UNIVERSITY of WASHINGTON

LECTURE 13 · CSE 333 · Summer 2026

C++: inheritance + templates (pt2)

Discussion: What's a really cool hobby that you admire but don't do?

Reminders

Assessments:

- Exercise 5 is due today at 11:59pm
- HW2 is due tomorrow 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 ☹

- 1 lectures before the midterm
  - Will have “personalized exams” (details next lecture)

Review

REVIEW

The Six Special Members

Point.c (what you see)

```
class Point {
 private:
  int x_;   // data member
  int y_;
};
```

Point.c (what you get)

```
class Point {
  Point() = default;                                   // constructor
  Point(const Point& other) = default;                 // copy constructor
  Point& operator=(const Point& other) = default;      // copy assignment
  Point(Poin&t&& other) = default;                     // move constructor
  Point& operator=(Point&& other) = default;           // move assignment
  ~Point() = default;                                  // destructor

 private:
  int x_;   // data member
  int y_;
};
```

REVIEW

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[]
```

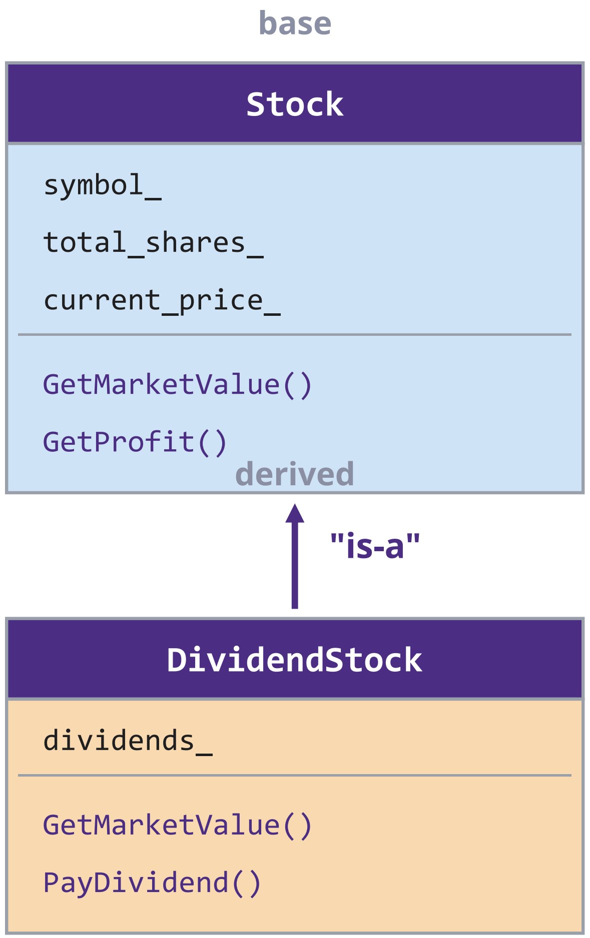
```
// operator== is just a function with a special name
bool operator==(const Point& a, const Point& b) {
  return a.GetX() == b.GetX()
      && a.GetY() == b.GetY();
}

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.

You can look up the predefined function signatures for other operators on cppreference

REVIEW

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

```cpp
#include "Stock.h"

class DividendStock : public Stock {    // is-a Stock
 public:
  DividendStock(...);                   // NOT inherited
  double GetMarketValue() const override;
  void PayDividend(double amount);      // new behavior
 private:
  double dividends_;                    // new state
};
```

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.

REVIEW

Static dispatch, the C++ default

```cpp
class Stock {
 public:
  double GetMarketValue() const;         // NOT virtual
};
class DividendStock : public Stock {
 public:
  double GetMarketValue() const;         // hides the base
};

DividendStock dividend;
Stock* s = &dividend;     // base ptr, derived object
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.

REVIEW

virtual and override

```
class Stock {
 public:
  virtual double GetMarketValue() const;   // opt in
  double GetProfit() const;                // inherited
};
class DividendStock : public Stock {
 public:
  double GetMarketValue() const override; // Google style
};
```

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.

Dynamic dispatch with virtual

```cpp
class Stock {
 public:
  virtual double GetMarketValue() const;    // opt in
  double GetProfit() const;                 // inherited
};
class DividendStock : public Stock {
 public:
  double GetMarketValue() const override;
};

Stock* s = &dividend;   // still a Stock*
s->GetMarketValue();    // DividendStock::GetMarketValue
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.

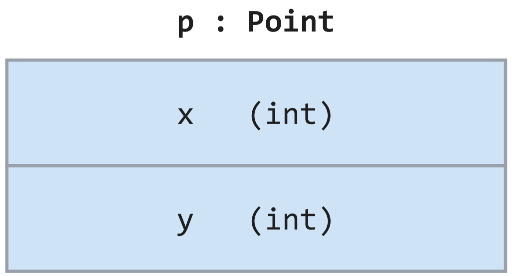
Takeaway: Dynamic dispatch picks the most-derived version from the actual object, even through a base pointer or from inside inherited code.

REVIEW

Memory of a struct vs an object

C: a plain struct

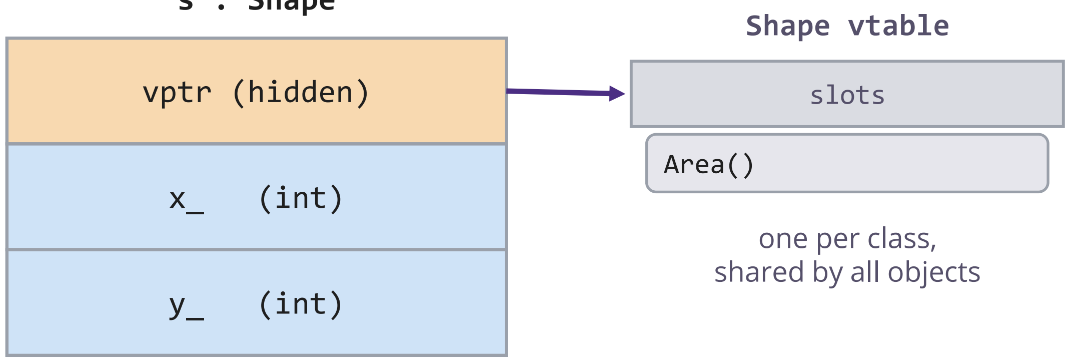
```c
struct Point {
  int x;
  int y;
};
```



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

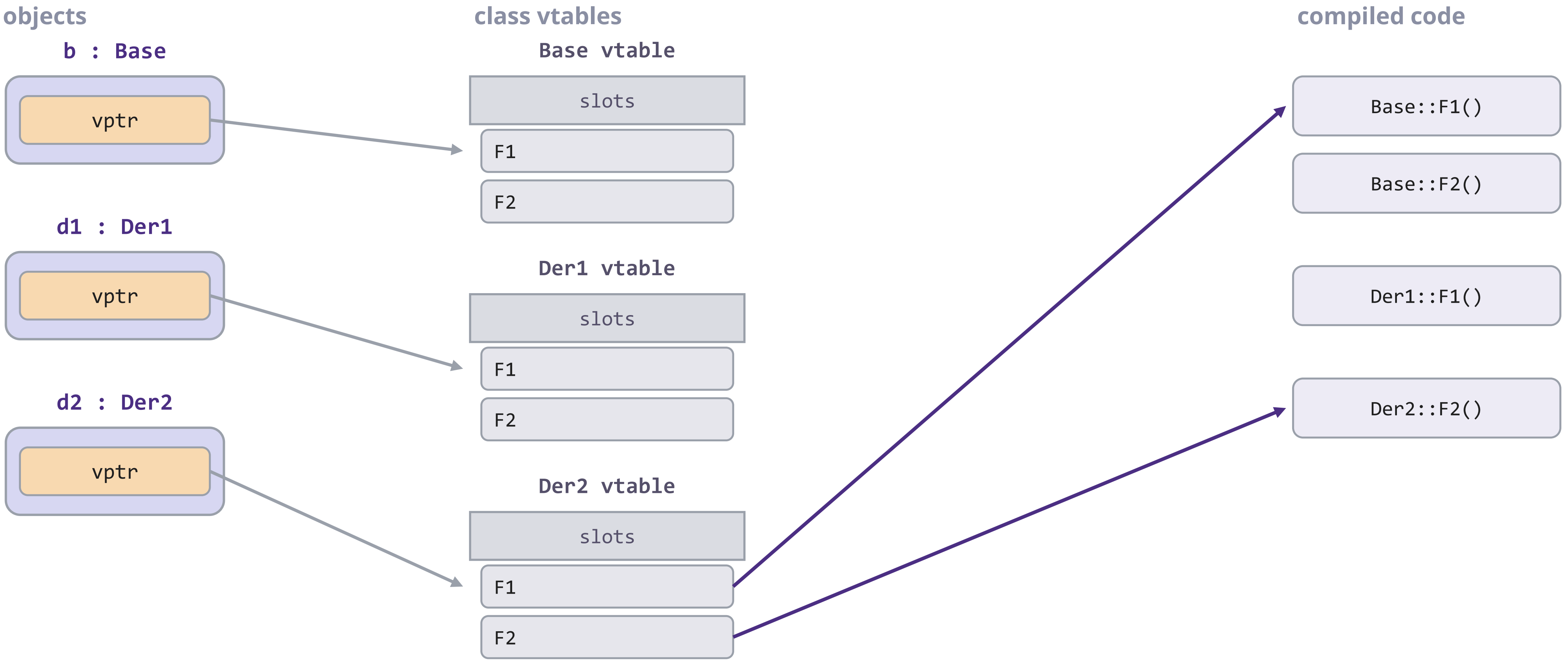
C++: an object with a virtual method

```cpp
class Shape {
 public:
  virtual double Area() const;
  int x_;
  int y_;
};
```



REVIEW

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.

More Inheritance

Tricky!

```
class Base {
 public:
  Base(int x) : x_(x) { }
  int x_;
};

class Der1 : public Base {
 public:
  Der1(int y) : Base(16), y_(y) { }
  int y_;
};

void Foo() {
  Base b(1);
  Der1 d(2);
  b = d;         // What gets called here?
}
```

What happens here:
- Constructor?
- Function?
- Operator?

Where is that defined?

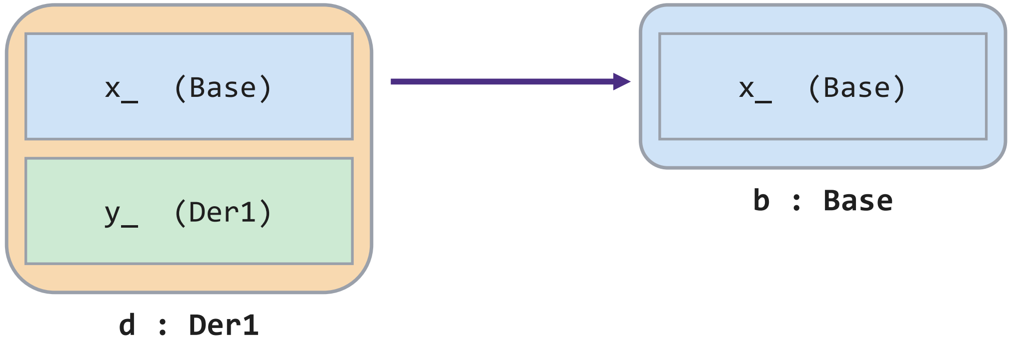
What will happen?

NEW (NOT REVIEW)

Object "slicing"

```cpp
class Base {
 public:
  Base(int x) : x_(x) { }
  int x_;
};

class Der1 : public Base {
 public:
  Der1(int y) : Base(16), y_(y) { }
  int y_;
};

void Foo() {
  Base b(1);
  Der1 d(2);
  b = d;    // SLICED: copies x_, drops y_
}
```

b = d; copies only the Base part



y_ has nowhere to go: dropped.

We call the **copy assignment** of the Base class. That’s the default copy assignment so it will only copy **x**.

Example: vector<Stock> slices derived types

```cpp
#include <vector>
#include <memory>
#include "Stock.h"
#include "DividendStock.h"

std::vector<Stock> v;          // stores Stock BY VALUE
DividendStock ds;
v.push_back(ds);               // OUCH: sliced to a Stock

// Fix: store pointers (ideally smart pointers)
std::vector<std::unique_ptr<Stock> > v2;
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

```
double d = 3.9;
int n = (int) d;                // truncates 3.9 -> 3

int x = 42;
char* p = (char*) &x;           // reinterpret the bits

// One blunt syntax for very different operations:
// numeric conversion vs pointer reinterpretation.
// 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.

CASTING

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

```
double d = 3.9;
int n = static_cast<int>(d);          // 3: numeric

Stock* s = portfolio[i];              // base pointer
// Is it really a DividendStock?  Ask at run time:
DividendStock* ds = dynamic_cast<DividendStock*>(s);
if (ds != nullptr) {
  ds->PayDividend(1.0);   // safe: it really is one
}
```

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

```cpp
class Foo {
 public:
  Foo(int x) : x_(x) { }   // one-arg = conversion path
  int x_;
};
int Bar(Foo f) { return f.x_; }

int main(int argc, char** argv) {
  return Bar(5);   // becomes Bar(Foo(5)) implicitly!
}
```

```cpp
class Foo {
 public:
  explicit Foo(int x) : x_(x) { }   // no implicit path
  int x_;
};
int Bar(Foo f) { return f.x_; }

int main(int argc, char** argv) {
  return Bar(5);   // compiler error (good!)
}
```

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

```cpp
int compare(const int& a, const int& b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
int compare(const string& a, const string& b) {
  if (a < b) return -1;      // exact same body
  if (b < a) return 1;
  return 0;
}
int compare(const double& a, const double& b) {
  if (a < b) return -1;      // ...and again
  if (b < a) return 1;
  return 0;
}
```

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“ for the compiler

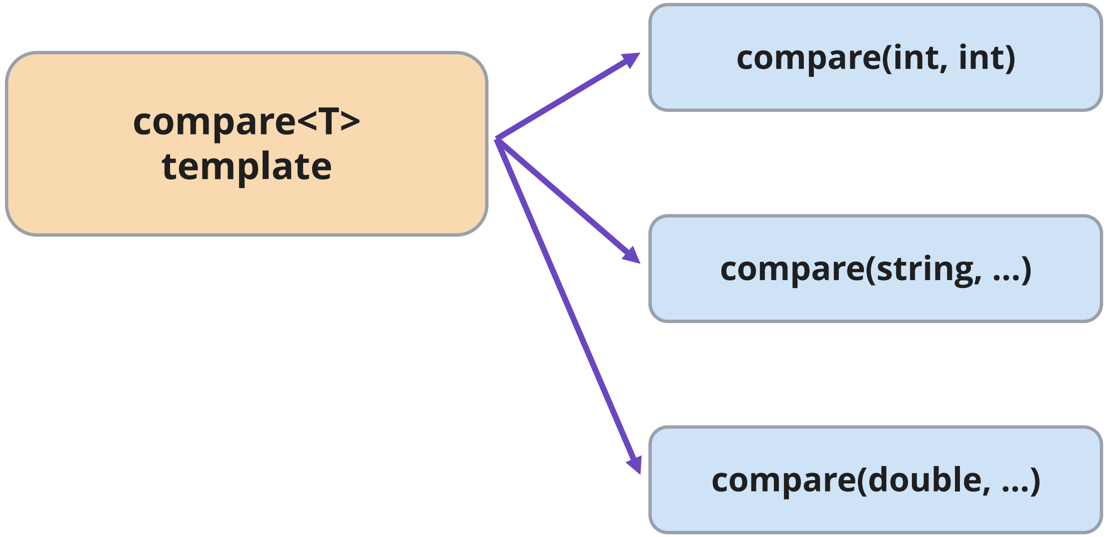
Parametric polymorphism

❖ 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

```
// T is a type parameter, filled in by the compiler
template <typename T>
int compare(const T& a, const T& b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
```

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

```
compare<int>(10, 20);          // explicit:  T = int
compare(10, 20);               // inferred:  T = int
compare(3.14, 2.72);           // inferred:  T = double

string s1 = "Hello", s2 = "World";
compare(s1, s2);               // inferred:  T = string

compare("Hello", "World");      // T = const char*  !
                               // 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

```cpp
// compare.h
template <typename T>
int compare(const T& a, const T& b);    // declaration

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

// main.cc
#include "compare.h"
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

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

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

```
// compare.h
template <typename T>
int compare(const T& a, const T& b);

#include "compare.cc"   // pull definitions

// 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

```cpp
template <typename T>
T smaller(T a, T b) { return (a < b) ? a : b; }

cout << smaller(4, 9)           << endl;   // ?
cout << smaller('Z', 'A')       << endl;   // ?
cout << smaller(3, 2.5)         << endl;   // ?
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

```
template <typename T>
T smaller(T a, T b) { return (a < b) ? a : b; }

cout << smaller(4, 9)           << endl;   // int
cout << smaller('Z', 'A')       << endl;   // char
cout << smaller(3, 2.5)         << endl;   // compiler error
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

```
template <typename T>
class Box {
 public:
  Box(T v) : val_(v) { }
  T Get() const { return val_; }
 private:
  T val_;
};

Box<double> a(7);              // T = ?
Box<int>    b(7);              // T = ?
cout << a.Get() / 2 << endl;   // ?
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

```
template <typename T>
class Box {
 public:
  Box(T v) : val_(v) { }
  T Get() const { return val_; }
 private:
  T val_;
};

Box<double> a(7);              // T = double
Box<int>    b(7);              // T = int
cout << a.Get() / 2 << endl;   // 3
cout << b.Get() / 2 << endl;   // 3.5
```

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

A class template: Pair

```cpp
template <typename T>
class Pair {
 public:
  Pair(const T& a, const T& b) : first_(a), second_(b) { }
  T First()  const { return first_; }
  T Second() const { return second_; }
  void Swap() { T t = first_; first_ = second_; second_ = t; }
 private:
  T first_, second_;    // members have the parameter type
};
```

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

```cpp
Pair<string> names("Alice", "Bob");
names.Swap();
cout << names.First() << endl;   // Bob

Pair<int> point(3, 4);           // a fresh instantiation
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

```cpp
template <typename T>
T smaller(T a, T b) { return (a < b) ? a : b; }

cout << smaller(4, 9)           << endl;    // ?
cout << smaller('Z', 'A')       << endl;    // ?
cout << smaller(3, 2.5)         << endl;    // ?
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

```cpp
template <typename T>
class Box {
 public:
  Box(T v) : val_(v) { }
  T Get() const { return val_; }
 private:
  T val_;
};

Box<double> a(7);             // int arg, T = double
Box<int>    b(7);
cout << a.Get() / 2 << endl;   // ?
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.

STL

(Standard Template Library)

The standard library, in three pieces

Containers: hold your data

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

Examples

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

Iterators

Algorithms

A container is an object that owns a collection of elements by value, and vector, list, and map each store them with a different structure and cost.

The standard library, in three pieces

Containers: hold your data

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

Examples

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

Iterators

Algorithms: operate on data

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

Examples

```
sort( ... );
find( ..., x );
for_each( ..., f );
```

Algorithms like sort and find operate on your elements, but notice they never name a container. What goes in the parentheses?

The standard library, in three pieces

Containers: hold your data

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

Examples

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

Iterators: the bridge

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

Examples

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

Algorithms: operate on data

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

Examples

```
sort( ... );
find( ..., x );
for_each( ..., f );
```

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

Java Collections vs. C++ containers

You already know (Java)	C++ STL equivalent
`ArrayList<T>`	**vector<T>** - growable array, stores values.
`LinkedList<T>`	**list<T>** - doubly-linked list.
`ArrayDeque<T>`	**deque<T>** - double-ended queue.
`HashMap<K,V>`	**unordered_map<K,V>** - hash table, average O(1).
`TreeMap<K,V>`	**map<K,V>** - sorted tree, O(log n).
`HashSet<T>`	**unordered_set<T>** - hashed unique keys.
`TreeSet<T>`	**set<T>** - sorted unique keys.

vector: a resizable array

```cpp
#include <vector>
std::vector<T> v;
T thing1, thing2;            // ctor
v.push_back(thing1);
v.push_back(thing2);


```

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 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.

```
class Tracer {
 public:
  Tracer()                     { cout << "ctor\n"; }
  Tracer(const Tracer& t) { cout << "copy ctor\n"; }
  Tracer& operator=(const Tracer& t) {
    cout << "operator=\n";  return *this;
  }
  ~Tracer()                    { cout << "dtor\n"; }
};
```

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

One copy per push_back

```cpp
std::vector<Tracer> v;
v.reserve(3);             // no reallocation
Tracer t;                 // ctor
v.push_back(t);           // copy ctor
v.push_back(t);           // copy ctor
v.push_back(t);           // copy ctor
```

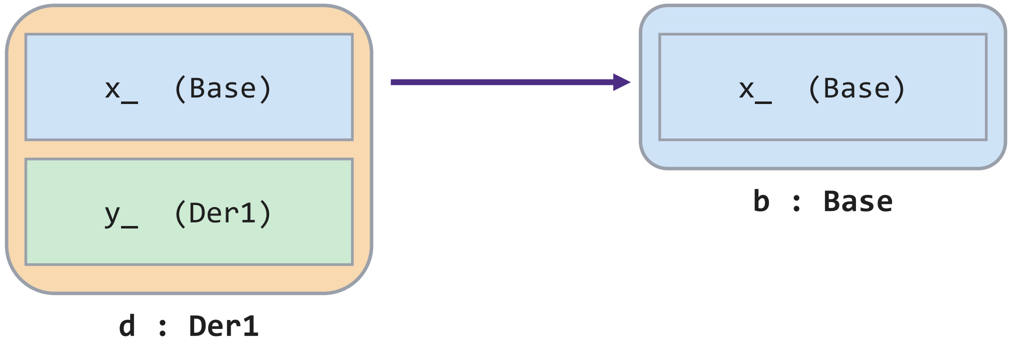
```
$ ./a.out

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.

Recall: Object "slicing"

```cpp
class Base {
 public:
  Base(int x) : x_(x) { }
  int x_;
};

class Der1 : public Base {
 public:
  Der1(int y) : Base(16), y_(y) { }
  int y_;
};

void Foo() {
  Base b(1);
  Der1 d(2);
  b = d;    // SLICED: copies x_, drops y_
}
```

b = d; copies only the Base part



y_ has nowhere to go: dropped.

We call the **copy assignment** of the Base class. That's the default copy assignment so it will only copy **x**.

Object “slicing” in STL

```
#include <vector>
#include <memory>
#include "Stock.h"
#include "DividendStock.h"

std::vector<Stock> v;          // stores Stock BY VALUE
DividendStock ds;
v.push_back(ds);               // OUCH: sliced to a Stock

// Fix: store pointers (ideally smart pointers)
std::vector<std::unique_ptr<Stock> > v2;
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

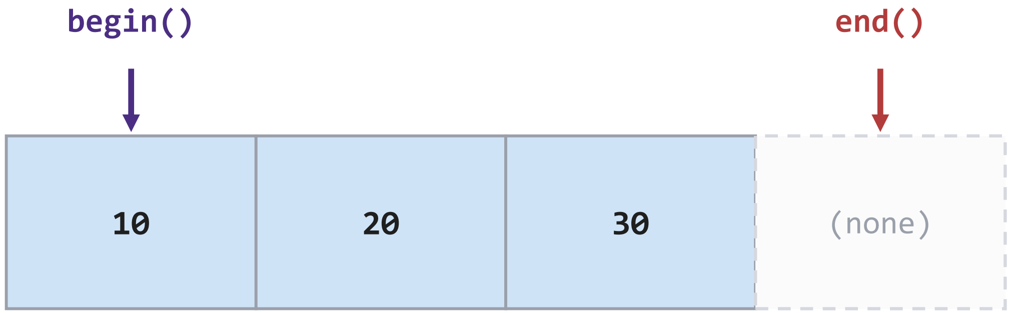
```cpp
std::vector<int> v = {10, 20, 30};

// every container defines its own iterator type
std::vector<int>::iterator it;
for (it = v.begin(); it != v.end(); ++it) {
  cout << *it << endl;    // * dereferences the iterator
}
```

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

```
// the verbose iterator loop
for (std::vector<int>::iterator it = v.begin();
     it != v.end(); ++it) { cout << *it; }

// auto infers the iterator type (C++11)
for (auto it = v.begin(); it != v.end(); ++it) { }

// range-for hides the iterator entirely
for (auto& x : v) {
  cout << x << endl;
}
```

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

```cpp
#include <algorithm>

std::sort(v.begin(), v.end());  // uses element's operator<

auto it = std::find(v.begin(), v.end(), 20);
if (it != v.end()) { /* found */ }

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

```cpp
#include <list>
std::list<int> lst = {3, 1, 2};

lst.push_front(0);    // O(1) at either end or middle
lst.sort();           // member sort: SPLICES nodes,
                      // it does not copy the elements

// 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.

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

```
#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
```

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

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.

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

```cpp
#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
```

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

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.

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.

Example #1: sort

```
std::vector<std::string> names = {"Bob", "Alice", "Eve", "Dan"};

std::sort(names.begin(), names.end());
// names: Alice  Bob  Dan  Eve

std::sort(names.begin(), names.end(), std::greater<std::string>());
// names: Eve  Dan  Bob  Alice
```

Example #2: count frequencies

```cpp
std::vector<std::string> words = {"red", "blue", "red"};
std::map<std::string, int> freq;

for (const std::string& w : words) {
  freq[w]++;          // [] inserts 0 the first time
}
// freq: {blue: 1, red: 2}   (kept sorted by key)
```

Example #3: unique

```cpp
std::vector<int> v = {3, 1, 3, 2, 1, 2, 3};

// a set keeps unique keys, kept sorted
std::set<int> uniq(v.begin(), v.end());   // {1, 2, 3}

// or in place: sort, then unique + erase
std::sort(v.begin(), v.end());
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

```
struct Point { int x, y; };

// a custom key needs a hash AND an equality
struct PtHash {
  std::size_t operator()(const Point& p) const {
    return std::hash<int>()(p.x) * 31 + std::hash<int>()(p.y);
  }
};
bool operator==(const Point& a, const Point& b) {
  return a.x == b.x && a.y == b.y;
}

std::unordered_map<Point, std::string, PtHash> label;
label[{1, 2}] = "start";
```

Reminders

Assessments:

- Exercise 5 is due today at 11:59pm
- HW2 is due tomorrow 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 ☹

- 1 lectures before the midterm
  - Will have “personalized exams” (details next lecture)

Questions?

Thanks for coming - see you next lecture!