MODERN C++ DESIGN PATTERNS

Smart Pointers

by Prasanna Ghali

Plan for Today

- `std::unique_ptr<T>`
- `std::unique_ptr<T[]>`
- `std::shared_ptr<T>`

Raw Pointers: Usage (1/4)

- Non-copying view of object owned by caller ["*in*" parameter]
- For callee to modify object owned by caller ["*in/out*" parameter]
- One-half of pointer/length pair for passing arrays ["*in*" or "*in/out*" parameter]
- Express "*no value*" in parameter or return value
- To manage heap memory

Raw Pointers: Usage (2/4)

- Non-copying view of object owned by caller [“*in*” parameter]
  - Replaced with native references [such as `X const&`]
- For callee to modify object owned by caller [“*in/out*” parameter]
  - Replaced with native references [such as `X&`]

Raw Pointers: Usage (3/4)

- One-half of pointer/length pair for passing arrays ["*in*" or "*in/out*" parameter]
  - Replaced with standard library containers such as `std::string`, `std::array<>`, `std::deque<>`, and `std::vector<>`

Raw Pointers: Usage (4/4)

- Express "*no value*" in parameter or return value
  - C++17 provides vocabulary type `std::optional<>` to simulate use of `nullptr` by raw pointers to express having no value
- To manage heap memory
  - C++11 provides smart pointers

Why Not Raw Pointers To Manage Heap? (1/2)

- Declaration doesn't indicate whether it is pointer to single object or array
- Declaration doesn't indicate whether pointer owns thing it points to
- If you want to destroy what pointer points to, there's no way to tell how
  - If `delete` is way to go, can't say whether to use `delete` or `delete[]`
- Difficult to ensure memory is released exactly once along every path in your code
- No way to tell if pointer dangles

Why Not Raw Pointers To Manage Heap? (2/2)

- Memory leaks
  - You might allocate object on heap and accidentally forget to write code that frees it
  - You might have written freeing code, but due to early return or exception being thrown, that code never runs and memory remains unfreed
- Use-after-free
  - You make copy of a pointer to heap object, and then free that object thro' original pointer; holder of copied pointer doesn't realize their pointer is no longer valid
- Heap corruption via pointer arithmetic
  - You allocate array on heap starting at address *A*; using raw pointer you do pointer arithmetic; you accidentally free pointer to address *A+k* where $k \neq 0$

Zombie Objects


```
// memory that can never be recovered ...
size_t make_a_wish(std::string owner, int id) {
  Wish *wish = new Wish(wishes[id], owner);
  return wish->size();
}
```

```
// possible problems: memory leak
// pre-mature deletion, double deletion
Wish* make_a_wish(std::string owner, int id) {
  Wish *wish = new Wish(wishes[id], owner);
  return wish;
}
```

What are Smart Pointers? (1/2)

- Class wrappers around raw pointers so that heap resource is managed using RAII idiom
  - Behaves syntactically just like a pointer
  - Special member functions [ctors, dtors, copy/move] have additional bookkeeping to ensure certain constraints

What are Smart Pointers? (2/2)

- Fundamental property: overload operator *
- Overload special member functions to preserve its class invariants, whatever those are:
  - Pointer’s dtor also free its pointee
  - Maybe pointer cannot be copied
  - Or, maybe pointer can be copied, but it knows how many copies exist and won’t free pointee until last pointer to it has been destroyed
  - Or maybe pointer can be copied, and you can free pointee, but if you do, all other pointers to it magically become null
  - Or, maybe pointer has no built-in operator +
  - Or, maybe you’re allowed to adjust pointer’s value arithmetically, but arithmetic “what object is pointed-to”is managed separately from identity of “what object is to be freed”

Smart Pointers: <`memory`>

Name	Description
`std::unique_ptr`	*Exclusively owns resources* Can't be copied Uses RAII to automatically delete resource when owner goes out-of-scope
`std::shared_ptr`	Uses *reference counter* to keep track of users of resource Deletes resource when reference counter is 0
`std::weak_ptr`	Doesn't own resources Merely observes objects being shared by `shared_ptr`s

`std::unique_ptr<T>`

- Embodies *exclusive ownership* semantics
- Can neither implicitly nor explicitly copy such a pointer – you can only *move* it
- Automatically releases resource when it goes out of scope
- No pointer arithmetic is defined
- Equally sized and equally fast as raw pointers

std::unique_ptr<> Methods

Name	Description
`get`	Returns pointer to resource
`get_deleter`	Returns `delete` function
`release`	Returns pointer to resource and releases it
`reset`	Resets resource
`swap`	Swaps resource

Using std::unique_ptr<T> (1/3)

- Same interface as ordinary pointer
  - Operator * dereferences object to which it points
  - Operator -> provides access to member if object is class or structure

```
std::unique_ptr<std::string> up{new std::string{"hlp3"}};
(*up)[0] = 'H';              // replace first character
up->append("good");          // append some characters
std::unique_ptr<int> up2 = new int; // error
std::unique_ptr<int> up3(new int);  // ok
std::unique_ptr<std::string> up4;   // ok: empty
up.reset(); // up = nullptr;
std::unique_ptr<std::string> up4{new std::string{"hlp3"}};
std::string *ps = up4.release(); // up4 loses ownership
```

Using std::unique_ptr<T> (2/3)


```cpp
std::unique_ptr<std::string> up{new std::string{"hlp3"}};
if (up) { // call operator bool()
  std::cout << *up << '\n';
}

if (up != nullptr) { // if up is not empty
  std::cout << *up << '\n';
}

if (up.get() != nullptr) { // if up is not empty
  std::cout << *up << '\n';
}
```

Using std::unique_ptr<T> (3/3)

- See *using_up.cpp*

```
std::unique_ptr<int> up1 {new int {10}};
// make code exception-safe
std::unique_ptr<int> up2 {std::make_unique<int>(10)};
std::unique_ptr<int> up3 = up2; // error: no copies
std::unique_ptr<int> up4 = std::move(up1); // ok
```

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique( Args&&... args ) {
    return std::unique_ptr<T>(
      new T( std::forward<Args>(args)... ) );
}
```

std::unique_ptr<T> Clone (1/6)


```cpp
template <typename T>
class ToyPtr {
public:
  ToyPtr() noexcept = default;
  ToyPtr(T *rhs) noexcept : m_ptr{rhs} {}
  ToyPtr(ToyPtr const& rhs) = delete;

  ToyPtr& operator=(ToyPtr const& rhs) = delete;

  T* get() const noexcept { return m_ptr; }
  operator bool() const noexcept { return bool(get()); }
  T& operator*() const noexcept { return *get(); }
  T* operator->() const noexcept { return get(); }
private:
  T *m_ptr{nullptr};
};
```

SIDENOTE: Dereferencing (1/2)

- Dereferencing operator `->` can be defined as unary postfix operator:

```
struct X { int m; };

struct Ptr {
  // ...
  X *x;
  X* operator->() { return x; }
};
```

```
void f(Ptr p) {
  p->m = 7; // (p.operator())->m = 7
}
```

- Objects of type `Ptr` can be used to access members of `X` similar to way pointers are used

SIDENOTE: Dereferencing (2/2)

- If used, return type of operator -> must be pointer or object of class to which you can apply ->

```
struct A { int a; };

struct BA {
  A *p;
  A* operator->()  { return p; }
};

struct CBA {
  BA *p;
  BA& operator->()  { return *p; }
};
```

```
A   a{2};
BA  ba{&a};
CBA cba{&ba};

std::cout << a.a
          << (ba.operator->())->a
          << cba->a
          << '\n';
```

std::unique_ptr<T> Clone (2/6)


```
template <typename T>
class ToyPtr {
public:
  // ...
  void reset(T *p = nullptr) noexcept;
private:
  T *m_ptr{nullptr};
};

// p.reset(q) frees current contents of p, and
// then puts raw pointer q in its place:
template <typename T>
void ToyPtr<T>::reset(T *p) noexcept {
  T *old_ptr = std::exchange(m_ptr, p);
  delete old_ptr;
}
```

std::unique_ptr<T> Clone (3/6)


```
template <typename T>
class ToyPtr {
public:
  // ...
  T* release() noexcept;
private:
  T *m_ptr{nullptr};
};

// p.release is just like p.get, but in addition to returning a
// copy of original raw pointer, it nulls out contents of p without
// freeing original pointer, because presumably caller wants to
// take ownership of pointer
template <typename T>
T* ToyPtr<T>::release() noexcept {
  return std::exchange(m_ptr, nullptr);
}
```

std::unique_ptr<T> Clone (4/6)

- Need to implement special member functions of `unique_ptr<>` so as to preserve invariant:
  - *Once raw pointer is acquired by `unique_ptr` object, it will remain valid as long as the `unique_ptr` object has same value, and when that's no longer true – when the `unique_ptr` is adjusted to point elsewhere, or destroyed – the raw pointer will be freed correctly*

std::unique_ptr<T> Clone (5/6)


```
template <typename T>
class ToyPtr {
public:
  // ...
  ToyPtr(ToyPtr&& rhs) noexcept;
  ~ToyPtr();
  ToyPtr& operator=(ToyPtr&& rhs) noexcept;
private:
  T *m_ptr{nullptr};
};

template <typename T>
ToyPtr<T>::ToyPtr(ToyPtr&& rhs) noexcept {
  this->reset(rhs.release());
}

template <typename T>
ToyPtr<T>& ToyPtr<T>::operator=(ToyPtr&& rhs) noexcept {
  this->reset(rhs.release());
  return *this;
}
```

```
template <typename T>
ToyPtr<T>::~ToyPtr() { reset(); }
```

std::unique_ptr<T> Clone (6/6)

- Need helper function `make_toyptr` so as to
  - Never touch raw pointers with our hands
  - Make code using `ToyPtr` exception safe

```
template<typename T, typename... Args>
ToyPtr<T> make_toyptr( Args&&... args ) {
    return ToyPtr<T>(
      new T( std::forward<Args>(args)... ) );
}
```

std::unique_ptrs As Members

- By using unique_ptrs within a class, you avoid …
  - Resource leaks caused by exceptions thrown during initialization of an object [see following slides labeled Exception-Safe Function Calls]
  - Defining a destructor

Rules For Function Call Evaluation

- Function arguments may generally be evaluated in any order including being interleaved
- All functions arguments must be completely evaluated before function is called
- Execution of callee and called functions cannot be interleaved
  - Once called function begins execution, no expressions from calling function may begin or continue to be evaluated until called function's execution is completed

Exception-Safe Function Calls: Example 1 (1/2)

- Assuming `expr1` and `expr2` don't contain function calls, what can you say about following function call? `f(expr1, expr2)`
  - All we can say is that `expr1` and `expr2` must be fully evaluated before `f` is called
  - Compiler may choose to evaluate `expr1` before, after, or interleaved with evaluation of `expr2`

Exception-Safe Function Calls: Example 1 (2/2)

- Assuming `expr1` and `expr2` don't contain function calls, what can you say about following function call?

```
f(g(expr1), h(expr2))
```

- Functions and expressions may be evaluated in any order as long as following rules are respected:
  - `expr1` must be fully evaluated before `g` is called
  - `expr2` must be fully evaluated before `h` is called
  - Both `g` and `h` must complete execution before `f` is called
  - Evaluations of `expr1` and `expr2` may be interleaved with each other, but nothing may be interleaved with any of function calls

Exception-Safe Function Calls: Example 2 (1/3)

- What do you think about following function call?

```
void f(T1*, T2*); // function declaration

f(new T1, new T2); // function call
```

SIDEBAR: What Does `new` Expression Do?

- What does `new` expression do?
  1. Call `operator new` function to allocate memory
  2. Call new object's ctor to initialize object in that memory
  3. Free allocated memory if construction fails because of exception

Exception-Safe Function Calls: Example 2 (2/3)

- What do you think about following function call?

```
void f(T1*, T2*); // function declaration
f(new T1, new T2); // function call
```

1. Allocate memory for T1 object
2. Construct T1 object
3. Allocate memory for T2 object
4. Construct T2 object
5. Call f

Possible evaluation order of arguments

Memory leak occurs if either step 3 or step 4 fails due to exception. C++ standard doesn't require T1 object be destroyed and its memory deallocated.

Exception-Safe Function Calls: Example 2 (3/3)

- Another possible sequence of calls:

```
void f(T1*, T2*); // function declaration
f(new T1, new T2); // function call
```

1. Allocate memory for T1 object
2. Allocate memory for T2 object
3. Construct T1 object
4. Construct T2 object
5. Call f

If step 3 fails, C++ standard requires memory for T1 object be automatically deallocated but memory allocated for T2 object is leaked.
If step 4 fails, memory allocated for T2 object is freed but standard doesn't require fully constructed T1 object be destroyed and its memory deallocated.

Exception-Safe Function Calls: Example 3 (1/2)

- Does the following function call offer improvements?

```
// declaration of non-template function
void f(std::unique_ptr<T1>, std::unique_ptr<T2>);

// call in some source file
f(std::unique_ptr<T1>{new T1}, std::unique_ptr<T2>{new T2});
```

Each resource is safe if they're captured by their `unique_ptr`, but same problems in Example 2 occur before `unique_ptr` objects are created. Therefore, nothing has changed!!!
This is not a problem with `unique_ptr`; it's just being used the wrong way!!!

Exception-Safe Function Calls: Example 3 (2/2)

- Possible sequence of calls would be:

```
void f(std::unique_ptr<T1>, std::unique_ptr<T2>);
f(std::unique_ptr<T1>{new T1}, std::unique_ptr<T2>{new T2});
```

1. Allocate memory for `T1` object
2. Construct `T1` object
3. Allocate memory for `T2` object
4. Construct `T2` object
5. Construct `unique_ptr<T1>` object
6. Construct `unique_ptr<T2>` object
7. Call `f`

Same problems are present if either step 3 or step 4 throws.

Exception-Safe Function Calls: Solution (1/2)

- We want single function that does work of memory allocation, construction of object, and construction of `unique_ptr` object
  - Such a function will be used to build `unique_ptr` object for each argument
  - Since execution of functions cannot be interleaved, each argument of `f` will execute to completion
  - Or not if exception is thrown in which case allocated memory is returned to free store

Exception-Safe Function Calls: Solution (2/2)

- Standard library provides necessary function:

Function is template because it should work for any type

Function template is variadic because ctors of various types will have different parameters

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(
      new T(std::forward<Args>(args)...));
}
```

Because caller will want to pass ctor parameters from outside `make_unique`, perfect forwarding is necessary to pass not only values but also value categories

std::unique_ptrs As Members (1/3)

- By using unique_ptrs within a class, you avoid …
  - Resource leaks caused by exceptions thrown during initialization of an object [see preceding slides labeled Exception-Safe Function Calls]
  - Defining a destructor

std::unique_ptrs As Members (2/3)


```
// possible resource leaks when using raw pointers ...
class A {
private:
  B *pb;
  C *pc;
public:
  // might cause leak if second new throws ...
  A(int i1, int i2) : pb{new B{i1}}, pc{new C{i2}} {}
  // might cause leaks if second new throws ...
  A(A const& rhs) : pb{new B{*rhs.pb}}, pc{new C{*rhs.pc}} {}
  A const& operator=(A const& rhs) {
    *pb = *rhs.pb;
    *pc = *rhs.pc;
    return *this;
  }
  ~A() { delete pb; delete pc; }
};
```

std::unique_ptrs As Members (3/3)


```
// to avoid possible resource leaks, you can use unique_ptrs ...
class A {
private:
  std::unique_ptr<B> pb;
  std::unique_ptr<C> pc;
public:
  // no resource leak possible anymore ...
  A(int i1, int i2) : pb{std::make_unique<B>(i1)},
                      pc{ std::make_unique<C>(i2)} {}
  A(A const& rhs) : pb{std::make_unique<B>(*rhs.pb)},
         pc{std::make_unique<C>(*rhs.pc)} { return *this; }
  A const& operator=(A const& rhs) {
    pb.reset(std::make_unique<B>(*rhs.pb));
    pc.reset(std::make_unique<C>(*rhs.pc));
    return *this;
  }
  // default dtor lets pb and pc delete their objects]
};
```

Containers of `std::unique_ptr`s

- Should a large number of non-trivial objects [of same type] be stored in a container [such as `std::vector`] by value, or by pointer, or by `unique_ptr`s?
- See *uptr-cont.cpp* for an answer …

Deletion Callback (1/2)


```
template <typename T, typename... Types>
unique_ptr<T> make_unique(Types&&... params) {
  return unique_ptr<T>(new T(std::forward<Types>(params)...));
}
```

make_unique uses new operator to allocate and initialize memory

```
unique_ptr<double> ud {make_unique<double>(1.9)};
```

dtor of class unique_ptr will use delete operator to return memory pointed to by raw pointer [encapsulated by ud] back to free store

Deletion Callback (2/2)

- In some cases, memory provided to `unique_ptr` cannot be released using `delete`
- `std::unique_ptr<T,D>` has 2nd template type parameter: a *deletion callback type*
  - Parameter `D` defaults to `std::default_delete<T>` which uses `delete` to deallocate memory
- See *fred-deleter.cpp* and *file-deleter.cpp*

std::unique_ptr<T[]> Specialization for Arrays (1/2)


```
// value initializes 3 ints to 0
// i.e., new T[3]{}
std::unique_ptr<int[]>
upi{std::make_unique<int[]>(3)};

// partial specialization doesn't overload
// operators * and ->
// operator[] is provided to access
// one of the elements inside the array

upi[0] = 11; upi[1] = 12; upi[2] = 13;
```

`std::unique_ptr<T[]>` Specialization for Arrays (2/2)

- Better to use `vector<>` container because it is more flexible and powerful than smart pointer

Conclusion (1/4)

- What can you say about semantics by looking at following function signatures?

```
void foo(std::unique_ptr<Widget> p);

std::unique_ptr<Widget> boo();

void coo(Widget *p);
```

Conclusion (2/4)

- `foo` is a *consumer* of widgets
- When we call `foo`, we must have unique ownership of a `Widget` that was allocated with `new`, and which is safe to `delete`!

```
void foo(std::unique_ptr<Widget>);
```

Conclusion (3/4)

- `boo` is a *producer* of widgets
- When we call `boo`, we get unique ownership of a `Widget` that was allocated with `new`, and which is safe to `delete`!

```
std::unique_ptr<Widget> boo();
```

Conclusion (4/4)

- `coo` expresses *ambiguity*
- `unique_ptr<T>` is a *vocabulary type* for expressing ownership transfer, whereas `T*` is C++'s equivalent of nonsense word that no two people will necessarily agree on what it means

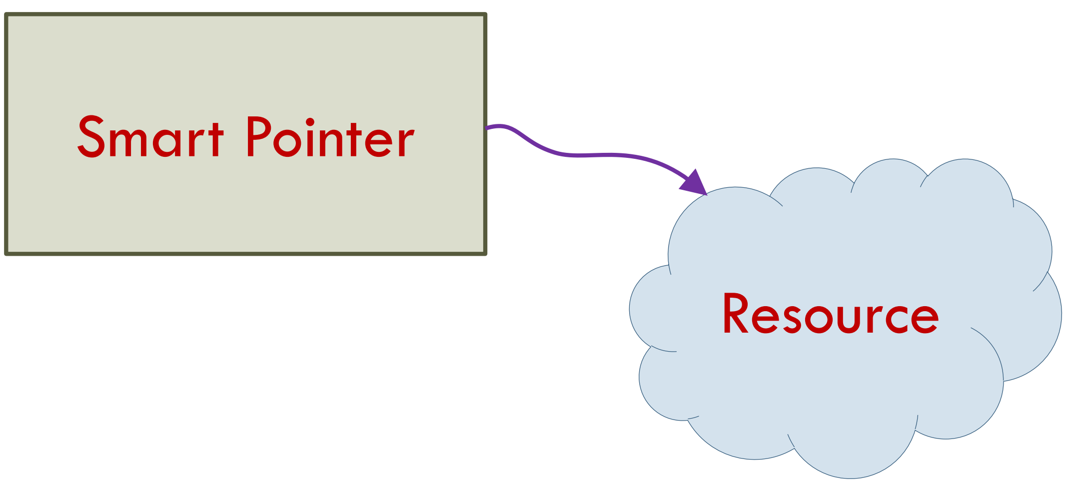
```
void coo(Widget *p);
```

std::shared_ptr<T>

- Unique pointers embody *exclusive ownership* semantics
- Shared pointers embody *unclear ownership* of resource using technique called *reference counting*

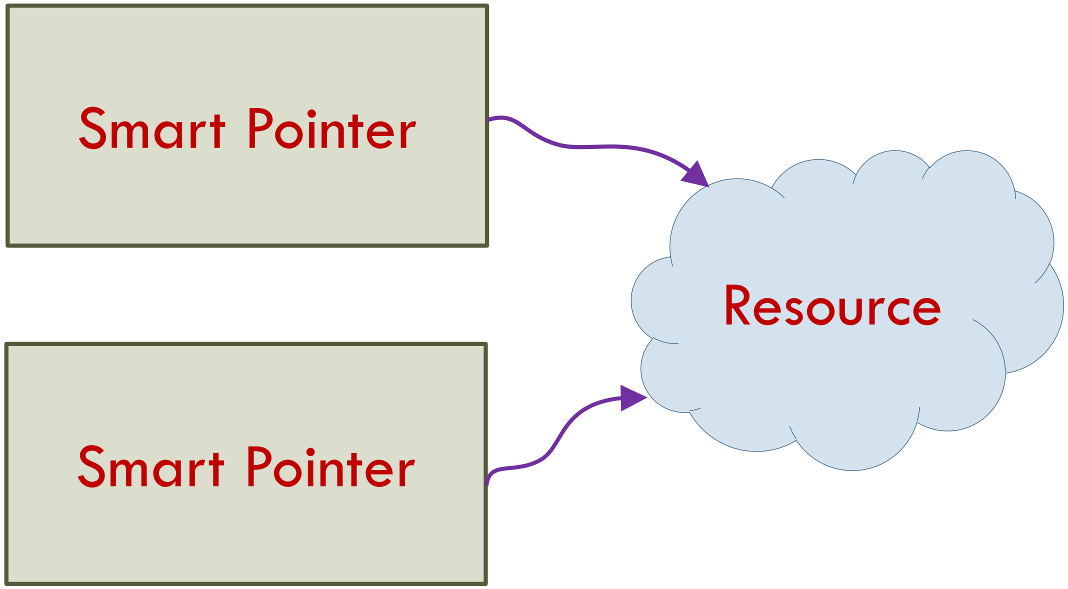
Reference Counting: Idea (1/6)

- Consider smart pointer class that stores pointer to resource
- Dtor can then delete resource automatically, so clients of smart pointer never need to explicitly clean up any resources



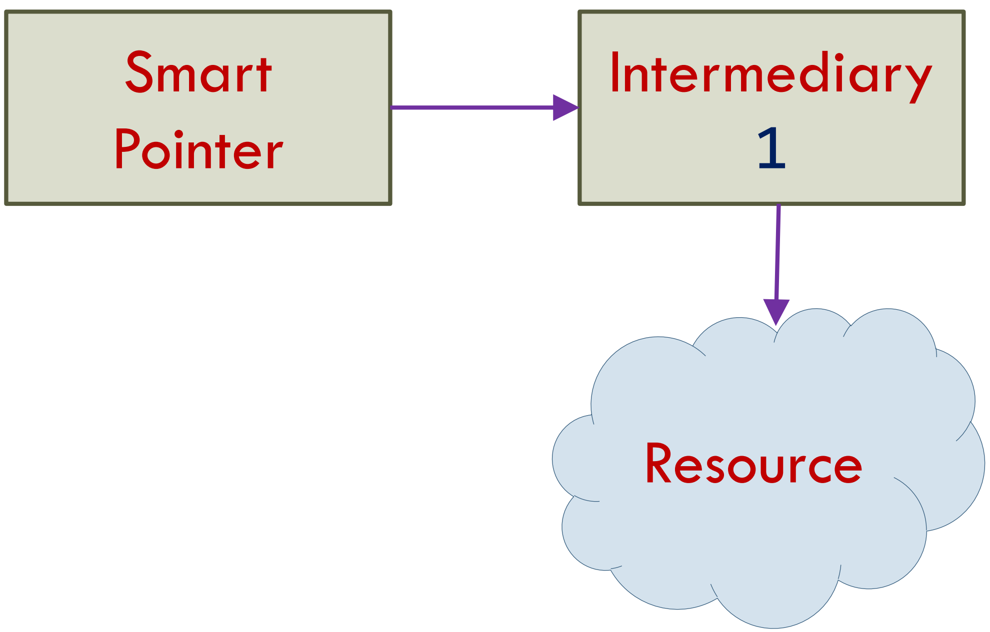
Reference Counting: Idea (2/6)

- We hit a snag when several smart pointers point to same resource



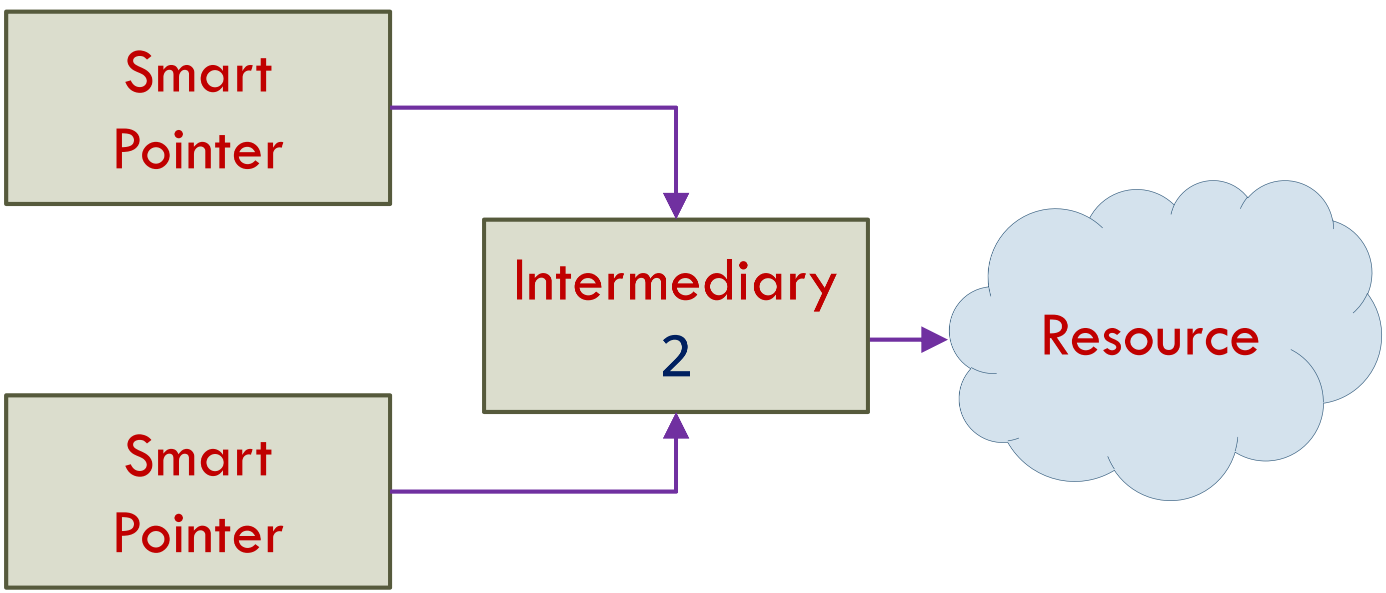
Reference Counting: Idea (3/6)

- Reference counting keeps track of number of pointers to dynamically-allocated resource



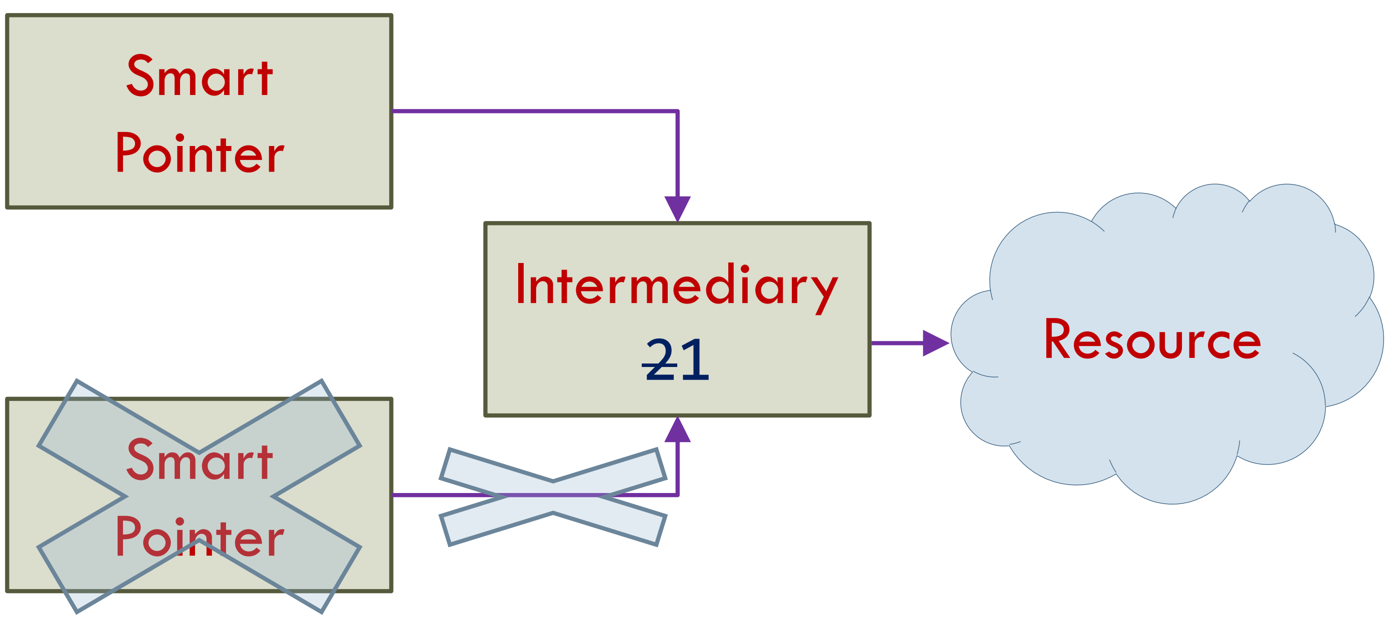
Reference Counting: Idea (4/6)

- Suppose we want to share the resource with another smart pointer



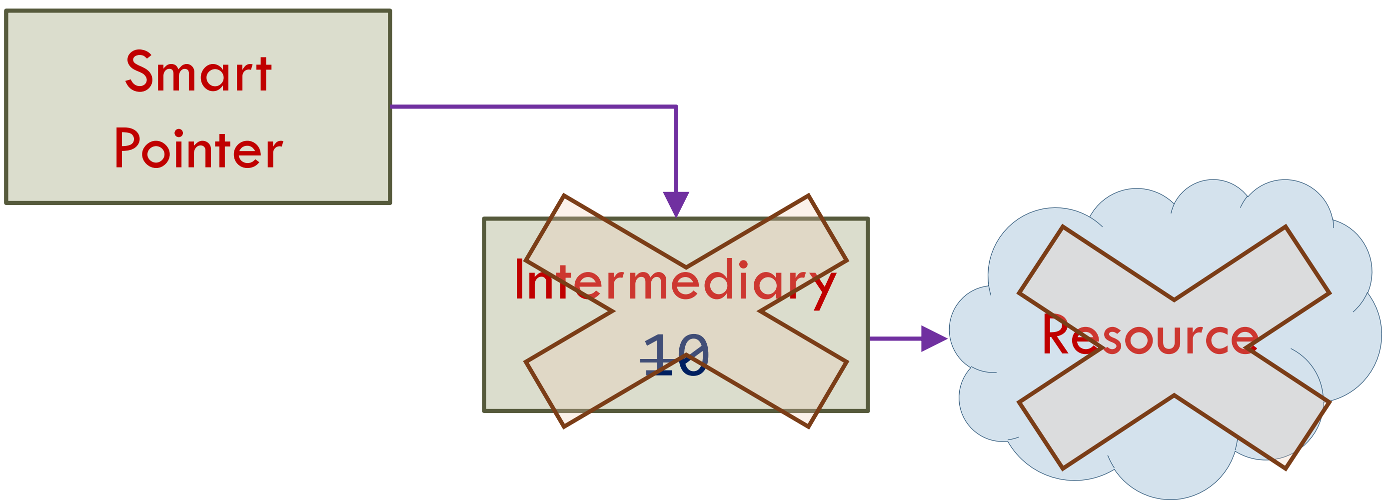
Reference Counting: Idea (5/6)

- Now, suppose one smart pointer needs to stop pointing to the resource:



Reference Counting: Idea (6/6)

- Finally, suppose last smart pointer needs to stop pointing to the resource:



Reference Counting: Summary (1/2)

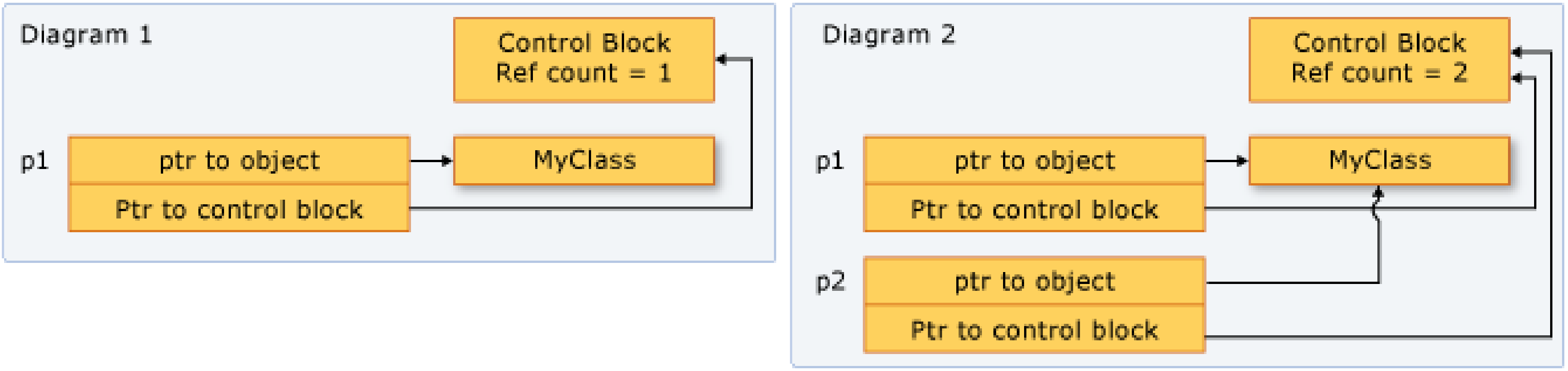
- When creating a smart pointer to manage newly allocated memory, 1st create intermediary object and make the intermediary object point to resource; then attach smart pointer to intermediary and set reference count to one
- To make new smart pointer point to same resource as existing one, make new smart pointer point to old smart pointer’s intermediary object; then increment intermediary’s reference count
- To remove smart pointer from resource, decrement intermediary object’s reference count; if count reaches zero, deallocate resource and intermediary object

Reference Counting: Summary (2/2)


```cpp
template <typename T> class smart_ptr {
public:
  explicit smart_ptr(T *memory);
  smart_ptr(smart_ptr const&);
  smart_ptr(smart_ptr &&);
  smart_ptr& operator=(smart_ptr const&);
  smart_ptr& operator=(smart_ptr &&);
  ~smart_ptr();

  T& operator* () const;
  T* operator->() const;

  T* get() const;
  size_t get_ref_count() const;
  void reset(T *new_resource);
```

```cpp
private:
  struct Intermediary {
    T* resource;
    size_t ref_cnt;
  };
  Intermediary *data;

  void detach();
  void attach(Intermediary *other);
};
```

Standard Library: Possible Implementation



Picture from here

`std::shared_ptr<>` Methods

Name	Description
`get`	Returns pointer to resource
`get_deleter`	Returns `delete` function
`reset`	Resets resource
`swap`	Swaps resource
`unique`	Checks if `std::shared_ptr` is exclusive owner of resource
`use_count`	Returns value of reference counter

Using std::shared_ptr<T> (1/3)


```
std::shared_ptr<std::string> sp1{new std::string{"a"}};

std::shared_ptr<std::string> sp2; // uninitialized
sp2 = new std::string{"b"}; // error
sp2.reset(new std::string{"c"}); // ok

// faster, safer to use convenience function std::make_shared
std::shared_ptr<std::string> sp3
                       {std::make_shared< std::string>("d")};
```

Using std::shared_ptr<T> (2/3)


```
using sps = std::shared_ptr<std::string>;
sps sp1{std::make_shared<std::string>("john")};
sps sp2{std::make_shared<std::string>("mary")};

(*sp1)[0] = 'J';          // use like ordinary pointer ...
sp2->replace(0, 1, "M"); // use like ordinary pointer ...

std::vector<sps> names;
names.push_back(sp1); names.push_back(sp1); names.push_back(sp2);
names.push_back(sp1); names.push_back(sp2);

for (sps const& x : names) { // print all elements ...
  std::cout << *x << '\n';
}

*sp1 = "Johnson"; // overwrite name ...
sp2.reset(new std::string("Johnston")); // replace resource ...

for (sps const& x : names) {std::cout << *x << '\n'; }
std::cout << sp1.use_count() << " | " << sp2.use_count() << '\n';
```

Using `std::shared_ptr<T>` (3/3)

- See *using_shared.cpp* and *sharedptr-cont.cpp*

Misusing std::shared_ptr<T> (1/2)

- You've to ensure only one group of shared pointers owns an object

```
// ERROR: two shared pointers manage allocated int
int *pi {new int {10}};
std::shared_ptr<int> spi1(pi);
std::shared_ptr<int> spi2(pi);
```

```
// directly initialize smart pointer the moment
// you create object with associated resource
std::shared_ptr<int> spi1(std::make_shared<int>(10));
std::shared_ptr<int> spi2{spi1}; // ok
```

Misusing std::shared_ptr<T> (2/2)

- Don't double-manage!!!

```
// never touch raw pointers with your hands!!!
// using word shared_ptr explicitly in your code!!!
std::shared_ptr<int> pa, pb, pc;
pa = std::make_shared<int>(11); // 1
pb = pa; // 2
pc = std::shared_ptr<int>(pb.get()); // WRONG!!!
// give the same pointer to shared_ptr again
// which tells shared_ptr to manage it twice
assert(pb.use_count() == 2);
assert(pc.use_count() == 1);
pc = nullptr;
// pc's use count drops to zero and shared_ptr
// calls delete on the int object
*pb = 12; // WRONG!!!
// accessing freed object yields undefined behavior
```

std::weak_ptr<T> (1/4)

- Consider [rare] situation where we're using `shared_ptr` to manage ownership of shared object, and we'd like to keep pointer to an object without explicitly expressing ownership of that object

std::weak_ptr<T> (2/4)

- We could use raw pointer to express idea of "non-owning reference"

```
struct DangerousWatcher {
  int *m_ptr{nullptr};

  void watch(std::shared_ptr<int> const& sp) {
    m_ptr = sp.get();
  }

  int current_value() const {
    // possible that object m_ptr is pointing to
    // has been deallocated!!!
    return *m_ptr;
  }
};
```

std::weak_ptr<T> (3/4)

- We could use shared_ptr to express idea of "reference"

```
// more participant than watcher ...
struct NotReallyWatcher {
  std::shared_ptr<int> m_ptr;

  void watch(std::shared_ptr<int> const& sp) {
    m_ptr = sp;
  }

  int current_value() const {
    // now object pointed to by m_ptr cannot ever
    // be deallocated - mere existence of m_ptr
    // is keeping that object alive!!!
    return *m_ptr;
  }
};
```

std::weak_ptr<T> (4/4)

- What we really want is non-owing reference that is nevertheless aware of the `shared_ptr` system for managing memory and is able to query whether referenced object still exists

```
struct CorrectWatcher {
  std::weak_ptr<int> m_ptr;

  void watch(std::shared_ptr<int> const& sp) {
    m_ptr = std::weak_ptr<int>(sp);
  }

  int current_value() const {
    // safely ask whether m_ptr has been deallocated or not
    if (std::shared_ptr<int> p = m_ptr.lock()) {
      return *p;
    } else {
      throw "it has no value - its' been deallocated!!!";
    }
  }
};
```

What is RAII Idiom?

- Resource Acquisition Is Allocation
- Resource acquisition and release are bound to lifetime of an object
- Resource is allocated in ctor and deallocated in dtor
- Works because dtor is called when stack-based object goes out of scope

RAII Classes: Rule of Three

- If your class manages a resource, you'll ***need to write*** three special member functions:
  - Destructor to release the resource
  - Copy constructor to clone the resource
  - Copy assignment operator to release current resource and acquire cloned resource
- Caveat: You'll need to define swap function to implement copy assignment operator using copy-swap idiom

Rule of Five (1/3)

- C++11 introduced move operations, transforming ROT into ROF
  - ROF because move operations were implicitly generated under certain circumstances
- Lots of rules for implicit move operations but generalized like this:
  - You get default move ctor or move assignment operator if and only if none of other four are defined/defaulted by class
  - Compiler will enforce this rule

Rule of Five (2/3)


```
class P {
public:
  P(int x) : i{x} {}
  ~P() {}
  P(P&& rhs) : i{rhs.i} {}
  int I() const { return i; }
  void I(int x) { i = x; }
private:
  int i;
};

int main() {
  P a1{10}, a2{20};
  a2 = a1; // compiler error!!!
}
```

Rule of Five (3/3)

	default ctor	dtor	copy ctor	copy assignment	move ctor	move assignment
none defined	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any ctor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default ctor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
dtor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy ctor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move ctor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Rule of Zero

- Rule of Five transitions into Rule of Zero

Write your classes in a way so that you don't need to declare/define neither destructor, nor copy/move constructor, nor copy/move assignment operator

Use smart pointers & standard library classes for managing resources

Exceptions to ROZ Guideline

- Two cases where users generally bypass compiler and write their own declarations:
  - Managing resources
  - Polymorphic deletion and/or virtual functions