v.begin(), v.end());  // uses element's operator<
4
5  auto it = std::find(v.begin(), v.end(), 20);
6  if (it != v.end()) { /* found */ }
7
8  std::for_each(v.begin(), v.end(), Print);   // call Print(x)
```

Ranges, not containers

- Algorithms take an **iterator range [begin, end)**, not a container.
- **sort** uses the element's **operator<**.
- **find** returns an iterator, or end() if absent.
- **for_each** calls your function on every element.

- Same algorithm, any container: because they speak iterators, sort and find work on a vector's range, a plain array's pointers, and more.

list: a doubly-linked list

```
1  #include <list>
2  std::list<int> lst = {3, 1, 2};
3
4  lst.push_front(0);   // O(1) at either end or middle
5  lst.sort();          // member sort: SPLICES nodes,
6                       // it does not copy the elements
7
8  // lst[2] does NOT compile: no random access
```

list<T> vs vector<T>

- **Doubly-linked:** nodes, not a contiguous block.
- **No random access:** there is no lst[i].
- **O(1)** insert/erase anywhere, given an iterator.
- Member **sort()** relinks nodes (splices); it does NOT copy elements.

Takeaway: list trades vector's random access for O(1) insertion anywhere and a sort that rewires pointers instead of copying elements. Pick the container by the operations you need.

map: ordered key to value

```
1  #include <map>
2  std::map<string, int> ages;       // key -> value, kept sor
3  ages["Alice"] = 30;           // insert or update
4  ages["Bob"]   = 25;
5
6  auto it = ages.find("Alice");
7  if (it != ages.end())
8    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)`

Takeaway: 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).

map: ordered key to value

```
1  #include <map>
2  std::map<string, int> ages;       // key -> value, kept sor
3  ages["Alice"] = 30;           // insert or update
4  ages["Bob"]   = 25;
5
6  auto it = ages.find("Alice");
7  if (it != ages.end())
8    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)`

Takeaway: 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).

Same idea as Java, different rules

You already know (Java)	The C++ STL twist
`ArrayList<Foo>`	**vector<Foo>** stores Foo **values**, not references: inserting copies.
`HashMap (unordered)`	**map** is a **sorted tree**, O(log n). **unordered_map** is the hash map.
`map.get(x) -> null`	**m[x]** default-constructs and **inserts** a missing key.
`list.get(i) is checked`	**v[i]** doesn't check whether the index is in bound. Use **v.at(i)** for a bounds check.
`ConcurrentMod.Exception`	mutating while iterating is **undefined behavior**, no exception.

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
6  }
7  // freq: {blue: 1, red: 2}   (kept sorted by key)
```

Example #3: unique

```
1  std::vector<int> v = {3, 1, 3, 2, 1, 2, 3};
2
3  // a set keeps unique keys, kept sorted
4  std::set<int> uniq(v.begin(), v.end());   // {1, 2, 3}
5
6  // or in place: sort, then unique + erase
7  std::sort(v.begin(), v.end());
8  v.erase(std::unique(v.begin(), v.end()), v.end());   // {1, 2, 3}
```

Example #4: accumulate and max_element

```
#include <numeric>     // for std::accumulate
std::vector<int> v = {4, 8, 15, 16, 23, 42};

int total = std::accumulate(v.begin(), v.end(), 0);
// total == 108

auto it = std::max_element(v.begin(), v.end());
int biggest = *it;     // biggest == 42
```

Example #5: a custom hash

```
1  struct Point { int x, y; };
2
3  // a custom key needs a hash AND an equality
4  struct PtHash {
5    std::size_t operator()(const Point& p) const {
6      return std::hash<int>()(p.x) * 31 + std::hash<int>()(p.y);
7    }
8  };
9  bool operator==(const Point& a, const Point& b) {
10   return a.x == b.x && a.y == b.y;
11 }
12
13 std::unordered_map<Point, std::string, PtHash> label;
14 label[{1, 2}] = "start";
```

Reminders

Assessments:

- Exercise 5 is due Wednesday at 11:59pm
- HW2 is due Thursday at 11:59pm

General:

- Mid-quarter feedback this week
  - Anonymous, but I see whether you completed it after 1 week
  - 1 participation pt for filling it out
  - Be honest and critical! I can't improve without clear issues to work on ☹
- 2 lectures before the midterm
  - Personalized exams?

Questions?

Thanks for coming - see you next lecture!