ENPM702

Introductory Robot Programming

L6: Smart Pointers

v1.0

Lecturer: Z. Kootbally

Semester/Year: Summer/2025

MARYLAND APPLIED GRADUATE ENGINEERING

Table of Contents

Changelog

Changelog

- **v1.0**: Original version.

Learning Objectives

Learning Objectives

By the end of this session, you will be able to:

- Explain RAII principles and implement automatic resource management using smart pointers to prevent memory leaks and ensure exception safety.
- Analyze ownership semantics and select appropriate smart pointer types (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`) based on resource sharing requirements.
- Implement exclusive ownership patterns using `std::unique_ptr` with proper initialization, move semantics, and method usage (`get()`, `release()`, `reset()`, `swap()`).
- Design shared ownership systems using `std::shared_ptr` with reference counting, control blocks, and safe resource sharing across multiple objects.
- Apply `std::weak_ptr` to break circular dependencies and implement safe resource observation without affecting object lifetimes.
- Evaluate function parameter strategies (sink, reseat, return) and implement proper ownership transfer patterns in modern C++ applications.

RAII

RAII (Resource Acquisition Is Initialization) is a programming idiom used in C++ and other languages that ensures resource management (like memory, file handles, sockets, etc.) is tied to the lifetime of objects. The **primary goal of RAII** is to acquire resources within a constructor and release them in a destructor, ensuring proper cleanup and preventing resource leaks.

> RAII is widely employed in the standard library, with examples including `std::vector` for dynamic array management, `std::unique_ptr` and `std::shared_ptr` for memory management, and various other resource-managing classes.

Smart Pointers

A **smart pointer** (from `<memory>`) is a class that wraps [1] a raw pointer (also called **stored pointer**) to manage the lifetime of the resource being pointed to.

> Smart pointers are designed to manage dynamic memory allocation on the heap, ensuring that resources are properly released when they are no longer needed. They store a pointer to the allocated memory and automatically call the appropriate cleanup function (typically `delete`) when the object is destroyed or goes out of scope.

[1] A wrapper is a data structure or software that contains (wraps around) other data or software, so that the contained elements can exist in the newer system.

Types of Smart Pointers

C++ provides three distinct smart pointer types, each designed to abstract raw pointer management while clearly expressing ownership semantics and programmer intent.

- `std::unique_ptr` – **Exclusive ownership**: single object controls the resource lifetime.
- `std::shared_ptr` – **Shared ownership**: multiple objects collectively manage the resource.
- `std::weak_ptr` – **Non-owning observer**: monitors resource state without affecting lifetime, prevents circular dependencies.

Unique Pointers

A **unique pointer** (`std::unique_ptr`) implements **exclusive ownership** semantics for dynamically allocated resources. This exclusivity guarantees that exactly one `std::unique_ptr` instance controls any given memory location at any time.

- Automatic Resource Management – The `std::unique_ptr` owns its resource and automatically invokes `delete` when the pointer is destroyed, reassigned, or goes out of scope, ensuring deterministic cleanup.
- Move-Only Semantics – `std::unique_ptr` is **non-copyable** but **movable**, enforcing single ownership through the type system and preventing accidental resource sharing.

Initialization

```cpp
// Preferred: Exception-safe, concise
std::unique_ptr<T> identifier = std::make_unique<T>(args ... );
auto identifier = std::make_unique<T>(args ... );
```

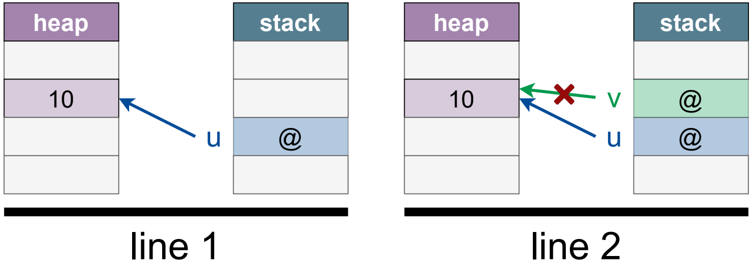
```cpp
// Discouraged: Potential exception safety issues
std::unique_ptr<T> identifier(new T(args ... ));
std::unique_ptr<T> identifier = std::unique_ptr<T>(new T(args ... ));
auto identifier = std::unique_ptr<T>(new T(args ... ));
```

- `T` – Type of the managed object allocated on the heap.
- `identifier` – Variable name for the unique pointer instance.
- `args ...` – Constructor arguments passed to the managed object
- R.23: Use make_unique() to make unique_ptrs

Exclusive Ownership

A unique pointer **exclusively** manages the lifetime of a resource.

```cpp
std::unique_ptr<int> u = std::make_unique<int>(10);
std::unique_ptr<int> v{u}; // Error
```



Example

```cpp
int main(){
    {
         // Create managed resource on heap
         auto u = std::make_unique<int>(10);
         std::cout << *u << '\n';   // 10 (dereference to access value)
         *u = 20;                   // modify the managed resource
         std::cout << *u << '\n';   // 20
        std::cout << u << '\n';    // Compilation error: no operator<< for unique_ptr
    } // u destructor automatically calls delete on managed resource
}
```

The line `std::cout << u << '\n';` fails to compile because `std::unique_ptr` does not provide an `operator`<< overload. The smart pointer wrapper cannot be directly streamed.

A solution for accessing the raw pointer is demonstrated in slide 12.

Methods

`std::unique_ptr` provides a comprehensive set of member functions for resource management and inspection. Many of these methods share consistent interfaces with `std::shared_ptr` and `std::weak_ptr`, facilitating code maintainability across different smart pointer types.

☰ `get()`

The `get()` method returns the raw pointer to the managed resource without transferring ownership or modifying the `std::unique_ptr` state.

- The returned raw pointer should be used exclusively for **non-owning observation**.
  - **Null-checking**: verify if the `std::unique_ptr` manages a valid resource.
  - **Address inspection**: obtain the memory address of the managed object.
  - **Dereferencing**: access the managed object's value through the raw pointer.

Never call `delete` on the returned raw pointer. The `std::unique_ptr` retains ownership and will automatically `delete` the resource upon destruction, resulting in **double-delete undefined behavior**.

Example

```cpp
// Create managed resource on heap
auto u = std::make_unique<int>(10);
if (u) { // implicit conversion to bool (checks if not null)
   std::cout << "Value at " << u.get() << " is " << *u << '\n';
}
```

or

```cpp
// Create managed resource on heap
auto u = std::make_unique<int>(10);
int* raw_ptr{u.get()}; // extract raw pointer for observation
if (raw_ptr) { // null-check the raw pointer
   std::cout << "Value at " << raw_ptr << " is " << *raw_ptr << '\n';
}
```

`release()`

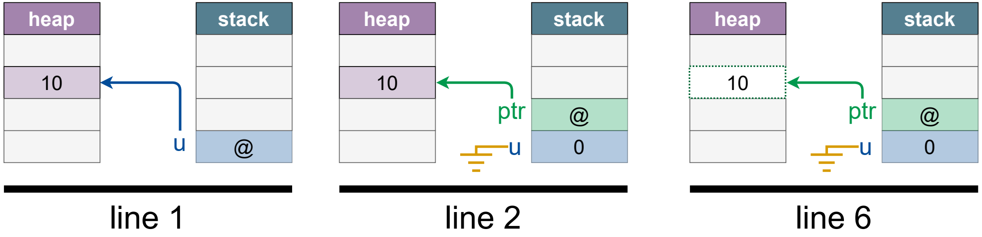
The `release()` method transfers ownership of the managed resource from the `std::unique_ptr` to the caller without destroying the resource.

- The `std::unique_ptr` relinquishes ownership of the managed resource.
- Returns a raw pointer to the previously managed resource.
- The `std::unique_ptr` is reset to `nullptr` and no longer manages any resource.

After calling `release()`, **manual memory management becomes your responsibility**. The returned raw pointer must be explicitly deleted using `delete` when no longer needed, or **memory leaks** will occur.

Example

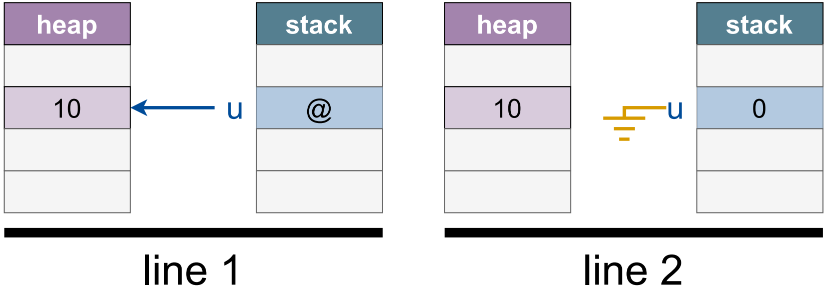
```cpp
auto u = std::make_unique<int>(10);
auto ptr = u.release(); // transfer ownership to ptr
std::cout << *ptr << '\n'; // 10
assert(u.get() == nullptr); // u no longer owns the resource
assert(u == nullptr); // implicit bool conversion check
delete ptr; // Mandatory: prevent memory leak
```



Example:Memory Leak

The following code introduces a **memory leak**.

```cpp
auto u = std::make_unique<int>(10);
u.release();
```



When to Use `release()`?

Use `release()` when transferring ownership from a `std::unique_ptr` to legacy code, C-style APIs, or systems that require raw pointer management.

```cpp
void legacy_function(int* ptr) {
    if (ptr) {
        std::cout << "Processing: " << *ptr << '\n';
        delete ptr; // Legacy code handles cleanup
        }
}

int main() {
    auto u = std::make_unique<int>(42);

    // Transfer ownership to raw pointer for legacy interface
    int* ptr{ u.release() };

    // Verify ownership transfer
    assert(u == nullptr); // u no longer owns resource

    // Pass to legacy system that expects raw pointer ownership
    legacy_function(ptr); // ptr now responsible for deletion
}
```

When to Use `release()`?

Transferring ownership to another `std::unique_ptr` (though move semantics is preferred).

```cpp
auto u1 = std::make_unique<int>(10);
int* ptr{u1.release()};
std::unique_ptr<int> u2(ptr); // u2 assumes ownership
```

- u2 takes ownership of the resource pointed to by `ptr`
- `ptr` continues pointing to the resource but no longer manages its lifetime.
- If u2 is destroyed or reset, the resource is deleted and `ptr` becomes a dangling pointer.

Use move semantics (slide 24) instead of `release()` for ownership transfer between smart pointers.

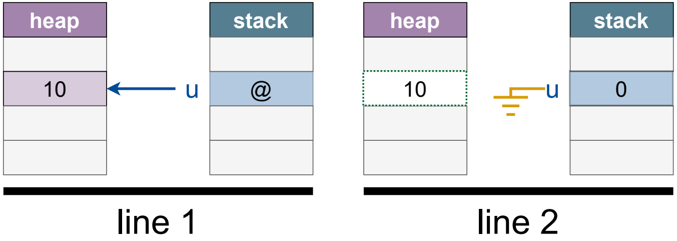
reset()

The `reset()` method provides controlled resource replacement with automatic cleanup of the previously managed resource.

- Destroys the currently managed resource (if any) by calling `delete`
- Optionally assumes ownership of a new dynamically allocated resource.
- Sets the `std::unique_ptr` to `nullptr` if no new resource is provided.

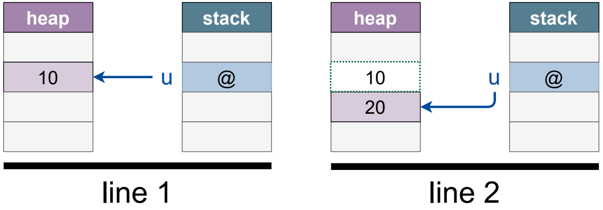
Calling `reset()` without arguments destroys (calls `delete`) the currently managed resource and resets the `std::unique_ptr` to `nullptr`.

```cpp
auto u = std::make_unique<int>(10);
u.reset(); // destroy managed resource, set to nullptr
assert(u.get() == nullptr); // verification: u is now null
```



When `reset()` receives a pointer argument, it first destroys the currently managed resource (calls `delete`), then assumes ownership of the new resource in a single atomic operation.

```
auto u = std::make_unique<int>(10);
u.reset(new int(20)); // destroy old resource, manage new one
```



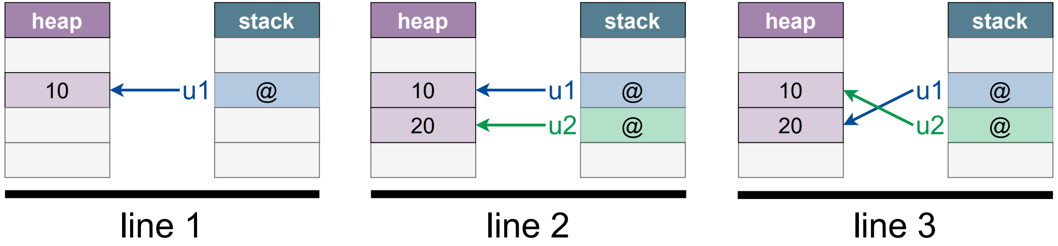
swap()

The `swap()` method exchanges ownership of managed resources between two `std::unique_ptr` instances. Each pointer assumes control of the resource previously managed by the other.

Swapping `std::unique_ptr` instances is an O(1) constant-time operation that only exchanges internal pointer values (no resource copying, moving, or reallocation occurs).

Example

```cpp
auto u1 = std::make_unique<int>(10);
auto u2 = std::make_unique<int>(20);
u1.swap(u2);
```



Move Semantics

Move semantics, introduced in C++11, enables efficient resource transfer between objects by moving ownership rather than performing expensive deep copies.

The compiler automatically applies move semantics in many contexts. For explicit control, use `std::move()` to force move operations when the compiler cannot deduce the intent.

§23.11.1 – After move-transferring ownership from `source_ptr` to `dest_ptr`, the source pointer is guaranteed to be in a valid but unspecified state (typically `nullptr`).

What does the program output?

```cpp
void display(std::unique_ptr<int> v){
    // Implicit: std::unique_ptr<int> v{u};
    std::cout << *v << '\n';
}

int main(){
    auto u = std::make_unique<int>(10);
    display(u);
}
```

What does the program output?

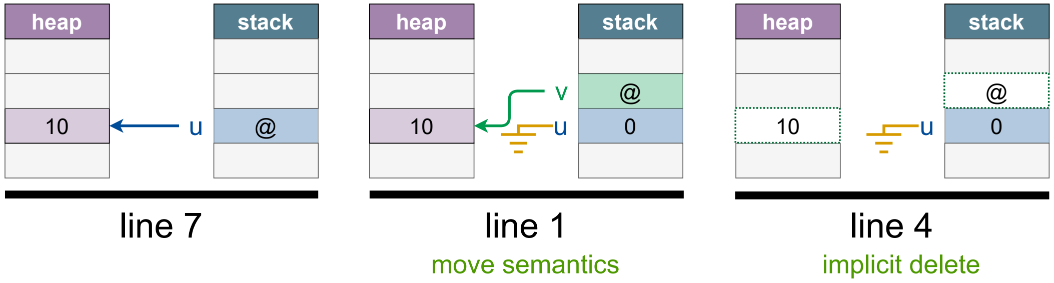
```cpp
void display(std::unique_ptr<int> v){
    // Implicit: std::unique_ptr<int> v{u};
    std::cout << *v << '\n';
}

int main(){
    auto u = std::make_unique<int>(10);
    display(u);
}
```

- This code attempts to pass a `std::unique_ptr` by value to the function.
- The function call implicitly tries to copy-construct v from u
- `std::unique_ptr` has a **deleted copy constructor** to enforce exclusive ownership.
- **Result:** **Compilation error** - the program will not compile.

Correct Way: Using Move Semantics

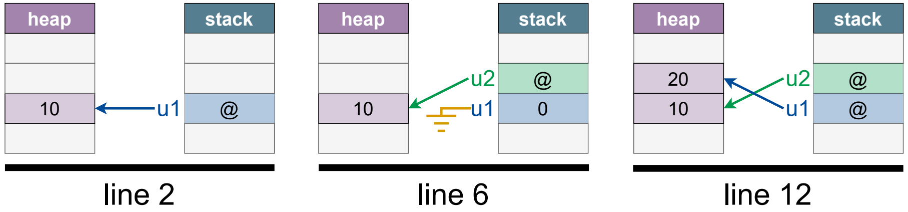
```cpp
 1  void display(std::unique_ptr<int> v){
 2      // Implicit: std::unique_ptr<int> v{std::move(u)};
 3      std::cout << *v << '\n'; // 10
 4  } // v is destroyed here, resource is deleted
 5
 6  int main(){
 7      auto u = std::make_unique<int>(10);
 8      display(std::move(u)); // Transfer ownership to function
 9      // u is now nullptr - ownership transferred
10  }
```



Unique Pointers ▶ Move Semantics

Example

```cpp
 1  // Create managed resource on heap
 2  auto u1 = std::make_unique<int>(10);
 3  std::cout << "u1: " << u1.get() << '\n';  // @1
 4
 5  // Transfer ownership using move constructor
 6  auto u2{std::move(u1)}; // u1 transfers ownership to u2
 7  std::cout << "u2: " << u2.get() << '\n'; // @1 (same address)
 8  assert(u1 == nullptr);  // u1 is now empty
 9
10  // u1 can be reused for new resource management
11  u1.reset(new int{20});
12  std::cout << u1.get() << '\n';  // @2 (different address)
```



move semantics

Sink Function

A sink function accepts ownership of a resource, typically through move semantics. The function becomes responsible for the resource's lifetime and cleanup.

R.32: Take a unique_ptr<widget> parameter to express that a function assumes ownership of a widget.

```
void process_widget(std::unique_ptr<Widget> widget_ptr);
// Caller transfers ownership: process_widget(std::move(my_widget));
```

Example

See `display(std::unique_ptr<int> v)` in slide 27.

Reseat Function

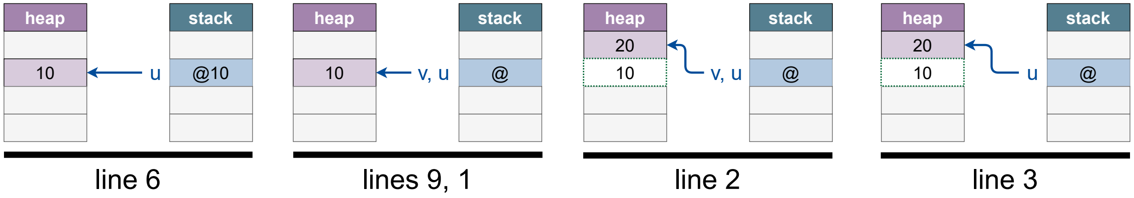
A reseat function modifies a smart pointer to manage a different resource. The function may destroy the current resource and assign a new one, or simply replace the managed object.

R.33: Take a unique_ptr<widget>& parameter to express that a function reseats the widget.

```
void configure_widget(std::unique_ptr<Widget>& widget_ptr);
// Function may call: widget_ptr.reset(new EnhancedWidget());
```

Unique Pointers ▶ Pass to Functions: Reseat

```cpp
1  void reseat(std::unique_ptr<int>& v) { // Pass by reference to modify original
2      v.reset(new int(20)); // Destroy current resource, create new one
3  }
4
5  int main(){
6      auto u = std::make_unique<int>(10); // Create managed resource
7      std::cout << "*u: " << *u << '\n';          // 10
8      std::cout << "u: " << u.get() << '\n';      // @1
9      reseat(u); // u will be modified to point to new resource
10     std::cout << "*u: " << *u << '\n';          // 20
11     std::cout << "u: " << u.get() << '\n';      // @2
12 }
```



Return from Functions

F.26: Use a unique_ptr<T> to transfer ownership where a pointer is needed.

Example

```cpp
std::unique_ptr<int> create_resource() {
    auto v = std::make_unique<int>(10);
    std::cout << *v << '\n';      // 10
    std::cout << &v << '\n';      // @1 (address of local variable)
    return v; // Ownership transferred to caller
}
int main(){
 auto u{create_resource()};
 std::cout << *u << '\n';         // 10
 std::cout << &u << '\n';         // @1 (same address due to optimization)
}
```

Which compiler optimization technique eliminates the move operation?

Shared Pointers

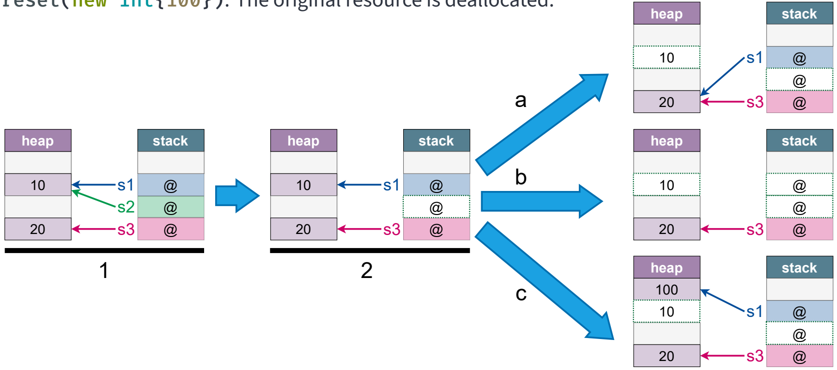
A shared pointer (`std::shared_ptr`) implements shared ownership semantics, allowing multiple smart pointers to collectively manage a single resource through a reference-counted control block.

- Resource acquisition occurs during `std::shared_ptr` construction.
- Multiple `std::shared_ptr` instances can share ownership by copying from an existing instance.
- The resource remains valid while at least one `std::shared_ptr` maintains ownership.
- Automatic resource deallocation occurs when the reference count reaches zero:
  - The last `std::shared_ptr` is reassigned to manage a different resource.
  - The last `std::shared_ptr` is destroyed (scope exit).
  - The last `std::shared_ptr` is explicitly reset via `reset()`

Shared Pointers

Demonstration

1. `s1` and `s2` are both managing the same resource (`10`). `s3` is managing a different resource (`20`).
2. `s2` goes out of scope. The resource (`10`) is not deallocated because there is still a pointer (`s1`) managing the resource.
   a. `s1=s3`: `s1` is now managing a different resource. The original resource is deallocated.
   b. `s1` goes out of scope. The original resource is deallocated.
   c. `s1.reset(new int{100})`: The original resource is deallocated.



Initialization

```cpp
// Preferred: Exception-safe, efficient single allocation
std::shared_ptr<T> identifier = std::make_shared<T>(args ... );
auto identifier = std::make_shared<T>(args ... );
```

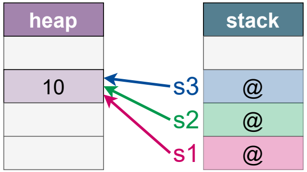
```cpp
// Discouraged: Two allocations, potential exception issues
std::shared_ptr<T> identifier(new T(args ... ));
std::shared_ptr<T> identifier = std::shared_ptr<T>(new T(args ... ));
auto identifier = std::shared_ptr<T>(new T(args ... ));
```

- T – Type of the managed object allocated on the heap
- `identifier` – Variable name for the shared pointer instance
- `args ...` – Constructor arguments passed to the managed object

R.22: Use make_shared() to make shared_ptrs.

Example: Multiple shared pointers managing the same resource.

```cpp
auto s1 = std::make_shared<int>(10);
auto s2{s1};
auto s3 = s2;
```



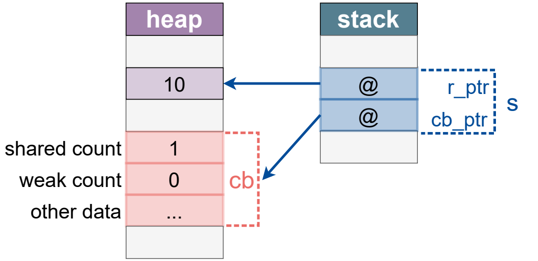
Structure of a `std::shared_ptr`

A `std::shared_ptr` maintains two essential pointers to implement shared ownership semantics.

- **Resource Pointer** (`resource_ptr`) – Points directly to the managed object on the heap.
- **Control Block Pointer** (`control_ptr`) – Points to the control block containing reference count and metadata.

Example

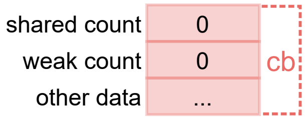
```
auto s = std::make_shared<int>(10);
```



Control Block

A control block is a heap-allocated data structure that manages shared ownership metadata for `std::shared_ptr` and `std::weak_ptr`, enabling reference counting and coordinated resource cleanup.

- Created when the first `std::shared_ptr` is constructed; **subsequent copies share the same control block instance**.
- Contains reference counts, custom deleters, and allocator information to ensure thread-safe resource management.
- Introduces memory overhead compared to `std::unique_ptr` or raw pointers, but enables safe shared ownership semantics.



Shared Count (or Reference Count)

Tracks the number of `std::shared_ptr` instances sharing ownership of the resource.

- Incremented when a `std::shared_ptr` copies or moves to share the resource.
- Decremented when a `std::shared_ptr` is destroyed, reset, or reassigned.
- **Resource is automatically deleted when count reaches zero**.

Weak Count

Tracks the number of `std::weak_ptr` instances observing the resource without owning it.

- Incremented/decremented as `std::weak_ptr` instances are created/destroyed.
- Does not influence resource lifetime, only control block lifetime.
- **Control block is deallocated when both reference count and weak count reach zero**.

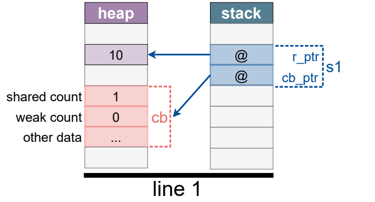
Custom Deleter & Allocator (Optional)

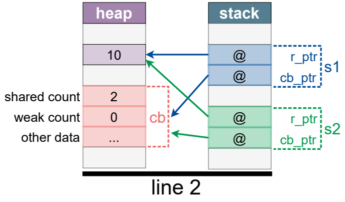
Type-erased function objects specifying custom resource cleanup and control block memory management strategies.

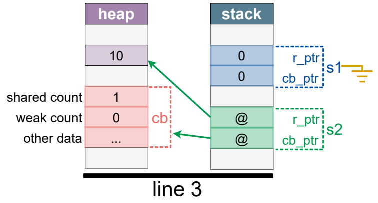
Shared Pointers ▶ Structure ▶ Control Block

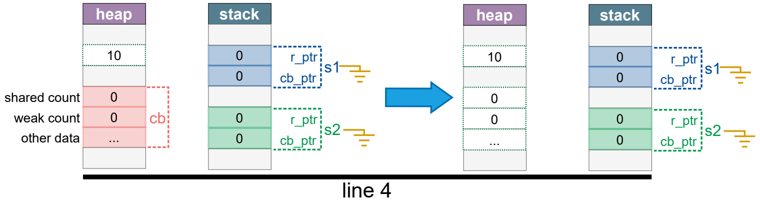
Example

```cpp
1 auto s1 = std::make_shared<int>(10);
2 auto s2{s1};
3 s1.reset();
4 s2.reset();
```









Methods

- `get()` returns the raw pointer to the managed resource (non-owning access).
- `reset()` releases current resource ownership and optionally assumes ownership of a new resource; decrements reference count of previous control block.
- `swap()` exchanges both resource and control block pointers with another `std::shared_ptr`.
- `use_count()` returns the current reference count (number of `std::shared_ptr` instances sharing ownership).
  - Primarily intended for debugging and testing; avoid using in production logic.

Exercise #1

Analyze the code from **lecture6.cpp**.

Sink Function

R.34: Take a shared_ptr<widget> parameter to express shared ownership.

```cpp
void process_widget(std::shared_ptr<Widget> widget_ptr);
// Call: process_widget(my_shared_widget); // Copy shares ownership
```

Example

```cpp
void sink_shared_ptr(std::shared_ptr<int> ptr) { // Copy constructor increments ref count
  std::cout << ptr.use_count() << '\n';      // 2 (original + copy)
}

int main() {
  auto original = std::make_shared<int>(10);
  std::cout << original.use_count() << '\n';   // 1
  sink_shared_ptr(original); // Pass by value creates copy
  std::cout << original.use_count() << '\n';   // 1 (copy destroyed after function)
}
```

Reseat Function

R.35: Take a shared_ptr<widget>& parameter to express that a function might reseat the shared pointer.

```cpp
void configure_widget(std::shared_ptr<Widget>& widget_ptr);
// Function may call: widget_ptr.reset(std::make_shared<EnhancedWidget>());
```

Example

```cpp
void reseat_shared_ptr(std::shared_ptr<int>& ptr) { // Pass by reference - no copy
  std::cout << ptr.use_count() << '\n';     // 1 (same instance, no increment)
  // Could modify ptr here: ptr.reset(std::make_shared<int>(20));
}

int main() {
  auto original = std::make_shared<int>(10);
  std::cout << original.use_count() << '\n';   // 1
  reseat_shared_ptr(original); // Pass by reference
  std::cout << original.use_count() << '\n';   // 1 (no copy made)
}
```

Return From Functions

Return Value Optimization (RVO) eliminates unnecessary copies when returning `std::shared_ptr` by value, resulting in efficient ownership transfer.

Example

```cpp
std::shared_ptr<int> create_shared_resource() {
  auto local_ptr = std::make_shared<int>(10);
  std::cout << &local_ptr << '\n';    // @1 (local variable address)
  return local_ptr; // RVO: no copy, direct construction in caller's context
}

int main() {
  auto result{create_shared_resource()};
  std::cout << &result << '\n'; // @1 (same address due to RVO)
}
```

Weak Pointers

A weak pointer (`std::weak_ptr`) provides non-owning observation of resources managed by `std::shared_ptr`, enabling safe access without affecting resource lifetime.

> The primary purpose of `std::weak_ptr` is to break circular dependencies and provide safe resource monitoring without extending object lifetimes.

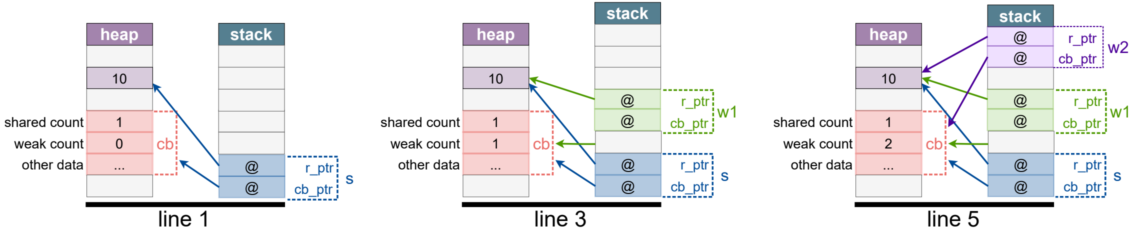
- Key use cases for `std::weak_ptr`:
  - Breaking circular references that would cause memory leaks with `std::shared_ptr`
    - Essential for parent-child relationships and observer patterns.
  - Safe checking of resource validity using `expired()` before access.
  - Temporary promotion to `std::shared_ptr` via `lock()` when needed.
- Like `std::shared_ptr`, contains both a resource pointer and control block pointer.

Initialization

A `std::weak_ptr` is constructed from an existing `std::shared_ptr` or copied from another `std::weak_ptr`. Creation increments the weak count but not the reference count.

Example

```cpp
auto s = std::make_shared<int>(10);
// Create weak_ptr from s
std::weak_ptr<int> w1{s};
// Create weak_ptr from another weak_ptr
std::weak_ptr<int> w2{w1};
```



`std::weak_ptr` cannot be directly dereferenced or provide raw pointer access. It is purely an observer that requires promotion to `std::shared_ptr` for resource access.

Example

```cpp
auto s = std::make_shared<int>(10);
// Create weak_ptr from shared_ptr
std::weak_ptr<int> w{s};

std::cout << *w << '\n';          // Error: No operator*
std::cout << w.get() << '\n';     // Error: No get() method
```

Methods

- `use_count()` returns the current reference count of `std::shared_ptr` instances managing the resource.
  - Primarily for debugging; avoid using in production logic due to race conditions.
- `reset()` releases the weak reference, setting the `std::weak_ptr` to empty state.
- `swap()` exchanges resource and control block references with another `std::weak_ptr`
- `lock()` attempts to create a `std::shared_ptr` from the weak reference.
  - Returns valid `std::shared_ptr` if resource still exists.
  - Returns null `std::shared_ptr` if resource has been destroyed.
- `expired()` returns `true` if the managed resource has been destroyed, `false` otherwise.

☰ lock()

```
 1 // Declare empty weak pointer
 2 std::weak_ptr<int> weak_observer;
 3 {
 4     auto shared_resource = std::make_shared<int>(10);
 5     weak_observer = shared_resource;  // weak_observer now observes the resource
 6     if (auto locked_ptr = weak_observer.lock(); locked_ptr)
 7         std::cout << "Resource value: " << *locked_ptr << "\n";
 8     else
 9         std::cout << "Unable to access resource\n";
10 }  // shared_resource destroyed, resource deallocated
11
12 if (auto locked_ptr = weak_observer.lock(); locked_ptr)
13     std::cout << "Resource value: " << *locked_ptr << "\n";
14 else
15     std::cout << "Unable to access resource\n";
```

Lines 6 and 12 use C++17's **init-statement** feature in `if` statements. The variable `locked_ptr` is initialized and then evaluated as a boolean condition (non-null check).

expired()

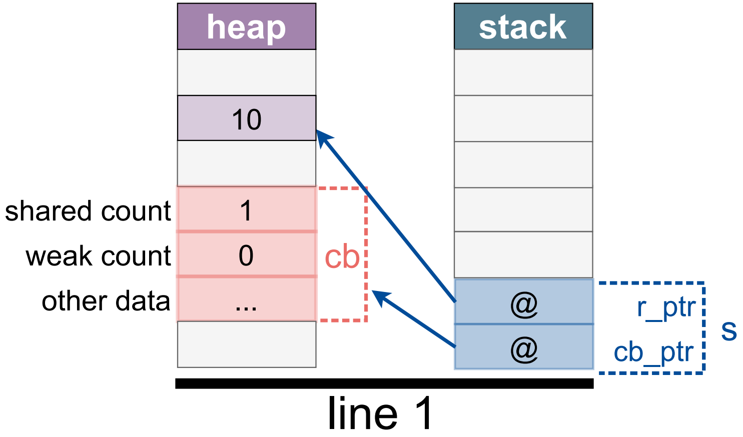
```cpp
// Declare empty weak pointer
std::weak_ptr<int> weak_observer;
{
    auto shared_resource = std::make_shared<int>(10);
    weak_observer = shared_resource;  // weak_observer now observes the resource
    if (!weak_observer.expired())
        std::cout << "Resource is still valid\n";
    else
        std::cout << "Resource has been destroyed\n";
}  // shared_resource destroyed, resource deallocated

if (!weak_observer.expired())
    std::cout << "Resource is still valid\n";
else
    std::cout << "Resource has been destroyed\n";
```

Control Block Lifetime

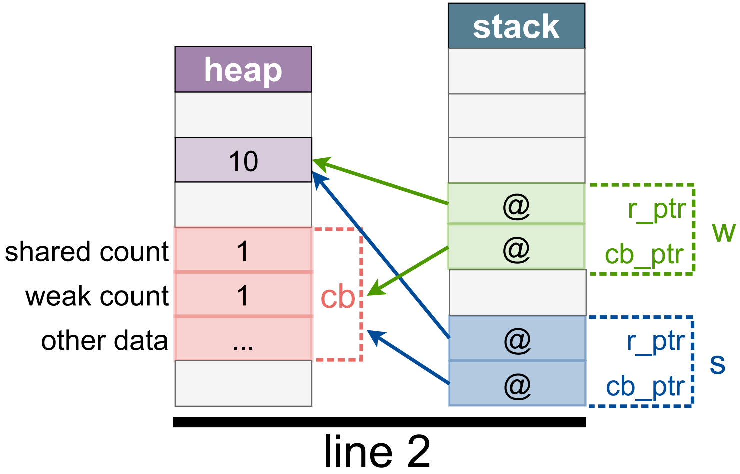
The control block persists on the heap until both shared count and weak count reach zero. This ensures `std::weak_ptr` instances can safely check resource validity even after the resource is destroyed.

```
auto s = std::make_shared<int>(10);
std::weak_ptr<int> w = s;
s.reset();
w.reset();
```



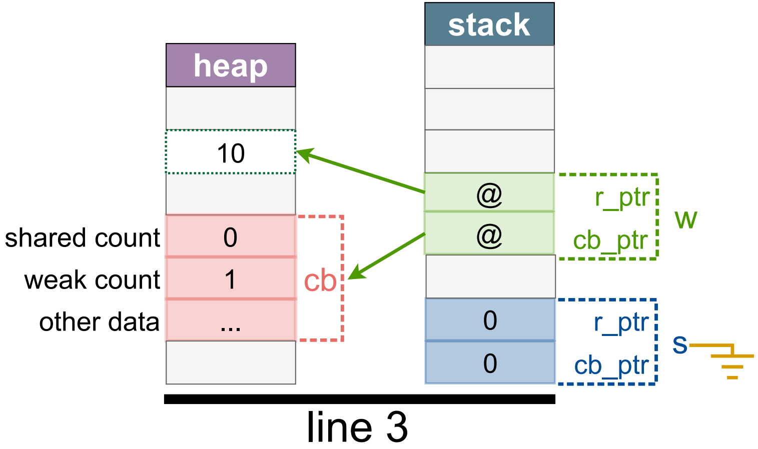
Weak Pointers ▸ Control Block Lifetime

```cpp
auto s = std::make_shared<int>(10);
std::weak_ptr<int> w = s;
s.reset();
w.reset();
```

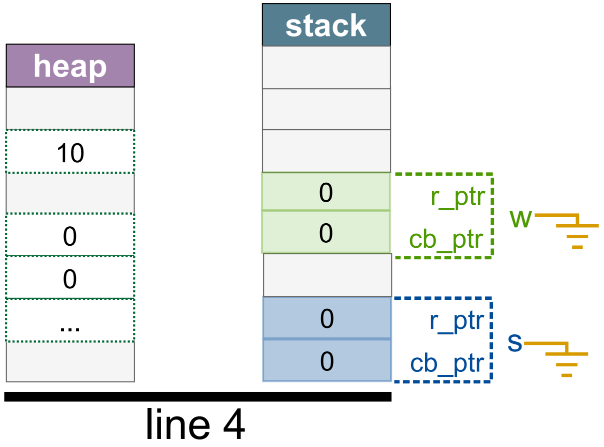


Weak Pointers ▸ Control Block Lifetime

```cpp
auto s = std::make_shared<int>(10);
std::weak_ptr<int> w = s;
s.reset();
w.reset();
```



```cpp
auto s = std::make_shared<int>(10);
std::weak_ptr<int> w = s;
s.reset();
w.reset();
```



Summary

Prefer smart pointers over raw pointers for all dynamic memory management to ensure automatic resource cleanup and exception safety.

Use `std::unique_ptr` as the default choice for single ownership. Only use `std::shared_ptr` when multiple owners are genuinely required.

Use `std::weak_ptr` to break circular dependencies and provide safe observation of shared resources without affecting their lifetime.

Next Class

- Lecture7: Object Oriented Programming (Part I).