UNIVERSITY of WASHINGTON

LECTURE 15 · CSE 333 · Summer 2026

# STL and Smart Pointers

**Discussion:** If you could go back in time and change one decision in your life, what would it be?

# Reminders

**Assessments**:

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

**General**:

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

# Review

# Motivation for templates

```
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**

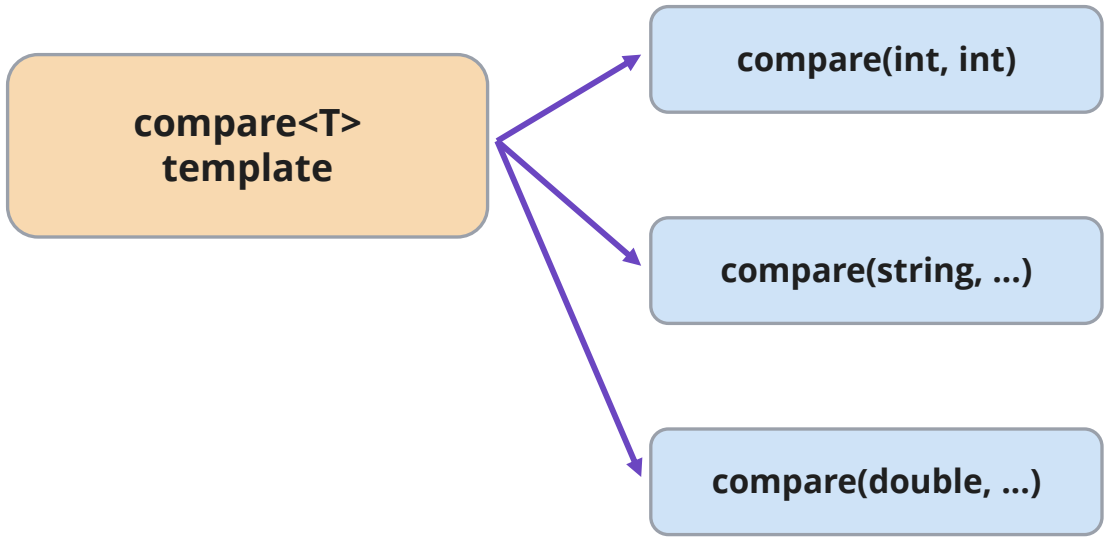
- ❖ We want a **compare** for many types.
- ❖ Overloading forces one copy per type: int, string, double, …
- ❖ Every body is identical; only the type name changes.
- ❖ The logic only needs **<** to work on the type.
- ❖ **We want** generic (type-independent) code.

REVIEW

# 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

```
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.

# 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
```

# 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.

# 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**

```
1  vector<int> v;
2  list<string> lst;
3  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**

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

Iterators

## Algorithms: operate on data

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

**Examples**

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

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

## Iterators: the bridge

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

**Examples**

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

## Algorithms: operate on data

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

**Examples**

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

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

# 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

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

**vector<T>**

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

# Containers 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
```

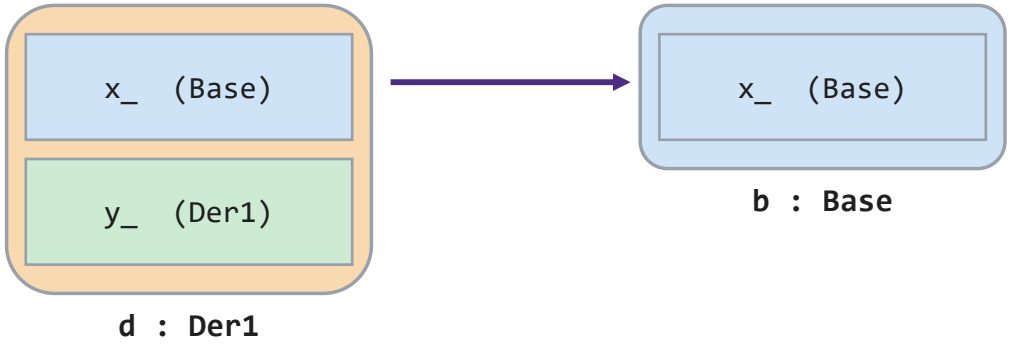
```
$ ./a.out

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.

# Recall: Object "slicing"

```
1  class Base {
2   public:
3    Base(int x) : x_(x) { }
4    int x_;
5  };
6
7  class Der1 : public Base {
8   public:
9    Der1(int y) : Base(16), y_(y) { }
10   int y_;
11 };
12
13 void Foo() {
14   Base b(1);
15   Der1 d(2);
16   b = d;   // SLICED: copies x_, drops y_
17 }
```

**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

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

### Containers copy by value

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

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

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

# Iterators

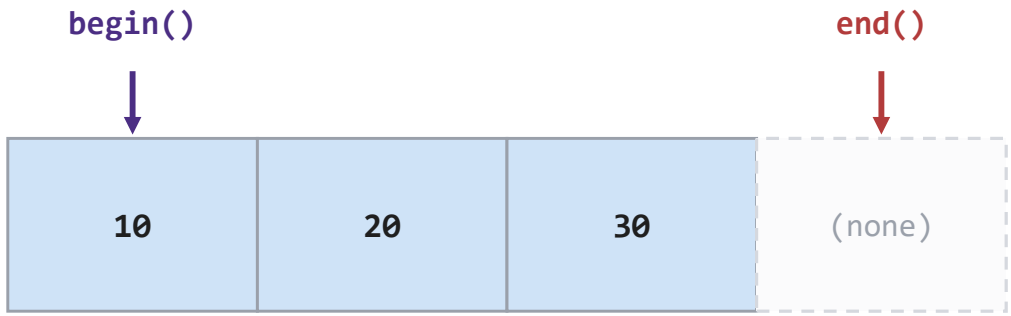
```
1  std::vector<int> v = {10, 20, 30};
2
3  // every container defines its own iterator type
4  std::vector<int>::iterator it;
5  for (it = v.begin(); it != v.end(); ++it) {
6    cout << *it << endl;   // * dereferences the iterator
7  }
```

### What an iterator is

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

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

# The range [begin, end)




### Why one-past-the-end?

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

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

# 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.

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
```

**map<Key, Value>**

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

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

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

# 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)`

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

```
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";
```

# STL Challenges

STL CHALLENGE

# A browser's Back button

### The scenario

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

### Discuss with a neighbor

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

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

# A browser's Back button

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

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

## Why / tradeoffs

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

STL CHALLENGE

# A music playlist

### The scenario

- Users build a playlist and constantly **reorder it**: drag a song up, delete one, insert another.
- Edits happen **anywhere in the middle**, not just the ends.
- Playback walks **song to song** (next / previous).
- You rarely jump to "song #427" by index.

### Discuss with a neighbor

- What operation DOMINATES here?
- What does a mid-list insert cost in a **vector?** In a **list?**
- What do you GIVE UP by leaving vector?
- Is random access actually needed?

**Predict first:** constant middle insert / erase / reorder, neighbor-to-neighbor playback. Sound familiar?

# A music playlist

**Use: std::list<Song> (a doubly-linked list)**

```
1  list<Song> playlist;
2  auto it = find(playlist.begin(), playlist.end(), cur);
3  playlist.insert(it, next);   // O(1): relink
4  playlist.erase(it);          // O(1): relink
5  // no playlist[i]; walk with iterators.
```

## Why / tradeoffs

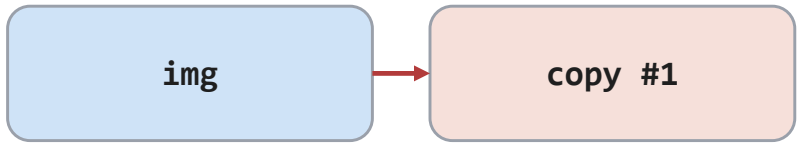
- Middle insert / erase is **O(1)** given an iterator - just relink.
- A **vector** shifts every later element: O(n) per edit.
- Playback walks neighbors, so no random access needed.
- Cost you accept: **no playlist[i]**, worse cache behavior.

# Smart Pointers

# Containers store by value

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

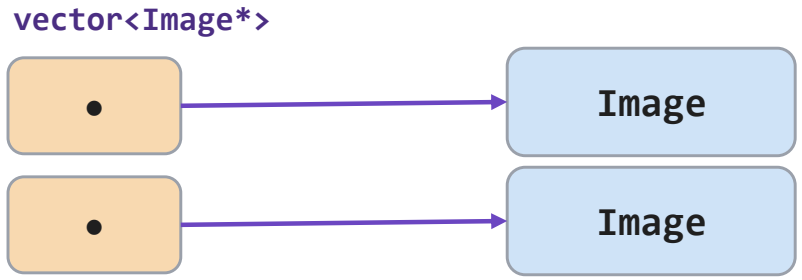
push_back copies INTO the container:




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

# So we store pointers instead

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




copying the vector copies addresses,
not pixels - cheap!

# ...but who deletes the Images?


**The container frees its cells, not your heap objects**

- The vector's destructor destroys the **pointers** it holds.
- **It has** no idea it should delete what they point at.
- **Every Image is now** unreachable and never freed: a leak.
- Raw pointers **carry no ownership information.**

# Deleting it manually

```
1  void render() {
2    Image* img = new Image("a.png");
3    if (!img->ok()) {
4      return;          // LEAK: img never deleted
5    }
6    draw(img);         // may THROW -> also leaks
7    delete img;        // only reached on the happy path
8  }
```

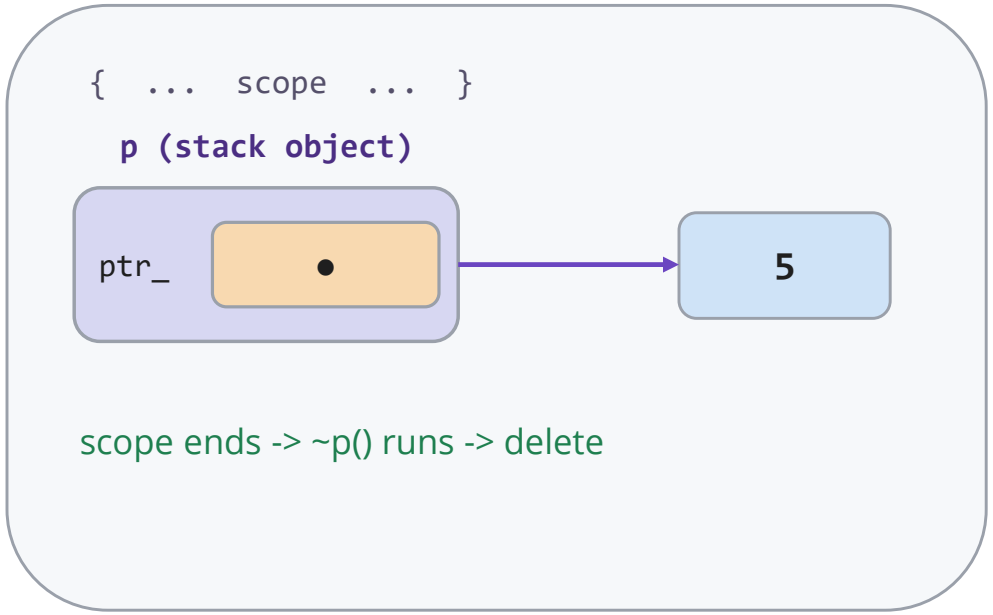
**Manual delete does not compose**

- An early **return** skips the delete -> leak.
- An exception thrown by draw() skips it too -> leak.
- **Delete it twice on two paths ->** double free.
- Every **new** needs a matching delete on **EVERY** path.

# Let a local object own it

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

  (resource allocation is initialization)



# Suggestion: ToyPtr

```
1  template <typename T> class ToyPtr {
2   public:
3    ToyPtr(T* p) : ptr_(p) { }   // adopt the pointer
4    ~ToyPtr() { delete ptr_; }   // free it for us
5
6   private:
7    T* ptr_;
8  };
```

**Just the Rule of Three**

- The **constructor** takes a raw pointer and adopts it.
- The **destructor** deletes it - so scope-exit frees it.
- It is a **template**, so it wraps a pointer to any type.
- So far it just holds and frees. Next: let us use it.

# Suggestion: ToyPtr

```
1  template <typename T> class ToyPtr {
2   public:
3    ToyPtr(T* p) : ptr_(p) { }
4    ~ToyPtr() { delete ptr_; }
5    T& deref() const { return *ptr_; }  // read/write
6    T* get()   const { return ptr_; }   // raw pointer
7
8   private:
9    T* ptr_;
10 };
```

**Access through methods**

- **deref()** hands back the pointee, so you can read or write it.
- **get()** returns the raw pointer for member access.
- It still **owns and frees** the object on scope exit.
- No new operators yet, just two **ordinary methods.**

# Example Usage #1

```
1  struct Node { int val; };
2
3  {
4    ToyPtr<Node> p(new Node{5});
5    Node& reference = p.deref();
6    cout << reference.val;   // read a field
7    reference.val = 10;      // write through
8    cout << p.get()->val;    // via the raw pointer
   }   // p leaves scope -> Node deleted
```

**It works!**

- ❖ You never touch **new** or **delete** - ToyPtr frees the Node for you.
- ❖ But **p.deref().val** is noisier than plain pointer code.
- ❖ A raw pointer would just say **(*p).val** or **p->val**.
- ❖ Can we make ToyPtr read like that? **Yes, overload operators.**

# ToyPtr with operator overloading

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

**operator* and operator->**

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

# Using *p and p->

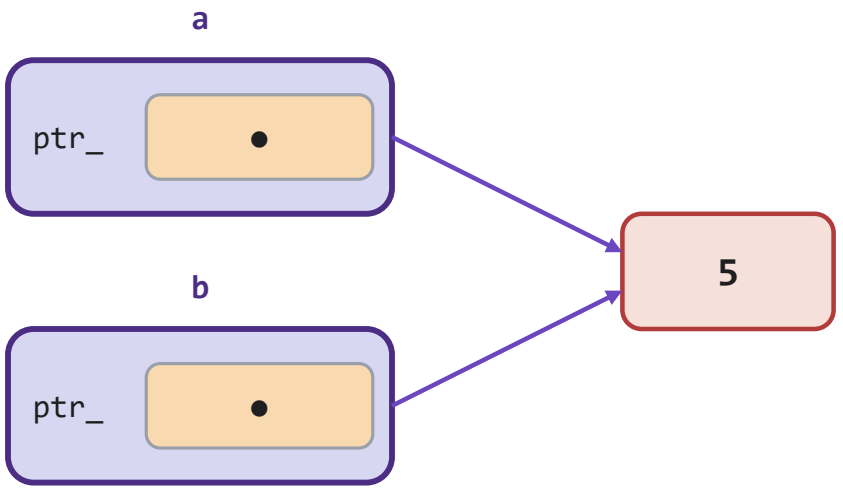
```
1  struct Node { int val; };
2
3  {
4    ToyPtr<Node> p(new Node{5});
5    cout << (*p).val;     // operator*
6    (*p).val = 10;        // write through
7    cout << p->val;       // operator-> : cleaner
8  }   // p leaves scope -> Node deleted
```

**Reads like a raw pointer**

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

# The copy bug

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




**both destructors delete the SAME int**

# What should copying mean?

**Copying an owner of one heap object must do WHAT?**

## Answer A: forbid the copy

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

## Answer B: share and count

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

# unique_ptr: exactly one owner

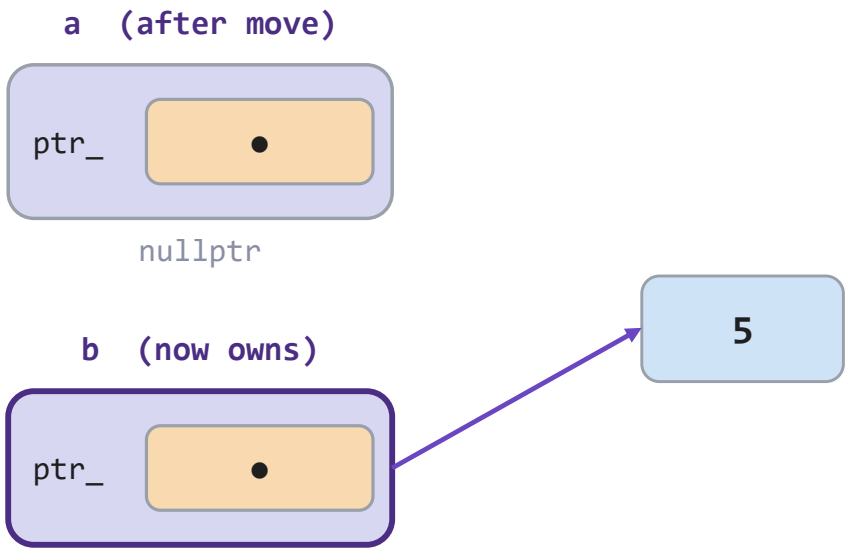
```
1  #include <memory>
2  using namespace std;
3
4  unique_ptr<int> a(new int(5));
5  unique_ptr<int> b = a;    // COMPILE ERROR: no copy
6  cout << *a << endl;       // a is the sole owner
7  // preferred:
8  auto c = make_unique<int>(5);
```

**Sole ownership**

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

# Moving a unique_ptr

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



# vector<unique_ptr<Image>>

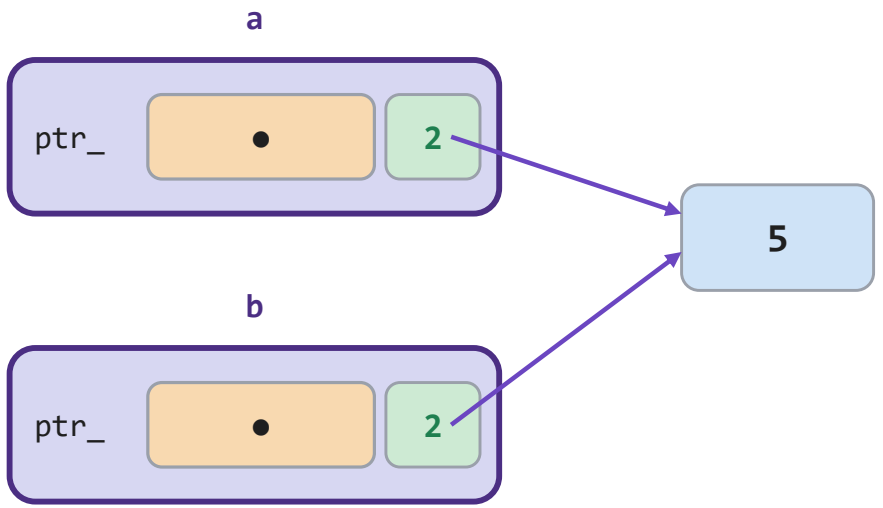
```
1  vector<unique_ptr<Image>> gallery;
2  gallery.push_back(make_unique<Image>("a.png"));
3  gallery.push_back(make_unique<Image>("b.png"));
4
5  // container MOVES pointers, never copies an Image;
6  // when gallery dies, every Image is deleted.
```

**Container owns the objects**

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

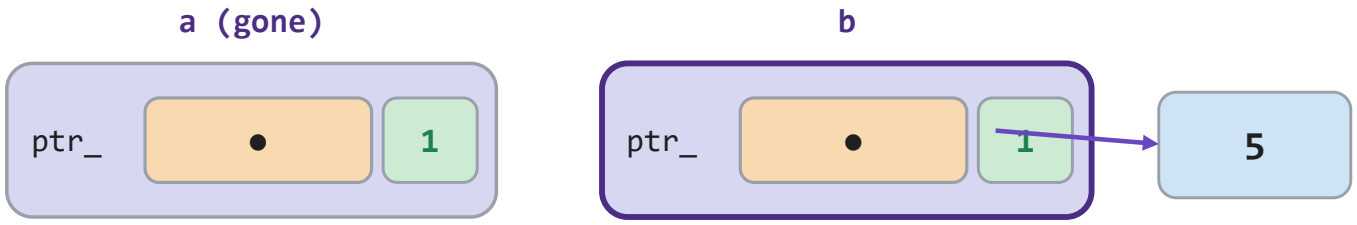
# shared_ptr: count the owners

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




one object, count = 2 owners

# Freed when the last owner leaves




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

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

## Reference counting

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

# Choosing: unique vs shared

**Reach for unique_ptr when…**

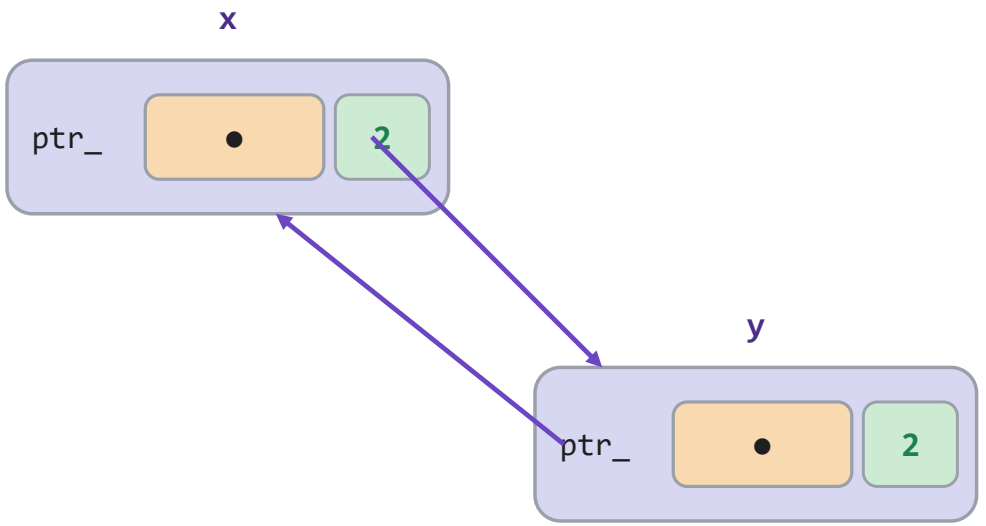
- …there is **one clear owner** at a time (the common case).
- Factory returns, members that own a resource.
- **Free, and it** documents ownership in the type.
- Move it to hand ownership on.

**Reach for shared_ptr when…**

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

# Uh-oh! Reference cycles

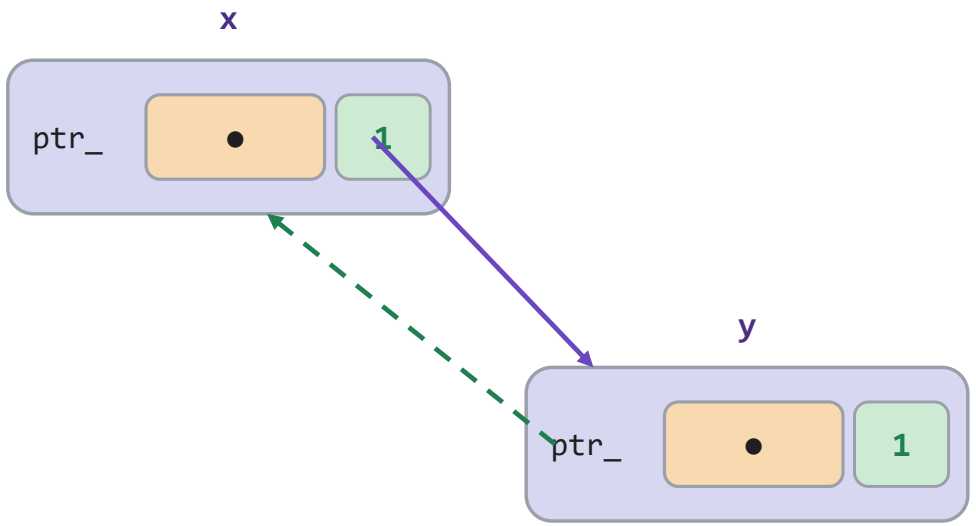
```
1  struct Node {
2    shared_ptr<Node> other;
3  };
4
5  auto x = make_shared<Node>();
6  auto y = make_shared<Node>();
7  x->other = y;   // y's count -> 2
8  y->other = x;   // x's count -> 2
```




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

# weak_ptr: observe without owning

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




**back-edge is weak: does not count**

# The three, side by side

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

# STL Challenges (harder)

# A drawing canvas

### The scenario

- A canvas holds many shapes of **different types**: Circle, Rectangle, Line, ...
- Each frame you call **draw()** on every shape (polymorphism).
- Shapes live on the **heap** and the canvas OWNS them.
- When the canvas is destroyed, every shape must be freed.

### Discuss with a neighbor

- Why can't this be **vector<Shape>?** (think slicing)
- If you store **Shape***, who calls delete, and when?
- What element type makes cleanup **automatic?**
- One owner, or shared?

**Predict first:** polymorphic, heap-owned objects in one container. What element type frees them for you?

# A drawing canvas

**Use: vector<unique_ptr<Shape>>**

```
1  vector<unique_ptr<Shape>> canvas;
2  canvas.push_back(make_unique<Circle>(r));
3  canvas.push_back(make_unique<Rect>(w, h));
4  for (const auto& s : canvas) {
5    s->draw();       // virtual dispatch
6  }                  // canvas dies -> shapes freed
```

## Why / tradeoffs

- Pointers keep **polymorphism**: Shape by value would SLICE.
- **unique_ptr** = the canvas is the sole owner.
- **Canvas destroyed ->** every Shape deleted automatically.
- push_back(**make_unique**<Circle>(...)); the vector moves pointers.

# An LRU cache (harder)

### The scenario

- A cache of at most **N** key/value entries.
- Every access must be **fast (O(1))** by key.
- When full, evict the **least-recently-used** entry.
- So you must also track **recency order**, cheaply.

### Discuss with a neighbor

- No single container does BOTH. Why?
- Which gives **O(1) key lookup?**
- Which gives **O(1) move-to-front / drop-the-back?**
- **How could two structures** point at each other?

**Predict first:** you need O(1) lookup AND O(1) recency reordering. One container can't. Combine two?

# An LRU cache (harder)

**Use: unordered_map + list working together**

```
1  list<Key> recent;          // front = newest
2  unordered_map<Key,
3                list<Key>::iterator> pos;
4
5  // on access(k): move k to the front, O(1)
6  recent.splice(recent.begin(), recent, pos[k]);
7
8  Key evict = recent.back(); // least recently used
```

## Why / tradeoffs

- **list<Key>** keeps keys in recency order (front = newest).
- **unordered_map** maps key -> its NODE in that list.
- **Access: splice** the node to the front - **O(1)**.
- Evict: drop **list.back()** and erase it from the map - **O(1)**.

No single container does it all: pair unordered_map (O(1) lookup) with list (O(1) recency reorder), each holding a link into the other.

# Reminders

**Assessments**:

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

**General**:

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

# Questions?

Thanks for coming - see you next lecture!